Table of Contents

ArangoDB v3.1.3 Documentation

Welcome to the ArangoDB documentation!

New and eager to try out ArangoDB? Start right away with our beginner's guide: Getting Started

The documentation is organized in four handbooks:

- This manual describes ArangoDB and its features in detail for you as a user, developer and administrator.
- The AQL handbook explains ArangoDB's query language AQL.
- The HTTP handbook describes the internal API of ArangoDB that is used to communicate with clients. In general, the HTTP handbook will be of interest to driver developers. If you use any of the existing drivers for the language of your choice, you can skip this handbook.
- Our cookbook with recipes for specific problems and solutions.

Features are illustrated with interactive usage examples; you can cut'n'paste them into arangosh to try them out. The HTTP REST-API for driver developers is demonstrated with cut'n'paste recepies intended to be used with the cURL. Drivers may provide their own examples based on these .js based examples to improve understandeability for their respective users, i.e. for the java driver some of the samples are re-implemented.

Overview

ArangoDB is a multi-model, open-source database with flexible data models for documents, graphs, and key-values. Build high performance applications using a convenient SQL-like query language or JavaScript extensions. Use ACID transactions if you require them. Scale horizontally and vertically with a few mouse clicks.

Key features include:

- installing ArangoDB on a **cluster** is as easy as installing an app on your mobile
- **Flexible data modeling**: model your data as combination of key-value pairs, documents or graphs - perfect for social relations
- **Powerful query language** (AQL) to retrieve and modify data
- Use ArangoDB as an **application server** and fuse your application and database together for maximal throughput
- **Transactions**: run queries on multiple documents or collections with optional transactional consistency and isolation
- **Replication** and **Sharding**: set up the database in a master-slave configuration or spread bigger datasets across multiple servers
- Configurable **durability**: let the application decide if it needs more durability or more performance
- No-nonsense storage: ArangoDB uses all of the power of **modern storage hardware**, like SSD and large caches
- JavaScript for all: **no language zoo**, you can use one language from your browser to your back-end
- ArangoDB can be easily deployed as a **fault-tolerant distributed state machine**, which can serve as the animal brain of distributed appliances
- It is **open source** (Apache License 2.0)

Community

If you have questions regarding ArangoDB, Foxx, drivers, or this documentation don't hesitate to contact us on:

- GitHub for issues and misbehavior or pull requests
- Google Groups for discussions about ArangoDB in general or to announce your new Foxx App
- StackOverflow for questions about AQL, usage scenarios etc.
- Slack, our community chat

When reporting issues, please describe:

- the environment you run ArangoDB in
- the ArangoDB version you use
- whether you're using Foxx
- the client you're using
- which parts of the documentation you're working with (link)
- what you expect to happen
- what is actually happening

We will respond as soon as possible.

Getting started

Overview

This beginner's guide will make you familiar with ArangoDB. We will cover how to

- install and run a local ArangoDB server
- use the web interface to interact with it
- store example data in the database
- query the database to retrieve the data again
- edit and remove existing data

Installation

Head to arangodb.com/download, select your operating system and download ArangoDB. You may also follow the instructions on how to install with a package manager, if available.

If you installed a binary package under Linux, the server is automatically started.

If you installed ArangoDB using homebrew under MacOS X, start the server by running `/usr/local/sbin/arangod`.

If you installed ArangoDB under Windows as a service, the server is automatically started. Otherwise, run the `arangod.exe` located in the installation folder's `bin` directory. You may have to run it as administrator to grant it write permissions to `C:\Program Files`.

For more in-depth information on how to install ArangoDB, as well as available startup parameters, installation in a cluster and so on, see Installing.

Securing the installation

The default installation contains one database *_system* and a user named *root*.

Debian based packages and the Windows installer will ask for a password during the installation process. Red-Hat based packages will set a random password. For all other installation packages you need to execute

```
shell> arango-secure-installation
```

This will asked for a root password and sets this password.

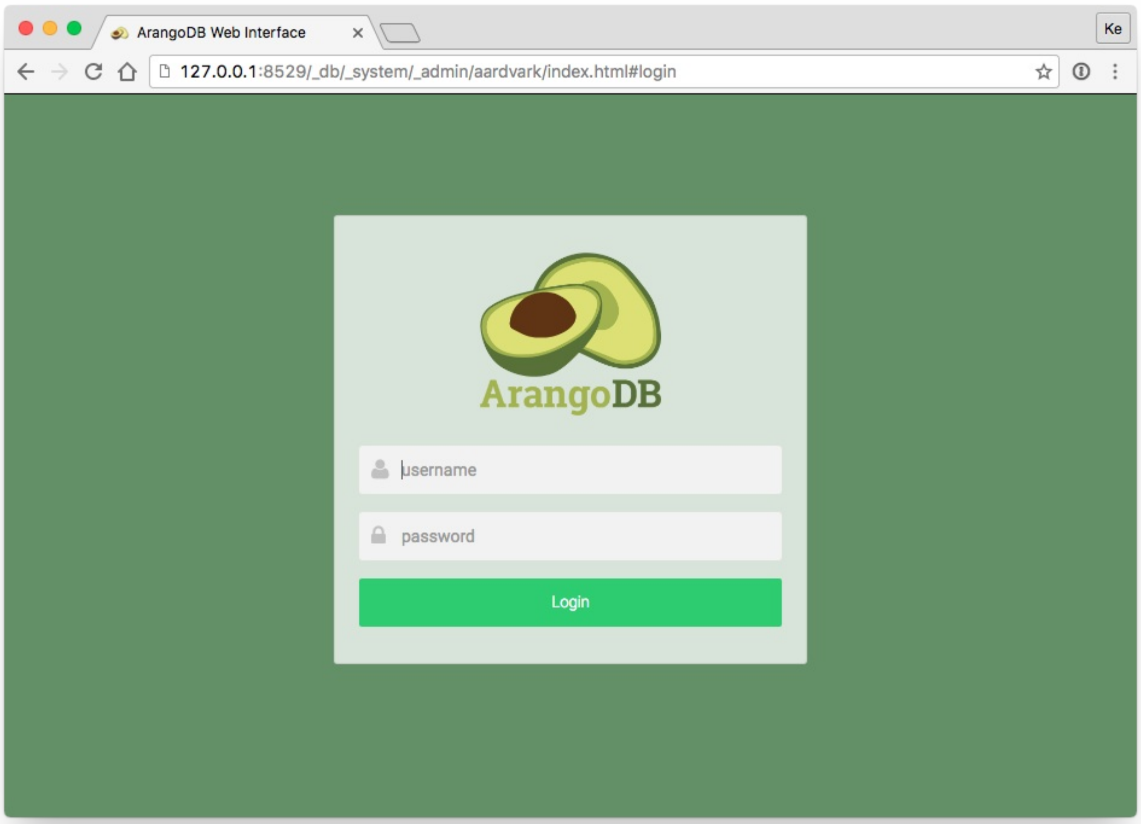
Web interface

The server itself (*arangod*) speaks HTTP / REST, but you can use the graphical web interface to keep it simple. There's also arangosh, a synchronous shell for interaction with the server. If you're a developer, you might prefer the shell over the GUI. It does not provide features like syntax highlighting however.

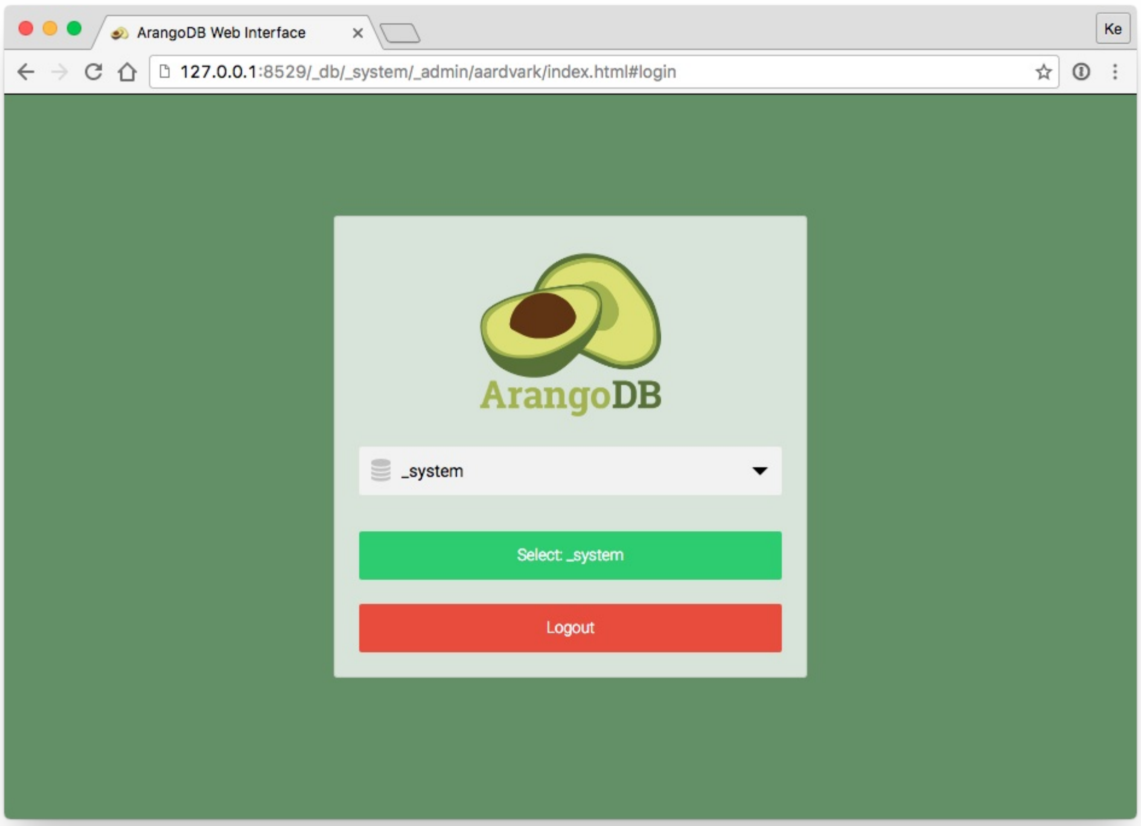
When you start using ArangoDB in your project, you will likely use an official or community-made driver written in the same language as your project. Drivers implement a programming interface that should feel natural for that programming language, and do all the talking to the server. Therefore, you can most certainly ignore the HTTP API unless you want to write a driver yourself or explicitly want to use the raw interface.

To get familiar with the database system you can even put drivers aside and use the web interface (code name *Aardvark*) for basic interaction. The web interface will become available shortly after you started `arangod`. You can access it in your browser at http://localhost:8529 - if not, please see Troubleshooting.

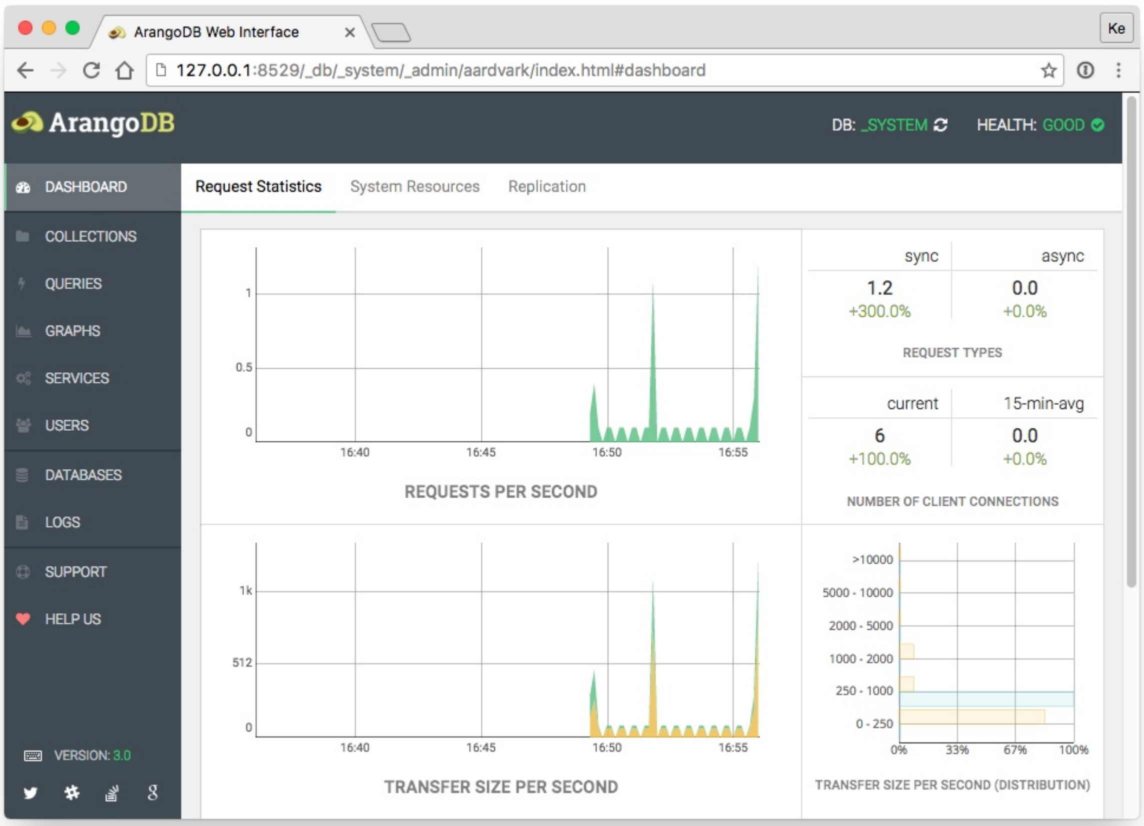
By default, authentication is enabled. The default user is `root`. Depending on the installation method used, the installation process either prompted for the root password or the default root password is empty (see above).



Next you will be asked which database to use. Every server instance comes with a `_system` database. Select this database to continue.



You should then be presented the dashboard with server statistics like this:



For a more detailed description of the interface, see Web Interface.

Databases, collections and documents

Databases are sets of collections. Collections store records, which are referred to as documents. Collections are the equivalent of tables in RDBMS, and documents can be thought of as rows in a table. The difference is that you don't define what columns (or rather attributes) there will be in advance. Every document in any collection can have arbitrary attribute keys and values. Documents in a single collection will likely have a similar structure in practice however, but the database system itself does not impose it and will operate stable and fast no matter how your data looks like.

Read more in the data-model concepts chapter.

For now, you can stick with the default `_system` database and use the web interface to create collections and documents. Start by clicking the *COLLECTIONS* menu entry, then the *Add Collection* tile. Give it a name, e.g. *users*, leave the other settings unchanged (we want it to be a document collection) and *Save* it. A new tile labeled *users* should show up, which you can click to open.

There will be *No documents* yet. Click the green circle with the white plus on the right-hand side to create a first document in this collection. A dialog will ask you for a `_key`. You can leave the field blank and click *Create* to let the database system assign an automatically generated (unique) key. Note that the `_key` property is immutable, which means you can not change it once the document is created. What you can use as document key is described in the naming conventions.

An automatically generated key could be `"9883"` (`_key` is always a string!), and the document `_id` would be `"users/9883"` in that case. Aside from a few system attributes, there is nothing in this document yet. Let's add a custom attribute by clicking the icon to the left of *(empty object)*, then *Append*. Two input fields will become available, *FIELD* (attribute key) and *VALUE* (attribute value). Type `name` as key and your name as value. *Append* another attribute, name it `age` and set it to your age. Click *Save* to persist the changes. If you click on *Collection: users* at the top on the right-hand side of the ArangoDB logo, the document browser will show the documents in the *users* collection and you will see the document you just created in the list.

Querying the database

Time to retrieve our document using AQL, ArangoDB's query language. We can directly look up the document we created via the `_id` , but there are also other options. Click the *QUERIES* menu entry to bring up the query editor and type the following (adjust the document ID to match your document):

```
RETURN DOCUMENT("users/9883")
```

Then click *Execute* to run the query. The result appears below the query editor:

```
[
  {
    "_key": "9883",
    "_id": "users/9883",
    "_rev": "9883",
    "age": 32,
    "name": "John Smith"
  }
]
```

As you can see, the entire document including the system attributes is returned. DOCUMENT() is a function to retrieve a single document or a list of documents of which you know the `_key` s or `_id` s. We return the result of the function call as our query result, which is our document inside of the result array (we could have returned more than one result with a different query, but even for a single document as result, we still get an array at the top level).

This type of query is called data access query. No data is created, changed or deleted. There is another type of query called data modification query. Let's insert a second document using a modification query:

```
INSERT { name: "Katie Foster", age: 27 } INTO users
```

The query is pretty self-explanatory: the `INSERT` keyword tells ArangoDB that we want to insert something. What to insert, a document with two attributes in this case, follows next. The curly braces `{ }` signify documents, or objects. When talking about records in a collection, we call them documents. Encoded as JSON, we call them objects. Objects can also be nested. Here's an example:

```
{
  "name": {
    "first": "Katie",
    "last": "Foster"
  }
}
```

`INTO` is a mandatory part of every `INSERT` operation and is followed by the collection name that we want to store the document in. Note that there are no quote marks around the collection name.

If you run above query, there will be an empty array as result because we did not specify what to return using a `RETURN` keyword. It is optional in modification queries, but mandatory in data access queries. Even with `RESULT` , the return value can still be an empty array, e.g. if the specified document was not found. Despite the empty result, the above query still created a new user document. You can verify this with the document browser.

Let's add another user, but return the newly created document this time:

```
INSERT { name: "James Hendrix", age: 69 } INTO users
RETURN NEW
```

`NEW` is a pseudo-variable, which refers to the document created by `INSERT` . The result of the query will look like this:

```
[
  {
    "_key": "10074",
    "_id": "users/10074",
    "_rev": "10074",
    "age": 69,
    "name": "James Hendrix"
  }
]
```

Now that we have 3 users in our collection, how to retrieve them all with a single query? The following **does not work**:

```
RETURN DOCUMENT("users/9883")
RETURN DOCUMENT("users/9915")
RETURN DOCUMENT("users/10074")
```

There can only be a single `RETURN` statement here and a syntax error is raised if you try to execute it. The `DOCUMENT()` function offers a secondary signature to specify multiple document handles, so we could do:

```
RETURN DOCUMENT( ["users/9883", "users/9915", "users/10074"] )
```

An array with the `_id` s of all 3 documents is passed to the function. Arrays are denoted by square brackets `[ ]` and their elements are separated by commas.

But what if we add more users? We would have to change the query to retrieve the newly added users as well. All we want to say with our query is: "For every user in the collection users, return the user document". We can formulate this with a `FOR` loop:

```
FOR user IN users
  RETURN user
```

It expresses to iterate over every document in `users` and to use `user` as variable name, which we can use to refer to the current user document. It could also be called `doc` , `u` or `ahuacatlguacamole` , this is up to you. It is advisable to use a short and self-descriptive name however.

The loop body tells the system to return the value of the variable `user` , which is a single user document. All user documents are returned this way:

```
[
  {
    "_key": "9915",
    "_id": "users/9915",
    "_rev": "9915",
    "age": 27,
    "name": "Katie Foster"
  },
  {
    "_key": "9883",
    "_id": "users/9883",
    "_rev": "9883",
    "age": 32,
    "name": "John Smith"
  },
  {
    "_key": "10074",
    "_id": "users/10074",
    "_rev": "10074",
    "age": 69,
    "name": "James Hendrix"
  }
]
```

You may have noticed that the order of the returned documents is not necessarily the same as they were inserted. There is no order guaranteed unless you explicitly sort them. We can add a `SORT` operation very easily:

```
FOR user IN users
  SORT user._key
  RETURN user
```

This does still not return the desired result: James (10074) is returned before John (9883) and Katie (9915). The reason is that the `_key` attribute is a string in ArangoDB, and not a number. The individual characters of the strings are compared. `1` is lower than `9` and the result is therefore "correct". If we wanted to use the numerical value of the `_key` attributes instead, we could convert the string to a number and use it for sorting. There are some implications however. We are better off sorting something else. How about the age, in descending order?

```
FOR user IN users
  SORT user.age DESC
  RETURN user
```

The users will be returned in the following order: James (69), John (32), Katie (27). Instead of `DESC` for descending order, `ASC` can be used for ascending order. `ASC` is the default though and can be omitted.

We might want to limit the result set to a subset of users, based on the age attribute for example. Let's return users older than 30 only:

```
FOR user IN users
  FILTER user.age > 30
  SORT user.age
  RETURN user
```

This will return John and James (in this order). Katie's age attribute does not fulfill the criterion (greater than 30), she is only 27 and therefore not part of the result set. We can make her age to return her user document again, using a modification query:

```
UPDATE "9915" WITH { age: 40 } IN users
RETURN NEW
```

`UPDATE` allows to partially edit an existing document. There is also `REPLACE`, which would remove all attributes (except for `_key` and `_id`, which remain the same) and only add the specified ones. `UPDATE` on the other hand only replaces the specified attributes and keeps everything else as-is.

The `UPDATE` keyword is followed by the document key (or a document / object with a `_key` attribute) to identify what to modify. The attributes to update are written as object after the `WITH` keyword. `IN` denotes in which collection to perform this operation in, just like `INTO` (both keywords are actually interchangable here). The full document with the changes applied is returned if we use the `NEW` pseudo-variable:

```
[
  {
    "_key": "9915",
    "_id": "users/9915",
    "_rev": "12864",
    "age": 40,
    "name": "Katie Foster"
  }
```

If we used `REPLACE` instead, the name attribute would be gone. With `UPDATE`, the attribute is kept (the same would apply to additional attributes if we had them).

Let us run our `FILTER` query again, but only return the user names this time:

```
FOR user IN users
  FILTER user.age > 30
  SORT user.age
  RETURN user.name
```

This will return the names of all 3 users:

```
[
  "John Smith",
  "Katie Foster",
  "James Hendrix"
]
```

It is called a projection if only a subset of attributes is returned. Another kind of projection is to change the structure of the results:

```
FOR user IN users
  RETURN { userName: user.name, age: user.age }
```

The query defines the output format for every user document. The user name is returned as `userName` instead of `name`, the age keeps the attribute key in this example:

```
[
  {
    "userName": "James Hendrix",
    "age": 69
  },
  {
    "userName": "John Smith",
    "age": 32
  },
  {
    "userName": "Katie Foster",
    "age": 40
  }
]
```

It is also possible to compute new values:

```
FOR user IN users
  RETURN CONCAT(user.name, "'s age is ", user.age)
```

`CONCAT()` is a function that can join elements together to a string. We use it here to return a statement for every user. As you can see, the result set does not always have to be an array of objects:

```
[
  "James Hendrix's age is 69",
  "John Smith's age is 32",
  "Katie Foster's age is 40"
]
```

Now let's do something crazy: for every document in the users collection, iterate over all user documents again and return user pairs, e.g. John and Katie. We can use a loop inside a loop for this to get the cross product (every possible combination of all user records, 3 *3 = 9). We don't want pairings like* John + John* however, so let's eliminate them with a filter condition:

```
FOR user1 IN users
  FOR user2 IN users
    FILTER user1 != user2
    RETURN [user1.name, user2.name]
```

We get 6 pairings. Pairs like *James + John* and *John + James* are basically redundant, but fair enough:

```
[
  [ "James Hendrix", "John Smith" ],
  [ "James Hendrix", "Katie Foster" ],
  [ "John Smith", "James Hendrix" ],
  [ "John Smith", "Katie Foster" ],
  [ "Katie Foster", "James Hendrix" ],
  [ "Katie Foster", "John Smith" ]
]
```

We could calculate the sum of both ages and compute something new this way:

```
FOR user1 IN users
  FOR user2 IN users
    FILTER user1 != user2
    RETURN {
        pair: [user1.name, user2.name],
        sumOfAges: user1.age + user2.age
    }
```

We introduce a new attribute `sumOfAges` and add up both ages for the value:

```
[
  {
    "pair": [ "James Hendrix", "John Smith" ],
    "sumOfAges": 101
  },
  {
    "pair": [ "James Hendrix", "Katie Foster" ],
    "sumOfAges": 109
  },
  {
    "pair": [ "John Smith", "James Hendrix" ],
    "sumOfAges": 101
  },
  {
    "pair": [ "John Smith", "Katie Foster" ],
    "sumOfAges": 72
  },
  {
    "pair": [ "Katie Foster", "James Hendrix" ],
    "sumOfAges": 109
  },
  {
    "pair": [ "Katie Foster", "John Smith" ],
    "sumOfAges": 72
  }
]
```

If we wanted to post-filter on the new attribute to only return pairs with a sum less than 100, we should define a variable to temporarily store the sum, so that we can use it in a `FILTER` statement as well as in the `RETURN` statement:

```
FOR user1 IN users
  FOR user2 IN users
    FILTER user1 != user2
    LET sumOfAges = user1.age + user2.age
    FILTER sumOfAges < 100
    RETURN {
        pair: [user1.name, user2.name],
        sumOfAges: sumOfAges
    }
```

The `LET` keyword is followed by the designated variable name (`sumOfAges`), then there's a `=` symbol and the value or an expression to define what value the variable is supposed to have. We re-use our expression to calculate the sum here. We then have another `FILTER` to skip the unwanted pairings and make use of the variable we declared before. We return a projection with an array of the user names and the calculated age, for which we use the variable again:

```
[
  {
    "pair": [ "John Smith", "Katie Foster" ],
    "sumOfAges": 72
  },
  {
    "pair": [ "Katie Foster", "John Smith" ],
    "sumOfAges": 72
  }
]
```

Pro tip: when defining objects, if the desired attribute key and the variable to use for the attribute value are the same, you can use a shorthand notation: `{ sumOfAges }` instead of `{ sumOfAges: sumOfAges }` .

Finally, let's delete one of the user documents:

```
REMOVE "9883" IN users
```

It deletes the user John (`_key: "9883"`). We could also remove documents in a loop (same goes for `INSERT` , `UPDATE` and `REPLACE`):

```
FOR user IN users
    FILTER user.age >= 30
    REMOVE user IN users
```

The query deletes all users whose age is greater than or equal to 30.

How to continue

There is a lot more to discover in AQL and much more functionality that ArangoDB offers. Continue reading the other chapters and experiment with a test database to foster your knowledge.

If you want to write more AQL queries right now, have a look here:

- Data Queries: data access and modification queries
- High-level operations: detailed descriptions of `FOR` , `FILTER` and more operations not shown in this introduction
- Functions: a reference of all provided functions

ArangoDB programs

The ArangoDB package comes with the following programs:

- `arangod` : The ArangoDB database daemon. This server program is intended to run as a daemon process and to serve the various clients connection to the server via TCP / HTTP.
- `arangosh` : The ArangoDB shell. A client that implements a read-eval-print loop (REPL) and provides functions to access and administrate the ArangoDB server.
- `arangoimp` : A bulk importer for the ArangoDB server. It supports JSON and CSV.
- `arangodump` : A tool to create backups of an ArangoDB database in JSON format.
- `arangorestore` : A tool to load data of a backup back into an ArangoDB database.
- `arango-dfdb` : A datafile debugger for ArangoDB. It is primarily intended to be used during development of ArangoDB.
- `arangobench` : A benchmark and test tool. It can be used for performance and server function testing.

Installing

First of all, download and install the corresponding RPM or Debian package or use homebrew on MacOS X. You can find packages for various operation systems at our install section, including installers for Windows.

How to do that in detail is described in the subchapters of this section.

On how to set up a cluster, checkout the Deployment chapter.

Linux

- Visit the official ArangoDB install page and download the correct package for your Linux distribution. You can find binary packages for the most common distributions there.
- Follow the instructions to use your favorite package manager for the major distributions. After setting up the ArangoDB repository you can easily install ArangoDB using yum, aptitude, urpmi or zypper.
- Debian based packages will ask for a password during installation. For an unattended installation for Debian, see below. Red-Hat based packages will set a random password during installation. For other distributions or to change the password, run `arango-secure-installation` to set a root password.
- Alternatively, see Compiling if you want to build ArangoDB yourself.
- Start up the database server.

Normally, this is done by executing the following command:

```
unix> /etc/init.d/arangod start
```

It will start the server, and do that as well at system boot time.

To stop the server you can use the following command:

```
unix> /etc/init.d/arangod stop
```

The exact commands depend on your Linux distribution. You may require root privileges to execute these commands.

Linux Mint

Please use the corresponding Ubuntu or Debian packages.

Unattended Installation

Debian based package will ask for a password during installation. For unattended installation, you can set the password using the debconf helpers.

```
echo arangodb3 arangodb3/password password NEWPASSWORD | debconf-set-selections
echo arangodb3 arangodb3/password_again password NEWPASSWORD | debconf-set-selections
```

The commands should be executed prior to the installation.

Red-Hat based packages will set a random password during installation. If you want to force a password, execute

```
ARANGODB_DEFAULT_ROOT_PASSWORD=NEWPASSWORD arango-secure-installation
```

The command should be executed after the installation.

Non-Standard Installation

If you compiled ArangoDB from source and did not use any installation package – or using non-default locations and/or multiple ArangoDB instances on the same host – you may want to start the server process manually. You can do so by invoking the arangod binary from the command line as shown below:

```
unix> /usr/local/sbin/arangod /tmp/vocbase
20ZZ-XX-YYT12:37:08Z [8145] INFO using built-in JavaScript startup files
20ZZ-XX-YYT12:37:08Z [8145] INFO ArangoDB (version 1.x.y) is ready for business
20ZZ-XX-YYT12:37:08Z [8145] INFO Have Fun!
```

To stop the database server gracefully, you can either press CTRL-C or by send the SIGINT signal to the server process. On many systems this can be achieved with the following command:

```
unix> kill -2 `pidof arangod`
```

Once you started the server, there should be a running instance of *arangod* - the ArangoDB database server.

```
unix> ps auxw | fgrep arangod
arangodb 14536 0.1 0.6 5307264 23464 s002 S 1:21pm 0:00.18 /usr/local/sbin/arangod
```

If there is no such process, check the log file */var/log/arangodb/arangod.log* for errors. If you see a log message like

```
2012-12-03T11:35:29Z [12882] ERROR Database directory version (1) is lower than server version (1.2).
2012-12-03T11:35:29Z [12882] ERROR It seems like you have upgraded the ArangoDB binary. If this is what you wanted to do
2012-12-03T11:35:29Z [12882] FATAL Database version check failed. Please start the server with the --database.auto-upgra
```

make sure to start the server once with the --*database.auto-upgrade* option.

Note that you may have to enable logging first. If you start the server in a shell, you should see errors logged there as well.

Mac OS X

The preferred method for installing ArangoDB under Mac OS X is homebrew. However, in case you are not using homebrew, we provide a command-line app or graphical app which contains all the executables.

Homebrew

If you are using homebrew, then you can install the ArangoDB using *brew* as follows:

```
brew install arangodb
```

This will install the current stable version of ArangoDB and all dependencies within your Homebrew tree. Note that the server will be installed as:

```
/usr/local/sbin/arangod
```

You can start the server by running the command `/usr/local/sbin/arangod &` .

The ArangoDB shell will be installed as:

```
/usr/local/bin/arangosh
```

You can uninstall ArangoDB using:

```
brew uninstall arangodb
```

However, in case you started ArangoDB using the launchctl, you need to unload it before uninstalling the server:

```
launchctl unload ~/Library/LaunchAgents/homebrew.mxcl.arangodb.plist
```

Then remove the LaunchAgent:

```
rm ~/Library/LaunchAgents/homebrew.mxcl.arangodb.plist
```

Note: If the latest ArangoDB Version is not shown in homebrew, you also need to update homebrew:

```
brew update
```

Known issues

- Performance - the LLVM delivered as of Mac OS X El Capitan builds slow binaries. Use GCC instead, until this issue has been fixed by Apple.
- the Commandline argument parsing doesn't accept blanks in filenames; the CLI version below does.

Graphical App

In case you are not using homebrew, we also provide a graphical app. You can download it from here.

Choose *Mac OS X*. Download and install the application *ArangoDB* in your application folder.

Command-Line App

In case you are not using homebrew, we also provide a command-line app. You can download it from here.

Choose *Mac OS X*. Download and install the application *ArangoDB-CLI* in your application folder.

Starting the application will start the server and open a terminal window showing you the log-file.

```
ArangoDB server has been started

The database directory is located at
   '/Applications/ArangoDB-CLI.app/Contents/MacOS/opt/arangodb/var/lib/arangodb'

The log file is located at
   '/Applications/ArangoDB-CLI.app/Contents/MacOS/opt/arangodb/var/log/arangodb/arangod.log'

You can access the server using a browser at 'http://127.0.0.1:8529/'
or start the ArangoDB shell
   '/Applications/ArangoDB-CLI.app/Contents/MacOS/arangosh'

Switching to log-file now, killing this windows will NOT stop the server.


2013-10-27T19:42:04Z [23840] INFO ArangoDB (version 1.4.devel [darwin]) is ready for business. Have fun!
```

Note that it is possible to install both, the homebrew version and the command-line app. You should, however, edit the configuration files of one version and change the port used.

Windows

The default installation directory is *C:\Program Files\ArangoDB-3.x.x*. During the installation process you may change this. In the following description we will assume that ArangoDB has been installed in the location *<ROOTDIR>*.

You have to be careful when choosing an installation directory. You need either write permission to this directory or you need to modify the config file for the server process. In the latter case the database directory and the Foxx directory have to be writable by the user.

Single User Installation

Select a different directory during installation. For example *C:\Users\<Username>\ArangoDB* or *C:\ArangoDB*.

Multiple Users Installation

Keep the default directory. After the installation edit the file *<ROOTDIR>\etc\ArangoDB\arangod.conf*. Adjust the *directory* and *app-path* so that these paths point into your home directory.

```
[database]
directory = @HOMEDRIVE@\@HOMEPATH@\arangodb\databases

[javascript]
app-path = @HOMEDRIVE@\@HOMEPATH@\arangodb\apps
```

Create the directories for each user that wants to use ArangoDB.

Service Installation

Keep the default directory. After the installation open a command line as administrator (search for *cmd* and right click *run as administrator*).

```
cmd> arangod --install-service
INFO: adding service 'ArangoDB - the multi-model database' (internal 'ArangoDB')
INFO: added service with command line '"C:\Program Files (x86)\ArangoDB 3.x.x\bin\arangod.exe" --start-service'
```

Open the service manager and start ArangoDB. In order to enable logging edit the file "\etc\arangodb\arangod.conf" and uncomment the file option.

```
[log]
file = @ROOTDIR@\var\log\arangodb\arangod.log
```

Starting

If you installed ArangoDB as a service it is automatically started.

Otherwise, use the executable *arangod.exe* located in *<ROOTDIR>\bin*. This will use the configuration file *arangod.conf* located in *<ROOTDIR>\etc\arangodb*, which you can adjust to your needs and use the data directory *<ROOTDIR>\var\lib\arangodb*. This is the place where all your data (databases and collections) will be stored by default.

Please check the output of the *arangod.exe* executable before going on. If the server started successfully, you should see a line `ArangoDB is ready for business. Have fun!` at the end of its output.

We now wish to check that the installation is working correctly and to do this we will be using the administration web interface. Execute *arangod.exe* if you have not already done so, then open up your web browser and point it to the page:

```
http://127.0.0.1:8529/
```

Advanced Starting

If you want to provide our own start scripts, you can set the environment variable *ARANGODB_CONFIG_PATH*. This variable should point to a directory containing the configuration files.

Using the Client

To connect to an already running ArangoDB server instance, there is a shell *arangosh.exe* located in *<ROOTDIR>\bin*. This starts a shell which can be used – amongst other things – to administer and query a local or remote ArangoDB server.

Note that *arangosh.exe* does NOT start a separate server, it only starts the shell. To use it you must have a server running somewhere, e.g. by using the *arangod.exe* executable.

arangosh.exe uses configuration from the file *arangosh.conf* located in *<ROOTDIR>\etc\arangodb\*. Please adjust this to your needs if you want to use different connection settings etc.

Uninstalling

To uninstall the Arango server application you can use the windows control panel (as you would normally uninstall an application). Note however, that any data files created by the Arango server will remain as well as the *<ROOTDIR>* directory. To complete the uninstallation process, remove the data files and the *<ROOTDIR>* directory manually.

Limitations for Cygwin

Please note some important limitations when running ArangoDB under Cygwin: Starting ArangoDB can be started from out of a Cygwin terminal, but pressing *CTRL-C* will forcefully kill the server process without giving it a chance to handle the kill signal. In this case, a regular server shutdown is not possible, which may leave a file *LOCK* around in the server's data directory. This file needs to be removed manually to make ArangoDB start again. Additionally, as ArangoDB does not have a chance to handle the kill signal, the server cannot forcefully flush any data to disk on shutdown, leading to potential data loss. When starting ArangoDB from a Cygwin terminal it might also happen that no errors are printed in the terminal output. Starting ArangoDB from an MS-DOS command prompt does not impose these limitations and is thus the preferred method.

Please note that ArangoDB uses UTF-8 as its internal encoding and that the system console must support a UTF-8 codepage (65001) and font. It may be necessary to manually switch the console font to a font that supports UTF-8.

Compiling ArangoDB from scratch

The following sections describe how to compile and build the ArangoDB from scratch. ArangoDB will compile on most Linux and Mac OS X systems. We assume that you use the GNU C/C++ compiler or clang/clang++ to compile the source. ArangoDB has been tested with these compilers, but should be able to compile with any Posix-compliant, C++11-enabled compiler. Please let us know whether you successfully compiled it with another C/C++ compiler.

By default, cloning the github repository will checkout **devel**. This version contains the development version of the ArangoDB. Use this branch if you want to make changes to the ArangoDB source.

Please checkout the cookbook on how to compile ArangoDB.

Authentication

ArangoDB allows to restrict access to databases to certain users. All users of the system database are considered administrators. During installation a default user *root* is created, which has access to all databases.

You should create a database for your application together with a user that has access rights to this database. See Managing Users.

Use the *arangosh* to create a new database and user.

```
arangosh> db._createDatabase("example");
arangosh> var users = require("@arangodb/users");
arangosh> users.save("root@example", "password");
arangosh> users.grantDatabase("root@example", "example");
```

You can now connect to the new database using the user *root@example*.

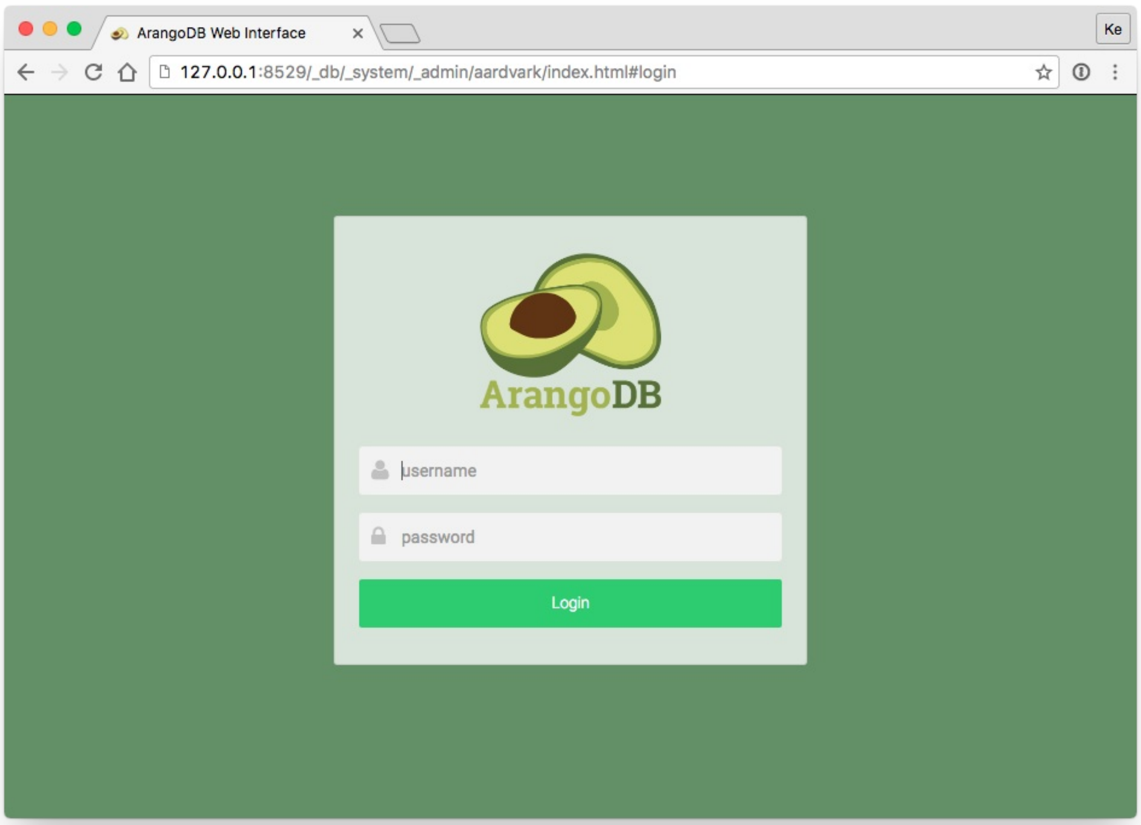
```
shell> arangosh --server.username "root@example" --server.database example
```

Accessing the Web Interface

ArangoDB comes with a built-in web interface for administration. The web interface can be accessed via the URL:

```
http://127.0.0.1:8529
```

If everything works as expected, you should see the login view:



For more information on the ArangoDB web interface, see Web Interface

Coming from SQL

If you worked with a relational database management system (RDBMS) such as MySQL, MariaDB or PostgreSQL, you will be familiar with its query language, a dialect of SQL (Structured Query Language).

ArangoDB's query language is called AQL. There are some similarities between both languages despite the different data models of the database systems. The most notable difference is probably the concept of loops in AQL, which makes it feel more like a programming language. It suites the schema-less model more natural and makes the query language very powerful while remaining easy to read and write.

To get started with AQL, have a look at our detailed comparison of SQL and AQL. It will also help you to translate SQL queries to AQL when migrating to ArangoDB.

How do browse vectors translate into document queries?

In traditional SQL you may either fetch all columns of a table row by row, using `SELECT * FROM table`, or select a subset of the columns. The list of table columns to fetch is commonly called *column list* or *browse vector*:

```
SELECT columnA, columnB, columnZ FROM table
```

Since documents aren't two-dimensional, and neither do you want to be limited to returning two-dimensional lists, the requirements for a query language are higher. AQL is thus a little bit more complex than plain SQL at first, but offers much more flexibility in the long run. It lets you handle arbitrarily structured documents in convenient ways, mostly leaned on the syntax used in JavaScript.

Composing the documents to be returned

The AQL `RETURN` statement returns one item per document it is handed. You can return the whole document, or just parts of it. Given that *oneDocument* is a document (retrieved like `LET oneDocument = DOCUMENT("myusers/3456789")` for instance), it can be returned as-is like this:

```
RETURN oneDocument
```

```
[
    {
        "_id": "myusers/3456789",
        "_key": "3456789"
        "_rev": "14253647",
        "firstName": "John",
        "lastName": "Doe",
        "address": {
            "city": "Gotham",
            "street": "Road To Nowhere 1"
        },
        "hobbies": [
            { name: "swimming", howFavorite: 10 },
            { name: "biking", howFavorite: 6 },
            { name: "programming", howFavorite: 4 }
        ]
    }
]
```

Return the hobbies sub-structure only:

```
RETURN oneDocument.hobbies
```

```
[
    [
        { name: "swimming", howFavorite: 10 },
        { name: "biking", howFavorite: 6 },
        { name: "programming", howFavorite: 4 }
    ]
]
```

Return the hobbies and the address:

```
RETURN {
    hobbies: oneDocument.hobbies,
    address: oneDocument.address
}
```

```
[
    {
        hobbies: [
            { name: "swimming", howFavorite: 10 },
            { name: "biking", howFavorite: 6 },
            { name: "programming", howFavorite: 4 }
        ],
        address: {
            "city": "Gotham",
            "street": "Road To Nowhere 1"
        }
    }
]
```

Return the first hobby only:

```
RETURN oneDocument.hobbies[0].name
```

```
[
    "swimming"
]
```

Return a list of all hobby strings:

```
RETURN { hobbies: oneDocument.hobbies[*].name }
```

```
[
    { hobbies: ["swimming", "biking", "porgramming"] }
]
```

More complex array and object manipulations can be done using AQL functions and operators.

Scalability

ArangoDB is a distributed database supporting multiple data models, and can thus be scaled horizontally, that is, by using many servers, typically based on commodity hardware. This approach not only delivers performance as well as capacity improvements, but also achieves resilience by means of replication and automatic fail-over. Furthermore, one can build systems that scale their capacity dynamically up and down automatically according to demand.

One can also scale ArangoDB vertically, that is, by using ever larger servers. There is no built in limitation in ArangoDB, for example, the server will automatically use more threads if more CPUs are present.

However, scaling vertically has the disadvantage that the costs grow faster than linear with the size of the server, and none of the resilience and dynamical capabilities can be achieved in this way.

In this chapter we explain the distributed architecture of ArangoDB and discuss its scalability features and limitations:

- ArangoDB's distributed architecture
- Different data models and scalability
- Limitations

Architecture

The cluster architecture of ArangoDB is a CP master/master model with no single point of failure. With "CP" we mean that in the presence of a network partition, the database prefers internal consistency over availability. With "master/master" we mean that clients can send their requests to an arbitrary node, and experience the same view on the database regardless. "No single point of failure" means that the cluster can continue to serve requests, even if one machine fails completely.

In this way, ArangoDB has been designed as a distributed multi-model database. This section gives a short outline on the cluster architecture and how the above features and capabilities are achieved.

Structure of an ArangoDB cluster

An ArangoDB cluster consists of a number of ArangoDB instances which talk to each other over the network. They play different roles, which will be explained in detail below. The current configuration of the cluster is held in the "Agency", which is a highly-available resilient key/value store based on an odd number of ArangoDB instances running Raft Consensus Protocol.

For the various instances in an ArangoDB cluster there are 4 distinct roles: Agents, Coordinators, Primary and Secondary DBservers. In the following sections we will shed light on each of them. Note that the tasks for all roles run the same binary from the same Docker image.

Agents

One or multiple Agents form the Agency in an ArangoDB cluster. The Agency is the central place to store the configuration in a cluster. It performs leader elections and provides other synchronization services for the whole cluster. Without the Agency none of the other components can operate.

While generally invisible to the outside it is the heart of the cluster. As such, fault tolerance is of course a must have for the Agency. To achieve that the Agents are using the Raft Consensus Algorithm. The algorithm formally guarantees conflict free configuration management within the ArangoDB cluster.

At its core the Agency manages a big configuration tree. It supports transactional read and write operations on this tree, and other servers can subscribe to HTTP callbacks for all changes to the tree.

Coordinators

Coordinators should be accessible from the outside. These are the ones the clients talk to. They will coordinate cluster tasks like executing queries and running Foxx services. They know where the data is stored and will optimize where to run user supplied queries or parts thereof. Coordinators are stateless and can thus easily be shut down and restarted as needed.

Primary DBservers

Primary DBservers are the ones where the data is actually hosted. They host shards of data and using synchronous replication a primary may either be leader or follower for a shard.

They should not be accessed from the outside but indirectly through the coordinators. They may also execute queries in part or as a whole when asked by a coordinator.

Secondaries

Secondary DBservers are asynchronous replicas of primaries. If one is using only synchronous replication, one does not need secondaries at all. For each primary, there can be one or more secondaries. Since the replication works asynchronously (eventual consistency), the replication does not impede the performance of the primaries. On the other hand, their replica of the data can be slightly out of date. The secondaries are perfectly suitable for backups as they don't interfere with the normal cluster operation.

Cluster ID

Every non-Agency ArangoDB instance in a cluster is assigned a unique ID during its startup. Using its ID a node is identifiable throughout the cluster. All cluster operations will communicate via this ID.

Sharding

Using the roles outlined above an ArangoDB cluster is able to distribute data in so called shards across multiple primaries. From the outside this process is fully transparent and as such we achieve the goals of what other systems call "master-master replication". In an ArangoDB cluster you talk to any coordinator and whenever you read or write data it will automatically figure out where the data is stored (read) or to be stored (write). The information about the shards is shared across the coordinators using the Agency.

Also see Sharding in the Administration chapter.

Many sensible configurations

This architecture is very flexible and thus allows many configurations, which are suitable for different usage scenarios:

1. The default configuration is to run exactly one coordinator and one primary DBserver on each machine. This achieves the classical master/master setup, since there is a perfect symmetry between the different nodes, clients can equally well talk to any one of the coordinators and all expose the same view to the data store.
2. One can deploy more coordinators than DBservers. This is a sensible approach if one needs a lot of CPU power for the Foxx services, because they run on the coordinators.
3. One can deploy more DBservers than coordinators if more data capacity is needed and the query performance is the lesser bottleneck
4. One can deploy a coordinator on each machine where an application server (e.g. a node.js server) runs, and the Agents and DBservers on a separate set of machines elsewhere. This avoids a network hop between the application server and the database and thus decreases latency. Essentially, this moves some of the database distribution logic to the machine where the client runs.

These for shall suffice for now. The important piece of information here is that the coordinator layer can be scaled and deployed independently from the DBserver layer.

Replication

ArangoDB offers two ways of data replication within a cluster, synchronous and asynchronous. In this section we explain some details and highlight the advantages and disadvantages respectively.

Synchronous replication with automatic fail-over

Synchronous replication works on a per-shard basis. One configures for each collection, how many copies of each shard are kept in the cluster. At any given time, one of the copies is declared to be the "leader" and all other replicas are "followers". Write operations for this shard are always sent to the DBserver which happens to hold the leader copy, which in turn replicates the changes to all followers before the operation is considered to be done and reported back to the coordinator. Read operations are all served by the server holding the leader copy, this allows to provide snapshot semantics for complex transactions.

If a DBserver fails that holds a follower copy of a shard, then the leader can no longer synchronize its changes to that follower. After a short timeout (3 seconds), the leader gives up on the follower, declares it to be out of sync, and continues service without the follower. When the server with the follower copy comes back, it automatically resynchronizes its data with the leader and synchronous replication is restored.

If a DBserver fails that holds a leader copy of a shard, then the leader can no longer serve any requests. It will no longer send a heartbeat to the Agency. Therefore, a supervision process running in the Raft leader of the Agency, can take the necessary action (after 15 seconds of missing heartbeats), namely to promote one of the servers that hold in-sync replicas of the shard to leader for that shard. This involves a reconfiguration in the Agency and leads to the fact that coordinators now contact a different DBserver for requests to this shard. Service resumes. The other surviving replicas automatically resynchronize their data with the new leader. When the DBserver with the original leader copy comes back, it notices that it now holds a follower replica, resynchronizes its data with the new leader and order is restored.

All shard data synchronizations are done in an incremental way, such that resynchronizations are quick. This technology allows to move shards (follower and leader ones) between DBservers without service interruptions. Therefore, an ArangoDB cluster can move all the data on a specific DBserver to other DBservers and then shut down that server in a controlled way. This allows to scale down an ArangoDB cluster without service interruption, loss of fault tolerance or data loss. Furthermore, one can re-balance the distribution of the shards, either manually or automatically.

All these operations can be triggered via a REST/JSON API or via the graphical web UI. All fail-over operations are completely handled within the ArangoDB cluster.

Obviously, synchronous replication involves a certain increased latency for write operations, simply because there is one more network hop within the cluster for every request. Therefore the user can set the replication factor to 1, which means that only one copy of each shard is kept, thereby switching off synchronous replication. This is a suitable setting for less important or easily recoverable data for which low latency write operations matter.

Asynchronous replication with automatic fail-over

Asynchronous replication works differently, in that it is organized using primary and secondary DBservers. Each secondary server replicates all the data held on a primary by polling in an asynchronous way. This process has very little impact on the performance of the primary. The disadvantage is that there is a delay between the confirmation of a write operation that is sent to the client and the actual replication of the data. If the master server fails during this delay, then committed and confirmed data can be lost.

Nevertheless, we also offer automatic fail-over with this setup. Contrary to the synchronous case, here the fail-over management is done from outside the ArangoDB cluster. In a future version we might move this management into the supervision process in the Agency, but as of now, the management is done via the Mesos framework scheduler for ArangoDB (see below).

The granularity of the replication is a whole ArangoDB instance with all data that resides on that instance, which means that you need twice as many instances as without asynchronous replication. Synchronous replication is more flexible in that respect, you can have smaller and larger instances, and if one fails, the data can be rebalanced across the remaining ones.

Microservices and zero administation

The design and capabilities of ArangoDB are geared towards usage in modern microservice architectures of applications. With the Foxx services it is very easy to deploy a data centric microservice within an ArangoDB cluster.

In addition, one can deploy multiple instances of ArangoDB within the same project. One part of the project might need a scalable document store, another might need a graph database, and yet another might need the full power of a multi-model database actually mixing the various data models. There are enormous efficiency benefits to be reaped by being able to use a single technology for various roles in a project.

To simplify live of the devops in such a scenario we try as much as possible to use a zero administration approach for ArangoDB. A running ArangoDB cluster is resilient against failures and essentially repairs itself in case of temporary failures. See the next section for further capabilities in this direction.

Apache Mesos integration

For the distributed setup, we use the Apache Mesos infrastructure by default. ArangoDB is a fully certified package for DC/OS and can thus be deployed essentially with a few mouse clicks or a single command, once you have an existing DC/OS cluster. But even on a plain Apache Mesos cluster one can deploy ArangoDB via Marathon with a single API call and some JSON configuration.

The advantage of this approach is that we can not only implement the initial deployment, but also the later management of automatic replacement of failed instances and the scaling of the ArangoDB cluster (triggered manually or even automatically). Since all manipulations are either via the graphical web UI or via JSON/REST calls, one can even implement auto-scaling very easily.

A DC/OS cluster is a very natural environment to deploy microservice architectures, since it is so convenient to deploy various services, including potentially multiple ArangoDB cluster instances within the same DC/OS cluster. The built-in service discovery makes it extremely simple to connect the various microservices and Mesos automatically takes care of the distribution and deployment of the various tasks.

As of June 2016, we offer Apache Mesos integration, later there will be integration with other cluster management infrastructures. See the Deployment chapter and its subsections for instructions.

It is possible to deploy an ArangoDB cluster by simply launching a bunch of Docker containers with the right command line options to link them up, or even on a single machine starting multiple ArangoDB processes. In that case, synchronous replication will work within the deployed ArangoDB cluster, and automatic fail-over in the sense that the duties of a failed server will automatically be assigned to another, surviving one. However, since the ArangoDB cluster cannot within itself launch additional instances, replacement of failed nodes is not automatic and scaling up and down has to be managed manually. This is why we do not recommend this setup for production deployment.

Authentication

As of version 3.0 ArangoDB authentication is **NOT** supported within a cluster. You **HAVE** to properly secure your cluster to the outside. Most setups will have a secured data center anyway and ArangoDB will be accessed from the outside via an application. To this application only the coordinators need to be made available. If you want to isolate even further you can install a reverse proxy like haproxy or nginx in front of the coordinators (that will also allow easy access from the application).

Authentication in the cluster will be added soon after the initial 3.0 release.

Different data models and scalability

In this section we discuss scalability in the context of the different data models supported by ArangoDB.

Key/value pairs

The key/value store data model is the easiest to scale. In ArangoDB, this is implemented in the sense that a document collection always has a primary key `_key` attribute and in the absence of further secondary indexes the document collection behaves like a simple key/value store.

The only operations that are possible in this context are single key lookups and key/value pair insertions and updates. If `_key` is the only sharding attribute then the sharding is done with respect to the primary key and all these operations scale linearly. If the sharding is done using different shard keys, then a lookup of a single key involves asking all shards and thus does not scale linearly.

Document store

For the document store case even in the presence of secondary indexes essentially the same arguments apply, since an index for a sharded collection is simply the same as a local index for each shard. Therefore, single document operations still scale linearly with the size of the cluster, unless a special sharding configuration makes lookups or write operations more expensive.

For a deeper analysis of this topic see this blog post in which good linear scalability of ArangoDB for single document operations is demonstrated.

Complex queries and joins

The AQL query language allows complex queries, using multiple collections, secondary indexes as well as joins. In particular with the latter, scaling can be a challenge, since if the data to be joined resides on different machines, a lot of communication has to happen. The AQL query execution engine organizes a data pipeline across the cluster to put together the results in the most efficient way. The query optimizer is aware of the cluster structure and knows what data is where and how it is indexed. Therefore, it can arrive at an informed decision about what parts of the query ought to run where in the cluster.

Nevertheless, for certain complicated joins, there are limits as to what can be achieved.

Graph database

Graph databases are particularly good at queries on graphs that involve paths in the graph of an a priori unknown length. For example, finding the shortest path between two vertices in a graph, or finding all paths that match a certain pattern starting at a given vertex are such examples.

However, if the vertices and edges along the occurring paths are distributed across the cluster, then a lot of communication is necessary between nodes, and performance suffers. To achieve good performance at scale, it is therefore necessary to get the distribution of the graph data across the shards in the cluster right. Most of the time, the application developers and users of ArangoDB know best, how their graphs ara structured. Therefore, ArangoDB allows users to specify, according to which attributes the graph data is sharded. A useful first step is usually to make sure that the edges originating at a vertex reside on the same cluster node as the vertex.

Limitations

ArangoDB has no built-in limitations to horizontal scalability. The central resilient Agency will easily sustain hundreds of DBservers and coordinators, and the usual database operations work completely decentrally and do not require assistance of the Agency.

Likewise, the supervision process in the Agency can easily deal with lots of servers, since all its activities are not performance critical.

Obviously, an ArangoDB cluster is limited by the available resources of CPU, memory, disk and network bandwidth and latency.

Data models & modeling

This chapter introduces ArangoDB's core concepts and covers

- its data model (or data models respectively),
- the terminology used throughout the database system and in this documentation, as well as
- aspects to consider when modeling your data to strike a balance between natural data structures and great performance

You will also find usage examples on how to interact with the database system using arangosh, e.g. how to create and drop databases / collections, or how to save, update, replace and remove documents. You can do all this using the web interface as well and may therefore skip these sections as beginner.

Concepts

Database Interaction

ArangoDB is a database that serves documents to clients. These documents are transported using JSON via a TCP connection, using the HTTP protocol. A REST API is provided to interact with the database system.

The web interface that comes with ArangoDB, called *Aardvark*, provides graphical user interface that is easy to use. An interactive shell, called *Arangosh*, is also shipped. In addition, there are so called drivers that make it easy to use the database system in various environments and programming languages. All these tools use the HTTP interface of the server and remove the necessity to roll own low-level code for basic communication in most cases.

Data model

The documents you can store in ArangoDB closely follow the JSON format, although they are stored in a binary format called VelocyPack. A **document** contains zero or more attributes, each of these attributes having a value. A value can either be an atomic type, i. e. number, string, boolean or null, or a compound type, i.e. an array or embedded document / object. Arrays and sub-objects can contain all of these types, which means that arbitrarily nested data structures can be represented in a single document.

Documents are grouped into **collections**. A collection contains zero or more documents. If you are familiar with relational database management systems (RDBMS) then it is safe to compare collections to tables and documents to rows. The difference is that in a traditional RDBMS, you have to define columns before you can store records in a table. Such definitions are also known as schemas. ArangoDB is schema-less, which means that there is no need to define what attributes a document can have. Every single document can have a completely different structure and still be stored together with other documents in a single collection. In practice, there will be common denominators among the documents in a collection, but the database system itself doesn't force you to limit yourself to a certain data structure.

There are two types of collections: **document collection** (also refered to as *vertex collections* in the context of graphs) as well as **edge collections**. Edge collections store documents as well, but they include two special attributes, *_from* and *_to*, which are used to create relations between documents. Usually, two documents (**vertices**) stored in document collections are linked by a document (**edge**) stored in an edge collection. This is ArangoDB's graph data model. It follows the mathematical concept of a directed, labeled graph, except that edges don't just have labels, but are full-blown documents.

Collections exist inside of **databases**. There can be one or many databases. Different databases are usually used for multi tenant setups, as the data inside them (collections, documents etc.) is isolated from one another. The default database *_system* is special, because it cannot be removed. Database users are managed in this database, and their credentials are valid for all databases of a server instance.

Data Retrieval

Queries are used to filter documents based on certain criteria, to compute new data, as well as to manipulate or delete existing documents. Queries can be as simple as a "query by example" or as complex as "joins" using many collections or traversing graph structures. They are written in the ArangoDB Query Language (AQL).

Cursors are used to iterate over the result of queries, so that you get easily processable batches instead of one big hunk.

Indexes are used to speed up searches. There are various types of indexes, such as hash indexes and geo indexes.

Handling Databases

This is an introduction to managing databases in ArangoDB from within JavaScript.

When you have an established connection to ArangoDB, the current database can be changed explicitly using the *db._useDatabase()* method. This will switch to the specified database (provided it exists and the user can connect to it). From this point on, any following action in the same shell or connection will use the specified database, unless otherwise specified.

Note: If the database is changed, client drivers need to store the current database name on their side, too. This is because connections in ArangoDB do not contain any state information. All state information is contained in the HTTP request/response data.

To connect to a specific database after arangosh has started use the command described above. It is also possible to specify a database name when invoking arangosh. For this purpose, use the command-line parameter --*server.database*, e.g.

```
> arangosh --server.database test
```

Please note that commands, actions, scripts or AQL queries should never access multiple databases, even if they exist. The only intended and supported way in ArangoDB is to use one database at a time for a command, an action, a script or a query. Operations started in one database must not switch the database later and continue operating in another.

Working with Databases

Database Methods

The following methods are available to manage databases via JavaScript. Please note that several of these methods can be used from the _system database only.

Name

return the database name `db._name()`

Returns the name of the current database as a string.

Examples

```
arangosh> require("@arangodb").db._name();
_system
```

ID

return the database id `db._id()`

Returns the id of the current database as a string.

Examples

```
arangosh> require("@arangodb").db._id();
1
```

Path

return the path to database files `db._path()`

Returns the filesystem path of the current database as a string.

Examples

```
arangosh> require("@arangodb").db._path();
/tmp/arangosh_AXHvJt/tmp-18149-359421054/data/databases/database-1
```

isSystem

return the database type `db._isSystem()`

Returns whether the currently used database is the *_system* database. The system database has some special privileges and properties, for example, database management operations such as create or drop can only be executed from within this database. Additionally, the *_system* database itself cannot be dropped.

Use Database

change the current database `db._useDatabase(name)`

Changes the current database to the database specified by *name*. Note that the database specified by *name* must already exist.

Changing the database might be disallowed in some contexts, for example server-side actions (including Foxx).

When performing this command from arangosh, the current credentials (username and password) will be re-used. These credentials might not be valid to connect to the database specified by *name*. Additionally, the database only be accessed from certain endpoints only. In this case, switching the database might not work, and the connection / session should be closed and restarted with different username and password credentials and/or endpoint data.

List Databases

return the list of all existing databases `db._databases()`

Returns the list of all databases. This method can only be used from within the *_system* database.

Create Database

create a new database `db._createDatabase(name, options, users)`

Creates a new database with the name specified by *name*. There are restrictions for database names (see DatabaseNames).

Note that even if the database is created successfully, there will be no change into the current database to the new database. Changing the current database must explicitly be requested by using the *db._useDatabase* method.

The *options* attribute currently has no meaning and is reserved for future use.

The optional *users* attribute can be used to create initial users for the new database. If specified, it must be a list of user objects. Each user object can contain the following attributes:

- *username*: the user name as a string. This attribute is mandatory.
- *passwd*: the user password as a string. If not specified, then it defaults to an empty string.
- *active*: a boolean flag indicating whether the user account should be active or not. The default value is *true*.
- *extra*: an optional JSON object with extra user information. The data contained in *extra* will be stored for the user but not be interpreted further by ArangoDB.

If no initial users are specified, a default user *root* will be created with an empty string password. This ensures that the new database will be accessible via HTTP after it is created.

You can create users in a database if no initial user is specified. Switch into the new database (username and password must be identical to the current session) and add or modify users with the following commands.

```
require("@arangodb/users").save(username, password, true);
require("@arangodb/users").update(username, password, true);
require("@arangodb/users").remove(username);
```

Alternatively, you can specify user data directly. For example:

```
db._createDatabase("newDB", [], [{ username: "newUser", passwd: "123456", active: true}])
```

Those methods can only be used from within the *_system* database.

Drop Database

drop an existing database `db._dropDatabase(name)`

Drops the database specified by *name*. The database specified by *name* must exist.

Note: Dropping databases is only possible from within the *_system* database. The *_system* database itself cannot be dropped.

Databases are dropped asynchronously, and will be physically removed if all clients have disconnected and references have been garbage-collected.

Notes about Databases

Please keep in mind that each database contains its own system collections, which need to be set up when a database is created. This will make the creation of a database take a while.

Replication is configured on a per-database level, meaning that any replication logging or applying for a new database must be configured explicitly after a new database has been created.

Foxx applications are also available only in the context of the database they have been installed in. A new database will only provide access to the system applications shipped with ArangoDB (that is the web interface at the moment) and no other Foxx applications until they are explicitly installed for the particular database.

JavaScript Interface to Collections

This is an introduction to ArangoDB's interface for collections and how to handle collections from the JavaScript shell *arangosh*. For other languages see the corresponding language API.

The most important call is the call to create a new collection.

Address of a Collection

All collections in ArangoDB have a unique identifier and a unique name. ArangoDB internally uses the collection's unique identifier to look up collections. This identifier, however, is managed by ArangoDB and the user has no control over it. In order to allow users to use their own names, each collection also has a unique name which is specified by the user. To access a collection from the user perspective, the collection name should be used, i.e.:

Collection

`db._collection(collection-name)`

A collection is created by a "db._create" call.

For example: Assume that the collection identifier is *7254820* and the name is *demo*, then the collection can be accessed as:

```
db._collection("demo")
```

If no collection with such a name exists, then *null* is returned.

There is a short-cut that can be used for non-system collections:

Collection name

`db.collection-name`

This call will either return the collection named *db.collection-name* or create a new one with that name and a set of default properties.

Note: Creating a collection on the fly using *db.collection-name* is not recommend and does not work in *arangosh*. To create a new collection, please use

Create

`db._create(collection-name)`

This call will create a new collection called *collection-name*. This method is a database method and is documented in detail at Database Methods

Synchronous replication

Starting in ArangoDB 3.0, the distributed version offers synchronous replication, which means that there is the option to replicate all data automatically within the ArangoDB cluster. This is configured for sharded collections on a per collection basis by specifying a "replication factor" when the collection is created. A replication factor of k means that altogether k copies of each shard are kept in the cluster on k different servers, and are kept in sync. That is, every write operation is automatically replicated on all copies.

This is organised using a leader/follower model. At all times, one of the servers holding replicas for a shard is "the leader" and all others are "followers", this configuration is held in the Agency (see Scalability for details of the ArangoDB cluster architecture). Every write operation is sent to the leader by one of the coordinators, and then replicated to all followers before the operation is reported to have succeeded. The leader keeps a record of which followers are currently in sync. In case of network problems or a failure of a follower, a leader can and will drop a follower temporarily after 3 seconds, such that service can resume. In due course, the follower will automatically resynchronize with the leader to restore resilience.

If a leader fails, the cluster Agency automatically initiates a failover routine after around 15 seconds, promoting one of the followers to leader. The other followers (and the former leader, when it comes back), automatically resynchronize with the new leader to restore resilience. Usually, this whole failover procedure can be handled transparently for the coordinator, such that the user code does not even see an error message.

Obviously, this fault tolerance comes at a cost of increased latency. Each write operation needs an additional network roundtrip for the synchronous replication of the followers, but all replication operations to all followers happen concurrently. This is, why the default replication factor is 1, which means no replication.

For details on how to switch on synchronous replication for a collection, see the database method `db._create(collection-name)` in the section about Database Methods.

Collection Methods

Drop

drops a collection `collection.drop(options)`

Drops a *collection* and all its indexes and data. In order to drop a system collection, an *options* object with attribute *isSystem* set to *true* must be specified.

Examples

```
arangosh> col = db.example;
[ArangoCollection 14451, "example" (type document, status loaded)]
arangosh> col.drop();
arangosh> col;
[ArangoCollection 14451, "example" (type document, status deleted)]
```

```
arangosh> col = db._example;
[ArangoCollection 14454, "_example" (type document, status loaded)]
arangosh> col.drop({ isSystem: true });
arangosh> col;
[ArangoCollection 14454, "_example" (type document, status deleted)]
```

Truncate

truncates a collection `collection.truncate()`

Truncates a *collection*, removing all documents but keeping all its indexes.

Examples

Truncates a collection:

```
arangosh> col = db.example;
arangosh> col.save({ "Hello" : "World" });
arangosh> col.count();
arangosh> col.truncate();
arangosh> col.count();
```

show execution results

Properties

gets or sets the properties of a collection `collection.properties()` Returns an object containing all collection properties.

- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.
- *journalSize* : The size of the journal in bytes.
- *isVolatile*: If *true* then the collection data will be kept in memory only and ArangoDB will not write or sync the data to disk.
- *keyOptions* (optional) additional options for key generation. This is a JSON array containing the following attributes (note: some of the attributes are optional):
  - *type*: the type of the key generator used for the collection.
  - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the *_key* attribute of a document. If set to *false*, then the key generator will solely be responsible for generating keys and supplying own key values in the *_key* attribute of documents is considered an error.
  - *increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
  - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.

- *indexBuckets*: number of buckets into which indexes using a hash table are split. The default is 16 and this number has to be a power of 2 and less than or equal to 1024. For very large collections one should increase this to avoid long pauses when the hash table has to be initially built or resized, since buckets are resized individually and can be initially built in parallel. For example, 64 might be a sensible value for a collection with 100 000 000 documents. Currently, only the edge index respects this value, but other index types might follow in future ArangoDB versions. Changes (see below) are applied when the collection is loaded the next time. In a cluster setup, the result will also contain the following attributes:
- *numberOfShards*: the number of shards of the collection.
- *shardKeys*: contains the names of document attributes that are used to determine the target shard for documents. `collection.properties(properties)` Changes the collection properties. *properties* must be a object with one or more of the following attribute(s):
- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.
- *journalSize* : The size of the journal in bytes.
- *indexBuckets* : See above, changes are only applied when the collection is loaded the next time. *Note*: it is not possible to change the journal size after the journal or datafile has been created. Changing this parameter will only effect newly created journals. Also note that you cannot lower the journal size to less then size of the largest document already stored in the collection. **Note**: some other collection properties, such as *type*, *isVolatile*, or *keyOptions* cannot be changed once the collection is created.

Examples

Read all properties

```
arangosh> db.example.properties();
```

show execution results
Change a property

```
arangosh> db.example.properties({ waitForSync : true });
```

show execution results

Figures

returns the figures of a collection `collection.figures()`

Returns an object containing statistics about the collection. **Note** : Retrieving the figures will always load the collection into memory.

- *alive.count*: The number of currently active documents in all datafiles and journals of the collection. Documents that are contained in the write-ahead log only are not reported in this figure.
- *alive.size*: The total size in bytes used by all active documents of the collection. Documents that are contained in the write-ahead log only are not reported in this figure.
- *dead.count*: The number of dead documents. This includes document versions that have been deleted or replaced by a newer version. Documents deleted or replaced that are contained in the write-ahead log only are not reported in this figure.
- *dead.size*: The total size in bytes used by all dead documents.
- *dead.deletion*: The total number of deletion markers. Deletion markers only contained in the write-ahead log are not reporting in this figure.
- *datafiles.count*: The number of datafiles.
- *datafiles.fileSize*: The total filesize of datafiles (in bytes).
- *journals.count*: The number of journal files.
- *journals.fileSize*: The total filesize of the journal files (in bytes).
- *compactors.count*: The number of compactor files.
- *compactors.fileSize*: The total filesize of the compactor files (in bytes).
- *shapefiles.count*: The number of shape files. This value is deprecated and kept for compatibility reasons only. The value will always be 0 since ArangoDB 2.0 and higher.
- *shapefiles.fileSize*: The total filesize of the shape files. This value is deprecated and kept for compatibility reasons only. The value will always be 0 in ArangoDB 2.0 and higher.
- *shapes.count*: The total number of shapes used in the collection. This includes shapes that are not in use anymore. Shapes that are contained in the write-ahead log only are not reported in this figure.

- *shapes.size*: The total size of all shapes (in bytes). This includes shapes that are not in use anymore. Shapes that are contained in the write-ahead log only are not reported in this figure.
- *attributes.count*: The total number of attributes used in the collection. Note: the value includes data of attributes that are not in use anymore. Attributes that are contained in the write-ahead log only are not reported in this figure.
- *attributes.size*: The total size of the attribute data (in bytes). Note: the value includes data of attributes that are not in use anymore. Attributes that are contained in the write-ahead log only are not reported in this figure.
- *indexes.count*: The total number of indexes defined for the collection, including the pre-defined indexes (e.g. primary index).
- *indexes.size*: The total memory allocated for indexes in bytes.
- *maxTick*: The tick of the last marker that was stored in a journal of the collection. This might be 0 if the collection does not yet have a journal.
- *uncollectedLogfileEntries*: The number of markers in the write-ahead log for this collection that have not been transferred to journals or datafiles.
- *documentReferences*: The number of references to documents in datafiles that JavaScript code currently holds. This information can be used for debugging compaction and unload issues.
- *waitingFor*: An optional string value that contains information about which object type is at the head of the collection's cleanup queue. This information can be used for debugging compaction and unload issues.
- *compactionStatus.time*: The point in time the compaction for the collection was last executed. This information can be used for debugging compaction issues.
- *compactionStatus.message*: The action that was performed when the compaction was last run for the collection. This information can be used for debugging compaction issues.

Note: collection data that are stored in the write-ahead log only are not reported in the results. When the write-ahead log is collected, documents might be added to journals and datafiles of the collection, which may modify the figures of the collection. Also note that `waitingFor` and `compactionStatus` may be empty when called on a coordinator in a cluster.

Additionally, the filesizes of collection and index parameter JSON files are not reported. These files should normally have a size of a few bytes each. Please also note that the *fileSize* values are reported in bytes and reflect the logical file sizes. Some filesystems may use optimisations (e.g. sparse files) so that the actual physical file size is somewhat different. Directories and sub-directories may also require space in the file system, but this space is not reported in the *fileSize* results.

That means that the figures reported do not reflect the actual disk usage of the collection with 100% accuracy. The actual disk usage of a collection is normally slightly higher than the sum of the reported *fileSize* values. Still the sum of the *fileSize* values can still be used as a lower bound approximation of the disk usage.

Examples

```
arangosh> db.demo.figures()
```

show execution results

Load

loads a collection `collection.load()`

Loads a collection into memory.

Examples

```
arangosh> col = db.example;
[ArangoCollection 14542, "example" (type document, status loaded)]
arangosh> col.load();
arangosh> col;
[ArangoCollection 14542, "example" (type document, status loaded)]
```

Reserve

`collection.reserve(number)`

Sends a resize hint to the indexes in the collection. The resize hint allows indexes to reserve space for additional documents (specified by number) in one go.

The reserve hint can be sent before a mass insertion into the collection is started. It allows indexes to allocate the required memory at once and avoids re-allocations and possible re-locations.

Not all indexes implement the reserve function at the moment. The indexes that don't implement it will simply ignore the request. returns the revision id of a collection

Revision

returns the revision id of a collection `collection.revision()`

Returns the revision id of the collection

The revision id is updated when the document data is modified, either by inserting, deleting, updating or replacing documents in it.

The revision id of a collection can be used by clients to check whether data in a collection has changed or if it is still unmodified since a previous fetch of the revision id.

The revision id returned is a string value. Clients should treat this value as an opaque string, and only use it for equality/non-equality comparisons.

Checksum

calculates a checksum for the data in a collection `collection.checksum(withRevisions, withData)`

The *checksum* operation calculates an aggregate hash value for all document keys contained in collection *collection*.

If the optional argument *withRevisions* is set to *true*, then the revision ids of the documents are also included in the hash calculation.

If the optional argument *withData* is set to *true*, then all user-defined document attributes are also checksummed. Including the document data in checksumming will make the calculation slower, but is more accurate.

The checksum calculation algorithm changed in ArangoDB 3.0, so checksums from 3.0 and earlier versions for the same data will differ.

Note: this method is not available in a cluster.

Unload

unloads a collection `collection.unload()`

Starts unloading a collection from memory. Note that unloading is deferred until all query have finished.

Examples

```
arangosh> col = db.example;
[ArangoCollection 7351, "example" (type document, status loaded)]
arangosh> col.unload();
arangosh> col;
[ArangoCollection 7351, "example" (type document, status unloaded)]
```

Rename

renames a collection `collection.rename(new-name)`

Renames a collection using the *new-name*. The *new-name* must not already be used for a different collection. *new-name* must also be a valid collection name. For more information on valid collection names please refer to the naming conventions.

If renaming fails for any reason, an error is thrown. If renaming the collection succeeds, then the collection is also renamed in all graph definitions inside the `_graphs` collection in the current database.

Note: this method is not available in a cluster.

Examples

```
arangosh> c = db.example;
[ArangoCollection 14620, "example" (type document, status loaded)]
arangosh> c.rename("better-example");
arangosh> c;
[ArangoCollection 14620, "better-example" (type document, status loaded)]
```

Rotate

rotates the current journal of a collection `collection.rotate()`

Rotates the current journal of a collection. This operation makes the current journal of the collection a read-only datafile so it may become a candidate for garbage collection. If there is currently no journal available for the collection, the operation will fail with an error.

Note: this method is not available in a cluster.

Database Methods

Collection

returns a single collection or null `db._collection(collection-name)`

Returns the collection with the given name or null if no such collection exists.

`db._collection(collection-identifier)`

Returns the collection with the given identifier or null if no such collection exists. Accessing collections by identifier is discouraged for end users. End users should access collections using the collection name.

Examples

Get a collection by name:

```
arangosh> db._collection("demo");
[ArangoCollection 93, "demo" (type document, status loaded)]
```

Get a collection by id:

```
arangosh> db._collection(123456);
[ArangoCollection 123456, "demo" (type document, status loaded)]
```

Unknown collection:

```
arangosh> db._collection("unknown");
null
```

Create

creates a new document or edge collection `db._create(collection-name)`

Creates a new document collection named *collection-name*. If the collection name already exists or if the name format is invalid, an error is thrown. For more information on valid collection names please refer to the naming conventions.

`db._create(collection-name, properties)`

properties must be an object with the following attributes:

- *waitForSync* (optional, default *false*): If *true* creating a document will only return after the data was synced to disk.
- *journalSize* (optional, default is a configuration parameter: The maximal size of a journal or datafile. Note that this also limits the maximal size of a single object. Must be at least 1MB.
- *isSystem* (optional, default is *false*): If *true*, create a system collection. In this case *collection-name* should start with an underscore. End users should normally create non-system collections only. API implementors may be required to create system collections in very special occasions, but normally a regular collection will do.
- *isVolatile* (optional, default is *false*): If *true* then the collection data is kept in-memory only and not made persistent. Unloading the collection will cause the collection data to be discarded. Stopping or re-starting the server will also cause full loss of data in the collection. The collection itself will remain however (only the data is volatile). Setting this option will make the resulting collection be slightly faster than regular collections because ArangoDB does not enforce any synchronization to disk and does not calculate any CRC checksums for datafiles (as there are no datafiles).
- *keyOptions* (optional): additional options for key generation. If specified, then *keyOptions* should be a JSON array containing the following attributes (**note**: some of them are optional):
  - *type*: specifies the type of the key generator. The currently available generators are *traditional* and *autoincrement*.
  - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the *_key* attribute of a document. If set to *false*,

then the key generator will solely be responsible for generating keys and supplying own key values in the *_key* attribute of documents is considered an error.
  - *increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
  - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.
- *numberOfShards* (optional, default is *1*): in a cluster, this value determines the number of shards to create for the collection. In a single server setup, this option is meaningless.

- *shardKeys* (optional, default is `[ "_key" ]`): in a cluster, this attribute determines which document attributes are used to determine the target shard for documents. Documents are sent to shards based on the values they have in their shard key attributes. The values of all shard key attributes in a document are hashed, and the hash value is used to determine the target shard. Note that values of shard key attributes cannot be changed once set. This option is meaningless in a single server setup.

  When choosing the shard keys, one must be aware of the following rules and limitations: In a sharded collection with more than one shard it is not possible to set up a unique constraint on an attribute that is not the one and only shard key given in *shardKeys*. This is because enforcing a unique constraint would otherwise make a global index necessary or need extensive communication for every single write operation. Furthermore, if *_key* is not the one and only shard key, then it is not possible to set the *_key* attribute when inserting a document, provided the collection has more than one shard. Again, this is because the database has to enforce the unique constraint on the *_key* attribute and this can only be done efficiently if this is the only shard key by delegating to the individual shards.

- *replicationFactor* (optional, default is 1): in a cluster, this attribute determines how many copies of each shard are kept on different DBServers. The value 1 means that only one copy (no synchronous replication) is kept. A value of k means that k-1 replicas are kept. Any two copies reside on different DBServers. Replication between them is synchronous, that is, every write operation to the "leader" copy will be replicated to all "follower" replicas, before the write operation is reported successful.

  If a server fails, this is detected automatically and one of the servers holding copies take over, usually without an error being reported.

`db._create(collection-name, properties, type)`

Specifies the optional *type* of the collection, it can either be *document* or *edge*. On default it is document. Instead of giving a type you can also use *db._createEdgeCollection* or *db._createDocumentCollection*.

Examples

With defaults:

```
arangosh> c = db._create("users");
arangosh> c.properties();
```

show execution results
With properties:

```
arangosh> c = db._create("users", { waitForSync : true,
........> journalSize : 1024 * 1204});
arangosh> c.properties();
```

show execution results
With a key generator:

```
arangosh> db._create("users",
........> { keyOptions: { type: "autoincrement", offset: 10, increment: 5 } });
arangosh> db.users.save({ name: "user 1" });
arangosh> db.users.save({ name: "user 2" });
arangosh> db.users.save({ name: "user 3" });
```

show execution results
With a special key option:

```
arangosh> db._create("users", { keyOptions: { allowUserKeys: false } });
arangosh> db.users.save({ name: "user 1" });
arangosh> db.users.save({ name: "user 2", _key: "myuser" });
arangosh> db.users.save({ name: "user 3" });
```

show execution results

creates a new edge collection `db._createEdgeCollection(collection-name)`

Creates a new edge collection named *collection-name*. If the collection name already exists an error is thrown. The default value for *waitForSync* is *false*.

`db._createEdgeCollection(collection-name, properties)`

properties must be an object with the following attributes:

- *waitForSync* (optional, default *false*): If *true* creating a document will only return after the data was synced to disk.
- *journalSize* (optional, default is "configuration parameter"): The maximal size of a journal or datafile. Note that this also limits the maximal size of a single object and must be at least 1MB.

creates a new document collection `db._createDocumentCollection(collection-name)`

Creates a new document collection named *collection-name*. If the document name already exists and error is thrown.

All Collections

returns all collections `db._collections()`

Returns all collections of the given database.

Examples

```
arangosh> db._collections();
```

show execution results

Collection Name

selects a collection from the vocbase `db.collection-name`

Returns the collection with the given *collection-name*. If no such collection exists, create a collection named *collection-name* with the default properties.

Examples

```
arangosh> db.example;
[ArangoCollection 14343, "example" (type document, status loaded)]
```

Drop

drops a collection `db._drop(collection)`

Drops a *collection* and all its indexes and data.

`db._drop(collection-identifier)`

Drops a collection identified by *collection-identifier* with all its indexes and data. No error is thrown if there is no such collection.

`db._drop(collection-name)`

Drops a collection named *collection-name* and all its indexes. No error is thrown if there is no such collection.

`db._drop(collection-name, options)`

In order to drop a system collection, one must specify an *options* object with attribute *isSystem* set to *true*. Otherwise it is not possible to drop system collections.

Examples

Drops a collection:

```
arangosh> col = db.example;
[ArangoCollection 14394, "example" (type document, status loaded)]
arangosh> db._drop(col);
arangosh> col;
[ArangoCollection 14394, "example" (type document, status loaded)]
```

Drops a collection identified by name:

```
arangosh> col = db.example;
[ArangoCollection 14397, "example" (type document, status loaded)]
arangosh> db._drop("example");
arangosh> col;
[ArangoCollection 14397, "example" (type document, status deleted)]
```

Drops a system collection

```
arangosh> col = db._example;
[ArangoCollection 14400, "_example" (type document, status loaded)]
arangosh> db._drop("_example", { isSystem: true });
arangosh> col;
[ArangoCollection 14400, "_example" (type document, status deleted)]
```

Truncate

truncates a collection `db._truncate(collection)`

Truncates a *collection*, removing all documents but keeping all its indexes.

`db._truncate(collection-identifier)`

Truncates a collection identified by *collection-identified*. No error is thrown if there is no such collection.

`db._truncate(collection-name)`

Truncates a collection named *collection-name*. No error is thrown if there is no such collection.

Examples

Truncates a collection:

```
arangosh> col = db.example;
arangosh> col.save({ "Hello" : "World" });
arangosh> col.count();
arangosh> db._truncate(col);
arangosh> col.count();
```

show execution results
Truncates a collection identified by name:

```
arangosh> col = db.example;
arangosh> col.save({ "Hello" : "World" });
arangosh> col.count();
arangosh> db._truncate("example");
arangosh> col.count();
```

show execution results

Documents

This is an introduction to ArangoDB's interface for working with documents from the JavaScript shell *arangosh* or in JavaScript code in the server. For other languages see the corresponding language API.

- Basics and Terminology: section on the basic approach
- Collection Methods: detailed API description for collection objects
- Database Methods: detailed API description for database objects

Basics and Terminology

Documents in ArangoDB are JSON objects. These objects can be nested (to any depth) and may contain lists. Each document has a unique primary key which identifies it within its collection. Furthermore, each document is uniquely identified by its document handle across all collections in the same database. Different revisions of the same document (identified by its handle) can be distinguished by their document revision. Any transaction only ever sees a single revision of a document. For example:

```
{
  "_id" : "myusers/3456789",
  "_key" : "3456789",
  "_rev" : "14253647",
  "firstName" : "John",
  "lastName" : "Doe",
  "address" : {
    "street" : "Road To Nowhere 1",
    "city" : "Gotham"
  },
  "hobbies" : [
    {name: "swimming", howFavorite: 10},
    {name: "biking", howFavorite: 6},
    {name: "programming", howFavorite: 4}
  ]
}
```

All documents contain special attributes: the document handle is stored as a string in `_id`, the document's primary key in `_key` and the document revision in `_rev`. The value of the `_key` attribute can be specified by the user when creating a document. `_id` and `_key` values are immutable once the document has been created. The `_rev` value is maintained by ArangoDB automatically.

Document Handle

A document handle uniquely identifies a document in the database. It is a string and consists of the collection's name and the document key (`_key` attribute) separated by `/`.

Document Key

A document key uniquely identifies a document in the collection it is stored in. It can and should be used by clients when specific documents are queried. The document key is stored in the `_key` attribute of each document. The key values are automatically indexed by ArangoDB in a collection's primary index. Thus looking up a document by its key is a fast operation. The _key value of a document is immutable once the document has been created. By default, ArangoDB will auto-generate a document key if no _key attribute is specified, and use the user-specified _key otherwise. The generated _key is guaranteed to be unique in the collection it was generated for. This also applies to sharded collections in a cluster. It can't be guaranteed that the _key is unique within a database or across a whole node or instance however.

This behavior can be changed on a per-collection level by creating collections with the `keyOptions` attribute.

Using `keyOptions` it is possible to disallow user-specified keys completely, or to force a specific regime for auto-generating the `_key` values.

Document Revision

As ArangoDB supports MVCC (Multiple Version Concurrency Control), documents can exist in more than one revision. The document revision is the MVCC token used to specify a particular revision of a document (identified by its `_id`). It is a string value that contained (up to ArangoDB 3.0) an integer number and is unique within the list of document revisions for a single document. In ArangoDB >= 3.1 the _rev strings are in fact time stamps. They use the local clock of the DBserver that actually writes the document and have millisecond accuracy. Actually, a "Hybrid Logical Clock" is used (for this concept see this paper).

Within one shard it is guaranteed that two different document revisions have a different _rev string, even if they are written in the same millisecond, and that these stamps are ascending.

Note however that different servers in your cluster might have a clock skew, and therefore between different shards or even between different collections the time stamps are not guaranteed to be comparable.

The Hybrid Logical Clock feature does one thing to address this issue: Whenever a message is sent from some server A in your cluster to another one B, it is ensured that any timestamp taken on B after the message has arrived is greater than any timestamp taken on A before the message was sent. This ensures that if there is some "causality" between events on different servers, time stamps increase from cause to effect. A direct consequence of this is that sometimes a server has to take timestamps that seem to come from the future of its own clock. It will however still produce ever increasing timestamps. If the clock skew is small, then your timestamps will relatively accurately describe the time when the document revision was actually written.

ArangoDB uses 64bit unsigned integer values to maintain document revisions internally. At this stage we intentionally do not document the exact format of the revision values. When returning document revisions to clients, ArangoDB will put them into a string to ensure the revision is not clipped by clients that do not support big integers. Clients should treat the revision returned by ArangoDB as an opaque string when they store or use it locally. This will allow ArangoDB to change the format of revisions later if this should be required (as has happened with 3.1 with the Hybrid Logical Clock). Clients can use revisions to perform simple equality/non-equality comparisons (e.g. to check whether a document has changed or not), but they should not use revision ids to perform greater/less than comparisons with them to check if a document revision is older than one another, even if this might work for some cases.

Document revisions can be used to conditionally query, update, replace or delete documents in the database. In order to find a particular revision of a document, you need the document handle or key, and the document revision.

Multiple Documents in a single Command

Beginning with ArangoDB 3.0 the basic document API has been extended to handle not only single documents but multiple documents in a single command. This is crucial for performance, in particular in the cluster situation, in which a single request can involve multiple network hops within the cluster. Another advantage is that it reduces the overhead of individual network round trips between the client and the server. The general idea to perform multiple document operations in a single command is to use JSON arrays of objects in the place of a single document. As a consequence, document keys, handles and revisions for preconditions have to be supplied embedded in the individual documents given. Multiple document operations are restricted to a single document or edge collection. See the API descriptions for collection objects for details. Note that the API for database objects do not offer these operations.

Collection Methods

All

`collection.all()`

Fetches all documents from a collection and returns a cursor. You can use *toArray*, *next*, or *hasNext* to access the result. The result can be limited using the *skip* and *limit* operator.

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.five.save({ name : "one" });
arangosh> db.five.save({ name : "two" });
arangosh> db.five.save({ name : "three" });
arangosh> db.five.save({ name : "four" });
arangosh> db.five.save({ name : "five" });
arangosh> db.five.all().toArray();
```

show execution results
Use *limit* to restrict the documents:

```
arangosh> db.five.save({ name : "one" });
arangosh> db.five.save({ name : "two" });
arangosh> db.five.save({ name : "three" });
arangosh> db.five.save({ name : "four" });
arangosh> db.five.save({ name : "five" });
arangosh> db.five.all().limit(2).toArray();
```

show execution results

Query by example

`collection.byExample(example)`

Fetches all documents from a collection that match the specified example and returns a cursor.

You can use *toArray*, *next*, or *hasNext* to access the result. The result can be limited using the *skip* and *limit* operator.

An attribute name of the form *a.b* is interpreted as attribute path, not as attribute. If you use

```
{ "a" : { "c" : 1 } }
```

as example, then you will find all documents, such that the attribute *a* contains a document of the form *{c : 1 }*. For example the document

```
{ "a" : { "c" : 1 }, "b" : 1 }
```

will match, but the document

```
{ "a" : { "c" : 1, "b" : 1 } }
```

will not.

However, if you use

```
{ "a.c" : 1 }
```

then you will find all documents, which contain a sub-document in *a* that has an attribute *c* of value *1*. Both the following documents

```
{ "a" : { "c" : 1 }, "b" : 1 }
```

and

```
{ "a" : { "c" : 1, "b" : 1 } }
```

will match.

```
collection.byExample(path1, value1, ...)
```

As alternative you can supply an array of paths and values.

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.users.save({ name: "Gerhard" });
arangosh> db.users.save({ name: "Helmut" });
arangosh> db.users.save({ name: "Angela" });
arangosh> db.users.all().toArray();
arangosh> db.users.byExample({ "_id" : "users/20" }).toArray();
arangosh> db.users.byExample({ "name" : "Gerhard" }).toArray();
arangosh> db.users.byExample({ "name" : "Helmut", "_id" : "users/15" }).toArray();
```

show execution results
Use *next* to loop over all documents:

```
arangosh> db.users.save({ name: "Gerhard" });
arangosh> db.users.save({ name: "Helmut" });
arangosh> db.users.save({ name: "Angela" });
arangosh> var a = db.users.byExample( {"name" : "Angela" } );
arangosh> while (a.hasNext()) print(a.next());
```

show execution results

First Example

`collection.firstExample(example)`

Returns some document of a collection that matches the specified example. If no such document exists, *null* will be returned. The example has to be specified as paths and values. See *byExample* for details.

`collection.firstExample(path1, value1, ...)`

As alternative you can supply an array of paths and values.

Examples

```
arangosh> db.users.firstExample("name", "Angela");
```

show execution results

Range

`collection.range(attribute, left, right)`

Returns all documents from a collection such that the *attribute* is greater or equal than *left* and strictly less than *right*.

You can use *toArray*, *next*, or *hasNext* to access the result. The result can be limited using the *skip* and *limit* operator.

An attribute name of the form *a.b* is interpreted as attribute path, not as attribute.

Note: the *range* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for retrieving documents from a collection within a specific range is to use an AQL query as follows:

```
FOR doc IN @@collection
  FILTER doc.value >= @left && doc.value < @right
  LIMIT @skip, @limit
  RETURN doc
```

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.old.ensureIndex({ type: "skiplist", fields: [ "age" ] });
arangosh> db.old.save({ age: 15 });
arangosh> db.old.save({ age: 25 });
arangosh> db.old.save({ age: 30 });
arangosh> db.old.range("age", 10, 30).toArray();
```

show execution results

Closed range

`collection.closedRange(attribute, left, right)`

Returns all documents of a collection such that the *attribute* is greater or equal than *left* and less or equal than *right*.

You can use *toArray*, *next*, or *hasNext* to access the result. The result can be limited using the *skip* and *limit* operator.

An attribute name of the form *a.b* is interpreted as attribute path, not as attribute.

Note: the *closedRange* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for retrieving documents from a collection within a specific range is to use an AQL query as follows:

```
FOR doc IN @@collection
  FILTER doc.value >= @left && doc.value <= @right
  LIMIT @skip, @limit
  RETURN doc
```

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.old.ensureIndex({ type: "skiplist", fields: [ "age" ] });
arangosh> db.old.save({ age: 15 });
arangosh> db.old.save({ age: 25 });
arangosh> db.old.save({ age: 30 });
arangosh> db.old.closedRange("age", 10, 30).toArray();
```

show execution results

Any

`collection.any()`

Returns a random document from the collection or *null* if none exists.

Count

`collection.count()`

Returns the number of living documents in the collection.

Examples

```
arangosh> db.users.count();
0
```

toArray

`collection.toArray()`

Converts the collection into an array of documents. Never use this call in a production environment as it will basically create a copy of your collection in RAM which will use resources depending on the number and size of the documents in your collecion.

Document

`collection.document(object)`

The *document* method finds a document given an object *object* containing the *_id* or *_key* attribute. The method returns the document if it can be found. If both attributes are given, the *_id* takes precedence, it is an error, if the collection part of the *_id* does not match the *collection*.

An error is thrown if *_rev* is specified but the document found has a different revision already. An error is also thrown if no document exists with the given *_id* or *_key* value.

Please note that if the method is executed on the arangod server (e.g. from inside a Foxx application), an immutable document object will be returned for performance reasons. It is not possible to change attributes of this immutable object. To update or patch the returned document, it needs to be cloned/copied into a regular JavaScript object first. This is not necessary if the *document* method is called from out of arangosh or from any other client.

`collection.document(document-handle)`

As before. Instead of *object* a *document-handle* can be passed as first argument. No revision can be specified in this case.

`collection.document(document-key)`

As before. Instead of *object* a *document-key* can be passed as first argument.

`collection.document(array)`

This variant allows to perform the operation on a whole array of arguments. The behavior is exactly as if *document* would have been called on all members of the array and all results are returned in an array. If an error occurs with any of the documents, the operation stops immediately returning only an error object.

Examples

Returns the document for a document-handle:

```
arangosh> db.example.document("example/2873916");
```

show execution results
Returns the document for a document-key:

```
arangosh> db.example.document("2873916");
```

show execution results
Returns the document for an object:

```
arangosh> db.example.document({_id: "example/2873916"});
```

show execution results

Returns the document for an array of two keys:

```
arangosh> db.example.document(["2873916","2873917"]);
```

show execution results

An error is raised if the document is unknown:

```
arangosh> db.example.document("example/4472917");
[ArangoError 1202: document not found]
```

An error is raised if the handle is invalid:

```
arangosh> db.example.document("");
[ArangoError 1205: illegal document handle]
```

Changes in 3.0 from 2.8:

document can now query multiple documents with one call.

Exists

checks whether a document exists `collection.exists(object)`

The *exists* method determines whether a document exists given an object `object` containing the *_id* or *_key* attribute. If both attributes are given, the *_id* takes precedence, it is an error, if the collection part of the *_id* does not match the *collection*.

An error is thrown if *_rev* is specified but the document found has a different revision already.

Instead of returning the found document or an error, this method will only return an object with the attributes *_id*, *_key* and *_rev*, or *false* if no document with the given *_id* or *_key* exists. It can thus be used for easy existence checks.

This method will throw an error if used improperly, e.g. when called with a non-document handle, a non-document, or when a cross-collection request is performed.

`collection.exists(document-handle)`

As before. Instead of *object* a *document-handle* can be passed as first argument.

`collection.exists(document-key)`

As before. Instead of *object* a *document-key* can be passed as first argument.

`collection.exists(array)`

This variant allows to perform the operation on a whole array of arguments. The behavior is exactly as if *exists* would have been called on all members of the array and all results are returned in an array. If an error occurs with any of the documents, the operation stops immediately returning only an error object.

Changes in 3.0 from 2.8:

In the case of a revision mismatch *exists* now throws an error instead of simply returning *false*. This is to make it possible to tell the difference between a revision mismatch and a non-existing document.

exists can now query multiple documents with one call.

Lookup By Keys

`collection.documents(keys)`

Looks up the documents in the specified collection using the array of keys provided. All documents for which a matching key was specified in the *keys* array and that exist in the collection will be returned. Keys for which no document can be found in the underlying collection are ignored, and no exception will be thrown for them.

This method is deprecated in favour of the array variant of *document*.

Examples

```
arangosh> keys = [ ];
arangosh> for (var i = 0; i < 10; ++i) {
........>   db.example.insert({ _key: "test" + i, value: i });
........>   keys.push("test" + i);
........> }
arangosh> db.example.documents(keys);
```

show execution results

Insert

`collection.insert(data)`

Creates a new document in the *collection* from the given *data*. The *data* must be an object. The attributes *_id* and *_rev* are ignored and are automatically generated. A unique value for the attribute *_key* will be automatically generated if not specified. If specified, there must not be a document with the given *_key* in the collection.

The method returns a document with the attributes *_id*, *_key* and *_rev*. The attribute *_id* contains the document handle of the newly created document, the attribute *_key* the document key and the attribute *_rev* contains the document revision.

`collection.insert(data, options)`

Creates a new document in the *collection* from the given *data* as above. The optional *options* parameter must be an object and can be used to specify the following options:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *silent*: If this flag is set to *true*, the method does not return any output.
- *returnNew*: If this flag is set to *true*, the complete new document is returned in the output under the attribute *new*.

Note: since ArangoDB 2.2, *insert* is an alias for *save*.

`collection.insert(array)`

`collection.insert(array, options)`

These two variants allow to perform the operation on a whole array of arguments. The behavior is exactly as if *insert* would have been called on all members of the array and all results are returned in an array. If an error occurs with any of the documents, the operation stops immediately returning only an error object. The options behave exactly as before.

Changes in 3.0 from 2.8:

The options *silent* and *returnNew* are new. The method can now insert multiple documents with one call.

Examples

```
arangosh> db.example.insert({ Hello : "World" });
arangosh> db.example.insert({ Hello : "World" }, {waitForSync: true});
```

show execution results

```
arangosh> db.example.insert([{ Hello : "World" }, {Hello: "there"}])
arangosh> db.example.insert([{ Hello : "World" }, {}], {waitForSync: true});
```

show execution results

Replace

`collection.replace(selector, data)`

Replaces an existing document described by the *selector*, which must be an object containing the *_id* or *_key* attribute. There must be a document with that *_id* or *_key* in the current collection. This document is then replaced with the *data* given as second argument. Any attribute *_id*, *_key* or *_rev* in *data* is ignored.

The method returns a document with the attributes *_id*, *_key*, *_rev* and *_oldRev*. The attribute *_id* contains the document handle of the updated document, the attribute *_rev* contains the document revision of the updated document, the attribute *_oldRev* contains the revision of the old (now replaced) document.

If the selector contains a *_rev* attribute, the method first checks that the specified revision is the current revision of that document. If not, there is a conflict, and an error is thrown.

`collection.replace(selector, data, options)`

As before, but *options* must be an object that can contain the following boolean attributes:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *overwrite*: If this flag is set to *true*, a *_rev* attribute in the selector is ignored.
- *returnNew*: If this flag is set to *true*, the complete new document is returned in the output under the attribute *new*.
- *returnOld*: If this flag is set to *true*, the complete previous revision of the document is returned in the output under the attribute *old*.
- *silent*: If this flag is set to *true*, no output is returned.

`collection.replace(document-handle, data)`

`collection.replace(document-handle, data, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision precondition is tested.

`collection.replace(document-key, data)`

`collection.replace(document-key, data, options)`

As before. Instead of *selector* a *document-key* can be passed as first argument. No revision precondition is tested.

`collection.replace(selectorarray, dataarray)`

`collection.replace(selectorarray, dataarray, options)`

These two variants allow to perform the operation on a whole array of selector/data pairs. The two arrays given as *selectorarray* and *dataarray* must have the same length. The behavior is exactly as if *replace* would have been called on all respective members of the two arrays and all results are returned in an array. If an error occurs with any of the documents, the operation stops immediately returning only an error object. The options behave exactly as before.

Examples

Create and update a document:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> a2 = db.example.replace(a1, { a : 2 });
arangosh> a3 = db.example.replace(a1, { a : 3 });
```

show execution results
Use a document handle:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> a2 = db.example.replace("example/3903044", { a : 2 });
```

show execution results

Changes in 3.0 from 2.8:

The options *silent*, *returnNew* and *returnOld* are new. The method can now replace multiple documents with one call.

Update

`collection.update(selector, data)`

Updates an existing document described by the *selector*, which must be an object containing the *_id* or *_key* attribute. There must be a document with that *_id* or *_key* in the current collection. This document is then patched with the *data* given as second argument. Any attribute *_id*, *_key* or *_rev* in *data* is ignored.

The method returns a document with the attributes *_id*, *_key*, *_rev* and *_oldRev*. The attribute *_id* contains the document handle of the updated document, the attribute *_rev* contains the document revision of the updated document, the attribute *_oldRev* contains the revision of the old (now updated) document.

If the selector contains a *_rev* attribute, the method first checks that the specified revision is the current revision of that document. If not, there is a conflict, and an error is thrown.

`collection.update(selector, data, options)`

As before, but *options* must be an object that can contain the following boolean attributes:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *overwrite*: If this flag is set to *true*, a *_rev* attribute in the selector is ignored.
- *returnNew*: If this flag is set to *true*, the complete new document is returned in the output under the attribute *new*.
- *returnOld*: If this flag is set to *true*, the complete previous revision of the document is returned in the output under the attribute *old*.
- *silent*: If this flag is set to *true*, no output is returned.
- *keepNull*: The optional *keepNull* parameter can be used to modify the behavior when handling *null* values. Normally, *null* values are stored in the database. By setting the *keepNull* parameter to *false*, this behavior can be changed so that all attributes in *data* with *null* values will be removed from the target document.
- *mergeObjects*: Controls whether objects (not arrays) will be merged if present in both the existing and the patch document. If set to *false*, the value in the patch document will overwrite the existing document's value. If set to *true*, objects will be merged. The default is *true*.

`collection.update(document-handle, data)`

`collection.update(document-handle, data, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision precondition is tested.

`collection.update(document-key, data)`

`collection.update(document-key, data, options)`

As before. Instead of *selector* a *document-key* can be passed as first argument. No revision precondition is tested.

`collection.update(selectorarray, dataarray)`

`collection.update(selectorarray, dataarray, options)`

These two variants allow to perform the operation on a whole array of selector/data pairs. The two arrays given as *selectorarray* and *dataarray* must have the same length. The behavior is exactly as if *update* would have been called on all respective members of the two arrays and all results are returned in an array. If an error occurs with any of the documents, the operation stops immediately returning only an error object. The options behave exactly as before.

Examples

Create and update a document:

```
arangosh> a1 = db.example.insert({"a" : 1});
arangosh> a2 = db.example.update(a1, {"b" : 2, "c" : 3});
arangosh> a3 = db.example.update(a1, {"d" : 4});
arangosh> a4 = db.example.update(a2, {"e" : 5, "f" : 6 });
arangosh> db.example.document(a4);
arangosh> a5 = db.example.update(a4, {"a" : 1, c : 9, e : 42 });
arangosh> db.example.document(a5);
```

show execution results

Use a document handle:

```
arangosh> a1 = db.example.insert({"a" : 1});
arangosh> a2 = db.example.update("example/18612115", { "x" : 1, "y" : 2 });
```

show execution results

Use the keepNull parameter to remove attributes with null values:

```
arangosh> db.example.insert({"a" : 1});
arangosh> db.example.update("example/19988371",
........> { "b" : null, "c" : null, "d" : 3 });
arangosh> db.example.document("example/19988371");
arangosh> db.example.update("example/19988371", { "a" : null }, false, false);
arangosh> db.example.document("example/19988371");
arangosh> db.example.update("example/19988371",
........> { "b" : null, "c": null, "d" : null }, false, false);
arangosh> db.example.document("example/19988371");
```

show execution results

Patching array values:

```
arangosh>  db.example.insert({"a" : { "one" : 1, "two" : 2, "three" : 3 },
........> "b" : { }});
arangosh> db.example.update("example/20774803", {"a" : { "four" : 4 },
........> "b" : { "b1" : 1 }});
arangosh> db.example.document("example/20774803");
arangosh> db.example.update("example/20774803", { "a" : { "one" : null },
........>                                         "b" : null },
........> false, false);
arangosh> db.example.document("example/20774803");
```

show execution results

Changes in 3.0 from 2.8:

The options *silent*, *returnNew* and *returnOld* are new. The method can now update multiple documents with one call.

Remove

`collection.remove(selector)`

Removes a document described by the *selector*, which must be an object containing the *_id* or *_key* attribute. There must be a document with that *_id* or *_key* in the current collection. This document is then removed.

The method returns a document with the attributes *_id*, *_key* and *_rev*. The attribute *_id* contains the document handle of the removed document, the attribute *_rev* contains the document revision of the removed document.

If the selector contains a *_rev* attribute, the method first checks that the specified revision is the current revision of that document. If not, there is a conflict, and an error is thrown.

`collection.remove(selector, options)`

As before, but *options* must be an object that can contain the following boolean attributes:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *overwrite*: If this flag is set to *true*, a *_rev* attribute in the selector is ignored.
- *returnOld*: If this flag is set to *true*, the complete previous revision of the document is returned in the output under the attribute *old*.
- *silent*: If this flag is set to *true*, no output is returned.

`collection.remove(document-handle)`

`collection.remove(document-handle, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision check is performed.

`collection.remove(document-key)`

`collection.remove(document-handle, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision check is performed.

`collection.remove(selectorarray)`

`collection.remove(selectorarray,options)`

These two variants allow to perform the operation on a whole array of selectors. The behavior is exactly as if *remove* would have been called on all members of the array and all results are returned in an array. If an error occurs with any of the documents, the operation stops immediately returning only an error object. The options behave exactly as before.

Examples

Remove a document:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> db.example.document(a1);
arangosh> db.example.remove(a1);
arangosh> db.example.document(a1);
```

show execution results
Remove a document with a conflict:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> a2 = db.example.replace(a1, { a : 2 });
arangosh> db.example.remove(a1);
arangosh> db.example.remove(a1, true);
arangosh> db.example.document(a1);
```

show execution results

Changes in 3.0 from 2.8:

The method now returns not only *true* but information about the removed document(s). The options *silent* and *returnOld* are new. The method can now remove multiple documents with one call.

Remove By Keys

`collection.removeByKeys(keys)`

Looks up the documents in the specified collection using the array of keys provided, and removes all documents from the collection whose keys are contained in the *keys* array. Keys for which no document can be found in the underlying collection are ignored, and no exception will be thrown for them.

The method will return an object containing the number of removed documents in the *removed* sub-attribute, and the number of not-removed/ignored documents in the *ignored* sub-attribute.

This method is deprecated in favour of the array variant of *remove*.

Examples

```
arangosh> keys = [ ];
arangosh> for (var i = 0; i < 10; ++i) {
........>   db.example.insert({ _key: "test" + i, value: i });
........>   keys.push("test" + i);
........> }
arangosh> db.example.removeByKeys(keys);
```

show execution results

Remove By Example

`collection.removeByExample(example)`

Removes all documents matching an example.

`collection.removeByExample(document, waitForSync)`

The optional *waitForSync* parameter can be used to force synchronization of the document deletion operation to disk even in case that the *waitForSync* flag had been disabled for the entire collection. Thus, the *waitForSync* parameter can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.

`collection.removeByExample(document, waitForSync, limit)`

The optional *limit* parameter can be used to restrict the number of removals to the specified value. If *limit* is specified but less than the number of documents in the collection, it is undefined which documents are removed.

Examples

```
arangosh> db.example.removeByExample( {Hello : "world"} );
1
```

Replace By Example

`collection.replaceByExample(example, newValue)`

Replaces all documents matching an example with a new document body. The entire document body of each document matching the *example* will be replaced with *newValue*. The document meta-attributes *_id*, *_key* and *_rev* will not be replaced.

`collection.replaceByExample(document, newValue, waitForSync)`

The optional *waitForSync* parameter can be used to force synchronization of the document replacement operation to disk even in case that the *waitForSync* flag had been disabled for the entire collection. Thus, the *waitForSync* parameter can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.

`collection.replaceByExample(document, newValue, waitForSync, limit)`

The optional *limit* parameter can be used to restrict the number of replacements to the specified value. If *limit* is specified but less than the number of documents in the collection, it is undefined which documents are replaced.

Examples

```
arangosh> db.example.save({ Hello : "world" });
arangosh> db.example.replaceByExample({ Hello: "world" }, {Hello: "mars"}, false, 5);
```

show execution results

Update By Example

`collection.updateByExample(example, newValue)`

Partially updates all documents matching an example with a new document body. Specific attributes in the document body of each document matching the *example* will be updated with the values from *newValue*. The document meta-attributes *_id*, *_key* and *_rev* cannot be updated.

Partial update could also be used to append new fields, if there were no old field with same name.

`collection.updateByExample(document, newValue, keepNull, waitForSync)`

The optional *keepNull* parameter can be used to modify the behavior when handling *null* values. Normally, *null* values are stored in the database. By setting the *keepNull* parameter to *false*, this behavior can be changed so that all attributes in *data* with *null* values will be removed from the target document.

The optional *waitForSync* parameter can be used to force synchronization of the document replacement operation to disk even in case that the *waitForSync* flag had been disabled for the entire collection. Thus, the *waitForSync* parameter can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.

`collection.updateByExample(document, newValue, keepNull, waitForSync, limit)`

The optional *limit* parameter can be used to restrict the number of updates to the specified value. If *limit* is specified but less than the number of documents in the collection, it is undefined which documents are updated.

`collection.updateByExample(document, newValue, options)`

Using this variant, the options for the operation can be passed using an object with the following sub-attributes:

- *keepNull*
- *waitForSync*
- *limit*
- *mergeObjects*

Examples

```
arangosh> db.example.save({ Hello : "world", foo : "bar" });
arangosh> db.example.updateByExample({ Hello: "world" }, { Hello: "foo", World: "bar" }, f
arangosh> db.example.byExample({ Hello: "foo" }).toArray()
```

show execution results

Collection type

`collection.type()`

Returns the type of a collection. Possible values are:

- 2: document collection
- 3: edge collection

Get the Version of ArangoDB

`db._version()`

Returns the server version string. Note that this is not the version of the database.

Examples

```
arangosh> require("@arangodb").db._version();
3.1.3
```

Edges

Edges are normal documents that always contain a `_from` and a `_to` attribute. Therefore, you can use the document methods to operate on edges. The following methods, however, are specific to edges.

`edge-collection.edges(vertex)`

The *edges* operator finds all edges starting from (outbound) or ending in (inbound) *vertex*.

`edge-collection.edges(vertices)`

The *edges* operator finds all edges starting from (outbound) or ending in (inbound) a document from *vertices*, which must a list of documents or document handles.

```
arangosh> db._create("vertex");
arangosh> db._createEdgeCollection("relation");
arangosh> myGraph.v1 = db.vertex.insert({ name : "vertex 1" });
arangosh> myGraph.v2 = db.vertex.insert({ name : "vertex 2" });
arangosh> myGraph.e1 = db.relation.insert(myGraph.v1, myGraph.v2,
........> { label : "knows"});
arangosh> db._document(myGraph.e1);
arangosh> db.relation.edges(myGraph.e1._id);
```

show execution results
`edge-collection.inEdges(vertex)`

The *edges* operator finds all edges ending in (inbound) *vertex*.

`edge-collection.inEdges(vertices)`

The *edges* operator finds all edges ending in (inbound) a document from *vertices*, which must a list of documents or document handles.

Examples

```
arangosh> db._create("vertex");
arangosh> db._createEdgeCollection("relation");
arangosh> myGraph.v1 = db.vertex.insert({ name : "vertex 1" });
arangosh> myGraph.v2 = db.vertex.insert({ name : "vertex 2" });
arangosh> myGraph.e1 = db.relation.insert(myGraph.v1, myGraph.v2,
........> { label : "knows"});
arangosh> db._document(myGraph.e1);
arangosh> db.relation.inEdges(myGraph.v1._id);
arangosh> db.relation.inEdges(myGraph.v2._id);
```

show execution results
`edge-collection.outEdges(vertex)`

The *edges* operator finds all edges starting from (outbound) *vertices*.

`edge-collection.outEdges(vertices)`

The *edges* operator finds all edges starting from (outbound) a document from *vertices*, which must a list of documents or document handles.

Examples

```
arangosh> db._create("vertex");
arangosh> db._createEdgeCollection("relation");
arangosh> myGraph.v1 = db.vertex.insert({ name : "vertex 1" });
arangosh> myGraph.v2 = db.vertex.insert({ name : "vertex 2" });
arangosh> myGraph.e1 = db.relation.insert(myGraph.v1, myGraph.v2,
........> { label : "knows"});
arangosh> db._document(myGraph.e1);
arangosh> db.relation.outEdges(myGraph.v1._id);
arangosh> db.relation.outEdges(myGraph.v2._id);
```

show execution results

Misc

`collection.iterate(iterator, options)`

Iterates over some elements of the collection and apply the function *iterator* to the elements. The function will be called with the document as first argument and the current number (starting with 0) as second argument.

options must be an object with the following attributes:

- *limit* (optional, default none): use at most *limit* documents.
- *probability* (optional, default all): a number between *0* and *1*. Documents are chosen with this probability.

Examples

```
arangosh> for (i = -90;  i <= 90;  i += 10) {
........>  for (j = -180;  j <= 180;  j += 10) {
........>    db.example.save({ name : "Name/" + i + "/" + j,
........>                      home : [ i, j ],
........>                      work : [ -i, -j ] });
........>  }
........> }
........>
arangosh> db.example.ensureIndex({ type: "geo", fields: [ "home" ] });
arangosh> items = db.example.getIndexes().map(function(x) { return x.id; });
........> db.example.index(items[1]);
```

show execution results

Database Methods

Document

`db._document(object)`

The *_document* method finds a document given an object *object* containing the *_id* attribute. The method returns the document if it can be found.

An error is thrown if *_rev* is specified but the document found has a different revision already. An error is also thrown if no document exists with the given *_id*.

Please note that if the method is executed on the arangod server (e.g. from inside a Foxx application), an immutable document object will be returned for performance reasons. It is not possible to change attributes of this immutable object. To update or patch the returned document, it needs to be cloned/copied into a regular JavaScript object first. This is not necessary if the *_document* method is called from out of arangosh or from any other client.

`db._document(document-handle)`

As before. Instead of *object* a *document-handle* can be passed as first argument. No revision can be specified in this case.

Examples

Returns the document:

```
arangosh> db._document("example/12345");
```

show execution results

Exists

`db._exists(object)`

The *_exists* method determines whether a document exists given an object `object` containing the *_id* attribute.

An error is thrown if *_rev* is specified but the document found has a different revision already.

Instead of returning the found document or an error, this method will only return an object with the attributes *_id*, *_key* and *_rev*, or *false* if no document with the given *_id* or *_key* exists. It can thus be used for easy existence checks.

This method will throw an error if used improperly, e.g. when called with a non-document handle, a non-document, or when a cross-collection request is performed.

`db._exists(document-handle)`

As before. Instead of *object* a *document-handle* can be passed as first argument.

Changes in 3.0 from 2.8:

In the case of a revision mismatch *_exists* now throws an error instead of simply returning *false*. This is to make it possible to tell the difference between a revision mismatch and a non-existing document.

Replace

`db._replace(selector, data)`

Replaces an existing document described by the *selector*, which must be an object containing the *_id* attribute. There must be a document with that *_id* in the current database. This document is then replaced with the *data* given as second argument. Any attribute *_id*, *_key* or *_rev* in *data* is ignored.

The method returns a document with the attributes *_id*, *_key*, *_rev* and *_oldRev*. The attribute *_id* contains the document handle of the updated document, the attribute *_rev* contains the document revision of the updated document, the attribute *_oldRev* contains the revision of the old (now replaced) document.

If the selector contains a *_rev* attribute, the method first checks that the specified revision is the current revision of that document. If not, there is a conflict, and an error is thrown.

`collection.replace(selector, data, options)`

As before, but *options* must be an object that can contain the following boolean attributes:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *overwrite*: If this flag is set to *true*, a *_rev* attribute in the selector is ignored.
- *returnNew*: If this flag is set to *true*, the complete new document is returned in the output under the attribute *new*.
- *returnOld*: If this flag is set to *true*, the complete previous revision of the document is returned in the output under the attribute *old*.
- *silent*: If this flag is set to *true*, no output is returned.

`db._replace(document-handle, data)`

`db._replace(document-handle, data, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision precondition is tested.

Examples

Create and replace a document:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> a2 = db._replace(a1, { a : 2 });
arangosh> a3 = db._replace(a1, { a : 3 });
```

show execution results

Changes in 3.0 from 2.8:

The options *silent*, *returnNew* and *returnOld* are new.

Update

`db._update(selector, data)`

Updates an existing document described by the *selector*, which must be an object containing the *_id* attribute. There must be a document with that *_id* in the current database. This document is then patched with the *data* given as second argument. Any attribute *_id*, *_key* or *_rev* in *data* is ignored.

The method returns a document with the attributes *_id*, *_key*, *_rev* and *_oldRev*. The attribute *_id* contains the document handle of the updated document, the attribute *_rev* contains the document revision of the updated document, the attribute *_oldRev* contains the revision of the old (now updated) document.

If the selector contains a *_rev* attribute, the method first checks that the specified revision is the current revision of that document. If not, there is a conflict, and an error is thrown.

`db._update(selector, data, options)`

As before, but *options* must be an object that can contain the following boolean attributes:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *overwrite*: If this flag is set to *true*, a *_rev* attribute in the selector is ignored.
- *returnNew*: If this flag is set to *true*, the complete new document is returned in the output under the attribute *new*.
- *returnOld*: If this flag is set to *true*, the complete previous revision of the document is returned in the output under the attribute *old*.

- *silent*: If this flag is set to *true*, no output is returned.
- *keepNull*: The optional *keepNull* parameter can be used to modify the behavior when handling *null* values. Normally, *null* values are stored in the database. By setting the *keepNull* parameter to *false*, this behavior can be changed so that all attributes in *data* with *null* values will be removed from the target document.
- *mergeObjects*: Controls whether objects (not arrays) will be merged if present in both the existing and the patch document. If set to *false*, the value in the patch document will overwrite the existing document's value. If set to *true*, objects will be merged. The default is *true*.

`db._update(document-handle, data)`

`db._update(document-handle, data, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision precondition is tested.

Examples

Create and update a document:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> a2 = db._update(a1, { b : 2 });
arangosh> a3 = db._update(a1, { c : 3 });
```

show execution results

Changes in 3.0 from 2.8:

The options *silent*, *returnNew* and *returnOld* are new.

Remove

`db._remove(selector)`

Removes a document described by the *selector*, which must be an object containing the *_id* attribute. There must be a document with that *_id* in the current database. This document is then removed.

The method returns a document with the attributes *_id*, *_key* and *_rev*. The attribute *_id* contains the document handle of the removed document, the attribute *_rev* contains the document revision of the removed eocument.

If the selector contains a *_rev* attribute, the method first checks that the specified revision is the current revision of that document. If not, there is a conflict, and an error is thrown.

`db._remove(selector, options)`

As before, but *options* must be an object that can contain the following boolean attributes:

- *waitForSync*: One can force synchronization of the document creation operation to disk even in case that the *waitForSync* flag is been disabled for the entire collection. Thus, the *waitForSync* option can be used to force synchronization of just specific operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* parameter cannot be used to disable synchronization for collections that have a default *waitForSync* value of *true*.
- *overwrite*: If this flag is set to *true*, a *_rev* attribute in the selector is ignored.
- *returnOld*: If this flag is set to *true*, the complete previous revision of the document is returned in the output under the attribute *old*.
- *silent*: If this flag is set to *true*, no output is returned.

`db._remove(document-handle)`

`db._remove(document-handle, options)`

As before. Instead of *selector* a *document-handle* can be passed as first argument. No revision check is performed.

Examples

Remove a document:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> db._remove(a1);
arangosh> db._remove(a1);
arangosh> db._remove(a1, {overwrite: true});
```

show execution results

Remove the document in the revision `a1` with a conflict:

```
arangosh> a1 = db.example.insert({ a : 1 });
arangosh> a2 = db._replace(a1, { a : 2 });
arangosh> db._remove(a1);
arangosh> db._remove(a1, {overwrite: true} );
arangosh> db._document(a1);
```

show execution results

Remove a document using new signature:

```
arangosh> db.example.insert({ _key: "11265325374", a:  1 } );
arangosh> db.example.remove("example/11265325374",
........> { overwrite: true, waitForSync: false})
```

show execution results

Changes in 3.0 from 2.8:

The method now returns not only *true* but information about the removed document(s). The options *silent* and *returnOld* are new.

Graphs, Vertices & Edges

Graphs, vertices & edges are defined in the Graphs chapter in details.

Related blog posts:

- Graphs in data modeling - is the emperor naked?
- Index Free Adjacency or Hybrid Indexes for Graph Databases

Naming Conventions in ArangoDB

The following naming conventions should be followed by users when creating databases, collections and documents in ArangoDB.

Database Names

ArangoDB will always start up with a default database, named *_system*. Users can create additional databases in ArangoDB, provided the database names conform to the following constraints:

- Database names must only consist of the letters *a* to *z* (both lower and upper case allowed), the numbers *0* to *9*, and the underscore (_) or dash (-) symbols This also means that any non-ASCII database names are not allowed
- Database names must always start with a letter. Database names starting with an underscore are considered to be system databases, and users should not create or delete those
- The maximum allowed length of a database name is 64 bytes
- Database names are case-sensitive

Collection Names

Users can pick names for their collections as desired, provided the following naming constraints are not violated:

- Collection names must only consist of the letters *a* to *z* (both in lower and upper case), the numbers *0* to *9*, and the underscore (_) or dash (-) symbols. This also means that any non-ASCII collection names are not allowed
- User-defined collection names must always start with a letter. System collection names must start with an underscore. All collection names starting with an underscore are considered to be system collections that are for ArangoDB's internal use only. System collection names should not be used by end users for their own collections
- The maximum allowed length of a collection name is 64 bytes
- Collection names are case-sensitive

Document Keys

Users can define their own keys for documents they save. The document key will be saved along with a document in the *_key* attribute. Users can pick key values as required, provided that the values conform to the following restrictions:

- The key must be a string value. Numeric keys are not allowed, but any numeric value can be put into a string and can then be used as document key.
- The key must be at least 1 byte and at most 254 bytes long. Empty keys are disallowed when specified (though it may be valid to completely omit the *_key* attribute from a document)
- It must consist of the letters a-z (lower or upper case), the digits 0-9 or any of the following punctuation characters: `_` `-` `:` `.` `@` `(` `)` `+` `,` `=` `;` `$` `!` `*` `'` `%`
- Any other characters, especially multi-byte UTF-8 sequences, whitespace or punctuation characters cannot be used inside key values
- The key must be unique within the collection it is used

Keys are case-sensitive, i.e. *myKey* and *MyKEY* are considered to be different keys.

Specifying a document key is optional when creating new documents. If no document key is specified by the user, ArangoDB will create the document key itself as each document is required to have a key.

There are no guarantees about the format and pattern of auto-generated document keys other than the above restrictions. Clients should therefore treat auto-generated document keys as opaque values and not rely on their format.

The current format for generated keys is a string containing numeric digits. The numeric values reflect chronological time in the sense that _key values generated later will contain higher numbers than _key values generated earlier. But the exact value that will be generated by the server is not predictable. Note that if you sort on the _key attribute, string comparison will be used, which means `"100"` is less than `"99"` etc.

Attribute Names

Users can pick attribute names for document attributes as desired, provided the following attribute naming constraints are not violated:

- Attribute names starting with an underscore are considered to be system attributes for ArangoDB's internal use. Such attribute names are already used by ArangoDB for special purposes:

  - *_id* is used to contain a document's handle
  - *_key* is used to contain a document's user-defined key
  - *_rev* is used to contain the document's revision number
  - In edge collections, the

    - *_from*
    - *_to*

    attributes are used to reference other documents.

  More system attributes may be added in the future without further notice so end users should try to avoid using their own attribute names starting with underscores.

- Theoretically, attribute names can include punctuation and special characters as desired, provided the name is a valid UTF-8 string. For maximum portability, special characters should be avoided though. For example, attribute names may contain the dot symbol, but the dot has a special meaning in JavaScript and also in AQL, so when using such attribute names in one of these languages, the attribute name needs to be quoted by the end user. Overall it might be better to use attribute names which don't require any quoting/escaping in all languages used. This includes languages used by the client (e.g. Ruby, PHP) if the attributes are mapped to object members there.

- Attribute names starting with an at-mark (@) will need to be enclosed in backticks when used in an AQL query to tell them apart from bind variables. Therefore we do not encourage the use of attributes starting with at-marks, though they will work when used properly.
- ArangoDB does not enforce a length limit for attribute names. However, long attribute names may use more memory in result sets etc. Therefore the use of long attribute names is discouraged.
- Attribute names are case-sensitive.
- Attributes with empty names (an empty string) are disallowed.

Handling Indexes

This is an introduction to ArangoDB's interface for indexes in general.
There are special sections for

- Index Basics: Introduction to all index types
- Which index to use when: Index type and options adviser
- Index Utilization: How ArangoDB uses indexes
- Working with Indexes: How to handle indexes programmatically using the `db` object
    - Hash Indexes
    - Skiplists
    - Persistent Indexes
    - Fulltext Indexes
    - Geo-spatial Indexes

Index basics

Indexes allow fast access to documents, provided the indexed attribute(s) are used in a query. While ArangoDB automatically indexes some system attributes, users are free to create extra indexes on non-system attributes of documents.

User-defined indexes can be created on collection level. Most user-defined indexes can be created by specifying the names of the index attributes. Some index types allow indexing just one attribute (e.g. fulltext index) whereas other index types allow indexing multiple attributes at the same time.

The system attributes `_id` , `_key` , `_from` and `_to` are automatically indexed by ArangoDB, without the user being required to create extra indexes for them. `_id` and `_key` are covered by a collection's primary key, and `_from` and `_to` are covered by an edge collection's edge index automatically.

Using the system attribute `_id` in user-defined indexes is not possible, but indexing `_key` , `_rev` , `_from` , and `_to` is.

ArangoDB provides the following index types:

Primary Index

For each collection there will always be a *primary index* which is a hash index for the document keys (`_key` attribute) of all documents in the collection. The primary index allows quick selection of documents in the collection using either the `_key` or `_id` attributes. It will be used from within AQL queries automatically when performing equality lookups on `_key` or `_id` .

There are also dedicated functions to find a document given its `_key` or `_id` that will always make use of the primary index:

```
db.collection.document("<document-key>");
db._document("<document-id>");
```

As the primary index is an unsorted hash index, it cannot be used for non-equality range queries or for sorting.

The primary index of a collection cannot be dropped or changed, and there is no mechanism to create user-defined primary indexes.

Edge Index

Every edge collection also has an automatically created *edge index*. The edge index provides quick access to documents by either their `_from` or `_to` attributes. It can therefore be used to quickly find connections between vertex documents and is invoked when the connecting edges of a vertex are queried.

Edge indexes are used from within AQL when performing equality lookups on `_from` or `_to` values in an edge collections. There are also dedicated functions to find edges given their `_from` or `_to` values that will always make use of the edge index:

```
db.collection.edges("<from-value>");
db.collection.edges("<to-value>");
db.collection.outEdges("<from-value>");
db.collection.outEdges("<to-value>");
db.collection.inEdges("<from-value>");
db.collection.inEdges("<to-value>");
```

Internally, the edge index is implemented as a hash index, which stores the union of all `_from` and `_to` attributes. It can be used for equality lookups, but not for range queries or for sorting. Edge indexes are automatically created for edge collections. It is not possible to create user-defined edge indexes. However, it is possible to freely use the `_from` and `_to` attributes in user-defined indexes.

An edge index cannot be dropped or changed.

Hash Index

A hash index can be used to quickly find documents with specific attribute values. The hash index is unsorted, so it supports equality lookups but no range queries or sorting.

A hash index can be created on one or multiple document attributes. A hash index will only be used by a query if all index attributes are present in the search condition, and if all attributes are compared using the equality (`==`) operator. Hash indexes are used from within AQL and several query functions, e.g. `byExample` , `firstExample` etc.

Hash indexes can optionally be declared unique, then disallowing saving the same value(s) in the indexed attribute(s). Hash indexes can optionally be sparse.

The different types of hash indexes have the following characteristics:

- **unique hash index**: all documents in the collection must have different values for the attributes covered by the unique index. Trying to insert a document with the same key value as an already existing document will lead to a unique constraint violation.

  This type of index is not sparse. Documents that do not contain the index attributes or that have a value of `null` in the index attribute(s) will still be indexed. A key value of `null` may only occur once in the index, so this type of index cannot be used for optional attributes.

  The unique option can also be used to ensure that no duplicate edges are created, by adding a combined index for the fields `_from` and `_to` to an edge collection.

- **unique, sparse hash index**: all documents in the collection must have different values for the attributes covered by the unique index. Documents in which at least one of the index attributes is not set or has a value of `null` are not included in the index. This type of index can be used to ensure that there are no duplicate keys in the collection for documents which have the indexed attributes set. As the index will exclude documents for which the indexed attributes are `null` or not set, it can be used for optional attributes.

- **non-unique hash index**: all documents in the collection will be indexed. This type of index is not sparse. Documents that do not contain the index attributes or that have a value of `null` in the index attribute(s) will still be indexed. Duplicate key values can occur and do not lead to unique constraint violations.

- **non-unique, sparse hash index**: only those documents will be indexed that have all the indexed attributes set to a value other than `null` . It can be used for optional attributes.

The amortized complexity of lookup, insert, update, and removal operations in unique hash indexes is O(1).

Non-unique hash indexes have an amortized complexity of O(1) for insert, update, and removal operations. That means non-unique hash indexes can be used on attributes with low cardinality.

If a hash index is created on an attribute that is missing in all or many of the documents, the behavior is as follows:

- if the index is sparse, the documents missing the attribute will not be indexed and not use index memory. These documents will not influence the update or removal performance for the index.

- if the index is non-sparse, the documents missing the attribute will be contained in the index with a key value of `null` .

Hash indexes support indexing array values if the index attribute name is extended with a *[*]*.

Skiplist Index

A skiplist is a sorted index structure. It can be used to quickly find documents with specific attribute values, for range queries and for returning documents from the index in sorted order. Skiplists will be used from within AQL and several query functions, e.g. `byExample` , `firstExample` etc.

Skiplist indexes will be used for lookups, range queries and sorting only if either all index attributes are provided in a query, or if a leftmost prefix of the index attributes is specified.

For example, if a skiplist index is created on attributes `value1` and `value2` , the following filter conditions can use the index (note: the `<=` and `>=` operators are intentionally omitted here for the sake of brevity):

```
FILTER doc.value1 == ...
FILTER doc.value1 < ...
FILTER doc.value1 > ...
FILTER doc.value1 > ... && doc.value1 < ...

FILTER doc.value1 == ... && doc.value2 == ...
FILTER doc.value1 == ... && doc.value2 > ...
FILTER doc.value1 == ... && doc.value2 > ... && doc.value2 < ...
```

In order to use a skiplist index for sorting, the index attributes must be specified in the `SORT` clause of the query in the same order as they appear in the index definition. Skiplist indexes are always created in ascending order, but they can be used to access the indexed elements in both ascending or descending order. However, for a combined index (an index on multiple attributes) this requires that the sort orders in a single query as specified in the `SORT` clause must be either all ascending (optionally ommitted as ascending is the default) or all descending.

For example, if the skiplist index is created on attributes `value1` and `value2` (in this order), then the following sorts clauses can use the index for sorting:

- `SORT value1 ASC, value2 ASC` (and its equivalent `SORT value1, value2`)
- `SORT value1 DESC, value2 DESC`
- `SORT value1 ASC` (and its equivalent `SORT value1`)
- `SORT value1 DESC`

The following sort clauses cannot make use of the index order, and require an extra sort step:

- `SORT value1 ASC, value2 DESC`
- `SORT value1 DESC, value2 ASC`
- `SORT value2` (and its equivalent `SORT value2 ASC`)
- `SORT value2 DESC` (because first indexed attribute `value1` is not used in sort clause)

Note: the latter two sort clauses cannot use the index because the sort clause does not refer to a leftmost prefix of the index attributes.

Skiplists can optionally be declared unique, disallowing saving the same value in the indexed attribute. They can be sparse or non-sparse.

The different types of skiplist indexes have the following characteristics:

- **unique skiplist index**: all documents in the collection must have different values for the attributes covered by the unique index. Trying to insert a document with the same key value as an already existing document will lead to a unique constraint violation.

  This type of index is not sparse. Documents that do not contain the index attributes or that have a value of `null` in the index attribute(s) will still be indexed. A key value of `null` may only occur once in the index, so this type of index cannot be used for optional attributes.

- **unique, sparse skiplist index**: all documents in the collection must have different values for the attributes covered by the unique index. Documents in which at least one of the index attributes is not set or has a value of `null` are not included in the index. This type of index can be used to ensure that there are no duplicate keys in the collection for documents which have the indexed attributes set. As the index will exclude documents for which the indexed attributes are `null` or not set, it can be used for optional attributes.

- **non-unique skiplist index**: all documents in the collection will be indexed. This type of index is not sparse. Documents that do not contain the index attributes or that have a value of `null` in the index attribute(s) will still be indexed. Duplicate key values can occur and do not lead to unique constraint violations.

- **non-unique, sparse skiplist index**: only those documents will be indexed that have all the indexed attributes set to a value other than `null` . It can be used for optional attributes.

The operational amortized complexity for skiplist indexes is logarithmically correlated with the number of documents in the index.

Skiplist indexes support indexing array values if the index attribute name is extended with a *[*]*`.

Persistent Index

The persistent index is a sorted index with persistence. The index entries are written to disk when documents are stored or updated. That means the index entries do not need to be rebuilt from the collection data when the server is restarted or the indexed collection is initially loaded. Thus using persistent indexes may reduce collection loading times.

The persistent index type can be used for secondary indexes at the moment. That means the persistent index currently cannot be made the only index for a collection, because there will always be the in-memory primary index for the collection in addition, and potentially more indexes (such as the edges index for an edge collection).

The index implementation is using the RocksDB engine, and it provides logarithmic complexity for insert, update, and remove operations. As the persistent index is not an in-memory index, it does not store pointers into the primary index as all the in-memory indexes do, but instead it stores a document's primary key. To retrieve a document via a persistent index via an index value lookup, there will therefore be an additional O(1) lookup into the primary index to fetch the actual document.

As the persistent index is sorted, it can be used for point lookups, range queries and sorting operations, but only if either all index attributes are provided in a query, or if a leftmost prefix of the index attributes is specified.

Geo Index

Users can create additional geo indexes on one or multiple attributes in collections. A geo index is used to find places on the surface of the earth fast.

The geo index stores two-dimensional coordinates. It can be created on either two separate document attributes (latitude and longitude) or a single array attribute that contains both latitude and longitude. Latitude and longitude must be numeric values.

Th geo index provides operations to find documents with coordinates nearest to a given comparison coordinate, and to find documents with coordinates that are within a specifiable radius around a comparison coordinate.

The geo index is used via dedicated functions in AQL or the simple queries functions, but will not be used for other types of queries or conditions.

Fulltext Index

A fulltext index can be used to find words, or prefixes of words inside documents. A fulltext index can be created on a single attribute only, and will index all words contained in documents that have a textual value in that attribute. Only words with a (specifiable) minimum length are indexed. Word tokenization is done using the word boundary analysis provided by libicu, which is taking into account the selected language provided at server start. Words are indexed in their lower-cased form. The index supports complete match queries (full words) and prefix queries, plus basic logical operations such as `and` , `or` and `not` for combining partial results.

The fulltext index is sparse, meaning it will only index documents for which the index attribute is set and contains a string value. Additionally, only words with a configurable minimum length will be included in the index.

The fulltext index is used via dedicated functions in AQL or the simple queries, but will not be enabled for other types of queries or conditions.

Indexing attributes and sub-attributes

Top-level as well as nested attributes can be indexed. For attributes at the top level, the attribute names alone are required. To index a single field, pass an array with a single element (string of the attribute key) to the *fields* parameter of the ensureIndex() method. To create a combined index over multiple fields, simply add more members to the *fields* array:

```
// { name: "Smith", age: 35 }
db.posts.ensureIndex({ type: "hash", fields: [ "name" ] })
db.posts.ensureIndex({ type: "hash", fields: [ "name", "age" ] })
```

To index sub-attributes, specify the attribute path using the dot notation:

```
// { name: {last: "Smith", first: "John" } }
db.posts.ensureIndex({ type: "hash", fields: [ "name.last" ] })
db.posts.ensureIndex({ type: "hash", fields: [ "name.last", "name.first" ] })
```

Indexing array values

If an index attribute contains an array, ArangoDB will store the entire array as the index value by default. Accessing individual members of the array via the index is not possible this way.

To make an index insert the individual array members into the index instead of the entire array value, a special array index needs to be created for the attribute. Array indexes can be set up like regular hash or skiplist indexes using the `collection.ensureIndex()` function. To make a hash or skiplist index an array index, the index attribute name needs to be extended with *[*]* when creating the index and when filtering in an AQL query using the `IN` operator.

The following example creates an array hash index on the `tags` attribute in a collection named `posts` :

```
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*]" ] });
db.posts.insert({ tags: [ "foobar", "baz", "quux" ] });
```

This array index can then be used for looking up individual `tags` values from AQL queries via the `IN` operator:

```
FOR doc IN posts
  FILTER 'foobar' IN doc.tags
  RETURN doc
```

It is possible to add the array expansion operator *[*]*, but it is not mandatory. You may use it to indicate that an array index is used, it is purely cosmetic however:

```
FOR doc IN posts
  FILTER 'foobar' IN doc.tags[*]
  RETURN doc
```

The following FILTER conditions will **not use** the array index:

```
FILTER doc.tags ANY == 'foobar'
FILTER doc.tags ANY IN 'foobar'
FILTER doc.tags IN 'foobar'
FILTER doc.tags == 'foobar'
FILTER 'foobar' == doc.tags
```

It is also possible to create an index on subattributes of array values. This makes sense if the index attribute is an array of objects, e.g.

```
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*].name" ] });
db.posts.insert({ tags: [ { name: "foobar" }, { name: "baz" }, { name: "quux" } ] });
```

The following query will then use the array index (this does require the array expansion operator):

```
FOR doc IN posts
  FILTER 'foobar' IN doc.tags[*].name
  RETURN doc
```

If you store a document having the array which does contain elements not having the subattributes this document will also be indexed with the value `null`, which in ArangoDB is equal to attribute not existing.

ArangoDB supports creating array indexes with a single *[*]* operator per index attribute. For example, creating an index as follows is **not supported**:

```
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*].name[*].value" ] });
```

Array values will automatically be de-duplicated before being inserted into an array index. For example, if the following document is inserted into the collection, the duplicate array value `bar` will be inserted only once:

```
db.posts.insert({ tags: [ "foobar", "bar", "bar" ] });
```

If an array index is declared **unique**, the de-duplication of array values will happen before inserting the values into the index, so the above insert operation will not necessarily fail. It will fail if the index already contains an instance of the `bar` value, but will succeed if the value `bar` is not already present in the index.

If an array index is declared and you store documents that do not have an array at the specified attribute this document will not be inserted in the index. Hence the following objects will not be indexed:

```
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*]" ] });
db.posts.insert({ something: "else" });
db.posts.insert({ tags: null });
db.posts.insert({ tags: "this is no array" });
db.posts.insert({ tags: { content: [1, 2, 3] } });
```

An array index is able to index an explicit `null` value and when queried for it, it will only return those documents having explicitly `null` stored in the array, it will not return any documents that do not have the array at all.

```
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*]" ] });
db.posts.insert({tags: null}) // Will not be indexed
db.posts.insert({tags: []})  // Will not be indexed
db.posts.insert({tags: [null]}); // Will be indexed for null
db.posts.insert({tags: [null, 1, 2]}); // Will be indexed for null, 1 and 2
```

Declaring an array index as **sparse** does not have an effect on the array part of the index, this in particular means that explicit `null` values are also indexed in the **sparse** version. If an index is combined from an array and a normal attribute the sparsity will apply for the attribute e.g.:

```
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*]", "name" ], sparse: true });
db.posts.insert({tags: null, name: "alice"}) // Will not be indexed
db.posts.insert({tags: [], name: "alice"}) // Will not be indexed
db.posts.insert({tags: [1, 2, 3]}) // Will not be indexed
db.posts.insert({tags: [1, 2, 3], name: null}) // Will not be indexed
db.posts.insert({tags: [1, 2, 3], name: "alice"})
// Will be indexed for [1, "alice"], [2, "alice"], [3, "alice"]
db.posts.insert({tags: [null], name: "bob"})
// Will be indexed for [null, "bob"]
```

Please note that filtering using array indexes only works from within AQL queries and only if the query filters on the indexed attribute using the `IN` operator. The other comparison operators (`==` , `!=` , `>` , `>=` , `<` , `<=` , `ANY` , `ALL` , `NONE`) currently cannot use array indexes.

Vertex centric indexes

As mentioned above, the most important indexes for graphs are the edge indexes, indexing the `_from` and `_to` attributes of edge collections. They provide very quick access to all edges originating in or arriving at a given vertex, which allows to quickly find all neighbours of a vertex in a graph.

In many cases one would like to run more specific queries, for example finding amongst the edges originating in a given vertex only those with the 20 latest time stamps. Exactly this is achieved with "vertex centric indexes". In a sense these are localized indexes for an edge collection, which sit at every single vertex.

Technically, they are implemented in ArangoDB as indexes, which sort the complete edge collection first by `_from` and then by other attributes. If we for example have a skiplist index on the attributes `_from` and `timestamp` of an edge collection, we can answer the above question very quickly with a single range lookup in the index.

Since ArangoDB 3.0 one can create sorted indexes (type "skiplist" and "persistent") that index the special edge attributes `_from` or `_to` and additionally other attributes. Since ArangoDB 3.1, these are used in graph traversals, when appropriate `FILTER` statements are found by the optimizer.

For example, to create a vertex centric index of the above type, you would simply do

```
db.edges.ensureIndex({"type":"skiplist", "fields": ["_from", "timestamp"]});
```

Then, queries like

```
FOR v, e, p IN 1..1 OUTBOUND "V/1" edges
  FILTER e.timestamp >= "2016-11-09"
  RETURN p
```

will be considerably faster in case there are many edges originating in vertex `"V/1"` but only few with a recent time stamp.

Which Index to use when

ArangoDB automatically indexes the `_key` attribute in each collection. There is no need to index this attribute separately. Please note that a document's `_id` attribute is derived from the `_key` attribute, and is thus implicitly indexed, too.

ArangoDB will also automatically create an index on `_from` and `_to` in any edge collection, meaning incoming and outgoing connections can be determined efficiently.

Index types

Users can define additional indexes on one or multiple document attributes. Several different index types are provided by ArangoDB. These indexes have different usage scenarios:

- hash index: provides quick access to individual documents if (and only if) all indexed attributes are provided in the search query. The index will only be used for equality comparisons. It does not support range queries and cannot be used for sorting.

  The hash index is a good candidate if all or most queries on the indexed attribute(s) are equality comparisons. The unique hash index provides an amortized complexity of O(1) for insert, update, remove and lookup operations. The non-unique hash index provides O(1) inserts, updates and removes, and will allow looking up documents by index value with amortized O(n) complexity, with *n* being the number of documents with that index value.

  A non-unique hash index on an optional document attribute should be declared sparse so that it will not index documents for which the index attribute is not set.

- skiplist index: skiplists keep the indexed values in an order, so they can be used for equality lookups, range queries and for sorting. For high selectivity attributes, skiplist indexes will have a higher overhead than hash indexes. For low selectivity attributes, skiplist indexes will be more efficient than non-unique hash indexes.

  Additionally, skiplist indexes allow more use cases (e.g. range queries, sorting) than hash indexes. Furthermore, they can be used for lookups based on a leftmost prefix of the index attributes.

- persistent index: a persistent index behaves much like the sorted skiplist index, except that all index values are persisted on disk and do not need to be rebuilt in memory when the server is restarted or the indexed collection is reloaded. The operations in a persistent index have logarithmic complexity, but operations have may have a higher constant factor than the operations in a skiplist index, because the persistent index may need to make extra roundtrips to the primary index to fetch the actual documents.

  A persistent index can be used for equality lookups, range queries and for sorting. For high selectivity attributes, persistent indexes will have a higher overhead than skiplist or hash indexes.

  Persistent indexes allow more use cases (e.g. range queries, sorting) than hash indexes. Furthermore, they can be used for lookups based on a leftmost prefix of the index attributes. In contrast to the in-memory skiplist indexes, persistent indexes do not need to be rebuilt in-memory so they don't influence the loading time of collections as other in-memory indexes do.

- geo index: the geo index provided by ArangoDB allows searching for documents within a radius around a two-dimensional earth coordinate (point), or to find documents with are closest to a point. Document coordinates can either be specified in two different document attributes or in a single attribute, e.g.

  ```
  { "latitude": 50.9406645, "longitude": 6.9599115 }
  ```

  or

  ```
  { "coords": [ 50.9406645, 6.9599115 ] }
  ```

  Geo indexes will only be invoked via special functions.

- fulltext index: a fulltext index can be used to index all words contained in a specific attribute of all documents in a collection. Only words with a (specifiable) minimum length are indexed. Word tokenization is done using the word boundary analysis provided by libicu, which is taking into account the selected language provided at server start.

The index supports complete match queries (full words) and prefix queries. Fulltext indexes will only be invoked via special functions.

Sparse vs. non-sparse indexes

Hash indexes and skiplist indexes can optionally be created sparse. A sparse index does not contain documents for which at least one of the index attribute is not set or contains a value of `null` .

As such documents are excluded from sparse indexes, they may contain fewer documents than their non-sparse counterparts. This enables faster indexing and can lead to reduced memory usage in case the indexed attribute does occur only in some, but not all documents of the collection. Sparse indexes will also reduce the number of collisions in non-unique hash indexes in case non-existing or optional attributes are indexed.

In order to create a sparse index, an object with the attribute `sparse` can be added to the index creation commands:

```
db.collection.ensureIndex({ type: "hash", fields: [ "attributeName" ], sparse: true });
db.collection.ensureIndex({ type: "hash", fields: [ "attributeName1", "attributeName2" ], sparse: true });
db.collection.ensureIndex({ type: "hash", fields: [ "attributeName" ], unique: true, sparse: true });
db.collection.ensureIndex({ type: "hash", fields: [ "attributeName1", "attributeName2" ], unique: true, sparse: true });

db.collection.ensureIndex({ type: "skiplist", fields: [ "attributeName" ], sparse: true });
db.collection.ensureIndex({ type: "skiplist", fields: [ "attributeName1", "attributeName2" ], sparse: true });
db.collection.ensureIndex({ type: "skiplist", fields: [ "attributeName" ], unique: true, sparse: true });
db.collection.ensureIndex({ type: "skiplist", fields: [ "attributeName1", "attributeName2" ], unique: true, sparse: true
```

When not explicitly set, the `sparse` attribute defaults to `false` for new indexes. Other indexes than hash and skiplist do not support sparsity.

As sparse indexes may exclude some documents from the collection, they cannot be used for all types of queries. Sparse hash indexes cannot be used to find documents for which at least one of the indexed attributes has a value of `null` . For example, the following AQL query cannot use a sparse index, even if one was created on attribute `attr` :

```
FOR doc In collection
  FILTER doc.attr == null
  RETURN doc
```

If the lookup value is non-constant, a sparse index may or may not be used, depending on the other types of conditions in the query. If the optimizer can safely determine that the lookup value cannot be `null` , a sparse index may be used. When uncertain, the optimizer will not make use of a sparse index in a query in order to produce correct results.

For example, the following queries cannot use a sparse index on `attr` because the optimizer will not know beforehand whether the values which are compared to `doc.attr` will include `null` :

```
FOR doc In collection
  FILTER doc.attr == SOME_FUNCTION(...)
  RETURN doc

FOR other IN otherCollection
  FOR doc In collection
    FILTER doc.attr == other.attr
    RETURN doc
```

Sparse skiplist indexes can be used for sorting if the optimizer can safely detect that the index range does not include `null` for any of the index attributes.

Note that if you intend to use joins it may be clever to use non-sparsity and maybe even uniqueness for that attribute, else all items containing the `null` value will match against each other and thus produce large results.

Index Utilization

In most cases ArangoDB will use a single index per collection in a given query. AQL queries can use more than one index per collection when multiple FILTER conditions are combined with a logical `OR` and these can be covered by indexes. AQL queries will use a single index per collection when FILTER conditions are combined with logical `AND` .

Creating multiple indexes on different attributes of the same collection may give the query optimizer more choices when picking an index. Creating multiple indexes on different attributes can also help in speeding up different queries, with FILTER conditions on different attributes.

It is often beneficial to create an index on more than just one attribute. By adding more attributes to an index, an index can become more selective and thus reduce the number of documents that queries need to process.

ArangoDB's primary indexes, edges indexes and hash indexes will automatically provide selectivity estimates. Index selectivity estimates are provided in the web interface, the `getIndexes()` return value and in the `explain()` output for a given query.

The more selective an index is, the more documents it will filter on average. The index selectivity estimates are therefore used by the optimizer when creating query execution plans when there are multiple indexes the optimizer can choose from. The optimizer will then select a combination of indexes with the lowest estimated total cost. In general, the optimizer will pick the indexes with the highest estimated selectivity.

Sparse indexes may or may not be picked by the optimizer in a query. As sparse indexes do not contain `null` values, they will not be used for queries if the optimizer cannot safely determine whether a FILTER condition includes `null` values for the index attributes. The optimizer policy is to produce correct results, regardless of whether or which index is used to satisfy FILTER conditions. If it is unsure about whether using an index will violate the policy, it will not make use of the index.

Troubleshooting

When in doubt about whether and which indexes will be used for executing a given AQL query, click the *Explain* button in the web interface in the *Queries* view or use the `explain()` method for the statement as follows (from the ArangoShell):

```
var query = "FOR doc IN collection FILTER doc.value > 42 RETURN doc";
var stmt = db._createStatement(query);
stmt.explain();
```

The `explain()` command will return a detailed JSON representation of the query's execution plan. The JSON explain output is intended to be used by code. To get a human-readable and much more compact explanation of the query, there is an explainer tool:

```
var query = "FOR doc IN collection FILTER doc.value > 42 RETURN doc";
require("@arangodb/aql/explainer").explain(query);
```

If any of the explain methods shows that a query is not using indexes, the following steps may help:

- check if the attribute names in the query are correctly spelled. In a schema-free database, documents in the same collection can have varying structures. There is no such thing as a *non-existing attribute* error. A query that refers to attribute names not present in any of the documents will not return an error, and obviously will not benefit from indexes.

- check the return value of the `getIndexes()` method for the collections used in the query and validate that indexes are actually present on the attributes used in the query's filter conditions.

- if indexes are present but not used by the query, the indexes may have the wrong type. For example, a hash index will only be used for equality comparisons (i.e. `==`) but not for other comparison types such as `<` , `<=` , `>` , `>=` . Additionally hash indexes will only be used if all of the index attributes are used in the query's FILTER conditions. A skiplist index will only be used if at least its first attribute is used in a FILTER condition. If additionally of the skiplist index attributes are specified in the query (from left-to-right), they may also be used and allow to filter more documents.

- using indexed attributes as function parameters or in arbitrary expressions will likely lead to the index on the attribute not being used. For example, the following queries will not use an index on `value` :

```
FOR doc IN collection FILTER TO_NUMBER(doc.value) == 42 RETURN doc
FOR doc IN collection FILTER doc.value - 1 == 42 RETURN doc
```

In these cases the queries should be rewritten so that only the index attribute is present on one side of the operator, or additional filters and indexes should be used to restrict the amount of documents otherwise.

- certain AQL functions such as `WITHIN()` or `FULLTEXT()` do utilize indexes internally, but their use is not mentioned in the query explanation for functions in general. These functions will raise query errors (at runtime) if no suitable index is present for the collection in question.
- the query optimizer will in general pick one index per collection in a query. It can pick more than one index per collection if the FILTER condition contains multiple branches combined with logical `OR` . For example, the following queries can use indexes:

```
FOR doc IN collection FILTER doc.value1 == 42 || doc.value1 == 23 RETURN doc
FOR doc IN collection FILTER doc.value1 == 42 || doc.value2 == 23 RETURN doc
FOR doc IN collection FILTER doc.value1 < 42 || doc.value2 > 23 RETURN doc
```

The two `OR`s in the first query will be converted to an `IN` list, and if there is a suitable index on `value1` , it will be used. The second query requires two separate indexes on `value1` and `value2` and will use them if present. The third query can use the indexes on `value1` and `value2` when they are sorted.

Working with Indexes

Index Identifiers and Handles

An *index handle* uniquely identifies an index in the database. It is a string and consists of the collection name and an *index identifier* separated by a `/`. The index identifier part is a numeric value that is auto-generated by ArangoDB.

A specific index of a collection can be accessed using its *index handle* or *index identifier* as follows:

```
db.collection.index("<index-handle>");
db.collection.index("<index-identifier>");
db._index("<index-handle>");
```

For example: Assume that the index handle, which is stored in the `_id` attribute of the index, is `demo/362549736` and the index was created in a collection named `demo`. Then this index can be accessed as:

```
db.demo.index("demo/362549736");
```

Because the index handle is unique within the database, you can leave out the *collection* and use the shortcut:

```
db._index("demo/362549736");
```

Collection Methods

Listing all indexes of a collection

returns information about the indexes `getIndexes()`

Returns an array of all indexes defined for the collection.

Note that `_key` implicitly has an index assigned to it.

```
arangosh> db.test.ensureHashIndex("hashListAttribute",
........> "hashListSecondAttribute.subAttribute");
arangosh> db.test.getIndexes();
```

show execution results

Creating an index

Indexes should be created using the general method *ensureIndex*. This method obsoletes the specialized index-specific methods *ensureHashIndex*, *ensureSkiplist*, *ensureUniqueConstraint* etc.

ensures that an index exists `collection.ensureIndex(index-description)`

Ensures that an index according to the *index-description* exists. A new index will be created if none exists with the given description.

The *index-description* must contain at least a *type* attribute. Other attributes may be necessary, depending on the index type.

type can be one of the following values:

- *hash*: hash index
- *skiplist*: skiplist index
- *fulltext*: fulltext index
- *geo1*: geo index, with one attribute
- *geo2*: geo index, with two attributes

sparse can be *true* or *false*.

For *hash*, and *skiplist* the sparsity can be controlled, *fulltext* and *geo* are sparse by definition.

unique can be *true* or *false* and is supported by *hash* or *skiplist*

Calling this method returns an index object. Whether or not the index object existed before the call is indicated in the return attribute *isNewlyCreated*.

Examples

```
arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a" ], sparse: true });
arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a", "b" ], unique: true });
```

show execution results

Dropping an index

drops an index `collection.dropIndex(index)`

Drops the index. If the index does not exist, then *false* is returned. If the index existed and was dropped, then *true* is returned. Note that you cannot drop some special indexes (e.g. the primary index of a collection or the edge index of an edge collection).

`collection.dropIndex(index-handle)`

Same as above. Instead of an index an index handle can be given.

```
arangosh> db.example.ensureSkiplist("a", "b");
arangosh> var indexInfo = db.example.getIndexes();
arangosh> indexInfo;
arangosh> db.example.dropIndex(indexInfo[0])
arangosh> db.example.dropIndex(indexInfo[1].id)
arangosh> indexInfo = db.example.getIndexes();
```

show execution results

Database Methods

Fetching an index by handle

finds an index `db._index(index-handle)`

Returns the index with *index-handle* or null if no such index exists.

```
arangosh> db.example.ensureIndex({ type: "skiplist", fields: [ "a", "b" ] });
arangosh> var indexInfo = db.example.getIndexes().map(function(x) { return x.id; });
arangosh> indexInfo;
arangosh> db._index(indexInfo[0])
arangosh> db._index(indexInfo[1])
```

show execution results

Dropping an index

drops an index `db._dropIndex(index)`

Drops the *index*. If the index does not exist, then *false* is returned. If the index existed and was dropped, then *true* is returned.

`db._dropIndex(index-handle)`

Drops the index with *index-handle*.

```
arangosh> db.example.ensureIndex({ type: "skiplist", fields: [ "a", "b" ] });
arangosh> var indexInfo = db.example.getIndexes();
arangosh> indexInfo;
arangosh> db._dropIndex(indexInfo[0])
arangosh> db._dropIndex(indexInfo[1].id)
arangosh> indexInfo = db.example.getIndexes();
```

show execution results

Revalidating whether an index is used

finds an index

So you've created an index, and since its maintainance isn't for free, you definitely want to know whether your query can utilize it.

You can use explain to verify whether **skiplists** or **hash indexes** are used (if you omit `colors: false` you will get nice colors in ArangoShell):

```
arangosh> var explain = require("@arangodb/aql/explainer").explain;
arangosh> db.example.ensureIndex({ type: "skiplist", fields: [ "a", "b" ] });
arangosh> explain("FOR doc IN example FILTER doc.a < 23 RETURN doc", {colors:false});
```

show execution results

Hash Indexes

Introduction to Hash Indexes

It is possible to define a hash index on one or more attributes (or paths) of a document. This hash index is then used in queries to locate documents in O(1) operations. If the hash index is unique, then no two documents are allowed to have the same set of attribute values.

Creating a new document or updating a document will fail if the uniqueness is violated. If the index is declared sparse, a document will be excluded from the index and no uniqueness checks will be performed if any index attribute value is not set or has a value of `null` .

Accessing Hash Indexes from the Shell

Unique Hash Indexes

Ensures that a unique constraint exists: `collection.ensureIndex({ type: "hash", fields: [ "field1", ..., "fieldn" ], unique: true })`

Creates a unique hash index on all documents using *field1*, ... *fieldn* as attribute paths. At least one attribute path has to be given. The index will be non-sparse by default.

All documents in the collection must differ in terms of the indexed attributes. Creating a new document or updating an existing document will will fail if the attribute uniqueness is violated.

To create a sparse unique index, set the *sparse* attribute to `true` :

`collection.ensureIndex({ type: "hash", fields: [ "field1", ..., "fieldn" ], unique: true, sparse: true })`

In case that the index was successfully created, the index identifier is returned.

Non-existing attributes will default to `null` . In a sparse index all documents will be excluded from the index for which all specified index attributes are `null` . Such documents will not be taken into account for uniqueness checks.

In a non-sparse index, **all** documents regardless of `null` - attributes will be indexed and will be taken into account for uniqueness checks.

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a", "b.c" ], unique: true });
arangosh> db.test.save({ a : 1, b : { c : 1 } });
arangosh> db.test.save({ a : 1, b : { c : 1 } });
arangosh> db.test.save({ a : 1, b : { c : null } });
arangosh> db.test.save({ a : 1 });
```

show execution results

Non-unique Hash Indexes

Ensures that a non-unique hash index exists: `collection.ensureIndex({ type: "hash", fields: [ "field1", ..., "fieldn" ] })`

Creates a non-unique hash index on all documents using *field1*, ... *fieldn* as attribute paths. At least one attribute path has to be given. The index will be non-sparse by default.

To create a sparse unique index, set the *sparse* attribute to `true` :

`collection.ensureIndex({ type: "hash", fields: [ "field1", ..., "fieldn" ], sparse: true })`

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a" ] });
arangosh> db.test.save({ a : 1 });
arangosh> db.test.save({ a : 1 });
arangosh> db.test.save({ a : null });
```

show execution results

Hash Array Indexes

Ensures that a hash array index exists (non-unique): `collection.ensureIndex({ type: "hash", fields: [ "field1[*]", ..., "fieldn[*]" ] })`

Creates a non-unique hash array index for the individual elements of the array attributes *field1[*]*, ... *fieldn[*]* found in the documents. At least one attribute path has to be given. The index always treats the indexed arrays as sparse.

It is possible to combine array indexing with standard indexing: `collection.ensureIndex({ type: "hash", fields: [ "field1[*]", "field2" ] })`

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a[*]" ] });
arangosh> db.test.save({ a : [ 1, 2 ] });
arangosh> db.test.save({ a : [ 1, 3 ] });
arangosh> db.test.save({ a : null });
```

show execution results

Ensure uniqueness of relations in edge collections

It is possible to create secondary indexes using the edge attributes `_from` and `_to`, starting with ArangoDB 3.0. A combined index over both fields together with the unique option enabled can be used to prevent duplicate relations from being created.

For example, a document collection *verts* might contain vertices with the document handles `verts/A`, `verts/B` and `verts/C`. Relations between these documents can be stored in an edge collection *edges* for instance. Now, you may want to make sure that the vertex `verts/A` is never linked to `verts/B` by an edge more than once. This can be achieved by adding a unique, non-sparse hash index for the fields `_from` and `_to`:

```
db.edges.ensureIndex({ type: "hash", fields: [ "_from", "_to" ], unique: true });
```

Creating an edge `{ _from: "verts/A", _to: "verts/B" }` in *edges* will be accepted, but only once. Another attempt to store an edge with the relation **A** → **B** will be rejected by the server with a *unique constraint violated* error. This includes updates to the `_from` and `_to` fields.

Note that adding a relation **B** → **A** is still possible, so is **A** → **A** and **B** → **B**, because they are all different relations in a directed graph. Each one can only occur once however.

Skiplists

Introduction to Skiplist Indexes

This is an introduction to ArangoDB's skiplists.

It is possible to define a skiplist index on one or more attributes (or paths) of documents. This skiplist is then used in queries to locate documents within a given range. If the skiplist is declared unique, then no two documents are allowed to have the same set of attribute values.

Creating a new document or updating a document will fail if the uniqueness is violated. If the skiplist index is declared sparse, a document will be excluded from the index and no uniqueness checks will be performed if any index attribute value is not set or has a value of `null` .

Accessing Skiplist Indexes from the Shell

Unique Skiplist Index

Ensures that a unique skiplist index exists: `collection.ensureIndex({ type: "skiplist", fields: [ "field1", ..., "fieldn" ], unique: true })`

Creates a unique skiplist index on all documents using *field1*, ... *fieldn* as attribute paths. At least one attribute path has to be given. The index will be non-sparse by default.

All documents in the collection must differ in terms of the indexed attributes. Creating a new document or updating an existing document will will fail if the attribute uniqueness is violated.

To create a sparse unique index, set the *sparse* attribute to `true` :

`collection.ensureIndex({ type: "skiplist", fields: [ "field1", ..., "fieldn" ], unique: true, sparse: true })`

In a sparse index all documents will be excluded from the index that do not contain at least one of the specified index attributes or that have a value of `null` in any of the specified index attributes. Such documents will not be indexed, and not be taken into account for uniqueness checks.

In a non-sparse index, these documents will be indexed (for non-present indexed attributes, a value of `null` will be used) and will be taken into account for uniqueness checks.

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.ids.ensureIndex({ type: "skiplist", fields: [ "myId" ], unique: true });
arangosh> db.ids.save({ "myId": 123 });
arangosh> db.ids.save({ "myId": 456 });
arangosh> db.ids.save({ "myId": 789 });
arangosh> db.ids.save({ "myId": 123 });
```

show execution results

```
arangosh> db.ids.ensureIndex({ type: "skiplist", fields: [ "name.first", "name.last" ], un
arangosh> db.ids.save({ "name" : { "first" : "hans", "last": "hansen" }});
arangosh> db.ids.save({ "name" : { "first" : "jens", "last": "jensen" }});
arangosh> db.ids.save({ "name" : { "first" : "hans", "last": "jensen" }});
arangosh> db.ids.save({ "name" : { "first" : "hans", "last": "hansen" }});
```

show execution results

Non-unique Skiplist Index

Ensures that a non-unique skiplist index exists: `collection.ensureIndex({ type: "skiplist", fields: [ "field1", ..., "fieldn" ] })`

Creates a non-unique skiplist index on all documents using *field1*, ... *fieldn* as attribute paths. At least one attribute path has to be given. The index will be non-sparse by default.

To create a sparse unique index, set the *sparse* attribute to `true`.

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.names.ensureIndex({ type: "skiplist", fields: [ "first" ] });
arangosh> db.names.save({ "first" : "Tim" });
arangosh> db.names.save({ "first" : "Tom" });
arangosh> db.names.save({ "first" : "John" });
arangosh> db.names.save({ "first" : "Tim" });
arangosh> db.names.save({ "first" : "Tom" });
```

show execution results

Skiplist Array Index

Ensures that a skiplist array index exists (non-unique): `collection.ensureIndex({ type: "skiplist", fields: [ "field1[*]", ..., "fieldn[*]" ] })`

Creates a non-unique skiplist array index for the individual elements of the array attributes *field1[*]*, ... *fieldn[*]* found in the documents. At least one attribute path has to be given. The index always treats the indexed arrays as sparse.

It is possible to combine array indexing with standard indexing: `collection.ensureIndex({ type: "skiplist", fields: [ "field1[*]", "field2" ] })`

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.test.ensureIndex({ type: "skiplist", fields: [ "a[*]" ] });
arangosh> db.test.save({ a : [ 1, 2 ] });
arangosh> db.test.save({ a : [ 1, 3 ] });
arangosh> db.test.save({ a : null });
```

show execution results

Query by example using a skiplist index

Constructs a query-by-example using a skiplist index: `collection.byExample(example)`

Selects all documents from the collection that match the specified example and returns a cursor. A skiplist index will be used if present.

You can use *toArray*, *next*, or *hasNext* to access the result. The result can be limited using the *skip* and *limit* operator.

An attribute name of the form *a.b* is interpreted as attribute path, not as attribute. If you use

```
{ "a" : { "c" : 1 } }
```

as example, then you will find all documents, such that the attribute *a* contains a document of the form *{c : 1 }*. For example the document

```
{ "a" : { "c" : 1 }, "b" : 1 }
```

will match, but the document

```
{ "a" : { "c" : 1, "b" : 1 } }
```

will not.

However, if you use

```
{ "a.c" : 1 },
```

then you will find all documents, which contain a sub-document in *a* that has an attribute *c* of value *1*. Both the following documents

```
{ "a" : { "c" : 1 }, "b" : 1 }
```

and

```
{ "a" : { "c" : 1, "b" : 1 } }
```

will match.

Introduction to Persistent Indexes

This is an introduction to ArangoDB's persistent indexes.

It is possible to define a persistent index on one or more attributes (or paths) of documents. The index is then used in queries to locate documents within a given range. If the index is declared unique, then no two documents are allowed to have the same set of attribute values.

Creating a new document or updating a document will fail if the uniqueness is violated. If the index is declared sparse, a document will be excluded from the index and no uniqueness checks will be performed if any index attribute value is not set or has a value of `null`.

Accessing Persistent Indexes from the Shell

ensures that a unique persistent index exists `collection.ensureIndex({ type: "persistent", fields: [ "field1", ..., "fieldn" ], unique: true })`

Creates a unique persistent index on all documents using *field1*, ... *fieldn* as attribute paths. At least one attribute path has to be given. The index will be non-sparse by default.

All documents in the collection must differ in terms of the indexed attributes. Creating a new document or updating an existing document will will fail if the attribute uniqueness is violated.

To create a sparse unique index, set the *sparse* attribute to `true`:

`collection.ensureIndex({ type: "persistent", fields: [ "field1", ..., "fieldn" ], unique: true, sparse: true })`

In a sparse index all documents will be excluded from the index that do not contain at least one of the specified index attributes or that have a value of `null` in any of the specified index attributes. Such documents will not be indexed, and not be taken into account for uniqueness checks.

In a non-sparse index, these documents will be indexed (for non-present indexed attributes, a value of `null` will be used) and will be taken into account for uniqueness checks.

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.ids.ensureIndex({ type: "persistent", fields: [ "myId" ], unique: true });
arangosh> db.ids.save({ "myId": 123 });
arangosh> db.ids.save({ "myId": 456 });
arangosh> db.ids.save({ "myId": 789 });
arangosh> db.ids.save({ "myId": 123 });
```

show execution results

```
arangosh> db.ids.ensureIndex({ type: "persistent", fields: [ "name.first", "name.last" ],
arangosh> db.ids.save({ "name" : { "first" : "hans", "last": "hansen" }});
arangosh> db.ids.save({ "name" : { "first" : "jens", "last": "jensen" }});
arangosh> db.ids.save({ "name" : { "first" : "hans", "last": "jensen" }});
arangosh> db.ids.save({ "name" : { "first" : "hans", "last": "hansen" }});
```

show execution results

ensures that a non-unique persistent index exists `collection.ensureIndex({ type: "persistent", fields: [ "field1", ..., "fieldn" ] })`

Creates a non-unique persistent index on all documents using *field1*, ... *fieldn* as attribute paths. At least one attribute path has to be given. The index will be non-sparse by default.

To create a sparse unique index, set the *sparse* attribute to `true`.

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

```
arangosh> db.names.ensureIndex({ type: "persistent", fields: [ "first" ] });
arangosh> db.names.save({ "first" : "Tim" });
arangosh> db.names.save({ "first" : "Tom" });
arangosh> db.names.save({ "first" : "John" });
arangosh> db.names.save({ "first" : "Tim" });
arangosh> db.names.save({ "first" : "Tom" });
```

show execution results

Query by example using a persistent index

constructs a query-by-example using a persistent index `collection.byExample(example)`

Selects all documents from the collection that match the specified example and returns a cursor. A persistent index will be used if present.

You can use *toArray*, *next*, or *hasNext* to access the result. The result can be limited using the *skip* and *limit* operator.

An attribute name of the form *a.b* is interpreted as attribute path, not as attribute. If you use

```
{ "a" : { "c" : 1 } }
```

as example, then you will find all documents, such that the attribute *a* contains a document of the form *{c : 1 }*. For example the document

```
{ "a" : { "c" : 1 }, "b" : 1 }
```

will match, but the document

```
{ "a" : { "c" : 1, "b" : 1 } }
```

will not.

However, if you use

```
{ "a.c" : 1 },
```

then you will find all documents, which contain a sub-document in *a* that has an attribute *c* of value *1*. Both the following documents

```
{ "a" : { "c" : 1 }, "b" : 1 }
```

and

```
{ "a" : { "c" : 1, "b" : 1 } }
```

will match.

Persistent Indexes and Server Language

The order of index entries in persistent indexes adheres to the configured server language. If, however, the server is restarted with a different language setting as when the persistent index was created, not all documents may be returned anymore and the sort order of those which are returned can be wrong (whenever the persistent index is consulted).

To fix persistent indexes after a language change, delete and re-create them. Skiplist indexes are not affected, because they are not persisted and automatically rebuilt on every server start.

Fulltext indexes

This is an introduction to ArangoDB's fulltext indexes.

Introduction to Fulltext Indexes

A fulltext index can be used to find words, or prefixes of words inside documents.

A fulltext index can be defined on one attribute only, and will include all words contained in documents that have a textual value in the index attribute. Since ArangoDB 2.6 the index will also include words from the index attribute if the index attribute is an array of strings, or an object with string value members.

For example, given a fulltext index on the `translations` attribute and the following documents, then searching for `лиса` using the fulltext index would return only the first document. Searching for the index for the exact string `Fox` would return the first two documents, and searching for `prefix:Fox` would return all three documents:

```
{ translations: { en: "fox", de: "Fuchs", fr: "renard", ru: "лиса" } }
{ translations: "Fox is the English translation of the German word Fuchs" }
{ translations: [ "ArangoDB", "document", "database", "Foxx" ] }
```

Note that deeper nested objects are ignored. For example, a fulltext index on *translations* would index *Fuchs*, but not *fox*, given the following document structure:

```
{ translations: { en: { US: "fox" }, de: "Fuchs" }
```

If you need to search across multiple fields and/or nested objects, you may write all the strings into a special attribute, which you then create the index on (it might be necessary to clean the strings first, e.g. remove line breaks and strip certain words).

If the index attribute is neither a string, an object or an array, its contents will not be indexed. When indexing the contents of an array attribute, an array member will only be included in the index if it is a string. When indexing the contents of an object attribute, an object member value will only be included in the index if it is a string. Other data types are ignored and not indexed.

Accessing Fulltext Indexes from the Shell

Ensures that a fulltext index exists:

`collection.ensureIndex({ type: "fulltext", fields: [ "field" ], minLength: minLength })`

Creates a fulltext index on all documents on attribute *field*.

Fulltext indexes are implicitly sparse: all documents which do not have the specified *field* attribute or that have a non-qualifying value in their *field* attribute will be ignored for indexing.

Only a single attribute can be indexed. Specifying multiple attributes is unsupported.

The minimum length of words that are indexed can be specified via the *minLength* parameter. Words shorter than minLength characters will not be indexed. *minLength* has a default value of 2, but this value might be changed in future versions of ArangoDB. It is thus recommended to explicitly specify this value.

In case that the index was successfully created, an object with the index details is returned.

```
arangosh> db.example.ensureIndex({ type: "fulltext", fields: [ "text" ], minLength: 3 });
arangosh> db.example.save({ text : "the quick brown", b : { c : 1 } });
arangosh> db.example.save({ text : "quick brown fox", b : { c : 2 } });
arangosh> db.example.save({ text : "brown fox jums", b : { c : 3 } });
arangosh> db.example.save({ text : "fox jumps over", b : { c : 4 } });
arangosh> db.example.save({ text : "jumps over the", b : { c : 5 } });
arangosh> db.example.save({ text : "over the lazy", b : { c : 6 } });
arangosh> db.example.save({ text : "the lazy dog", b : { c : 7 } });
arangosh> db._query("FOR document IN FULLTEXT(example, 'text', 'the') RETURN document");
```

show execution results

Looks up a fulltext index:

`collection.lookupFulltextIndex(attribute, minLength)`

Checks whether a fulltext index on the given attribute *attribute* exists.

Fulltext AQL Functions

Fulltext AQL functions are detailed in Fulltext functions.

Geo Indexes

Introduction to Geo Indexes

This is an introduction to ArangoDB's geo indexes.

AQL's geographic features are described in Geo functions.

ArangoDB uses Hilbert curves to implement geo-spatial indexes. See this blog for details.

A geo-spatial index assumes that the latitude is between -90 and 90 degree and the longitude is between -180 and 180 degree. A geo index will ignore all documents which do not fulfill these requirements.

Accessing Geo Indexes from the Shell

ensures that a geo index exists `collection.ensureIndex({ type: "geo", fields: [ "location" ] })`

Creates a geo-spatial index on all documents using *location* as path to the coordinates. The value of the attribute has to be an array with at least two numeric values. The array must contain the latitude (first value) and the longitude (second value).

All documents, which do not have the attribute path or have a non-conforming value in it are excluded from the index.

A geo index is implicitly sparse, and there is no way to control its sparsity.

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

To create a geo on an array attribute that contains longitude first, set the *geoJson* attribute to `true`. This corresponds to the format described in positions

`collection.ensureIndex({ type: "geo", fields: [ "location" ], geoJson: true })`

To create a geo-spatial index on all documents using *latitude* and *longitude* as separate attribute paths, two paths need to be specified in the *fields* array:

`collection.ensureIndex({ type: "geo", fields: [ "latitude", "longitude" ] })`

In case that the index was successfully created, an object with the index details, including the index-identifier, is returned.

Examples

Create a geo index for an array attribute:

```
arangosh> db.geo.ensureIndex({ type: "geo", fields: [ "loc" ] });
arangosh> for (i = -90;  i <= 90;  i += 10) {
........>     for (j = -180; j <= 180; j += 10) {
........>         db.geo.save({ name : "Name/" + i + "/" + j, loc: [ i, j ] });
........>     }
........> }
arangosh> db.geo.count();
arangosh> db.geo.near(0, 0).limit(3).toArray();
arangosh> db.geo.near(0, 0).count();
```

show execution results
Create a geo index for a hash array attribute:

```
arangosh> db.geo2.ensureIndex({ type: "geo", fields: [ "location.latitude", "location.long
arangosh> for (i = -90;  i <= 90;  i += 10) {
........>     for (j = -180; j <= 180; j += 10) {
........>         db.geo2.save({ name : "Name/" + i + "/" + j, location: { latitude : i, l
........>     }
........> }
arangosh> db.geo2.near(0, 0).limit(3).toArray();
```

show execution results

constructs a geo index selection `collection.geo(location-attribute)` Looks up a geo index defined on attribute *location_attribute*. Returns a geo index object if an index was found. The `near` or `within` operators can then be used to execute a geo-spatial query on this particular index. This is useful for collections with multiple defined geo indexes. `collection.geo(location_attribute, true)` Looks up a geo index on a compound attribute *location_attribute*. Returns a geo index object if an index was found. The `near` or `within` operators can then be used to execute a geo-spatial query on this particular index. `collection.geo(latitude_attribute, longitude_attribute)` Looks up a geo index defined on the two attributes *latitude_attribute* and *longitude-attribute*. Returns a geo index object if an index was found. The `near` or `within` operators can then be used to execute a geo-spatial query on this particular index. Note: the *geo* simple query helper function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for running geo queries is to use their AQL equivalents.

Examples

Assume you have a location stored as list in the attribute *home* and a destination stored in the attribute *work*. Then you can use the `geo` operator to select which geo-spatial attributes (and thus which index) to use in a `near` query.

```
arangosh> for (i = -90;  i <= 90;  i += 10) {
........>  for (j = -180;  j <= 180;  j += 10) {
........>    db.complex.save({ name : "Name/" + i + "/" + j,
........>                      home : [ i, j ],
........>                      work : [ -i, -j ] });
........>  }
........> }
........>
arangosh> db.complex.near(0, 170).limit(5);
arangosh> db.complex.ensureIndex({ type: "geo", fields: [ "home" ] });
arangosh> db.complex.near(0, 170).limit(5).toArray();
arangosh> db.complex.geo("work").near(0, 170).limit(5);
arangosh> db.complex.ensureIndex({ type: "geo", fields: [ "work" ] });
arangosh> db.complex.geo("work").near(0, 170).limit(5).toArray();
```

show execution results

constructs a near query for a collection `collection.near(latitude, longitude)` The returned list is sorted according to the distance, with the nearest document to the coordinate (*latitude*, *longitude*) coming first. If there are near documents of equal distance, documents are chosen randomly from this set until the limit is reached. It is possible to change the limit using the *limit* operator. In order to use the *near* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* operator to select a particular index. *Note*: `near` does not support negative skips. // However, you can still use `limit` followed to skip. `collection.near(latitude, longitude).limit(limit)` Limits the result to limit documents instead of the default 100. *Note*: Unlike with multiple explicit limits, `limit` will raise the implicit default limit imposed by `within`. `collection.near(latitude, longitude).distance()` This will add an attribute `distance` to all documents returned, which contains the distance between the given point and the document in meters. `collection.near(latitude, longitude).distance(name)` This will add an attribute *name* to all documents returned, which contains the distance between the given point and the document in meters. Note: the *near* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for retrieving documents from a collection using the near operator is to use the AQL *NEAR* function in an AQL query as follows:

```
FOR doc IN NEAR(@@collection, @latitude, @longitude, @limit)
    RETURN doc
```

Examples

To get the nearest two locations:

```
arangosh> db.geo.ensureIndex({ type: "geo", fields: [ "loc" ] });
arangosh> for (var i = -90;  i <= 90;  i += 10) {
........>   for (var j = -180; j <= 180; j += 10) {
........>     db.geo.save({
........>        name : "Name/" + i + "/" + j,
........>        loc: [ i, j ] });
........> } }
arangosh> db.geo.near(0, 0).limit(2).toArray();
```

show execution results

If you need the distance as well, then you can use the `distance` operator:

```
arangosh> db.geo.ensureIndex({ type: "geo", fields: [ "loc" ] });
arangosh> for (var i = -90;  i <= 90;  i += 10) {
........>  for (var j = -180; j <= 180; j += 10) {
........>     db.geo.save({
........>         name : "Name/" + i + "/" + j,
........>         loc: [ i, j ] });
........> } }
arangosh> db.geo.near(0, 0).distance().limit(2).toArray();
```

show execution results

constructs a within query for a collection `collection.within(latitude, longitude, radius)` This will find all documents within a given radius around the coordinate (*latitude*, *longitude*). The returned array is sorted by distance, beginning with the nearest document. In order to use the *within* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the `geo` operator to select a particular index. `collection.within(latitude, longitude, radius).distance()` This will add an attribute `_distance` to all documents returned, which contains the distance between the given point and the document in meters. `collection.within(latitude, longitude, radius).distance(name)` This will add an attribute *name* to all documents returned, which contains the distance between the given point and the document in meters. Note: the *within* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for retrieving documents from a collection using the within operator is to use the AQL *WITHIN* function in an AQL query as follows:

```
FOR doc IN WITHIN(@@collection, @latitude, @longitude, @radius, @distanceAttributeName)
    RETURN doc
```

Examples

To find all documents within a radius of 2000 km use:

```
arangosh> for (var i = -90;  i <= 90;  i += 10) {
........>  for (var j = -180; j <= 180; j += 10) {
........> db.geo.save({ name : "Name/" + i + "/" + j, loc: [ i, j ] }); } }
arangosh> db.geo.within(0, 0, 2000 * 1000).distance().toArray();
```

show execution results

ensures that a geo index exists `collection.ensureIndex({ type: "geo", fields: [ "location" ] })`

Since ArangoDB 2.5, this method is an alias for *ensureGeoIndex* since geo indexes are always sparse, meaning that documents that do not contain the index attributes or have non-numeric values in the index attributes will not be indexed. *ensureGeoConstraint* is deprecated and *ensureGeoIndex* should be used instead.

The index does not provide a `unique` option because of its limited usability. It would prevent identical coordinates from being inserted only, but even a slightly different location (like 1 inch or 1 cm off) would be unique again and not considered a duplicate, although it probably should. The desired threshold for detecting duplicates may vary for every project (including how to calculate the distance even) and needs to be implemented on the application layer as needed. You can write a Foxx service for this purpose and make use of the AQL geo functions to find nearby coordinates supported by a geo index.

ArangoDB Graphs

First Steps with Graphs

A Graph consists of *vertices* and *edges*. Edges are stored as documents in *edge collections*. A vertex can be a document of a *document collection* or of an *edge collection* (so *edges* can be used as *vertices*). Which collections are used within a named graph is defined via *edge definitions*. A named graph can contain more than one *edge definition*, at least one is needed. Graphs allow you to structure your models in line with your domain and group them logically in collections and giving you the power to query them in the same graph queries.

Coming from a relational background - what's a graph?

In SQL you commonly have the construct of a relation table to store *n:m* relations between two data tables. An *edge collection* is somewhat similar to these *relation tables*; *vertex collections* resemble the data tables with the objects to connect. While simple graph queries with fixed number of hops via the relation table may be doable in SQL with several nested joins, graph databases can handle an arbitrary number of these hops over edge collections - this is called *traversal*. Also edges in one edge collection may point to several vertex collections. Its common to have attributes attached to edges, i.e. a *label* naming this interconnection. Edges have a direction, with their relations `_from` and `_to` pointing *from* one document *to* another document stored in vertex collections. In queries you can define in which directions the edge relations may be followed (`OUTBOUND` : `_from` → `_to` , `INBOUND` : `_from` ← `_to` , `ANY` : `_from` ↔ `_to`).

Named Graphs

Named graphs are completely managed by arangodb, and thus also visible in the webinterface. They use the full spectrum of ArangoDBs graph features. You may access them via several interfaces.

- AQL Graph Operations with several flavors:
  - AQL Traversals on both named and anonymous graphs
  - AQL Shortest Path on both named and anonymous graph
- JavaScript General Graph implementation, as you may use it in Foxx Services
  - Graph Management; creating & manipualating graph definitions; inserting, updating and deleting vertices and edges into graphs
  - Graph Functions for working with edges and vertices, to analyze them and their relations
- JavaScript Smart Graph implementation, for scalable graphs
  - Smart Graph Management; creating & manipualating SmartGraph definitions; Differences to General Graph
- RESTful General Graph interface used to implement graph management in client drivers

Manipulating collections of named graphs with regular document functions

The underlying collections of the named graphs are still accessible using the standard methods for collections. However the graph module adds an additional layer on top of these collections giving you the following guarantees:

- All modifications are executed transactional
- If you delete a vertex all edges will be deleted, you will never have loose ends
- If you insert an edge it is checked if the edge matches the *edge definitions*, your edge collections will only contain valid edges

These guarantees are lost if you access the collections in any other way than the graph module or AQL, so if you delete documents from your vertex collections directly, the edges pointing to them will be remain in place.

Anonymous graphs

Sometimes you may not need all the powers of named graphs, but some of its bits may be valuable to you. You may use anonymous graphs in the traversals and in the Working with Edges chapter. Anonymous graphs don't have *edge definitions* describing which *vertex collection* is connected by which *edge collection*. The graph model has to be maintained in the client side code. This gives you more freedom than the strict *named graphs*.

- AQL Graph Operations are available for both, named and anonymous graphs:

- AQL Traversals
- AQL Shortest Path

When to choose anonymous or named graphs?

As noted above, named graphs ensure graph integrity, both when inserting or removing edges or vertices. So you won't encounter dangling edges, even if you use the same vertex collection in several named graphs. This involves more operations inside the database which don't come for free. Therefore anonymous graphs may be faster in many operations. So this question boils down to 'Can I afford the additional effort or do I need the warranty for integrity?'.

Multiple edge collections vs. `FILTER` s on edge document attributes

If you want to only traverse edges of a specific type, there are two ways to achieve this. The first would be an attribute in the edge document - i.e. `type` , where you specify a differentiator for the edge - i.e. `"friends"` , `"family"` , `"married"` or `"workmates"` , so you can later `FILTER e.type = "friends"` if you only want to follow the friend edges.

Another way, which may be more efficient in some cases, is to use different edge collections for different types of edges, so you have `friend_eges` , `family_edges` , `married_edges` and `workmate_edges` as collection names. You can then configure several named graphs including a subset of the available edge and vertex collections - or you use anonymous graph queries, where you specify a list of edge collections to take into account in that query. To only follow friend edges, you would specify `friend_edges` as sole edge collection.

Both approaches have advantages and disadvantages. `FILTER` operations on ede attributes will do comparisons on each traversed edge, which may become CPU-intense. When not *finding* the edges in the first place because of the collection containing them is not traversed at all, there will never be a reason to actualy check for their `type` attribute with `FILTER` .

The multiple edge collections approach is limited by the number of collections that can be used simultaneously in one query. Every collection used in a query requires some resources inside of ArangoDB and the number is therefore limited to cap the resource requirements. You may also have constraints on other edge attributes, such as a hash index with a unique constraint, which requires the documents to be in a single collection for the uniqueness guarantee, and it may thus not be possible to store the different types of edges in multiple edeg collections.

So, if your edges have about a dozen different types, it's okay to choose the collection approach, otherwise the `FILTER` approach is preferred. You can still use `FILTER` operations on edges of course. You can get rid of a `FILTER` on the `type` with the former approach, everything else can stay the same.

Which part of my data is an Edge and which a Vertex?

The main objects in your data model, such as users, groups or articles, are usually considered to be vertices. For each type of object, a document collection (also called vertex collection) should store the individual entities. Entities can be connected by edges to express and classify relations between vertices. It often makes sense to have an edge collection per relation type.

ArangoDB does not require you to store your data in graph structures with edges and vertices, you can also decide to embed attributes such as which groups a user is part of, or `_id` s of documents in another document instead of connecting the documents with edges. It can be a meaningful performance optimization for *1:n* relationships, if your data is not very focused on relations and you don't need graph traversal with varying depth. It usually means to introduce some redundancy and possibly inconsistencies if you embed data, but it can be an acceptable tradeoff.

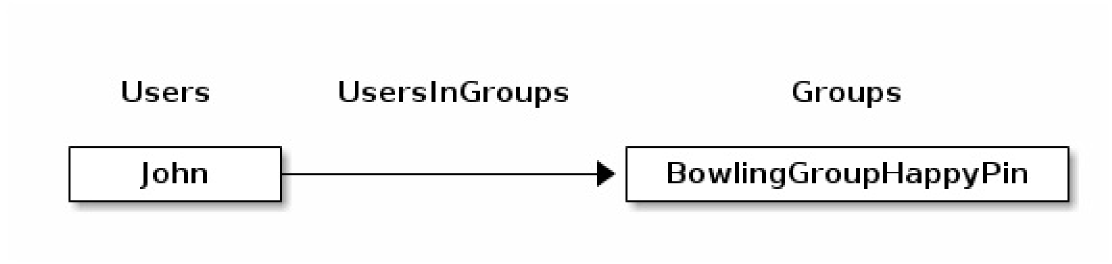
Vertices

Let's say we have two vertex collections, `Users` and `Groups` . Documents in the `Groups` collection contain the attributes of the Group, i.e. when it was founded, its subject, an icon URL and so on. `Users` documents contain the data specific to a user - like all names, birthdays, Avatar URLs, hobbies...

Edges

We can use an edge collection to store relations between users and groups. Since multiple users may be in an arbitrary number of groups, this is an **m:n** relation. The edge collection can be called `UsersInGroups` with i.e. one edge with `_from` pointing to `Users/John` and `_to` pointing to `Groups/BowlingGroupHappyPin` . This makes the user **John** a member of the group **Bowling Group Happy Pin**.

Attributes of this relation may contain qualifiers to this relation, like the permissions of **John** in this group, the date when he joined the group etc.



Advantages of this approach

Graphs give you the advantage of not just being able to have a fixed number of **m:n** relations in a row, but an arbitrary number. Edges can be traversed in both directions, so it's very easy to determine all groups a user is in, but also to find out which members a certain group has. Users could also be interconnected to create a social network.

Using the graph data model, dealing with data that has lots of relations stays manageable and can be queried in very flexible ways, whereas it would cause headache to handle it in a relational database system.

Backup and restore

For sure you want to have backups of your graph data, you can use Arangodump to create the backup, and Arangorestore to restore a backup into a new ArangoDB. You should however note that:

- you need the system collection `_graphs` if you backup named graphs.
- you need to backup the complete set of all edge and vertex collections your graph consists of. Partial dump/restore may not work.

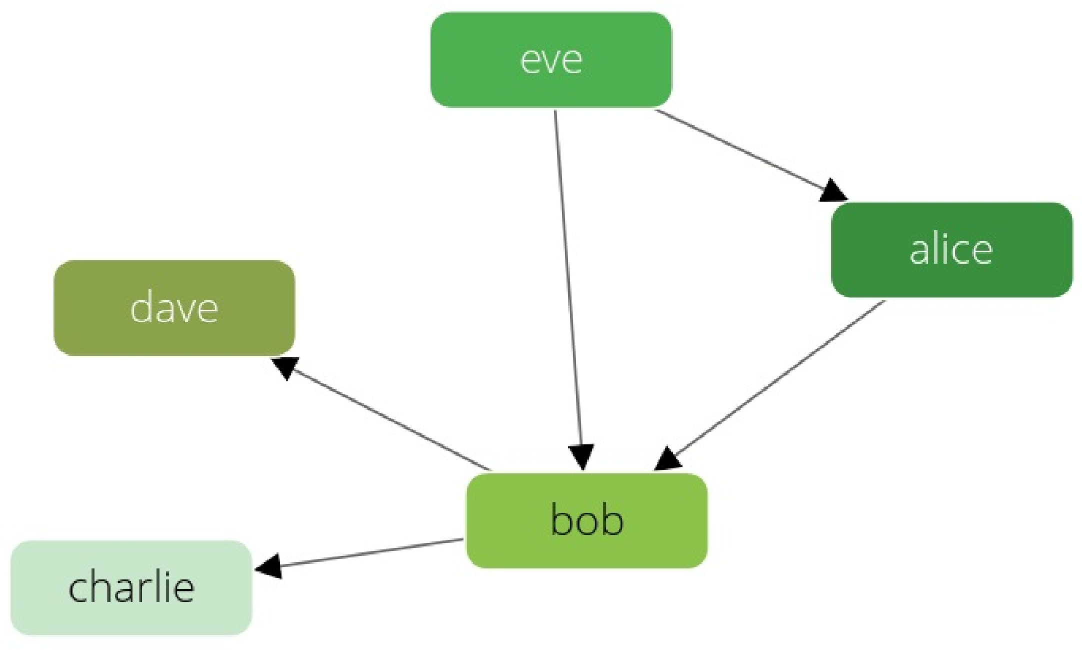
Example Graphs

ArangoDB comes with a set of easily graspable graphs that are used to demonstrate the APIs. You can use the `add samples` tab in the `create graph` window in the webinterface, or load the module `@arangodb/graph-examples/example-graph` in arangosh and use it to create instances of these graphs in your ArangoDB. Once you've created them, you can inspect them in the webinterface - which was used to create the pictures below.

You can easily look into the innards of this script for reference about howto manage graphs programatically.

The Knows_Graph

A set of persons knowing each other:



The *knows* graph consists of one *vertex collection* `persons` connected via one *edge collection* `knows`. It will contain five persons *Alice*, *Bob*, *Charlie*, *Dave* and *Eve*. We will have the following directed relations:

- *Alice* knows *Bob*
- *Bob* knows *Charlie*
- *Bob* knows *Dave*
- *Eve* knows *Alice*
- *Eve* knows *Bob*

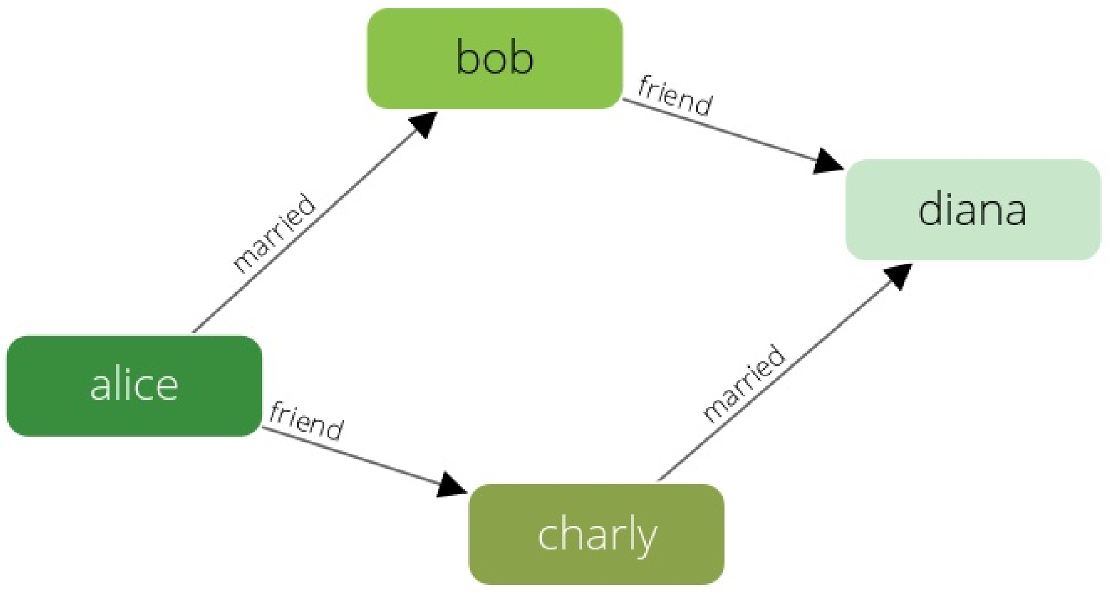
This is how we create it, inspect its *vertices* and *edges*, and drop it again:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("knows_graph");
arangosh> db.persons.toArray()
arangosh> db.knows.toArray();
arangosh> examples.dropGraph("knows_graph");
```

show execution results

The Social Graph

A set of persons and their relations:



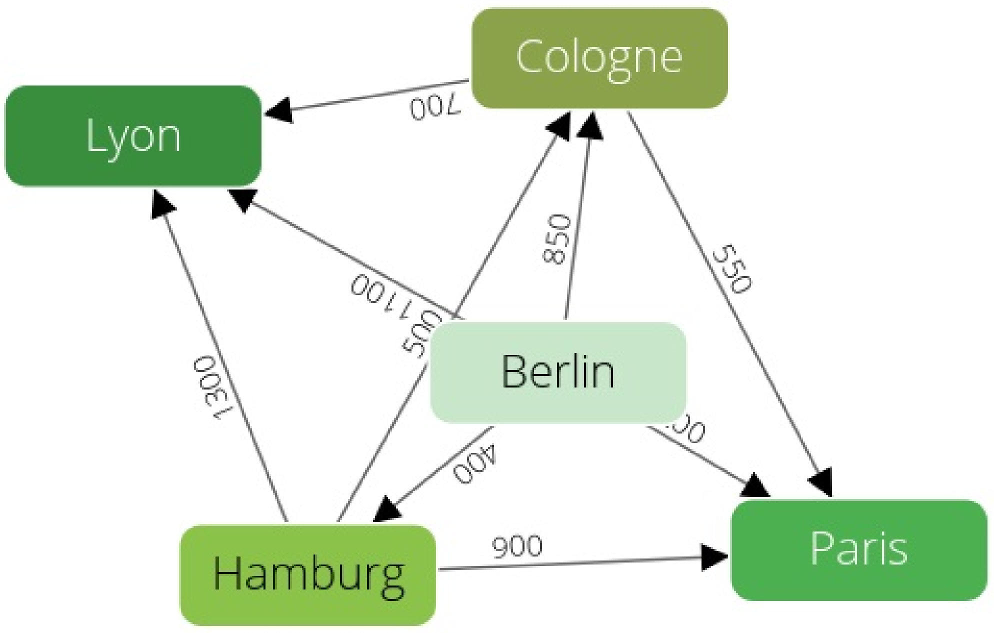
This example has female and male persons as *vertices* in two *vertex collections* - `female` and `male`. The *edges* are their connections in the `relation` *edge collection*. This is how we create it, inspect its *vertices* and *edges*, and drop it again:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> db.female.toArray()
arangosh> db.male.toArray()
arangosh> db.relation.toArray()
arangosh> examples.dropGraph("social");
```

show execution results

The City Graph

A set of european cities, and their fictional traveling distances as connections:



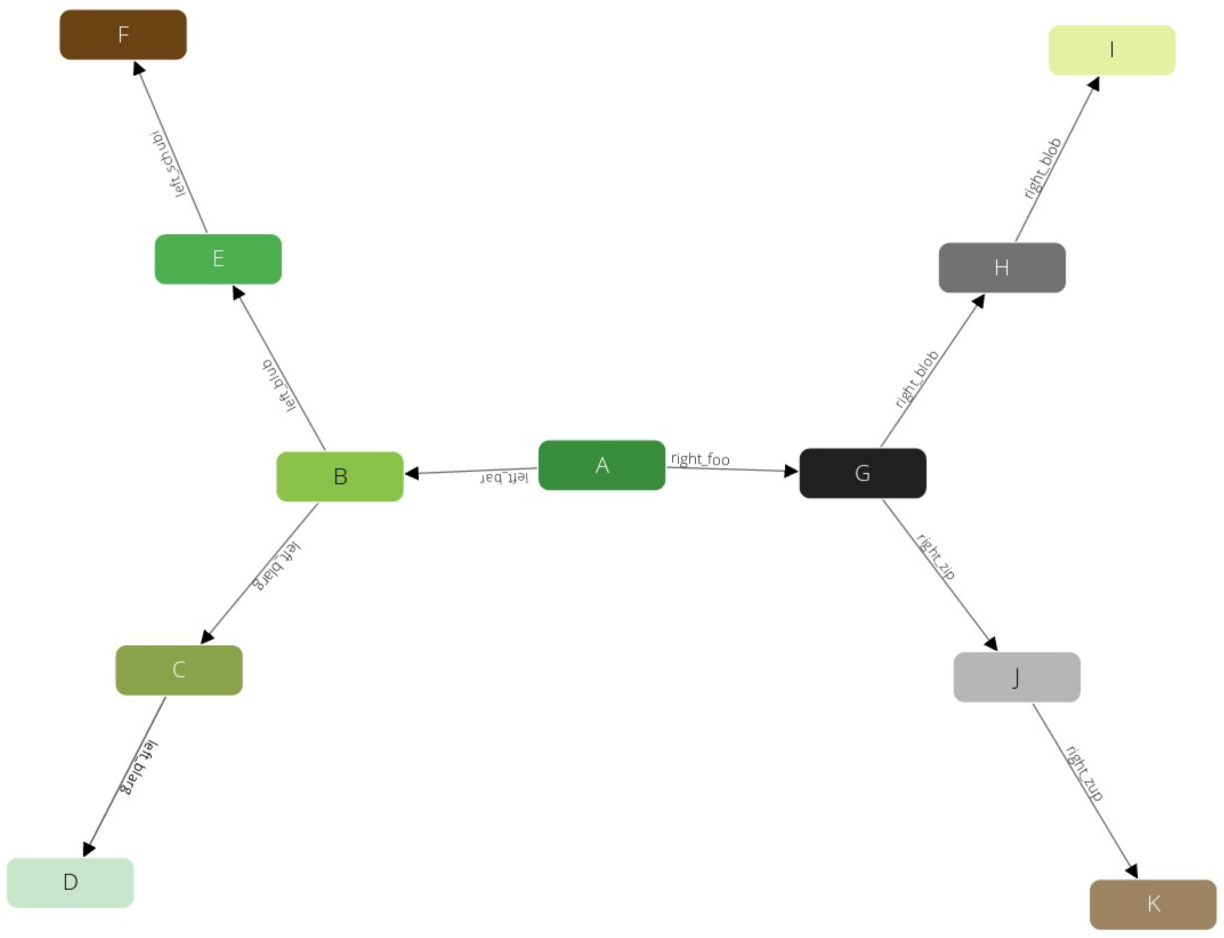
The example has the cities as *vertices* in several *vertex collections* - `germanCity` and `frenchCity` . The *edges* are their interconnections in several *edge collections* `french / german / international Highway` . This is how we create it, inspect its *edges* and *vertices*, and drop it again:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db.frenchCity.toArray();
arangosh> db.germanCity.toArray();
arangosh> db.germanHighway.toArray();
arangosh> db.frenchHighway.toArray();
arangosh> db.internationalHighway.toArray();
arangosh> examples.dropGraph("routeplanner");
```

show execution results

The Traversal Graph

This graph was designed to demonstrate filters in traversals. It has some labels to filter on it.



The example has all its vertices in the *circles* collection, and an *edges* edge collection to connect them. Circles have unique numeric labels. Edges have two boolean attributes (*theFalse* always being false, *theTruth* always being true) and a label sorting *B* - *D* to the left side, *G* - *K* to the right side. Left and right side split into Paths - at *B* and *G* which are each direct neighbours of the root-node *A*. Starting from *A* the graph has a depth of 3 on all its paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("traversalGraph");
arangosh> db.circles.toArray();
arangosh> db.edges.toArray();
arangosh> examples.dropGraph("traversalGraph");
```

show execution results

The World Graph

The world country graph structures its nodes like that: world → continent → country → capital. In some cases edge directions aren't forward (therefore it will be displayed disjunct in the graph viewer). It has two ways of creating it. One using the named graph utilities (*worldCountry*), one without (*worldCountryUnManaged*). It is used to demonstrate raw traversal operations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("worldCountry");
arangosh> db.worldVertices.toArray();
arangosh> db.worldEdges.toArray();
arangosh> examples.dropGraph("worldCountry");
arangosh> var g = examples.loadGraph("worldCountryUnManaged");
arangosh> examples.dropGraph("worldCountryUnManaged");
```

show execution results

Cookbook examples

The above referenced chapters describe the various APIs of ArangoDBs graph engine with small examples. Our cookbook has some more real life examples:

- Traversing a graph in full depth
- Using an example vertex with the java driver
- Retrieving documents from ArangoDB without knowing the structure
- Using a custom visitor from node.js
- AQL Example Queries on an Actors and Movies Database

Higher volume graph examples

All of the above examples are rather small so they are easier to comprehend and can demonstrate the way the functionality works. There are however several datasets freely available on the web that are a lot bigger. We collected some of them with import scripts so you may play around with them. Another huge graph is the Pokec social network from Slovakia that we used for performance testing on several databases; You will find importing scripts etc. in this blogpost.

Graphs

This chapter describes the general-graph module. It allows you to define a graph that is spread across several edge and document collections. This allows you to structure your models in line with your domain and group them logically in collections giving you the power to query them in the same graph queries. There is no need to include the referenced collections within the query, this module will handle it for you.

Three Steps to create a graph

- Create a graph

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> var graph = graph_module._create("myGraph");
arangosh> graph;
```

show execution results

- Add some vertex collections

```
arangosh> graph._addVertexCollection("shop");
arangosh> graph._addVertexCollection("customer");
arangosh> graph._addVertexCollection("pet");
arangosh> graph;
```

show execution results

- Define relations on the Graph

```
arangosh> var rel = graph_module._relation("isCustomer", ["shop"], ["customer"]);
arangosh> graph._extendEdgeDefinitions(rel);
arangosh> graph;
```

show execution results

Graph Management

This chapter describes the javascript interface for creating and modifying named graphs. In order to create a non empty graph the functionality to create edge definitions has to be introduced first:

Edge Definitions

An edge definition is always a directed relation of a graph. Each graph can have arbitrary many relations defined within the edge definitions array.

Initialize the list

Create a list of edge definitions to construct a graph.

`graph_module._edgeDefinitions(relation1, relation2, ..., relationN)`

The list of edge definitions of a graph can be managed by the graph module itself. This function is the entry point for the management and will return the correct list.

Parameters

- relationX (optional) An object representing a definition of one relation in the graph

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> directed_relation = graph_module._relation("lives_in", "user", "city");
arangosh> undirected_relation = graph_module._relation("knows", "user", "user");
arangosh> edgedefinitions = graph_module._edgeDefinitions(directed_relation, undirected_re
```

show execution results

Extend the list

Extend the list of edge definitions to construct a graph.

`graph_module._extendEdgeDefinitions(edgeDefinitions, relation1, relation2, ..., relationN)`

In order to add more edge definitions to the graph before creating this function can be used to add more definitions to the initial list.

Parameters

- edgeDefinitions (required) A list of relation definition objects.
- relationX (required) An object representing a definition of one relation in the graph

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> directed_relation = graph_module._relation("lives_in", "user", "city");
arangosh> undirected_relation = graph_module._relation("knows", "user", "user");
arangosh> edgedefinitions = graph_module._edgeDefinitions(directed_relation);
arangosh> edgedefinitions = graph_module._extendEdgeDefinitions(undirected_relation);
```

show execution results

Relation

Define a directed relation.

`graph_module._relation(relationName, fromVertexCollections, toVertexCollections)`

The *relationName* defines the name of this relation and references to the underlying edge collection. The *fromVertexCollections* is an Array of document collections holding the start vertices. The *toVertexCollections* is an Array of document collections holding the target vertices. Relations are only allowed in the direction from any collection in *fromVertexCollections* to any collection in *toVertexCollections*.

Parameters

- relationName (required) The name of the edge collection where the edges should be stored. Will be created if it does not yet exist.
- fromVertexCollections (required) One or a list of collection names. Source vertices for the edges have to be stored in these collections. Collections will be created if they do not exist.
- toVertexCollections (required) One or a list of collection names. Target vertices for the edges have to be stored in these collections. Collections will be created if they do not exist.

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph_module._relation("has_bought", ["Customer", "Company"], ["Groceries", "Ele
```

show execution results

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph_module._relation("has_bought", "Customer", "Product");
```

show execution results

Create a graph

After having introduced edge definitions a graph can be created.

Create a graph

`graph_module._create(graphName, edgeDefinitions, orphanCollections)`

The creation of a graph requires the name of the graph and a definition of its edges.

For every type of edge definition a convenience method exists that can be used to create a graph. Optionally a list of vertex collections can be added, which are not used in any edge definition. These collections are referred to as orphan collections within this chapter. All collections used within the creation process are created if they do not exist.

Parameters

- graphName (required) Unique identifier of the graph
- edgeDefinitions (optional) List of relation definition objects
- orphanCollections (optional) List of additional vertex collection names

Examples

Create an empty graph, edge definitions can be added at runtime:

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph = graph_module._create("myGraph");
```

show execution results

Create a graph using an edge collection `edges` and a single vertex collection `vertices`

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> var edgeDefinitions = [ { collection: "edges", "from": [ "vertices" ], "to" : [
arangosh> graph = graph_module._create("myGraph", edgeDefinitions);
```

show execution results
Create a graph with edge definitions and orphan collections:

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph = graph_module._create("myGraph",
........> [graph_module._relation("myRelation", ["male", "female"], ["male", "female"])],
```

show execution results

Complete Example to create a graph

Example Call:

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> var edgeDefinitions = graph_module._edgeDefinitions();
arangosh> graph_module._extendEdgeDefinitions(edgeDefinitions, graph_module._relation("fri
arangosh> graph_module._extendEdgeDefinitions(
........> edgeDefinitions, graph_module._relation(
........> "has_bought", ["Customer", "Company"], ["Groceries", "Electronics"]));
arangosh> graph_module._create("myStore", edgeDefinitions);
```

show execution results
alternative call:

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh>  var edgeDefinitions = graph_module._edgeDefinitions(
........>  graph_module._relation("friend_of", ["Customer"], ["Customer"]), graph_module._
........> "has_bought", ["Customer", "Company"], ["Groceries", "Electronics"]));
arangosh> graph_module._create("myStore", edgeDefinitions);
```

show execution results

List available graphs

List all graphs.

`graph_module._list()`

Lists all graph names stored in this database.

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph_module._list();
[ ]
```

Load a graph

Get a graph

`graph_module._graph(graphName)`

A graph can be retrieved by its name.

Parameters

- graphName (required) Unique identifier of the graph

Examples

Get a graph:

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph = graph_module._graph("social");
```

show execution results

Remove a graph

Remove a graph

`graph_module._drop(graphName, dropCollections)`

A graph can be dropped by its name. This can drop all collections contained in the graph as long as they are not used within other graphs. To drop the collections only belonging to this graph, the optional parameter *drop-collections* has to be set to *true*.

Parameters

- graphName (required) Unique identifier of the graph
- dropCollections (optional) Define if collections should be dropped (default: false)

Examples

Drop a graph and keep collections:

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph_module._drop("social");
true
arangosh> db._collection("female");
[ArangoCollection 15546, "female" (type document, status loaded)]
arangosh> db._collection("male");
[ArangoCollection 15548, "male" (type document, status loaded)]
arangosh> db._collection("relation");
[ArangoCollection 15550, "relation" (type edge, status loaded)]
```

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> graph_module._drop("social", true);
true
arangosh> db._collection("female");
null
arangosh> db._collection("male");
null
arangosh> db._collection("relation");
null
```

Modify a graph definition during runtime

After you have created an graph its definition is not immutable. You can still add, delete or modify edge definitions and vertex collections.

Extend the edge definitions

Add another edge definition to the graph

`graph._extendEdgeDefinitions(edgeDefinition)`

Extends the edge definitions of a graph. If an orphan collection is used in this edge definition, it will be removed from the orphanage. If the edge collection of the edge definition to add is already used in the graph or used in a different graph with different *from* and/or *to* collections an error is thrown.

Parameters

- edgeDefinition (required) The relation definition to extend the graph

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph")
arangosh> var ed1 = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var ed2 = graph_module._relation("myEC2", ["myVC1"], ["myVC3"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._extendEdgeDefinitions(ed2);
```

Modify an edge definition

Modify an relation definition

`graph_module._editEdgeDefinition(edgeDefinition)`

Edits one relation definition of a graph. The edge definition used as argument will replace the existing edge definition of the graph which has the same collection. Vertex Collections of the replaced edge definition that are not used in the new definition will transform to an orphan. Orphans that are used in this new edge definition will be deleted from the list of orphans. Other graphs with the same edge definition will be modified, too.

Parameters

- edgeDefinition (required) The edge definition to replace the existing edge definition with the same attribute *collection*.

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph")
arangosh> var original = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var modified = graph_module._relation("myEC1", ["myVC2"], ["myVC3"]);
arangosh> var graph = graph_module._create("myGraph", [original]);
arangosh> graph._editEdgeDefinitions(modified);
```

Delete an edge definition

Delete one relation definition

`graph_module._deleteEdgeDefinition(edgeCollectionName, dropCollection)`

Deletes a relation definition defined by the edge collection of a graph. If the collections defined in the edge definition (collection, from, to) are not used in another edge definition of the graph, they will be moved to the orphanage.

Parameters

- edgeCollectionName (required) Name of edge collection in the relation definition.
- dropCollection (optional) Define if the edge collection should be dropped. Default false.

Examples

Remove an edge definition but keep the edge collection:

```
arangosh> var graph_module = require("@arangodb/general-graph")
arangosh> var ed1 = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var ed2 = graph_module._relation("myEC2", ["myVC1"], ["myVC3"]);
arangosh> var graph = graph_module._create("myGraph", [ed1, ed2]);
arangosh> graph._deleteEdgeDefinition("myEC1");
arangosh> db._collection("myEC1");
[ArangoCollection 19360, "myEC1" (type edge, status loaded)]
```

Remove an edge definition and drop the edge collection:

```
arangosh> var graph_module = require("@arangodb/general-graph")
arangosh> var ed1 = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var ed2 = graph_module._relation("myEC2", ["myVC1"], ["myVC3"]);
arangosh> var graph = graph_module._create("myGraph", [ed1, ed2]);
arangosh> graph._deleteEdgeDefinition("myEC1", true);
arangosh> db._collection("myEC1");
null
```

Extend vertex Collections

Each graph can have an arbitrary amount of vertex collections, which are not part of any edge definition of the graph. These collections are called orphan collections. If the graph is extended with an edge definition using one of the orphans, it will be removed from the set of orphan collection automatically.

Add a vertex collection

Add a vertex collection to the graph

`graph._addVertexCollection(vertexCollectionName, createCollection)`

Adds a vertex collection to the set of orphan collections of the graph. If the collection does not exist, it will be created. If it is already used by any edge definition of the graph, an error will be thrown.

Parameters

- vertexCollectionName (required) Name of vertex collection.
- createCollection (optional) If true the collection will be created if it does not exist. Default: true.

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph");
arangosh> var ed1 = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._addVertexCollection("myVC3", true);
```

Get the orphaned collections

Get all orphan collections

`graph._orphanCollections()`

Returns all vertex collections of the graph that are not used in any edge definition.

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph")
arangosh> var ed1 = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._addVertexCollection("myVC3", true);
arangosh> graph._orphanCollections();
[
  "myVC3"
]
```

Remove a vertex collection

Remove a vertex collection from the graph

`graph._removeVertexCollection(vertexCollectionName, dropCollection)`

Removes a vertex collection from the graph. Only collections not used in any relation definition can be removed. Optionally the collection can be deleted, if it is not used in any other graph.

Parameters

- vertexCollectionName (required) Name of vertex collection.
- dropCollection (optional) If true the collection will be dropped if it is not used in any other graph. Default: false.

Examples

```
arangosh> var graph_module = require("@arangodb/general-graph")
arangosh> var ed1 = graph_module._relation("myEC1", ["myVC1"], ["myVC2"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._addVertexCollection("myVC3", true);
arangosh> graph._addVertexCollection("myVC4", true);
arangosh> graph._orphanCollections();
arangosh> graph._removeVertexCollection("myVC3");
arangosh> graph._orphanCollections();
```

show execution results

Maniuplating Vertices

Save a vertex

Create a new vertex in vertexCollectionName

`graph.vertexCollectionName.save(data)`

Parameters

- data (required) JSON data of vertex.

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.male.save({name: "Floyd", _key: "floyd"});
```

show execution results

Replace a vertex

Replaces the data of a vertex in collection vertexCollectionName `graph.vertexCollectionName.replace(vertexId, data, options)`

Parameters

- vertexId (required) *_id* attribute of the vertex
- data (required) JSON data of vertex.
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.male.save({neym: "Jon", _key: "john"});
arangosh> graph.male.replace("male/john", {name: "John"});
```

show execution results

Update a vertex

Updates the data of a vertex in collection vertexCollectionName

`graph.vertexCollectionName.update(vertexId, data, options)`

Parameters

- vertexId (required) *_id* attribute of the vertex
- data (required) JSON data of vertex.
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.female.save({name: "Lynda", _key: "linda"});
arangosh> graph.female.update("female/linda", {name: "Linda", _key: "linda"});
```

show execution results

Remove a vertex

Removes a vertex in collection *vertexCollectionName*

`graph.vertexCollectionName.remove(vertexId, options)`

Additionally removes all ingoing and outgoing edges of the vertex recursively (see edge remove).

Parameters

- vertexId (required) *_id* attribute of the vertex
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.male.save({name: "Kermit", _key: "kermit"});
arangosh> db._exists("male/kermit")
arangosh> graph.male.remove("male/kermit")
arangosh> db._exists("male/kermit")
```

show execution results

Manipulating Edges

Save a new edge

Creates an edge from vertex *from* to vertex *to* in collection edgeCollectionName

`graph.edgeCollectionName.save(from, to, data, options)`

Parameters

- from (required) *_id* attribute of the source vertex
- to (required) *_id* attribute of the target vertex
- data (required) JSON data of the edge
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("male/bob", "female/alice", {type: "married", _key: "bobAndA
```

show execution results

If the collections of *from* and *to* are not defined in an edge definition of the graph, the edge will not be stored.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save(
........>  "relation/aliceAndBob",
........>   "female/alice",
........> {type: "married", _key: "bobAndAlice"});
[ArangoError 1906: invalid edge between relation/aliceAndBob and female/alice. Doesn't con
```

Replace an edge

Replaces the data of an edge in collection edgeCollectionName. Note that `_from` and `_to` are mandatory.

`graph.edgeCollectionName.replace(edgeId, data, options)`

Parameters

- edgeId (required) *_id* attribute of the edge
- data (required) JSON data of the edge
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("female/alice", "female/diana", {typo: "nose", _key: "aliceA
arangosh> graph.relation.replace("relation/aliceAndDiana", {type: "knows", _from: "female/
```

show execution results

Update an edge

Updates the data of an edge in collection edgeCollectionName

`graph.edgeCollectionName.update(edgeId, data, options)`

Parameters

- edgeId (required) *_id* attribute of the edge
- data (required) JSON data of the edge
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("female/alice", "female/diana", {type: "knows", _key: "alice
arangosh> graph.relation.update("relation/aliceAndDiana", {type: "quarreled", _key: "alice
```

show execution results

Remove an edge

Removes an edge in collection edgeCollectionName

`graph.edgeCollectionName.remove(edgeId, options)`

If this edge is used as a vertex by another edge, the other edge will be removed (recursively).

Parameters

- edgeId (required) *_id* attribute of the edge
- options (optional) See collection documentation

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("female/alice", "female/diana", {_key: "aliceAndDiana"});
arangosh> db._exists("relation/aliceAndDiana")
arangosh> graph.relation.remove("relation/aliceAndDiana")
arangosh> db._exists("relation/aliceAndDiana")
```

show execution results

Connect edges

Get all connecting edges between 2 groups of vertices defined by the examples

`graph._connectingEdges(vertexExample, vertexExample2, options)`

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertexExample.

Parameters

- vertexExample1 (optional) See Definition of examples
- vertexExample2 (optional) See Definition of examples
- options (optional) An object defining further options. Can have the following values:
  - *edgeExamples*: Filter the edges, see Definition of examples
  - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
  - *vertex1CollectionRestriction* : One or a list of vertex-collection names that should be considered on the intermediate vertex steps.
  - *vertex2CollectionRestriction* : One or a list of vertex-collection names that should be considered on the intermediate vertex steps.

Examples

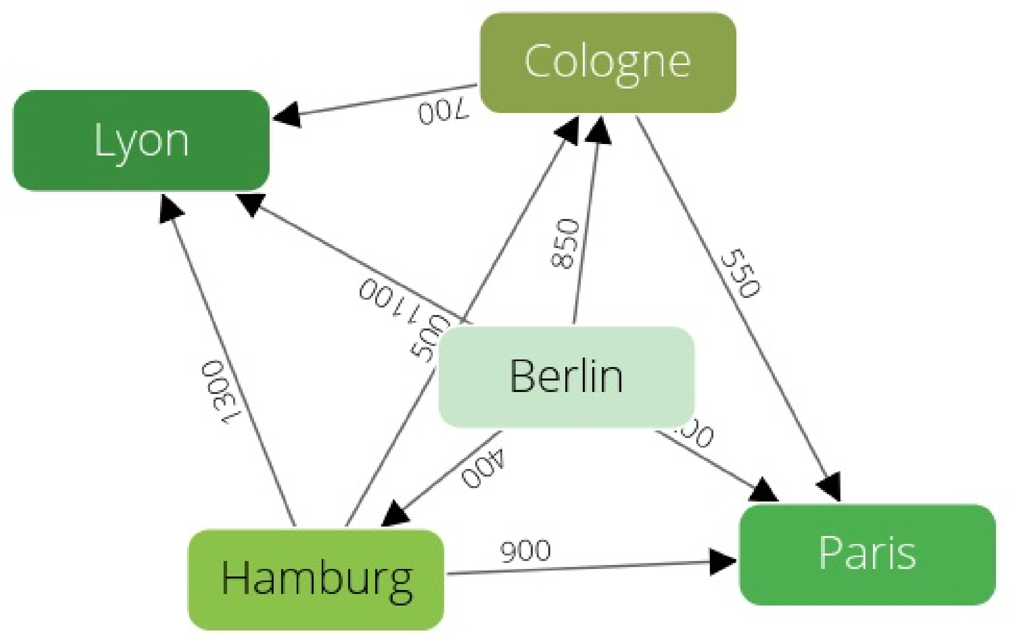
A route planner example, all connecting edges between capitals.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._getConnectingEdges({isCapital : true}, {isCapital : true});
[ ]
```

Graph Functions

This chapter describes various functions on a graph. A lot of these accept a vertex (or edge) example as parameter as defined in the next section.

Examples will explain the API on the the city graph:



Definition of examples

For many of the following functions *examples* can be passed in as a parameter. *Examples* are used to filter the result set for objects that match the conditions. These *examples* can have the following values:

- *null*, there is no matching executed all found results are valid.
- A *string*, only results are returned, which *_id* equal the value of the string
- An example *object*, defining a set of attributes. Only results having these attributes are matched.
- A *list* containing example *objects* and/or *strings*. All results matching at least one of the elements in the list are returned.

Get vertices from edges.

Get vertex *from* of an edge

Get the source vertex of an edge

`graph._fromVertex(edgeId)`

Returns the vertex defined with the attribute *_from* of the edge with *edgeId* as its *_id*.

Parameters

- edgeId (required) *_id* attribute of the edge

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._fromVertex("relation/aliceAndBob")
```

show execution results

Get vertex *to* of an edge

Get the target vertex of an edge

`graph._toVertex(edgeId)`

Returns the vertex defined with the attribute *_to* of the edge with *edgeId* as its *_id*.

Parameters

- edgeId (required) *_id* attribute of the edge

Examples

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._toVertex("relation/aliceAndBob")
```

show execution results

_neighbors

Get all neighbors of the vertices defined by the example

`graph._neighbors(vertexExample, options)`

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertexExample. The complexity of this method is **O(n*m^x)** with *n* being the vertices defined by the parameter vertexExamplex, *m* the average amount of neighbors and *x* the maximal depths. Hence the default call would have a complexity of **O(n*m)**;

Parameters

- vertexExample (optional) See Definition of examples
- options (optional) An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeExamples*: Filter the edges, see Definition of examples
  - *neighborExamples*: Filter the neighbor vertices, see Definition of examples
  - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
  - *vertexCollectionRestriction* : One or a list of vertex-collection names that should be considered on the intermediate vertex steps.
  - *minDepth*: Defines the minimal number of intermediate steps to neighbors (default is 1).
  - *maxDepth*: Defines the maximal number of intermediate steps to neighbors (default is 1).

Examples

A route planner example, all neighbors of capitals.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._neighbors({isCapital : true});
```

show execution results
A route planner example, all outbound neighbors of Hamburg.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._neighbors('germanCity/Hamburg', {direction : 'outbound', maxDepth : 2});
```

show execution results

_commonNeighbors

Get all common neighbors of the vertices defined by the examples.

`graph._commonNeighbors(vertex1Example, vertex2Examples, optionsVertex1, optionsVertex2)`

This function returns the intersection of *graph_module._neighbors(vertex1Example, optionsVertex1)* and *graph_module._neighbors(vertex2Example, optionsVertex2)*. For parameter documentation see _neighbors.

The complexity of this method is **O(n*m^x)** with *n* being the maximal amount of vertices defined by the parameters vertexExamples, *m* the average amount of neighbors and *x* the maximal depths. Hence the default call would have a complexity of **O(n*m)**;

Examples

A route planner example, all common neighbors of capitals.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonNeighbors({isCapital : true}, {isCapital : true});
```

show execution results
A route planner example, all common outbound neighbors of Hamburg with any other location which have a maximal depth of 2 :

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonNeighbors(
........>   'germanCity/Hamburg',
........>   {},
........>   {direction : 'outbound', maxDepth : 2},
........> {direction : 'outbound', maxDepth : 2});
```

show execution results

_countCommonNeighbors

Get the amount of common neighbors of the vertices defined by the examples.

`graph._countCommonNeighbors(vertex1Example, vertex2Examples, optionsVertex1, optionsVertex2)`

Similar to _commonNeighbors but returns count instead of the elements.

Examples

A route planner example, all common neighbors of capitals.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> var example = { isCapital: true };
arangosh> var options = { includeData: true };
arangosh> graph._countCommonNeighbors(example, example, options, options);
```

show execution results

A route planner example, all common outbound neighbors of Hamburg with any other location which have a maximal depth of 2 :

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> var options = { direction: 'outbound', maxDepth: 2, includeData: true };
arangosh> graph._countCommonNeighbors('germanCity/Hamburg', {}, options, options);
```

show execution results

_commonProperties

Get the vertices of the graph that share common properties.

`graph._commonProperties(vertex1Example, vertex2Examples, options)`

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertex1Example and vertex2Example.

The complexity of this method is **O(n)** with *n* being the maximal amount of vertices defined by the parameters vertexExamples.

Parameters

- vertex1Examples (optional) Filter the set of source vertices, see Definition of examples
- vertex2Examples (optional) Filter the set of vertices compared to, see Definition of examples
- options (optional) An object defining further options. Can have the following values:
  - *vertex1CollectionRestriction* : One or a list of vertex-collection names that should be searched for source vertices.
  - *vertex2CollectionRestriction* : One or a list of vertex-collection names that should be searched for compare vertices.
  - *ignoreProperties* : One or a list of attribute names of a document that should be ignored.

Examples

A route planner example, all locations with the same properties:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonProperties({}, {});
```

show execution results

A route planner example, all cities which share same properties except for population.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonProperties({}, {}, {ignoreProperties: 'population'});
```

show execution results

_countCommonProperties

Get the amount of vertices of the graph that share common properties.

`graph._countCommonProperties(vertex1Example, vertex2Examples, options)`

Similar to _commonProperties but returns count instead of the objects.

Examples

A route planner example, all locations with the same properties:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._countCommonProperties({}, {});
```

show execution results
A route planner example, all German cities which share same properties except for population.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._countCommonProperties({}, {}, {vertex1CollectionRestriction : 'germanCity
........> vertex2CollectionRestriction : 'germanCity' ,ignoreProperties: 'population'});
```

show execution results

_paths

The _paths function returns all paths of a graph.

`graph._paths(options)`

This function determines all available paths in a graph.

The complexity of this method is **O(n*n*m)** with *n* being the amount of vertices in the graph and *m* the average amount of connected edges;

Parameters

- options (optional) An object containing options, see below:
  - *direction*: The direction of the edges. Possible values are *any*, *inbound* and *outbound* (default).
  - *followCycles* (optional): If set to *true* the query follows cycles in the graph, default is false.
  - *minLength* (optional): Defines the minimal length a path must have to be returned (default is 0).
  - *maxLength* (optional): Defines the maximal length a path must have to be returned (default is 10).

Examples

Return all paths of the graph "social":

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("social");
arangosh> g._paths();
```

show execution results
Return all inbound paths of the graph "social" with a maximal length of 1 and a minimal length of 2:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("social");
arangosh> g._paths({direction : 'inbound', minLength : 1, maxLength :  2});
```

show execution results

_shortestPath

The _shortestPath function returns all shortest paths of a graph.

`graph._shortestPath(startVertexExample, endVertexExample, options)`

This function determines all shortest paths in a graph. The function accepts an id, an example, a list of examples or even an empty example as parameter for start and end vertex. If one wants to call this function to receive nearly all shortest paths for a graph the option *algorithm* should be set to Floyd-Warshall to increase performance. If no algorithm is provided in the options the function chooses the appropriate one (either Floyd-Warshall or Dijkstra) according to its parameters. The length of a path is by default the amount of edges from one start vertex to an end vertex. The option weight allows the user to define an edge attribute representing the length.

Parameters

- startVertexExample (optional) An example for the desired start Vertices (see Definition of examples).
- endVertexExample (optional) An example for the desired end Vertices (see Definition of examples).
- options (optional) An object containing options, see below:
  - *direction*: The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction*: One or multiple edge collection names. Only edges from these collections will be considered for the path.
  - *startVertexCollectionRestriction*: One or multiple vertex collection names. Only vertices from these collections will be considered as start vertex of a path.
  - *endVertexCollectionRestriction*: One or multiple vertex collection names. Only vertices from these collections will be considered as end vertex of a path.
  - *edgeExamples*: A filter example for the edges in the shortest paths (see example).
  - *algorithm*: The algorithm to calculate the shortest paths. If both start and end vertex examples are empty Floyd-Warshall is used, otherwise the default is Dijkstra
  - *weight*: The name of the attribute of the edges containing the length as a string.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path could not be calculated.

Examples

A route planner example, shortest path from all german to all french cities:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._shortestPath({}, {}, {weight : 'distance', endVertexCollectionRestriction : '
........> startVertexCollectionRestriction : 'germanCity'});
```

show execution results

A route planner example, shortest path from Hamburg and Cologne to Lyon:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._shortestPath([{_id: 'germanCity/Cologne'},{_id: 'germanCity/Munich'}], 'frenc
........> {weight : 'distance'});
```

show execution results

_distanceTo

The _distanceTo function returns all paths and there distance within a graph.

`graph._distanceTo(startVertexExample, endVertexExample, options)`

This function is a wrapper of graph._shortestPath. It does not return the actual path but only the distance between two vertices.

Examples

A route planner example, shortest distance from all german to all french cities:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._distanceTo({}, {}, {weight : 'distance', endVertexCollectionRestriction : 'fr
........> startVertexCollectionRestriction : 'germanCity'});
```

show execution results

A route planner example, shortest distance from Hamburg and Cologne to Lyon:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._distanceTo([{_id: 'germanCity/Cologne'},{_id: 'germanCity/Munich'}], 'frenchC
........> {weight : 'distance'});
```

show execution results

_absoluteEccentricity

Get the eccentricity of the vertices defined by the examples.

`graph._absoluteEccentricity(vertexExample, options)`

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertexExample.

Parameters

- vertexExample (optional) Filter the vertices, see Definition of examples
- options (optional) An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
  - *startVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for source vertices.
  - *endVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for target vertices.
  - *edgeExamples*: Filter the edges to be followed, see Definition of examples
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are Floyd-Warshall and Dijkstra.
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the absolute eccentricity of all locations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteEccentricity({});
```

show execution results

A route planner example, the absolute eccentricity of all locations. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteEccentricity({}, {weight : 'distance'});
```

show execution results

A route planner example, the absolute eccentricity of all cities regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteEccentricity({}, {startVertexCollectionRestriction : 'germanCity'
........> direction : 'outbound', weight : 'distance'});
```

show execution results

_eccentricity

Get the normalized eccentricity of the vertices defined by the examples.

`graph._eccentricity(vertexExample, options)`

Similar to _absoluteEccentricity but returns a normalized result.

Examples

A route planner example, the eccentricity of all locations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._eccentricity();
```

show execution results

A route planner example, the weighted eccentricity.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._eccentricity({weight : 'distance'});
```

show execution results

_absoluteCloseness

Get the closeness of the vertices defined by the examples.

`graph._absoluteCloseness(vertexExample, options)`

The function accepts an id, an example, a list of examples or even an empty example as parameter for *vertexExample*.

Parameters

- vertexExample (optional) Filter the vertices, see Definition of examples
- options (optional) An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
  - *startVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for source vertices.
  - *endVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for target vertices.
  - *edgeExamples*: Filter the edges to be followed, see Definition of examples
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are Floyd-Warshall and Dijkstra.
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the closeness can not be calculated.

Examples

A route planner example, the absolute closeness of all locations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteCloseness({});
```

show execution results

A route planner example, the absolute closeness of all locations. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteCloseness({}, {weight : 'distance'});
```

show execution results

A route planner example, the absolute closeness of all German Cities regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteCloseness({}, {startVertexCollectionRestriction : 'germanCity',
........> direction : 'outbound', weight : 'distance'});
```

show execution results

_closeness

Get the normalized closeness of graphs vertices.

`graph._closeness(options)`

Similar to _absoluteCloseness but returns a normalized value.

Examples

A route planner example, the normalized closeness of all locations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._closeness();
```

show execution results

A route planner example, the closeness of all locations. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._closeness({weight : 'distance'});
```

show execution results

A route planner example, the closeness of all cities regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._closeness({direction : 'outbound', weight : 'distance'});
```

show execution results

_absoluteBetweenness

Get the betweenness of all vertices in the graph.

`graph._absoluteBetweenness(vertexExample, options)`

Parameters

- vertexExample (optional) Filter the vertices, see Definition of examples
- options (optional) An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the betweeness can not be calculated.

Examples

A route planner example, the absolute betweenness of all locations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteBetweenness({});
```

show execution results

A route planner example, the absolute betweenness of all locations. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteBetweenness({weight : 'distance'});
{
}
```

A route planner example, the absolute betweenness of all cities regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteBetweenness({direction : 'outbound', weight : 'distance'});
{
}
```

_betweenness

Get the normalized betweenness of graphs vertices.

`graph_module._betweenness(options)`

Similar to _absoluteBetweeness but returns normalized values.

Examples

A route planner example, the betweenness of all locations.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._betweenness();
```

show execution results

A route planner example, the betweenness of all locations. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._betweenness({weight : 'distance'});
```

show execution results

A route planner example, the betweenness of all cities regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._betweenness({direction : 'outbound', weight : 'distance'});
```

show execution results

_radius

Get the radius of a graph.

`

Parameters

- options (optional) An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are Floyd-Warshall and Dijkstra.
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the radius can not be calculated.

Examples

A route planner example, the radius of the graph.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._radius();
1
```

A route planner example, the radius of the graph. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._radius({weight : 'distance'});
1
```

A route planner example, the radius of the graph regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._radius({direction : 'outbound', weight : 'distance'});
1
```

_diameter

Get the diameter of a graph.

`graph._diameter(graphName, options)`

Parameters

- options (optional) An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are Floyd-Warshall and Dijkstra.
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the radius can not be calculated.

Examples

A route planner example, the diameter of the graph.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._diameter();
1
```

A route planner example, the diameter of the graph. This considers the actual distances.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._diameter({weight : 'distance'});
1
```

A route planner example, the diameter of the graph regarding only outbound paths.

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._diameter({direction : 'outbound', weight : 'distance'});
1
```

SmartGraphs

This feature is only available in the Enterprise Edition.

This chapter describes the smart-graph module. It enables you to manage graphs at scale, it will give a vast performance benefit for all graphs sharded in an ArangoDB Cluster. On a single server this feature is pointless, hence it is only available in a cluster mode. In terms of querying there is no difference between smart and General Graphs. The former are a transparent replacement for the latter. So for querying the graph please refer to AQL Graph Operations and Graph Functions sections. The optimizer is clever enough to identify if we are on a SmartGraph or not.

The difference is only in the management section: creating and modifying the underlying collections of the graph. For a detailed API reference please refer to SmartGraph Management.

What makes a graph smart?

Most graphs have one feature that divides the entire graph into several smaller subgraphs. These subgraphs have a large amount of edges that only connect vertices in the same subgraph and only have few edges connecting vertices from other subgraphs. Examples for these graphs are:

- Social Networks

  Typically the feature here is the region/country users live in. Every user typicalliy has more contacts in the same region/country then she has in other regions/countries

- Transport Systems

  For those also the feature is the region/country. You have many local transportion but only few accross countries.

- E-Commerce

  In this case probably the category of products is a good feature. Often products of the same category are bought together.

If this feature is known, SmartGraphs can make use if it. When creating a SmartGraph you have to define a smartAttribute, which is the name of an attribute stored in every vertex. The graph will than be automatically sharded in such a way that all vertices with the same value are stored on the same physical machine, all edges connecting vertices with identical smartAttribute values are stored on this machine as well. During query time the query optimizer and the query executor both know for every document exactly where it is stored and can thereby minimize network overhead. Everything that can be computed locally will be computed locally.

Benefits of SmartGraphs

Because of the above described guaranteed sharding, the performance of queries that only cover one subgraph have a performance almost equal to an only local computation. Queries that cover more than one subgraph require some network overhead. The more subgraphs are touched the more network cost will apply. However the overall performance is never worse than the same query on a General Graph.

Getting started

First of all SmartGraphs *cannot use existing collections*, when switching to SmartGraph from an existing data set you have to import the data into a fresh SmartGraph. This switch can be easily achieved with arangodump and arangorestore. The only thing you have to change in this pipeline is that you create the new collections with the SmartGraph before starting `arangorestore`.

- Create a graph

  In comparison to General Graph we have to add more options when creating the graph. The two options `smartGraphAttribute` and `numberOfShards` are required and cannot be modifed later.

  ```
  arangosh> var graph_module = require("@arangodb/smart-graph");
  arangosh> var graph = graph_module._create("myGraph", [], [], {smartGraphAttribute: "region", numberOfShards: 9});
  arangosh> graph;
  [ SmartGraph myGraph EdgeDefinitions: [ ] VertexCollections: [ ] ]
  ```

- Add some vertex collections

  This is again identical to General Graph. The module will setup correct sharding for all these collections. *Note*: The collections have to be new.

  ```
  arangosh> graph._addVertexCollection("shop");
  arangosh> graph._addVertexCollection("customer");
  arangosh> graph._addVertexCollection("pet");
  arangosh> graph;
  [ SmartGraph myGraph EdgeDefinitions: [ ] VertexCollections: [ "shop", "customer", "pet" ] ]
  ```

- Define relations on the Graph

  ```
  arangosh> var rel = graph_module._relation("isCustomer", ["shop"], ["customer"]);
  arangosh> graph._extendEdgeDefinitions(rel);
  arangosh> graph;
  [ SmartGraph myGraph EdgeDefinitions: [   "isCustomer: [shop] -> [customer]" ] VertexCollections: [ "pet" ] ]
  ```

Smart Graph Management

This chapter describes the JavaScript interface for creating and modifying SmartGraphs. At first you have to note that every SmartGraph is a specialized version of a General Graph, which means all of the General Graph functionality is available on a SmartGraph as well. The major difference of both modules is handling of the underlying collections, the General Graph does not enforce or maintain any sharding of the collections and can therefor combine arbitrary sets of existing collections. SmartGraphs enforce and rely on a special sharding of the underlying collections and hence can only work with collections that are created through the SmartGraph itself. This also means that SmartGraphs cannot be overlapping, a collection can either be sharded for one SmartGraph or for the other. If you need to make sure that all queries can be executed with SmartGraph performance, just create one large SmartGraph covering everything and query it stating the subset of edge collections explicitly. To generally understand the concept of this module please read the chapter about General Graph Management first. In the following we will only describe the overloaded functionality. Everything else works identical in both modules.

Create a graph

Also SmartGraphs require edge relations to be created, the format of the relations is identical. The only difference is that all collections used within the relations to create a new SmartGraph cannot exist yet. They have to be created by the Graph in order to enforce the correct sharding.

Create a graph

`graph_module._create(graphName, edgeDefinitions, orphanCollections, smartOptions)`

The creation of a graph requires the name and some SmartGraph options. Due to the API `edgeDefinitions` and `orphanCollections` have to be given, but both can be empty arrays and can be created later. The `edgeDefinitions` can be created using the convenience method `_relation` known from the `general-graph` module, which is also available here. `orphanCollections` again is just a list of additional vertex collections which are not yet connected via edges but should follow the same sharding to be connected later on. All collections used within the creation process are newly created. The process will fail if one of them already exists. All newly created collections will immediately be dropped again in the failed case.

Parameters

- graphName (required) Unique identifier of the graph
- edgeDefinitions (required) List of relation definition objects, may be empty
- orphanCollections (required) List of additional vertex collection names, may be empty
- smartOptions (required) A JSON object having the following keys:
  - numberOfShards (required) The number of shards that will be created for each collection. To maintain the correct sharding all collections need an identical number of shards. This cannot be modified after creation of the graph.
  - smartGraphAttribute (required) The attribute that will be used for sharding. All vertices are required to have this attribute set and it has to be a string. Edges derive the attribute from their connected vertices.

Examples

Create an empty graph, edge definitions can be added at runtime:

```
arangosh> var graph_module = require("@arangodb/smart-graph");
arangosh> var graph = graph_module._create("myGraph", [], [], {smartGraphAttribute: "region", numberOfShards: 9});
[ SmartGraph myGraph EdgeDefinitions: [ ] VertexCollections: [ ] ]
```

Create a graph using an edge collection `edges` and a single vertex collection `vertices`

```
arangosh> var graph_module = require("@arangodb/smart-graph");
arangosh> var edgeDefinitions = [ graph_module._relation("edges", "vertices", "vertices") ];
arangosh> var graph = graph_module._create("myGraph", edgeDefinitions, [], {smartGraphAttribute: "region", numberOfShard
[ SmartGraph myGraph EdgeDefinitions: [ "edges: [vertices] -> [vertices]" ] VertexCollections: [ ] ]
```

Create a graph with edge definitions and orphan collections:

```
arangosh> var graph_module = require("@arangodb/smart-graph");
arangosh> var edgeDefinitions = [ graph_module._relation("myRelation", ["male", "female"], ["male", "female"]) ];
arangosh> var graph = graph_module._create("myGraph", edgeDefinitions, ["sessions"], {smartGraphAttribute: "region", num
[ Graph myGraph EdgeDefinitions: [
  "myRelation: [female, male] -> [female, male]"
] VertexCollections: [
  "sessions"
] ]
```

Modify a graph definition during runtime

After you have created a SmartGraph its definition is also not immutable. You can still add or remove relations. This is again identical to General Graphs. However there is one important difference: You can only add collections that either *do not exist*, or that have been created by this graph earlier. The later can be achieved if you for example remove an orphan collection from this graph, without dropping the collection itself. Than after some time you decide to add it again, it can be used. This is because the enforced sharding is still applied to this vertex collection, hence it is suitable to be added again.

Remove a vertex collection

Remove a vertex collection from the graph

`graph._removeVertexCollection(vertexCollectionName, dropCollection)`

In most cases this function works identically to the General Graph one. But there is one special case: The first vertex collection added to the graph (either orphan or within a relation) defines the sharding for all collections within the graph. This collection can never be removed from the graph.

Parameters

- vertexCollectionName (required) Name of vertex collection.
- dropCollection (optional) If true the collection will be dropped if it is not used in any other graph. Default: false.

Examples

The following example shows that you cannot drop the initial collection. You have to drop the complete graph. If you just want to get rid of the data `truncate` it.

```
  arangosh> var graph_module = require("@arangodb/smart-graph")
  arangosh> var relation = graph_module._relation("edges", "vertices", "vertices");
  arangosh> var graph = graph_module._create("myGraph", [relation], ["other"], {smartGraphAttribute: "region", numberOfSh
  arangosh> graph._orphanCollections();
  [
    "other"
  ]
  arangosh> graph._deleteEdgeDefinition("edges");
  arangosh> graph._orphanCollections();
  [
    "vertices",
    "other"
  ]
  arangosh> graph._removeVertexCollection("other");
  arangosh> graph._orphanCollections();
  [
    "vertices"
  ]
  arangosh> graph._removeVertexCollection("vertices");
  ArangoError 4002: cannot drop this smart collection
```

Traversals

ArangoDB provides several ways to query graph data. Very simple operations can be composed with the low-level edge methods *edges*, *inEdges*, and *outEdges* for edge collections. These work on named and anonymous graphs. For more complex operations, ArangoDB provides predefined traversal objects.

Also Traversals have been added to AQL. Please read the chapter about AQL traversersals before you continue reading here. Most of the traversal cases are covered by AQL and will be executed in an optimized way. Only if the logic for your is too complex to be defined using AQL filters you can use the traversal object defined here which gives you complete programmatic access to the data.

For any of the following examples, we'll be using the example collections *v* and *e*, populated with continents, countries and capitals data listed below (see Example Data).

Starting from Scratch

ArangoDB provides the *edges*, *inEdges*, and *outEdges* methods for edge collections. These methods can be used to quickly determine if a vertex is connected to other vertices, and which. This functionality can be exploited to write very simple graph queries in JavaScript.

For example, to determine which edges are linked to the *world* vertex, we can use *inEdges*:

```
db.e.inEdges('v/world').forEach(function(edge) {
  require("@arangodb").print(edge._from, "->", edge.type, "->", edge._to);
});
```

inEdges will give us all ingoing edges for the specified vertex *v/world*. The result is a JavaScript array that we can iterate over and print the results:

```
v/continent-africa -> is-in -> v/world
v/continent-south-america -> is-in -> v/world
v/continent-asia -> is-in -> v/world
v/continent-australia -> is-in -> v/world
v/continent-europe -> is-in -> v/world
v/continent-north-america -> is-in -> v/world
```

Note: *edges*, *inEdges*, and *outEdges* return an array of edges. If we want to retrieve the linked vertices, we can use each edges' *_from* and *_to* attributes as follows:

```
db.e.inEdges('v/world').forEach(function(edge) {
  require("@arangodb").print(db._document(edge._from).name, "->", edge.type, "->", db._document(edge._to).name);
});
```

We are using the *document* method from the *db* object to retrieve the connected vertices now.

While this may be sufficient for one-level graph operations, writing a traversal by hand may become too complex for multi-level traversals.

Getting started

To use a traversal object, we first need to require the *traversal* module:

```
var traversal = require("@arangodb/graph/traversal");
var examples = require("@arangodb/graph-examples/example-graph.js");
examples.loadGraph("worldCountry");
```

We then need to setup a configuration for the traversal and determine at which vertex to start the traversal:

```
var config = {
  datasource: traversal.generalGraphDatasourceFactory("worldCountry"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander,
  maxDepth: 1
};

var startVertex = db._document("v/world");
```

Note: The startVertex needs to be a document, not only a document id.

We can then create a traverser and start the traversal by calling its *traverse* method. Note that *traverse* needs a *result* object, which it can modify in place:

```
var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};
var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);
```

Finally, we can print the contents of the *results* object, limited to the visited vertices. We will only print the name and type of each visited vertex for brevity:

```
require("@arangodb").print(result.visited.vertices.map(function(vertex) {
  return vertex.name + " (" + vertex.type + ")";
}));
```

The full script, which includes all steps carried out so far is thus:

```
var traversal = require("@arangodb/graph/traversal");

var config = {
  datasource: traversal.generalGraphDatasourceFactory("worldCountry"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander,
  maxDepth: 1
};

var startVertex = db._document("v/world");
var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);

require("@arangodb").print(result.visited.vertices.map(function(vertex) {
  return vertex.name + " (" + vertex.type + ")";
}));
```

The result is an array of vertices that were visited during the traversal, starting at the start vertex (i.e. *v/world* in our example):

```
[
  "World (root)",
  "Africa (continent)",
  "Asia (continent)",
  "Australia (continent)",
  "Europe (continent)",
  "North America (continent)",
  "South America (continent)"
]
```

Note: The result is limited to vertices directly connected to the start vertex. We achieved this by setting the *maxDepth* attribute to *1*. Not setting it would return the full array of vertices.

Traversal Direction

For the examples contained in this manual, we'll be starting the traversals at vertex *v/world*. Vertices in our graph are connected like this:

```
v/world <- is-in <- continent (Africa) <- is-in <- country (Algeria) <- is-in <- capital (Algiers)
```

To get any meaningful results, we must traverse the graph in **inbound** order. This means, we'll be following all incoming edges of to a vertex. In the traversal configuration, we have specified this via the *expander* attribute:

```
var config = {
  ...
  expander: traversal.inboundExpander
};
```

For other graphs, we might want to traverse via the **outgoing** edges. For this, we can use the *outboundExpander*. There is also an *anyExpander*, which will follow both outgoing and incoming edges. This should be used with care and the traversal should always be limited to a maximum number of iterations (e.g. using the *maxIterations* attribute) in order to terminate at some point.

To invoke the default outbound expander for a graph, simply use the predefined function:

```
var config = {
  ...
  expander: traversal.outboundExpander
};
```

Please note the outbound expander will not produce any output for the examples if we still start the traversal at the *v/world* vertex.

Still, we can use the outbound expander if we start somewhere else in the graph, e.g.

```
var traversal = require("@arangodb/graph/traversal");

var config = {
  datasource: traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.outboundExpander
};

var startVertex = db._document("v/capital-algiers");
var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);

require("@arangodb").print(result.visited.vertices.map(function(vertex) {
  return vertex.name + " (" + vertex.type + ")";
}));
```

The result is:

```
[
  "Algiers (capital)",
  "Algeria (country)",
  "Africa (continent)",
  "World (root)"
]
```

which confirms that now we're going outbound.

Traversal Strategy

Depth-first traversals

The visitation order of vertices is determined by the *strategy* and *order* attributes set in the configuration. We chose *depthfirst* and *preorder*, meaning the traverser will visit each vertex **before** handling connected edges (pre-order), and descend into any connected edges before processing other vertices on the same level (depth-first).

Let's remove the *maxDepth* attribute now. We'll now be getting all vertices (directly and indirectly connected to the start vertex):

```
var config = {
  datasource: traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander
};

var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);

require("@arangodb").print(result.visited.vertices.map(function(vertex) {
  return vertex.name + " (" + vertex.type + ")";
}));
```

The result will be a longer array, assembled in depth-first, pre-order order. For each continent found, the traverser will descend into linked countries, and then into the linked capital:

```
[
  "World (root)",
  "Africa (continent)",
  "Algeria (country)",
  "Algiers (capital)",
  "Angola (country)",
  "Luanda (capital)",
  "Botswana (country)",
  "Gaborone (capital)",
  "Burkina Faso (country)",
  "Ouagadougou (capital)",
  ...
]
```

Let's switch the *order* attribute from *preorder* to *postorder*. This will make the traverser visit vertices **after** all connected vertices were visited (i.e. most distant vertices will be emitted first):

```
[
  "Algiers (capital)",
  "Algeria (country)",
  "Luanda (capital)",
  "Angola (country)",
  "Gaborone (capital)",
  "Botswana (country)",
  "Ouagadougou (capital)",
  "Burkina Faso (country)",
  "Bujumbura (capital)",
  "Burundi (country)",
  "Yaounde (capital)",
  "Cameroon (country)",
  "N'Djamena (capital)",
  "Chad (country)",
  "Yamoussoukro (capital)",
  "Cote d'Ivoire (country)",
  "Cairo (capital)",
  "Egypt (country)",
  "Asmara (capital)",
  "Eritrea (country)",
  "Africa (continent)",
  ...
]
```

Breadth-first traversals

If we go back to *preorder*, but change the strategy to *breadth-first* and re-run the traversal, we'll see that the return order changes, and items on the same level will be returned adjacently:

```
[
  "World (root)",
  "Africa (continent)",
  "Asia (continent)",
  "Australia (continent)",
  "Europe (continent)",
  "North America (continent)",
  "South America (continent)",
  "Burkina Faso (country)",
  "Burundi (country)",
  "Cameroon (country)",
  "Chad (country)",
  "Algeria (country)",
  "Angola (country)",
  ...
]
```

Note: The order of items returned for the same level is undefined. This is because there is no natural order of edges for a vertex with multiple connected edges. To explicitly set the order for edges on the same level, you can specify an edge comparator function with the *sort* attribute:

```
var config = {
  ...
  sort: function (l, r) { return l._key < r._key ? 1 : -1; }
  ...
};
```

The arguments l and r are edge documents. This will traverse edges of the same vertex in backward *_key* order:

```
[
  "World (root)",
  "South America (continent)",
  "North America (continent)",
  "Europe (continent)",
  "Australia (continent)",
  "Asia (continent)",
  "Africa (continent)",
  "Ecuador (country)",
  "Colombia (country)",
  "Chile (country)",
  "Brazil (country)",
  "Bolivia (country)",
  "Argentina (country)",
  ...
]
```

Note: This attribute only works for the usual expanders *traversal.inboundExpander*, *traversal.outboundExpander*, *traversal.anyExpander* and their corresponding "WithLabels" variants. If you are using custom expanders you have to organize the sorting within the specified expander.

Writing Custom Visitors

So far we have used much of the traverser's default functions. The traverser is very configurable and many of the default functions can be overridden with custom functionality.

For example, we have been using the default visitor function (which is always used if the configuration does not contain the *visitor* attribute). The default visitor function is called for each vertex in a traversal, and will push it into the result. This is the reason why the *result* variable looked different after the traversal, and needed to be initialized before the traversal was started.

Note that the default visitor (named `trackingVisitor`) will add every visited vertex into the result, including the full paths from the start vertex. This is useful for learning and debugging purposes, but should be avoided in production because it might produce (and copy) huge amounts of data. Instead, only those data should be copied into the result that are actually necessary.

The traverser comes with the following predefined visitors:

- *trackingVisitor*: this is the default visitor. It will copy all data of each visited vertex plus the full path information into the result. This can be slow if the result set is huge or vertices contain a lot of data.
- *countingVisitor*: this is a very lightweight visitor: all it does is increase a counter in the result for each vertex visited. Vertex data and paths will not be copied into the result.
- *doNothingVisitor*: if no action shall be carried out when a vertex is visited, this visitor can be employed. It will not do anything and will thus be fast. It can be used for performance comparisons with other visitors.

We can also write our own visitor function if we want to. The general function signature for visitor functions is as follows:

```
var config = {
  ...
  visitor: function (config, result, vertex, path, connected) { ... }
};
```

Note: the *connected* parameter value will only be set if the traversal order is set to *preorder-expander*. Otherwise, this parameter won't be set by the traverser.

Visitor functions are not expected to return any values. Instead, they can modify the *result* variable (e.g. by pushing the current vertex into it), or do anything else. For example, we can create a simple visitor function that only prints information about the current vertex as we traverse:

```
var config = {
  datasource: traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander,
  visitor: function (config, result, vertex, path) {
    require("@arangodb").print("visiting vertex", vertex.name);
  }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(undefined, startVertex);
```

To write a visitor that increments a counter each time a vertex is visited, we could write the following custom visitor:

```
config.visitor = function (config, result, vertex, path, connected) {
  if (! result) {
    result = { };
  }

  if (! result.hasOwnProperty('count')) {
    result.count = 0;
  }

  ++result.count;
}
```

Note that such visitor is already predefined (it's the countingVisitor described above). It can be used as follows:

```
config.visitor = traversal.countingVisitor;
```

Another example of a visitor is one that collects the `_id` values of all vertices visited:

```
config.visitor = function (config, result, vertex, path, connected) {
  if (! result) {
    result = { };
  }
  if (! result.hasOwnProperty("visited")) {
    result.visited = { vertices: [ ] };
  }

  result.visited.vertices.push(vertex._id);
}
```

When the traversal order is set to *preorder-expander*, the traverser will pass a fifth parameter value into the visitor function. This parameter contains the connected edges of the visited vertex as an array. This can be handy because in this case the visitor will get all information about the vertex and the connected edges together.

For example, the following visitor can be used to print only leaf nodes (that do not have any further connected edges):

```
config.visitor = function (config, result, vertex, path, connected) {
  if (connected && connected.length === 0) {
    require("@arangodb").print("found a leaf-node: ", vertex);
  }
}
```

Note that for this visitor to work, the traversal *order* attribute needs to be set to the value *preorder-expander*.

Filtering Vertices and Edges

Filtering Vertices

So far we have printed or returned all vertices that were visited during the traversal. This is not always required. If the result shall be restrict to just specific vertices, we can use a filter function for vertices. It can be defined by setting the *filter* attribute of a traversal configuration, e.g.:

```
var config = {
  filter: function (config, vertex, path) {
    if (vertex.type !== 'capital') {
      return 'exclude';
    }
  }
}
```

The above filter function will exclude all vertices that do not have a *type* value of *capital*. The filter function will be called for each vertex found during the traversal. It will receive the traversal configuration, the current vertex, and the full path from the traversal start vertex to the current vertex. The path consists of an array of edges, and an array of vertices. We could also filter everything but capitals by checking the length of the path from the start vertex to the current vertex. Capitals will have a distance of 3 from the *v/world* start vertex (capital → is-in → country → is-in → continent → is-in → world):

```
var config = {
  ...
  filter: function (config, vertex, path) {
    if (path.edges.length < 3) {
      return 'exclude';
    }
  }
}
```

Note: If a filter function returns nothing (or *undefined*), the current vertex will be included, and all connected edges will be followed. If a filter function returns *exclude* the current vertex will be excluded from the result, and all still all connected edges will be followed. If a filter function returns *prune*, the current vertex will be included, but no connected edges will be followed.

For example, the following filter function will not descend into connected edges of continents, limiting the depth of the traversal. Still, continent vertices will be included in the result:

```
var config = {
  ...
  filter: function (config, vertex, path) {
    if (vertex.type === 'continent') {
      return 'prune';
    }
  }
}
```

It is also possible to combine *exclude* and *prune* by returning an array with both values:

```
return [ 'exclude', 'prune' ];
```

Filtering Edges

It is possible to exclude certain edges from the traversal. To filter on edges, a filter function can be defined via the *expandFilter* attribute. The *expandFilter* is a function which is called for each edge during a traversal.

It will receive the current edge (*edge* variable) and the vertex which the edge connects to (in the direction of the traversal). It also receives the current path from the start vertex up to the current vertex (excluding the current edge and the vertex the edge points to).

If the function returns *true*, the edge will be followed. If the function returns *false*, the edge will not be followed. Here is a very simple custom edge filter function implementation, which simply includes edges if the (edges) path length is less than 1, and will exclude any other edges. This will effectively terminate the traversal after the first level of edges:

```
var config = {
  ...
  expandFilter: function (config, vertex, edge, path) {
    return (path.edges.length < 1);
  }
};
```

Writing Custom Expanders

The edges connected to a vertex are determined by the expander. So far we have used a default expander (the default inbound expander to be precise). The default inbound expander simply enumerates all connected ingoing edges for a vertex, based on the edge collection specified in the traversal configuration.

There is also a default outbound expander, which will enumerate all connected outgoing edges. Finally, there is an any expander, which will follow both ingoing and outgoing edges.

If connected edges must be determined in some different fashion for whatever reason, a custom expander can be written and registered by setting the *expander* attribute of the configuration. The expander function signature is as follows:

```
var config = {
  ...
  expander: function (config, vertex, path) { ... }
}
```

It is the expander's responsibility to return all edges and vertices directly connected to the current vertex (which is passed via the *vertex* variable). The full path from the start vertex up to the current vertex is also supplied via the *path* variable. An expander is expected to return an array of objects, which need to have an *edge* and a *vertex* attribute each.

Note: If you want to rely on a particular order in which the edges are traversed, you have to sort the edges returned by your expander within the code of the expander. The functions to get outbound, inbound or any edges from a vertex do not guarantee any particular order!

A custom implementation of an inbound expander could look like this (this is a non-deterministic expander, which randomly decides whether or not to include connected edges):

```
var config = {
  ...
  expander: function (config, vertex, path) {
    var connected = [ ];
    var datasource = config.datasource;
    datasource.getInEdges(vertex._id).forEach(function (edge) {
      if (Math.random() >= 0.5) {
        connected.push({ edge: edge, vertex: (edge._from) });
      }
    });
    return connected;
  }
};
```

A custom expander can also be used as an edge filter because it has full control over which edges will be returned.

Following are two examples of custom expanders that pick edges based on attributes of the edges and the connected vertices.

Finding the connected edges / vertices based on an attribute *when* in the connected vertices. The goal is to follow the edge that leads to the vertex with the highest value in the *when* attribute:

```
var config = {
  ...
  expander: function (config, vertex, path) {
    var datasource = config.datasource;
    // determine all outgoing edges
    var outEdges = datasource.getOutEdges(vertex);

    if (outEdges.length === 0) {
      return [ ];
    }

    var data = [ ];
    outEdges.forEach(function (edge) {
      data.push({ edge: edge, vertex: datasource.getInVertex(edge) });
    });

    // sort outgoing vertices according to "when" attribute value
    data.sort(function (l, r) {
      if (l.vertex.when === r.vertex.when) {
        return 0;
      }

      return (l.vertex.when < r.vertex.when ? 1 : -1);
    });

    // pick first vertex found (with highest "when" attribute value)
    return [ data[0] ];
  }
  ...
};
```

Finding the connected edges / vertices based on an attribute *when* in the edge itself. The goal is to pick the one edge (out of potentially many) that has the highest *when* attribute value:

```
var config = {
  ...
  expander: function (config, vertex, path) {
    var datasource = config.datasource;
    // determine all outgoing edges
    var outEdges = datasource.getOutEdges(vertex);

    if (outEdges.length === 0) {
      return [ ]; // return an empty array
    }

    // sort all outgoing edges according to "when" attribute
    outEdges.sort(function (l, r) {
      if (l.when === r.when) {
        return 0;
      }
      return (l.when < r.when ? -1 : 1);
    });

    // return first edge (the one with highest "when" value)
    var edge = outEdges[0];
    try {
      var v = datasource.getInVertex(edge);
      return [ { edge: edge, vertex: v } ];
    }
    catch (e) { }

    return [ ];
  }
  ...
};
```

Handling Uniqueness

Graphs may contain cycles. To be on top of what happens when a traversal encounters a vertex or an edge it has already visited, there are configuration options.

The default configuration is to visit every vertex, regardless of whether it was already visited in the same traversal. However, edges will by default only be followed if they are not already present in the current path.

Imagine the following graph which contains a cycle:

```
A -> B -> C -> A
```

When the traversal finds the edge from *C* to *A*, it will by default follow it. This is because we have not seen this edge yet. It will also visit vertex *A* again. This is because by default all vertices will be visited, regardless of whether already visited or not.

However, the traversal will not again following the outgoing edge from *A* to *B*. This is because we already have the edge from *A* to *B* in our current path.

These default settings will prevent infinite traversals.

To adjust the uniqueness for visiting vertices, there are the following options for *uniqueness.vertices*:

- *"none"*: always visit a vertices, regardless of whether it was already visited or not
- *"global"*: visit a vertex only if it was not visited in the traversal
- *"path"*: visit a vertex if it is not included in the current path

To adjust the uniqueness for following edges, there are the following options for *uniqueness.edges*:

- *"none"*: always follow an edge, regardless of whether it was followed before
- *"global"*: follow an edge only if it wasn't followed in the traversal
- *"path"*: follow an edge if it is not included in the current path

Note that uniqueness checking will have some effect on both runtime and memory usage. For example, when uniqueness checks are set to *"global"*, arrays of visited vertices and edges must be kept in memory while the traversal is executed. Global uniqueness should thus only be used when a traversal is expected to visit few nodes.

In terms of runtime, turning off uniqueness checks (by setting both options to *"none"*) is the best choice, but it is only safe for graphs that do not contain cycles. When uniqueness checks are deactivated in a graph with cycles, the traversal might not abort in a sensible amount of time.

Optimizations

There are a few options for making a traversal run faster.

The best option is to make the amount of visited vertices and followed edges as small as possible. This can be achieved by writing custom filter and expander functions. Such functions should only include vertices of interest, and only follow edges that might be interesting.

Traversal depth can also be bounded with the *minDepth* and *maxDepth* options.

Another way to speed up traversals is to write a custom visitor function. The default visitor function (*trackingVisitor*) will copy every visited vertex into the result. If vertices contain lots of data, this might be expensive. It is therefore recommended to only copy such data into the result that is actually needed. The default visitor function will also copy the full path to the visited document into the result. This is even more expensive and should be avoided if possible.

If the goal of a traversal is to only count the number of visited vertices, the prefab *countingVisitor*
will be much more efficient than the default visitor.

For graphs that are known to not contain any cycles, uniqueness checks should be turned off. This can achieved via the *uniqueness* configuration options. Note that uniqueness checks should not be turned off for graphs that are known contain cycles or if there is no information about the graph's structure.

By default, a traversal will only process a limited number of vertices. This is protect the user from unintentionally run a never-ending traversal on a graph with cyclic data. How many vertices will be processed at most is determined by the *maxIterations* configuration option. If a traversal hits the cap specified by *maxIterations*, it will abort and throw a *too many iterations* exception. If this error is encountered, the *maxIterations* value should be increased if it is made sure that the other traversal configuration parameters are sane and the traversal will abort naturally at some point.

Finally, the *buildVertices* configuration option can be set to *false* to avoid looking up and fully constructing vertex data. If all that's needed from vertices are the *_id* or *_key* attributes, the *buildvertices* option can be set to *false*. If visitor, filter or expandFilter functions need to access other vertex attributes, the option should not be changed.

Configuration Overview

This section summarizes the configuration attributes for the traversal object. The configuration can consist of the following attributes:

- *visitor*: visitor function for vertices. It will be called for all non-excluded vertices. The general visitor function signature is *function (config, result, vertex, path)*. If the traversal order is *preorder-expander*, the connecting edges of the visited vertex will be passed as the fifth parameter, extending the function signature to: *function (config, result, vertex, path, edges)*.

  Visitor functions are not expected to return values, but they may modify the *result* variable as needed (e.g. by pushing vertex data into the result).

- *expander*: expander function that is responsible for returning edges and vertices directly connected to a vertex. The function signature is *function (config, vertex, path)*. The expander function is required to return an array of connection objects, consisting of an *edge* and *vertex* attribute each. If there are no connecting edges, the expander is expected to return an empty array.
- *filter*: vertex filter function. The function signature is *function (config, vertex, path)*. It may return one of the following values:
  - *undefined*: vertex will be included in the result and connected edges will be traversed
  - *"exclude"*: vertex will not be included in the result and connected edges will be traversed
  - *"prune"*: vertex will be included in the result but connected edges will not be traversed
  - [*"prune"*, *"exclude"*]: vertex will not be included in the result and connected edges will not be returned
- *expandFilter*: filter function applied on each edge/vertex combination determined by the expander. The function signature is *function (config, vertex, edge, path)*. The function should return *true* if the edge/vertex combination should be processed, and *false* if it should be ignored.
- *sort*: a filter function to determine the order in which connected edges are processed. The function signature is *function (l, r)*. The function is required to return one of the following values:
  - *-1* if *l* should have a sort value less than *r*

  - *1* if *l* should have a higher sort value than *r*
  - *0* if *l* and *r* have the same sort value
- *strategy*: determines the visitation strategy. Possible values are *depthfirst* and *breadthfirst*.
- *order*: determines the visitation order. Possible values are *preorder*, *postorder*, and *preorder-expander*. *preorder-expander* is the same as *preorder*, except that the signature of the *visitor* function will change as described above.
- *itemOrder*: determines the order in which connections returned by the expander will be processed. Possible values are *forward* and *backward*.
- *maxDepth*: if set to a value greater than *0*, this will limit the traversal to this maximum depth.
- *minDepth*: if set to a value greater than *0*, all vertices found on a level below the *minDepth* level will not be included in the result.
- *maxIterations*: the maximum number of iterations that the traversal is allowed to perform. It is sensible to set this number so unbounded traversals will terminate at some point.
- *uniqueness*: an object that defines how repeated visitations of vertices should be handled. The *uniqueness* object can have a sub-attribute *vertices*, and a sub-attribute *edges*. Each sub-attribute can have one of the following values:
  - *"none"*: no uniqueness constraints
  - *"path"*: element is excluded if it is already contained in the current path. This setting may be sensible for graphs that contain cycles (e.g. A → B → C → A).
  - *"global"*: element is excluded if it was already found/visited at any point during the traversal.
- *buildVertices*: this attribute controls whether vertices encountered during the traversal will be looked up in the database and will be made available to visitor, filter, and expandFilter functions. By default, vertices will be looked up and made available. However, there are some special use cases when fully constructing vertex objects is not necessary and can be avoided. For example, if a traversal is meant to only count the number of visited vertices but do not read any data from vertices, this option might be set to *true*.

Example Data

The following examples all use a vertex collection *v* and an edge collection *e*. The vertex collection *v* contains continents, countries, and capitals. The edge collection *e* contains connections between continents and countries, and between countries and capitals.

To set up the collections and populate them with initial data, the following script was used:

```
db._create("v");
db._createEdgeCollection("e");

// vertices: root node
db.v.save({ _key: "world", name: "World", type: "root" });

// vertices: continents
db.v.save({ _key: "continent-africa", name: "Africa", type: "continent" });
db.v.save({ _key: "continent-asia", name: "Asia", type: "continent" });
db.v.save({ _key: "continent-australia", name: "Australia", type: "continent" });
db.v.save({ _key: "continent-europe", name: "Europe", type: "continent" });
db.v.save({ _key: "continent-north-america", name: "North America", type: "continent" });
db.v.save({ _key: "continent-south-america", name: "South America", type: "continent" });

// vertices: countries
db.v.save({ _key: "country-afghanistan", name: "Afghanistan", type: "country", code: "AFG" });
db.v.save({ _key: "country-albania", name: "Albania", type: "country", code: "ALB" });
db.v.save({ _key: "country-algeria", name: "Algeria", type: "country", code: "DZA" });
db.v.save({ _key: "country-andorra", name: "Andorra", type: "country", code: "AND" });
db.v.save({ _key: "country-angola", name: "Angola", type: "country", code: "AGO" });
db.v.save({ _key: "country-antigua-and-barbuda", name: "Antigua and Barbuda", type: "country", code: "ATG" });
db.v.save({ _key: "country-argentina", name: "Argentina", type: "country", code: "ARG" });
db.v.save({ _key: "country-australia", name: "Australia", type: "country", code: "AUS" });
db.v.save({ _key: "country-austria", name: "Austria", type: "country", code: "AUT" });
db.v.save({ _key: "country-bahamas", name: "Bahamas", type: "country", code: "BHS" });
db.v.save({ _key: "country-bahrain", name: "Bahrain", type: "country", code: "BHR" });
db.v.save({ _key: "country-bangladesh", name: "Bangladesh", type: "country", code: "BGD" });
db.v.save({ _key: "country-barbados", name: "Barbados", type: "country", code: "BRB" });
db.v.save({ _key: "country-belgium", name: "Belgium", type: "country", code: "BEL" });
db.v.save({ _key: "country-bhutan", name: "Bhutan", type: "country", code: "BTN" });
db.v.save({ _key: "country-bolivia", name: "Bolivia", type: "country", code: "BOL" });
db.v.save({ _key: "country-bosnia-and-herzegovina", name: "Bosnia and Herzegovina", type: "country", code: "BIH" });
db.v.save({ _key: "country-botswana", name: "Botswana", type: "country", code: "BWA" });
db.v.save({ _key: "country-brazil", name: "Brazil", type: "country", code: "BRA" });
db.v.save({ _key: "country-brunei", name: "Brunei", type: "country", code: "BRN" });
db.v.save({ _key: "country-bulgaria", name: "Bulgaria", type: "country", code: "BGR" });
db.v.save({ _key: "country-burkina-faso", name: "Burkina Faso", type: "country", code: "BFA" });
db.v.save({ _key: "country-burundi", name: "Burundi", type: "country", code: "BDI" });
db.v.save({ _key: "country-cambodia", name: "Cambodia", type: "country", code: "KHM" });
db.v.save({ _key: "country-cameroon", name: "Cameroon", type: "country", code: "CMR" });
db.v.save({ _key: "country-canada", name: "Canada", type: "country", code: "CAN" });
db.v.save({ _key: "country-chad", name: "Chad", type: "country", code: "TCD" });
db.v.save({ _key: "country-chile", name: "Chile", type: "country", code: "CHL" });
db.v.save({ _key: "country-colombia", name: "Colombia", type: "country", code: "COL" });
db.v.save({ _key: "country-cote-d-ivoire", name: "Cote d'Ivoire", type: "country", code: "CIV" });
db.v.save({ _key: "country-croatia", name: "Croatia", type: "country", code: "HRV" });
db.v.save({ _key: "country-czech-republic", name: "Czech Republic", type: "country", code: "CZE" });
db.v.save({ _key: "country-denmark", name: "Denmark", type: "country", code: "DNK" });
db.v.save({ _key: "country-ecuador", name: "Ecuador", type: "country", code: "ECU" });
db.v.save({ _key: "country-egypt", name: "Egypt", type: "country", code: "EGY" });
db.v.save({ _key: "country-eritrea", name: "Eritrea", type: "country", code: "ERI" });
db.v.save({ _key: "country-finland", name: "Finland", type: "country", code: "FIN" });
db.v.save({ _key: "country-france", name: "France", type: "country", code: "FRA" });
db.v.save({ _key: "country-germany", name: "Germany", type: "country", code: "DEU" });
db.v.save({ _key: "country-people-s-republic-of-china", name: "People's Republic of China", type: "country", code: "CHN"

// vertices: capitals
db.v.save({ _key: "capital-algiers", name: "Algiers", type: "capital" });
db.v.save({ _key: "capital-andorra-la-vella", name: "Andorra la Vella", type: "capital" });
db.v.save({ _key: "capital-asmara", name: "Asmara", type: "capital" });
db.v.save({ _key: "capital-bandar-seri-begawan", name: "Bandar Seri Begawan", type: "capital" });
db.v.save({ _key: "capital-beijing", name: "Beijing", type: "capital" });
db.v.save({ _key: "capital-berlin", name: "Berlin", type: "capital" });
db.v.save({ _key: "capital-bogota", name: "Bogota", type: "capital" });
db.v.save({ _key: "capital-brasilia", name: "Brasilia", type: "capital" });
db.v.save({ _key: "capital-bridgetown", name: "Bridgetown", type: "capital" });
db.v.save({ _key: "capital-brussels", name: "Brussels", type: "capital" });
db.v.save({ _key: "capital-buenos-aires", name: "Buenos Aires", type: "capital" });
db.v.save({ _key: "capital-bujumbura", name: "Bujumbura", type: "capital" });
db.v.save({ _key: "capital-cairo", name: "Cairo", type: "capital" });
db.v.save({ _key: "capital-canberra", name: "Canberra", type: "capital" });
db.v.save({ _key: "capital-copenhagen", name: "Copenhagen", type: "capital" });
db.v.save({ _key: "capital-dhaka", name: "Dhaka", type: "capital" });
db.v.save({ _key: "capital-gaborone", name: "Gaborone", type: "capital" });
db.v.save({ _key: "capital-helsinki", name: "Helsinki", type: "capital" });
db.v.save({ _key: "capital-kabul", name: "Kabul", type: "capital" });
db.v.save({ _key: "capital-la-paz", name: "La Paz", type: "capital" });
db.v.save({ _key: "capital-luanda", name: "Luanda", type: "capital" });
db.v.save({ _key: "capital-manama", name: "Manama", type: "capital" });
db.v.save({ _key: "capital-nassau", name: "Nassau", type: "capital" });
db.v.save({ _key: "capital-n-djamena", name: "N'Djamena", type: "capital" });
db.v.save({ _key: "capital-ottawa", name: "Ottawa", type: "capital" });
db.v.save({ _key: "capital-ouagadougou", name: "Ouagadougou", type: "capital" });
db.v.save({ _key: "capital-paris", name: "Paris", type: "capital" });
db.v.save({ _key: "capital-phnom-penh", name: "Phnom Penh", type: "capital" });
db.v.save({ _key: "capital-prague", name: "Prague", type: "capital" });
db.v.save({ _key: "capital-quito", name: "Quito", type: "capital" });
db.v.save({ _key: "capital-saint-john-s", name: "Saint John's", type: "capital" });
db.v.save({ _key: "capital-santiago", name: "Santiago", type: "capital" });
db.v.save({ _key: "capital-sarajevo", name: "Sarajevo", type: "capital" });
db.v.save({ _key: "capital-sofia", name: "Sofia", type: "capital" });
db.v.save({ _key: "capital-thimphu", name: "Thimphu", type: "capital" });
db.v.save({ _key: "capital-tirana", name: "Tirana", type: "capital" });
db.v.save({ _key: "capital-vienna", name: "Vienna", type: "capital" });
db.v.save({ _key: "capital-yamoussoukro", name: "Yamoussoukro", type: "capital" });
db.v.save({ _key: "capital-yaounde", name: "Yaounde", type: "capital" });
db.v.save({ _key: "capital-zagreb", name: "Zagreb", type: "capital" });

// edges: continent -> world
db.e.save("v/continent-africa", "v/world", { type: "is-in" });
db.e.save("v/continent-asia", "v/world", { type: "is-in" });
db.e.save("v/continent-australia", "v/world", { type: "is-in" });
db.e.save("v/continent-europe", "v/world", { type: "is-in" });
db.e.save("v/continent-north-america", "v/world", { type: "is-in" });
db.e.save("v/continent-south-america", "v/world", { type: "is-in" });

// edges: country -> continent
db.e.save("v/country-afghanistan", "v/continent-asia", { type: "is-in" });
db.e.save("v/country-albania", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-algeria", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-andorra", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-angola", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-antigua-and-barbuda", "v/continent-north-america", { type: "is-in" });
db.e.save("v/country-argentina", "v/continent-south-america", { type: "is-in" });
db.e.save("v/country-australia", "v/continent-australia", { type: "is-in" });
db.e.save("v/country-austria", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-bahamas", "v/continent-north-america", { type: "is-in" });
db.e.save("v/country-bahrain", "v/continent-asia", { type: "is-in" });
db.e.save("v/country-bangladesh", "v/continent-asia", { type: "is-in" });
db.e.save("v/country-barbados", "v/continent-north-america", { type: "is-in" });
db.e.save("v/country-belgium", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-bhutan", "v/continent-asia", { type: "is-in" });
db.e.save("v/country-bolivia", "v/continent-south-america", { type: "is-in" });
db.e.save("v/country-bosnia-and-herzegovina", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-botswana", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-brazil", "v/continent-south-america", { type: "is-in" });
db.e.save("v/country-brunei", "v/continent-asia", { type: "is-in" });
db.e.save("v/country-bulgaria", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-burkina-faso", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-burundi", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-cambodia", "v/continent-asia", { type: "is-in" });
db.e.save("v/country-cameroon", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-canada", "v/continent-north-america", { type: "is-in" });
db.e.save("v/country-chad", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-chile", "v/continent-south-america", { type: "is-in" });
db.e.save("v/country-colombia", "v/continent-south-america", { type: "is-in" });
db.e.save("v/country-cote-d-ivoire", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-croatia", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-czech-republic", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-denmark", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-ecuador", "v/continent-south-america", { type: "is-in" });
db.e.save("v/country-egypt", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-eritrea", "v/continent-africa", { type: "is-in" });
db.e.save("v/country-finland", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-france", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-germany", "v/continent-europe", { type: "is-in" });
db.e.save("v/country-people-s-republic-of-china", "v/continent-asia", { type: "is-in" });

// edges: capital -> country
db.e.save("v/capital-algiers", "v/country-algeria", { type: "is-in" });
db.e.save("v/capital-andorra-la-vella", "v/country-andorra", { type: "is-in" });
db.e.save("v/capital-asmara", "v/country-eritrea", { type: "is-in" });
db.e.save("v/capital-bandar-seri-begawan", "v/country-brunei", { type: "is-in" });
db.e.save("v/capital-beijing", "v/country-people-s-republic-of-china", { type: "is-in" });
db.e.save("v/capital-berlin", "v/country-germany", { type: "is-in" });
db.e.save("v/capital-bogota", "v/country-colombia", { type: "is-in" });
db.e.save("v/capital-brasilia", "v/country-brazil", { type: "is-in" });
db.e.save("v/capital-bridgetown", "v/country-barbados", { type: "is-in" });
db.e.save("v/capital-brussels", "v/country-belgium", { type: "is-in" });
db.e.save("v/capital-buenos-aires", "v/country-argentina", { type: "is-in" });
db.e.save("v/capital-bujumbura", "v/country-burundi", { type: "is-in" });
db.e.save("v/capital-cairo", "v/country-egypt", { type: "is-in" });
db.e.save("v/capital-canberra", "v/country-australia", { type: "is-in" });
db.e.save("v/capital-copenhagen", "v/country-denmark", { type: "is-in" });
db.e.save("v/capital-dhaka", "v/country-bangladesh", { type: "is-in" });
db.e.save("v/capital-gaborone", "v/country-botswana", { type: "is-in" });
db.e.save("v/capital-helsinki", "v/country-finland", { type: "is-in" });
db.e.save("v/capital-kabul", "v/country-afghanistan", { type: "is-in" });
db.e.save("v/capital-la-paz", "v/country-bolivia", { type: "is-in" });
db.e.save("v/capital-luanda", "v/country-angola", { type: "is-in" });
db.e.save("v/capital-manama", "v/country-bahrain", { type: "is-in" });
db.e.save("v/capital-nassau", "v/country-bahamas", { type: "is-in" });
db.e.save("v/capital-n-djamena", "v/country-chad", { type: "is-in" });
db.e.save("v/capital-ottawa", "v/country-canada", { type: "is-in" });
db.e.save("v/capital-ouagadougou", "v/country-burkina-faso", { type: "is-in" });
db.e.save("v/capital-paris", "v/country-france", { type: "is-in" });
db.e.save("v/capital-phnom-penh", "v/country-cambodia", { type: "is-in" });
db.e.save("v/capital-prague", "v/country-czech-republic", { type: "is-in" });
db.e.save("v/capital-quito", "v/country-ecuador", { type: "is-in" });
db.e.save("v/capital-saint-john-s", "v/country-antigua-and-barbuda", { type: "is-in" });
db.e.save("v/capital-santiago", "v/country-chile", { type: "is-in" });
db.e.save("v/capital-sarajevo", "v/country-bosnia-and-herzegovina", { type: "is-in" });
db.e.save("v/capital-sofia", "v/country-bulgaria", { type: "is-in" });
db.e.save("v/capital-thimphu", "v/country-bhutan", { type: "is-in" });
db.e.save("v/capital-tirana", "v/country-albania", { type: "is-in" });
db.e.save("v/capital-vienna", "v/country-austria", { type: "is-in" });
db.e.save("v/capital-yamoussoukro", "v/country-cote-d-ivoire", { type: "is-in" });
db.e.save("v/capital-yaounde", "v/country-cameroon", { type: "is-in" });
db.e.save("v/capital-zagreb", "v/country-croatia", { type: "is-in" });
```

Edges, Identifiers, Handles

This is an introduction to ArangoDB's interface for edges. Edges may be used in graphs. Here we work with edges from the JavaScript shell *arangosh*. For other languages see the corresponding language API.

A graph data model always consists of at least two collections: the relations between the nodes in the graphs are stored in an "edges collection", the nodes in the graph are stored in documents in regular collections.

Edges in ArangoDB are special documents. In addition to the system attributes *_key*, *_id* and *_rev*, they have the attributes *_from* and *_to*, which contain document handles, namely the start-point and the end-point of the edge.

Example:

- the "edge" collection stores the information that a company's reception is sub-unit to the services unit and the services unit is sub-unit to the CEO. You would express this relationship with the *_from* and *_to* attributes
- the "normal" collection stores all the properties about the reception, e.g. that 20 people are working there and the room number etc
- *_from* is the document handle of the linked vertex (incoming relation)
- *_to* is the document handle of the linked vertex (outgoing relation)

Edge collections are special collections that store edge documents. Edge documents are connection documents that reference other documents. The type of a collection must be specified when a collection is created and cannot be changed afterwards.

To change edge endpoints you would need to remove old document/edge and insert new one. Other fields can be updated as in default collection.

Working with Edges

Edges are normal documents that always contain a `_from` and a `_to` attribute.

Foxx

Traditionally, server-side projects have been developed as standalone applications that guide the communication between the client-side frontend and the database backend. This has led to applications that were either developed as single monoliths or that duplicated data access and domain logic across all services that had to access the database. Additionally, tools to abstract away the underlying database calls could incur a lot of network overhead when using remote databases without careful optimization.

ArangoDB allows application developers to write their data access and domain logic as microservices running directly within the database with native access to in-memory data. The **Foxx microservice framework** makes it easy to extend ArangoDB's own REST API with custom HTTP endpoints using modern JavaScript running on the same V8 engine you know from Node.js and the Google Chrome web browser.

Unlike traditional approaches to storing logic in the database (like stored procedures), these microservices can be written as regular structured JavaScript applications that can be easily distributed and version controlled. Depending on your project's needs Foxx can be used to build anything from optimized REST endpoints performing complex data access to entire standalone applications running directly inside the database.

Foxx at a glance

Each Foxx service is defined by a JSON manifest specifying the entry point, any scripts defined by the service, possible configuration options and Foxx dependencies, as well as other metadata. Within a service, these options are exposed as the service context.

At the heart of the Foxx framework lies the Foxx Router which is used to define HTTP endpoints. A service can access the database either directly from its context using prefixed collections or the ArangoDB database API.

While Foxx is primarily designed to be used to access the database itself, ArangoDB also provides an API to make HTTP requests to external services.

Finally, scripts can be used to perform one-off tasks, which can also be scheduled to be performed asynchronously using the built-in job queue.

How does it work

Foxx services consist of JavaScript code running in the V8 JavaScript runtime embedded inside ArangoDB. Each service is mounted in each available V8 context (the number of contexts can be adjusted in the ArangoDB configuration). Incoming requests are distributed accross these contexts automatically.

If you're coming from another JavaScript environment like Node.js this is similar to running multiple Node.js processes behind a load balancer: you should not rely on server-side state (other than the database itself) between different requests as there is no way of making sure consecutive requests will be handled in the same context.

Because the JavaScript code is running inside the database another difference is that all Foxx and ArangoDB APIs are purely synchronous and should be considered blocking. This is especially important for transactions, which in ArangoDB can execute arbitrary code but may have to lock entire collections (effectively preventing any data to be written) until the code has completed.

For information on how this affects interoperability with third-party JavaScript modules written for other JavaScript environments see the chapter on dependencies.

Development mode

Development mode allows you to make changes to deployed services in-place directly on the database server's file system without downloading and re-uploading the service bundle.

You can toggle development mode on and off in the service settings tab of the web interface. Once activated the service's file system path will be shown in the info tab.

Once enabled the service's source files and manifest will be re-evaluated every time a route of the service is accessed, effectively re-deploying the service on every request. As the name indicates this is intended to be used strictly during development and is most definitely a bad idea on production servers.

Also note that if you are serving static files as part of your service, accessing these files from a browser may also trigger a re-deployment of the service. Finally, making HTTP requests to a service running in development mode from within the service (i.e. using the `@arangodb/request` module to access the service itself) is probably not a good idea either.

Beware of deleting the database the service is deployed on: it will erase the source files of the service along with the collections. You should backup the code you worked on in development before doing that to avoid losing your progress.

Foxx store

The Foxx store provides access to a number of ready-to-use official and community-maintained Foxx services you can install with a single click, including example services and wrappers for external SaaS tools like transactional e-mail services, bug loggers or analytics trackers.

You can find the Foxx store in the web interface by using the *Add Service* button in the service list.

Cluster-Foxx

When running ArangoDB in a cluster the Foxx services will run on each coordinator. Installing, upgrading and uninstalling services on any coordinator will automatically affect the other coordinators, making deployments as easy as in single-server mode. However, this means there are some limitations:

You should avoid any kind of file system state beyond the deployed service bundle itself. Don't write data to the file system or encode any expectations of the file system state other than the files in the service folder that were installed as part of the service (e.g. file uploads or custom log files).

Additionally, the development mode is not supported in cluster mode. The development mode is intended to allow modifying the service's code and seeing the effect of those changes in realtime. The service is automatically deployed to multiple coordinators, but with (temporarily) different copies of the service, the inconsistent state would lead to unpredictable behavior. It is recommended that you either re-deploy services when making changes to code running in a cluster or use development mode on a single-server installation.

Getting Started

We're going to start with an empty folder. This will be the root folder of our services. You can name it something clever but for the course of this guide we'll assume it's called the name of your service: `getting-started` .

First we need to create a manifest. Create a new file called `manifest.json` and add the following content:

```
{
  "engines": {
    "arangodb": "^3.0.0"
  }
}
```

This just tells ArangoDB the service is compatible with versions 3.0.0 and later (all the way up to but not including 4.0.0), allowing older versions of ArangoDB to understand that this service likely won't work for them and newer versions what behavior to emulate should they still support it.

The little hat to the left of the version number is not a typo, it's called a "caret" and indicates the version range. Foxx uses semantic versioning (also called "semver") for most of its version handling. You can find out more about how semver works at the official semver website.

Next we'll need to specify an entry point to our service. This is the JavaScript file that will be executed to define our service's HTTP endpoints. We can do this by adding a "main" field to our manifest:

```
{
  "engines": {
    "arangodb": "^3.0.0"
  },
  "main": "index.js"
}
```

That's all we need in our manifest for now, so let's next create the `index.js` file:

```
'use strict';
const createRouter = require('@arangodb/foxx/router');
const router = createRouter();

module.context.use(router);
```

The first line causes our file to be interpreted using strict mode. All examples in the ArangoDB documentation assume strict mode, so you might want to familiarize yourself with it if you haven't encountered it before.

The second line imports the `@arangodb/foxx/router` module which provides a function for creating new Foxx routers. We're using this function to create a new `router` object which we'll be using for our service.

The `module.context` is the so-called Foxx context or service context. This variable is available in all files that are part of your Foxx service and provides access to Foxx APIs specific to the current service, like the `use` method, which tells Foxx to mount the `router` in this service (and to expose its routes to HTTP).

Next let's define a route that prints a generic greeting:

```
// continued
router.get('/hello-world', function (req, res) {
  res.send('Hello World!');
})
.response(['text/plain'], 'A generic greeting.')
.summary('Generic greeting')
.description('Prints a generic greeting.');
```

The `router` provides the methods `get` , `post` , etc corresponding to each HTTP verb as well as the catch-all `all` . These methods indicate that the given route should be used to handle incoming requests with the given HTTP verb (or any method when using `all`).

These methods take an optional path (if omitted, it defaults to `"/"`) as well as a request handler, which is a function taking the `req` (request) and `res` (response) objects to handle the incoming request and generate the outgoing response. If you have used the express framework in Node.js, you may already be familiar with how this works, otherwise check out the chapter on routes.

The object returned by the router's methods provides additional methods to attach metadata and validation to the route. We're using `summary` and `description` to document what the route does -- these aren't strictly necessary but give us some nice auto-generated documentation. The `response` method lets us additionally document the response content type and what the response body will represent.

Try it out

At this point you can upload the service folder as a zip archive from the web interface using the *Services* tab.

Click *Add Service* then pick the *Zip* option in the dialog. You will need to provide a *mount path*, which is the URL prefix at which the service will be mounted (e.g. `/getting-started`).

Once you have picked the zip archive using the file picker, the upload should begin immediately and your service should be installed. Otherwise press the *Install* button and wait for the dialog to disappear and the service to show up in the service list.

Click anywhere on the card with your mount path on the label to open the service's details.

In the API documentation you should see the route we defined earlier (`/hello-world`) with the word `GET` next to it indicating the HTTP method it supports and the `summary` we provided on the right. By clicking on the route's path you can open the documentation for the route.

Note that the `description` we provided appears in the generated documentation as well as the description we added to the `response` (which should correctly indicate the content type `text/plain` , i.e. plain text).

Click the *Try it out!* button to send a request to the route and you should see an example request with the service's response: "Hello World!".

Congratulations! You have just created, installed and used your first Foxx service.

Parameter validation

Let's add another route that provides a more personalized greeting:

```
// continued
const joi = require('joi');

router.get('/hello/:name', function (req, res) {
  res.send(`Hello ${req.pathParams.name}`);
})
.pathParam('name', joi.string().required(), 'Name to greet.')
.response(['text/plain'], 'A personalized greeting.')
.summary('Personalized greeting')
.description('Prints a personalized greeting.');
```

The first line imports the `joi` module from npm which comes bundled with ArangoDB. Joi is a validation library that is used throughout Foxx to define schemas and parameter types.

Note: You can bundle your own modules from npm by installing them in your service folder and making sure the `node_modules` folder is included in your zip archive. For more information see the section on module dependencies in the chapter on dependencies.

The `pathParam` method allows us to specify parameters we are expecting in the path. The first argument corresponds to the parameter name in the path, the second argument is a joi schema the parameter is expected to match and the final argument serves to describe the parameter in the API documentation.

The path parameters are accessible from the `pathParams` property of the request object. We're using a template string to generate the server's response containing the parameter's value.

Note that routes with path parameters that fail to validate for the request URL will be skipped as if they wouldn't exist. This allows you to define multiple routes that are only distinguished by the schemas of their path parameters (e.g. a route taking only numeric parameters and one taking any string as a fallback).

Let's take this further and create a route that takes a JSON request body:

```
// continued
router.post('/sum', function (req, res) {
  const values = req.body.values;
  res.send({
    result: values.reduce(function (a, b) {
      return a + b;
    }, 0)
  });
})
.body(joi.object({
  values: joi.array().items(joi.number().required()).required()
}).required(), 'Values to add together.')
.response(joi.object({
  result: joi.number().required()
}).required(), 'Sum of the input values.')
.summary('Add up numbers')
.description('Calculates the sum of an array of number values.');
```

Note that we used `post` to define this route instead of `get` (which does not support request bodies). Trying to send a GET request to this route's URL (in the absence of a `get` route for the same path) will result in Foxx responding with an appropriate error response, indicating the supported HTTP methods.

As this route not only expects a JSON object as input but also responds with a JSON object as output we need to define two schemas. We don't strictly need a response schema but it helps documenting what the route should be expected to respond with and will show up in the API documentation.

Because we're passing a schema to the `response` method we don't need to explicitly tell Foxx we are sending a JSON response. The presence of a schema in the absence of a content type always implies we want JSON. Though we could just add `["application/json"]` as an additional argument after the schema if we wanted to make this more explicit.

The `body` method works the same way as the `response` method except the schema will be used to validate the request body. If the request body can't be parsed as JSON or doesn't match the schema, Foxx will reject the request with an appropriate error response.

Creating collections

The real power of Foxx comes from interacting with the database itself. In order to be able to use a collection from within our service, we should first make sure that the collection actually exists. The right place to create collections your service is going to use is in a *setup* script, which Foxx will execute for you when installing or updating the service.

First create a new folder called "scripts" in the service folder, which will be where our scripts are going to live. For simplicity's sake, our setup script will live in a file called `setup.js` inside that folder:

```
// continued
'use strict';
const db = require('@arangodb').db;
const collectionName = 'myFoxxCollection';

if (!db._collection(collectionName)) {
  db._createDocumentCollection(collectionName);
}
```

The script uses the `db` object from the `@arangodb` module, which lets us interact with the database the Foxx service was installed in and the collections inside that database. Because the script may be executed multiple times (i.e. whenever we update the service or when the server is restarted) we need to make sure we don't accidentally try to create the same collection twice (which would result in an exception); we do that by first checking whether it already exists before creating it.

The `_collection` method looks up a collection by name and returns `null` if no collection with that name was found. The `_createDocumentCollection` method creates a new document collection by name (`_createEdgeCollection` also exists and works analogously for edge collections).

Note: Because we have hardcoded the collection name, multiple copies of the service installed alongside each other in the same database will share the same collection. Because this may not always be what you want, the Foxx context also provides the `collectionName` method which applies a mount point specific prefix to any given collection name to make it unique to the service. It also provides the `collection` method, which behaves almost exactly like `db._collection` except it also applies the prefix before looking the collection up.

Next we need to tell our service about the script by adding it to the manifest file:

```
{
  "engines": {
    "arangodb": "^3.0.0"
  },
  "main": "index.js",
  "scripts": {
    "setup": "scripts/setup.js"
  }
}
```

The only thing that has changed is that we added a "scripts" field specifying the path of the setup script we just wrote.

Go back to the web interface and update the service with our new code, then check the *Collections* tab. If everything worked right, you should see a new collection called "myFoxxCollection".

Accessing collections

Let's expand our service by adding a few more routes to our `index.js` :

```
// continued
const db = require('@arangodb').db;
const errors = require('@arangodb').errors;
const foxxColl = db._collection('myFoxxCollection');
const DOC_NOT_FOUND = errors.ERROR_ARANGO_DOCUMENT_NOT_FOUND.code;

router.post('/entries', function (req, res) {
  const data = req.body;
  const meta = foxxColl.save(req.body);
  res.send(Object.assign(data, meta));
})
.body(joi.object().required(), 'Entry to store in the collection.')
.response(joi.object().required(), 'Entry stored in the collection.')
.summary('Store an entry')
.description('Stores an entry in the "myFoxxCollection" collection.');

router.get('/entries/:key', function (req, res) {
  try {
    const data = foxxColl.document(req.pathParams.key);
    res.send(data)
  } catch (e) {
    if (!e.isArangoError || e.errorNum !== DOC_NOT_FOUND) {
      throw e;
    }
    res.throw(404, 'The entry does not exist', e);
  }
})
.pathParam('key', joi.string().required(), 'Key of the entry.')
.response(joi.object().required(), 'Entry stored in the collection.')
.summary('Retrieve an entry')
.description('Retrieves an entry from the "myFoxxCollection" collection by key.');
```

We're using the `save` and `document` methods of the collection object to store and retrieve documents in the collection we created in our setup script. Because we don't care what the documents look like we allow any attributes on the request body and just accept an object.

Because the key will be automatically generated by ArangoDB when one wasn't specified in the request body, we're using `Object.assign` to apply the attributes of the metadata object returned by the `save` method to the document before returning it from our first route.

The `document` method returns a document in a collection by its `_key` or `_id` . However when no matching document exists it throws an `ArangoError` exception. Because we want to provide a more descriptive error message than ArangoDB does out of the box, we need to handle that error explicitly.

All `ArangoError` exceptions have a truthy attribute `isArangoError` that helps you recognizing these errors without having to worry about `instanceof` checks. They also provide an `errorNum` and an `errorMessage` . If you want to check for specific errors you can just import the `errors` object from the `@arangodb` module instead of having to memorize numeric error codes.

Instead of defining our own response logic for the error case we just use `res.throw` , which makes the response object throw an exception Foxx can recognize and convert to the appropriate server response. We also pass along the exception itself so Foxx can provide more diagnostic information when we want it to.

We could extend the post route to support arrays of objects as well, each object following a certain schema:

```
// store schema in variable to make it re-usable, see .body()
const docSchema = joi.object().required().keys({
  name: joi.string().required(),
  age: joi.number().required()
}).unknown(); // allow additional attributes

router.post('/entries', function (req, res) {
  const multiple = Array.isArray(req.body);
  const body = multiple ? req.body : [req.body];

  let data = [];
  for (var doc of body) {
    const meta = foxxColl.save(doc);
    data.push(Object.assign(doc, meta));
  }
  res.send(multiple ? data : data[0]);

})
.body(joi.alternatives().try(
  docSchema,
  joi.array().items(docSchema)
), 'Entry or entries to store in the collection.')
.response(joi.alternatives().try(
  joi.object().required(),
  joi.array().items(joi.object().required())
), 'Entry or entries stored in the collection.')
.summary('Store entry or entries')
.description('Store a single entry or multiple entries in the "myFoxxCollection" collection.');
```

Writing database queries

Storing and retrieving entries is fine, but right now we have to memorize each key when we create an entry. Let's add a route that gives us a list of the keys of all entries so we can use those to look an entry up in detail.

The naïve approach would be to use the `toArray()` method to convert the entire collection to an array and just return that. But we're only interested in the keys and there might potentially be so many entries that first retrieving every single document might get unwieldy. Let's write a short AQL query to do this instead:

```
// continued
const aql = require('@arangodb').aql;

router.get('/entries', function (req, res) {
  const keys = db._query(aql`
    FOR entry IN ${foxxColl}
    RETURN entry._key
  `);
  res.send(keys);
})
.response(joi.array().items(
  joi.string().required()
).required(), 'List of entry keys.')
.summary('List entry keys')
.description('Assembles a list of keys of entries in the collection.');
```

Here we're using two new things:

The `_query` method executes an AQL query in the active database.

The `aql` template string handler allows us to write multi-line AQL queries and also handles query parameters and collection names. Instead of hardcoding the name of the collection we want to use in the query we can simply reference the `foxxColl` variable we defined earlier -- it recognizes the value as an ArangoDB collection object and knows we are specifying a collection rather than a regular value even though AQL distinguishes between the two.

Note: If you aren't used to JavaScript template strings and template string handlers just think of `aql` as a function that receives the multiline string split at every `${}` expression as well as an array of the values of those expressions -- that's actually all there is to it.

Alternatively, here's a version without template strings (notice how much cleaner the `aql` version will be in comparison when you have multiple variables):

```
const keys = db._query(
  'FOR entry IN @@coll RETURN entry._key',
  {'@coll': foxxColl}
);
```

Next steps

You now know how to create a Foxx service from scratch, how to handle user input and how to access the database from within your Foxx service to store, retrieve and query data you store inside ArangoDB. This should allow you to build meaningful APIs for your own applications but there are many more things you can do with Foxx:

- Need to go faster? Turn on development mode and hack on your code right on the server.
- Concerned about security? You could add authentication to your service to protect access to the data before it even leaves the database.
- Writing a single page app? You could store some basic assets right inside your Foxx service.
- Need to integrate external services? You can make HTTP requests from inside Foxx and use queued jobs to perform that work in the background.
- Tired of reinventing the wheel? Learn about dependencies.

Manifest files

Every service comes with a `manifest.json` file providing metadata. The following fields are allowed in manifests:

- **configuration**: `Object` (optional)

  An object defining the configuration options this service requires.

- **defaultDocument**: `string` (optional)

  If specified, the `/` (root) route of the service will automatically redirect to the given relative path, e.g.:

  ```
  "defaultDocument": "index.html"
  ```

  This would have the same effect as creating the following route in JavaScript:

  ```
  const createRouter = require('@arangodb/foxx/router');
  const indexRouter = createRouter();
  indexRouter.all('/', function (req, res) {
    res.redirect('index.html');
  });
  module.context.use(indexRouter);
  ```

  Note: As of 3.0.0 this field can safely be omitted; the value no longer defaults to `"index.html"` .

- **dependencies**: `Object` (optional) and **provides**: `Object` (optional)

  Objects specifying other services this service has as dependencies and what dependencies it can provide to other services.

- **engines**: `Object` (optional)

  An object indicating the semantic version ranges of ArangoDB (or compatible environments) the service will be compatible with, e.g.:

  ```
  "engines": {
    "arangodb": "^3.0.0"
  }
  ```

  This should correctly indicate the minimum version of ArangoDB the service has been tested against. Foxx maintains a strict semantic versioning policy as of ArangoDB 3.0.0 so it is generally safe to use semver ranges (e.g. `^3.0.0` to match any version greater or equal to `3.0.0` and below `4.0.0`) for maximum compatibility.

- **files**: `Object` (optional)

  An object defining file assets served by this service.

- **lib**: `string` (Default: `"."`)

  The relative path to the Foxx JavaScript files in the service, e.g.:

  ```
  "lib": "lib"
  ```

  This would result in the main entry point (see below) and other JavaScript paths being resolved as relative to the `lib` folder inside the service folder.

- **main**: `string` (optional)

  The relative path to the main entry point of this service (relative to *lib*, see above), e.g.:

  ```
  "main": "index.js"
  ```

  This would result in Foxx loading and executing the file `index.js` when the service is mounted or started.

Note: while it is technically possible to omit this field, you will likely want to provide an entry point to your service as this is the only way to expose HTTP routes or export a JavaScript API.

- **scripts**: `Object` (optional)

  An object defining named scripts provided by this service, which can either be used directly or as queued jobs by other services.

- **tests**: `string` or `Array<string>` (optional)

  A path or list of paths of JavaScript tests provided for this service.

Additionally manifests can provide the following metadata:

- **author**: `string` (optional)

  The full name of the author of the service (i.e. you). This will be shown in the web interface.

- **contributors**: `Array<string>` (optional)

  A list of names of people that have contributed to the development of the service in some way. This will be shown in the web interface.

- **description**: `string` (optional)

  A human-readable description of the service. This will be shown in the web interface.

- **keywords**: `Array<string>` (optional)

  A list of keywords that help categorize this service. This is used by the Foxx Store installers to organize services.

- **license**: `string` (optional)

  A string identifying the license under which the service is published, ideally in the form of an SPDX license identifier. This will be shown in the web interface.

- **name**: `string` (optional)

  The name of the Foxx service. Allowed characters are A-Z, 0-9, the ASCII hyphen (`-`) and underscore (`_`) characters. The name must not start with a number. This will be shown in the web interface.

- **thumbnail**: `string` (optional)

  The filename of a thumbnail that will be used alongside the service in the web interface. This should be a JPEG or PNG image that looks good at sizes 50x50 and 160x160.

- **version**: `string` (optional)

  The version number of the Foxx service. The version number must follow the semantic versioning format. This will be shown in the web interface.

Examples

```
{
  "name": "example-foxx-service",
  "version": "3.0.0-dev",
  "license": "MIT",
  "description": "An example service with a relatively full-featured manifest.",
  "thumbnail": "foxx-icon.png",
  "keywords": ["demo", "service"],
  "author": "ArangoDB GmbH",
  "contributors": [
    "Alan Plum <alan@arangodb.example>"
  ],

  "lib": "dist",
  "main": "entry.js",
  "defaultDocument": "welcome.html",
  "engines": {
    "arangodb": "^3.0.0"
  },

  "files": {
    "welcome.html": "assets/index.html",
    "hello.jpg": "assets/hello.jpg"
    "world.jpg": {
      "path": "assets/world.jpg",
      "type": "image/jpeg",
      "gzip": false
    }
  },

  "tests": "dist/**.spec.js"
}
```

Foxx service context

The service context provides access to methods and attributes that are specific to a given service. In a Foxx service the context is generally available as the `module.context` variable. Within a router's request handler the request and response objects' `context` attribute also provide access to the context of the service the route was mounted in (which may be different from the one the route handler was defined in).

Examples

```
// in service /my-foxx-1
const createRouter = require('@arangodb/foxx/router');
const router = createRouter();

// See the chapter on dependencies for more info on
// how exports and dependencies work across services
module.exports = {routes: router};

router.get(function (req, res) {
  module.context.mount === '/my-foxx-1';
  req.context.mount === '/my-foxx-2';
  res.write('Hello from my-foxx-1');
});

// in service /my-foxx-2
const createRouter = require('@arangodb/foxx/router');
const router2 = createRouter();

module.context.use(router2);

router2.post(function (req, res) {
  module.context.mount === '/my-foxx-2';
  req.context.mount === '/my-foxx-2';
  res.write('Hello from my-foxx-2');
});

const router1 = module.context.dependencies.myFoxx1.routes;
module.context.use(router1);
```

The service context specifies the following properties:

- **argv**: `any`

  Any arguments passed in if the current file was executed as a script or queued job.

- **basePath**: `string`

  The file system path of the service, i.e. the folder in which the service was installed to by ArangoDB.

- **baseUrl**: `string`

  The base URL of the service, relative to the ArangoDB server, e.g. `/_db/_system/my-foxx` .

- **collectionPrefix**: `string`

  The prefix that will be used by *collection* and *collectionName* to derive the names of service-specific collections. This is derived from the service's mount point, e.g. `/my-foxx` becomes `my_foxx` .

- **configuration**: `Object`

  Configuration options for the service.

- **dependencies**: `Object`

  Configured dependencies for the service.

- **isDevelopment**: `boolean`

  Indicates whether the service is running in development mode.

- **isProduction**: `boolean`

  The inverse of *isDevelopment*.

- **manifest**: `Object`

  The parsed manifest file of the service.

- **mount**: `string`

  The mount point of the service, e.g. `/my-foxx` .

apiDocumentation

`module.context.apiDocumentation([options]): Function`

DEPRECATED

Creates a request handler that serves the API documentation.

Note: This method has been deprecated in ArangoDB 3.1 and replaced with the more straightforward `createDocumentationRouter` method providing the same functionality.

Arguments

See `createDocumentationRouter` below.

Examples

```
// Serve the API docs for the current service
router.get('/docs/*', module.context.apiDocumentation());

// Note that the path must end with a wildcard
// and the route must use HTTP GET.
```

createDocumentationRouter

`module.context.createDocumentationRouter([options]): Router`

Creates a router that serves the API documentation.

Note: The router can be mounted like any other child router (see examples below).

Arguments

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **mount**: `string` (Default: `module.context.mount`)

    The mount path of the service to serve the documentation of.

  - **indexFile**: `string` (Default: `"index.html"`)

    File name of the HTML file serving the API documentation.

  - **swaggerRoot**: `string` (optional)

    Full path of the folder containing the Swagger assets and the *indexFile*. Defaults to the Swagger assets used by the web interface.

  - **before**: `Function` (optional)

    A function that will be executed before a request is handled.

    If the function returns `false` the request will not be processed any further.

    If the function returns an object, its attributes will be used to override the *options* for the current request.

Any other return value will be ignored.

If *options* is a function it will be used as the *before* option.

If *options* is a string it will be used as the *swaggerRoot* option.

Returns a Foxx router.

Examples

```
// Serve the API docs for the current service
router.use('/docs', module.context.createDocumentationRouter());

// -- or --

// Serve the API docs for the service the router is mounted in
router.use('/docs', module.context.createDocumentationRouter(function (req) {
  return {mount: req.context.mount};
}));

// -- or --

// Serve the API docs only for users authenticated with ArangoDB
router.use('/docs', module.context.createDocumentationRouter(function (req, res) {
  if (req.suffix === 'swagger.json' && !req.arangoUser) {
    res.throw(401, 'Not authenticated');
  }
}));
```

collection

`module.context.collection(name): ArangoCollection | null`

Passes the given name to *collectionName*, then looks up the collection with the prefixed name.

Arguments

- **name**: `string`

  Unprefixed name of the service-specific collection.

Returns a collection or `null` if no collection with the prefixed name exists.

collectionName

`module.context.collectionName(name): string`

Prefixes the given name with the *collectionPrefix* for this service.

Arguments

- **name**: `string`

  Unprefixed name of the service-specific collection.

Returns the prefixed name.

Examples

```
module.context.mount === '/my-foxx'
module.context.collectionName('doodads') === 'my_foxx_doodads'
```

file

`module.context.file(name, [encoding]): Buffer | string`

Passes the given name to *fileName*, then loads the file with the resulting name.

Arguments

- **name**: `string`

  Name of the file to load, relative to the current service.

- **encoding**: `string` (optional)

  Encoding of the file, e.g. `utf-8`. If omitted the file will be loaded as a raw buffer instead of a string.

Returns the file's contents.

fileName

`module.context.fileName(name): string`

Resolves the given file name relative to the current service.

Arguments

- **name**: `string`

  Name of the file, relative to the current service.

Returns the absolute file path.

use

`module.context.use([path], router): Endpoint`

Mounts a given router on the service to expose the router's routes on the service's mount point.

Arguments

- **path**: `string` (Default: `"/"`)

  Path to mount the router at, relative to the service's mount point.

- **router**: `Router | Middleware`

  A router or middleware to mount.

Returns an Endpoint for the given router or middleware.

Note: Mounting services at run time (e.g. within request handlers or queued jobs) is not supported.

Foxx configuration

Foxx services can define configuration parameters to make them more re-usable.

The `configuration` object maps names to configuration parameters:

- The key is the name under whicht the parameter will be available on the service context's `configuration` property.
- The value is a parameter definition.

The parameter definition can have the following properties:

- **description**: `string`

  Human readable description of the parameter.

- **type**: `string` (Default: `"string"`)

  Type of the configuration parameter. Supported values are:

  - `"integer"` or `"int"` : any finite integer number.
  - `"boolean"` or `"bool"` : the values `true` or `false` .
  - `"number"` : any finite decimal or integer number.
  - `"string"` : any string value.
  - `"json"` : any well-formed JSON value.
  - `"password"` : like *string* but will be displayed as a masked input field in the web frontend.

- **default**: `any`

  Default value of the configuration parameter.

- **required**: (Default: `true`)

  Whether the parameter is required.

If the configuration has parameters that do not specify a default value, you need to configure the service before it becomes active. In the meantime a fallback servicelication will be mounted that responds to all requests with a HTTP 500 status code indicating a server-side error.

The configuration parameters of a mounted service can be adjusted from the web interface by clicking the *Configuration* button in the service details.

Examples

```
"configuration": {
  "currency": {
    "description": "Currency symbol to use for prices in the shop.",
    "default": "$",
    "type": "string"
  },
  "secretKey": {
    "description": "Secret key to use for signing session tokens.",
    "type": "password"
  }
}
```

Dependency management

There are two things commonly called "dependencies" in Foxx:

- Module dependencies, i.e. dependencies on external JavaScript modules (e.g. from the public npm registry)
- Foxx dependencies, i.e. dependencies between Foxx services

Let's look at them in more detail:

Module dependencies

You can use the `node_modules` folder to bundle third-party Foxx-compatible npm and Node.js modules with your Foxx service. Typically this is achieved by adding a `package.json` file to your project specifying npm dependencies using the `dependencies` attribute and installing them with the npm command-line tool.

Make sure to include the actual `node_modules` folder in your Foxx service bundle as ArangoDB will not do anything special to install these dependencies. Also keep in mind that bundling extraneous modules like development dependencies may bloat the file size of your Foxx service bundle.

Compatibility caveats

Unlike JavaScript in browsers or Node.js, the JavaScript environment in ArangoDB is synchronous. This means any modules that depend on asynchronous behaviour like promises or `setTimeout` will not behave correctly in ArangoDB or Foxx. Additionally unlike Node.js ArangoDB does not support native extensions. All modules have to be implemented in pure JavaScript.

While ArangoDB provides a lot of compatibility code to support modules written for Node.js, some Node.js built-in modules can not be provided by ArangoDB. For a closer look at the Node.js modules ArangoDB does or does not provide check out the appendix on JavaScript modules.

Also note that these restrictions not only apply on the modules you wish to install but also the dependencies of those modules. As a rule of thumb: modules written to work in Node.js and the browser that do not rely on async behaviour should generally work; modules that rely on network or filesystem I/O or make heavy use of async behaviour most likely will not.

Foxx dependencies

Foxx dependencies can be declared in a service's manifest using the `provides` and `dependencies` fields:

- `provides` lists the dependencies a given service provides, i.e. which APIs it claims to be compatible with
- `dependencies` lists the dependencies a given service uses, i.e. which APIs its dependencies need to be compatible with

A dependency name should generally use the same format as a namespaced (org-scoped) NPM module, e.g. `@foxx/sessions`.

Dependency names refer to the external JavaScript API of a service rather than specific services implementing those APIs. Some dependency names defined by officially maintained services are:

- `@foxx/auth` (version `1.0.0`)
- `@foxx/api-keys` (version `1.0.0`)
- `@foxx/bugsnag` (versions `1.0.0` and `2.0.0`)
- `@foxx/mailgun` (versions `1.0.0` and `2.0.0`)
- `@foxx/postageapp` (versions `1.0.0` and `2.0.0`)
- `@foxx/postmark` (versions `1.0.0` and `2.0.0`)
- `@foxx/sendgrid` (versions `1.0.0` and `2.0.0`)
- `@foxx/oauth2` (versions `1.0.0` and `2.0.0`)
- `@foxx/segment-io` (versions `1.0.0` and `2.0.0`)
- `@foxx/sessions` (versions `1.0.0` and `2.0.0`)
- `@foxx/users` (versions `1.0.0`, `2.0.0` and `3.0.0`)

A `provides` definition maps each provided dependency's name to the provided version:

```
"provides": {
  "@foxx/auth": "1.0.0"
}
```

A `dependencies` definition maps the local alias of a given dependency against its name and the supported version range (either as a JSON object or a shorthand string):

```
"dependencies": {
  "mySessions": "@foxx/sessions:^2.0.0",
  "myAuth": {
    "name": "@foxx/auth",
    "version": "^1.0.0",
    "description": "This description is entirely optional.",
    "required": false
  }
}
```

Dependencies can be configured from the web interface in a service's settings tab using the *Dependencies* button.

The value for each dependency should be the database-relative mount path of the service (including the leading slash). In order to be usable as the dependency of another service both services need to be mounted in the same database. A service can be used to provide multiple dependencies for the same service (as long as the expected JavaScript APIs don't conflict).

A service that has unconfigured required dependencies can not be used until all of its dependencies have been configured.

It is possible to specify the mount path of a service that does not actually declare the dependency as provided. There is currently no validation beyond the manifest formats.

When a service uses another mounted service as a dependency the dependency's `main` entry file's `exports` object becomes available in the `module.context.dependencies` object of the other service:

Examples

Service A and Service B are mounted in the same database. Service B has a dependency with the local alias `"greeter"`. The dependency is configured to use the mount path of Service B.

```
// Entry file of Service A
module.exports = {
  sayHi () {
    return 'Hello';
  }
};

// Somewhere in Service B
const greeter = module.context.dependencies.greeter;
res.write(greeter.sayHi());
```

Routers

`const createRouter = require('@arangodb/foxx/router');`

Routers let you define routes that extend ArangoDB's HTTP API with custom endpoints.

Routers need to be mounted using the `use` method of a service context to expose their HTTP routes at a service's mount path.

You can pass routers between services mounted in the same database as dependencies. You can even nest routers within each other.

Creating a router

`createRouter(): Router`

This returns a new, clean router object that has not yet been mounted in the service and can be exported like any other object.

Request handlers

`router.get([path], [...middleware], handler, [name]): Endpoint`

`router.post([path], [...middleware], handler, [name]): Endpoint`

`router.put([path], [...middleware], handler, [name]): Endpoint`

`router.patch([path], [...middleware], handler, [name]): Endpoint`

`router.delete([path], [...middleware], handler, [name]): Endpoint`

`router.all([path], [...middleware], handler, [name]): Endpoint`

These methods let you specify routes on the router. The `all` method defines a route that will match any supported HTTP verb, the other methods define routes that only match the HTTP verb with the same name.

Arguments

- **path**: `string` (Default: `"/"`)

  The path of the request handler relative to the base path the Router is mounted at. If omitted, the request handler will handle requests to the base path of the Router. For information on defining dynamic routes see the section on path parameters in the chapter on router endpoints.

- **middleware**: `Function` (optional)

  Zero or more middleware functions that take the following arguments:

  - **req**: `Request`

    An incoming server request object.

  - **res**: `Response`

    An outgoing server response object.

  - **next**: `Function`

    A callback that passes control over to the next middleware function and returns when that function has completed.

    If a truthy argument is passed, that argument will be thrown as an error.

    If there is no next middleware function, the `handler` will be invoked instead (see below).

- **handler**: `Function`

  A function that takes the following arguments:

  - **req**: `Request`

    An incoming server request object.

  - **res**: `Response`

    An outgoing server response.

- **name**: `string` (optional)

  A name that can be used to generate URLs for the endpoint. For more information see the `reverse` method of the request object.

Returns an Endpoint for the route.

Examples

Simple index route:

```
router.get(function (req, res) {
  res.set('content-type', 'text/plain');
  res.write('Hello World!');
});
```

Restricting access to authenticated ArangoDB users:

```
router.get('/secrets', function (req, res, next) {
  if (req.arangoUser) {
    next();
  } else {
    res.throw(404, 'Secrets? What secrets?');
  }
}, function (req, res) {
  res.download('allOurSecrets.zip');
});
```

Multiple middleware functions:

```
function counting (req, res, next) {
  if (!req.counter) req.counter = 0;
  req.counter++;
  next();
  req.counter--;
}
router.get(counting, counting, counting, function (req, res) {
  res.json({counter: req.counter}); // {"counter": 3}
});
```

Mounting child routers and middleware

`router.use([path], middleware, [name]): Endpoint`

The `use` method lets you mount a child router or middleware at a given path.

Arguments

- **path**: `string` (optional)

  The path of the middleware relative to the base path the Router is mounted at. If omitted, the middleware will handle requests to the base path of the Router. For information on defining dynamic routes see the section on path parameters in the chapter on router endpoints.

- **middleware**: `Router | Middleware`

  An unmounted router object or a middleware.

- **name**: `string` (optional)

  A name that can be used to generate URLs for endpoints of this router. For more information see the `reverse` method of the request object. Has no effect if *handler* is a Middleware.

Returns an Endpoint for the middleware or child router.

Endpoints

Endpoints are returned by the `use`, `all` and HTTP verb (e.g. `get`, `post`) methods of routers as well as the `use` method of the service context. They can be used to attach metadata to mounted routes, middleware and child routers that affects how requests and responses are processed or provides API documentation.

Endpoints should only be used to invoke the following methods. Endpoint methods can be chained together (each method returns the endpoint itself).

header

`endpoint.header(name, [schema], [description]): this`

Defines a request header recognized by the endpoint. Any additional non-defined headers will be treated as optional string values. The definitions will also be shown in the route details in the API documentation.

If the endpoint is a child router, all routes of that router will use this header definition unless overridden.

Arguments

- **name**: `string`

  Name of the header. This should be considered case insensitive as all header names will be converted to lowercase.

- **schema**: `Schema` (optional)

  A schema describing the format of the header value. This can be a joi schema or anything that has a compatible `validate` method.

  The value of this header will be set to the `value` property of the validation result. A validation failure will result in an automatic 400 (Bad Request) error response.

- **description**: `string` (optional)

  A human readable string that will be shown in the API documentation.

Returns the endpoint.

Examples

```
router.get(/* ... */)
.header('arangoVersion', joi.number().min(30000).default(30000));
```

pathParam

`endpoint.pathParam(name, [schema], [description]): this`

Defines a path parameter recognized by the endpoint. Path parameters are expected to be filled as part of the endpoint's mount path. Any additional non-defined path parameters will be treated as optional string values. The definitions will also be shown in the route details in the API documentation.

If the endpoint is a child router, all routes of that router will use this parameter definition unless overridden.

Arguments

- **name**: `string`

  Name of the parameter.

- **schema**: `Schema` (optional)

  A schema describing the format of the parameter. This can be a joi schema or anything that has a compatible `validate` method.

The value of this parameter will be set to the `value` property of the validation result. A validation failure will result in the route failing to match and being ignored (resulting in a 404 (Not Found) error response if no other routes match).

- **description**: `string` (optional)

  A human readable string that will be shown in the API documentation.

Returns the endpoint.

Examples

```
router.get('/some/:num/here', /* ... */)
.pathParam('num', joi.number().required());
```

queryParam

`endpoint.queryParam(name, [schema], [description]): this`

Defines a query parameter recognized by the endpoint. Any additional non-defined query parameters will be treated as optional string values. The definitions will also be shown in the route details in the API documentation.

If the endpoint is a child router, all routes of that router will use this parameter definition unless overridden.

Arguments

- **name**: `string`

  Name of the parameter.

- **schema**: `Schema` (optional)

  A schema describing the format of the parameter. This can be a joi schema or anything that has a compatible `validate` method.

  The value of this parameter will be set to the `value` property of the validation result. A validation failure will result in an automatic 400 (Bad Request) error response.

- **description**: `string` (optional)

  A human readable string that will be shown in the API documentation.

Returns the endpoint.

Examples

```
router.get(/* ... */)
.queryParam('num', joi.number().required());
```

body

`endpoint.body([model], [mimes], [description]): this`

Defines the request body recognized by the endpoint. There can only be one request body definition per endpoint. The definition will also be shown in the route details in the API documentation.

If the endpoint is a child router, all routes of that router will use this body definition unless overridden. If the endpoint is a middleware, the request body will only be parsed once (i.e. the MIME types of the route matching the same request will be ignored but the body will still be validated again).

Arguments

- **model**: `Model | Schema | null` (optional)

  A model or joi schema describing the request body. A validation failure will result in an automatic 400 (Bad Request) error response.

  If the value is a model with a `fromClient` method, that method will be applied to the parsed request body.

  If the value is a schema or a model with a schema, the schema will be used to validate the request body and the `value` property of the validation result of the parsed request body will be used instead of the parsed request body itself.

  If the value is a model or a schema and the MIME type has been omitted, the MIME type will default to JSON instead.

  If the value is explicitly set to `null`, no request body will be expected.

  If the value is an array containing exactly one model or schema, the request body will be treated as an array of items matching that model or schema.

- **mimes**: `Array<string>` (optional)

  An array of MIME types the route supports.

  Common non-mime aliases like "json" or "html" are also supported and will be expanded to the appropriate MIME type (e.g. "application/json" and "text/html").

  If the MIME type is recognized by Foxx the request body will be parsed into the appropriate structure before being validated. Currently only JSON, `application/x-www-form-urlencoded` and multipart formats are supported in this way.

  If the MIME type indicated in the request headers does not match any of the supported MIME types, the first MIME type in the list will be used instead.

  Failure to parse the request body will result in an automatic 400 (Bad Request) error response.

- **description**: `string` (optional)

  A human readable string that will be shown in the API documentation.

Returns the endpoint.

Examples

```
router.post('/expects/some/json', /* ... */)
.body(
  joi.object().required(),
  'This implies JSON.'
);

router.post('/expects/nothing', /* ... */)
.body(null); // No body allowed

router.post('/expects/some/plaintext', /* ... */)
.body(['text/plain'], 'This body will be a string.');
```

response

`endpoint.response([status], [model], [mimes], [description]): this`

Defines a response body for the endpoint. When using the response object's `send` method in the request handler of this route, the definition with the matching status code will be used to generate the response body. The definitions will also be shown in the route details in the API documentation.

If the endpoint is a child router, all routes of that router will use this response definition unless overridden. If the endpoint is a middleware, this method has no effect.

Arguments

- **status**: `number | string` (Default: `200` or `204`)

  HTTP status code the response applies to. If a string is provided instead of a numeric status code it will be used to look up a numeric status code using the statuses module.

- **model**: `Model | Schema | null` (optional)

  A model or joi schema describing the response body.

If the value is a model with a `forClient` method, that method will be applied to the data passed to `response.send` within the route if the response status code matches (but also if no status code has been set).

If the value is a schema or a model with a schema, the actual schema will not be used to validate the response body and only serves to document the response in more detail in the API documentation.

If the value is a model or a schema and the MIME type has been omitted, the MIME type will default to JSON instead.

If the value is explicitly set to `null` and the status code has been omitted, the status code will default to `204` ("no content") instead of `200`.

If the value is an array containing exactly one model or schema, the response body will be an array of items matching that model or schema.

- **mimes**: `Array<string>` (optional)

  An array of MIME types the route might respond with for this status code.

  Common non-mime aliases like "json" or "html" are also supported and will be expanded to the appropriate MIME type (e.g. "application/json" and "text/html").

  When using the `response.send` method the response body will be converted to the appropriate MIME type if possible.

- **description**: `string` (optional)

  A human-readable string that briefly describes the response and will be shown in the endpoint's detailed documentation.

Returns the endpoint.

Examples

```
// This example only provides documentation
// and implies a generic JSON response body.
router.get(/* ... */)
.response(
  joi.array().items(joi.string()),
  'A list of doodad identifiers.'
);

// No response body will be expected here.
router.delete(/* ... */)
.response(null, 'The doodad no longer exists.');

// An endpoint can define multiple response types
// for different status codes -- but never more than
// one for each status code.
router.post(/* ... */)
.response('found', 'The doodad is located elsewhere.')
.response(201, ['text/plain'], 'The doodad was created so here is a haiku.');

// Here the response body will be set to
// the querystring-encoded result of
// FormModel.forClient({some: 'data'})
// because the status code defaults to 200.
router.patch(function (req, res) {
  // ...
  res.send({some: 'data'});
})
.response(FormModel, ['application/x-www-form-urlencoded'], 'OMG.');

// In this case the response body will be set to
// SomeModel.forClient({some: 'data'}) because
// the status code has been set to 201 before.
router.put(function (req, res) {
  // ...
  res.status(201);
  res.send({some: 'data'});
})
.response(201, SomeModel, 'Something amazing happened.');
```

error

`endpoint.error(status, [description]): this`

Documents an error status for the endpoint.

If the endpoint is a child router, all routes of that router will use this error description unless overridden. If the endpoint is a middleware, this method has no effect.

This method only affects the generated API documentation and has not other effect within the service itself.

Arguments

- **status**: `number | string`

  HTTP status code for the error (e.g. `400` for "bad request"). If a string is provided instead of a numeric status code it will be used to look up a numeric status code using the statuses module.

- **description**: `string` (optional)

  A human-readable string that briefly describes the error condition and will be shown in the endpoint's detailed documentation.

Returns the endpoint.

Examples

```
router.get(function (req, res) {
  // ...
  res.throw(403, 'Validation error at x.y.z');
})
.error(403, 'Indicates that a validation has failed.');
```

summary

`endpoint.summary(summary): this`

Adds a short description to the endpoint's API documentation.

If the endpoint is a child router, all routes of that router will use this summary unless overridden. If the endpoint is a middleware, this method has no effect.

This method only affects the generated API documentation and has not other effect within the service itself.

Arguments

- **summary**: `string`

  A human-readable string that briefly describes the endpoint and will appear next to the endpoint's path in the documentation.

Returns the endpoint.

Examples

```
router.get(/* ... */)
.summary('List all discombobulated doodads')
```

description

`endpoint.description(description): this`

Adds a long description to the endpoint's API documentation.

If the endpoint is a child router, all routes of that router will use this description unless overridden. If the endpoint is a middleware, this method has no effect.

This method only affects the generated API documentation and has not other effect within the service itself.

Arguments

- **description**: `string`

  A human-readable string that describes the endpoint in detail and will be shown in the endpoint's detailed documentation.

Returns the endpoint.

Examples

```
// The "dedent" library helps formatting
// multi-line strings by adjusting indentation
// and removing leading and trailing blank lines
const dd = require('dedent');
router.post(/* ... */)
.description(dd`
  This route discombobulates the doodads by
  frobnicating the moxie of the request body.
`)
```

deprecated

`endpoint.deprecated([deprecated]): this`

Marks the endpoint as deprecated.

If the endpoint is a child router, all routes of that router will also be marked as deprecated. If the endpoint is a middleware, this method has no effect.

This method only affects the generated API documentation and has not other effect within the service itself.

Arguments

- **deprecated**: `boolean` (Default: `true`)

  Whether the endpoint should be marked as deprecated. If set to `false` the endpoint will be explicitly marked as *not* deprecated.

Returns the endpoint.

Examples

```
router.get(/* ... */)
.deprecated();
```

Middleware

Middleware in Foxx refers to functions that can be mounted like routes and can manipulate the request and response objects before and after the route itself is invoked. They can also be used to control access or to provide common logic like logging etc. Unlike routes, middleware is mounted with the `use` method like a router.

Instead of a function the `use` method can also accept an object with a `register` function that will be passed the endpoint the middleware will be mounted at and returns the actual middleware function. This allows manipulating the endpoint before creating the middleware (e.g. to document headers, request bodies, path parameters or query parameters).

Examples

Restrict access to ArangoDB-authenticated users:

```
module.context.use(function (req, res, next) {
  if (!req.arangoUser) {
    res.throw(401, 'Not authenticated with ArangoDB');
  }
  next();
});
```

Any truthy argument passed to the `next` function will be thrown as an error:

```
module.context.use(function (req, res, next) {
  let err = null;
  if (!req.arangoUser) {
    err = new Error('This should never happen');
  }
  next(err); // throws if the error was set
})
```

Trivial logging middleware:

```
module.context.use(function (req, res, next) {
  const start = Date.now();
  try {
    next();
  } finally {
    console.log(`Handled request in ${Date.now() - start}ms`);
  }
});
```

More complex example for header-based sessions:

```
const sessions = module.context.collection('sessions');
module.context.use({
  register (endpoint) {
    endpoint.header('x-session-id', joi.string().optional(), 'The session ID.');
    return function (req, res, next) {
      const sid = req.get('x-session-id');
      if (sid) {
        try {
          req.session = sessions.document(sid);
        } catch (e) {
          delete req.headers['x-session-id'];
        }
      }
      next();
      if (req.session) {
        if (req.session._rev) {
          sessions.replace(req.session, req.session);
          res.set('x-session-id', req.session._key);
        } else {
          const meta = sessions.save(req.session);
          res.set('x-session-id', meta._key);
        }
      }
    };
  }
});
```

Request objects

The request object specifies the following properties:

- **arangoUser**: `string | null`

  The authenticated ArangoDB username used to make the request. This value is only set if authentication is enabled in ArangoDB and the request set an `authorization` header ArangoDB was able to verify. You are strongly encouraged to implement your own authentication logic for your own services but this property can be useful if you need to integrate with ArangoDB's own authentication mechanisms.

- **arangoVersion**: `number`

  The numeric value of the `x-arango-version` header or the numeric version of the ArangoDB server (e.g. `30102` for version 3.1.2) if no valid header was provided.

- **baseUrl**: `string`

  Root-relative base URL of the service, i.e. the prefix `"/_db/"` followed by the value of *database*.

- **body**: `any`

  The processed and validated request body for the current route.

  For details on how request bodies can be processed and validated by Foxx see the body method of the endpoint object.

- **context**: `Context`

  The service context in which the router was mounted (rather than the context in which the route was defined).

- **database**: `string`

  The name of the database in which the request is being handled, e.g. `"_system"`.

- **headers**: `object`

  The raw headers object.

  For details on how request headers can be validated by Foxx see the header method of the endpoint object.

- **hostname**: `string`

  The hostname (domain name) indicated in the request headers.

  Defaults to the hostname portion (i.e. excluding the port) of the `Host` header and falls back to the listening address of the server.

- **method**: `string`

  The HTTP verb used to make the request, e.g. `"GET"`.

- **originalUrl**: `string`

  Root-relative URL of the request, i.e. *path* followed by the raw query parameters, if any.

- **path**: `string`

  Database-relative path of the request URL (not including the query parameters).

- **pathParams**: `object`

  An object mapping the names of path parameters of the current route to their validated values.

  For details on how path parameters can be validated by Foxx see the pathParam method of the endpoint object.

- **port**: `number`

  The port indicated in the request headers.

Defaults to the port portion (i.e. excluding the hostname) of the `Host` header and falls back to the listening port or the appropriate default port (`443` for HTTPS or `80` for HTTP, depending on *secure*) if the header only indicates a hostname.

If the request was made using a trusted proxy (see *trustProxy*), this is set to the port portion of the `X-Forwarded-Host` header (or approriate default port) if present.

- **protocol**: `string`

  The protocol used for the request.

  Defaults to `"https"` or `"http"` depending on whether ArangoDB is configured to use SSL or not.

  If the request was made using a trusted proxy (see *trustProxy*), this is set to the value of the `X-Forwarded-Proto` header if present.

- **queryParams**: `object`

  An object mapping the names of query parameters of the current route to their validated values.

  For details on how query parameters can be validated by Foxx see the queryParam method of the endpoint object.

- **rawBody**: `Buffer`

  The raw, unparsed, unvalidated request body as a buffer.

- **remoteAddress**: `string`

  The IP of the client that made the request.

  If the request was made using a trusted proxy (see *trustProxy*), this is set to the first IP listed in the `X-Forwarded-For` header if present.

- **remoteAddresses**: `Array<string>`

  A list containing the IP addresses used to make the request.

  Defaults to the value of *remoteAddress* wrapped in an array.

  If the request was made using a trusted proxy (see *trustProxy*), this is set to the list of IPs specified in the `X-Forwarded-For` header if present.

- **remotePort**: `number`

  The listening port of the client that made the request.

  If the request was made using a trusted proxy (see *trustProxy*), this is set to the port specified in the `X-Forwarded-Port` header if present.

- **secure**: `boolean`

  Whether the request was made over a secure connection (i.e. HTTPS).

  This is set to `false` when *protocol* is `"http"` and `true` when *protocol* is `"https"` .

- **suffix**: `string`

  The trailing path relative to the current route if the current route ends in a wildcard.

- **trustProxy**: `boolean`

  Indicates whether the request was made using a trusted proxy. If the origin server's address was specified in the ArangoDB configuration using `--frontend.trusted-proxy` or the service's `trustProxy` setting is enabled, this will be `true` , otherwise it will be `false` .

- **url**: `string`

  The URL of the request.

- **xhr**: `boolean`

  Whether the request indicates it was made within a browser using AJAX.

  This is set to `true` if the `X-Requested-With` header is present and is a case-insensitive match for the value `"xmlhttprequest"` .

Note that this value does not guarantee whether the request was made from inside a browser or whether AJAX was used and is merely a convention established by JavaScript frameworks like jQuery.

accepts

`req.accepts(types): string | false`

`req.accepts(...types): string | false`

`req.acceptsCharsets(charsets): string | false`

`req.acceptsCharsets(...charsets): string | false`

`req.acceptsEncodings(encodings): string | false`

`req.acceptsEncodings(...encodings): string | false`

`req.acceptsLanguages(languages): string | false`

`req.acceptsLanguages(...languages): string | false`

These methods wrap the corresponding content negotiation methods of the accepts module for the current request.

Examples

```
if (req.accepts(['json', 'html']) === 'html') {
  // Client explicitly prefers HTML over JSON
  res.write('<h1>Client prefers HTML</h1>');
} else {
  // Otherwise just send JSON
  res.json({success: true});
}
```

cookie

`req.cookie(name, options): string | null`

Gets the value of a cookie by name.

Arguments

- **name**: `string`

  Name of the cookie.

- **options**: `object` (optional)

  An object with any of the following properties:

  - **secret**: `string` (optional)

    Secret that was used to sign the cookie.

    If a secret is specified, the cookie's signature is expected to be present in a second cookie with the same name and the suffix `.sig`. Otherwise the signature (if present) will be ignored.

  - **algorithm**: `string` (Default: `"sha256"`)

    Algorithm that was used to sign the cookie.

If a string is passed instead of an options object it will be interpreted as the *secret* option.

Returns the value of the cookie or `null` if the cookie is not set or its signature is invalid.

get / header

`req.get(name): string`

`req.header(name): string`

Gets the value of a header by name. You can validate request headers using the header method of the endpoint.

Arguments

- **name**: `string`

  Name of the header.

Returns the header value.

is

`req.is(types): string`

`req.is(...types): string`

This method wraps the (request body) content type detection method of the type-is module for the current request.

Examples

```
const type = req.is('html', 'application/xml', 'application/*+xml');
if (type === false) { // no match
  handleDefault(req.rawBody);
} else if (type === 'html') {
  handleHtml(req.rawBody);
} else { // is XML
  handleXml(req.rawBody);
}
```

json

`req.json(): any`

Attempts to parse the raw request body as JSON and returns the result.

It is generally more useful to define a request body on the endpoint and use the `req.body` property instead.

Returns `undefined` if the request body is empty. May throw a `SyntaxError` if the body could not be parsed.

makeAbsolute

`req.makeAbsolute(path, [query]): string`

Resolves the given path relative to the `req.context.service` 's mount path to a full URL.

Arguments

- **path**: `string`

  The path to resovle.

- **query**: `string | object`

  A string or object with query parameters to add to the URL.

Returns the formatted absolute URL.

params

`req.param(name): any`

Arguments

Looks up a parameter by name, preferring `pathParams` over `queryParams` .

It's probably better style to use the `req.pathParams` or `req.queryParams` objects directly.

- **name**: `string`

  Name of the parameter.

Returns the (validated) value of the parameter.

range

`req.range([size]): Ranges | number`

This method wraps the range header parsing method of the range-parser module for the current request.

Arguments

- **size**: `number` (Default: `Infinity`)

  Length of the satisfiable range (e.g. number of bytes in the full response). If present, ranges exceeding the size will be considered unsatisfiable.

Returns `undefined` if the `Range` header is absent, `-2` if the header is present but malformed, `-1` if the range is invalid (e.g. start offset is larger than end offset) or unsatisfiable for the given size.

Otherwise returns an array of objects with the properties *start* and *end* values for each range. The array has an additional property *type* indicating the request range type.

Examples

```
console.log(req.headers.range); // "bytes=40-80"
const ranges = req.range(100);
console.log(ranges); // [{start: 40, end: 80}]
console.log(ranges.type); // "bytes"
```

reverse

`req.reverse(name, [params]): string`

Looks up the URL of a named route for the given parameters.

Arguments

- **name**: `string`

  Name of the route to look up.

- **params**: `object` (optional)

  An object containing values for the (path or query) parameters of the route.

Returns the URL of the route for the given parameters.

Examples

```
router.get('/items/:id', function (req, res) {
  /* ... */
}, 'getItemById');

router.post('/items', function (req, res) {
  // ...
  const url = req.reverse('getItemById', {id: createdItem._key});
  res.set('location', req.makeAbsolute(url));
});
```

Response objects

The response object specifies the following properties:

- **body**: `Buffer | string`

  Response body as a string or buffer. Can be set directly or using some of the response methods.

- **context**: `Context`

  The service context in which the router was mounted (rather than the context in which the route was defined).

- **headers**: `object`

  The raw headers object.

- **statusCode**: `number`

  Status code of the response. Defaults to `200` (body set and not an empty string or buffer) or `204` (otherwise) if not changed from `undefined`.

attachment

`res.attachment([filename]): this`

Sets the `content-disposition` header to indicate the response is a downloadable file with the given name.

Note: This does not actually modify the response body or access the file system. To send a file from the file system see the `download` or `sendFile` methods.

Arguments

- **filename**: `string` (optional)

  Name of the downloadable file in the response body.

  If present, the extension of the filename will be used to set the response `content-type` if it has not yet been set.

Returns the response object.

cookie

`res.cookie(name, value, [options]): this`

Sets a cookie with the given name.

Arguments

- **name**: `string`

  Name of the cookie.

- **value**: `string`

  Value of the cookie.

- **options**: `object` (optional)

  An object with any of the following properties:

  - **secret**: `string` (optional)

    Secret that will be used to sign the cookie.

    If a secret is specified, the cookie's signature will be stored in a second cookie with the same options, the same name and the suffix `.sig`. Otherwise no signature will be added.

  - **algorithm**: `string` (Default: `"sha256"`)

    Algorithm that will be used to sign the cookie.

  - **ttl**: `number` (optional)

    Time to live of the cookie.

  - **path**: `number` (optional)

    Path of the cookie.

  - **domain**: `number` (optional)

    Domain of the cookie.

  - **secure**: `number` (Default: `false`)

    Whether the cookie should be marked as secure (i.e. HTTPS/SSL-only).

  - **httpOnly**: `boolean` (Default: `false`)

    Whether the cookie should be marked as HTTP-only.

If a string is passed instead of an options object it will be interpreted as the *secret* option.

If a number is passed instead of an options object it will be interpreted as the *ttl* option.

Returns the response object.

download

`res.download(path, [filename]): this`

The equivalent of calling `res.attachment(filename).sendFile(path)` .

Arguments

- **path**: `string`

  Path to the file on the local filesystem to be sent as the response body.

- **filename**: `string` (optional)

  Filename to indicate in the `content-disposition` header.

  If omitted the *path* will be used instead.

Returns the response object.

getHeader

`res.getHeader(name): string`

Gets the value of the header with the given name.

Arguments

- **name**: `string`

  Name of the header to get.

Returns the value of the header or `undefined` .

json

`res.json(data): this`

Sets the response body to the JSON string value of the given data.

Arguments

- **data**: `any`

  The data to be used as the response body.

Returns the response object.

redirect

`res.redirect([status], path): this`

Redirects the response by setting the response `location` header and status code.

Arguments

- **status**: `number | string` (optional)

  Response status code to set.

  If the status code is the string value `"permanent"` it will be treated as the value `301` .

  If the status code is a string it will be converted to a numeric status code using the statuses module first.

  If the status code is omitted but the response status has not already been set, the response status will be set to `302` .

- **path**: `string`

  URL to set the `location` header to.

Returns the response object.

removeHeader

`res.removeHeader(name): this`

Removes the header with the given name from the response.

Arguments

- **name**: `string`

  Name of the header to remove.

Returns the response object.

send

`res.send(data, [type]): this`

Sets the response body to the given data with respect to the response definition for the response's current status code.

Arguments

- **data**: `any`

  The data to be used as the response body. Will be converted according the response definition for the response's current status code (or `200`) in the following way:

  If the data is an ArangoDB result set, it will be converted to an array first.

  If the response definition specifies a model with a `forClient` method, that method will be applied to the data first. If the data is an array and the response definition has the `multiple` flag set, the method will be applied to each entry individually instead.

  Finally the data will be processed by the response type handler to conver the response body to a string or buffer.

- **type**: `string` (Default: `"auto"`)

Content-type of the response body.

If set to `"auto"` the first MIME type specified in the response definition for the response's current status code (or `200`) will be used instead.

If set to `"auto"` and no response definition exists, the MIME type will be determined the following way:

If the data is a buffer the MIME type will be set to binary (`application/octet-stream`).

If the data is an object the MIME type will be set to JSON and the data will be converted to a JSON string.

Otherwise the MIME type will be set to HTML and the data will be converted to a string.

Returns the response object.

sendFile

`res.sendFile(path, [options]): this`

Sends a file from the local filesystem as the response body.

Arguments

- **path**: `string`

  Path to the file on the local filesystem to be sent as the response body.

  If no `content-type` header has been set yet, the extension of the filename will be used to set the value of that header.

- **options**: `object` (optional)

  An object with any of the following properties:

  - **lastModified**: `boolean` (optional)

    If set to `true` or if no `last-modified` header has been set yet and the value is not set to `false` the `last-modified` header will be set to the modification date of the file in milliseconds.

Returns the response object.

Examples

```
// Send the file "favicon.ico" from this service's folder
res.sendFile(module.context.fileName('favicon.ico'));
```

sendStatus

`res.sendStatus(status): this`

Sends a plaintext response for the given status code. The response status will be set to the given status code, the response body will be set to the status message corresponding to that status code.

Arguments

- **status**: `number | string`

  Response status code to set.

  If the status code is a string it will be converted to a numeric status code using the statuses module first.

Returns the response object.

setHeader / set

`res.setHeader(name, value): this`

`res.set(name, value): this`

`res.set(headers): this`

Sets the value of the header with the given name.

Arguments

- **name**: `string`

  Name of the header to set.

- **value**: `string`

  Value to set the header to.

- **headers**: `object`

  Header object mapping header names to values.

Returns the response object.

status

`res.status(status): this`

Sets the response status to the given status code.

Arguments

- **status**: `number | string`

  Response status code to set.

  If the status code is a string it will be converted to a numeric status code using the statuses module first.

Returns the response object.

throw

`res.throw(status, [reason], [options]): void`

Throws an HTTP exception for the given status, which will be handled by Foxx to serve the appropriate JSON error response.

Arguments

- **status**: `number | string`

  Response status code to set.

  If the status code is a string it will be converted to a numeric status code using the statuses module first.

  If the status code is in the 500-range (500-599), its stacktrace will always be logged as if it were an unhandled exception.

  If development mode is enabled, the error's stacktrace will be logged as a warning if the status code is in the 400-range (400-499) or as a regular message otherwise.

- **reason**: `string` (optional)

  Message for the exception.

  If omitted, the status message corresponding to the status code will be used instead.

- **options**: `object` (optional)

  An object with any of the following properties:

  - **cause**: `Error` (optional)

    Cause of the exception that will be logged as part of the error's stacktrace (recursively, if the exception also has a `cause` property and so on).

  - **extra**: `object` (optional)

    Additional properties that will be added to the error response body generated by Foxx.

    If development mode is enabled, an `exception` property will be added to this value containing the error message and a `stacktrace` property will be added containing an array with each line of the error's stacktrace.

If an error is passed instead of an options object it will be interpreted as the *cause* option. If no reason was provided the error's `message` will be used as the reason instead.

Returns nothing.

vary

`res.vary(names): this`

`res.vary(...names): this`

This method wraps the `vary` header manipulation method of the vary module for the current response.

The given names will be added to the response's `vary` header if not already present.

Returns the response object.

Examples

```
res.vary('user-agent');
res.vary('cookie');
res.vary('cookie'); // duplicates will be ignored

// -- or --

res.vary('user-agent', 'cookie');

// -- or --

res.vary(['user-agent', 'cookie']);
```

write

`res.write(data): this`

Appends the given data to the response body.

Arguments

- **data**: `string | Buffer`

  Data to append.

  If the data is a buffer the response body will be converted to a buffer first.

  If the response body is a buffer the data will be converted to a buffer first.

  If the data is an object it will be converted to a JSON string first.

  If the data is any other non-string value it will be converted to a string first.

Returns the response object.

Using GraphQL in Foxx

`const createGraphQLRouter = require('@arangodb/foxx/graphql');`

Foxx bundles the `graphql-sync` module, which is a synchronous wrapper for the official JavaScript GraphQL reference implementation, to allow writing GraphQL schemas directly inside Foxx.

Additionally the `@arangodb/foxx/graphql` lets you create routers for serving GraphQL requests, which closely mimicks the behaviour of the `express-graphql` module.

For more information on `graphql-sync` see the `graphql-js` API reference (note that `graphql-sync` always uses raw values instead of wrapping them in promises).

For an example of a GraphQL schema in Foxx that resolves fields using the database see the GraphQL example service (also available from the Foxx store).

Examples

```
const graphql = require('graphql-sync');
const graphqlSchema = new graphql.GraphQLSchema({
  // ...
});

// Mounting a graphql endpoint directly in a service:
module.context.use('/graphql', createGraphQLRouter({
  schema: graphqlSchema,
  graphiql: true
}));

// Or at the service's root URL:
module.context.use(createGraphQLRouter({
  schema: graphqlSchema,
  graphiql: true
}));

// Or inside an existing router:
router.get('/hello', function (req, res) {
  res.write('Hello world!');
});
router.use('/graphql', createGraphQLRouter({
  schema: graphqlSchema,
  graphiql: true
}));
```

Creating a router

`createGraphQLRouter(options): Router`

This returns a new router object with POST and GET routes for serving GraphQL requests.

Arguments

- **options**: `object`

  An object with any of the following properties:

  - **schema**: `GraphQLSchema`

    A GraphQL Schema object from `graphql-sync`.

  - **context**: `any` (optional)

    The GraphQL context that will be passed to the `graphql()` function from `graphql-sync` to handle GraphQL queries.

  - **rootValue**: `object` (optional)

    The GraphQL root value that will be passed to the `graphql()` function from `graphql-sync` to handle GraphQL queries.

  - **pretty**: `boolean` (Default: `false`)

    If `true` , JSON responses will be pretty-printed.

  - **formatError**: `Function` (optional)

    A function that will be used to format errors produced by `graphql-sync` . If omitted, the `formatError` function from `graphql-sync` will be used instead.

  - **validationRules**: `Array<any>` (optional)

    Additional validation rules queries must satisfy in addition to those defined in the GraphQL spec.

  - **graphiql**: `boolean` (Default: `false`)

    If `true` , the GraphiQL explorer will be served when loaded directly from a browser.

If a GraphQL Schema object is passed instead of an options object it will be interpreted as the *schema* option.

Generated routes

The router handles GET and POST requests to its root path and accepts the following parameters, which can be provided either as query parameters or as the POST request body:

- **query**: `string`

  A GraphQL query that will be executed.

- **variables**: `object | string` (optional)

  An object or a string containing a JSON object with runtime values to use for any GraphQL query variables.

- **operationName**: `string` (optional)

  If the provided `query` contains multiple named operations, this specifies which operation should be executed.

- **raw**: `boolean` (Default: `false`)

  Forces a JSON response even if *graphiql* is enabled and the request was made using a browser.

The POST request body can be provided as JSON or as query string using `application/x-www-form-urlencoded` . A request body passed as `application/graphql` will be interpreted as the `query` parameter.

Session Middleware

`const sessionMiddleware = require('@arangodb/foxx/sessions');`

The session middleware adds the `session` and `sessionStorage` properties to the request object and deals with serializing and deserializing the session as well as extracting session identifiers from incoming requests and injecting them into outgoing responses.

Examples

```
// Create a session middleware
const sessions = sessionsMiddleware({
  storage: module.context.collection('sessions'),
  transport: ['header', 'cookie']
});
// First enable the middleware for this service
module.context.use(sessions);
// Now mount the routers that use the session
const router = createRouter();
module.context.use(router);

router.get('/', function (req, res) {
  res.send(`Hello ${req.session.uid || 'anonymous'}!`);
}, 'hello');

router.post('/login', function (req, res) {
  req.session.uid = req.body;
  req.sessionStorage.save(req.session);
  res.redirect(req.reverse('hello'));
});
.body(['text/plain'], 'Username');
```

Creating a session middleware

`sessionMiddleware(options): Middleware`

Creates a session middleware.

Arguments

- **options**: `Object`

  An object with the following properties:

  - **storage**: `Storage`

    Storage that will be used to persist the sessions.

    The storage is also exposed as the `sessionStorage` on all request objects and as the `storage` property of the middleware.

    If a string or collection is passed instead of a Storage, it will be used to create a Collection Storage.

  - **transport**: `Transport | Array<Transport>`

    Transport or array of transports that will be used to extract the session identifiers from incoming requests and inject them into outgoing responses. When attempting to extract a session identifier, the transports will be used in the order specified until a match is found. When injecting (or clearing) session identifiers, all transports will be invoked.

    The transports are also exposed as the `transport` property of the middleware.

    If the string `"cookie"` is passed instead of a Transport, the Cookie Transport will be used with the default settings instead.

    If the string `"header"` is passed instead of a Transport, the Header Transport will be used with the default settings instead.

  - **autoCreate**: `boolean` (Default: `true`)

    If enabled the session storage's `new` method will be invoked to create an empty session whenever the transport failed to return a session for the incoming request. Otherwise the session will be initialized as `null` .

Returns the session middleware.

Session Storages

Session storages are used by the sessions middleware to persist sessions across requests. Session storages must implement the `fromClient` and `forClient` methods and can optionally implement the `new` method.

The built-in session storages generally provide the following attributes:

- **uid**: `string` (Default: `null`)

  A unique identifier indicating the active user.

- **created**: `number` (Default: `Date.now()`)

  The numeric timestamp of when the session was created.

- **data**: `any` (Default: `null`)

  Arbitrary data to persisted in the session.

new

`storage.new(): Session`

Generates a new session object representing an empty session. The empty session object should not be persisted unless necessary. The return value will be exposed by the middleware as the `session` property of the request object if no session identifier was returned by the session transports and auto-creation is not explicitly disabled in the session middleware.

Examples

```
new() {
  return {
    uid: null,
    created: Date.now(),
    data: null
  };
}
```

fromClient

`storage.fromClient(sid): Session | null`

Resolves or deserializes a session identifier to a session object.

Arguments

- **sid**: `string`

  Session identifier to resolve or deserialize.

Returns a session object representing the session with the given session identifier that will be exposed by the middleware as the `session` property of the request object. This method will only be called if any of the session transports returned a session identifier. If the session identifier is invalid or expired, the method should return a `null` value to indicate no matching session.

Examples

```
fromClient(sid) {
  return db._collection('sessions').firstExample({_key: sid});
}
```

forClient

`storage.forClient(session): string | null`

Derives a session identifier from the given session object.

Arguments

- **session**: `Session`

  Session to derive a session identifier from.

Returns a session identifier for the session represented by the given session object. This method will be called with the `session` property of the request object unless that property is empty (e.g. `null`).

Examples

```
forClient(session) {
  if (!session._key) {
    const meta = db._collection('sessions').save(session);
    return meta._key;
  }
  db._collection('sessions').replace(session._key, session);
  return session._key;
}
```

Collection Session Storage

`const collectionStorage = require('@arangodb/foxx/sessions/storages/collection');`

The collection session storage persists sessions to a collection in the database.

Creating a storage

`collectionStorage(options): Storage`

Creates a Storage that can be used in the sessions middleware.

Arguments

- **options**: `Object`

  An object with the following properties:

  - **collection**: `ArangoCollection`

    The collection that should be used to persist the sessions. If a string is passed instead of a collection it is assumed to be the fully qualified name of a collection in the current database.

  - **ttl**: `number` (Default: `60 * 60`)

    The time in seconds since the last update until a session will be considered expired.

  - **pruneExpired**: `boolean` (Default: `false`)

    Whether expired sessions should be removed from the collection when they are accessed instead of simply being ignored.

  - **autoUpdate**: `boolean` (Default: `true`)

    Whether sessions should be updated in the collection every time they are accessed to keep them from expiring. Disabling this option **will improve performance** but means you will have to take care of keeping your sessions alive yourself.

If a string or collection is passed instead of an options object, it will be interpreted as the *collection* option.

prune

`storage.prune(): Array<string>`

Removes all expired sessions from the collection. This method should be called even if the *pruneExpired* option is enabled to clean up abandoned sessions.

Returns an array of the keys of all sessions that were removed.

save

`storage.save(session): Session`

Saves (replaces) the given session object in the collection. This method needs to be invoked explicitly after making changes to the session or the changes will not be persisted. Assigns a new `_key` to the session if it previously did not have one.

Arguments

- **session**: `Session`

  A session object.

Returns the modified session.

clear

`storage.clear(session): boolean`

Removes the session from the collection. Has no effect if the session was already removed or has not yet been saved to the collection (i.e. has no `_key`).

Arguments

- **session**: `Session`

  A session object.

Returns `true` if the session was removed or `false` if it had no effect.

JWT Session Storage

`const jwtStorage = require('@arangodb/foxx/sessions/storages/jwt');`

The JWT session storage converts sessions to and from JSON Web Tokens.

Examples

```
// Pass in a secure secret from the Foxx configuration
const secret = module.context.configuration.jwtSecret;
const sessions = sessionsMiddleware({
  storage: jwtStorage(secret),
  transport: 'header'
});
module.context.use(sessions);
```

Creating a storage

`jwtStorage(options): Storage`

Creates a Storage that can be used in the sessions middleware.

Note: while the "none" algorithm (i.e. no signature) is supported this dummy algorithm provides no security and allows clients to make arbitrary modifications to the payload and should not be used unless you are certain you specifically need it.

Arguments

- **options**: `Object`

  An object with the following properties:

  - **algorithm**: `string` (Default: `"HS512"`)

    The algorithm to use for signing the token.

    Supported values:

    - `"HS256"` (HMAC-SHA256)
    - `"HS384"` (HMAC-SHA384)
    - `"HS512"` (HMAC-SHA512)
    - `"none"` (no signature)
  - **secret**: `string`

    The secret to use for signing the token.

    This field is forbidden when using the "none" algorithm but required otherwise.

  - **ttl**: `number` (Default: `3600`)

    The maximum lifetime of the token in seconds. You may want to keep this short as a new token is generated on every request allowing clients to refresh tokens automatically.

  - **verify**: `boolean` (Default: `true`)

    If set to `false` the signature will not be verified but still generated (unless using the "none" algorithm).

If a string is passed instead of an options object it will be interpreted as the *secret* option.

Session Transports

Session transports are used by the sessions middleware to store and retrieve session identifiers in requests and responses. Session transports must implement the `get` and/or `set` methods and can optionally implement the `clear` method.

get

`transport.get(request): string | null`

Retrieves a session identifier from a request object.

If present this method will automatically be invoked for each transport until a transport returns a session identifier.

Arguments

- **request**: `Request`

  Request object to extract a session identifier from.

Returns the session identifier or `null` if the transport can not find a session identifier in the request.

Examples

```
get(req) {
  return req.get('x-session-id') || null;
}
```

set

`transport.set(response, sid): void`

Attaches a session identifier to a response object.

If present this method will automatically be invoked at the end of a request regardless of whether the session was modified or not.

Arguments

- **response**: `Response`

  Response object to attach a session identifier to.

- **sid**: `string`

  Session identifier to attach to the response.

Returns nothing.

Examples

```
set(res) {
  res.set('x-session-id', value);
}
```

clear

`transport.clear(response): void`

Attaches a payload indicating that the session has been cleared to the response object. This can be used to clear a session cookie when the session has been destroyed (e.g. during logout).

If present this method will automatically be invoked instead of `set` when the `req.session` attribute was removed by the route handler.

Arguments

- **response**: `Response`

  Response object to remove the session identifier from.

Returns nothing.

Cookie Session Transport

`const cookieTransport = require('@arangodb/foxx/sessions/transports/cookie');`

The cookie transport stores session identifiers in cookies on the request and response object.

Examples

```
// Pass in a secure secret from the Foxx configuration
const secret = module.context.configuration.cookieSecret;
const sessions = sessionsMiddleware({
  storage: module.context.collection('sessions'),
  transport: cookieTransport({
    name: 'FOXXSESSID',
    ttl: 60 * 60 * 24 * 7, // one week in seconds
    algorithm: 'sha256',
    secret: secret
  })
});
module.context.use(sessions);
```

Creating a transport

`cookieTransport([options]): Transport`

Creates a Transport that can be used in the sessions middleware.

Arguments

- **options**: `Object` (optional)

  An object with the following properties:

  - **name**: `string` (Default: `"sid"`)

    The name of the cookie.

  - **ttl**: `number` (optional)

    Cookie lifetime in seconds.

  - **algorithm**: `string` (optional)

    The algorithm used to sign and verify the cookie. If no algorithm is specified, the cookie will not be signed or verified. See the cookie method on the response object.

  - **secret**: `string` (optional)

    Secret to use for the signed cookie. Will be ignored if no algorithm is provided.

If a string is passed instead of an options object, it will be interpreted as the *name* option.

Header Session Transport

`const headerTransport = require('@arangodb/foxx/sessions/transports/header');`

The header transport stores session identifiers in headers on the request and response objects.

Examples

```
const sessions = sessionsMiddleware({
  storage: module.context.collection('sessions'),
  transport: headerTransport('X-FOXXSESSID')
});
module.context.use(sessions);
```

Creating a transport

`headerTransport([options]): Transport`

Creates a Transport that can be used in the sessions middleware.

Arguments

- **options**: `Object` (optional)

  An object with the following properties:

  - **name**: `string` (Default: `X-Session-Id`)

    Name of the header that contains the session identifier (not case sensitive).

If a string is passed instead of an options object, it will be interpreted as the *name* option.

Static file assets

The most flexible way to serve files in your Foxx service is to simply pass them through in your router using the context object's `fileName` method and the response object's `sendFile` method:

```
router.get('/some/filename.png', function (req, res) {
  const filePath = module.context.fileName('some-local-filename.png');
  res.sendFile(filePath);
});
```

While allowing for greater control of how the file should be sent to the client and who should be able to access it, doing this for all your static assets can get tedious.

Alternatively you can specify file assets that should be served by your Foxx service directly in the service manifest using the `files` attribute:

```
"files": {
  "/some/filename.png": {
    "path": "some-local-filename.png",
    "type": "image/png",
    "gzip": false
  },
  "/favicon.ico": "bookmark.ico",
  "/static": "my-assets-folder"
}
```

Each entry in the `files` attribute can represent either a single file or a directory. When serving entire directories, the key acts as a prefix and requests to that prefix will be resolved within the given directory.

Options

- **path**: `string`

  The relative path of the file or folder within the service.

- **type**: `string` (optional)

  The MIME content type of the file. Defaults to an intelligent guess based on the filename's extension.

- **gzip**: `boolean` (Default: `false`)

  If set to `true` the file will be served with gzip-encoding if supported by the client. This can be useful when serving text files like client-side JavaScript, CSS or HTML.

If a string is provided instead of an object, it will be interpreted as the *path* option.

Writing tests

Foxx provides out of the box support for running tests against an installed service using the Mocha test runner.

Test files have full access to the service context and all ArangoDB APIs but like scripts can not define Foxx routes.

Running tests

An installed service's tests can be executed from the administrative web interface:

1. Open the "Services" tab of the web interface
2. Click on the installed service to be tested
3. Click on the "Settings" tab
4. Click on the flask icon in the top right
5. Accept the confirmation dialog

Note that running tests in a production database is not recommended and may result in data loss if the tests access the database.

When running a service in development mode special care needs to be taken as performing requests to the service's own routes as part of the test suites may result in tests being executed while the database is in an inconsistent state, leading to unexpected behaviour.

Test file paths

In order to tell Foxx about files containing test suites, one or more patterns need to be specified in the `tests` option of the service manifest:

```
{
  "tests": [
    "**/test_*.js",
    "**/*_test.js"
  ]
}
```

These patterns can be either relative file paths or "globstar" patterns where

- `*` matches zero or more characters in a filename
- `**` matches zero or more nested directories

For example, given the following directory structure:

```
++ test/
|++ a/
||+- a1.js
||+- a2.js
||+- test.js
|+- b.js
|+- c.js
|+- d_test.js
+- e_test.js
+- test.js
```

The following patterns would match the following files:

```
test.js:
  test.js

test/*.js:
  /test/b.js
  /test/c.js
  /test/d_test.js

test/**/*.js:
  /test/a/a1.js
  /test/a/a2.js
  /test/a/test.js
  /test/b.js
  /test/c.js
  /test/d_test.js

**/test.js:
  /test/a/test.js

**/*test.js:
  /test/a/test.js
  /test/d_test.js
  /e_test.js
  /test.js
```

Even if multiple patterns match the same file the tests in that file will only be run once.

The order of tests is always determined by the file paths, not the order in which they are matched or specified in the manifest.

Test structure

Mocha test suites can be defined using one of three interfaces: BDD, TDD or Exports.

The QUnit interface of Mocha is not supported in ArangoDB.

Like all ArangoDB code, test code is always synchronous.

BDD interface

The BDD interface defines test suites using the `describe` function and each test case is defined using the `it` function:

```
'use strict';
const assert = require('assert');
const trueThing = true;

describe('True things', () => {
  it('are true', () => {
    assert.equal(trueThing, true);
  });
});
```

The BDD interface also offers the alias `context` for `describe` and `specify` for `it` .

Test fixtures can be handled using `before` and `after` for suite-wide fixtures and `beforeEach` and `afterEach` for per-test fixtures:

```
describe('False things', () => {
  let falseThing;
  before(() => {
    falseThing = !true;
  });
  it('are false', () => {
    assert.equal(falseThing, false);
  });
});
```

TDD interface

The TDD interface defines test suites using the `suite` function and each test case is defined using the `test` function:

```
'use strict';
const assert = require('assert');
const trueThing = true;

suite('True things', () => {
  test('are true', () => {
    assert.equal(trueThing, true);
  });
});
```

Test fixtures can be handled using `suiteSetup` and `suiteTeardown` for suite-wide fixtures and `setup` and `teardown` for per-test fixtures:

```
suite('False things', () => {
  let falseThing;
  suiteSetup(() => {
    falseThing = !true;
  });
  test('are false', () => {
    assert.equal(falseThing, false);
  });
});
```

Exports interface

The Exports interface defines test cases as methods of plain object properties of the `module.exports` object:

```
'use strict';
const assert = require('assert');
const trueThing = true;

exports['True things'] = {
  'are true': function() {
    assert.equal(trueThing, true);
  }
};
```

The keys `before` , `after` , `beforeEach` and `afterEach` are special-cased and behave like the corresponding functions in the BDD interface:

```
let falseThing;
exports['False things'] = {
  before () {
    falseThing = false;
  },
  'are false': function() {
    assert.equal(falseThing, false);
  }
};
```

Assertions

ArangoDB provides two bundled modules to define assertions:

- `assert` corresponds to the Node.js `assert` module, providing low-level assertions that can optionally specify an error message.
- `chai` is the popular Chai Assertion Library, providing both BDD and TDD style assertions using a familiar syntax.

Foxx scripts and queued jobs

Foxx lets you define scripts that can be executed as part of the installation and removal process, invoked manually or scheduled to run at a later time using the job queue.

To register your script, just add a `scripts` section to your service manifest:

```
{
  ...
  "scripts": {
    "setup": "scripts/setup.js",
    "send-mail": "scripts/send-mail.js"
  }
  ...
}
```

The scripts you define in your service manifest can be invoked from the web interface in the service's settings page with the *Scripts* dropdown.

You can also use the scripts as queued jobs:

```
'use strict';
const queues = require('@arangodb/foxx/queues');
queues.get('default').push(
  {mount: '/my-service-mount-point', name: 'send-mail'},
  {to: 'user@example.com', body: 'Hello'}
);
```

Script arguments and return values

If the script was invoked with any arguments, you can access them using the `module.context.argv` array.

To return data from your script, you can assign the data to `module.exports` as usual. Please note that this data will be converted to JSON.

Any errors raised by the script will be handled depending on how the script was invoked:

- if the script was invoked from the HTTP API (e.g. using the web interface), it will return an error response using the exception's `statusCode` property if specified or 500.
- if the script was invoked from a Foxx job queue, the job's failure counter will be incremented and the job will be rescheduled or marked as failed if no attempts remain.

Examples

Let's say you want to define a script that takes two numeric values and returns the result of multiplying them:

```
'use strict';
const assert = require('assert');
const argv = module.context.argv;

assert.equal(argv.length, 2, 'Expected exactly two arguments');
assert.equal(typeof argv[0], 'number', 'Expected first argument to be a number');
assert.equal(typeof argv[1], 'number', 'Expected second argument to be a number');

module.exports = argv[0] * argv[1];
```

Lifecycle Scripts

Foxx recognizes lifecycle scripts if they are defined and will invoke them during the installation, update and removal process of the service if you want.

The following scripts are currently recognized as lifecycle scripts by their name: `"setup"` and `"teardown"` .

Setup Script

The setup script will be executed without arguments during the installation of your Foxx service.

The setup script is typically used to create collections your service needs or insert seed data like initial administrative user accounts and so on.

Examples

```
'use strict';
const db = require('@arangodb').db;
const textsCollectionName = module.context.collectionName('texts');
// `textsCollectionName` is now the prefixed name of this service's "texts" collection.
// e.g. "example_texts" if the service has been mounted at `/example`

if (db._collection(textsCollectionName) === null) {
  const collection = db._create(textsCollectionName);

  collection.save({text: 'entry 1 from collection texts'});
  collection.save({text: 'entry 2 from collection texts'});
  collection.save({text: 'entry 3 from collection texts'});
} else {
  console.log(`collection ${texts} already exists. Leaving it untouched.`);
}
```

Teardown Script

The teardown script will be executed without arguments during the removal of your Foxx service.

It can also optionally be executed before upgrading an service.

This script typically removes the collections and/or documents created by your service's setup script.

Examples

```
'use strict';
const db = require('@arangodb').db;

const textsCollection = module.context.collection('texts');

if (textsCollection) {
  textsCollection.drop();
}
```

Queues

Foxx allows defining job queues that let you perform slow or expensive actions asynchronously. These queues can be used to send e-mails, call external APIs or perform other actions that you do not want to perform directly or want to retry on failure.

enable or disable the Foxx queues feature `--foxx.queues flag` If *true*, the Foxx queues will be available and jobs in the queues will be executed asynchronously. The default is *true*. When set to `false` the queue manager will be disabled and any jobs are prevented from being processed, which may reduce CPU load a bit. Please note that Foxx job queues are database-specific. Queues and jobs are always relative to the database in which they are created or accessed.

poll interval for Foxx queues `--foxx.queues-poll-interval value` The poll interval for the Foxx queues manager. The value is specified in seconds. Lower values will mean more immediate and more frequent Foxx queue job execution, but will make the queue thread wake up and query the queues more often. When set to a low value, the queue thread might cause CPU load. The default is *1* second. If Foxx queues are not used much, then this value may be increased to make the queues thread wake up less. For the low-level functionality see the chapter on the task management module.

Creating or updating a queue

`queues.create(name, [maxWorkers]): Queue`

Returns the queue for the given name. If the queue does not exist, a new queue with the given name will be created. If a queue with the given name already exists and maxWorkers is set, the queue's maximum number of workers will be updated. The queue will be created in the current database.

Arguments

- **name**: `string`

  Name of the queue to create.

- **maxWorkers**: `number` (Default: `1`)

  The maximum number of workers.

Examples

```
// Create a queue with the default number of workers (i.e. one)
const queue1 = queues.create("my-queue");
// Create a queue with a given number of workers
const queue2 = queues.create("another-queue", 2);
// Update the number of workers of an existing queue
const queue3 = queues.create("my-queue", 10);
// queue1 and queue3 refer to the same queue
assertEqual(queue1, queue3);
```

Fetching an existing queue

`queues.get(name): Queue`

Returns the queue for the given name. If the queue does not exist an exception is thrown instead.

The queue will be looked up in the current database.

Arguments

- **name**: `string`

  Name of the queue to fetch.

Examples

If the queue does not yet exist an exception is thrown:

```
queues.get("some-queue");
// Error: Queue does not exist: some-queue
//     at ...
```

Otherwise the queue will be returned:

```
const queue1 = queues.create("some-queue");
const queue2 = queues.get("some-queue");
assertEqual(queue1, queue2);
```

Deleting a queue

`queues.delete(name): boolean`

Returns `true` if the queue was deleted successfully. If the queue did not exist, it returns `false` instead. The queue will be looked up and deleted in the current database.

When a queue is deleted, jobs on that queue will no longer be executed.

Deleting a queue will not delete any jobs on that queue.

Arguments

- **name**: `string`

Name of the queue to delete.

Examples

```
const queue = queues.create("my-queue");
queues.delete("my-queue"); // true
queues.delete("my-queue"); // false
```

Adding a job to a queue

`queue.push(script, data, [opts]): string`

The job will be added to the specified queue in the current database.

Returns the job id.

Arguments

- **script**: `object`

  A job type definition, consisting of an object with the following properties:

  - **name**: `string`

    Name of the script that will be invoked.

  - **mount**: `string`

    Mount path of the service that defines the script.

  - **backOff**: `Function | number` (Default: `1000`)

    Either a function that takes the number of times the job has failed before as input and returns the number of milliseconds to wait before trying the job again, or the delay to be used to calculate an exponential back-off, or `0` for no delay.

  - **maxFailures**: `number | Infinity` (Default: `0`):

    Number of times a single run of a job will be re-tried before it is marked as `"failed"` . A negative value or `Infinity` means that the job will be re-tried on failure indefinitely.

  - **schema**: `Schema` (optional)

    Schema to validate a job's data against before enqueuing the job.

  - **preprocess**: `Function` (optional)

    Function to pre-process a job's (validated) data before serializing it in the queue.

  - **repeatTimes**: `Function` (Default: `0`)

    If set to a positive number, the job will be repeated this many times (not counting recovery when using *maxFailures*). If set to a negative number or `Infinity` , the job will be repeated indefinitely. If set to `0` the job will not be repeated.

  - **repeatUntil**: `number | Date` (optional)

    If the job is set to automatically repeat, this can be set to a timestamp in milliseconds (or `Date` instance) after which the job will no longer repeat. Setting this value to zero, a negative value or `Infinity` has no effect.

  - **repeatDelay**: `number` (Default: `0`)

    If the job is set to automatically repeat, this can be set to a non-negative value to set the number of milliseconds for which the job will be delayed before it is started again.

- **data**: `any`

  Job data of the job; must be serializable to JSON.

- **opts**: `object` (optional)

  Object with any of the following properties:

- **success**: `Function` (optional)

  Function to be called after the job has been completed successfully.

- **failure**: `Function` (optional)

  Function to be called after the job has failed too many times.

- **delayUntil**: `number | Date` (Default: `Date.now()`)

  Timestamp in milliseconds (or `Date` instance) until which the execution of the job should be delayed.

- **backOff**: `Function | number` (Default: `1000`)

  See *script.backOff*.

- **maxFailures**: `number | Infinity` (Default: `0`):

  See *script.maxFailures*.

- **repeatTimes**: `Function` (Default: `0`)

  See *script.repeatTimes*.

- **repeatUntil**: `number | Date` (optional)

  See *script.repeatUntil*.

- **repeatDelay**: `number` (Default: `0`)

  See *script.repeatDelay*.

Note that if you pass a function for the *backOff* calculation, *success* callback or *failure* callback options the function will be serialized to the database as a string and therefore must not rely on any external scope or external variables.

When the job is set to automatically repeat, the *failure* callback will only be executed when a run of the job has failed more than *maxFailures* times. Note that if the job fails and *maxFailures* is set, it will be rescheduled according to the *backOff* until it has either failed too many times or completed successfully before being scheduled according to the *repeatDelay* again. Recovery attempts by *maxFailures* do not count towards *repeatTimes*.

The *success* and *failure* callbacks receive the following arguments:

- **result**: `any`

  The return value of the script for the current run of the job.

- **jobData**: `any`

  The data passed to this method.

- **job**: `object`

  ArangoDB document representing the job's current state.

Examples

Let's say we have an service mounted at `/mailer` that provides a script called `send-mail` :

```
'use strict';
const queues = require('@arangodb/foxx/queues');
const queue = queues.create('my-queue');
queue.push(
  {mount: '/mailer', name: 'send-mail'},
  {to: 'hello@example.com', body: 'Hello world'}
);
```

This will *not* work, because `log` was defined outside the callback function (the callback must be serializable to a string):

```
// WARNING: THIS DOES NOT WORK!
'use strict';
const queues = require('@arangodb/foxx/queues');
const queue = queues.create('my-queue');
const log = require('console').log; // outside the callback's function scope
queue.push(
  {mount: '/mailer', name: 'send-mail'},
  {to: 'hello@example.com', body: 'Hello world'},
  {success: function () {
    log('Yay!'); // throws 'log is not defined'
  }}
);
```

Here's an example of a job that will be executed every 5 seconds until tomorrow:

```
'use strict';
const queues = require('@arangodb/foxx').queues;
const queue = queues.create('my-queue');
queue.push(
  {mount: '/mailer', name: 'send-mail'},
  {to: 'hello@example.com', body: 'Hello world'},
  {
    repeatTimes: Infinity,
    repeatUntil: Date.now() + (24 * 60 * 60 * 1000),
    repeatDelay: 5 * 1000
  }
);
```

Fetching a job from the queue

`queue.get(jobId): Job`

Creates a proxy object representing a job with the given job id.

The job will be looked up in the specified queue in the current database.

Returns the job for the given jobId. Properties of the job object will be fetched whenever they are referenced and can not be modified.

Arguments

- **jobId**: `string`

  The id of the job to create a proxy object for.

Examples

```
const jobId = queue.push({mount: '/logger', name: 'log'}, 'Hello World!');
const job = queue.get(jobId);
assertEqual(job.id, jobId);
```

Deleting a job from the queue

`queue.delete(jobId): boolean`

Deletes a job with the given job id. The job will be looked up and deleted in the specified queue in the current database.

Arguments

- **jobId**: `string`

  The id of the job to delete.

Returns `true` if the job was deleted successfully. If the job did not exist it returns `false` instead.

Fetching an array of jobs in a queue

Examples

```
const logScript = {mount: '/logger', name: 'log'};
queue.push(logScript, 'Hello World!', {delayUntil: Date.now() + 50});
assertEqual(queue.pending(logScript).length, 1);
// 50 ms later...
assertEqual(queue.pending(logScript).length, 0);
assertEqual(queue.progress(logScript).length, 1);
// even later...
assertEqual(queue.progress(logScript).length, 0);
assertEqual(queue.complete(logScript).length, 1);
```

Fetching an array of pending jobs in a queue

`queue.pending([script]): Array<string>`

Returns an array of job ids of jobs in the given queue with the status `"pending"` , optionally filtered by the given job type. The jobs will be looked up in the specified queue in the current database.

Arguments

- **script**: `object` (optional)

  An object with the following properties:

  - **name**: `string`
  Name of the script.

  - **mount**: `string`
  Mount path of the service defining the script.

Fetching an array of jobs that are currently in progress

`queue.progress([script])`

Returns an array of job ids of jobs in the given queue with the status `"progress"` , optionally filtered by the given job type. The jobs will be looked up in the specified queue in the current database.

Arguments

- **script**: `object` (optional)

  An object with the following properties:

  - **name**: `string`
  Name of the script.

  - **mount**: `string`
  Mount path of the service defining the script.

Fetching an array of completed jobs in a queue

`queue.complete([script]): Array<string>`

Returns an array of job ids of jobs in the given queue with the status `"complete"` , optionally filtered by the given job type. The jobs will be looked up in the specified queue in the current database.

Arguments

- **script**: `object` (optional)

  An object with the following properties:

  - **name**: `string`
  Name of the script.

  - **mount**: `string`
  Mount path of the service defining the script.

Fetching an array of failed jobs in a queue

`queue.failed([script]): Array<string>`

Returns an array of job ids of jobs in the given queue with the status `"failed"`, optionally filtered by the given job type. The jobs will be looked up in the specified queue in the current database.

Arguments

- **script**: `object` (optional)

  An object with the following properties:

  - **name**: `string`
  Name of the script.

  - **mount**: `string`
  Mount path of the service defining the script.

Fetching an array of all jobs in a queue

`queue.all([script]): Array<string>`

Returns an array of job ids of all jobs in the given queue, optionally filtered by the given job type. The jobs will be looked up in the specified queue in the current database.

Arguments

- **script**: `object` (optional)

  An object with the following properties:

  - **name**: `string`
  Name of the script.

  - **mount**: `string`
  Mount path of the service defining the script.

Aborting a job

`job.abort(): void`

Aborts a non-completed job.

Sets a job's status to `"failed"` if it is not already `"complete"`, without calling the job's *onFailure* callback.

Migrating 2.x services to 3.0

When migrating services from older versions of ArangoDB it is generally recommended you make sure they work in legacy compatibility mode, which can also serve as a stop-gap solution.

This chapter outlines the major differences in the Foxx API between ArangoDB 2.8 and ArangoDB 3.0.

General changes

The `console` object in later versions of ArangoDB 2.x implemented a special Foxx console API and would optionally log messages to a collection. ArangoDB 3.0 restores the original behaviour where `console` is the same object available from the console module.

Migrating from pre-2.8

When migrating from a version older than ArangoDB 2.8 please note that starting with ArangoDB 2.8 the behaviour of the `require` function more closely mimics the behaviour observed in Node.js and module bundlers for browsers, e.g.:

In a file `/routes/examples.js` (relative to the root folder of the service):

- `require('./my-module')` will be attempted to be resolved in the following order:
    1. `/routes/my-module` (relative to service root)
    2. `/routes/my-module.js` (relative to service root)
    3. `/routes/my-module.json` (relative to service root)
    4. `/routes/my-module/index.js` (relative to service root)
    5. `/routes/my-module/index.json` (relative to service root)
- `require('lodash')` will be attempted to be resolved in the following order:
    1. `/routes/node_modules/lodash` (relative to service root)
    2. `/node_modules/lodash` (relative to service root)
    3. ArangoDB module `lodash`
    4. Node compatibility module `lodash`
    5. Bundled NPM module `lodash`
- `require('/abs/path')` will be attempted to be resolved in the following order:
    1. `/abs/path` (relative to file system root)
    2. `/abs/path.js` (relative to file system root)
    3. `/abs/path.json` (relative to file system root)
    4. `/abs/path/index.js` (relative to file system root)
    5. `/abs/path/index.json` (relative to file system root)

This behaviour is incompatible with the source code generated by the Foxx generator in the web interface before ArangoDB 2.8.

Note: The `org/arangodb` module is aliased to the new name `@arangodb` in ArangoDB 3.0.0 and the `@arangodb` module was aliased to the old name `org/arangodb` in ArangoDB 2.8.0. Either one will work in 2.8 and 3.0 but outside of legacy services you should use `@arangodb` going forward.

Foxx queue

In ArangoDB 2.6 Foxx introduced a new way to define queued jobs using Foxx scripts to replace the function-based job type definitions which were causing problems when restarting the server. The function-based jobs have been removed in 2.7 and are no longer supported at all.

CoffeeScript

ArangoDB 3.0 no longer provides built-in support for CoffeeScript source files, even in legacy compatibility mode. If you want to use an alternative language like CoffeeScript, make sure to pre-compile the raw source files to JavaScript and use the compiled JavaScript files in the service.

The request module

The `@arangodb/request` module when used with the `json` option previously overwrote the string in the `body` property of the response object of the response with the parsed JSON body. In 2.8 this was changed so the parsed JSON body is added as the `json` property of the response object in addition to overwriting the `body` property. In 3.0 and later (including legacy compatibility mode) the `body` property is no longer overwritten and must use the `json` property instead. Note that this only affects code using the `json` option when making the request.

Bundled NPM modules

The bundled NPM modules have been upgraded and may include backwards-incompatible changes, especially the API of `joi` has changed several times. If in doubt you should bundle your own versions of these modules to ensure specific versions will be used.

The utility module `lodash` is now available and should be used instead of `underscore`, but both modules will continue to be provided.

Manifest

Many of the fields that were required in ArangoDB 2.x are now optional and can be safely omitted.

To avoid compatibility problems with future versions of ArangoDB you should always specify the `engines` field, e.g.:

```
{
  "engines": {
    "arangodb": "^3.0.0"
  }
}
```

Controllers & exports

Previously Foxx distinguished between `exports` and `controllers`, each of which could be specified as an object. In ArangoDB 3.0 these have been merged into a single `main` field specifying an entry file.

The easiest way to migrate services using multiple exports and/or controllers is to create a separate entry file that imports these files:

Old (manifest.json):

```
{
  "exports": {
    "doodads": "doodads.js",
    "dingbats": "dingbats.js"
  },
  "controllers": {
    "/doodads": "routes/doodads.js",
    "/dingbats": "routes/dingbats.js",
    "/": "routes/root.js"
  }
}
```

New (manifest.json):

```
{
  "main": "index.js"
}
```

New (index.js):

```
'use strict';
module.context.use('/doodads', require('./routes/doodads'));
module.context.use('/dingbats', require('./routes/dingbats'));
module.context.use('/', require('./routes/root'));
module.exports = {
  doodads: require('./doodads'),
  dingbats: require('./dingbats')
};
```

Index redirect

If you previously did not define the `defaultDocument` field, please note that in ArangoDB 3.0 the field will no longer default to the value `index.html` when omitted:

Old:

```
{
  // no defaultDocument
}
```

New:

```
{
  "defaultDocument": "index.html"
}
```

This also means it is no longer necessary to specify the `defaultDocument` field with an empty value to prevent the redirect and be able to serve requests at the `/` (root) path of the mount point:

Old:

```
{
  "defaultDocument": ""
}
```

New:

```
{
  // no defaultDocument
}
```

Assets

The `assets` field is no longer supported in ArangoDB 3.0 outside of legacy compatibility mode.

If you previously used the field to serve individual files as-is you can simply use the `files` field instead:

Old:

```
{
  "assets": {
    "client.js": {
      "files": ["assets/client.js"],
      "contentType": "application/javascript"
    }
  }
}
```

New:

```
{
  "files": {
    "client.js": {
      "path": "assets/client.js",
      "type": "application/javascript"
    }
  }
}
```

If you relied on being able to specify multiple files that should be concatenated, you will have to use build tools outside of ArangoDB to prepare these files accordingly.

Root element

The `rootElement` field is no longer supported and has been removed entirely.

If your controllers relied on this field being available you need to adjust your schemas and routes to be able to handle the full JSON structure of incoming documents.

System services

The `isSystem` field is no longer supported. The presence or absence of the field had no effect in most recent versions of ArangoDB 2.x and has now been removed entirely.

The application context

The global `applicationContext` variable available in Foxx modules has been replaced with the `context` attribute of the `module` variable. For consistency it is now referred to as the *service* context throughout this documentation.

Some methods of the service context have changed in ArangoDB 3.0:

- `fileName()` now behaves like `path()` did in ArangoDB 2.x
- `path()` has been removed (use `fileName()` instead)
- `foxxFileName()` has been removed (use `fileName()` instead)

Additionally the `version` and `name` attributes have been removed and can now only be accessed via the `manifest` attribute (as `manifest.version` and `manifest.name`). Note that the corresponding manifest fields are now optional and may be omitted.

The `options` attribute has also been removed as it should be considered an implementation detail. You should instead access the `dependencies` and `configuration` attributes directly.

The internal `_prefix` attribute (which was an alias for `basePath`) and the internal `comment` and `clearComments` methods (which were used by the magical documentation comments in ArangoDB 2.x) have also been removed.

The internal `_service` attribute (which provides access to the service itself) has been renamed to `service` .

Repositories and models

Previously Foxx was heavily built around the concept of repositories and models, which provided complex but rarely necessary abstractions on top of ArangoDB collections and documents. In ArangoDB 3.0 these have been removed entirely.

Repositories vs collections

Repositories mostly wrapped methods that already existed on ArangoDB collection objects and primarily dealt with converting between plain ArangoDB documents and Foxx model instances. In ArangoDB 3.0 you can simply use these collections directly and treat documents as plain JavaScript objects.

Old:

```
'use strict';
const Foxx = require('org/arangodb/foxx');
const myRepo = new Foxx.Repository(
  applicationContext.collection('myCollection'),
  {model: Foxx.Model}
);

// ...

const models = myRepo.byExample({color: 'green'});
res.json(models.map(function (model) {
  return model.forClient();
}));
```

New:

```
'use strict';
const myDocs = module.context.collection('myCollection');

// ...

const docs = myDocs.byExample({color: 'green'});
res.json(docs);
```

Schema validation

The main purpose of models in ArangoDB 2.x was to validate incoming data using joi schemas. In more recent versions of ArangoDB 2.x it was already possible to pass these schemas directly in most places where a model was expected as an argument. The only difference is that schemas should now be considered the default.

If you previously relied on the automatic validation of Foxx model instances when setting attributes or instantiating models from untrusted data, you can simply use the schema's `validate` method directly.

Old:

```
'use strict';
const joi = require('joi');
const mySchema = {
  name: joi.string().required(),
  size: joi.number().required()
};
const Foxx = require('org/arangodb/foxx');
const MyModel = Foxx.Model.extend({schema: mySchema});

// ...

const model = new MyModel(req.json());
if (!model.isValid) {
  res.status(400);
  res.write('Bad request');
  return;
}
```

New:

```
'use strict';
const joi = require('joi');
// Note this is now wrapped in a joi.object()
const mySchema = joi.object({
  name: joi.string().required(),
  size: joi.number().required()
}).required();

// ...

const result = mySchema.validate(req.body);
if (result.errors) {
  res.status(400);
  res.write('Bad request');
  return;
}
```

Migrating models

While most use cases for models can now be replaced with plain joi schemas, there is still the concept of a "model" in Foxx in ArangoDB 3.0 although it is quite different from Foxx models in ArangoDB 2.x.

A model in Foxx now refers to a plain JavaScript object with an optional `schema` attribute and the optional methods `forClient` and `fromClient`. Models can be used instead of plain joi schemas to define request and response bodies but there are no model "instances" in ArangoDB 3.0.

Old:

```
'use strict';
const _ = require('underscore');
const joi = require('joi');
const Foxx = require('org/arangodb/foxx');
const MyModel = Foxx.Model.extend({
  schema: {
    name: joi.string().required(),
    size: joi.number().required()
  },
  forClient () {
    return _.omit(this.attributes, ['_key', '_id', '_rev']);
  }
});

// ...

ctrl.get(/* ... */)
.bodyParam('body', {type: MyModel});
```

New:

```
'use strict';
const _ = require('lodash');
const joi = require('joi');
const MyModel = {
  schema: joi.object({
    name: joi.string().required(),
    size: joi.number().required()
  }).required(),
  forClient (data) {
    return _.omit(data, ['_key', '_id', '_rev']);
  }
};

// ...

router.get(/* ... */)
.body(MyModel);
```

Triggers

When saving, updating, replacing or deleting models in ArangoDB 2.x using the repository methods the repository and model would fire events that could be subscribed to in order to perform side-effects.

Note that even in 2.x these events would not fire when using queries or manipulating documents in any other way than using the specific repository methods that operated on individual documents.

This behaviour is no longer available in ArangoDB 3.0 but can be emulated by using an `EventEmitter` directly if it is not possible to solve the problem differently:

Old:

```
'use strict';
const Foxx = require('org/arangodb/foxx');
const MyModel = Foxx.Model.extend({
  // ...
}, {
  afterRemove () {
    console.log(this.get('name'), 'was removed');
  }
});

// ...

const model = myRepo.firstExample({name: 'myName'});
myRepo.remove(model);
// -> "myName was removed successfully"
```

New:

```
'use strict';
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('afterRemove', function (doc) {
  console.log(doc.name, 'was removed');
});

// ...

const doc = myDocs.firstExample({name: 'myName'});
myDocs.remove(doc);
emitter.emit('afterRemove', doc);
// -> "myName was removed successfully"
```

Or simply:

```
'use strict';
function afterRemove(doc) {
  console.log(doc.name, 'was removed');
}

// ...

const doc = myDocs.firstExample({name: 'myName'});
myDocs.remove(doc);
afterRemove(doc);
// -> "myName was removed successfully"
```

Controllers vs routers

Foxx Controllers have been replaced with routers. This is more than a cosmetic change as there are significant differences in behaviour:

Controllers were automatically mounted when the file defining them was executed. Routers need to be explicitly mounted using the `module.context.use` method. Routers can also be exported, imported and even nested. This makes it easier to split up complex routing trees across multiple files.

Old:

```
'use strict';
const Foxx = require('org/arangodb/foxx');
const ctrl = new Foxx.Controller(applicationContext);

ctrl.get('/hello', function (req, res) {
  // ...
});
```

New:

```
'use strict';
const createRouter = require('org/arangodb/foxx/router');
const router = createRouter();
// If you are importing this file from your entry file ("main"):
module.exports = router;
// Otherwise: module.context.use(router);

router.get('/hello', function (req, res) {
  // ...
});
```

Some general changes in behaviour that might trip you up:

- When specifying path parameters with schemas Foxx will now ignore the route if the schema does not match (i.e. `/hello/foxx` will no longer match `/hello/:num` if `num` specifies a schema that doesn't match the value `"foxx"`). With controllers this could previously result in users seeing a 400 (bad request) error when they should instead be served a 404 (not found) response.
- When a request is made with an HTTP verb not supported by an endpoint, Foxx will now respond with a 405 (method not allowed) error with an appropriate `Allowed` header listing the supported HTTP verbs for that endpoint.
- Foxx will no longer parse your JSDoc comments to generate route documentation (use the `summary` and `description` methods of the endpoint instead).
- The `apiDocumentation` method now lives on the service context and behaves slightly differently.
- There is no router equivalent for the `activateAuthentication` and `activateSessions` methods. Instead you should use the session middleware (see the section on sessions below).
- There is no `del` alias for the `delete` method on routers. It has always been safe to use keywords as method names in Foxx, so the use of this alias was already discouraged before.
- The `allRoutes` proxy is no lot available on routers but can easily be replaced with middleware or child routers.

The request context

When defining a route on a controller the controller would return an object called *request context*. Routers return a similar object called *endpoint*. Routers also return endpoints when mounting child routers or middleware, as does the `use` method of the service context.

The main differences between the new endpoints and the objects returned by controllers in previous versions of ArangoDB are:

- `bodyParam` is now simply called `body`; it is no longer neccessary or possible to give the body a name and the request body will not show up in the request parameters. It's also possible to specify a MIME type
- `body`, `queryParam` and `pathParam` now take position arguments instead of an object. For specifics see the endpoint documentation.
- `notes` is now called `description` and takes a single string argument.
- `onlyIf` and `onlyIfAuthenticated` are no longer available; they can be emulated with middleware if necessary:

Old:

```
ctrl.get(/* ... */)
.onlyIf(function (req) {
  if (!req.user) {
    throw new Error('Not authenticated!');
  }
});
```

New:

```
router.use(function (req, res, next) {
  if (!req.arangoUser) {
    res.throw(403, 'Not authenticated!');
  }
  next();
});

router.get(/* ... */);
```

Error handling

The `errorResponse` method provided by controller request contexts has no equivalent in router endpoints. If you want to handle specific error types with specific status codes you need to catch them explicitly, either in the route or in a middleware:

Old:

```
ctrl.get('/puppies', function (req, res) {
  // Exception is thrown here
})
.errorResponse(TooManyPuppiesError, 400, 'Something went wrong!');
```

New:

```
ctrl.get('/puppies', function (req, res) {
  try {
    // Exception is thrown here
  } catch (e) {
    if (!(e instanceof TooManyPuppiesError)) {
      throw e;
    }
    res.throw(400, 'Something went wrong!');
  }
})
// The "error" method merely documents the meaning
// of the status code and has no other effect.
.error(400, 'Thrown if there are too many puppies.');
```

Note that errors created with `http-errors` are still handled by Foxx intelligently. In fact `res.throw` is just a helper method for creating and throwing these errors.

Before, after and around

The `before`, `after` and `around` methods can easily be replaced by middleware:

Old:

```
let start;
ctrl.before(function (req, res) {
  start = Date.now();
});
ctrl.after(function (req, res) {
  console.log('Request handled in ', (Date.now() - start), 'ms');
});
```

New:

```
router.use(function (req, res, next) {
  let start = Date.now();
  next();
  console.log('Request handled in ', (Date.now() - start), 'ms');
});
```

Note that unlike `around` middleware receives the `next` function as the *third* argument (the "opts" argument has no equivalent).

Request objects

The names of some attributes of the request object have been adjusted to more closely align with those of the corresponding methods on the endpoint objects and established conventions in other JavaScript frameworks:

- `req.urlParameters` is now called `req.pathParams`
- `req.parameters` is now called `req.queryParams`
- `req.params()` is now called `req.param()`
- `req.requestType` is now called `req.method`
- `req.compatibility` is now called `req.arangoVersion`
- `req.user` is now called `req.arangoUser`

Some attributes have been removed or changed:

- `req.cookies` has been removed entirely (use `req.cookie(name)`)
- `req.requestBody` has been removed entirely (see below)
- `req.suffix` is now a string rather than an array

Additionally the `req.server` and `req.client` attributes are no longer available. The information is now exposed in a way that can (optionally) transparently handle proxy forwarding headers:

- `req.hostname` defaults to `req.server.address`
- `req.port` defaults to `req.server.port`
- `req.remoteAddress` defaults to `client.address`
- `req.remotePort` defaults to `client.port`

Finally, the `req.cookie` method now takes the `signed` options directly.

Old:

```
const sid = req.cookie('sid', {
  signed: {
    secret: 'keyboardcat',
    algorithm: 'sha256'
  }
});
```

New:

```
const sid = req.cookie('sid', {
  secret: 'keyboardcat',
  algorithm: 'sha256'
});
```

Request bodies

The `req.body` is no longer a method and no longer automatically parses JSON request bodies unless a request body was defined. The `req.rawBody` now corresponds to the `req.rawBodyBuffer` of ArangoDB 2.x and is also no longer a method.

Old:

```
ctrl.post('/', function (req, res) {
  const data = req.body();
  // ...
});
```

New:

```
router.post('/', function (req, res) {
  const data = req.body;
  // ...
})
.body(['json']);
```

Or simply:

```
const joi = require('joi');
router.post('/', function (req, res) {
  const data = req.body;
  // ...
})
.body(joi.object().optional());
```

Multipart requests

The `req.requestParts` method has been removed entirely. If you need to accept multipart request bodies, you can simply define the request body using a multipart MIME type like `multipart/form-data`:

Old:

```
ctrl.post('/', function (req, res) {
  const parts = req.requestParts();
  // ...
});
```

New:

```
router.post('/', function (req, res) {
  const parts = req.body;
  // ...
})
.body(['multipart/form-data']);
```

Response objects

The response object has a lot of new methods in ArangoDB 3.0 but otherwise remains similar to the response object of previous versions:

The `res.send` method behaves very differently from how the method with the same name behaved in ArangoDB 2.x: the conversion now takes the response body definition of the route into account. There is a new method `res.write` that implements the old behaviour.

Note that consecutive calls to `res.write` will append to the response body rather than replacing it like `res.send` .

The `res.contentType` property is also no longer available. If you want to set the MIME type of the response body to an explicit value you should set the `content-type` header instead:

Old:

```
res.contentType = 'application/json';
res.body = JSON.stringify(results);
```

New:

```
res.set('content-type', 'application/json');
res.body = JSON.stringify(results);
```

Or simply:

```
// sets the content type to JSON
// if it has not already been set
res.json(results);
```

The `res.cookie` method now takes the `signed` options as part of the regular options object.

Old:

```
res.cookie('sid', 'abcdef', {
  ttl: 60 * 60,
  signed: {
    secret: 'keyboardcat',
    algorithm: 'sha256'
  }
});
```

New:

```
res.cookie('sid', 'abcdef', {
  ttl: 60 * 60,
  secret: 'keyboardcat',
  algorithm: 'sha256'
});
```

Dependency injection

There is no equivalent of the `addInjector` method available in ArangoDB 2.x controllers. Most use cases can be solved by simply using plain variables but if you need something more flexible you can also use middleware:

Old:

```
ctrl.addInjector('magicNumber', function () {
  return Math.random();
});

ctrl.get('/', function (req, res, injected) {
  res.json(injected.magicNumber);
});
```

New:

```
function magicMiddleware(name) {
  return {
    register () {
      let magic;
      return function (req, res, next) {
        if (!magic) {
          magic = Math.random();
        }
        req[name] = magic;
        next();
      };
    }
  };
}

router.use(magicMiddleware('magicNumber'));

router.get('/', function (req, res) {
  res.json(req.magicNumber);
});
```

Or simply:

```
const magicNumber = Math.random();

router.get('/', function (req, res) {
  res.json(magicNumber);
});
```

Sessions

The `ctrl.activateSessions` method and the related `util-sessions-local` Foxx service have been replaced with the Foxx sessions middleware. It is no longer possible to use the built-in session storage but you can simply pass in any document collection directly.

Old:

```
const localSessions = applicationContext.dependencies.localSessions;
const sessionStorage = localSessions.sessionStorage;
ctrl.activateSessions({
  sessionStorage: sessionStorage,
  cookie: {secret: 'keyboardcat'}
});

ctrl.destroySession('/logout', function (req, res) {
  res.json({message: 'Goodbye!'});
});
```

New:

```
const sessionMiddleware = require('@arangodb/foxx/sessions');
const cookieTransport = require('@arangodb/foxx/sessions/transports/cookie');
router.use(sessionMiddleware({
  storage: module.context.collection('sessions'),
  transport: cookieTransport('keyboardcat')
}));

router.post('/logout', function (req, res) {
  req.sessionStorage.clear(req.session);
  res.json({message: 'Goodbye!'});
});
```

Auth and OAuth2

The `util-simple-auth` and `util-oauth2` Foxx services have been replaced with the Foxx auth and Foxx OAuth2 modules. It is no longer necessary to install these services as dependencies in order to use the functionality.

Old:

```
'use strict';
const auth = applicationContext.dependencies.simpleAuth;

// ...

const valid = auth.verifyPassword(authData, password);
```

New:

```
'use strict';
const createAuth = require('@arangodb/foxx/auth');
const auth = createAuth(); // Use default configuration

// ...

const valid = auth.verifyPassword(authData, password);
```

Foxx queries

The `createQuery` method has been removed. It can be trivially replaced with plain JavaScript functions and direct calls to the `db._query` method:

Old:

```
'use strict';
const Foxx = require('org/arangodb/foxx');
const query = Foxx.createQuery({
    query: 'FOR u IN _users SORT u.user ASC RETURN u[@propName]',
    params: ['propName'],
    transform: function (results, uppercase) {
        return (
          uppercase
          ? results[0].toUpperCase()
          : results[0].toLowerCase()
        );
    }
});

query('user', true);
```

New:

```
'use strict';
const db = require('@arangodb').db;
const aql = require('@arangodb').aql;

function query(propName, uppercase) {
  const results = db._query(aql`
    FOR u IN _users
    SORT u.user ASC
    RETURN u[${propName}]
  `);
  return (
    uppercase
    ? results[0].toUpperCase()
    : results[0].toLowerCase()
  );
}

query('user', true);
```

Legacy compatibility mode for 2.8 services

ArangoDB 3 continues to support Foxx services written for ArangoDB 2.8 by running them in a special legacy compatibility mode that provides access to some of the modules and APIs no longer provided in 3.0 and beyond.

Note: Legacy compatibility mode is strictly intended as a temporary stop gap solution for supporting existing services while upgrading to ArangoDB 3.0 and should not be considered a permanent feature of ArangoDB or Foxx.

In order to mark an existing service as a legacy service, just make sure the following attribute is defined in the service manifest:

```
"engines": {
  "arangodb": "^2.8.0"
}
```

This semantic version range denotes that the service is known to work with ArangoDB 2.8.0 and supports all newer versions of ArangoDB up to but not including 3.0.0 (nor any development version of 3.0.0 and greater).

Any similar version range the does not include 3.0.0 or greater will have the same effect (e.g. `^2.5.0` will also trigger the legacy compatibility mode, as will `1.2.3` , but `>=2.8.0` will not as it indicates compatibility with *all* versions greater or equal 2.8.0, not just those within the 2.x version range).

Features supported in legacy compatibility mode

Legacy compatibility mode supports the old manifest format, specifically:

- `main` is ignored
- `controllers` will be mounted as in 2.8
- `exports` will be executed as in 2.8

Additionally the `isSystem` attribute will be ignored if present but does not result in a warning in legacy compatibility mode.

The Foxx console is available as the `console` pseudo-global variable (shadowing the global console object).

The service context is available as the `applicationContext` pseudo-global variable in the `controllers` , `exports` , `scripts` and `tests` as in 2.8. The following additional properties are available on the service context in legacy compatibility mode:

- `path()` is an alias for 3.x `fileName()` (using `path.join` to build file paths)
- `fileName()` behaves as in 2.x (using `fs.safeJoin` to build file paths)
- `foxxFileName()` is an alias for 2.x `fileName`
- `version` exposes the service manifest's `version` attribute
- `name` exposes the service manifest's `name` attribute
- `options` exposes the service's raw options

The following methods are removed on the service context in legacy compatibility mode:

- `use()` -- use `@arangodb/foxx/controller` instead
- `apiDocumentation()` -- use `controller.apiDocumentation()` instead
- `registerType()` -- not supported in legacy compatibility mode

The following modules that have been removed or replaced in 3.0.0 are available in legacy compatibility mode:

- `@arangodb/foxx/authentication`
- `@arangodb/foxx/console`
- `@arangodb/foxx/controller`
- `@arangodb/foxx/model`
- `@arangodb/foxx/query`
- `@arangodb/foxx/repository`
- `@arangodb/foxx/schema`
- `@arangodb/foxx/sessions`
- `@arangodb/foxx/template_middleware`

The `@arangodb/foxx` module also provides the same exports as in 2.8, namely:

- `Controller` from `@arangodb/foxx/controller`
- `createQuery` from `@arangodb/foxx/query`
- `Model` from `@arangodb/foxx/model`
- `Repository` from `@arangodb/foxx/repository`
- `toJSONSchema` from `@arangodb/foxx/schema`
- `getExports` and `requireApp` from `@arangodb/foxx/manager`
- `queues` from `@arangodb/foxx/queues`

Any feature not supported in 2.8 will also not work in legacy compatibility mode. When migrating from an older version of ArangoDB it is a good idea to migrate to ArangoDB 2.8 first for an easier upgrade path.

Additionally please note the differences laid out in the chapter *Migrating from pre-2.8* in the migration guide.

User management

Foxx does not provide any user management out of the box but it is very easy to roll your own solution:

The session middleware provides mechanisms for adding session logic to your service, using e.g. a collection or JSON Web Tokens to store the sessions between requests.

The auth module provides utilities for basic password verification and hashing.

The following example service demonstrates how user management can be implemented using these basic building blocks.

Setting up the collections

Let's say we want to store sessions and users in collections. We can use the setup script to make sure these collections are created before the service is mounted.

First add a setup script to your manifest if it isn't already defined:

```
"scripts": {
  "setup": "scripts/setup.js"
}
```

Then create the setup script with the following content:

```
'use strict';
const db = require('@arangodb').db;
const sessions = module.context.collectionName('sessions');
const users = module.context.collectionName('users');

if (!db._collection(sessions)) {
  db._createDocumentCollection(sessions);
}

if (!db._collection(users)) {
  db._createDocumentCollection(users);
}

db._collection(users).ensureIndex({
  type: 'hash',
  fields: ['username'],
  unique: true
});
```

Creating the router

The following main file demonstrates basic user management:

```
'use strict';
const joi = require('joi');
const createAuth = require('@arangodb/foxx/auth');
const createRouter = require('@arangodb/foxx/router');
const sessionsMiddleware = require('@arangodb/foxx/sessions');

const auth = createAuth();
const router = createRouter();
const users = module.context.collection('users');
const sessions = sessionsMiddleware({
  storage: module.context.collection('sessions'),
  transport: 'cookie'
});
module.context.use(sessions);
module.context.use(router);

router.get('/whoami', function (req, res) {
  try {
    const user = users.document(req.session.uid);
    res.send({username: user.username});
  } catch (e) {
    res.send({username: null});
  }
})
.description('Returns the currently active username.');

router.post('/login', function (req, res) {
  // This may return a user object or null
  const user = users.firstExample({
    username: req.body.username
  });
  const valid = auth.verify(
    // Pretend to validate even if no user was found
    user ? user.authData : {},
    req.body.password
  );
  if (!valid) res.throw('unauthorized');
  // Log the user in
  req.session.uid = user._key;
  req.sessionStorage.save(req.session);
  res.send({sucess: true});
})
.body(joi.object({
  username: joi.string().required(),
  password: joi.string().required()
}).required(), 'Credentials')
.description('Logs a registered user in.');

router.post('/logout', function (req, res) {
  if (req.session.uid) {
    req.session.uid = null;
    req.sessionStorage.save(req.session);
  }
  res.send({success: true});
})
.description('Logs the current user out.');

router.post('/signup', function (req, res) {
  const user = req.body;
  try {
    // Create an authentication hash
    user.authData = auth.create(user.password);
    delete user.password;
    const meta = users.save(user);
    Object.assign(user, meta);
  } catch (e) {
    // Failed to save the user
    // We'll assume the UniqueConstraint has been violated
    res.throw('bad request', 'Username already taken', e);
  }
  // Log the user in
  req.session.uid = user._key;
  req.sessionStorage.save(req.session);
  res.send({success: true});
})
.body(joi.object({
  username: joi.string().required(),
  password: joi.string().required()
}).required(), 'Credentials')
.description('Creates a new user and logs them in.');
```

Related modules

These are some of the modules outside of Foxx you will find useful when writing Foxx services.

Additionally there are modules providing some level of compatibility with Node.js as well as a number of bundled NPM modules (like lodash and joi). For more information on these modules see the JavaScript modules appendix.

The `@arangodb` module

This module provides access to various ArangoDB internals as well as three of the most important exports necessary to work with the database in Foxx:

The `db` object

`require('@arangodb').db`

The `db` object represents the current database and lets you access collections and run queries. For more information see the db object reference.

Examples

```
const db = require('@arangodb').db;

const thirteen = db._query('RETURN 5 + 8')[0];
```

The `aql` template string handler

`require('@arangodb').aql`

The `aql` function is a JavaScript template string handler. It can be used to write complex AQL queries as multi-line strings without having to worry about bindVars and the distinction between collections and regular parameters.

To use it just prefix a JavaScript template string (the ones with backticks instead of quotes) with its import name (e.g. `aql`) and pass in variables like you would with a regular template string. The string will automatically be converted into an object with `query` and `bindVars` attributes which you can pass directly to `db._query` to execute. If you pass in a collection it will be automatically recognized as a collection reference and handled accordingly.

To find out more about AQL see the AQL documentation.

Examples

```
const aql = require('@arangodb').aql;

const filterValue = 23;
const mydata = db._collection('mydata');
const result = db._query(aql`
  FOR d IN ${mydata}
  FILTER d.num > ${filterValue}
  RETURN d
`);
```

The `errors` object

`require('@arangodb').errors`

This object provides useful objects for each error code ArangoDB might use in `ArangoError` errors. This is helpful when trying to catch specific errors raised by ArangoDB, e.g. when trying to access a document that does not exist. Each object has a `code` property corresponding to the `errorNum` found on `ArangoError` errors.

For a complete list of the error names and codes you may encounter see the appendix on error codes.

Examples

```
const errors = require('@arangodb').errors;

try {
  mydata.document('does-not-exist');
} catch (e) {
  if (e.isArangoError && e.errorNum === errors.ERROR_ARANGO_DOCUMENT_NOT_FOUND.code) {
    res.throw(404, 'Document does not exist');
  }
  res.throw(500, 'Something went wrong', e);
}
```

The `@arangodb/request` module

`require('@arangodb/request')`

This module provides a function for making HTTP requests to external services. Note that while this allows communicating with third-party services it may affect database performance by blocking Foxx requests as ArangoDB waits for the remote service to respond. If you routinely make requests to slow external services and are not directly interested in the response it is probably a better idea to delegate the actual request/response cycle to a gateway service running outside ArangoDB.

You can find a full description of this module in the request module appendix.

The `@arangodb/general-graph` module

`require('@arangodb/general-graph')`

This module provides access to ArangoDB graph definitions and various low-level graph operations in JavaScript. For more complex queries it is probably better to use AQL but this module can be useful in your setup and teardown scripts to create and destroy graph definitions.

For more information see the chapter on the general graph module.

Authentication

`const createAuth = require('@arangodb/foxx/auth');`

Authenticators allow implementing basic password mechanism using simple built-in hashing functions.

For a full example of sessions with authentication and registration see the example in the chapter on User Management.

Creating an authenticator

`createAuth([options]): Authenticator`

Creates an authenticator.

Arguments

- **options**: `Object` (optional)

  An object with the following properties:

  - **method**: `string` (Default: `"sha256"`)

    The hashing algorithm to use to create password hashes. The authenticator will be able to verify passwords against hashes using any supported hashing algorithm. This only affects new hashes created by the authenticator.

    Supported values:

    - `"md5"`
    - `"sha1"`
    - `"sha224"`
    - `"sha256"`
    - `"sha384"`
    - `"sha512"`
  - **saltLength**: `number` (Default: `16`)

    Length of the salts that will be generated for password hashes.

Returns an authenticator.

Creating authentication data objects

`auth.create(password): AuthData`

Creates an authentication data object for the given password with the following properties:

- **method**: `string`

  The method used to generate the hash.

- **salt**: `string`

  A random salt used to generate this hash.

- **hash**: `string`

  The hash string itself.

Arguments

- **password**: `string`

  A password to hash.

Returns the authentication data object.

Validating passwords against authentication data objects

`auth.verify([hash, [password]]): boolean`

Verifies the given password against the given hash using a constant time string comparison.

Arguments

- **hash**: `AuthData` (optional)

  A authentication data object generated with the *create* method.

- **password**: `string` (optional)

  A password to verify against the hash.

Returns `true` if the hash matches the given password. Returns `false` otherwise.

OAuth 2.0

`const createOAuth2Client = require('@arangodb/foxx/oauth2');`

The OAuth2 module provides abstractions over OAuth2 providers like Facebook, GitHub and Google.

Examples

The following extends the user management example:

```
const crypto = require('@arangodb/crypto');
const router = createRouter();
const oauth2 = createOAuth2Client({
  // We'll use Facebook for this example
  authEndpoint: 'https://www.facebook.com/dialog/oauth',
  tokenEndpoint: 'https://graph.facebook.com/oauth/access_token',
  activeUserEndpoint: 'https://graph.facebook.com/v2.0/me',
  clientId: 'keyboardcat',
  clientSecret: 'keyboardcat'
});

module.context.use('/oauth2', router);

// See the user management example for setting up the
// sessions and users objects used in this example
router.use(sessions);

router.post('/auth', function (req, res) {
  const csrfToken = crypto.genRandomAlphaNumbers(32);
  const url = req.reverse('oauth2_callback', {csrfToken});
  const redirect_uri = req.makeAbsolute(url);
  // Set CSRF cookie for five minutes
  res.cookie('oauth2_csrf_token', csrfToken, {ttl: 60 * 5});
  // Redirect to the provider's authorization URL
  res.redirect(303, oauth2.getAuthUrl(url));
});

router.get('/auth', function (req, res) {
  // Some providers pass errors as query parameter
  if (req.queryParams.error) {
    res.throw(500, `Provider error: ${req.queryParams.error}`)
  }
  // Make sure CSRF cookie matches the URL
  const expectedToken = req.cookie('oauth2_csrf_token');
  if (!expectedToken || req.queryParams.csrfToken !== expectedToken) {
    res.throw(400, 'CSRF mismatch.');
  }
  // Make sure the URL contains a grant token
  if (!req.queryParams.code) {
    res.throw(400, 'Provider did not pass grant token.');
  }
  // Reconstruct the redirect_uri used for the grant token
  const url = req.reverse('oauth2_callback');
  const redirect_uri = req.makeAbsolute(url);
  // Fetch an access token from the provider
  const authData = oauth2.exchangeGrantToken(
    req.queryParams.code,
    redirect_uri
  );
  const facebookToken = authData.access_token;
  // Fetch the active user's profile info
  const profile = oauth2.fetchActiveUser(facebookToken);
  const facebookId = profile.id;
  // Try to find an existing user with the user ID
  // (this requires the users collection)
  let user = users.firstExample({facebookId});
  if (user) {
    // Update the access_token if it has changed
    if (user.facebookToken !== facebookToken) {
      users.update(user, {facebookToken});
    }
  } else {
    // Create a new user document
    user = {
      username: `fb:${facebookId}`,
      facebookId,
      access_token
    }
    const meta = users.save(user);
    Object.assign(user, meta);
  }
  // Log the user in (this requires the session middleware)
  req.session.uid = user._key;
  req.session.access_token = authData.access_token;
  req.sessionStorage.save(req.session);
  // Redirect to the default route
  res.redirect(303, req.makeAbsolute('/'));
}, 'oauth2_callback')
.queryParam('error', joi.string().optional())
.queryParam('csrfToken', joi.string().optional())
.queryParam('code', joi.string().optional());
```

Creating an OAuth2 client

`createOAuth2Client(options): OAuth2Client`

Creates an OAuth2 client.

Arguments

- **options**: `Object`

  An object with the following properties:

  - **authEndpoint**: `string`

    The fully-qualified URL of the provider's authorization endpoint.

  - **tokenEndpoint**: `string`

    The fully-qualified URL of the provider's token endpoint.

  - **refreshEndpoint**: `string` (optional)

    The fully-qualified URL of the provider's refresh token endpoint.

  - **activeUserEndpoint**: `string` (optional)

    The fully-qualified URL of the provider's endpoint for fetching details about the current user.

  - **clientId**: `string`

    The application's *Client ID* (or *App ID*) for the provider.

  - **clientSecret**: `string`

    The application's *Client Secret* (or *App Secret*) for the provider.

Returns an OAuth2 client for the given provider.

Setting up OAuth2 for Facebook

If you want to use Facebook as the OAuth2 provider, use the following options:

- *authEndpoint*: `https://www.facebook.com/dialog/oauth`
- *tokenEndpoint*: `https://graph.facebook.com/oauth/access_token`
- *activeUserEndpoint*: `https://graph.facebook.com/v2.0/me`

You also need to obtain a client ID and client secret from Facebook:

1. Create a regular account at Facebook or use an existing account you own.
2. Visit the Facebook Developers page.

3. Click on *Apps* in the menu, then select *Register as a Developer* (the only option) and follow the instructions provided. You may need to verify your account by phone.
4. Click on *Apps* in the menu, then select *Create a New App* and follow the instructions provided.
5. Open the app dashboard, then note down the *App ID* and *App Secret*. The secret may be hidden by default.
6. Click on *Settings*, then *Advanced* and enter one or more *Valid OAuth redirect URIs*. At least one of them must match your *redirect_uri* later. Don't forget to save your changes.
7. Set the option *clientId* to the *App ID* and the option *clientSecret* to the *App Secret*.

Setting up OAuth2 for GitHub

If you want to use GitHub as the OAuth2 provider, use the following options:

- *authEndpoint*: `https://github.com/login/oauth/authorize?scope=user`
- *tokenEndpoint*: `https://github.com/login/oauth/access_token`
- *activeUserEndpoint*: `https://api.github.com/user`

You also need to obtain a client ID and client secret from GitHub:

1. Create a regular account at GitHub or use an existing account you own.
2. Go to Account Settings > Applications > Register new application.
3. Provide an *authorization callback URL*. This must match your *redirect_uri* later.
4. Fill in the other required details and follow the instructions provided.
5. Open the application page, then note down the *Client ID* and *Client Secret*.
6. Set the option *clientId* to the *Client ID* and the option *clientSecret* to the *Client Secret*.

Setting up OAuth2 for Google

If you want to use Google as the OAuth2 provider, use the following options:

- *authEndpoint*: `https://accounts.google.com/o/oauth2/auth?access_type=offline&scope=profile`
- *tokenEndpoint*: `https://accounts.google.com/o/oauth2/token`
- *activeUserEndpoint*: `https://www.googleapis.com/plus/v1/people/me`

You also need to obtain a client ID and client secret from Google:

1. Create a regular account at Google or use an existing account you own.
2. Visit the Google Developers Console.
3. Click on *Create Project*, then follow the instructions provided.
4. When your project is ready, open the project dashboard, then click on *Enable an API*.
5. Enable the *Google+ API* to allow your app to distinguish between different users.
6. Open the *Credentials* page and click *Create new Client ID*, then follow the instructions provided. At least one *Authorized Redirect URI* must match your *redirect_uri* later. At least one *Authorized JavaScript Origin* must match your app's fully-qualified domain.
7. When the Client ID is ready, note down the *Client ID* and *Client secret*.
8. Set the option *clientId* to the *Client ID* and the option *clientSecret* to the *Client secret*.

Get the authorization URL

`oauth2.getAuthUrl(redirect_uri, args): string`

Generates the authorization URL for the authorization endpoint.

Arguments

- **redirect_uri**: `string`

  The fully-qualified URL of your application's OAuth2 callback.

- **args**: (optional)

  An object with any of the following properties:

  - **response_type**: `string` (Default: `"code"`)

See RFC 6749.

Returns a fully-qualified URL for the authorization endpoint of the provider by appending the client ID and any additional arguments from *args* to the *authEndpoint*.

Exchange a grant code for an access token

`oauth2.exchangeGrantToken(code, redirect_uri)`

Exchanges a grant code for an access token.

Performs a *POST* response to the *tokenEndpoint*.

Throws an exception if the remote server responds with an empty response body.

Arguments

- **code**: `string`

  A grant code returned by the provider's authorization endpoint.

- **redirect_uri**: `string`

  The original callback URL with which the code was requested.

- **args**: `Object` (optional)

  An object with any of the following properties:

  - **grant_type**: `string` (Default: `"authorization_code"`)

    See RFC 6749.

Returns the parsed response object.

Fetch the active user

`oauth2.fetchActiveUser(access_token): Object`

Fetches details of the active user.

Performs a *GET* response to the *activeUserEndpoint*.

Throws an exception if the remote server responds with an empty response body.

Also throws an exception if the *activeUserEndpoint* is not configured.

Arguments

- **access_token**: `string`

  An OAuth2 access token as returned by *exchangeGrantToken*.

Returns the parsed response object.

Examples

```
const authData = oauth2.exchangeGrantToken(code, redirect_uri);
const userData = oauth2.fetchActiveUser(authData.access_token);
```

Transactions

Starting with version 1.3, ArangoDB provides support for user-definable transactions.

Transactions in ArangoDB are atomic, consistent, isolated, and durable (*ACID*).

These *ACID* properties provide the following guarantees:

- The *atomicity* principle makes transactions either complete in their entirety or have no effect at all.
- The *consistency* principle ensures that no constraints or other invariants will be violated during or after any transaction.
- The *isolation* property will hide the modifications of a transaction from other transactions until the transaction commits.
- Finally, the *durability* proposition makes sure that operations from transactions that have committed will be made persistent. The amount of transaction durability is configurable in ArangoDB, as is the durability on collection level.

Transaction invocation

ArangoDB transactions are different from transactions in SQL.

In SQL, transactions are started with explicit *BEGIN* or *START TRANSACTION* command. Following any series of data retrieval or modification operations, an SQL transaction is finished with a *COMMIT* command, or rolled back with a *ROLLBACK* command. There may be client/server communication between the start and the commit/rollback of an SQL transaction.

In ArangoDB, a transaction is always a server-side operation, and is executed on the server in one go, without any client interaction. All operations to be executed inside a transaction need to be known by the server when the transaction is started.

There are no individual *BEGIN*, *COMMIT* or *ROLLBACK* transaction commands in ArangoDB. Instead, a transaction in ArangoDB is started by providing a description of the transaction to the *db._executeTransaction* JavaScript function:

```
db._executeTransaction(description);
```

This function will then automatically start a transaction, execute all required data retrieval and/or modification operations, and at the end automatically commit the transaction. If an error occurs during transaction execution, the transaction is automatically aborted, and all changes are rolled back.

Execute transaction

executes a transaction `db._executeTransaction(object)`

Executes a server-side transaction, as specified by *object*.

object must have the following attributes:

- *collections*: a sub-object that defines which collections will be used in the transaction. *collections* can have these attributes:
  - *read*: a single collection or a list of collections that will be used in the transaction in read-only mode
  - *write*: a single collection or a list of collections that will be used in the transaction in write or read mode.
- *action*: a Javascript function or a string with Javascript code containing all the instructions to be executed inside the transaction. If the code runs through successfully, the transaction will be committed at the end. If the code throws an exception, the transaction will be rolled back and all database operations will be rolled back.

Additionally, *object* can have the following optional attributes:

- *waitForSync*: boolean flag indicating whether the transaction is forced to be synchronous.
- *lockTimeout*: a numeric value that can be used to set a timeout for waiting on collection locks. If not specified, a default value will be used. Setting *lockTimeout* to *0* will make ArangoDB not time out waiting for a lock.
- *params*: optional arguments passed to the function specified in *action*.

Declaration of collections

All collections which are to participate in a transaction need to be declared beforehand. This is a necessity to ensure proper locking and isolation.

Collections can be used in a transaction in write mode or in read-only mode.

If any data modification operations are to be executed, the collection must be declared for use in write mode. The write mode allows modifying and reading data from the collection during the transaction (i.e. the write mode includes the read mode).

Contrary, using a collection in read-only mode will only allow performing read operations on a collection. Any attempt to write into a collection used in read-only mode will make the transaction fail.

Collections for a transaction are declared by providing them in the *collections* attribute of the object passed to the *_executeTransaction* function. The *collections* attribute has the sub-attributes *read* and *write*:

```
db._executeTransaction({
  collections: {
    write: [ "users", "logins" ],
    read: [ "recommendations" ]
  }
});
```

read and *write* are optional attributes, and only need to be specified if the operations inside the transactions demand for it.

The contents of *read* or *write* can each be lists arrays collection names or a single collection name (as a string):

```
db._executeTransaction({
  collections: {
    write: "users",
    read: "recommendations"
  }
});
```

Note: It is currently optional to specify collections for read-only access. Even without specifying them, it is still possible to read from such collections from within a transaction, but with relaxed isolation. Please refer to Transactions Locking for more details.

In order to make a transaction fail when a non-declared collection is used inside for reading, the optional *allowImplicit* sub-attribute of *collections* can be set to *false*:

```
db._executeTransaction({
  collections: {
    read: "recommendations",
    allowImplicit: false  /* this disallows read access to other collections
                             than specified */
  },
  action: function () {
    var db = require("@arangodb").db;
    return db.foobar.toArray(); /* will fail because db.foobar must not be accessed
                                   for reading inside this transaction */
  }
});
```

The default value for *allowImplicit* is *true*. Write-accessing collections that have not been declared in the *collections* array is never possible, regardless of the value of *allowImplicit*.

Declaration of data modification and retrieval operations

All data modification and retrieval operations that are to be executed inside the transaction need to be specified in a Javascript function, using the *action* attribute:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: function () {
    // all operations go here
  }
});
```

Any valid Javascript code is allowed inside *action* but the code may only access the collections declared in *collections*. *action* may be a Javascript function as shown above, or a string representation of a Javascript function:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: "function () { doSomething(); }"
});
```

Please note that any operations specified in *action* will be executed on the server, in a separate scope. Variables will be bound late. Accessing any JavaScript variables defined on the client-side or in some other server context from inside a transaction may not work. Instead, any variables used inside *action* should be defined inside *action* itself:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: function () {
    var db = require(...).db;
    db.users.save({ ... });
  }
});
```

When the code inside the *action* attribute is executed, the transaction is already started and all required locks have been acquired. When the code inside the *action* attribute finishes, the transaction will automatically commit. There is no explicit commit command.

To make a transaction abort and roll back all changes, an exception needs to be thrown and not caught inside the transaction:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: function () {
    var db = require("@arangodb").db;
    db.users.save({ _key: "hello" });
    // will abort and roll back the transaction
    throw "doh!";
  }
});
```

There is no explicit abort or roll back command.

As mentioned earlier, a transaction will commit automatically when the end of the *action* function is reached and no exception has been thrown. In this case, the user can return any legal JavaScript value from the function:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: function () {
    var db = require("@arangodb").db;
    db.users.save({ _key: "hello" });
    // will commit the transaction and return the value "hello"
    return "hello";
  }
});
```

Examples

The first example will write 3 documents into a collection named *c1*. The *c1* collection needs to be declared in the *write* attribute of the *collections* attribute passed to the *executeTransaction* function.

The *action* attribute contains the actual transaction code to be executed. This code contains all data modification operations (3 in this example).

```
// setup
db._create("c1");

db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("@arangodb").db;
    db.c1.save({ _key: "key1" });
    db.c1.save({ _key: "key2" });
    db.c1.save({ _key: "key3" });
  }
});
    db.c1.count(); // 3
```

Aborting the transaction by throwing an exception in the *action* function will revert all changes, so as if the transaction never happened:

```
// setup
db._create("c1");

db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("@arangodb").db;
    db.c1.save({ _key: "key1" });
    db.c1.count(); // 1
    db.c1.save({ _key: "key2" });
    db.c1.count(); // 2
    throw "doh!";
  }
});

db.c1.count(); // 0
```

The automatic rollback is also executed when an internal exception is thrown at some point during transaction execution:

```
// setup
db._create("c1");

db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("@arangodb").db;
    db.c1.save({ _key: "key1" });
    // will throw duplicate a key error, not explicitly requested by the user
    db.c1.save({ _key: "key1" });
    // we'll never get here...
  }
});

db.c1.count(); // 0
```

As required by the *consistency* principle, aborting or rolling back a transaction will also restore secondary indexes to the state at transaction start.

Cross-collection transactions

There's also the possibility to run a transaction across multiple collections. In this case, multiple collections need to be declared in the *collections* attribute, e.g.:

```
// setup
db._create("c1");
db._create("c2");

db._executeTransaction({
  collections: {
    write: [ "c1", "c2" ]
  },
  action: function () {
    var db = require("@arangodb").db;
    db.c1.save({ _key: "key1" });
    db.c2.save({ _key: "key2" });
  }
});

db.c1.count(); // 1
db.c2.count(); // 1
```

Again, throwing an exception from inside the *action* function will make the transaction abort and roll back all changes in all collections:

```
// setup
db._create("c1");
db._create("c2");

db._executeTransaction({
  collections: {
    write: [ "c1", "c2" ]
  },
  action: function () {
    var db = require("@arangodb").db;
    for (var i = 0; i < 100; ++i) {
      db.c1.save({ _key: "key" + i });
      db.c2.save({ _key: "key" + i });
    }
    db.c1.count(); // 100
    db.c2.count(); // 100
    // abort
    throw "doh!"
  }
});

db.c1.count(); // 0
db.c2.count(); // 0
```

Passing parameters to transactions

Arbitrary parameters can be passed to transactions by setting the *params* attribute when declaring the transaction. This feature is handy to re-use the same transaction code for multiple calls but with different parameters.

A basic example:

```
db._executeTransaction({
  collections: { },
  action: function (params) {
    return params[1];
  },
  params: [ 1, 2, 3 ]
});
```

The above example will return *2*.

Some example that uses collections:

```
db._executeTransaction({
  collections: {
    write: "users",
    read: [ "c1", "c2" ]
  },
  action: function (params) {
    var db = require('@arangodb').db;
    var doc = db.c1.document(params['c1Key']);
    db.users.save(doc);
    doc = db.c2.document(params['c2Key']);
    db.users.save(doc);
  },
  params: {
    c1Key: "foo",
    c2Key: "bar"
  }
});
```

Locking and Isolation

All collections specified in the *collections* attribute are locked in the requested mode (read or write) at transaction start. Locking of multiple collections is performed in alphabetical order. When a transaction commits or rolls back, all locks are released in reverse order. The locking order is deterministic to avoid deadlocks.

While locks are held, modifications by other transactions to the collections participating in the transaction are prevented. A transaction will thus see a consistent view of the participating collections' data.

Additionally, a transaction will not be interrupted or interleaved with any other ongoing operations on the same collection. This means each transaction will run in isolation. A transaction should never see uncommitted or rolled back modifications by other transactions. Additionally, reads inside a transaction are repeatable.

Note that the above is true only for all collections that are declared in the *collections* attribute of the transaction.

Lazily adding collections

There might be situations when declaring all collections a priori is not possible, for example, because further collections are determined by a dynamic AQL query inside the transaction, for example a query using AQL graph traversal.

In this case, it would be impossible to know beforehand which collection to lock, and thus it is legal to not declare collections that will be accessed in the transaction in read-only mode. Accessing a non-declared collection in read-only mode during a transaction will add the collection to the transaction lazily, and fetch data from the collection as usual. However, as the collection is added lazily, there is no isolation from other concurrent operations or transactions. Reads from such collections are potentially non-repeatable.

Examples:

```
db._executeTransaction({
  collections: {
    read: "users"
  },
  action: function () {
    const db = require("@arangodb").db;
    /* Execute an AQL query that traverses a graph starting at a "users" vertex.
       It is yet unknown into which other collections the query might traverse */
    db._createStatement({
      query: `FOR v IN ANY "users/1234" connections RETURN v`
    }).execute().toArray().forEach(function (d) {
      /* ... */
    });
  }
});
```

This automatic lazy addition of collections to a transaction also introduces the possibility of deadlocks. Deadlocks may occur if there are concurrent transactions that try to acquire locks on the same collections lazily.

In order to make a transaction fail when a non-declared collection is used inside a transaction for reading, the optional *allowImplicit* sub-attribute of *collections* can be set to *false*:

```
db._executeTransaction({
  collections: {
    read: "users",
    allowImplicit: false
  },
  action: function () {
    /* The below query will now fail because the collection "connections" has not
       been specified in the list of collections used by the transaction */
    const db = require("@arangodb").db;
    db._createStatement({
      query: `FOR v IN ANY "users/1234" connections RETURN v`
    }).execute().toArray().forEach(function (d) {
      /* ... */
    });
  }
});
```

The default value for *allowImplicit* is *true*. Write-accessing collections that have not been declared in the *collections* array is never possible, regardless of the value of *allowImplicit*.

If *users/1234* has an edge in *connections*, linking it to another document in the *users* collection, then the following explicit declaration will work:

```
db._executeTransaction({
  collections: {
    read: ["users", "connections"],
    allowImplicit: false
  },
  /* ... */
```

If the edge points to a document in another collection however, then the query will fail, unless that other collection is added to the declaration as well.

Note that if a document handle is used as starting point for a traversal, e.g. `FOR v IN ANY "users/not_linked" ...` or `FOR v IN ANY {_id: "users/not_linked"} ...`, then no error is raised in the case of the start vertex not having any edges to follow, with `allowImplicit: false` and *users* not being declared for read access. AQL only sees a string and does not consider it a read access, unless there are edges connected to it. `FOR v IN ANY DOCUMENT("users/not_linked") ...` will fail even without edges, as it is always considered to be a read access to the *users* collection.

Deadlocks and Deadlock detection

A deadlock is a situation in which two or more concurrent operations (user transactions or AQL queries) try to access the same resources (collections, documents) and need to wait for the others to finish, but none of them can make any progress.

A good example for a deadlock is two concurrently executing transactions T1 and T2 that try to access the same collections but that need to wait for each other. In this example, transaction T1 will write to collection `c1`, but will also read documents from collection `c2` without announcing it:

```
db._executeTransaction({
  collections: {
    write: "c1"
  },
  action: function () {
    const db = require("@arangodb").db;

    /* write into c1 (announced) */
    db.c1.insert({ foo: "bar" });

    /* some operation here that takes long to execute... */

    /* read from c2 (unannounced) */
    db.c2.toArray();
  }
});
```

Transaction T2 announces to write into collection `c2`, but will also read documents from collection `c1` without announcing it:

```
db._executeTransaction({
  collections: {
    write: "c2"
  },
  action: function () {
    var db = require("@arangodb").db;

    /* write into c2 (announced) */
    db.c2.insert({ bar: "baz" });

    /* some operation here that takes long to execute... */

    /* read from c1 (unannounced) */
    db.c1.toArray();
  }
});
```

In the above example, a deadlock will occur if transaction T1 and T2 have both acquired their write locks (T1 for collection `c1` and T2 for collection `c2`) and are then trying to read from the other other (T1 will read from `c2` , T2 will read from `c1`). T1 will then try to acquire the read lock on collection `c2` , which is prevented by transaction T2. T2 however will wait for the read lock on collection `c1` , which is prevented by transaction T1.

In case of such deadlock, there would be no progress for any of the involved transactions, and none of the involved transactions could ever complete. This is completely undesirable, so the automatic deadlock detection mechanism in ArangoDB will automatically abort one of the transactions involved in such deadlock. Aborting means that all changes done by the transaction will be rolled back and error 29 (`deadlock detected`) will be thrown.

Client code (AQL queries, user transactions) that accesses more than one collection should be aware of the potential of deadlocks and should handle the error 29 (`deadlock detected`) properly, either by passing the exception to the caller or retrying the operation.

To avoid both deadlocks and non-repeatable reads, all collections used in a transaction should be specified in the `collections` attribute when known in advance. In case this is not possible because collections are added dynamically inside the transaction, deadlocks may occur and the deadlock detection may kick in and abort the transaction.

Durability

Transactions are executed in main memory first until there is either a rollback or a commit. On rollback, no data will be written to disk, but the operations from the transaction will be reversed in memory.

On commit, all modifications done in the transaction will be written to the collection datafiles. These writes will be synchronized to disk if any of the modified collections has the *waitForSync* property set to *true*, or if any individual operation in the transaction was executed with the *waitForSync* attribute. Additionally, transactions that modify data in more than one collection are automatically synchronized to disk. This synchronization is done to not only ensure durability, but to also ensure consistency in case of a server crash.

That means if you only modify data in a single collection, and that collection has its *waitForSync* property set to *false*, the whole transaction will not be synchronized to disk instantly, but with a small delay.

There is thus the potential risk of losing data between the commit of the transaction and the actual (delayed) disk synchronization. This is the same as writing into collections that have the *waitForSync* property set to *false* outside of a transaction. In case of a crash with *waitForSync* set to false, the operations performed in the transaction will either be visible completely or not at all, depending on whether the delayed synchronization had kicked in or not.

To ensure durability of transactions on a collection that have the *waitForSync* property set to *false*, you can set the *waitForSync* attribute of the object that is passed to *executeTransaction*. This will force a synchronization of the transaction to disk even for collections that have *waitForSync* set to *false*:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  waitForSync: true,
  action: function () { ... }
});
```

An alternative is to perform an operation with an explicit *sync* request in a transaction, e.g.

```
db.users.save({ _key: "1234" }, true);
```

In this case, the *true* value will make the whole transaction be synchronized to disk at the commit.

In any case, ArangoDB will give users the choice of whether or not they want full durability for single collection transactions. Using the delayed synchronization (i.e. *waitForSync* with a value of *false*) will potentially increase throughput and performance of transactions, but will introduce the risk of losing the last committed transactions in the case of a crash.

In contrast, transactions that modify data in more than one collection are automatically synchronized to disk. This comes at the cost of several disk sync. For a multi-collection transaction, the call to the *_executeTransaction* function will only return after the data of all modified collections has been synchronized to disk and the transaction has been made fully durable. This not only reduces the risk of losing data in case of a crash but also ensures consistency after a restart.

In case of a server crash, any multi-collection transactions that were not yet committed or in preparation to be committed will be rolled back on server restart.

For multi-collection transactions, there will be at least one disk sync operation per modified collection. Multi-collection transactions thus have a potentially higher cost than single collection transactions. There is no configuration to turn off disk synchronization for multi-collection transactions in ArangoDB. The disk sync speed of the system will thus be the most important factor for the performance of multi-collection transactions.

Limitations

In General

Transactions in ArangoDB have been designed with particular use cases in mind. They will be mainly useful for short and small data retrieval and/or modification operations.

The implementation is not optimized for very long-running or very voluminous operations, and may not be usable for these cases.

One limitation is that a transaction operation information must fit into main memory. The transaction information consists of record pointers, revision numbers and rollback information. The actual data modification operations of a transaction are written to the write-ahead log and do not need to fit entirely into main memory.

Ongoing transactions will also prevent the write-ahead logs from being fully garbage-collected. Information in the write-ahead log files cannot be written to collection data files or be discarded while transactions are ongoing.

To ensure progress of the write-ahead log garbage collection, transactions should be kept as small as possible, and big transactions should be split into multiple smaller transactions.

Transactions in ArangoDB cannot be nested, i.e. a transaction must not start another transaction. If an attempt is made to call a transaction from inside a running transaction, the server will throw error *1651 (nested transactions detected)*.

It is also disallowed to execute user transaction on some of ArangoDB's own system collections. This shouldn't be a problem for regular usage as system collections will not contain user data and there is no need to access them from within a user transaction.

Some operations are not allowed inside transactions in general:

- creation and deletion of collections (`db._create()` , `db._drop()` , `db._rename()`)
- creation and deletion of indexes (`db.ensure...Index()` , `db.dropIndex()`)

If an attempt is made to carry out any of these operations during a transaction, ArangoDB will abort the transaction with error code *1653 (disallowed operation inside transaction)*.

Finally, all collections that may be modified during a transaction must be declared beforehand, i.e. using the *collections* attribute of the object passed to the *_executeTransaction* function. If any attempt is made to carry out a data modification operation on a collection that was not declared in the *collections* attribute, the transaction will be aborted and ArangoDB will throw error *1652 unregistered collection used in transaction*. It is legal to not declare read-only collections, but this should be avoided if possible to reduce the probability of deadlocks and non-repeatable reads.

Please refer to Locking and Isolation for more details.

In Clusters

Using a single instance of ArangoDB, multi-document / multi-collection queries are guaranteed to be fully ACID. This is more than many other NoSQL database systems support. In cluster mode, single-document operations are also fully ACID. Multi-document / multi-collection queries in a cluster are not ACID, which is equally the case with competing database systems. Transactions in a cluster will be supported in a future version of ArangoDB and make these operations fully ACID as well. Note that for non-sharded collections in a cluster, the transactional properties of a single server apply (fully ACID).

Deployment

In this chapter we describe various possibilities to deploy ArangoDB. In particular for the cluster mode, there are different ways and we want to highlight their advantages and disadvantages. We even document in detail, how to set up a cluster by simply starting various ArangoDB processes on different machines, either directly or using Docker containers.

Single instance deployment

The latest official builds of ArangoDB for all supported operating systems may be obtained from https://www.arangodb.com/download/.

Linux remarks

Besides the official images which are provided for the most popular linux distributions there are also a variety of unofficial images provided by the community. We are tracking most of the community contributions (including new or updated images) in our newsletter:

https://www.arangodb.com/category/newsletter/

Windows remarks

Please note that ArangoDB will only work on 64bit.

Docker

The simplest way to deploy ArangoDB is using Docker. To get a general understanding of Docker have a look at their excellent documentation.

Authentication

To start the official Docker container you will have to decide on an authentication method. Otherwise the container won't start.

Provide one of the arguments to Docker as an environment variable.

There are three options:

1. ARANGO_NO_AUTH=1

   Disable authentication completely. Useful for local testing or for operating in a trusted network (without a public interface).

2. ARANGO_ROOT_PASSWORD=password

   Start ArangoDB with the given password for root

3. ARANGO_RANDOM_ROOT_PASSWORD=1

   Let ArangoDB generate a random root password

To get going quickly:

```
docker run -e ARANGO_RANDOM_ROOT_PASSWORD=1 arangodb/arangodb
```

For an in depth guide about Docker and ArangoDB please check the official documentation: https://hub.docker.com/r/arangodb/arangodb/ . Note that we are using the image `arangodb/arangodb` here which is always the most current one. There is also the "official" one called `arangodb` whose documentation is here: https://hub.docker.com/_/arangodb/

Distributed deployment using Apache Mesos

ArangoDB has a sophisticated and yet easy way to use cluster mode. To leverage the full cluster feature set (monitoring, scaling, automatic failover and automatic replacement of failed nodes) you have to run ArangoDB on some kind of cluster management system. Currently ArangoDB relies on Apache Mesos in that matter. Mesos is a cluster operating system which powers some of the worlds biggest datacenters running several thousands of nodes.

DC/OS

DC/OS is the recommended way to install a cluster as it eases much of the process to install a Mesos cluster. You can deploy it very quickly on a variety of cloud hosters or setup your own DC/OS locally. DC/OS is a set of tools built on top of Apache Mesos. Apache Mesos is a so called "Distributed Cluster Operation System" and the core of DC/OS. Apache Mesos has the concept of so called persistent volumes which make it perfectly suitable for a database.

Installing

First prepare a DC/OS cluster by going to https://dcos.io and following the instructions there.

DC/OS comes with its own package management. Packages can be installed from the so called "Universe". As an official DC/OS partner ArangoDB can be installed from there straight away.

1. Installing via DC/OS UI
   i. Open your browser and go to the DC/OS admin interface
   ii. Open the "Universe" tab
   iii. Locate arangodb and hit "Install Package"
   iv. Press "Install Package"
2. Installing via the DC/OS command line
   i. Install the dcos cli
   ii. Open a terminal and issue `dcos install arangodb`

Both options are essentially doing the same in the background. Both are starting ArangoDB with its default options set.

To review the default options using the web interface simply click "Advanced Installation" in the web interface. There you will find a list of options including some explanation.

To review the default options using the CLI first type `dcos package describe --config arangodb`. This will give you a flat list of default settings.

To get an explanation of the various command line options please check the latest options here (choose the most recent number and have a look at `config.json`):

https://github.com/mesosphere/universe/tree/version-3.x/repo/packages/A/arangodb

After installation DC/OS will start deploying the ArangoDB cluster on the DC/OS cluster. You can watch ArangoDB starting on the "Services" tab in the web interface. Once it is listed as healthy click the link next to it and you should see the ArangoDB web interface.

ArangoDB Mesos framework

As soon as ArangoDB was deployed Mesos will keep your cluster running. The web interface has many monitoring facilities so be sure to make yourself familiar with the DC/OS web interface. As a fault tolerant system Mesos will take care of most failure scenarios automatically. Mesos does that by running ArangoDB as a so called "framework". This framework has been specifically built to keep ArangoDB running in a healthy condition on the Mesos cluster. From time to time a task might fail. The ArangoDB framework will then take care of rescheduling the failed task. As it knows about the very specifics of each cluster task and its role it will automatically take care of most failure scenarios.

To inspect what the framework is doing go to `http://web-interface-url/mesos` in your browser. Locate the task "arangodb" and inspect stderr in the "Sandbox". This can be of interest for example when a slave got lost and the framework is rescheduling the task.

Using ArangoDB

To use ArangoDB as a datastore in your DC/OS cluster you can facilitate the service discovery of DC/OS. Assuming you deployed a standard ArangoDB cluster the mesos dns will know about `arangodb.mesos`. By doing a SRV DNS request (check the documentation of mesos dns) you can find out the port where the internal HAProxy of ArangoDB is running. This will offer a round robin load balancer to access all ArangoDB coordinators.

Scaling ArangoDB

To change the settings of your ArangoDB Cluster access the ArangoDB UI and hit "Nodes". On the scale tab you will have the ability to scale your cluster up and down.

After changing the settings the ArangoDB framework will take care of the rest. Scaling your cluster up is generally a straightforward operation as Mesos will simply launch another task and be done with it. Scaling down is a bit more complicated as the data first has to be moved to some other place so that will naturally take somewhat longer.

Please note that scaling operations might not always work. For example if the underlying Mesos cluster is completely saturated with its running tasks scaling up will not be possible. Scaling down might also fail due to the cluster not being able to move all shards of a DBServer to a new destination because of size limitations. Be sure to check the output of the ArangoDB framework.

Deinstallation

Deinstalling ArangoDB is a bit more difficult as there is much state being kept in the Mesos cluster which is not automatically cleaned up. To deinstall from the command line use the following one liner:

```
dcos arangodb uninstall ; dcos package uninstall arangodb
```

This will first cleanup the state in the cluster and then uninstall arangodb.

arangodb-cleanup-framework

Should you forget to cleanup the state you can do so later by using the arangodb-cleanup-framework container. Otherwise you might not be able to deploy a new arangodb installation.

The cleanup framework will announce itself as a normal ArangoDB. Mesos will recognize this and offer all persistent volumes it still has for ArangoDB to this framework. The cleanup framework will then properly free the persistent volumes. Finally it will clean up any state left in zookeeper (the central configuration manager in a Mesos cluster).

To deploy the cleanup framework, follow the instructions in the github repository. After deployment watch the output in the sandbox of the Mesos web interface. After a while there shouldn't be any persistent resource offers anymore as everything was cleaned up. After that you can delete the cleanup framework again via Marathon.

Apache Mesos and Marathon

You can also install ArangoDB on a bare Apache Mesos cluster provided that Marathon is running on it.

Doing so has the following downsides:

1. Manual Mesos cluster setup
2. You need to implement your own service discovery
3. You are missing the dcos cli
4. Installation and deinstallation are tedious
5. You need to setup some kind of proxy tunnel to access ArangoDB from the outside
6. Sparse monitoring capabilities

However these are things which do not influence ArangoDB itself and operating your cluster like this is fully supported.

Installing via Marathon

To install ArangoDB via marathon you need a proper config file:

```
{
  "id": "arangodb",
  "cpus": 0.25,
  "mem": 256.0,
  "ports": [0, 0, 0],
  "instances": 1,
  "args": [
    "framework",
    "--framework_name=arangodb",
    "--master=zk://172.17.0.2:2181/mesos",
    "--zk=zk://172.17.0.2:2181/arangodb",
    "--user=",
    "--principal=pri",
    "--role=arangodb",
    "--mode=cluster",
    "--async_replication=true",
    "--minimal_resources_agent=mem(*):512;cpus(*):0.25;disk(*):512",
    "--minimal_resources_dbserver=mem(*):512;cpus(*):0.25;disk(*):1024",
    "--minimal_resources_secondary=mem(*):512;cpus(*):0.25;disk(*):1024",
    "--minimal_resources_coordinator=mem(*):512;cpus(*):0.25;disk(*):1024",
    "--nr_agents=1",
    "--nr_dbservers=2",
    "--nr_coordinators=2",
    "--failover_timeout=86400",
    "--arangodb_image=arangodb/arangodb-mesos:3.1",
    "--secondaries_with_dbservers=false",
    "--coordinators_with_dbservers=false"
  ],
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "arangodb/arangodb-mesos-framework:3.1",
      "network": "HOST"
    }
  },
  "healthChecks": [
    {
      "protocol": "HTTP",
      "path": "/framework/v1/health.json",
      "gracePeriodSeconds": 3,
      "intervalSeconds": 10,
      "portIndex": 0,
      "timeoutSeconds": 10,
      "maxConsecutiveFailures": 0
    }
  ]
}
```

Carefully review the settings (especially the IPs and the resources). Then you can deploy to Marathon:

```
curl -X POST -H "Content-Type: application/json" http://url-of-marathon/v2/apps -d @arangodb3.json
```

Alternatively use the web interface of Marathon to deploy ArangoDB. It has a JSON mode and you can use the above configuration file.

Deinstallation via Marathon

As with DC/OS you first need to properly cleanup any state leftovers.

The easiest is to simply delete ArangoDB and then deploy the cleanup-framework (see section arangodb-cleanup-framework).

Configuration options

The Arangodb Mesos framework has a ton of different options which are listed and described here:
https://github.com/arangodb/arangodb-mesos-framework/tree/3.1

Launching an ArangoDB cluster for testing

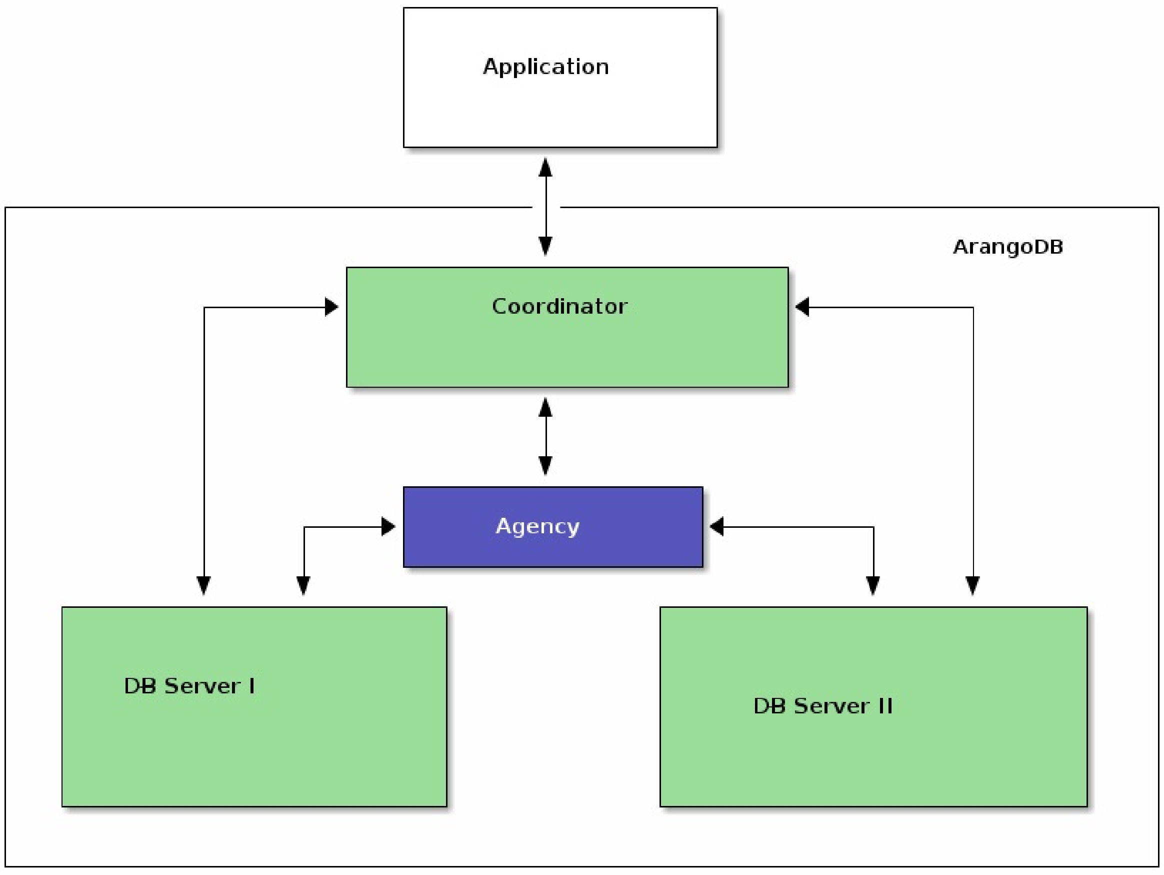
An ArangoDB cluster consists of several running tasks which form the cluster. ArangoDB itself won't start or monitor any of these tasks. So it will need some kind of supervisor which is monitoring and starting these tasks. For production usage we recommend using Apache Mesos as the cluster supervisor.

However starting a cluster manually is possible and is a very easy method to get a first impression of what an ArangoDB cluster looks like.

The easiest way to start a local cluster for testing purposes is to run `scripts/startLocalCluster.sh` from a clone of the source repository after compiling ArangoDB from source (see instructions in the file `README_maintainers.md` in the repository. This will start 1 Agency, 2 DBServers and 1 Coordinator. To stop the cluster issue `scripts/stopLocalCluster.sh` .

This section will discuss the required parameters for every role in an ArangoDB cluster. Be sure to read the Architecture documentation to get a basic understanding of the underlying architecture and the involved roles in an ArangoDB cluster.

In the following sections we will go through the relevant options per role.



Agency

To start up an agency you first have to activate it. This is done by providing `--agency.activate true` .

To start up the agency in its fault tolerant mode set the `--agency.size` to `3` . You will then have to provide at least 3 agents before the agency will start operation.

During initialization the agents have to find each other. To do so provide at least one common `--agency.endpoint` . The agents will then coordinate startup themselves. They will announce themselves with their external address which may be specified using `--agency.my-address` . This is required in bridged docker setups or NATed environments.

So in summary this is what your startup might look like:

```
arangod --server.endpoint tcp://0.0.0.0:5001 --agency.my-address=tcp://127.0.0.1:5001 --server.authentication false --ag
arangod --server.endpoint tcp://0.0.0.0:5002 --agency.my-address=tcp://127.0.0.1:5002 --server.authentication false --ag
arangod --server.endpoint tcp://0.0.0.0:5003 --agency.my-address=tcp://127.0.0.1:5003 --server.authentication false --ag
```

If you are happy with a single agent, then simply use a single command like this:

```
arangod --server.endpoint tcp://0.0.0.0:5001 --server.authentication false --agency.activate true --agency.size 1 --agen
```

Furthermore, in the following sections when `--cluster.agency-address` is used multiple times to specify all three agent addresses, just use a single option `--cluster.agency.address tcp://127.0.0.1:5001` instead.

Coordinators and DBServers

These two roles share a common set of relevant options. First you should specify the role using `--cluster.my-role`. This can either be `PRIMARY` (a database server) or `COORDINATOR`. Both need some unique information with which they will register in the agency, too. This could for example be some combination of host name and port or whatever you have at hand. However it must be unique for each instance and be provided as value for `--cluster.my-local-info`. Furthermore provide the external endpoint (IP and port) of the task via `--cluster.my-address`.

The following is a full-example of what it might look like:

```
arangod --server.authentication=false --server.endpoint tcp://0.0.0.0:8529 --cluster.my-address tcp://127.0.0.1:8529 --c
arangod --server.authentication=false --server.endpoint tcp://0.0.0.0:8530 --cluster.my-address tcp://127.0.0.1:8530 --c
arangod --server.authentication=false --server.endpoint tcp://0.0.0.0:8531 --cluster.my-address tcp://127.0.0.1:8531 --c
```

Note in particular that the endpoint descriptions given under `--cluster.my-address` and `--cluster.agency-endpoint` must not use the IP address `0.0.0.0` because they must contain an actual address that can be routed to the corresponding server. The `0.0.0.0` in `--server.endpoint` simply means that the server binds itself to all available network devices with all available IP addresses.

Upon registering with the agency during startup the cluster will assign an ID to every task. The generated ID will be printed out to the log or can be accessed via the http API by calling `http://server-address/_admin/server/id`. Should you ever have to restart a task, simply reuse the same value for `--cluster.my-local-info` and the same ID will be picked.

You have now launched a complete ArangoDB cluster and can contact its coordinator at the endpoint `tcp://127.0.0.1:8531`, which means that you can reach the web UI under `http://127.0.0.1:8531`.

Secondaries

Secondaries need a bit more work. Secondaries need to have some primary assigned. To do that there is a special route. To register a secondary you must first find out the Server-ID of the primary server. Then generate your own ID for the secondary you are about to start and call one of the coordinators like this (replace the value of "newSecondary" in the command):

```
curl -f -X PUT --data '{"primary": "DBServer001", "oldSecondary": "none", "newSecondary": "Secondary001"}' -H "Content-T
```

If that call was successful you can start the secondary. Instead of providing `--cluster.my-local-info` you should now provide the Id in the curl call above via `--cluster.my-id`. You can omit the `--cluster.my-role` in this case. The secondary will find out from the agency about its role.

To sum it up:

```
curl -f -X PUT --data '{"primary": "DBServer001", "oldSecondary": "none", "newSecondary": "Secondary001"}' -H "Content-T
curl -f -X PUT --data '{"primary": "DBServer002", "oldSecondary": "none", "newSecondary": "Secondary002"}' -H "Content-T
```

Launching an ArangoDB cluster on multiple machines

Essentially, one can use the method from the previous section to start an ArangoDB cluster on multiple machines as well. The only changes are that one has to replace all local addresses `127.0.0.1` by the actual IP address of the corresponding server.

If we assume that you want to start you ArangoDB cluster on three different machines with IP addresses

```
192.168.1.1
192.168.1.2
192.168.1.3
```

then the commands you have to use are (you can use host names if they can be resolved to IP addresses on all machines):

On 192.168.1.1:

```
arangod --server.endpoint tcp://0.0.0.0:5001 --server.authentication false --agency.activate true --agency.size 3 --agen
```

On 192.168.1.2:

```
arangod --server.endpoint tcp://0.0.0.0:5002 --server.authentication false --agency.activate true --agency.size 3 --agen
```

On 192.168.1.3:

```
arangod --server.endpoint tcp://0.0.0.0:5003 --server.authentication false --agency.activate true --agency.size 3 --agen
```

On 192.168.1.1:

```
arangod --server.authentication=false --server.endpoint tcp://0.0.0.0:8529 --cluster.my-address tcp://192.168.1.1:8529 -
```

On 192.168.1.2:

```
arangod --server.authentication=false --server.endpoint tcp://0.0.0.0:8530 --cluster.my-address tcp://192.168.1.2:8530 -
```

On 192.168.1.3:

```
arangod --server.authentication=false --server.endpoint tcp://0.0.0.0:8531 --cluster.my-address tcp://192.168.1.3:8531 -
```

Obviously, it would no longer be necessary to use different port numbers on different servers. We have chosen to keep all port numbers in comparison to the local setup to minimize the necessary changes.

If you want to setup secondaries, the following commands will do the job:

On 192.168.1.2:

```
curl -f -X PUT --data '{"primary": "DBServer001", "oldSecondary": "none", "newSecondary": "Secondary001"}' -H "Content-T
```

On 192.168.1.1:

```
curl -f -X PUT --data '{"primary": "DBServer002", "oldSecondary": "none", "newSecondary": "Secondary002"}' -H "Content-T
```

Note that we have started the `Secondary002` on the same machine as `DBServer001` and `Secondary001` on the same machine as `DBServer002` to avoid that a complete pair is lost when a machine fails. Furthermore, note that ArangoDB does not yet perform automatic failover to the secondary, if a primary fails. This only works in the Apache Mesos setting. For synchronous replication, automatic failover always works and you do not need to setup secondaries for this.

After having swallowed these longish commands, we hope that you appreciate the simplicity of the setup with Apache Mesos and DC/OS.

ArangoDB Cluster and Docker

Networking

A bit of extra care has to be invested due to the way in which Docker isolates its network. By default it fully isolates the network and by doing so an endpoint like `--server.endpoint tcp://0.0.0.0:8529` will only bind to all interfaces inside the Docker container which does not include any external interface on the host machine. This may be sufficient if you just want to access it locally but in case you want to expose it to the outside you must facilitate Dockers port forwarding using the `-p` command line option. Be sure to check the official Docker documentation.

To simply make arangodb available on all host interfaces on port 8529:

`docker run -p 8529:8529 -e ARANGO_NO_AUTH=1 arangodb`

Another possibility is to start Docker via network mode `host`. This is possible but generally not recommended. To do it anyway check the Docker documentation for details.

Docker and Cluster tasks

To start the cluster via Docker is basically the same as starting locally or on multiple machines. However just like with the single networking image we will face networking issues. You can simply use the `-p` flag to make the individual task available on the host machine or you could use Docker's links to enable task intercommunication.

Please note that there are some flags that specify how ArangoDB can reach a task from the outside. These are very important and built for this exact usecase. An example configuration might look like this:

```
docker run -e ARANGO_NO_AUTH=1 -p 192.168.1.1:10000:8529 arangodb/arangodb arangod --server.endpoint tcp://0.0.0.0:8529
```

This will start a primary DB server within a Docker container with an isolated network. Within the Docker container it will bind to all interfaces (this will be 127.0.0.1:8529 and some internal Docker ip on port 8529). By supplying `-p 192.168.1.1:10000:8529` we are establishing a port forwarding from our local IP (192.168.1.1 port 10000 in this example) to port 8529 inside the container. Within the command we are telling arangod how it can be reached from the outside `--cluster.my-address tcp://192.168.1.1:10000`. This information will be forwarded to the agency so that the other tasks in your cluster can see how this particular DBServer may be reached.

Launching ArangoDB's standalone "agency"

Multiple ArangoDB instances can be deployed as a fault-tolerant distributed state machine.

What is a fault-tolerant state machine in the first place?

In many service deployments consisting of arbitrary components distributed over multiple machines one is faced with the challenge of creating a dependable centralised knowledge base or configuration. Implementation of such a service turns out to be one of the most fundamental problems in information engineering. While it may seem as if the realisation of such a service is easily conceivable, dependablity formulates a paradoxon on computer networks per se. On the one hand, one needs a distributed system to avoid a single point of failure. On the other hand, one has to establish consensus among the computers involved.

Consensus is the keyword here and its realisation on a network proves to be far from trivial. Many papers and conference proceedings have discussed and evaluated this key challenge. Two algorithms, historically far apart, have become widely popular, namely Paxos and its derivatives and Raft. Discussing them and their differences, although highly enjoyable, must remain far beyond the scope of this document. Find the references to the main publications at the bottom of this page.

At ArangoDB, we decided to implement Raft as it is arguably the easier to understand and thus implement. In simple terms, Raft guarantees that a linear stream of transactions, is replicated in realtime among a group of machines through an elected leader, who in turn must have access to and project leadership upon an overall majority of participating instances. In ArangoDB we like to call the entirety of the components of the replicated transaction log, that is the machines and the ArangoDB instances, which constitute the replicated log, the agency.

Startup

The agency must consists of an odd number of agents in order to be able to establish an overall majority and some means for the agents to be able to find one another at startup.

The most obvious way would be to inform all agents of the addresses and ports of the rest. This however, is more information than needed. For example, it would suffice, if all agents would know the address and port of the next agent in a cyclic fashion. Another straitforward solution would be to inform all agents of the address and port of say the first agent.

Clearly all cases, which would form disjunct subsets of agents would break or in the least impair the functionality of the agency. From there on the agents will gossip the missing information about their peers.

Typically, one achieves fairly high fault-tolerance with low, odd number of agents while keeping the necessary network traffic at a minimum. It seems that the typical agency size will be in range of 3 to 7 agents.

The below commands start up a 3-host agency on one physical/logical box with ports 8529, 8530 and 8531 for demonstration purposes. The adress of the first instance, port 8529, is known to the other two. After atmost 2 rounds of gossipping, the last 2 agents will have a complete picture of their surrounding and persist it for the next restart.

```
./build/bin/arangod --agency.activate true --agency.size 3 --agency.my-address tcp://localhost:8529 --server.authenticat
./build/bin/arangod --agency.activate true --agency.size 3 --agency.endpoint tcp://localhost:8529 --agency.my-address tc
./build/bin/arangod --agency.activate true --agency.size 3 --agency.endpoint tcp://localhost:8529 --agency.my-address tc
```

The parameter `agency.endpoint` is the key ingredient for the second and third instances to find the first instance and thus form a complete agency. Please refer to the the shell-script `scripts/startStandaloneAgency.sh` on github or in the source directory.

Key-value-store API

The agency should be up and running within a couple of seconds, during which the instances have gossiped their way into knowing the other agents and elected a leader. The public API can be checked for the state of the configuration:

```
curl -s localhost:8529/_api/agency/config
```

```
{
  "term": 1,
  "leaderId": "f5d11cde-8468-4fd2-8747-b4ef5c7dfa98",
  "lastCommitted": 1,
  "lastAcked": {
    "ac129027-b440-4c4f-84e9-75c042942171": 0.21,
    "c54dbb8a-723d-4c82-98de-8c841a14a112": 0.21,
    "f5d11cde-8468-4fd2-8747-b4ef5c7dfa98": 0
  },
  "configuration": {
    "pool": {
      "ac129027-b440-4c4f-84e9-75c042942171": "tcp://localhost:8531",
      "c54dbb8a-723d-4c82-98de-8c841a14a112": "tcp://localhost:8530",
      "f5d11cde-8468-4fd2-8747-b4ef5c7dfa98": "tcp://localhost:8529"
    },
    "active": [
      "ac129027-b440-4c4f-84e9-75c042942171",
      "c54dbb8a-723d-4c82-98de-8c841a14a112",
      "f5d11cde-8468-4fd2-8747-b4ef5c7dfa98"
    ],
    "id": "f5d11cde-8468-4fd2-8747-b4ef5c7dfa98",
    "agency size": 3,
    "pool size": 3,
    "endpoint": "tcp://localhost:8529",
    "min ping": 0.5,
    "max ping": 2.5,
    "supervision": false,
    "supervision frequency": 5,
    "compaction step size": 1000,
    "supervision grace period": 120
  }
}
```

To highlight some details in the above output look for `"term"` and `"leaderId"`. Both are key information about the current state of the Raft algorithm. You may have noted that the first election term has established a random leader for the agency, who is in charge of replication of the state machine and for all external read and write requests until such time that the process gets isolated from the other two subsequenctly losing its leadership.

Read and Write APIs

Generally, all read and write accesses are transactions moreover any read and write access may consist of multiple such transactions formulated as arrays of arrays in JSON documents.

Read transaction

An agency started from scratch will deal with the simplest query as follows:

```
curl -L localhost:8529/_api/agency/read -d '[["/"]]'
```

```
[{}]
```

The above request for an empty key value store will return with an empty document. The inner array brackets will aggregate a result from multiple sources in the key-value-store while the outer array will deliver multiple such aggregated results. Also note the `-L` curl flag, which allows the request to follow redirects to the current leader.

Consider the following key-value-store:

```
{
  "baz": 12,
  "corge": {
    "e": 2.718281828459045,
    "pi": 3.14159265359
  },
  "foo": {
    "bar": "Hello World"
  },
  "qux": {
    "quux": "Hello World"
  }
}
```

The following array of read transactions will yield:

```
curl -L localhost:8529/_api/agency/read -d '[["/foo", "/foo/bar", "/baz"],["/qux"]]'
```

```
[
  {
    "baz": 12,
    "foo": {
      "bar": "Hello World"
    }
  },
  {
    "qux": {
      "quux": "Hello World"
    }
  }
]
```

Note that the result is an array of two results for the first and second read transactions from above accordingly. Also note that the results from the first read transaction are aggregated into

```
{
  "baz": 12,
  "foo": {
    "bar": "Hello World"
  }
}
```

The aggregation is performed on 2 levels:

1. `/foo/bar` is eliminated as a subset of `/foo`
2. The results from `/foo` and `/bar` are joined

The word transaction means here that it is guaranteed that all aggregations happen in quasi-realtime and that no write access could have happened in between.

Btw, the same transaction on the virgin key-value store would produce `[{},{}]`

Write API:

The write API must unfortunately be a little more complex. Multiple roads lead to Rome:

```
curl -L localhost:8529/_api/agency/write -d '[[{"/foo":{"op":"push","new":"bar"}}]]'
curl -L localhost:8529/_api/agency/write -d '[[{"/foo":{"op":"push","new":"baz"}}]]'
curl -L localhost:8529/_api/agency/write -d '[[{"/foo":{"op":"push","new":"qux"}}]]'
```

and

```
curl -L localhost:8529/_api/agency/write -d '[[{"foo":["bar","baz","qux"]}]]'
```

are equivalent for example and will create and fill an array at `/foo`. Here, again, the outermost array is the container for the transaction arrays.

We documentent a complete guide of the API in the API section.

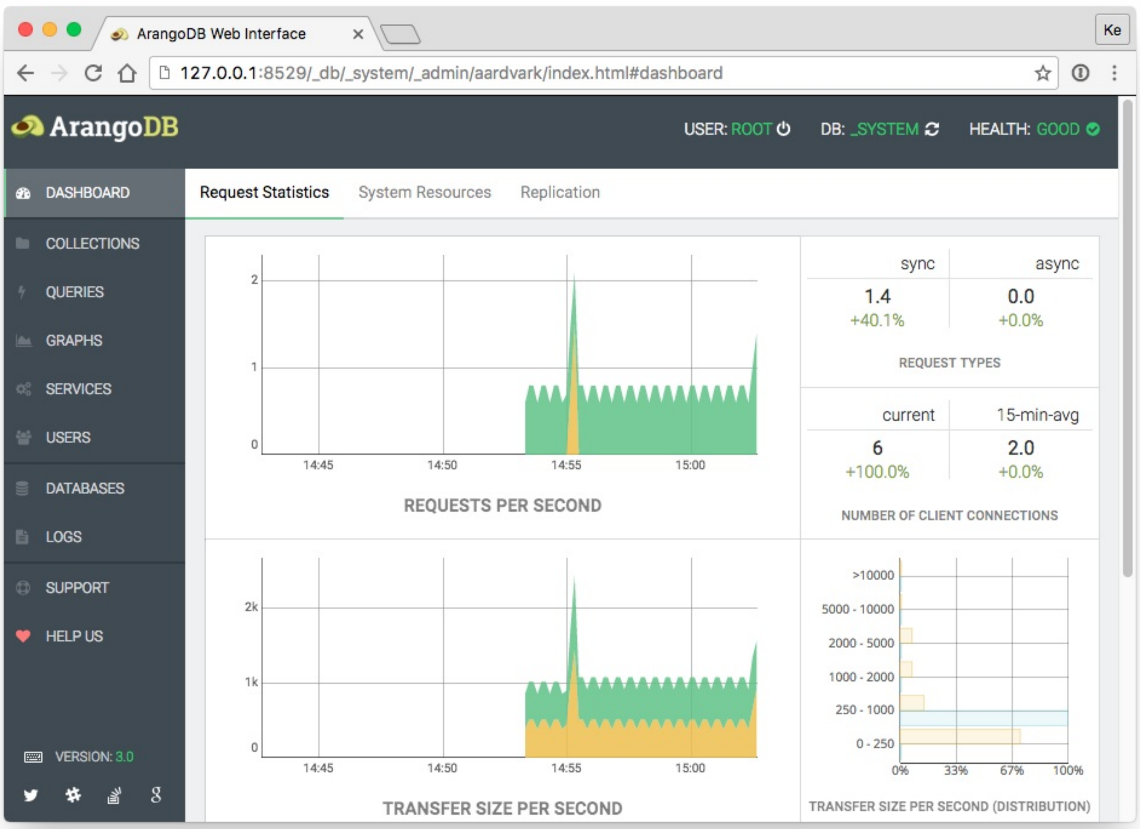
Administration

Most administration can be managed using the *arangosh*.

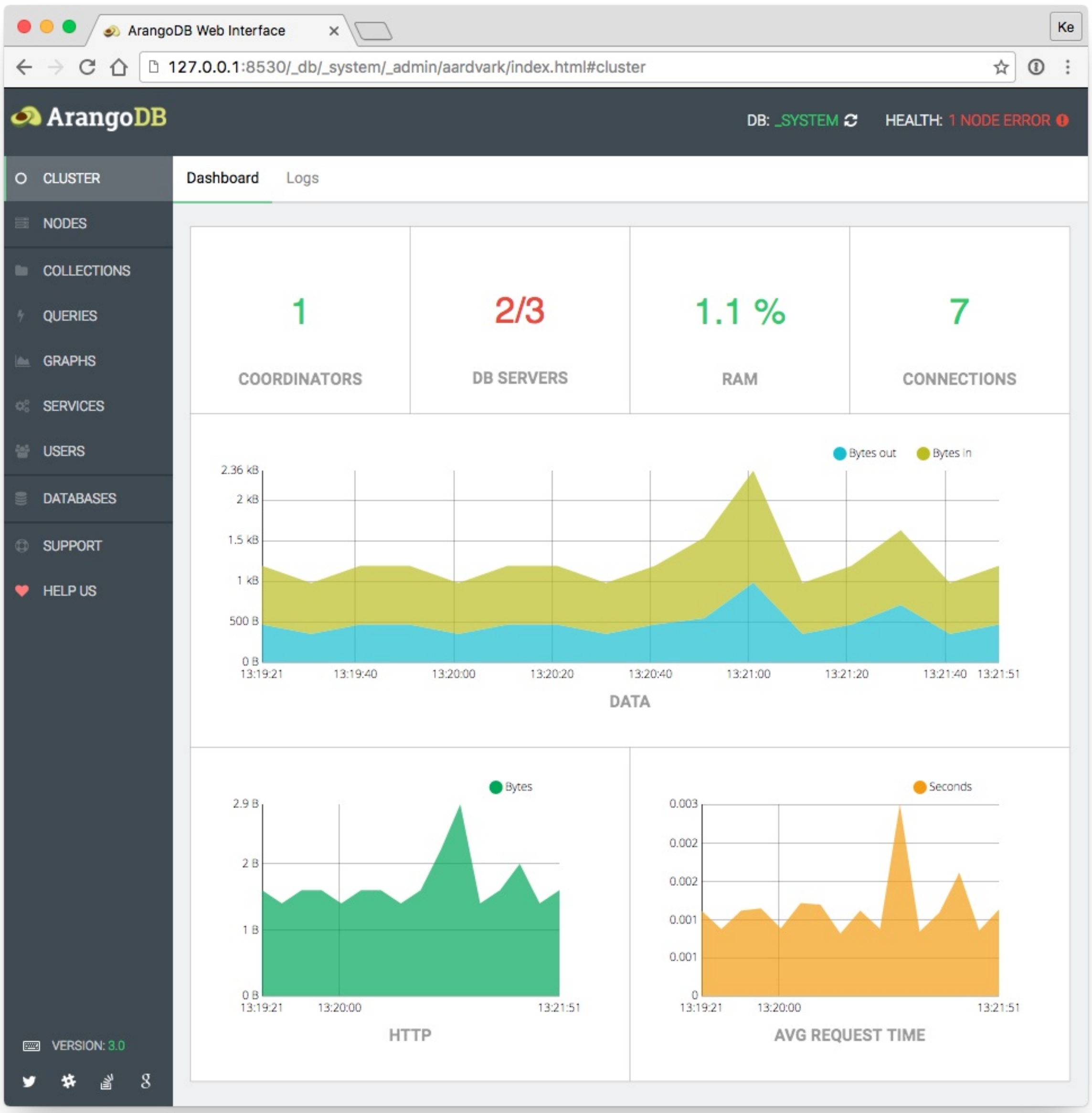
Web Interface

ArangoDB comes with a built-in web interface for administration. The interface differs for standalone instances and cluster setups.

Standalone:



Cluster:


ArangoDB Web Interface
127.0.0.1:8530/_db/_system/_admin/aardvark/index.html#cluster
ArangoDB
DB: _SYSTEM
HEALTH: 1 NODE ERROR
CLUSTER
NODES
COLLECTIONS
QUERIES
GRAPHS
SERVICES
USERS
DATABASES
SUPPORT
HELP US
VERSION: 3.0
Dashboard
Logs
1
COORDINATORS
2/3
DB SERVERS
1.1 %
RAM
7
CONNECTIONS
Bytes out
Bytes in
DATA
Bytes
HTTP
Seconds
AVG REQUEST TIME

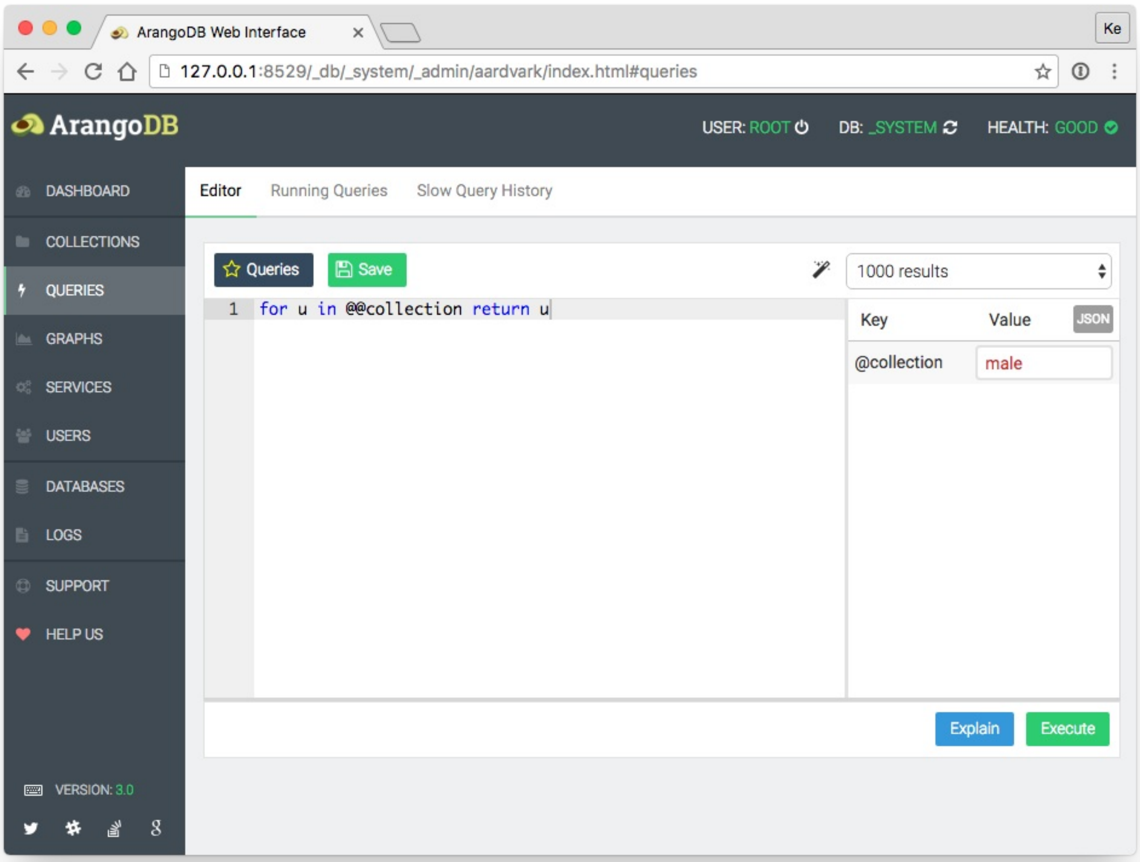
Query View

The query view offers you three different subviews:

- Editor
- Running Queries
- Slow Query History

AQL Query Editor

The web interface offers a AQL Query Editor:



```
for u in @@collection return u
```

Key	Value
@collection	male

The editor is split into two parts, the query editor pane and the bind parameter pane.

The left pane is your regular query input field, where you can edit and then execute or explain your queries. By default, the entered bind parameter will automatically be recognized and shown in the bind parameter table in the right pane, where you can easily edit them.

The input fields are equipped with type detection. This means you don't have to use quote marks around string, just write them as-is. Numbers will be treated as numbers, *true* and *false* as booleans, *null* as null-type value. Square brackets can be used to define arrays, and curly braces for objects (keys and values have to be surrounded by double quotes). This will mostly be what you want. But if you want to force something to be treated as string, use quotation marks for the value:

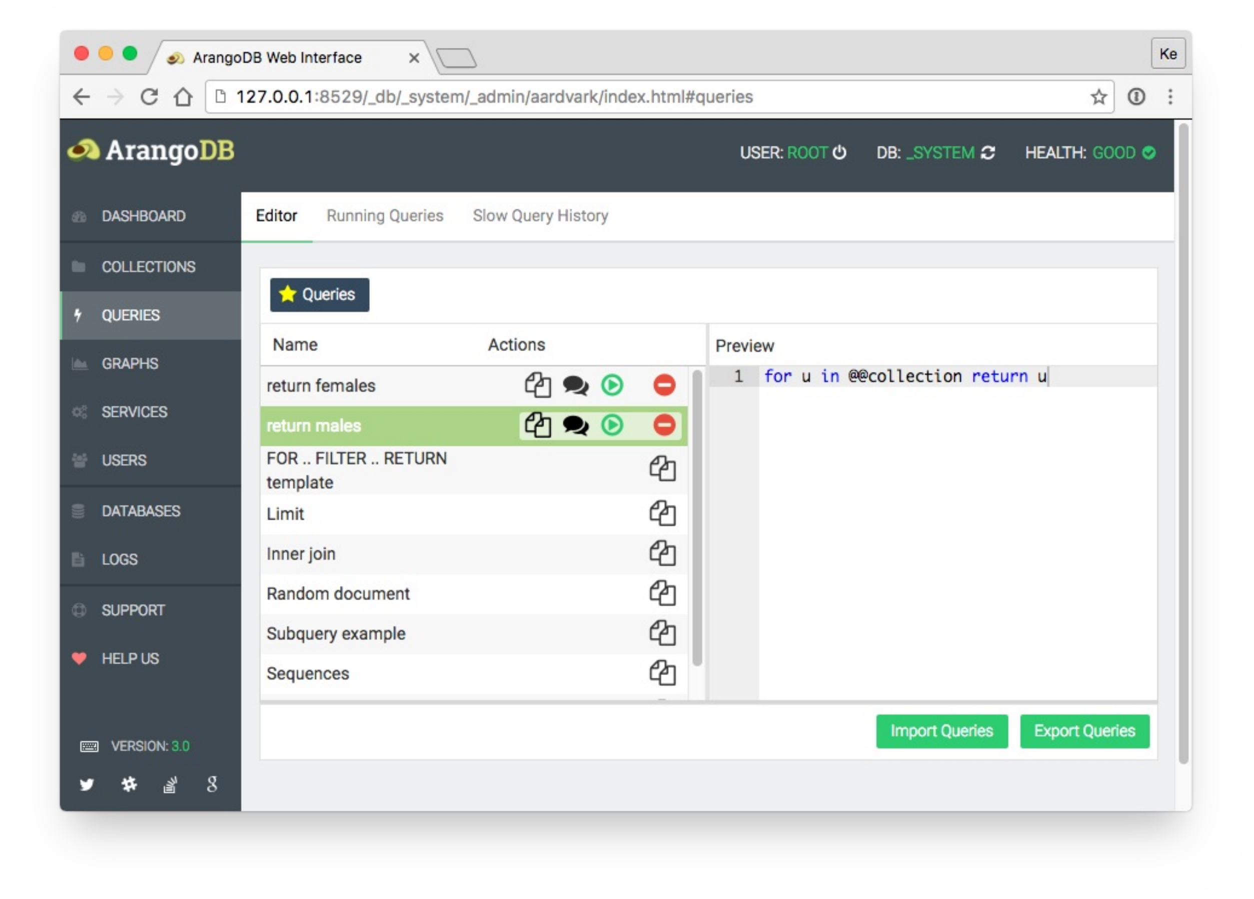
```
123   // interpreted as number
"123" // interpreted as string

["foo", "bar", 123, true] // interpreted as array
['foo', 'bar', 123, true] // interpreted as string
```

If you are used to work with JSON, you may want to switch the bind parameter editor to JSON mode by clicking on the upper right toggle button. You can then edit the bind parameters in raw JSON format.

Custom Queries

To save the current query use the *Save* button in the top left corner of the editor or use the shortcut (see below).



```
for u in @@collection return u
```

By pressing the *Queries* button in the top left corner of the editor you activate the custom queries view. Here you can select a previously stored custom query or one of our query examples.

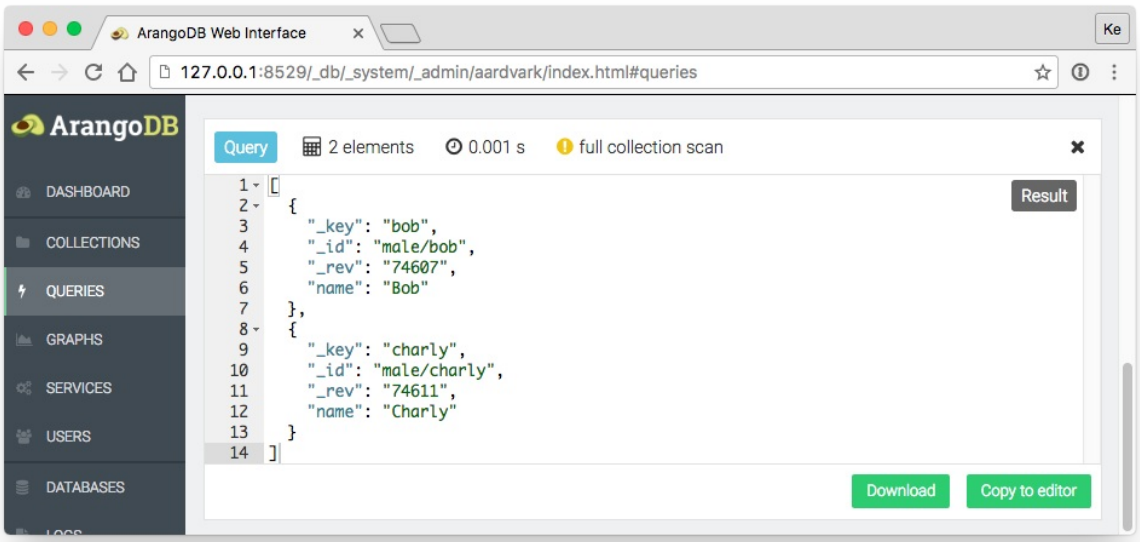
Click on a query title to get a code preview. In addition, there are action buttons to:

- Copy to editor
- Explain query
- Run query
- Delete query

For the built-in example queries, there is only *Copy to editor* available.

To export or import queries to and from JSON you can use the buttons on the right-hand side.

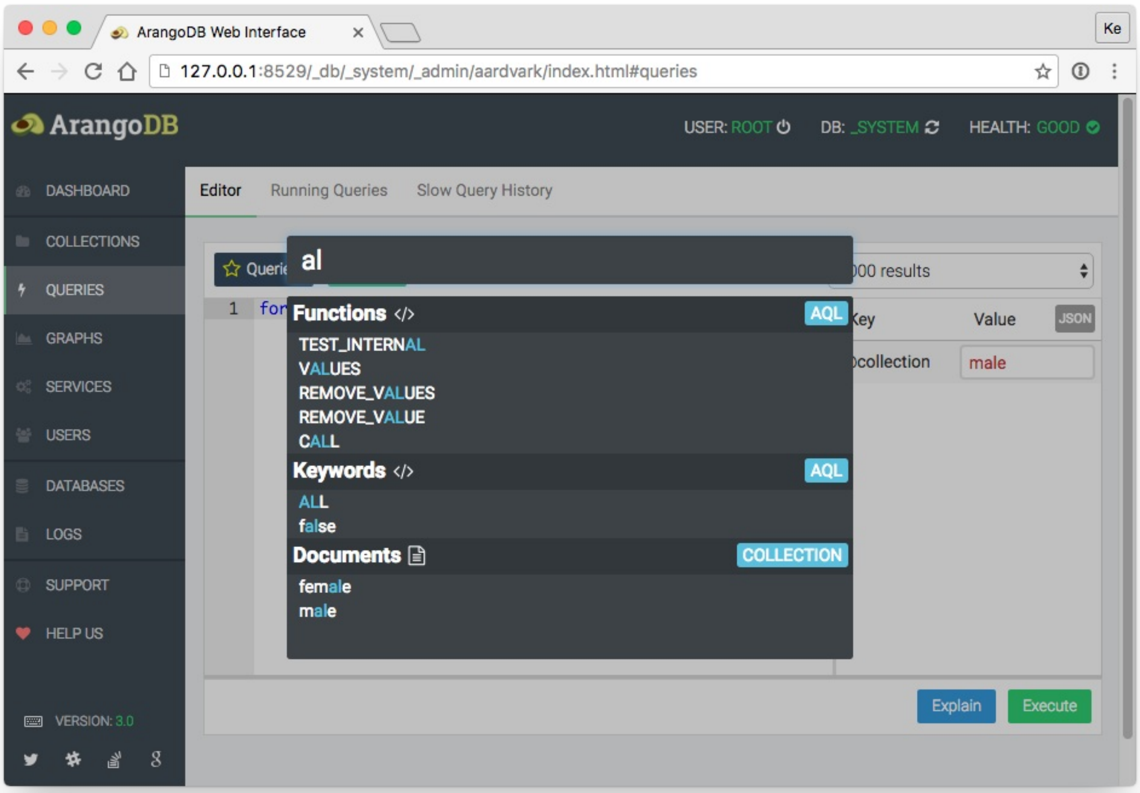
Result



```
[
  {
    "_key": "bob",
    "_id": "male/bob",
    "_rev": "74607",
    "name": "Bob"
  },
  {
    "_key": "charly",
    "_id": "male/charly",
    "_rev": "74611",
    "name": "Charly"
  }
]
```

Each query you execute or explain opens up a new result box, so you are able to fire up multiple queries and view their results at the same time. Every query result box gives you detailed query information and of course the query result itself. The result boxes can be dismissed individually, or altogether using the *Remove results* button. The toggle button in the top right corner of each box switches back and forth between the *Result* and *AQL* query with bind parameters.

Spotlight

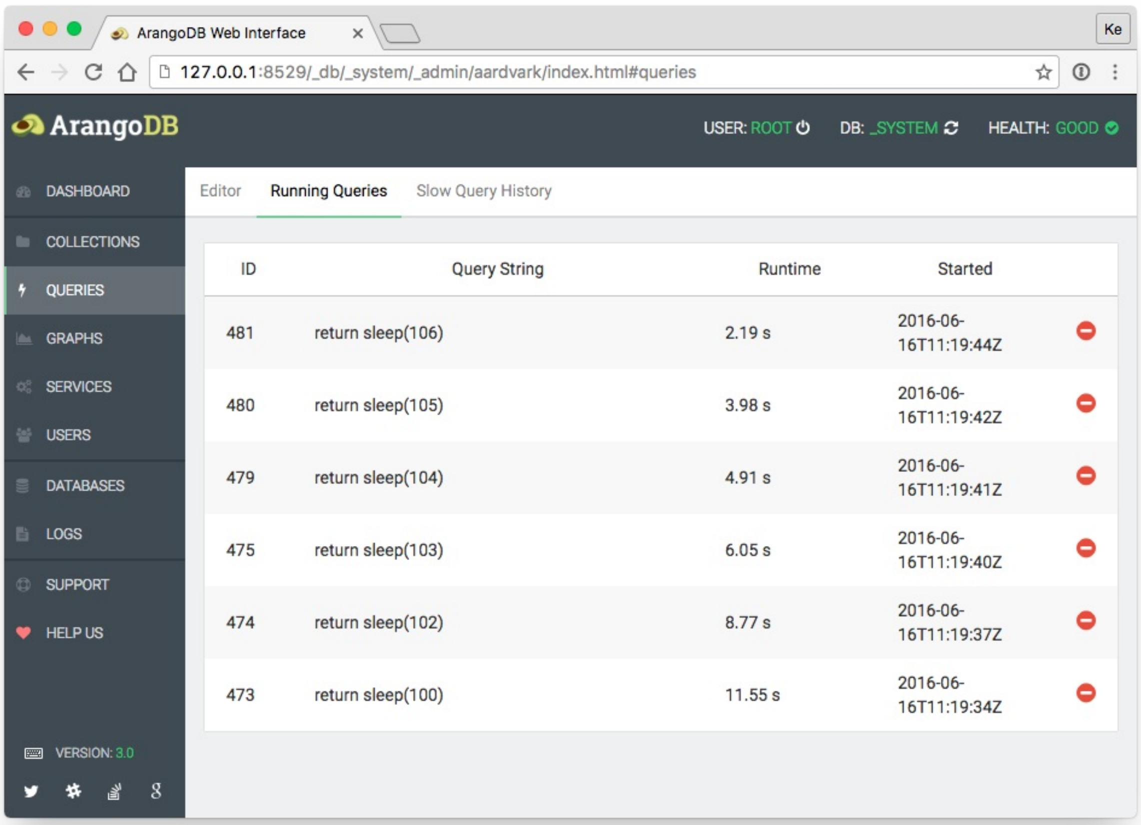


The spotlight feature opens up a modal view. There you can find all AQL keywords, AQL functions and collections (filtered by their type) to help you to be more productive in writing your queries. Spotlight can be opened by the magic wand icon in the toolbar or via shortcut (see below).

AQL Editor Shortcuts

- Ctrl / Cmd + Return to execute a query
- Ctrl / Cmd + Shift + Return to explain a query
- Ctrl / Cmd + Shift + S to save the current query
- Ctrl / Cmd + Shift + C to toggle comments
- Ctrl + Space to open up the spotlight search
- Ctrl + Cmd + Z to undo last change
- Ctrl + Cmd + Shift + Z to redo last change

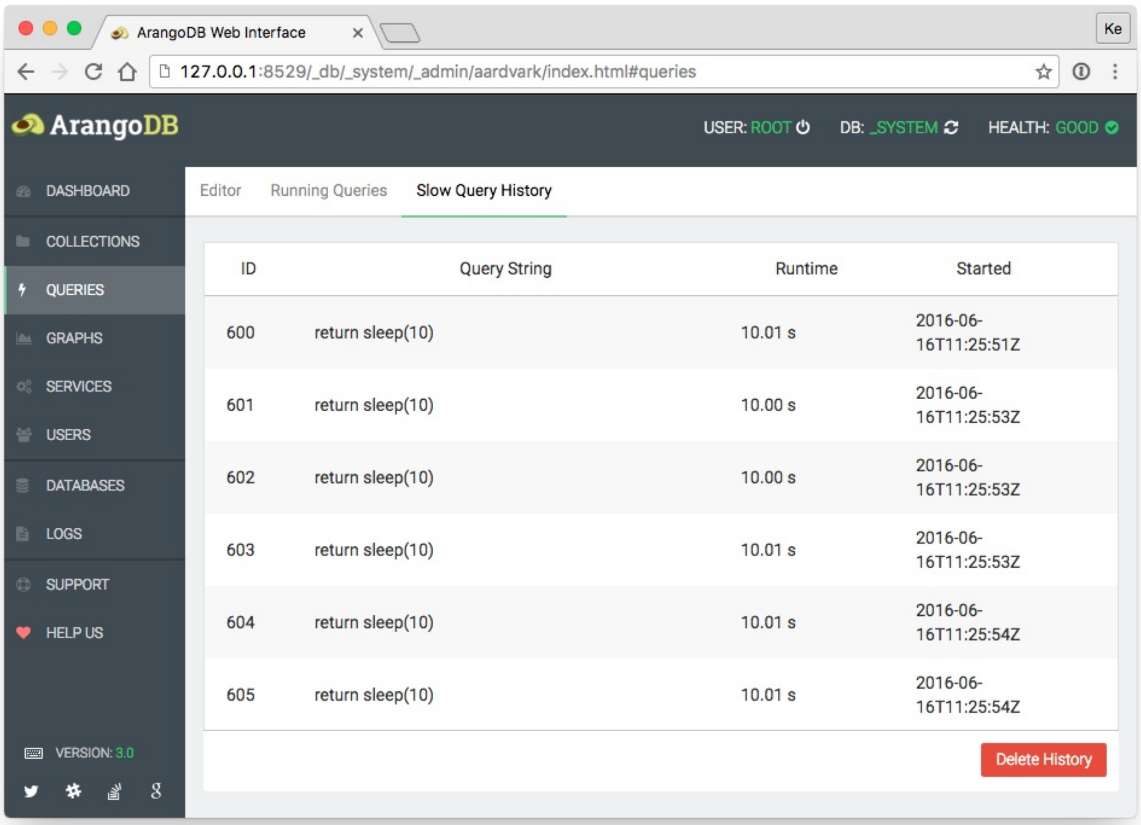
Running Queries



ID	Query String	Runtime	Started
481	return sleep(106)	2.19 s	2016-06-16T11:19:44Z
480	return sleep(105)	3.98 s	2016-06-16T11:19:42Z
479	return sleep(104)	4.91 s	2016-06-16T11:19:41Z
475	return sleep(103)	6.05 s	2016-06-16T11:19:40Z
474	return sleep(102)	8.77 s	2016-06-16T11:19:37Z
473	return sleep(100)	11.55 s	2016-06-16T11:19:34Z

The *Running Queries* tab gives you a compact overview of all running queries. By clicking the red minus button, you can abort the execution of a running query.

Slow Query History

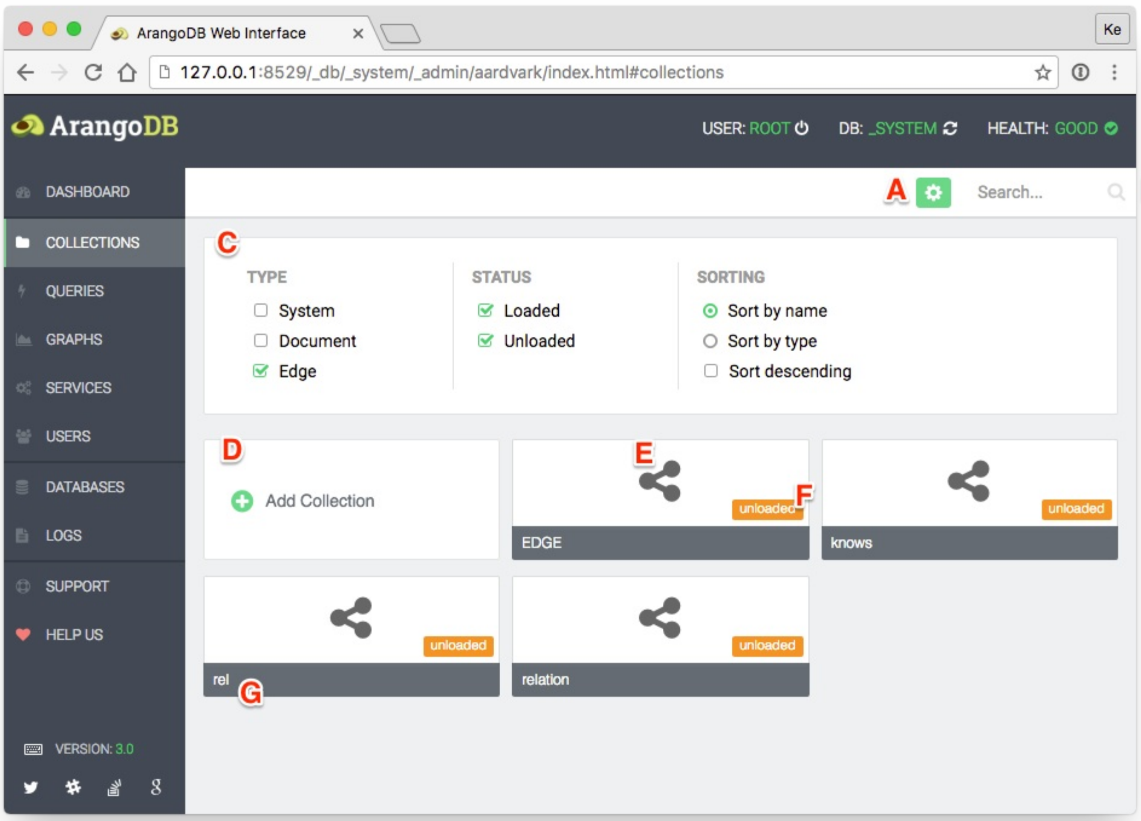


ID	Query String	Runtime	Started
600	return sleep(10)	10.01 s	2016-06-16T11:25:51Z
601	return sleep(10)	10.00 s	2016-06-16T11:25:53Z
602	return sleep(10)	10.00 s	2016-06-16T11:25:53Z
603	return sleep(10)	10.01 s	2016-06-16T11:25:53Z
604	return sleep(10)	10.01 s	2016-06-16T11:25:54Z
605	return sleep(10)	10.01 s	2016-06-16T11:25:54Z

The *Slow Query History* tab gives you a compact overview of all past slow queries.

Collections

The collections section displays all available collections. From here you can create new collections and jump into a collection for details (click on a collection tile).



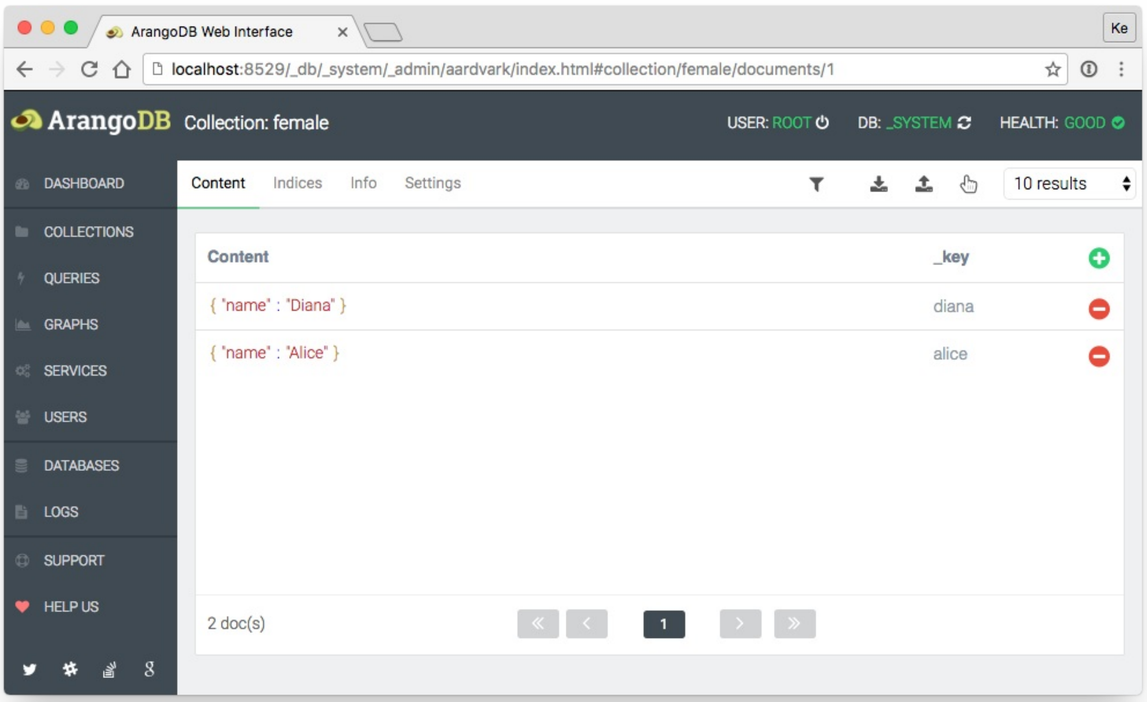
Functions:

- A: Toggle filter properties
- B: Search collection by name
- D: Create collection
- C: Filter properties
- H: Show collection details (click tile)

Information:

- E: Collection type
- F: Collection state(unloaded, loaded, ...)
- G: Collection name

Collection



Content	_key
{ "name" : "Diana" }	diana
{ "name" : "Alice" }	alice

There are four view categories:

1. Content:
    - Create a document
    - Delete a document
    - Filter documents
    - Download documents
    - Upload documents
2. Indices:
    - Create indices
    - Delete indices
3. Info:
    - Detailed collection information and statistics
4. Settings:
    - Configure name, journal size, index buckets, wait for sync
    - Delete collection
    - Truncate collection
    - Unload/Load collection
    - Save modifed properties (name, journal size, index buckets, wait for sync)

Additional information:

Upload format:

I. Line-wise

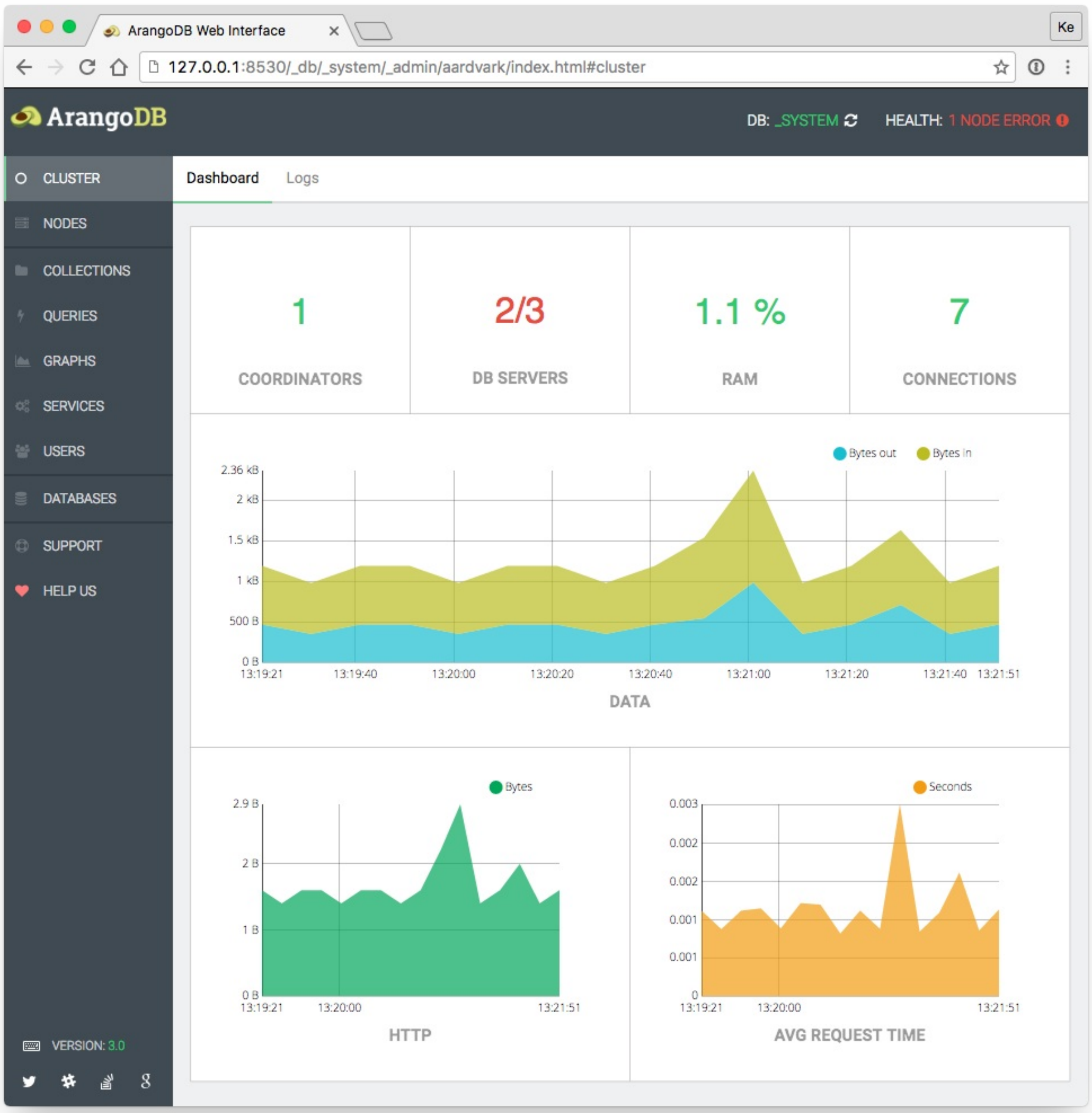
```
{ "_key": "key1", ... }
{ "_key": "key2", ... }
```

II. JSON documents in a list

```
[
  { "_key": "key1", ... },
  { "_key": "key2", ... }
]
```

Cluster

The cluster section displays statistics about the general cluster performance.



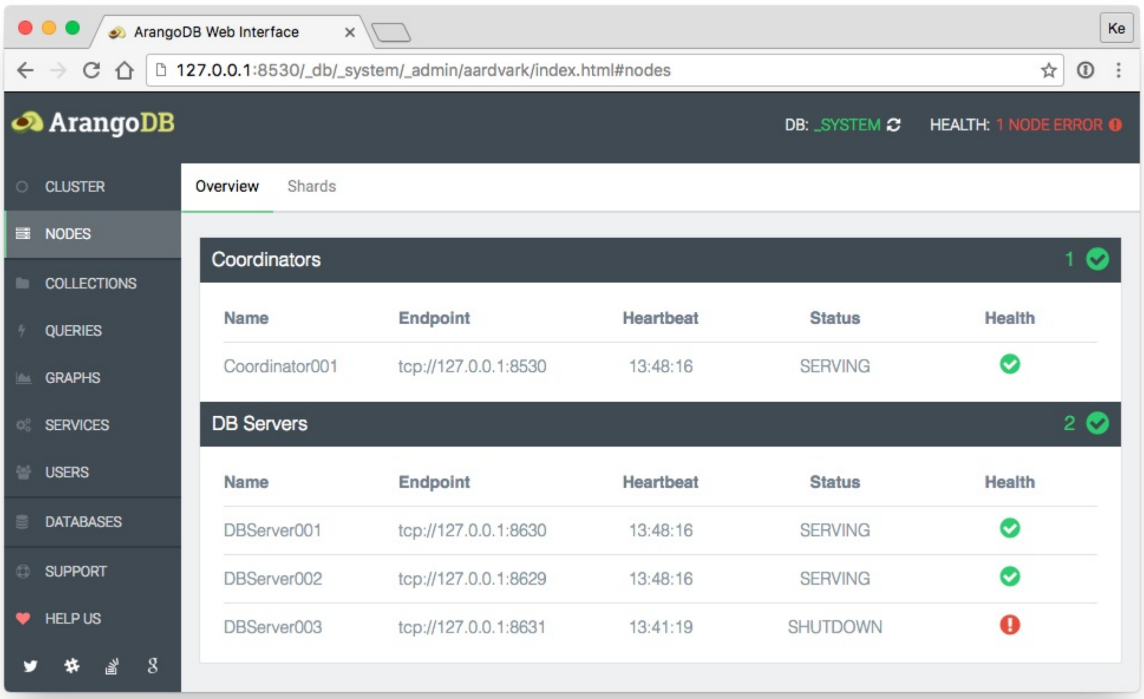
Statistics:

- Available and missing coordinators
- Available and missing database servers
- Memory usage (percent)
- Current connections
- Data (bytes)
- HTTP (bytes)
- Average request time (seconds)

Nodes

Overview

The overview shows available and missing coordinators and database servers.



Coordinators 1

Name	Endpoint	Heartbeat	Status	Health
Coordinator001	tcp://127.0.0.1:8530	13:48:16	SERVING	✓

DB Servers 2

Name	Endpoint	Heartbeat	Status	Health
DBServer001	tcp://127.0.0.1:8630	13:48:16	SERVING	✓
DBServer002	tcp://127.0.0.1:8629	13:48:16	SERVING	✓
DBServer003	tcp://127.0.0.1:8631	13:41:19	SHUTDOWN	!

Functions:

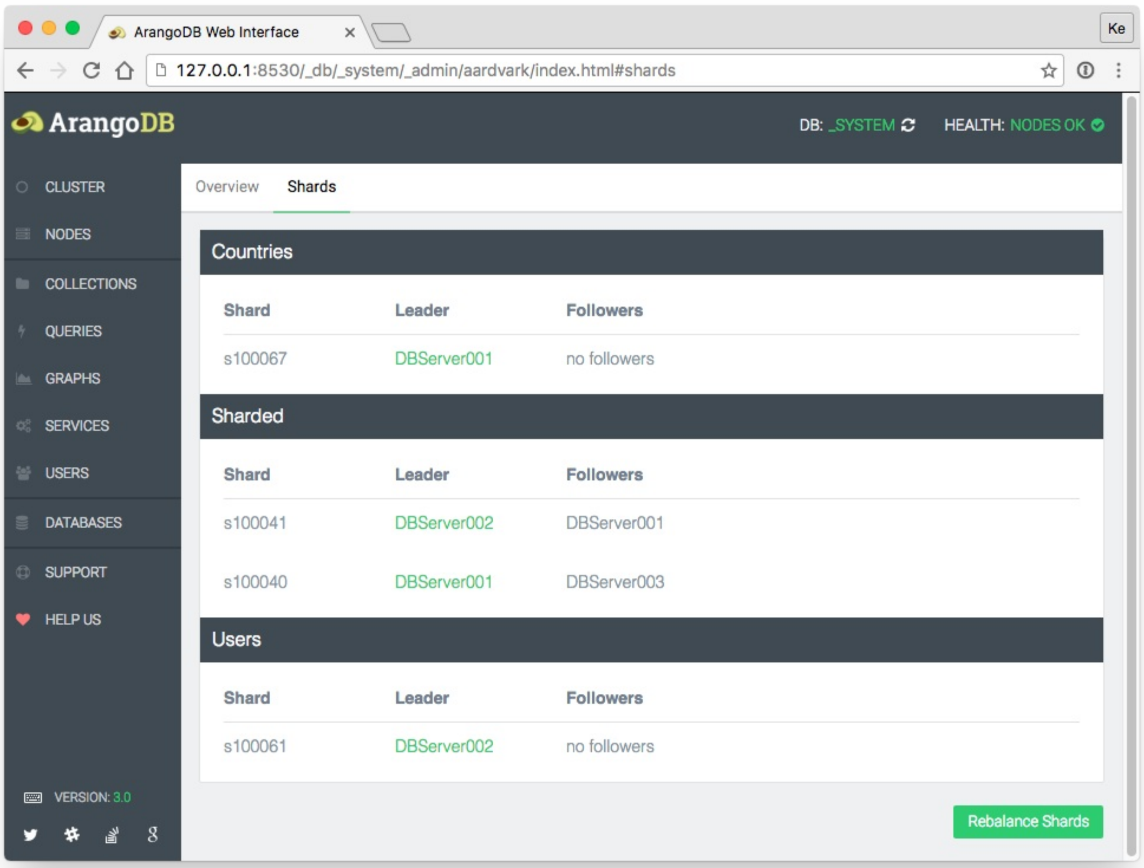
- Coordinator Dashboard: Click on a Coordinator will open a statistics dashboard.

Information (Coordinator / Database servers):

- Name
- Endpoint
- Last Heartbeat
- Status
- Health

Shards

The shard section displays all available sharded collections.



Countries

Shard	Leader	Followers
s100067	DBServer001	no followers

Sharded

Shard	Leader	Followers
s100041	DBServer002	DBServer001
s100040	DBServer001	DBServer003

Users

Shard	Leader	Followers
s100061	DBServer002	no followers

Functions:

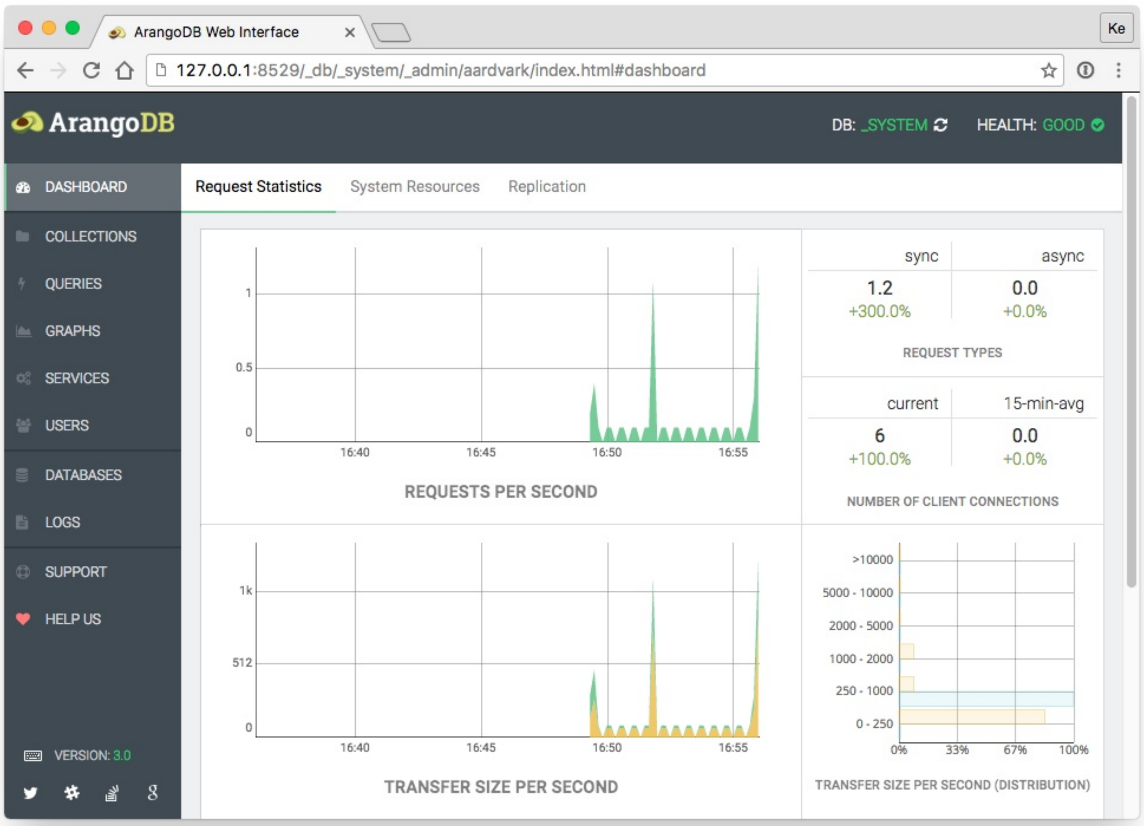
- Move Shard Leader: Click on a leader database of a shard server will open a move shard dialog. Shards can be transferred to all available databas servers, except the leading database server or an available follower.
- Move Shard Follower: Click on a follower database of a shard will open a move shard dialog. Shards can be transferred to all available databas servers, except the leading database server or an available follower.
- Rebalance Shards: A new database server will not have any shards. With the rebalance functionality the cluster will start to rebalance shards including empty database servers.

Information (collection):

- Shard
- Leader (green state: sync is complete)
- Followers

Dashboard

The *Dashboard* tab provides statistics which are polled regularly from the ArangoDB server.



Requests Statistics:

- Requests per second
- Request types
- Number of client connections
- Transfer size
- Transfer size (distribution)
- Average request time
- Average request time (distribution)

System Resources:

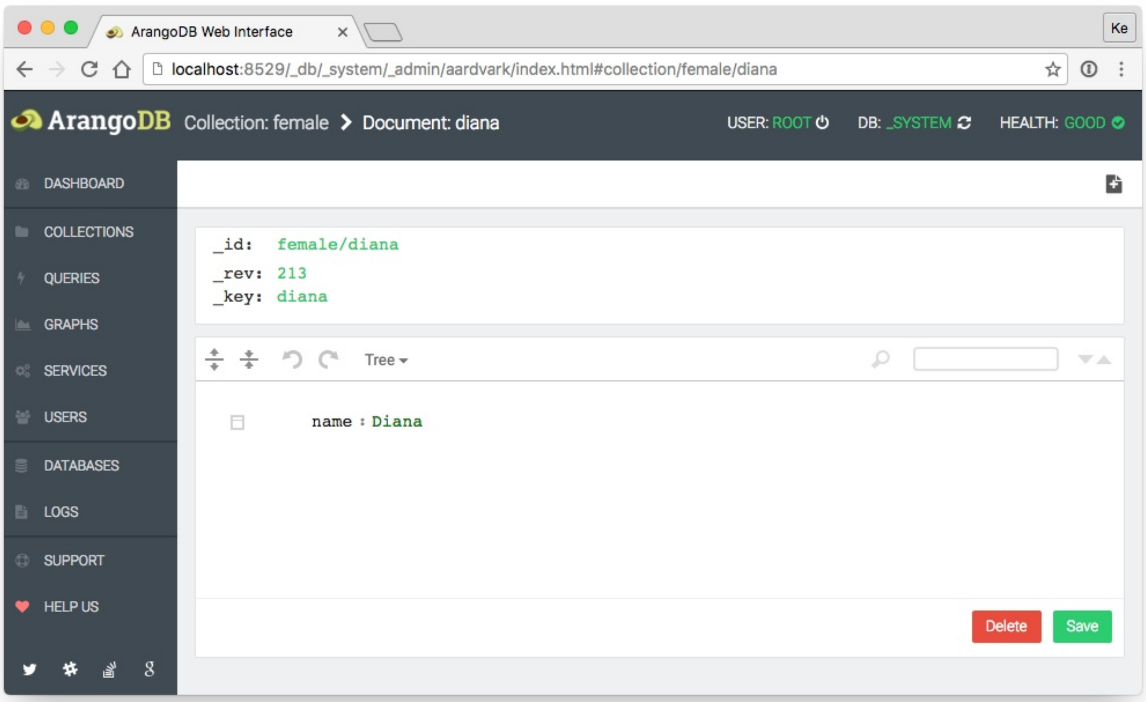
- Number of threads
- Memory
- Virtual size
- Major page faults
- Used CPU time

Replication:

- Replication state
- Totals
- Ticks
- Progress

Document

The document section offers a editor which let you edit documents and edges of a collection.



```
_id:  female/diana
_rev: 213
_key: diana
```

```
name : Diana
```

Functions:

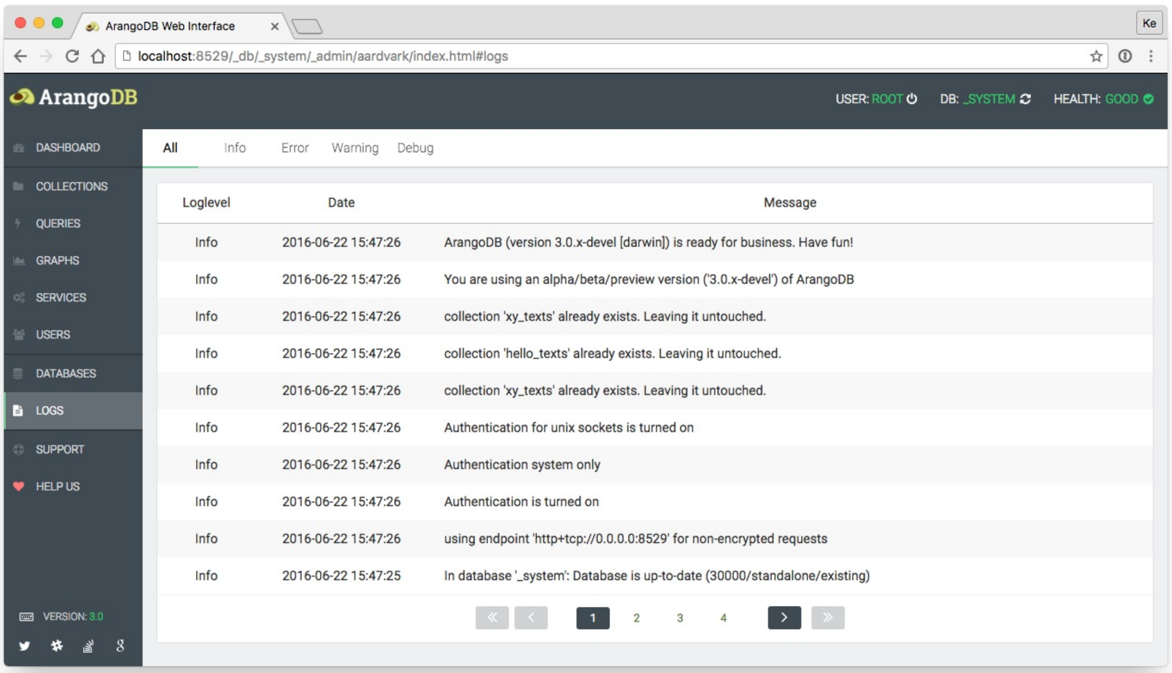
- Edit document
- Save document
- Delete docment
- Switch between Tree/Code - Mode
- Create a new document

Information:

- Displays: _id, _rev, _key properties

Logs

The logs section displays all available log entries. Log entries are filterable by their log level types.



Loglevel	Date	Message
Info	2016-06-22 15:47:26	ArangoDB (version 3.0.x-devel [darwin]) is ready for business. Have fun!
Info	2016-06-22 15:47:26	You are using an alpha/beta/preview version ('3.0.x-devel') of ArangoDB
Info	2016-06-22 15:47:26	collection 'xy_texts' already exists. Leaving it untouched.
Info	2016-06-22 15:47:26	collection 'hello_texts' already exists. Leaving it untouched.
Info	2016-06-22 15:47:26	collection 'xy_texts' already exists. Leaving it untouched.
Info	2016-06-22 15:47:26	Authentication for unix sockets is turned on
Info	2016-06-22 15:47:26	Authentication system only
Info	2016-06-22 15:47:26	Authentication is turned on
Info	2016-06-22 15:47:26	using endpoint 'http+tcp://0.0.0.0:8529' for non-encrypted requests
Info	2016-06-22 15:47:25	In database '_system': Database is up-to-date (30000/standalone/existing)

Functions:

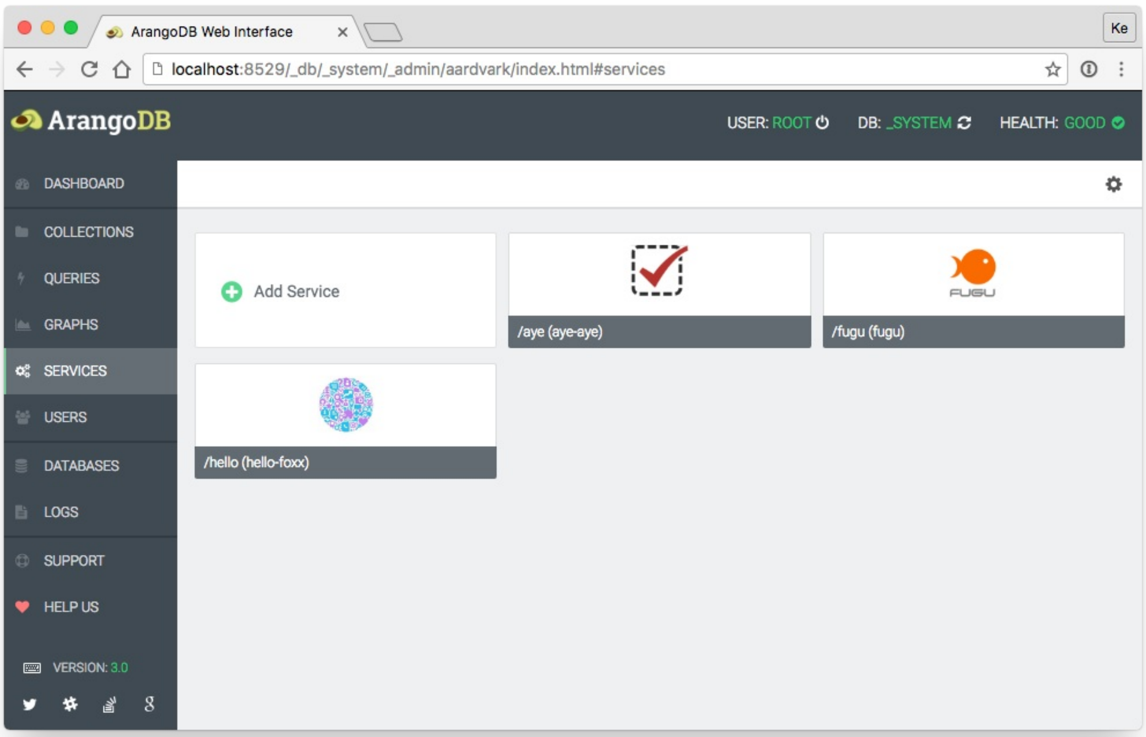
- Filter log entries by log level (all, info, error, warning, debug)

Information:

- Loglevel
- Date
- Message

Services

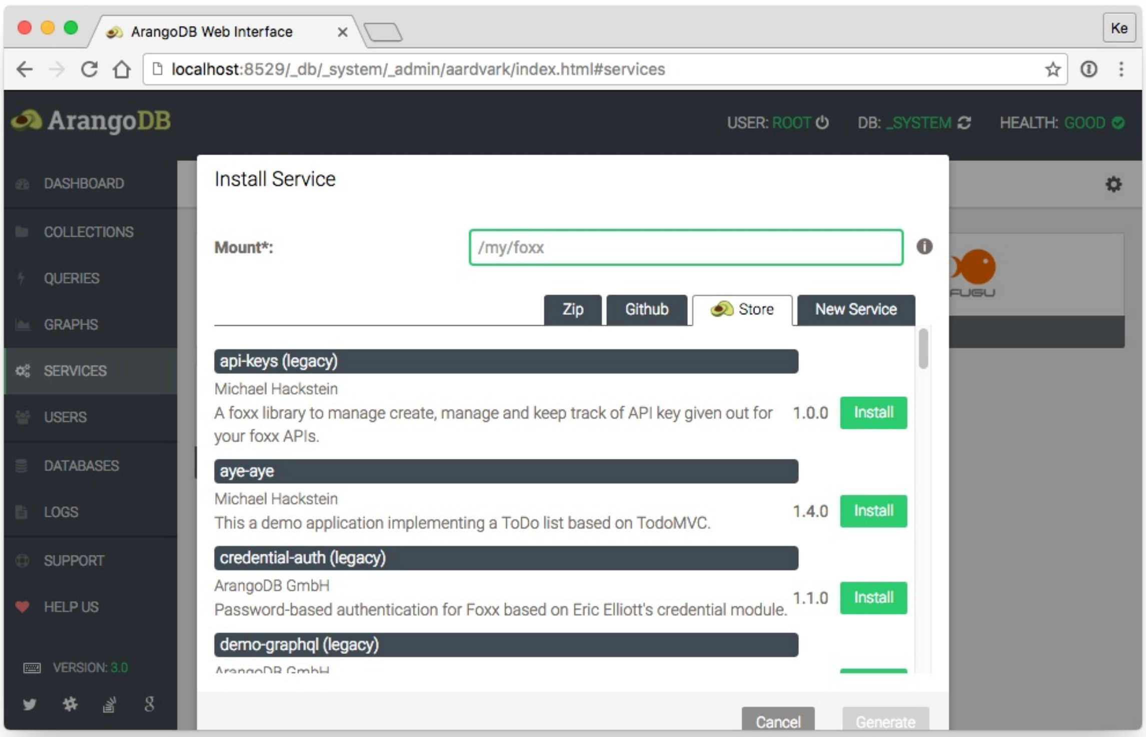
The services section displays all installed foxx applications. You can create new services or go into a detailed view of a choosen service.



Create Service

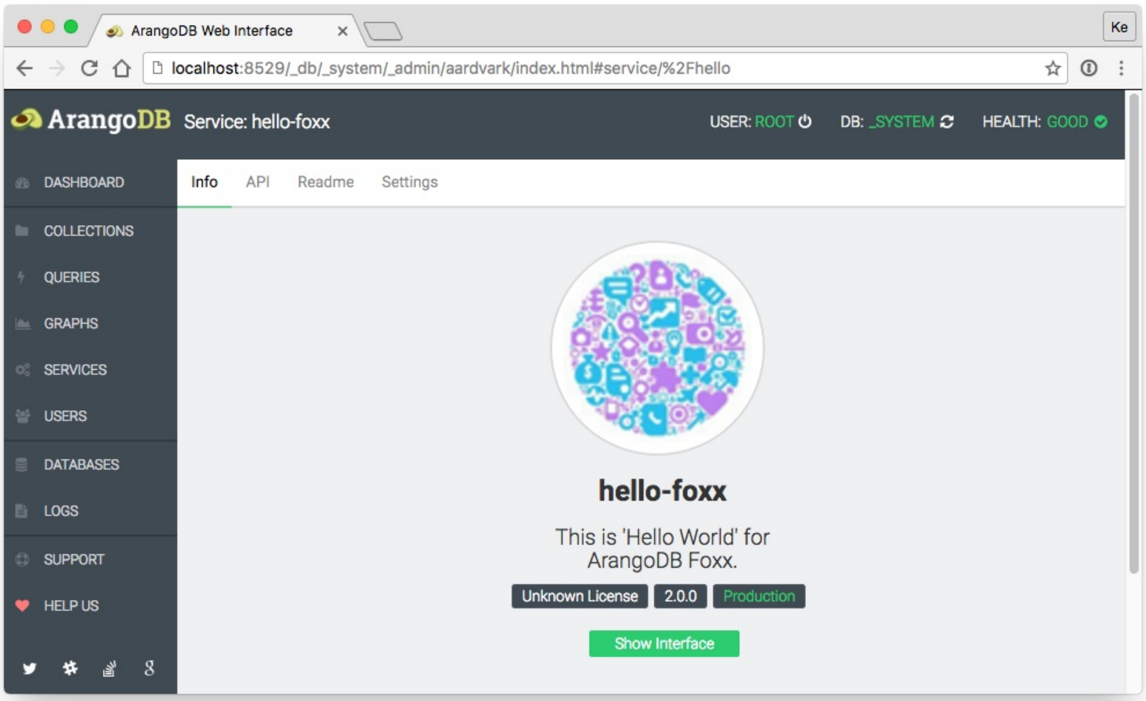
There are four different possibilities to create a new service:

1. Create service via zip file
2. Create service via github repository
3. Create service via official ArangoDB store
4. Create a blank service from scratch



Service View

This section offers several information about a specific service.



There are four view categories:

1. Info:
   - Displays name, short description, license, version, mode (production, development)
   - Offers a button to go to the services interface (if available)

2. Api:

    - Display API as SwaggerUI
    - Display API as RAW JSON

3. Readme:

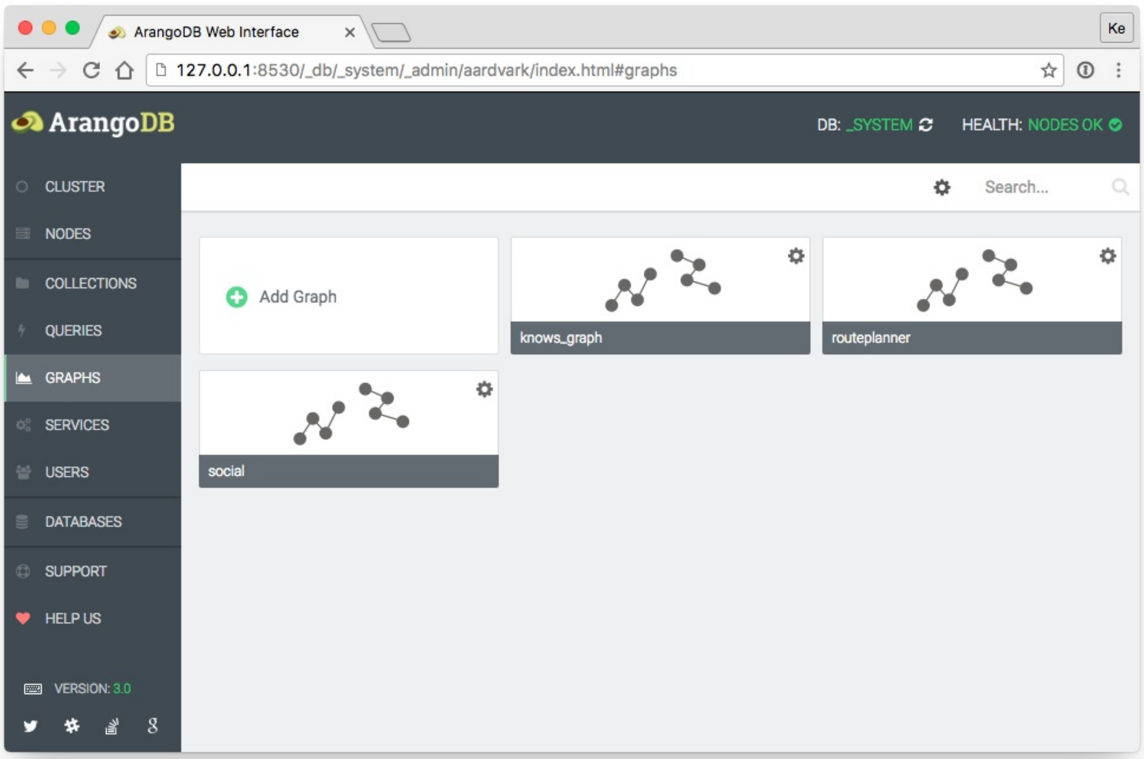
    - Displays the services manual (if available)

4. Settings:

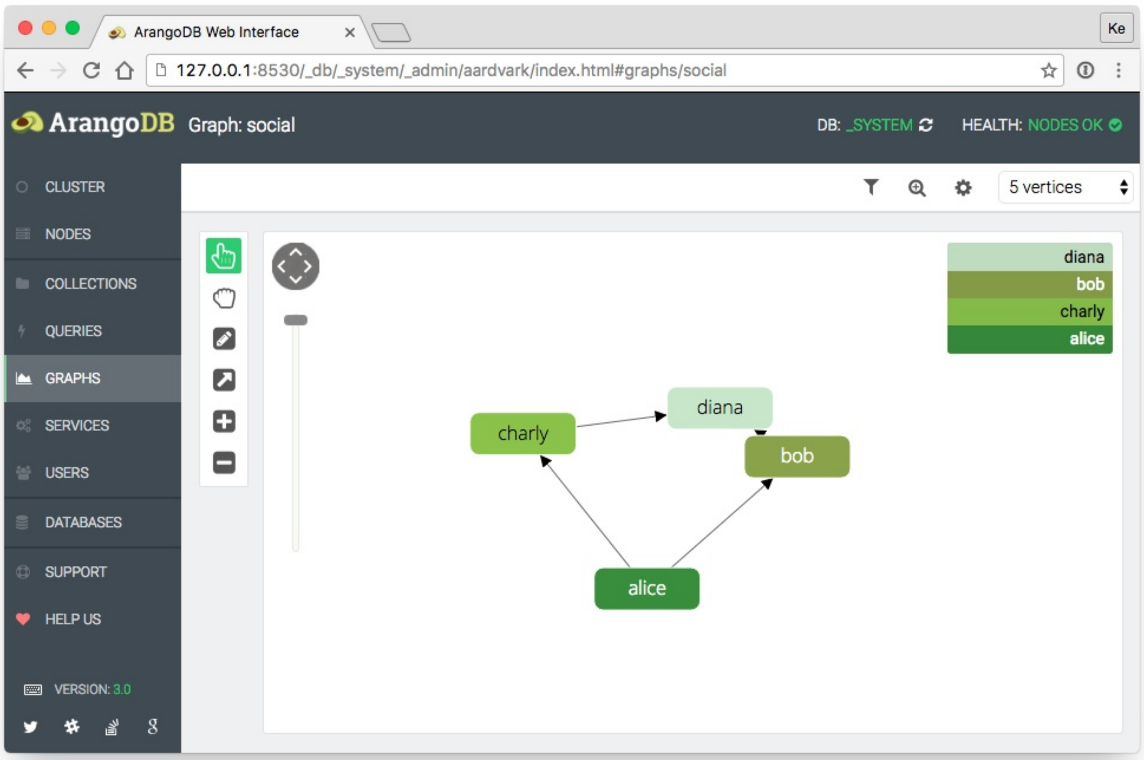
    - Download service as zip file
    - Run service tests (if available)
    - Run service scripts (if available)
    - Configure dependencies (if available)
    - Change service parameters (if available)
    - Change mode (production, development)
    - Replace the service
    - Delete the service

Graphs

The *Graphs* tab provides a viewer facility for graph data stored in ArangoDB. It allows browsing ArangoDB graphs stored in the *_graphs* system collection or a graph consisting of an arbitrary vertex and edge collection.



Please note that the graph viewer requires SVG support in your browser. Especially Internet Explorer browsers older than version 9 are likely to not support this.



Left Toolbar Functions:

- Fold/Unfold a node / drag graph
- Drag a node
- Edit a node or edge
- Drag and drop a new edge between two nodes
- Add a node
- Remove a node

Top Toolbar Functions:

- Filter graph by attribute
- Add node by attribute
- Configure labels of nodes or edges
- Group nodes by attribute
- Limit count of rendered nodes

ArangoDB Shell Introduction

The ArangoDB shell (*arangosh*) is a command-line tool that can be used for administration of ArangoDB, including running ad-hoc queries.

The *arangosh* binary is shipped with ArangoDB. It offers a JavaScript shell environment providing access to the ArangoDB server. Arangosh can be invoked like this:

```
unix> arangosh
```

By default *arangosh* will try to connect to an ArangoDB server running on server *localhost* on port *8529*. It will use the username *root* and an empty password by default. Additionally it will connect to the default database (*_system*). All these defaults can be changed using the following command-line options:

- *--server.database* : name of the database to connect to
- *--server.endpoint* : endpoint to connect to
- *--server.username* : database username
- *--server.password* : password to use when connecting
- *--server.authentication* : whether or not to use authentication

For example, to connect to an ArangoDB server on IP *192.168.173.13* on port 8530 with the user *foo* and using the database *test*, use:

```
unix> arangosh  \
  --server.endpoint tcp://192.168.173.13:8530  \
  --server.username foo  \
  --server.database test  \
  --server.authentication true
```

arangosh will then display a password prompt and try to connect to the server after the password was entered.

To change the current database after the connection has been made, you can use the `db._useDatabase()` command in arangosh:

```
arangosh> db._createDatabase("myapp");
true
arangosh> db._useDatabase("myapp");
true
arangosh> db._useDatabase("_system");
true
arangosh> db._dropDatabase("myapp");
true
```

To get a list of available commands, arangosh provides a *help()* function. Calling it will display helpful information.

arangosh also provides auto-completion. Additional information on available commands and methods is thus provided by typing the first few letters of a variable and then pressing the tab key. It is recommend to try this with entering *db.* (without pressing return) and then pressing tab.

By the way, *arangosh* provides the *db* object by default, and this object can be used for switching to a different database and managing collections inside the current database.

For a list of available methods for the *db* object, type

```
arangosh> db._help();
```

show execution results
you can paste multiple lines into arangosh, given the first line ends with an opening brace:

```
arangosh> for (var i = 0; i < 10; i ++) {
........>         require("@arangodb").print("Hello world " + i + "!\n");
........> }
```

show execution results

To load your own JavaScript code into the current JavaScript interpreter context, use the load command:

```
require("internal").load("/tmp/test.js")     // <- Linux / MacOS
require("internal").load("c:\\tmp\\test.js") // <- Windows
```

Exiting arangosh can be done using the key combination `<CTRL> + D` or by typing `quit<CR>`

Escaping

In AQL, escaping is done traditionally with the backslash character: `\`. As seen above, this leads to double backslashes when specifying Windows paths. Arangosh requires another level of escaping, also with the backslash character. It adds up to four backslashes that need to be written in Arangosh for a single literal backslash (`c:\tmp\test.js`):

```
db._query('RETURN "c:\\\\tmp\\\\test.js"')
```

You can use bind variables to mitigate this:

```
var somepath = "c:\\tmp\\test.js"
db._query(aql`RETURN ${somepath}`)
```

ArangoDB Shell Output

By default, the ArangoDB shell uses a pretty printer when JSON documents are printed. This ensures documents are printed in a human-readable way:

```
arangosh> db._create("five")
arangosh> for (i = 0; i < 5; i++) db.five.save({value:i})
arangosh> db.five.toArray()
```

show execution results
While the pretty-printer produces nice looking results, it will need a lot of screen space for each document. Sometimes, a more dense output might be better. In this case, the pretty printer can be turned off using the command *stop_pretty_print()*.

To turn on pretty printing again, use the *start_pretty_print()* command.

ArangoDB Shell Configuration

arangosh will look for a user-defined startup script named *.arangosh.rc* in the user's home directory on startup. The home directory will likely be `/home/<username>/` on Unix/Linux, and is determined on Windows by peeking into the environment variables `%HOMEDRIVE%` and `%HOMEPATH%`.

If the file *.arangosh.rc* is present in the home directory, *arangosh* will execute the contents of this file inside the global scope.

You can use this to define your own extra variables and functions that you need often. For example, you could put the following into the *.arangosh.rc* file in your home directory:

```
// "var" keyword avoided intentionally...
// otherwise "timed" would not survive the scope of this script
global.timed = function (cb) {
  console.time("callback");
  cb();
  console.timeEnd("callback");
};
```

This will make a function named *timed* available in *arangosh* in the global scope.

You can now start *arangosh* and invoke the function like this:

```
timed(function () {
  for (var i = 0; i < 1000; ++i) {
    db.test.save({ value: i });
  }
});
```

Please keep in mind that, if present, the *.arangosh.rc* file needs to contain valid JavaScript code. If you want any variables in the global scope to survive you need to omit the *var* keyword for them. Otherwise the variables will only be visible inside the script itself, but not outside.

Details about the ArangoDB Shell

After the server has been started, you can use the ArangoDB shell (*arangosh*) to administrate the server. Without any arguments, the ArangoDB shell will try to contact the server on port 8529 on the localhost. For more information see the ArangoDB Shell documentation. You might need to set additional options (endpoint, username and password) when connecting:

```
unix> ./arangosh --server.endpoint tcp://127.0.0.1:8529 --server.username root
```

The shell will print its own version number and if successfully connected to a server the version number of the ArangoDB server.

Command-Line Options

Use `--help` to get a list of command-line options:

```
unix> ./arangosh --help
STANDARD options:
  --audit-log <string>          audit log file to save commands and results to
  --configuration <string>      read configuration file
  --help                        help message
  --max-upload-size <uint64>    maximum size of import chunks (in bytes) (default: 500000)
  --no-auto-complete            disable auto completion
  --no-colors                   deactivate color support
  --pager <string>              output pager (default: "less -X -R -F -L")
  --pretty-print                pretty print values
  --quiet                       no banner
  --temp.path <string>          path for temporary files (default: "/tmp/arangodb")
  --use-pager                   use pager

JAVASCRIPT options:
  --javascript.check <string>                syntax check code JavaScript code from file
  --javascript.execute <string>              execute JavaScript code from file
  --javascript.execute-string <string>       execute JavaScript code from string
  --javascript.startup-directory <string>    startup paths containing the JavaScript files
  --javascript.unit-tests <string>           do not start as shell, run unit tests instead
  --jslint <string>                          do not start as shell, run jslint instead

LOGGING options:
  --log.level <string>    log level (default: "info")

CLIENT options:
  --server.connect-timeout <double>         connect timeout in seconds (default: 3)
  --server.authentication <bool>            whether or not to use authentication (default: true)
  --server.endpoint <string>                endpoint to connect to, use 'none' to start without a server (default: "tcp:
  --server.password <string>                password to use when connecting (leave empty for prompt)
  --server.request-timeout <double>         request timeout in seconds (default: 300)
  --server.username <string>                username to use when connecting (default: "root")
```

Database Wrappers

The `db` object is available in *arangosh* as well as on *arangod* i.e. if you're using Foxx. While its interface is persistant between the *arangosh* and the *arangod* implementations, its underpinning is not. The *arangod* implementation are JavaScript wrappers around ArangoDB's native C++ implementation, whereas the *arangosh* implementation wraps HTTP accesses to ArangoDB's RESTfull API.

So while this code may produce similar results when executed in *arangosh* and *arangod*, the cpu usage and time required will be really different:

```
for (i = 0; i < 100000; i++) {
    db.test.save({ name: { first: "Jan" }, count: i});
}
```

Since the *arangosh* version will be doing around 100k HTTP requests, and the *arangod* version will directly write to the database.

Using `arangosh` via unix shebang mechanisms

In unix operating systems you can start scripts by specifying the interpreter in the first line of the script. This is commonly called `shebang` or `hash bang`. You can also do that with `arangosh`, i.e. create `~/test.js`:

```
#!/usr/bin/arangosh --javascript.execute 
require("internal").print("hello world")
db._query("FOR x IN test RETURN x").toArray()
```

Note that the first line has to end with a blank in order to make it work. Mark it executable to the OS:

```
#> chmod a+x ~/test.js
```

and finaly try it out:

```
#> ~/test.js
```

Arangoimp

This manual describes the ArangoDB importer *arangoimp*, which can be used for bulk imports.

The most convenient method to import a lot of data into ArangoDB is to use the *arangoimp* command-line tool. It allows you to import data records from a file into an existing database collection.

It is possible to import document keys with the documents using the *_key* attribute. When importing into an edge collection, it is mandatory that all imported documents have the *_from* and *_to* attributes, and that they contain valid references.

Let's assume for the following examples you want to import user data into an existing collection named "users" on the server.

Importing Data into an ArangoDB Database

Importing JSON-encoded Data

Let's further assume the import at hand is encoded in JSON. We'll be using these example user records to import:

```
{ "name" : { "first" : "John", "last" : "Connor" }, "active" : true, "age" : 25, "likes" : [ "swimming"] }
{ "name" : { "first" : "Jim", "last" : "O'Brady" }, "age" : 19, "likes" : [ "hiking", "singing" ] }
{ "name" : { "first" : "Lisa", "last" : "Jones" }, "dob" : "1981-04-09", "likes" : [ "running" ] }
```

To import these records, all you need to do is to put them into a file (with one line for each record to import) and run the following command:

```
> arangoimp --file "data.json" --type json --collection "users"
```

This will transfer the data to the server, import the records, and print a status summary. To show the intermediate progress during the import process, the option *--progress* can be added. This option will show the percentage of the input file that has been sent to the server. This will only be useful for big import files.

```
> arangoimp --file "data.json" --type json --collection users --progress true
```

It is also possible to use the output of another command as an input for arangoimp. For example, the following shell command can be used to pipe data from the `cat` process to arangoimp:

```
> cat data.json | arangoimp --file - --type json --collection users
```

Note that you have to use `--file -` if you want to use another command as input for arangoimp. No progress can be reported for such imports as the size of the input will be unknown to arangoimp.

By default, the endpoint *tcp://127.0.0.1:8529* will be used. If you want to specify a different endpoint, you can use the *--server.endpoint* option. You probably want to specify a database user and password as well. You can do so by using the options *--server.username* and *--server.password*. If you do not specify a password, you will be prompted for one.

```
> arangoimp --server.endpoint tcp://127.0.0.1:8529 --server.username root --file "data.json" --type json --collection "u
```

Note that the collection (*users* in this case) must already exist or the import will fail. If you want to create a new collection with the import data, you need to specify the *--create-collection* option. Note that by default it will create a document collection and no ede collection.

```
> arangoimp --file "data.json" --type json --collection "users" --create-collection true
```

To create an edge collection instead, use the *--create-collection-type* option and set it to *edge*:

```
> arangoimp --file "data.json" --collection "myedges" --create-collection true --create-collection-type edge
```

When importing data into an existing collection it is often convenient to first remove all data from the collection and then start the import. This can be achieved by passing the *--overwrite* parameter to *arangoimp*. If it is set to *true*, any existing data in the collection will be removed prior to the import. Note that any existing index definitions for the collection will be preserved even if *--overwrite* is set to true.

```
> arangoimp --file "data.json" --type json --collection "users" --overwrite true
```

As the import file already contains the data in JSON format, attribute names and data types are fully preserved. As can be seen in the example data, there is no need for all data records to have the same attribute names or types. Records can be inhomogeneous.

Please note that by default, *arangoimp* will import data into the specified collection in the default database (*_system*). To specify a different database, use the *--server.database* option when invoking *arangoimp*.

JSON input file formats

Note: *arangoimp* supports two formats when importing JSON data from a file. The first format (also used above) requires the input file to contain one complete JSON document in each line, e.g.

```
{ "_key": "one", "value": 1 }
{ "_key": "two", "value": 2 }
{ "_key": "foo", "value": "bar" }
...
```

The above format can be imported sequentially by *arangoimp*. It will read data from the input file in chunks and send it in batches to the server. Each batch will be about as big as specified in the command-line parameter *--batch-size*.

An alternative is to put one big JSON document into the input file like this:

```
[
  { "_key": "one", "value": 1 },
  { "_key": "two", "value": 2 },
  { "_key": "foo", "value": "bar" },
  ...
]
```

This format allows line breaks within the input file as required. The downside is that the whole input file will need to be read by *arangoimp* before it can send the first batch. This might be a problem if the input file is big. By default, *arangoimp* will allow importing such files up to a size of about 16 MB.

If you want to allow your *arangoimp* instance to use more memory, you may want to increase the maximum file size by specifying the command-line option *--batch-size*. For example, to set the batch size to 32 MB, use the following command:

```
> arangoimp --file "data.json" --type json --collection "users" --batch-size 33554432
```

Please also note that you may need to increase the value of *--batch-size* if a single document inside the input file is bigger than the value of *--batch-size*.

Importing CSV Data

arangoimp also offers the possibility to import data from CSV files. This comes handy when the data at hand is in CSV format already and you don't want to spend time converting them to JSON for the import.

To import data from a CSV file, make sure your file contains the attribute names in the first row. All the following lines in the file will be interpreted as data records and will be imported.

The CSV import requires the data to have a homogeneous structure. All records must have exactly the same amount of columns as there are headers.

The cell values can have different data types though. If a cell does not have any value, it can be left empty in the file. These values will not be imported so the attributes will not "be there" in document created. Values enclosed in quotes will be imported as strings, so to import numeric values, boolean values or the null value, don't enclose the value in quotes in your file.

We'll be using the following import for the CSV import:

```
"first","last","age","active","dob"
"John","Connor",25,true,
"Jim","O'Brady",19,,
"Lisa","Jones",,,"1981-04-09"
Hans,dos Santos,0123,,
Wayne,Brewer,,false,
```

The command line to execute the import is:

```
> arangoimp --file "data.csv" --type csv --collection "users"
```

The above data will be imported into 5 documents which will look as follows:

```
{ "first" : "John", "last" : "Connor", "active" : true, "age" : 25 }
{ "first" : "Jim", "last" : "O'Brady", "age" : 19 }
{ "first" : "Lisa", "last" : "Jones", "dob" : "1981-04-09" }
{ "first" : "Hans", "last" : "dos Santos", "age" : 123 }
{ "first" : "Wayne", "last" : "Brewer", "active" : false }
```

As can be seen, values left completely empty in the input file will be treated as absent. Numeric values not enclosed in quotes will be treated as numbers. Note that leading zeros in numeric values will be removed. To import numbers with leading zeros, please use strings. The literals *true* and *false* will be treated as booleans if they are not enclosed in quotes. Other values not enclosed in quotes will be treated as strings. Any values enclosed in quotes will be treated as strings, too.

String values containing the quote character or the separator must be enclosed with quote characters. Within a string, the quote character itself must be escaped with another quote character (or with a backslash if the *--backslash-escape* option is used).

Note that the quote and separator characters can be adjusted via the *--quote* and *--separator* arguments when invoking *arangoimp*. The quote character defaults to the double quote (*"*). To use a literal quote in a string, you can use two quote characters. To use backslash for escaping quote characters, please set the option *--backslash-escape* to *true*.

The importer supports Windows (CRLF) and Unix (LF) line breaks. Line breaks might also occur inside values that are enclosed with the quote character.

Here's an example for using literal quotes and newlines inside values:

```
"name","password"
"Foo","r4ndom""123!"
"Bar","wow!
this is a
multine password!"
"Bartholomew ""Bart"" Simpson","Milhouse"
```

Extra whitespace at the end of each line will be ignored. Whitespace at the start of lines or between field values will not be ignored, so please make sure that there is no extra whitespace in front of values or between them.

Importing TSV Data

You may also import tab-separated values (TSV) from a file. This format is very simple: every line in the file represents a data record. There is no quoting or escaping. That also means that the separator character (which defaults to the tabstop symbol) must not be used anywhere in the actual data.

As with CSV, the first line in the TSV file must contain the attribute names, and all lines must have an identical number of values.

If a different separator character or string should be used, it can be specified with the *--separator* argument.

An example command line to execute the TSV import is:

```
> arangoimp --file "data.tsv" --type tsv --collection "users"
```

Importing into an Edge Collection

arangoimp can also be used to import data into an existing edge collection. The import data must, for each edge to import, contain at least the *_from* and *_to* attributes. These indicate which other two documents the edge should connect. It is necessary that these attributes are set for all records, and point to valid document ids in existing collections.

Examples

```
{ "_from" : "users/1234", "_to" : "users/4321", "desc" : "1234 is connected to 4321" }
```

Note: The edge collection must already exist when the import is started. Using the *--create-collection* flag will not work because arangoimp will always try to create a regular document collection if the target collection does not exist.

Updating existing documents

By default, arangoimp will try to insert all documents from the import file into the specified collection. In case the import file contains documents that are already present in the target collection (matching is done via the *_key* attributes), then a default arangoimp run will not import these documents and complain about unique key constraint violations.

However, arangoimp can be used to update or replace existing documents in case they already exist in the target collection. It provides the command-line option *--on-duplicate* to control the behavior in case a document is already present in the database.

The default value of *--on-duplicate* is *error*. This means that when the import file contains a document that is present in the target collection already, then trying to re-insert a document with the same *_key* value is considered an error, and the document in the database will not be modified.

Other possible values for *--on-duplicate* are:

- *update*: each document present in the import file that is also present in the target collection already will be updated by arangoimp. *update* will perform a partial update of the existing document, modifying only the attributes that are present in the import file and leaving all other attributes untouched.

  The values of system attributes *_id*, *_key*, *_rev*, *_from* and *_to* cannot be updated or replaced in existing documents.

- *replace*: each document present in the import file that is also present in the target collection already will be replace by arangoimp. *replace* will replace the existing document entirely, resulting in a document with only the attributes specified in the import file.

  The values of system attributes *_id*, *_key*, *_rev*, *_from* and *_to* cannot be updated or replaced in existing documents.

- *ignore*: each document present in the import file that is also present in the target collection already will be ignored and not modified in the target collection.

When *--on-duplicate* is set to either *update* or *replace*, arangoimp will return the number of documents updated/replaced in the *updated* return value. When set to another value, the value of *updated* will always be zero. When *--on-duplicate* is set to *ignore*, arangoimp will return the number of ignored documents in the *ignored* return value. When set to another value, *ignored* will always be zero.

It is possible to perform a combination of inserts and updates/replaces with a single arangoimp run. When *--on-duplicate* is set to *update* or *replace*, all documents present in the import file will be inserted into the target collection provided they are valid and do not already exist with the specified *_key*. Documents that are already present in the target collection (identified by *_key* attribute) will instead be updated/replaced.

Arangoimp result output

An *arangoimp* import run will print out the final results on the command line. It will show the

- number of documents created (*created*)
- number of documents updated/replaced (*updated/replaced*, only non-zero if *--on-duplicate* was set to *update* or *replace*, see below)
- number of warnings or errors that occurred on the server side (*warnings/errors*)
- number of ignored documents (only non-zero if *--on-duplicate* was set to *ignore*).

Example

```
created:          2
warnings/errors:  0
updated/replaced: 0
ignored:          0
```

For CSV and TSV imports, the total number of input file lines read will also be printed (*lines read*).

arangoimp will also print out details about warnings and errors that happened on the server-side (if any).

Attribute Naming and Special Attributes

Attributes whose names start with an underscore are treated in a special way by ArangoDB:

- the optional *_key* attribute contains the document's key. If specified, the value must be formally valid (e.g. must be a string and conform to the naming conventions). Additionally, the key value must be unique within the collection the import is run for.
- *_from*: when importing into an edge collection, this attribute contains the id of one of the documents connected by the edge. The value of *_from* must be a syntactically valid document id and the referred collection must exist.
- *_to*: when importing into an edge collection, this attribute contains the id of the other document connected by the edge. The value of *_to* must be a syntactically valid document id and the referred collection must exist.
- *_rev*: this attribute contains the revision number of a document. However, the revision numbers are managed by ArangoDB and cannot be specified on import. Thus any value in this attribute is ignored on import.

If you import values into *_key*, you should make sure they are valid and unique.

When importing data into an edge collection, you should make sure that all import documents can *_from* and *_to* and that their values point to existing documents.

To avoid specifying complete document ids (consisting of collection names and document keys) for *_from* and *_to* values, there are the options *--from-collection-prefix* and *--to-collection-prefix*. If specified, these values will be automatically prepended to each value in *_from* (or *_to* resp.). This allows specifying only document keys inside *_from* and/or *_to*.

Example

```
> arangoimp --from-collection-prefix users --to-collection-prefix products ...
```

Importing the following document will then create an edge between *users/1234* and *products/4321*:

```
{ "_from" : "1234", "_to" : "4321", "desc" : "users/1234 is connected to products/4321" }
```

Dumping Data from an ArangoDB database

To dump data from an ArangoDB server instance, you will need to invoke *arangodump*. Dumps can be re-imported with *arangorestore*. *arangodump* can be invoked by executing the following command:

```
unix> arangodump --output-directory "dump"
```

This will connect to an ArangoDB server and dump all non-system collections from the default database (*_system*) into an output directory named *dump*. Invoking *arangodump* will fail if the output directory already exists. This is an intentional security measure to prevent you from accidentally overwriting already dumped data. If you are positive that you want to overwrite data in the output directory, you can use the parameter *--overwrite true* to confirm this:

```
unix> arangodump --output-directory "dump" --overwrite true
```

arangodump will by default connect to the *_system* database using the default endpoint. If you want to connect to a different database or a different endpoint, or use authentication, you can use the following command-line options:

- *--server.database* : name of the database to connect to
- *--server.endpoint* : endpoint to connect to
- *--server.username* : username
- *--server.password* : password to use (omit this and you'll be prompted for the password)
- *--server.authentication* : whether or not to use authentication

Here's an example of dumping data from a non-standard endpoint, using a dedicated database name:

```
unix> arangodump --server.endpoint tcp://192.168.173.13:8531 --server.username backup --server.database mydb --output-di
```

When finished, *arangodump* will print out a summary line with some aggregate statistics about what it did, e.g.:

```
Processed 43 collection(s), wrote 408173500 byte(s) into datafiles, sent 88 batch(es)
```

By default, *arangodump* will dump both structural information and documents from all non-system collections. To adjust this, there are the following command-line arguments:

- *--dump-data* : set to *true* to include documents in the dump. Set to *false* to exclude documents. The default value is *true*.
- *--include-system-collections* : whether or not to include system collections in the dump. The default value is *false*.

For example, to only dump structural information of all collections (including system collections), use:

```
unix> arangodump --dump-data false --include-system-collections true --output-directory "dump"
```

To restrict the dump to just specific collections, there is is the *--collection* option. It can be specified multiple times if required:

```
unix> arangodump --collection myusers --collection myvalues --output-directory "dump"
```

Structural information for a collection will be saved in files with name pattern *.structure.json*. Each structure file will contains a JSON object with these attributes:

- *parameters*: contains the collection properties
- *indexes*: contains the collection indexes

Document data for a collection will be saved in files with name pattern *.data.json*. Each line in a data file is a document insertion/update or deletion marker, alongside with some meta data.

Starting with Version 2.1 of ArangoDB, the *arangodump* tool also supports sharding. Simply point it to one of the coordinators and it will behave exactly as described above, working on sharded collections in the cluster.

However, as opposed to the single instance situation, this operation does not guarantee to dump a consistent snapshot if write operations happen during the dump operation. It is therefore recommended not to perform any data-modifcation operations on the cluster whilst *arangodump* is running.

As above, the output will be one structure description file and one data file per sharded collection. Note that the data in the data file is sorted first by shards and within each shard by ascending timestamp. The structural information of the collection contains the number of shards and the shard keys.

Note that the version of the arangodump client tool needs to match the version of the ArangoDB server it connects to. By default, arangodump will produce a dump that can be restored with the arangorestore tool of the same version. An exception is arangodump in 3.0, which supports dumping data in a format compatible with ArangoDB 2.8. In order to produce a 2.8-compatible dump with a 3.0 ArangoDB, please specify the option `--compat28 true` when invoking arangodump.

```
unix> arangodump --compat28 true --collection myvalues --output-directory "dump"
```

Arangorestore

To reload data from a dump previously created with arangodump, ArangoDB provides the *arangorestore* tool.

Reloading Data into an ArangoDB database

Invoking arangorestore

arangorestore can be invoked from the command-line as follows:

```
unix> arangorestore --input-directory "dump"
```

This will connect to an ArangoDB server and reload structural information and documents found in the input directory *dump*. Please note that the input directory must have been created by running *arangodump* before.

arangorestore will by default connect to the *_system* database using the default endpoint. If you want to connect to a different database or a different endpoint, or use authentication, you can use the following command-line options:

- --*server.database* : name of the database to connect to
- --*server.endpoint* : endpoint to connect to
- --*server.username* : username
- --*server.password* : password to use (omit this and you'll be prompted for the password)
- --*server.authentication* : whether or not to use authentication

Since version 2.6 *arangorestore* provides the option --*create-database*. Setting this option to *true* will create the target database if it does not exist. When creating the target database, the username and passwords passed to *arangorestore* (in options --*server.username* and --*server.password*) will be used to create an initial user for the new database.

Here's an example of reloading data to a non-standard endpoint, using a dedicated database name:

```
unix> arangorestore --server.endpoint tcp://192.168.173.13:8531 --server.username backup --server.database mydb --input-
```

To create the target database whe restoring, use a command like this:

```
unix> arangorestore --server.username backup --server.database newdb --create-database true --input-directory "dump"
```

arangorestore will print out its progress while running, and will end with a line showing some aggregate statistics:

```
Processed 2 collection(s), read 2256 byte(s) from datafiles, sent 2 batch(es)
```

By default, *arangorestore* will re-create all non-system collections found in the input directory and load data into them. If the target database already contains collections which are also present in the input directory, the existing collections in the database will be dropped and re-created with the data found in the input directory.

The following parameters are available to adjust this behavior:

- --*create-collection* : set to *true* to create collections in the target database. If the target database already contains a collection with the same name, it will be dropped first and then re-created with the properties found in the input directory. Set to *false* to keep existing collections in the target database. If set to *false* and *arangorestore* encounters a collection that is present in both the target database and the input directory, it will abort. The default value is *true*.
- --*import-data* : set to *true* to load document data into the collections in the target database. Set to *false* to not load any document data. The default value is *true*.
- --*include-system-collections* : whether or not to include system collections when re-creating collections or reloading data. The default value is *false*.

For example, to (re-)create all non-system collections and load document data into them, use:

```
unix> arangorestore --create-collection true --import-data true --input-directory "dump"
```

This will drop potentially existing collections in the target database that are also present in the input directory.

To include system collections too, use *--include-system-collections true*:

```
unix> arangorestore --create-collection true --import-data true --include-system-collections true --input-directory "dum
```

To (re-)create all non-system collections without loading document data, use:

```
unix> arangorestore --create-collection true --import-data false --input-directory "dump"
```

This will also drop existing collections in the target database that are also present in the input directory.

To just load document data into all non-system collections, use:

```
unix> arangorestore --create-collection false --import-data true --input-directory "dump"
```

To restrict reloading to just specific collections, there is is the *--collection* option. It can be specified multiple times if required:

```
unix> arangorestore --collection myusers --collection myvalues --input-directory "dump"
```

Collections will be processed by in alphabetical order by *arangorestore*, with all document collections being processed before all edge collections. This is to ensure that reloading data into edge collections will have the document collections linked in edges (*_from* and *_to* attributes) loaded.

Restoring Revision Ids and Collection Ids

arangorestore will reload document and edges data with the exact same *_key*, *_from* and *_to* values found in the input directory. However, when loading document data, it will assign its own values for the *_rev* attribute of the reloaded documents. Though this difference is intentional (normally, every server should create its own *_rev* values) there might be situations when it is required to re-use the exact same *_rev* values for the reloaded data. This can be achieved by setting the *--recycle-ids* parameter to *true*:

```
unix> arangorestore --collection myusers --collection myvalues --recycle-ids true --input-directory "dump"
```

Note that setting *--recycle-ids* to *true* will also cause collections to be (re-)created in the target database with the exact same collection id as in the input directory. Any potentially existing collection in the target database with the same collection id will then be dropped.

Setting *--recycle-ids* to *false* or omitting it will only use the collection name from the input directory and allow the target database to create the collection with a different id (though with the same name) than in the input directory.

Reloading Data into a different Collection

With some creativity you can use *arangodump* and *arangorestore* to transfer data from one collection into another (either on the same server or not). For example, to copy data from a collection *myvalues* in database *mydb* into a collection *mycopyvalues* in database *mycopy*, you can start with the following command:

```
unix> arangodump --collection myvalues --server.database mydb --output-directory "dump"
```

This will create two files, *myvalues.structure.json* and *myvalues.data.json*, in the output directory. To load data from the datafile into an existing collection *mycopyvalues* in database *mycopy*, rename the files to *mycopyvalues.structure.json* and *mycopyvalues.data.json*. After that, run the following command:

```
unix> arangorestore --collection mycopyvalues --server.database mycopy --input-directory "dump"
```

Using arangorestore with sharding

As of Version 2.1 the *arangorestore* tool supports sharding. Simply point it to one of the coordinators in your cluster and it will work as usual but on sharded collections in the cluster.

If *arangorestore* is asked to drop and re-create a collection, it will use the same number of shards and the same shard keys as when the collection was dumped. The distribution of the shards to the servers will also be the same as at the time of the dump. This means in particular that DBservers with the same IDs as before must be present in the cluster at time of the restore.

If a collection was dumped from a single instance, one can manually add the structural description for the shard keys and the number and distribution of the shards and then the restore into a cluster will work.

If you restore a collection that was dumped from a cluster into a single ArangoDB instance, the number of shards and the shard keys will silently be ignored.

Note that in a cluster, every newly created collection will have a new ID, it is not possible to reuse the ID from the originally dumped collection. This is for safety reasons to ensure consistency of IDs.

Managing Users

The user management in ArangoDB 3 is similar to the one found in MySQL, Postgres, or other database systems.

An ArangoDB server contains a list of users. Each user can have access to one or more databases (or none for that matter).

In order to manage users use the web interface. Log into the *_system* database and go to the "User" section.

Using the ArangoDB shell

Alternatively, you can use the ArangoDB shell. Fire up *arangosh* and require the users module.

```
arangosh> var users = require("@arangodb/users");
arangosh> users.save("admin@testapp", "mypassword");
```

Creates an user call *admin@testapp*. This user will have no access at all.

```
arangosh> users.grantDatabase("admin@testapp", "testdb");
```

This grants the user access to the database *testdb*. `revokeDatabase` will revoke the right.

Save

`users.save(user, passwd, active, extra)`

This will create a new ArangoDB user. The username must be specified in *user* and must not be empty.

The password must be given as a string, too, but can be left empty if required. If you pass the special value *ARANGODB_DEFAULT_ROOT_PASSWORD*, the password will be set the value stored in the environment variable `ARANGODB_DEFAULT_ROOT_PASSWORD`. This can be used to pass an instance variable into ArangoDB. For example, the instance identifier from Amazon.

If the *active* attribute is not specified, it defaults to *true*. The *extra* attribute can be used to save custom data with the user.

This method will fail if either the username or the passwords are not specified or given in a wrong format, or there already exists a user with the specified name.

Note: the user will not have permission to access any database. You need to grant the access rights for one or more databases using grantDatabase.

Examples

```
arangosh> require("@arangodb/users").save("my-user", "my-secret-password");
```

show execution results

Grant Database

`users.grantDatabase(user, database)`

This grants read/write access to the *database* for the *user*.

If a user has access rights to the *_system* database, he is considered superuser.

Revoke Database

`users.revokeDatabase(user, database)`

This revokes read/write access to the *database* for the *user*.

Replace

`users.replace(user, passwd, active, extra)`

This will look up an existing ArangoDB user and replace its user data.

The username must be specified in *user*, and a user with the specified name must already exist in the database.

The password must be given as a string, too, but can be left empty if required.

If the *active* attribute is not specified, it defaults to *true*. The *extra* attribute can be used to save custom data with the user.

This method will fail if either the username or the passwords are not specified or given in a wrong format, or if the specified user cannot be found in the database.

Note: this function will not work from within the web interface

Examples

```
arangosh> require("@arangodb/users").replace("my-user", "my-changed-password");
```

show execution results

Update

`users.update(user, passwd, active, extra)`

This will update an existing ArangoDB user with a new password and other data.

The username must be specified in *user* and the user must already exist in the database.

The password must be given as a string, too, but can be left empty if required.

If the *active* attribute is not specified, the current value saved for the user will not be changed. The same is true for the *extra* attribute.

This method will fail if either the username or the passwords are not specified or given in a wrong format, or if the specified user cannot be found in the database.

Examples

```
arangosh> require("@arangodb/users").update("my-user", "my-secret-password");
```

show execution results

isValid

`users.isValid(user, password)`

Checks whether the given combination of username and password is valid. The function will return a boolean value if the combination of username and password is valid.

Each call to this function is penalized by the server sleeping a random amount of time.

Examples

```
arangosh> require("@arangodb/users").isValid("my-user", "my-secret-password");
true
```

Remove

`users.remove(user)`

Removes an existing ArangoDB user from the database.

The username must be specified in *User* and the specified user must exist in the database.

This method will fail if the user cannot be found in the database.

Examples

```
arangosh> require("@arangodb/users").remove("my-user");
```

Document

`users.document(user)`

Fetches an existing ArangoDB user from the database.

The username must be specified in *user*.

This method will fail if the user cannot be found in the database.

Examples

```
arangosh> require("@arangodb/users").document("my-user");
```

show execution results

all()

`users.all()`

Fetches all existing ArangoDB users from the database.

Examples

```
arangosh> require("@arangodb/users").all();
```

show execution results

Reload

`users.reload()`

Reloads the user authentication data on the server

All user authentication data is loaded by the server once on startup only and is cached after that. When users get added or deleted, a cache flush is done automatically, and this can be performed by called this method.

Examples

```
arangosh> require("@arangodb/users").reload();
```

Comparison to ArangoDB 2

ArangoDB 2 contained separate users per database. It was not possible to give an user access to two or more databases. This proved impractical. Therefore we switch to a more common user model in ArangoDB 3.

Command-Line Options for the Authentication and Authorization

`--server.authentication` Setting this option to *false* will turn off authentication on the server side so all clients can execute any action without authorization and privilege checks. The default value is *true*.

`--server.authentication-system-only boolean` Controls whether incoming requests need authentication only if they are directed to the ArangoDB's internal APIs and features, located at */_api/*, */_admin/* etc. If the flag is set to *true*, then HTTP authentication is only required for requests going to URLs starting with /_, but not for other URLs. The flag can thus be used to expose a user-made API without HTTP authentication to the outside world, but to prevent the outside world from using the ArangoDB API and the admin interface without authentication. Note that checking the URL is performed after any database name prefix has been removed. That means when the actual URL called is */_db/_system/myapp/myaction*, the URL */myapp/myaction* will be used for *authentication-system-only* check. The default is *true*. Note that authentication still needs to be enabled for the server regularly in order for HTTP authentication to be forced for the ArangoDB API and the web interface. Setting only this flag is not enough. You can control ArangoDB's general authentication feature with the --*server.authentication* flag.

Command-line options

General Options

General help

`--help`

`-h`

Prints a list of the most common options available and then exits. In order to see all options use *--help-all*.

Version

`--version`

`-v`

Prints the version of the server and exits.

Configuration Files

Options can be specified on the command line or in configuration files. If a string *Variable* occurs in the value, it is replaced by the corresponding environment variable.

`--configuration filename`

`-c filename`

Specifies the name of the configuration file to use.

If this command is not passed to the server, then by default, the server will attempt to first locate a file named *~/.arango/arangod.conf* in the user's home directory.

If no such file is found, the server will proceed to look for a file *arangod.conf* in the system configuration directory. The system configuration directory is platform-specific, and may be changed when compiling ArangoDB yourself. It may default to */etc/arangodb* or */usr/local/etc/arangodb*. This file is installed when using a package manager like rpm or dpkg. If you modify this file and later upgrade to a new version of ArangoDB, then the package manager normally warns you about the conflict. In order to avoid these warning for small adjustments, you can put local overrides into a file *arangod.conf.local*.

Only command line options with a value should be set within the configuration file. Command line options which act as flags should be entered on the command line when starting the server.

Whitespace in the configuration file is ignored. Each option is specified on a separate line in the form

```
key = value
```

Alternatively, a header section can be specified and options pertaining to that section can be specified in a shorter form

```
[log]
level = trace
```

rather than specifying

```
log.level = trace
```

So you see in general `--section.param value` translates to

```
[section]
param=value
```

Where one section may occur multiple times, and the last occurance of `param` will become the final value. In case of parameters being vectors, multiple occurance adds another item to the vector. Vectors can be identified by the `...` in the `--help` output of the binaries.

Comments can be placed in the configuration file, only if the line begins with one or more hash symbols (#).

There may be occasions where a configuration file exists and the user wishes to override configuration settings stored in a configuration file. Any settings specified on the command line will overwrite the same setting when it appears in a configuration file. If the user wishes to completely ignore configuration files without necessarily deleting the file (or files), then add the command line option

```
-c none
```

or

```
--configuration none
```

When starting up the server. Note that, the word *none* is case-insensitive.

Managing Endpoints

The ArangoDB server can listen for incoming requests on multiple *endpoints*.

The endpoints are normally specified either in ArangoDB's configuration file or on the command-line, using the `--server.endpoint`. ArangoDB supports different types of endpoints:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6
- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption
- unix:///path/to/socket - Unix domain socket endpoint

If a TCP/IP endpoint is specified without a port number, then the default port (8529) will be used. If multiple endpoints need to be used, the option can be repeated multiple times.

The default endpoint for ArangoDB is *tcp://127.0.0.1:8529* or *tcp://localhost:8529*.

EXAMPLES

```
unix> ./arangod --server.endpoint tcp://127.0.0.1:8529
                --server.endpoint ssl://127.0.0.1:8530
                --ssl.keyfile server.pem /tmp/vocbase
2012-07-26T07:07:47Z [8161] INFO using SSL protocol version 'TLSv1'
2012-07-26T07:07:48Z [8161] INFO using endpoint 'ssl://127.0.0.1:8530' for http ssl requests
2012-07-26T07:07:48Z [8161] INFO using endpoint 'tcp://127.0.0.1:8529' for http tcp requests
2012-07-26T07:07:49Z [8161] INFO ArangoDB (version 1.1.alpha) is ready for business
2012-07-26T07:07:49Z [8161] INFO Have Fun!
```

TCP Endpoints

Given a hostname:

`--server.endpoint tcp://hostname:port`

Given an IPv4 address:

`--server.endpoint tcp://ipv4-address:port`

Given an IPv6 address:

`--server.endpoint tcp://[ipv6-address]:port`

On one specific ethernet interface each port can only be bound **exactly once**. You can look up your available interfaces using the *ifconfig* command on Linux / MacOSX - the Windows equivalent is *ipconfig* (See Wikipedia for more details). The general names of the interfaces differ on OS's and hardwares they run on. However, typically every host has a so called loopback interface, which is a virtual interface. By convention it always has the address *127.0.0.1* or *::1* (ipv6), and can only be reached from exactly the very same host. Ethernet interfaces usually have names like *eth0*, *wlan0*, *eth1:17*, *le0* or a plain text name in Windows.

To find out which services already use ports (so ArangoDB can't bind them anymore), you can use the netstat command (it behaves a little different on each platform, run it with *-lnpt* on Linux, *-p tcp* on MacOSX or with *-an* on windows for valuable information).

ArangoDB can also do a so called *broadcast bind* using *tcp://0.0.0.0:8529*. This way it will be reachable on all interfaces of the host. This may be useful on development systems that frequently change their network setup like laptops.

Reuse address

`--tcp.reuse-address`

If this boolean option is set to *true* then the socket option SO_REUSEADDR is set on all server endpoints, which is the default. If this option is set to *false* it is possible that it takes up to a minute after a server has terminated until it is possible for a new server to use the same endpoint again. This is why this is activated by default.

Please note however that under some operating systems this can be a security risk because it might be possible for another process to bind to the same address and port, possibly hijacking network traffic. Under Windows, ArangoDB additionally sets the flag SO_EXCLUSIVEADDRUSE as a measure to alleviate this problem.

Backlog size

`--tcp.backlog-size`

Allows to specify the size of the backlog for the *listen* system call The default value is 10. The maximum value is platform-dependent. Specifying a higher value than defined in the system header's SOMAXCONN may result in a warning on server start. The actual value used by *listen* may also be silently truncated on some platforms (this happens inside the *listen* system call).

SSL Configuration

SSL Endpoints

Given a hostname:

`--server.endpoint tcp://hostname:port`

Given an IPv4 address:

`--server.endpoint tcp://ipv4-address:port`

Given an IPv6 address:

`--server.endpoint tcp://[ipv6-address]:port`

Note: If you are using SSL-encrypted endpoints, you must also supply the path to a server certificate using the `--ssl.keyfile` option.

Keyfile

`--ssl.keyfile filename`

If SSL encryption is used, this option must be used to specify the filename of the server private key. The file must be PEM formatted and contain both the certificate and the server's private key.

The file specified by *filename* can be generated using openssl:

```
# create private key in file "server.key"
openssl genrsa -des3 -out server.key 1024

# create certificate signing request (csr) in file "server.csr"
openssl req -new -key server.key -out server.csr

# copy away original private key to "server.key.org"
cp server.key server.key.org

# remove passphrase from the private key
openssl rsa -in server.key.org -out server.key

# sign the csr with the key, creates certificate PEM file "server.crt"
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

# combine certificate and key into single PEM file "server.pem"
cat server.crt server.key > server.pem
```

You may use certificates issued by a Certificate Authority or self-signed certificates. Self-signed certificates can be created by a tool of your choice. When using OpenSSL for creating the self-signed certificate, the following commands should create a valid keyfile:

```
-----BEGIN CERTIFICATE-----

(base64 encoded certificate)

-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----

(base64 encoded private key)

-----END RSA PRIVATE KEY-----
```

For further information please check the manuals of the tools you use to create the certificate.

CA File

`--ssl.cafile filename`

This option can be used to specify a file with CA certificates that are sent to the client whenever the server requests a client certificate. If the file is specified, The server will only accept client requests with certificates issued by these CAs. Do not specify this option if you want clients to be able to connect without specific certificates.

The certificates in *filename* must be PEM formatted.

SSL protocol

`--ssl.protocol value`

Use this option to specify the default encryption protocol to be used. The following variants are available:

- 1: SSLv2
- 2: SSLv23
- 3: SSLv3
- 4: TLSv1
- 5: TLSv1.2 (recommended)

The default *value* is 4 (i.e. TLSv1). If available, set it to 5 (i.e. TLSv1.2), because lower protocol versions are known to be vulnerable to POODLE attack variants.

SSL cache

`--ssl.session-cache value`

Set to true if SSL session caching should be used.

value has a default value of *false* (i.e. no caching).

SSL peer certificate

This feature is available in the Enterprise Edition.

`--ssl.require-peer-certificate`

Require a peer certificate from the client before connecting.

SSL options

`--ssl.options value`

This option can be used to set various SSL-related options. Individual option values must be combined using bitwise OR.

Which options are available on your platform is determined by the OpenSSL version you use. The list of options available on your platform might be retrieved by the following shell command:

```
> grep "#define SSL_OP_.*" /usr/include/openssl/ssl.h

#define SSL_OP_MICROSOFT_SESS_ID_BUG                    0x00000001L
#define SSL_OP_NETSCAPE_CHALLENGE_BUG                   0x00000002L
#define SSL_OP_LEGACY_SERVER_CONNECT                    0x00000004L
#define SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG         0x00000008L
#define SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG              0x00000010L
#define SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER               0x00000020L
...
```

A description of the options can be found online in the OpenSSL documentation

SSL cipher

`--ssl.cipher-list cipher-list`

This option can be used to restrict the server to certain SSL ciphers only, and to define the relative usage preference of SSL ciphers.

The format of *cipher-list* is documented in the OpenSSL documentation.

To check which ciphers are available on your platform, you may use the following shell command:

```
> openssl ciphers -v

ECDHE-RSA-AES256-SHA    SSLv3 Kx=ECDH     Au=RSA  Enc=AES(256)  Mac=SHA1
ECDHE-ECDSA-AES256-SHA  SSLv3 Kx=ECDH     Au=ECDSA Enc=AES(256)  Mac=SHA1
DHE-RSA-AES256-SHA      SSLv3 Kx=DH       Au=RSA  Enc=AES(256)  Mac=SHA1
DHE-DSS-AES256-SHA      SSLv3 Kx=DH       Au=DSS  Enc=AES(256)  Mac=SHA1
DHE-RSA-CAMELLIA256-SHA SSLv3 Kx=DH       Au=RSA  Enc=Camellia(256)
Mac=SHA1
...
```

The default value for *cipher-list* is "ALL".

Command-Line Options for Logging

Log levels and topics

ArangoDB's log output is grouped into topics. `--log.level` can be specified multiple times at startup, for as many topics as needed. The log verbosity and output files can be adjusted per log topic. For example

```
--log.level startup=trace --log.level queries=trace --log.level info
```

will log messages concerning startup at trace level, AQL queries at trace level and everything else at info level.

In a configuration file, it is written like this:

```
[log]
level = startup=trace
level = queries=trace
level = info
```

Note that there must not be any whitespace around the second `=` .

The available log levels are:

- `fatal` : only logs fatal errors
- `error` : only logs errors
- `warning` : only logs warnings and errors
- `info` : logs information messages, warnings and errors
- `debug` : logs debug and information messages, warnings and errors
- `trace` : logs trace, debug and information messages, warnings and errors

Note that levels `debug` and `trace` will be very verbose.

Some relevant log topics available in ArangoDB 3 are:

- `agency` : information about the agency
- `collector` : information about the WAL collector's state
- `compactor` : information about the collection datafile compactor
- `datafiles` : datafile-related operations
- `mmap` : information about memory-mapping operations (including msync)
- `performance` : performance-releated messages
- `queries` : executed AQL queries, slow queries
- `replication` : replication-related info
- `requests` : HTTP requests
- `startup` : information about server startup and shutdown
- `threads` : information about threads

Log outputs

The log option `--log.output <definition>` allows directing the global or per-topic log output to different outputs. The output definition `<definition>` can be one of

- `-` for stdin
- `+` for stderr
- `syslog://<syslog-facility>`
- `syslog://<syslog-facility>/<application-name>`
- `file://<relative-path>`

The option can be specified multiple times in order to configure the output for different log topics. To set up a per-topic output configuration, use `--log.output <topic>=<definition>` , e.g.

queries=file://queries.txt

logs all queries to the file "queries.txt".

The old option `--log.file` is still available in 3.0 for convenience reasons. In 3.0 it is a shortcut for the more general option `--log.output file://filename`.

The old option `--log.requests-file` is still available in 3.0. It is now a shortcut for the more general option `--log.output requests=file://...`.

Using `--log.output` also allows directing log output to different files based on topics. For example, to log all AQL queries to a file "queries.log" one can use the options:

```
--log.level queries=trace --log.output queries=file:///path/to/queries.log
```

To additionally log HTTP request to a file named "requests.log" add the options:

```
--log.level requests=info --log.output requests=file:///path/to/requests.log
```

Forcing direct output

The option `--log.force-direct` can be used to disable logging in an extra logging thread. If set to `true`, any log messages are immediately printed in the thread that triggered the log message. This is non-optimal for performance but can aid debugging. If set to `false`, log messages are handed off to an extra logging thread, which asynchronously writes the log messages.

Local time

Log dates and times in local time zone: `--log.use-local-time`

If specified, all dates and times in log messages will use the server's local time-zone. If not specified, all dates and times in log messages will be printed in UTC / Zulu time. The date and time format used in logs is always `YYYY-MM-DD HH:MM:SS`, regardless of this setting. If UTC time is used, a `Z` will be appended to indicate Zulu time.

Line number

Log line number: `--log.line-number`

Normally, if an human readable fatal, error, warning or info message is logged, no information about the file and line number is provided. The file and line number is only logged for debug and trace message. This option can be use to always log these pieces of information.

Prefix

Log prefix: `--log.prefix prefix`

This option is used specify an prefix to logged text.

Thread

Log thread identifier: `--log.thread`

Whenever log output is generated, the process ID is written as part of the log information. Setting this option appends the thread id of the calling thread to the process id. For example,

```
2010-09-20T13:04:01Z [19355] INFO ready for business
```

when no thread is logged and

```
2010-09-20T13:04:17Z [19371-18446744072487317056] ready for business
```

when this command line option is set.

General Options

Database Upgrade

`--database.auto-upgrade`

Specifying this option will make the server perform a database upgrade at start. A database upgrade will first compare the version number stored in the file VERSION in the database directory with the current server version.

If the two version numbers match, the server will start normally.

If the version number found in the database directory is higher than the version number the server is running, the server expects this is an unintentional downgrade and will warn about this. It will however start normally. Using the server in these conditions is however not recommended nor supported.

If the version number found in the database directory is lower than the version number the server is running, the server will check whether there are any upgrade tasks to perform. It will then execute all required upgrade tasks and print their statuses. If one of the upgrade tasks fails, the server will exit and refuse to start. Re-starting the server with the upgrade option will then again trigger the upgrade check and execution until the problem is fixed. If all tasks are finished, the server will start normally.

Whether or not this option is specified, the server will always perform a version check on startup. Running the server with a non-matching version number in the VERSION file will make the server refuse to start.

Daemon

`--daemon`

Runs the server as a daemon (as a background process). This parameter can only be set if the pid (process id) file is specified. That is, unless a value to the parameter pid-file is given, then the server will report an error and exit.

Default Language

`--default-language default-language`

The default language ist used for sorting and comparing strings. The language value is a two-letter language code (ISO-639) or it is composed by a two-letter language code with and a two letter country code (ISO-3166). Valid languages are "de", "en", "en_US" or "en_UK".

The default default-language is set to be the system locale on that platform.

Supervisor

`--supervisor`

Executes the server in supervisor mode. In the event that the server unexpectedly terminates due to an internal error, the supervisor will automatically restart the server. Setting this flag automatically implies that the server will run as a daemon. Note that, as with the daemon flag, this flag requires that the pid-file parameter will set.

```
unix> ./arangod --supervisor --pid-file /var/run/arangodb.pid /tmp/vocbase/
2012-06-27T15:58:28Z [10133] INFO starting up in supervisor mode
```

As can be seen (e.g. by executing the ps command), this will start a supervisor process and the actual database process:

```
unix> ps fax | grep arangod
10137 ?        Ssl    0:00 ./arangod --supervisor --pid-file /var/run/arangodb.pid /tmp/vocbase/
10142 ?        Sl     0:00  \_ ./arangod --supervisor --pid-file /var/run/arangodb.pid /tmp/vocbase/
```

When the database process terminates unexpectedly, the supervisor process will start up a new database process:

```
> kill -SIGSEGV 10142

> ps fax | grep arangod
10137 ?        Ssl    0:00 ./arangod --supervisor --pid-file /var/run/arangodb.pid /tmp/vocbase/
10168 ?        Sl     0:00  \_ ./arangod --supervisor --pid-file /var/run/arangodb.pid /tmp/vocbase/
```

User identity

`--uid uid`

The name (identity) of the user the server will run as. If this parameter is not specified, the server will not attempt to change its UID, so that the UID used by the server will be the same as the UID of the user who started the server. If this parameter is specified, then the server will change its UID after opening ports and reading configuration files, but before accepting connections or opening other files (such as recovery files). This is useful when the server must be started with raised privileges (in certain environments) but security considerations require that these privileges be dropped once the server has started work.

Observe that this parameter cannot be used to bypass operating system security. In general, this parameter (and its corresponding relative gid) can lower privileges but not raise them.

Group identity

`--gid gid`

The name (identity) of the group the server will run as. If this parameter is not specified, then the server will not attempt to change its GID, so that the GID the server runs as will be the primary group of the user who started the server. If this parameter is specified, then the server will change its GID after opening ports and reading configuration files, but before accepting connections or opening other files (such as recovery files).

This parameter is related to the parameter uid.

Process identity

`--pid-file filename`

The name of the process ID file to use when running the server as a daemon. This parameter must be specified if either the flag *daemon* or *supervisor* is set.

Console

`--console`

Runs the server in an exclusive emergency console mode. When starting the server with this option, the server is started with an interactive JavaScript emergency console, with all networking and HTTP interfaces of the server disabled.

No requests can be made to the server in this mode, and the only way to work with the server in this mode is by using the emergency console. Note that the server cannot be started in this mode if it is already running in this or another mode.

Random Generator

`--random.generator arg`

The argument is an integer (1,2,3 or 4) which sets the manner in which random numbers are generated. The default method (3) is to use the a non-blocking random (or pseudorandom) number generator supplied by the operating system.

Specifying an argument of 2, uses a blocking random (or pseudorandom) number generator. Specifying an argument 1 sets a pseudorandom number generator using an implication of the Mersenne Twister MT19937 algorithm. Algorithm 4 is a combination of the blocking random number generator and the Mersenne Twister.

Enable/disable authentication

`--server.authentication` Setting this option to *false* will turn off authentication on the server side so all clients can execute any action without authorization and privilege checks. The default value is *true*.

JWT Secret

`--server.jwt-secret secret`

ArangoDB will use JWTs to authenticate requests. Using this option lets you specify a JWT.

In single server setups and when not specifying this secret ArangoDB will generate a secret.

In cluster deployments which have authentication enabled a secret must be set consistently across all cluster tasks so they can talk to each other.

Enable/disable authentication for UNIX domain sockets

`--server.authentication-unix-sockets value`

Setting *value* to true will turn off authentication on the server side for requests coming in via UNIX domain sockets. With this flag enabled, clients located on the same host as the ArangoDB server can use UNIX domain sockets to connect to the server without authentication. Requests coming in by other means (e.g. TCP/IP) are not affected by this option.

The default value is *false*.

Note: this option is only available on platforms that support UNIX domain sockets.

Enable/disable authentication for system API requests only

`--server.authentication-system-only boolean` Controls whether incoming requests need authentication only if they are directed to the ArangoDB's internal APIs and features, located at */_api/*, */_admin/* etc. If the flag is set to *true*, then HTTP authentication is only required for requests going to URLs starting with /_, but not for other URLs. The flag can thus be used to expose a user-made API without HTTP authentication to the outside world, but to prevent the outside world from using the ArangoDB API and the admin interface without authentication. Note that checking the URL is performed after any database name prefix has been removed. That means when the actual URL called is */_db/_system/myapp/myaction*, the URL */myapp/myaction* will be used for *authentication-system-only* check. The default is *true*. Note that authentication still needs to be enabled for the server regularly in order for HTTP authentication to be forced for the ArangoDB API and the web interface. Setting only this flag is not enough. You can control ArangoDB's general authentication feature with the --*server.authentication* flag.

Enable/disable replication applier

`--database.replication-applier flag`

If *false* the server will start with replication appliers turned off, even if the replication appliers are configured with the *autoStart* option. Using the command-line option will not change the value of the *autoStart* option in the applier configuration, but will suppress auto-starting the replication applier just once.

If the option is not used, ArangoDB will read the applier configuration from the file *REPLICATION-APPLIER-CONFIG* on startup, and use the value of the *autoStart* attribute from this file.

The default is *true*.

Keep-alive timeout

`--http.keep-alive-timeout`

Allows to specify the timeout for HTTP keep-alive connections. The timeout value must be specified in seconds. Idle keep-alive connections will be closed by the server automatically when the timeout is reached. A keep-alive-timeout value 0 will disable the keep alive feature entirely.

Hide Product header

`--http.hide-product-header`

If *true*, the server will exclude the HTTP header "Server: ArangoDB" in HTTP responses. If set to *false*, the server will send the header in responses.

The default is *false*.

Allow method override

`--http.allow-method-override`

When this option is set to *true*, the HTTP request method will optionally be fetched from one of the following HTTP request headers if present in the request:

- *x-http-method*
- *x-http-method-override*
- *x-method-override*

If the option is set to *true* and any of these headers is set, the request method will be overridden by the value of the header. For example, this allows issuing an HTTP DELETE request which to the outside world will look like an HTTP GET request. This allows bypassing proxies and tools that will only let certain request types pass.

Setting this option to *true* may impose a security risk so it should only be used in controlled environments.

The default value for this option is *false*.

Server threads

`--server.threads number`

Specifies the *number* of threads that are spawned to handle requests.

Toggling server statistics

`--server.statistics value`

If this option is *value* is *false*, then ArangoDB's statistics gathering is turned off. Statistics gathering causes regular CPU activity so using this option to turn it off might relieve heavy-loaded instances a bit.

Session timeout

time to live for server sessions `--server.session-timeout value`

The timeout for web interface sessions, using for authenticating requests to the web interface (/_admin/aardvark) and related areas.

Sessions are only used when authentication is turned on.

Foxx queues

enable or disable the Foxx queues feature `--foxx.queues flag` If *true*, the Foxx queues will be available and jobs in the queues will be executed asynchronously. The default is *true*. When set to `false` the queue manager will be disabled and any jobs are prevented from being processed, which may reduce CPU load a bit.

Foxx queues poll interval

poll interval for Foxx queues `--foxx.queues-poll-interval value` The poll interval for the Foxx queues manager. The value is specified in seconds. Lower values will mean more immediate and more frequent Foxx queue job execution, but will make the queue thread wake up and query the queues more often. When set to a low value, the queue thread might cause CPU load. The default is *1* second. If Foxx queues are not used much, then this value may be increased to make the queues thread wake up less.

Directory

`--database.directory directory`

The directory containing the collections and datafiles. Defaults to */var/lib/arango*. When specifying the database directory, please make sure the directory is actually writable by the arangod process.

You should further not use a database directory which is provided by a network filesystem such as NFS. The reason is that networked filesystems might cause inconsistencies when there are multiple parallel readers or writers or they lack features required by arangod (e.g. flock()).

`directory`

When using the command line version, you can simply supply the database directory as argument.

Examples

```
> ./arangod --server.endpoint tcp://127.0.0.1:8529 --database.directory
/tmp/vocbase
```

Journal size

`--database.maximal-journal-size size` Maximal size of journal in bytes. Can be overwritten when creating a new collection. Note that this also limits the maximal size of a single document. The default is *32MB*.

Wait for sync

default wait for sync behavior `--database.wait-for-sync boolean` Default wait-for-sync value. Can be overwritten when creating a new collection. The default is *false*.

Force syncing of properties

force syncing of collection properties to disk `--database.force-sync-properties boolean` Force syncing of collection properties to disk after creating a collection or updating its properties. If turned off, no fsync will happen for the collection and database properties stored in `parameter.json` files in the file system. Turning off this option will speed up workloads that create and drop a lot of collections (e.g. test suites). The default is *true*.

Enable/disable AQL query tracking

`--query.tracking flag`

If *true*, the server's AQL slow query tracking feature will be enabled by default. Tracking of queries can be disabled by setting the option to *false*.

The default is *true*.

Threshold for slow AQL queries

`--query.slow-threshold value`

By setting *value* it can be controlled after what execution time an AQL query is considered "slow". Any slow queries that exceed the execution time specified in *value* will be logged when they are finished. The threshold value is specified in seconds. Tracking of slow queries can be turned off entirely by setting the option `--query.tracking` to *false*.

The default value is *10.0*.

Throw collection not loaded error

`--database.throw-collection-not-loaded-error flag`

Accessing a not-yet loaded collection will automatically load a collection on first access. This flag controls what happens in case an operation would need to wait for another thread to finalize loading a collection. If set to *true*, then the first operation that accesses an unloaded collection will load it. Further threads that try to access the same collection while it is still loading will get an error (1238, *collection not loaded*). When the initial operation has completed loading the collection, all operations on the collection can be carried out normally, and error 1238 will not be thrown.

If set to *false*, the first thread that accesses a not-yet loaded collection will still load it. Other threads that try to access the collection while loading will not fail with error 1238 but instead block until the collection is fully loaded. This configuration might lead to all server threads being blocked because they are all waiting for the same collection to complete loading. Setting the option to *true* will prevent this from happening, but requires clients to catch error 1238 and react on it (maybe by scheduling a retry for later).

The default value is *false*.

AQL Query caching mode

`--query.cache-mode`

Toggles the AQL query cache behavior. Possible values are:

- *off*: do not use query cache
- *on*: always use query cache, except for queries that have their *cache* attribute set to *false*
- *demand*: use query cache only for queries that have their *cache* attribute set to *true*

AQL Query cache size

`--query.cache-entries`

Maximum number of query results that can be stored per database-specific query cache. If a query is eligible for caching and the number of items in the database's query cache is equal to this threshold value, another cached query result will be removed from the cache.

This option only has an effect if the query cache mode is set to either *on* or *demand*.

Index threads

`--database.index-threads`

Specifies the *number* of background threads for index creation. When a collection contains extra indexes other than the primary index, these other indexes can be built by multiple threads in parallel. The index threads are shared among multiple collections and databases. Specifying a value of *0* will turn off parallel building, meaning that indexes for each collection are built sequentially by the thread that opened the collection. If the number of index threads is greater than 1, it will also be used to built the edge index of a collection in parallel (this also requires the edge index in the collection to be split into multiple buckets).

V8 contexts

`--javascript.v8-contexts number`

Specifies the *number* of V8 contexts that are created for executing JavaScript code. More contexts allow execute more JavaScript actions in parallel, provided that there are also enough threads available. Please note that each V8 context will use a substantial amount of memory and requires periodic CPU processing time for garbage collection.

Garbage collection frequency (time-based)

`--javascript.gc-frequency frequency`

Specifies the frequency (in seconds) for the automatic garbage collection of JavaScript objects. This setting is useful to have the garbage collection still work in periods with no or little numbers of requests.

Garbage collection interval (request-based)

`--javascript.gc-interval interval`

Specifies the interval (approximately in number of requests) that the garbage collection for JavaScript objects will be run in each thread.

V8 options

`--javascript.v8-options options`

Optional arguments to pass to the V8 Javascript engine. The V8 engine will run with default settings unless explicit options are specified using this option. The options passed will be forwarded to the V8 engine which will parse them on its own. Passing invalid options may result in an error being printed on stderr and the option being ignored.

Options need to be passed in one string, with V8 option names being prefixed with double dashes. Multiple options need to be separated by whitespace. To get a list of all available V8 options, you can use the value *"--help"* as follows:

```
--javascript.v8-options="--help"
```

Another example of specific V8 options being set at startup:

```
--javascript.v8-options="--log"
```

Names and features or usable options depend on the version of V8 being used, and might change in the future if a different version of V8 is being used in ArangoDB. Not all options offered by V8 might be sensible to use in the context of ArangoDB. Use the specific options only if you are sure that they are not harmful for the regular database operation.

Write-ahead log options

Since ArangoDB 2.2, the server will write all data-modification operations into its write-ahead log.

The write-ahead log is a sequence of logfiles that are written in an append-only fashion. Full logfiles will eventually be garbage-collected, and the relevant data might be transferred into collection journals and datafiles. Unneeded and already garbage-collected logfiles will either be deleted or kept for the purpose of keeping a replication backlog.

Directory

The WAL logfiles directory: `--wal.directory`

Specifies the directory in which the write-ahead logfiles should be stored. If this option is not specified, it defaults to the subdirectory *journals* in the server's global database directory. If the directory is not present, it will be created.

Logfile size

the size of each WAL logfile `--wal.logfile-size` Specifies the filesize (in bytes) for each write-ahead logfile. The logfile size should be chosen so that each logfile can store a considerable amount of documents. The bigger the logfile size is chosen, the longer it will take to fill up a single logfile, which also influences the delay until the data in a logfile will be garbage-collected and written to collection journals and datafiles. It also affects how long logfile recovery will take at server start.

Allow oversize entries

whether or not oversize entries are allowed `--wal.allow-oversize-entries` Whether or not it is allowed to store individual documents that are bigger than would fit into a single logfile. Setting the option to false will make such operations fail with an error. Setting the option to true will make such operations succeed, but with a high potential performance impact. The reason is that for each oversize operation, an individual oversize logfile needs to be created which may also block other operations. The option should be set to *false* if it is certain that documents will always have a size smaller than a single logfile.

Number of reserve logfiles

maximum number of reserve logfiles `--wal.reserve-logfiles` The maximum number of reserve logfiles that ArangoDB will create in a background process. Reserve logfiles are useful in the situation when an operation needs to be written to a logfile but the reserve space in the logfile is too low for storing the operation. In this case, a new logfile needs to be created to store the operation. Creating new logfiles is normally slow, so ArangoDB will try to pre-create logfiles in a background process so there are always reserve logfiles when the active logfile gets full. The number of reserve logfiles that ArangoDB keeps in the background is configurable with this option.

Number of historic logfiles

maximum number of historic logfiles `--wal.historic-logfiles` The maximum number of historic logfiles that ArangoDB will keep after they have been garbage-collected. If no replication is used, there is no need to keep historic logfiles except for having a local changelog. In a replication setup, the number of historic logfiles affects the amount of data a slave can fetch from the master's logs. The more historic logfiles, the more historic data is available for a slave, which is useful if the connection between master and slave is unstable or slow. Not having enough historic logfiles available might lead to logfile data being deleted on the master already before a slave has fetched it.

Sync interval

interval for automatic, non-requested disk syncs `--wal.sync-interval` The interval (in milliseconds) that ArangoDB will use to automatically synchronize data in its write-ahead logs to disk. Automatic syncs will only be performed for not-yet synchronized data, and only for operations that have been executed without the *waitForSync* attribute.

Throttling

Throttle writes to WAL when at least such many operations are waiting for garbage collection: `--wal.throttle-when-pending`

The maximum value for the number of write-ahead log garbage-collection queue elements. If set to *0*, the queue size is unbounded, and no write-throttling will occur. If set to a non-zero value, write-throttling will automatically kick in when the garbage-collection queue contains at least as many elements as specified by this option. While write-throttling is active, data-modification operations will intentionally be delayed by a configurable amount of time. This is to ensure the write-ahead log garbage collector can catch up with the operations executed. Write-throttling will stay active until the garbage-collection queue size goes down below the specified value. Write-throttling is turned off by default.

`--wal.throttle-wait`

This option determines the maximum wait time (in milliseconds) for operations that are write-throttled. If write-throttling is active and a new write operation is to be executed, it will wait for at most the specified amount of time for the write-ahead log garbage-collection queue size to fall below the throttling threshold. If the queue size decreases before the maximum wait time is over, the operation will be executed normally. If the queue size does not decrease before the wait time is over, the operation will be aborted with an error. This option only has an effect if `--wal.throttle-when-pending` has a non-zero value, which is not the default.

Number of slots

Maximum number of slots to be used in parallel: `--wal.slots`

Configures the amount of write slots the write-ahead log can give to write operations in parallel. Any write operation will lease a slot and return it to the write-ahead log when it is finished writing the data. A slot will remain blocked until the data in it was synchronized to disk. After that, a slot becomes reusable by following operations. The required number of slots is thus determined by the parallelity of write operations and the disk synchronization speed. Slow disks probably need higher values, and fast disks may only require a value lower than the default.

Ignore logfile errors

Ignore logfile errors when opening logfiles: `--wal.ignore-logfile-errors`

Ignores any recovery errors caused by corrupted logfiles on startup. When set to *false*, the recovery procedure on startup will fail with an error whenever it encounters a corrupted (that includes only half-written) logfile. This is a security precaution to prevent data loss in case of disk errors etc. When the recovery procedure aborts because of corruption, any corrupted files can be inspected and fixed (or removed) manually and the server can be restarted afterwards.

Setting the option to *true* will make the server continue with the recovery procedure even in case it detects corrupt logfile entries. In this case it will stop at the first corrupted logfile entry and ignore all others, which might cause data loss.

Ignore recovery errors

Ignore recovery errors: `--wal.ignore-recovery-errors`

Ignores any recovery errors not caused by corrupted logfiles but by logical errors. Logical errors can occur if logfiles or any other server datafiles have been manually edited or the server is somehow misconfigured.

Ignore (non-WAL) datafile errors

Ignore datafile errors when loading collections: `--database.ignore-datafile-errors boolean`

If set to `false`, CRC mismatch and other errors in collection datafiles will lead to a collection not being loaded at all. The collection in this case becomes unavailable. If such collection needs to be loaded during WAL recovery, the WAL recovery will also abort (if not forced with option `--wal.ignore-recovery-errors true`).

Setting this flag to `false` protects users from unintentionally using a collection with corrupted datafiles, from which only a subset of the original data can be recovered. Working with such collection could lead to data loss and follow up errors. In order to access such collection, it is required to inspect and repair the collection datafile with the datafile debugger (arango-dfdb).

If set to `true`, CRC mismatch and other errors during the loading of a collection will lead to the datafile being partially loaded, up to the position of the first error. All data up to until the invalid position will be loaded. This will enable users to continue with collection datafiles even if they are corrupted, but this will result in only a partial load of the original data and potential follow up errors. The WAL recovery will still abort when encountering a collection with a corrupted datafile, at least if `--wal.ignore-recovery-errors` is not set to `true`.

The default value is *false*, so collections with corrupted datafiles will not be loaded at all, preventing partial loads and follow up errors. However, if such collection is required at server startup, during WAL recovery, the server will abort the recovery and refuse to start.

Clusters Options

Node ID

This server's id: `--cluster.my-local-info info`

Some local information about the server in the cluster, this can for example be an IP address with a process ID or any string unique to the server. Specifying *info* is mandatory on startup if the server id (see below) is not specified. Each server of the cluster must have a unique local info. This is ignored if my-id below is specified.

Agency endpoint

List of agency endpoints: `--cluster.agency-endpoint endpoint`

An agency endpoint the server can connect to. The option can be specified multiple times, so the server can use a cluster of agency servers. Endpoints have the following pattern:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6
- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption

At least one endpoint must be specified or ArangoDB will refuse to start. It is recommended to specify at least two endpoints so ArangoDB has an alternative endpoint if one of them becomes unavailable.

Examples

```
--cluster.agency-endpoint tcp://192.168.1.1:4001 --cluster.agency-endpoint
tcp://192.168.1.2:4002
```

Agency prefix

Global agency prefix: `--cluster.agency-prefix prefix`

The global key prefix used in all requests to the agency. The specified prefix will become part of each agency key. Specifying the key prefix allows managing multiple ArangoDB clusters with the same agency server(s).

prefix must consist of the letters *a-z*, *A-Z* and the digits *0-9* only. Specifying a prefix is mandatory.

Examples

```
--cluster.prefix mycluster
```

MyId

This server's id: `--cluster.my-id id`

The local server's id in the cluster. Specifying *id* is mandatory on startup. Each server of the cluster must have a unique id.

Specifying the id is very important because the server id is used for determining the server's role and tasks in the cluster.

id must be a string consisting of the letters *a-z*, *A-Z* or the digits *0-9* only.

MyAddress

This server's address / endpoint: `--cluster.my-address endpoint`

The server's endpoint for cluster-internal communication. If specified, it must have the following pattern:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6

- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption

If no *endpoint* is specified, the server will look up its internal endpoint address in the agency. If no endpoint can be found in the agency for the server's id, ArangoDB will refuse to start.

Examples

```
--cluster.my-address tcp://192.168.1.1:8530
```

Asynchronous Tasks

maximal queue size

Maximum size of the queue for requests: `--server.maximal-queue-size size`

Specifies the maximum *size* of the queue for asynchronous task execution. If the queue already contains *size* tasks, new tasks will be rejected until other tasks are popped from the queue. Setting this value may help preventing from running out of memory if the queue is filled up faster than the server can process requests.

Durability Configuration

Global Configuration

There are global configuration values for durability, which can be adjusted by specifying the following configuration options:

default wait for sync behavior `--database.wait-for-sync boolean` Default wait-for-sync value. Can be overwritten when creating a new collection. The default is *false*.

force syncing of collection properties to disk `--database.force-sync-properties boolean` Force syncing of collection properties to disk after creating a collection or updating its properties. If turned off, no fsync will happen for the collection and database properties stored in `parameter.json` files in the file system. Turning off this option will speed up workloads that create and drop a lot of collections (e.g. test suites). The default is *true*.

interval for automatic, non-requested disk syncs `--wal.sync-interval` The interval (in milliseconds) that ArangoDB will use to automatically synchronize data in its write-ahead logs to disk. Automatic syncs will only be performed for not-yet synchronized data, and only for operations that have been executed without the *waitForSync* attribute.

Per-collection configuration

You can also configure the durability behavior on a per-collection basis. Use the ArangoDB shell to change these properties.

gets or sets the properties of a collection `collection.properties()` Returns an object containing all collection properties.

- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.
- *journalSize* : The size of the journal in bytes.
- *isVolatile*: If *true* then the collection data will be kept in memory only and ArangoDB will not write or sync the data to disk.
- *keyOptions* (optional) additional options for key generation. This is a JSON array containing the following attributes (note: some of the attributes are optional):
  - *type*: the type of the key generator used for the collection.
  - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the *_key* attribute of a document. If set to *false*, then the key generator will solely be responsible for generating keys and supplying own key values in the *_key* attribute of documents is considered an error.
  - *increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
  - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.
- *indexBuckets*: number of buckets into which indexes using a hash table are split. The default is 16 and this number has to be a power of 2 and less than or equal to 1024. For very large collections one should increase this to avoid long pauses when the hash table has to be initially built or resized, since buckets are resized individually and can be initially built in parallel. For example, 64 might be a sensible value for a collection with 100 000 000 documents. Currently, only the edge index respects this value, but other index types might follow in future ArangoDB versions. Changes (see below) are applied when the collection is loaded the next time. In a cluster setup, the result will also contain the following attributes:
- *numberOfShards*: the number of shards of the collection.
- *shardKeys*: contains the names of document attributes that are used to determine the target shard for documents. `collection.properties(properties)` Changes the collection properties. *properties* must be a object with one or more of the following attribute(s):
- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.
- *journalSize* : The size of the journal in bytes.
- *indexBuckets* : See above, changes are only applied when the collection is loaded the next time. *Note*: it is not possible to change the journal size after the journal or datafile has been created. Changing this parameter will only effect newly created journals. Also note that you cannot lower the journal size to less then size of the largest document already stored in the collection. **Note**: some other collection properties, such as *type*, *isVolatile*, or *keyOptions* cannot be changed once the collection is created.

Examples

Read all properties

```
arangosh> db.example.properties();
```

show execution results

Change a property

```
arangosh> db.example.properties({ waitForSync : true });
```

show execution results

Per-operation configuration

Many data-modification operations and also ArangoDB's transactions allow to specify a *waitForSync* attribute, which when set ensures the operation data has been synchronized to disk when the operation returns.

Disk-Usage Configuration

The amount of disk space used by ArangoDB is determined by a few configuration options.

Global Configuration

The total amount of disk storage required by ArangoDB is determined by the size of the write-ahead logfiles plus the sizes of the collection journals and datafiles.

There are the following options for configuring the number and sizes of the write-ahead logfiles:

maximum number of reserve logfiles `--wal.reserve-logfiles` The maximum number of reserve logfiles that ArangoDB will create in a background process. Reserve logfiles are useful in the situation when an operation needs to be written to a logfile but the reserve space in the logfile is too low for storing the operation. In this case, a new logfile needs to be created to store the operation. Creating new logfiles is normally slow, so ArangoDB will try to pre-create logfiles in a background process so there are always reserve logfiles when the active logfile gets full. The number of reserve logfiles that ArangoDB keeps in the background is configurable with this option.

maximum number of historic logfiles `--wal.historic-logfiles` The maximum number of historic logfiles that ArangoDB will keep after they have been garbage-collected. If no replication is used, there is no need to keep historic logfiles except for having a local changelog. In a replication setup, the number of historic logfiles affects the amount of data a slave can fetch from the master's logs. The more historic logfiles, the more historic data is available for a slave, which is useful if the connection between master and slave is unstable or slow. Not having enough historic logfiles available might lead to logfile data being deleted on the master already before a slave has fetched it.

the size of each WAL logfile `--wal.logfile-size` Specifies the filesize (in bytes) for each write-ahead logfile. The logfile size should be chosen so that each logfile can store a considerable amount of documents. The bigger the logfile size is chosen, the longer it will take to fill up a single logfile, which also influences the delay until the data in a logfile will be garbage-collected and written to collection journals and datafiles. It also affects how long logfile recovery will take at server start.

whether or not oversize entries are allowed `--wal.allow-oversize-entries` Whether or not it is allowed to store individual documents that are bigger than would fit into a single logfile. Setting the option to false will make such operations fail with an error. Setting the option to true will make such operations succeed, but with a high potential performance impact. The reason is that for each oversize operation, an individual oversize logfile needs to be created which may also block other operations. The option should be set to *false* if it is certain that documents will always have a size smaller than a single logfile. When data gets copied from the write-ahead logfiles into the journals or datafiles of collections, files will be created on the collection level. How big these files are is determined by the following global configuration value:

`--database.maximal-journal-size size` Maximal size of journal in bytes. Can be overwritten when creating a new collection. Note that this also limits the maximal size of a single document. The default is *32MB*.

Per-collection configuration

The journal size can also be adjusted on a per-collection level using the collection's *properties* method.

Introduction to Replication

Replication allows you to *replicate* data onto another machine. It forms the base of all disaster recovery and failover features ArangoDB offers.

ArangoDB offers asynchronous and synchronous replication which both have their pros and cons. Both modes may and should be combined in a real world scenario and be applied in the usecase where they excel most.

We will describe pros and cons of each of them in the following sections.

Synchronous replication

Synchronous replication only works in in a cluster and is typically used for mission critical data which must be accessible at all times. Synchronous replication generally stores a copy of the data on another host and keeps it in sync. Essentially when storing data after enabling synchronous replication the cluster will wait for all replicas to write all the data before greenlighting the write operation to the client. This will naturally increase the latency a bit, since one more network hop is needed for each write. However it will enable the cluster to immediately fail over to a replica whenever an outage has been detected, without losing any committed data, and mostly without even signaling an error condition to the client.

Synchronous replication is organized in a way that every shard has a leader and r-1 followers. The number of followers can be controlled using the `replicationFactor` whenever you create a collection, the `replicationFactor` is the total number of copies being kept, that is, it is one plus the number of followers.

Asynchronous replication

In ArangoDB any write operation will be logged to the write-ahead log. When using Asynchronous replication slaves will connect to a master and apply all the events from the log in the same order locally. After that, they will have the same state of data as the master database.

Asynchronous replication

Asynchronous replication works by logging every data modification on a *master* and replaying these events on a number of *slaves*.

Transactions are honored in replication, i.e. transactional write operations will become visible on slaves atomically.

As all write operations will be logged to a master database's write-ahead log, the replication in ArangoDB currently cannot be used for write-scaling. The main purposes of the replication in current ArangoDB are to provide read-scalability and "hot backups" of specific databases.

It is possible to connect multiple slave databases to the same master database. Slave databases should be used as read-only instances, and no user-initiated write operations should be carried out on them. Otherwise data conflicts may occur that cannot be solved automatically, and that will make the replication stop.

In an asynchronous replication scenario slaves will *pull* changes from the master database. Slaves need to know to which master database they should connect to, but a master database is not aware of the slaves that replicate from it. When the network connection between the master database and a slave goes down, write operations on the master can continue normally. When the network is up again, slaves can reconnect to the master database and transfer the remaining changes. This will happen automatically provided slaves are configured appropriately.

Replication lag

In this setup, write operations are applied first in the master database, and applied in the slave database(s) afterwards.

For example, let's assume a write operation is executed in the master database at point in time t0. To make a slave database apply the same operation, it must first fetch the write operation's data from master database's write-ahead log, then parse it and apply it locally. This will happen at some point in time after t0, let's say t1.

The difference between t1 and t0 is called the *replication lag*, and it is unavoidable in asynchronous replication. The amount of replication lag depends on many factors, a few of which are:

- the network capacity between the slaves and the master
- the load of the master and the slaves
- the frequency in which slaves poll the master for updates

Between t0 and t1, the state of data on the master is newer than the state of data on the slave(s). At point in time t1, the state of data on the master and slave(s) is consistent again (provided no new data modifications happened on the master in between). Thus, the replication will lead to an *eventually consistent* state of data.

Replication configuration

The replication is turned off by default. In order to create a master-slave setup, the so-called *replication applier* needs to be enabled on the slave databases.

Replication is configured on a per-database level. If multiple database are to be replicated, the replication must be set up individually per database.

The replication applier on the slave can be used to perform a one-time synchronization with the master (and then stop), or to perform an ongoing replication of changes. To resume replication on slave restart, the *autoStart* attribute of the replication applier must be set to *true*.

Replication overhead

As the master servers are logging any write operation in the write-ahead-log anyway replication doesn't cause any extra overhead on the master. However it will of course cause some overhead for the master to serve incoming read requests of the slaves. Returning the requested data is however a trivial task for the master and should not result in a notable performance degration in production.

Components

Replication Logger

Purpose

The replication logger will write all data-modification operations into the write-ahead log. This log may then be read by clients to replay any data modification on a different server.

Checking the state

To query the current state of the logger, use the *state* command:

```
require("@arangodb/replication").logger.state();
```

The result might look like this:

```
{
  "state" : {
    "running" : true,
      "lastLogTick" : "133322013",
      "totalEvents" : 16,
      "time" : "2014-07-06T12:58:11Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "40897075811372"
  },
  "clients" : {
  }
}
```

The *running* attribute will always be true. In earlier versions of ArangoDB the replication was optional and this could have been *false*.

The *totalEvents* attribute indicates how many log events have been logged since the start of the ArangoDB server. Finally, the *lastLogTick* value indicates the id of the last operation that was written to the server's write-ahead log. It can be used to determine whether new operations were logged, and is also used by the replication applier for incremental fetching of data.

Note: The replication logger state can also be queried via the HTTP API.

To query which data ranges are still available for replication clients to fetch, the logger provides the *firstTick* and *tickRanges* functions:

```
require("@arangodb/replication").logger.firstTick();
```

This will return the minimum tick value that the server can provide to replication clients via its replication APIs. The *tickRanges* function returns the minimum and maximum tick values per logfile:

```
require("@arangodb/replication").logger.tickRanges();
```

Replication Applier

Purpose

The purpose of the replication applier is to read data from a master database's event log, and apply them locally. The applier will check the master database for new operations periodically. It will perform an incremental synchronization, i.e. only asking the master for operations that occurred after the last synchronization.

The replication applier does not get notified by the master database when there are "new" operations available, but instead uses the pull principle. It might thus take some time (the so-called *replication lag*) before an operation from the master database gets shipped to and applied in a slave database.

The replication applier of a database is run in a separate thread. It may encounter problems when an operation from the master cannot be applied safely, or when the connection to the master database goes down (network outage, master database is down or unavailable etc.). In this case, the database's replication applier thread might terminate itself. It is then up to the administrator to fix the problem and restart the database's replication applier.

If the replication applier cannot connect to the master database, or the communication fails at some point during the synchronization, the replication applier will try to reconnect to the master database. It will give up reconnecting only after a configurable amount of connection attempts.

The replication applier state is queryable at any time by using the *state* command of the applier. This will return the state of the applier of the current database:

```
require("@arangodb/replication").applier.state();
```

The result might look like this:

```
{
  "state" : {
    "running" : true,
    "lastAppliedContinuousTick" : "152786205",
    "lastProcessedContinuousTick" : "152786205",
    "lastAvailableContinuousTick" : "152786205",
    "progress" : {
      "time" : "2014-07-06T13:04:57Z",
      "message" : "fetching master log from offset 152786205",
      "failedConnects" : 0
    },
    "totalRequests" : 38,
    "totalFailedConnects" : 0,
    "totalEvents" : 1,
    "lastError" : {
      "errorNum" : 0
    },
    "time" : "2014-07-06T13:04:57Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "210189384542896"
  },
  "endpoint" : "tcp://master.example.org:8529",
  "database" : "_system"
}
```

The *running* attribute indicates whether the replication applier of the current database is currently running and polling the server at *endpoint* for new events.

The *progress.failedConnects* attribute shows how many failed connection attempts the replication applier currently has encountered in a row. In contrast, the *totalFailedConnects* attribute indicates how many failed connection attempts the applier has made in total. The *totalRequests* attribute shows how many requests the applier has sent to the master database in total. The *totalEvents* attribute shows how many log events the applier has read from the master.

The *progress.message* sub-attribute provides a brief hint of what the applier currently does (if it is running). The *lastError* attribute also has an optional *errorMessage* sub-attribute, showing the latest error message. The *errorNum* sub-attribute of the *lastError* attribute can be used by clients to programmatically check for errors. It should be *0* if there is no error, and it should be non-zero if the applier terminated itself due to a problem.

Here is an example of the state after the replication applier terminated itself due to (repeated) connection problems:

```
{
  "state" : {
    "running" : false,
    "progress" : {
      "time" : "2014-07-06T13:14:37Z",
      "message" : "applier stopped",
      "failedConnects" : 6
    },
    "totalRequests" : 79,
    "totalFailedConnects" : 11,
    "totalEvents" : 0,
    "lastError" : {
      "time" : "2014-07-06T13:09:41Z",
      "errorMessage" : "could not connect to master at tcp://master.example.org:8529: Could not connect to 'tcp:/...",
      "errorNum" : 1400
    },
    ...
  }
}
```

Note: the state of a database's replication applier is queryable via the HTTP API, too. Please refer to HTTP Interface for Replication for more details.

All-in-one setup

To copy the initial data from the **slave** to the master and start the continuous replication, there is an all-in-one command *setupReplication*:

```
require("@arangodb/replication").setupReplication(configuration);
```

The following example demonstrates how to use the command for setting up replication for the *_system* database. Note that it should be run on the slave and not the master:

```
db._useDatabase("_system");
require("@arangodb/replication").setupReplication({
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd",
  verbose: false,
  includeSystem: false,
  incremental: true,
  autoResync: true
});
```

The command will return when the initial synchronization is finished and the continuous replication is started, or in case the initial synchronization has failed.

If the initial synchronization is successful, the command will store the given configuration on the slave. It also configures the continuous replication to start automatically if the slave is restarted, i.e. *autoStart* is set to *true*.

If the command is run while the slave's replication applier is already running, it will first stop the running applier, drop its configuration and do a resynchronization of data with the master. It will then use the provided configration, overwriting any previously existing replication configuration on the slave.

Starting and Stopping

To manually start and stop the applier in the current database, the *start* and *stop* commands can be used like this:

```
require("@arangodb/replication").applier.start(<tick>);
require("@arangodb/replication").applier.stop();
```

Note: Starting a replication applier without setting up an initial configuration will fail. The replication applier will look for its configuration in a file named *REPLICATION-APPLIER-CONFIG* in the current database's directory. If the file is not present, ArangoDB will use some default configuration, but it cannot guess the endpoint (the address of the master database) the applier should connect to. Thus starting the applier without configuration will fail.

Note that at the first time you start the applier, you should pass the value returned in the *lastLogTick* attribute of the initial sync operation.

Note: Starting a database's replication applier via the *start* command will not necessarily start the applier on the next and following ArangoDB server restarts. Additionally, stopping a database's replication applier manually will not necessarily prevent the applier from being started again on the next server start. All of this is configurable separately (hang on reading).

Note: when stopping and restarting the replication applier of database, it will resume where it last stopped. This is sensible because replication log events should be applied incrementally. If the replication applier of a database has never been started before, it needs some *tick* value from the master's log from which to start fetching events.

There is one caveat to consider when stopping a replication on the slave: if there are still ongoing replicated transactions that are neither committed or aborted, stopping the replication applier will cause these operations to be lost for the slave. If these transactions commit on the master later and the replication is resumed, the slave will not be able to commit these transactions, too. Thus stopping the replication applier on the slave manually should only be done if there is certainty that there are no ongoing transactions on the master.

Configuration

To configure the replication applier of a specific database, use the *properties* command. Using it without any arguments will return the applier's current configuration:

```
require("@arangodb/replication").applier.properties();
```

The result might look like this:

```
{
  "requestTimeout" : 600,
  "connectTimeout" : 10,
  "ignoreErrors" : 0,
  "maxConnectRetries" : 10,
  "chunkSize" : 0,
  "autoStart" : false,
  "adaptivePolling" : true,
  "includeSystem" : true,
  "requireFromPresent" : false,
  "autoResync" : false,
  "autoResyncRetries" : 2,
  "verbose" : false
}
```

Note: There is no *endpoint* attribute configured yet. The *endpoint* attribute is required for the replication applier to be startable. You may also want to configure a username and password for the connection via the *username* and *password* attributes.

```
require("@arangodb/replication").applier.properties({
  endpoint: "tcp://master.domain.org:8529",
  username:  "root",
  password: "secret",
  verbose: false
});
```

This will re-configure the replication applier for the current database. The configuration will be used from the next start of the replication applier. The replication applier cannot be re-configured while it is running. It must be stopped first to be re-configured.

To make the replication applier of the current database start automatically when the ArangoDB server starts, use the *autoStart* attribute.

Setting the *adaptivePolling* attribute to *true* will make the replication applier poll the master database for changes with a variable frequency. The replication applier will then lower the frequency when the master is idle, and increase it when the master can provide new events). Otherwise the replication applier will poll the master database for changes with a constant frequency.

The *idleMinWaitTime* attribute controls the minimum wait time (in seconds) that the replication applier will intentionally idle before fetching more log data from the master in case the master has already sent all its log data. This wait time can be used to control the frequency with which the replication applier sends HTTP log fetch requests to the master in case there is no write activity on the master.

The *idleMaxWaitTime* attribute controls the maximum wait time (in seconds) that the replication applier will intentionally idle before fetching more log data from the master in case the master has already sent all its log data and there have been previous log fetch attempts that resulted in no more log data. This wait time can be used to control the maximum frequency with which the replication applier sends HTTP log fetch requests to the master in case there is no write activity on the master for longer periods. Note that this configuration value will only be used if the option *adaptivePolling* is set to *true*.

To set a timeout for connection and following request attempts, use the *connectTimeout* and *requestTimeout* values. The *maxConnectRetries* attribute configures after how many failed connection attempts in a row the replication applier will give up and turn itself off. You may want to set this to a high value so that temporary network outages do not lead to the replication applier stopping itself. The *connectRetryWaitTime* attribute configures how long the replication applier will wait before retrying the connection to the master in case of connection problems.

The *chunkSize* attribute can be used to control the approximate maximum size of a master's response (in bytes). Setting it to a low value may make the master respond faster (less data is assembled before the master sends the response), but may require more request-response roundtrips. Set it to *0* to use ArangoDB's built-in default value.

The *includeSystem* attribute controls whether changes to system collections (such as *_graphs* or *_users*) should be applied. If set to *true*, changes in these collections will be replicated, otherwise, they will not be replicated. It is often not necessary to replicate data from system collections, especially because it may lead to confusion on the slave because the slave needs to have its own system collections in order to start and keep operational.

The *requireFromPresent* attribute controls whether the applier will start synchronizing in case it detects that the master cannot provide data for the initial tick value provided by the slave. This may be the case if the master does not have a big enough backlog of historic WAL logfiles, and when the replication is re-started after a longer pause. When *requireFromPresent* is set to *true*, then the replication applier will check at start whether the start tick from which it starts or resumes replication is still present on the master. If not, then there would be data loss. If *requireFromPresent* is *true*, the replication applier will abort with an appropriate error message. If set to *false*, then the replication applier will still start, and ignore the data loss.

The *autoResync* option can be used in conjunction with the *requireFromPresent* option as follows: when both *requireFromPresent* and *autoResync* are set to *true* and the master cannot provide the log data the slave requests, the replication applier will stop as usual. But due to the fact that *autoResync* is set to true, the slave will automatically trigger a full resync of all data with the master. After that, the replication applier will go into continuous replication mode again. Additionally, setting *autoResync* to *true* will trigger a full re-synchronization of data when the continuous replication is started and detects that there is no start tick value.

Automatic re-synchronization may transfer a lot of data from the master to the slave and can be expensive. It is therefore turned off by default. When turned off, the slave will never perform an automatic re-synchronization with the master.

The *autoResyncRetries* option can be used to control the number of resynchronization retries that will be performed in a row when automatic resynchronization is enabled and kicks in. Setting this to *0* will effectively disable *autoResync*. Setting it to some other value will limit the number of retries that are performed. This helps preventing endless retries in case resynchronizations always fail.

The *verbose* attribute controls the verbosity of the replication logger. Setting it to `true` will make the replication applier write a line to the log for every operation it performs. This should only be used for diagnosing replication problems.

The following example will set most of the discussed properties for the current database's applier:

```
require("@arangodb/replication").applier.properties({
  endpoint: "tcp://master.domain.org:8529",
  username: "root",
  password: "secret",
  adaptivePolling: true,
  connectTimeout: 15,
  maxConnectRetries: 100,
  chunkSize: 262144,
  autoStart: true,
  includeSystem: true,
  autoResync: true,
  autoResyncRetries: 2,
});
```

After the applier is now fully configured, it could theoretically be started. However, we may first need an initial synchronization of all collections and their data from the master before we start the replication applier.

The only safe method for doing a full synchronization (or re-synchronization) is thus to

- stop the replication applier on the slave (if currently running)
- perform an initial full sync with the master database
- note the master database's *lastLogTick* value and
- start the continuous replication applier on the slave using this tick value.

The initial synchronization for the current database is executed with the *sync* command:

```
require("@arangodb/replication").sync({
  endpoint: "tcp://master.domain.org:8529",
  username: "root",
  password: "secret,
  includeSystem: true
});
```

The *includeSystem* option controls whether data from system collections (such as *_graphs* and *_users*) shall be synchronized.

The initial synchronization can optionally be configured to include or exclude specific collections using the *restrictType* and *restrictCollection* parameters.

The following command only synchronizes collection *foo* and *bar*:

```
require("@arangodb/replication").sync({
  endpoint: "tcp://master.domain.org:8529",
  username: "root",
  password: "secret,
  restrictType: "include",
  restrictCollections: [ "foo", "bar" ]
});
```

Using a *restrictType* of *exclude*, all collections but the specified will be synchronized.

Warning: *sync* will do a full synchronization of the collections in the current database with collections present in the master database. Any local instances of the collections and all their data are removed! Only execute this command if you are sure you want to remove the local data!

As *sync* does a full synchronization, it might take a while to execute. When *sync* completes successfully, it returns an array of collections it has synchronized in its *collections* attribute. It will also return the master database's last log tick value at the time the *sync* was started on the master. The tick value is contained in the *lastLogTick* attribute of the *sync* command:

```
{
  "lastLogTick" : "231848833079705",
  "collections" : [ ... ]
}
```

Now you can start the continuous synchronization for the current database on the slave with the command

```
require("@arangodb/replication").applier.start("231848833079705");
```

Note: The tick values should be treated as strings. Using numeric data types for tick values is unsafe because they might exceed the 32 bit value and the IEEE754 double accuracy ranges.

Example Setup

Setting up a working master-slave replication requires two ArangoDB instances:

- **master**: this is the instance that all data-modification operations should be directed to
- **slave**: on this instance, we'll start a replication applier, and this will fetch data from the master database's write-ahead log and apply its operations locally

For the following example setup, we'll use the instance *tcp://master.domain.org:8529* as the master, and the instance *tcp://slave.domain.org:8530* as a slave.

The goal is to have all data from the database *_system* on master *tcp://master.domain.org:8529* be replicated to the database *_system* on the slave *tcp://slave.domain.org:8530*.

On the **master**, nothing special needs to be done, as all write operations will automatically be logged in the master's write-ahead log (WAL).

All-in-one setup

To make the replication copy the initial data from the **master** to the **slave** and start the continuous replication on the **slave**, there is an all-in-one command:

```
require("@arangodb/replication").setupReplication(configuration);
```

The following example demonstrates how to use the command for setting up replication for the *_system* database. Note that it should be run on the **slave** and not the **master**:

```
db._useDatabase("_system");
require("@arangodb/replication").setupReplication({
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd",
  verbose: false,
  includeSystem: false,
  incremental: true,
  autoResync: true
});
```

The command will return when the initial synchronization is finished and the continuous replication has been started, or in case the initial synchronization has failed.

If the initial synchronization is successful, the command will store the given configuration on the slave. It also configures the continuous replication to start automatically if the slave is restarted, i.e. *autoStart* is set to *true*.

If the command is run while the slave's replication applier is already running, it will first stop the running applier, drop its configuration and do a resynchronization of data with the **master**. It will then use the provided configration, overwriting any previously existing replication configuration on the **slave**.

Initial synchronization

The initial synchronization and continuous replication applier can also be started separately. To start replication on the **slave**, make sure there currently is no replication applier running.

The following commands stop a running applier in the slave's *_system* database:

```
db._useDatabase("_system");
require("@arangodb/replication").applier.stop();
```

The *stop* operation will terminate any replication activity in the _system database on the slave.

After that, the initial synchronization can be run. It will copy the collections from the **master** to the **slave**, overwriting existing data. To run the initial synchronization, execute the following commands on the **slave**:

```
db._useDatabase("_system");
require("@arangodb/replication").sync({
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd",
  verbose: false
});
```

Username and password only need to be specified when the **master** requires authentication. To check what the synchronization is currently doing, supply set the *verbose* option to *true*. If set, the synchronization will create log messages with the current synchronization status.

Warning: The *sync* command will replace data in the **slave** database with data from the **master** database! Only execute these commands if you have verified you are on the correct server, in the correct database!

The sync operation will return an attribute named *lastLogTick* which we'll need to note. The last log tick will be used as the starting point for subsequent replication activity. Let's assume we got the following last log tick:

```
{
  "lastLogTick" : "40694126",
  ...
}
```

Initial synchronization from the ArangoShell

The initial synchronization via the *sync* command may take a long time to complete. The shell will block until the slave has completed the initial synchronization or until an error occurs. By default, the *sync* command in the ArangoShell will poll the slave for a status update every 10 seconds.

Optionally the *sync* command can be made non-blocking by setting its *async* option to true. In this case, the *sync command* will return instantly with an id string, and the initial synchronization will run detached on the master. To fetch the current status of the *sync* progress from the ArangoShell, the *getSyncResult* function can be used as follows:

```
db._useDatabase("_system");
var replication = require("@arangodb/replication");

/* run command in async mode */
var id = replication.sync({
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd",
  async: true
});

/* now query the status of our operation */
print(replication.getSyncResult(id));
```

getSyncResult will return *false* as long as the synchronization is not complete, and return the synchronization result otherwise.

Continuous synchronization

When the initial synchronization is finished, the continuous replication applier can be started using the last log tick provided by the *sync* command. Before starting it, there is at least one configuration option to consider: replication on the **slave** will be running until the **slave** gets shut down. When the slave server gets restarted, replication will be turned off again. To change this, we first need to configure the slave's replication applier and set its *autoStart* attribute.

Here's the command to configure the replication applier with several options, including the *autoStart* attribute:

```
db._useDatabase("_system");
require("@arangodb/replication").applier.properties({
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd",
  autoStart: true,
  autoResync: true,
  autoResyncRetries: 2,
  adaptivePolling: true,
  includeSystem: false,
  requireFromPresent: false,
  idleMinWaitTime: 0.5,
  idleMaxWaitTime: 1.5,
  verbose: false
});
```

An important consideration for replication is whether data from system collections (such as *_graphs* or *_users*) should be applied. The *includeSystem* option controls that. If set to *true*, changes in system collections will be replicated. Otherwise, they will not be replicated. It is often not necessary to replicate data from system collections, especially because it may lead to confusion on the slave because the slave needs to have its own system collections in order to start and keep operational.

The *requireFromPresent* attribute controls whether the applier will start synchronizing in case it detects that the master cannot provide data for the initial tick value provided by the slave. This may be the case if the master does not have a big enough backlog of historic WAL logfiles, and when the replication is re-started after a longer pause. When *requireFromPresent* is set to *true*, then the replication applier will check at start whether the start tick from which it starts or resumes replication is still present on the master. If not, then there would be data loss. If *requireFromPresent* is *true*, the replication applier will abort with an appropriate error message. If set to *false*, then the replication applier will still start, and ignore the data loss.

The *autoResync* option can be used in conjunction with the *requireFromPresent* option as follows: when both *requireFromPresent* and *autoResync* are set to *true* and the master cannot provide the log data the slave had requested, the replication applier will stop as usual. But due to the fact that *autoResync* is set to true, the slave will automatically trigger a full resync of all data with the master. After that, the replication applier will go into continuous replication mode again. Additionally, setting *autoResync* to *true* will trigger a full re-synchronization of data when the continuous replication is started and detects that there is no start tick value.

Note that automatic re-synchronization (*autoResync* option set to *true*) may transfer a lot of data from the master to the slave and can therefore be expensive. Still it's turned on here so there's less need for manual intervention.

The *autoResyncRetries* option can be used to control the number of resynchronization retries that will be performed in a row when automatic resynchronization is enabled and kicks in. Setting this to *0* will effectively disable *autoResync*. Setting it to some other value will limit the number of retries that are performed. This helps preventing endless retries in case resynchronizations always fail.

Now it's time to start the replication applier on the slave using the last log tick we got before:

```
db._useDatabase("_system");
require("@arangodb/replication").applier.start("40694126");
```

This will replicate all operations happening in the master's system database and apply them on the slave, too.

After that, you should be able to monitor the state and progress of the replication applier by executing the *state* command on the slave server:

```
    db._useDatabase("_system");
    require("@arangodb/replication").applier.state();
```

Please note that stopping the replication applier on the slave using the *stop* command should be avoided. The reason is that currently ongoing transactions (that have partly been replicated to the slave) will be need to be restarted after a restart of the replication applier. Stopping and restarting the replication applier on the slave should thus only be performed if there is certainty that the master is currently fully idle and all transactions have been replicated fully.

Note that while a slave has only partly executed a transaction from the master, it might keep a write lock on the collections involved in the transaction.

You may also want to check the master and slave states via the HTTP APIs (see HTTP Interface for Replication).

Syncing Collections

In order to synchronize data for a single collection from a master to a slave instance, there is the *syncCollection* function:

It will fetch all documents of the specified collection from the master database and store them in the local instance. After the synchronization, the collection data on the slave will be identical to the data on the master, provided no further data changes happen on the master. Any data changes that are performed on the master after the synchronization was started will not be captured by *syncCollection*, but need to be replicated using the regular replication applier mechanism.

For the following example setup, we'll use the instance *tcp://master.domain.org:8529* as the master, and the instance *tcp://slave.domain.org:8530* as a slave.

The goal is to have all data from the collection *test* in database *_system* on master *tcp://master.domain.org:8529* be replicated to the collection *test* in database *_system* on the slave *tcp://slave.domain.org:8530*.

On the **master**, the collection *test* needs to be present in the *_system* database, with any data in it.

To transfer this collection to the **slave**, issue the following commands there:

```
db._useDatabase("_system");
require("@arangodb/replication").syncCollection("test", {
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd"
});
```

Warning: The syncCollection command will replace the collection's data in the slave database with data from the master database! Only execute these commands if you have verified you are on the correct server, in the correct database!

Setting the optional *incremental* attribute in the call to *syncCollection* will start an incremental transfer of data. This may be useful in case when the slave already has parts or almost all of the data in the collection and only the differences need to be synchronized. Note that to compute the differences the incremental transfer will build a sorted list of all document keys in the collection on both the slave and the master, which may still be expensive for huge collections in terms of memory usage and runtime. During building the list of keys the collection will be read-locked on the master.

The *initialSyncMaxWaitTime* attribute in the call to *syncCollection* controls how long the slave will wait for a master's response. This wait time can be used to control after what time the synchronization will give up and fail.

The *syncCollection* command may take a long time to complete if the collection is big. The shell will block until the slave has synchronized the entire collection from the master or until an error occurs. By default, the *syncCollection* command in the ArangoShell will poll for a status update every 10 seconds.

When *syncCollection* is called from the ArangoShell, the optional *async* attribute can be used to start the synchronization as a background process on the slave. If *async* is set to *true*, the call to *syncCollection* will return almost instantly with an id string. Using this id string, the status of the sync job on the slave can be queried using the *getSyncResult* function as follows:

```
db._useDatabase("_system");
var replication = require("@arangodb/replication");

/* run command in async mode */
var id = replication.syncCollection("test", {
  endpoint: "tcp://master.domain.org:8529",
  username: "myuser",
  password: "mypasswd",
  async: true
});

/* now query the status of our operation */
print(replication.getSyncResult(id));
```

getSyncResult will return *false* as long as the synchronization is not complete, and return the synchronization result otherwise.

Replication Limitations

The replication in ArangoDB has a few limitations. Some of these limitations may be removed in later versions of ArangoDB:

- there is no feedback from the slaves to the master. If a slave cannot apply an event it got from the master, the master will have a different state of data. In this case, the replication applier on the slave will stop and report an error. Administrators can then either "fix" the problem or re-sync the data from the master to the slave and start the applier again.
- at the moment it is assumed that only the replication applier executes write operations on a slave. ArangoDB currently does not prevent users from carrying out their own write operations on slaves, though this might lead to undefined behavior and the replication applier stopping.
- when a replication slave asks a master for log events, the replication master will return all write operations for user-defined collections, but it will exclude write operations for certain system collections. The following collections are excluded intentionally from replication: *_apps*, *_trx*, *_replication*, *_configuration*, *_jobs*, *_queues*, *_sessions*, *_foxxlog* and all statistics collections. Write operations for the following system collections can be queried from a master: *_aqlfunctions*, *_graphs*, *_users*.
- Foxx applications consist of database entries and application scripts in the file system. The file system parts of Foxx applications are not tracked anywhere and thus not replicated in current versions of ArangoDB. To replicate a Foxx application, it is required to copy the application to the remote server and install it there using the *foxx-manager* utility.
- master servers do not know which slaves are or will be connected to them. All servers in a replication setup are currently only loosely coupled. There currently is no way for a client to query which servers are present in a replication.
- when not using our mesos integration failover must be handled by clients or client APIs.
- there currently is one replication applier per ArangoDB database. It is thus not possible to have a slave apply operations from multiple masters into the same target database.
- replication is set up on a per-database level. When using ArangoDB with multiple databases, replication must be configured individually for each database.
- the replication applier is single-threaded, but write operations on the master may be executed in parallel if they affect different collections. Thus the replication applier might not be able to catch up with a very powerful and loaded master.
- replication is only supported between the two ArangoDB servers running the same ArangoDB version. It is currently not possible to replicate between different ArangoDB versions.
- a replication applier cannot apply data from itself.

Synchronous Replication

At its core synchronous replication will replicate write operations to multiple hosts. This feature is only available when operating ArangoDB in a cluster. Whenever a coordinator executes a sychronously replicated write operation it will only be reported to be successful if it was carried out on all replicas. In contrast to multi master replication setups known from other systems ArangoDB's synchronous operation guarantees a consistent state across the cluster.

Implementation

Architecture inside the cluster

Synchronous replication can be configured per collection via the property *replicationFactor*. Synchronous replication requires a cluster to operate.

Whenever you specify a *replicationFactor* greater than 1 when creating a collection, synchronous replication will be activated for this collection. The cluster will determine suitable *leaders* and *followers* for every requested shard (*numberOfShards*) within the cluster. When requesting data of a shard only the current leader will be asked whereas followers will only keep their copy in sync. This is due to the current implementation of transactions.

Using *synchronous replication* alone will guarantee consistency and high availabilty at the cost of reduced performance: Write requests will have a higher latency (due to every write-request having to be executed on the followers) and read requests won't scale out as only the leader is being asked.

In a cluster synchronous replication will be managed by the *coordinators* for the client. The data will always be stored on *primaries*.

The following example will give you an idea of how synchronous operation has been implemented in ArangoDB.

1. Connect to a coordinator via arangosh
2. Create a collection

   127.0.0.1:8530@_system> db._create("test", {"replicationFactor": 2})

3. the coordinator will figure out a *leader* and 1 *follower* and create 1 *shard* (as this is the default)

4. Insert data

   127.0.0.1:8530@_system> db.test.insert({"replication": "😎"})

5. The coordinator will write the data to the leader, which in turn will replicate it to the follower.

6. Only when both were successful the result is reported to be successful

   {

   ```
   "_id" : "test/7987",
   "_key" : "7987",
   "_rev" : "7987"
   ```

   }

   When a follower fails, the leader will give up on it after 3 seconds and proceed with the operation. As soon as the follower (or the network connection to the leader) is back up, the two will resynchronize and synchronous replication is resumed. This happens all transparently to the client.

The current implementation of ArangoDB does not allow changing the replicationFactor later. This is subject to change. In the meantime the only way is to dump and restore the collection. A cookbook recipe for this can be found here: https://docs.arangodb.com/cookbook/Administration/Migrate2.8to3.0.html#controling-the-number-of-shards-and-the-replication-factor

Automatic failover

Whenever the leader of a shard is failing and there is a query trying to access data of that shard the coordinator will continue trying to contact the leader until it timeouts. The internal cluster supervision will check cluster health every few seconds and will take action if there is no heartbeat from a server for 15 seconds. If the leader doesn't come back in time the supervision will reorganize the cluster by promoting for each shard a follower that is in sync with its leader to be the new leader. From then on the coordinators will contact the new leader.

The process is best outlined using an example:

1. The leader of a shard (lets name it DBServer001) is going down.
2. A coordinator is asked to return a document of a shard DBServer001 is managing:

127.0.0.1:8530@_system> db.test.document("100069")

3. The coordinator tries to contact the leader (DBServer001) and timeouts.
4. The coordinator retries to contact the leader (DBServer001) and timeouts.
5. The supervision detects outage of DBServer001.
6. The supervision promotes one of the followers (say DBServer002) that is in sync to be leader and makes DBServer001 a follower.
7. The coordinator retries to contact the leader (DBServer002) and returns the result:

   {

   ```
     "_key" : "100069",
     "_id" : "test/100069",
     "_rev" : "513",
     "replication" : "😎"
   ```

   }

8. After a while the supervision declares DBServer001 to be completely dead.
9. A new follower is determined from the pool of DBservers.
10. The new follower syncs its data from the leader and order is restored.

Please note that there may still be timeouts. Depending on when exactly the request has been done (in regard to the supervision) and depending on the time needed to reconfigure the cluster the coordinator might fail with a timeout error!

Configuration

Requirements

Synchronous replication requires an operational ArangoDB cluster.

Enabling synchronous replication

Synchronous replication can be enabled per collection. When creating you can specify the number of replicas using *replicationFactor*. The default is `1` which effectively *disables* synchronous replication.

Example:

```
127.0.0.1:8530@_system> db._create("test", {"replicationFactor": 3})
```

Any write operation will require 2 replicas to report success from now on.

Sharding

ArangoDB is organizing its collection data in shards. Sharding allows to use multiple machines to run a cluster of ArangoDB instances that together constitute a single database. This enables you to store much more data, since ArangoDB distributes the data automatically to the different servers. In many situations one can also reap a benefit in data throughput, again because the load can be distributed to multiple machines.

Shards are configured per collection so multiple shards of data form the collection as a whole. To determine in which shard the data is to be stored ArangoDB performs a hash across the values. By default this hash is being created from _key.

To configure the number of shards:

```
127.0.0.1:8529@_system> db._create("sharded_collection", {"numberOfShards": 4});
```

To configure the hashing for another attribute:

```
127.0.0.1:8529@_system> db._create("sharded_collection", {"numberOfShards": 4, "shardKeys": ["country"]});
```

This would be useful to keep data of every country in one shard which would result in better performance for queries working on a per country base. You can also specify multiple `shardKeys`. Note however that if you change the shard keys from their default `["_key"]`, then finding a document in the collection by its primary key involves a request to every single shard. Furthermore, in this case one can no longer prescribe the primary key value of a new document but must use the automatically generated one. This latter restriction comes from the fact that ensuring uniqueness of the primary key would be very inefficient if the user could specify the primary key.

On which node in a cluster a particular shard is kept is undefined. There is no option to configure an affinity based on certain shard keys.

Unique indexes (hash, skiplist, persistent) on sharded collections are only allowed if the fields used to determine the shard key are also included in the list of attribute paths for the index:

shardKeys	indexKeys	
a	a	ok
a	b	not ok
a	a, b	ok
a, b	a	not ok
a, b	b	not ok
a, b	a, b	ok
a, b	a, b, c	ok
a, b, c	a, b	not ok
a, b, c	a, b, c	ok

General Upgrade Information

Recommended major upgrade procedure

To upgrade an existing ArangoDB 2.x to 3.0 please use the procedure described here.

Recommended minor upgrade procedure

To upgrade an existing ArangoDB database to a newer version of ArangoDB (e.g. 3.0 to 3.1, or 3.3 to 3.4), the following method is recommended:

- Check the *CHANGELOG* and the list of incompatible changes for API or other changes in the new version of ArangoDB and make sure your applications can deal with them
- Stop the "old" arangod service or binary
- Copy the entire "old" data directory to a safe place (that is, a backup)
- Install the new version of ArangoDB and start the server with the *--database.auto-upgrade* option once. This might write to the logfile of ArangoDB, so you may want to check the logs for any issues before going on.
- Start the "new" arangod service or binary regularly and check the logs for any issues. When you're confident everything went well, you may want to check the database directory for any files with the ending *.old*. These files are created by ArangoDB during upgrades and can be safely removed manually later.

If anything goes wrong during or shortly after the upgrade:

- Stop the "new" arangod service or binary
- Revert to the "old" arangod binary and restore the "old" data directory
- Start the "old" version again

It is not supported to use datafiles created or modified by a newer version of ArangoDB with an older ArangoDB version. For example, it is unsupported and is likely to cause problems when using 2.3 datafiles with an ArangoDB 2.2 instance.

Upgrading to ArangoDB 3.1

Please read the following sections if you upgrade from a previous version to ArangoDB 3.1. Please be sure that you have checked the list of changes in 3.1 before upgrading.

Upgrading to ArangoDB 3.0

Please read the following sections if you upgrade from a previous version to ArangoDB 3.0. Please be sure that you have checked the list of changes in 3.0 before upgrading.

Migrating databases and collections from ArangoDB 2.8 to 3.0

ArangoDB 3.0 does not provide an automatic update mechanism for database directories created with the 2.x branches of ArangoDB.

In order to migrate data from ArangoDB 2.8 (or an older 2.x version) into ArangoDB 3.0, it is necessary to export the data from 2.8 using `arangodump`, and then import the dump into a fresh ArangoDB 3.0 with `arangorestore`.

To do this, first run the 2.8 version of `arangodump` to export the database data into a directory. `arangodump` will dump the `_system` database by default. In order to make it dump multiple databases, it needs to be invoked once per source database, e.g.

```
# in 2.8
arangodump --server.database _system --output-directory dump-system
arangodump --server.database mydb --output-directory dump-mydb
...
```

That will produce a dump directory for each database that `arangodump` is called for. If the server has authentication turned on, it may be necessary to provide the required credentials when invoking `arangodump`, e.g.

```
arangodump --server.database _system --server.username myuser --server.password mypasswd --output-directory dump-system
```

The dumps produced by `arangodump` can now be imported into ArangoDB 3.0 using the 3.0 version of `arangodump`:

```
# in 3.0
arangorestore --server.database _system --input-directory dump-system
arangorestore --server.database mydb --input-directory dump-mydb
...
```

arangorestore will by default fail if the target database does not exist. It can be told to create it automatically using the option `--create-database true`:

```
arangorestore --server.database mydb --create-database true --input-directory dump-mydb
```

And again it may be required to provide access credentials when invoking `arangorestore`:

```
arangorestore --server.database mydb --create-database true --server.username myuser --server.password mypasswd --input-
```

Please note that the version of dump/restore should match the server version, i.e. it is required to dump the original data with the 2.8 version of `arangodump` and restore it with the 3.0 version of `arangorestore`.

After that the 3.0 instance of ArangoDB will contain the databases and collections that were present in the 2.8 instance.

Adjusting authentication info

Authentication information was stored per database in ArangoDB 2.8, meaning there could be different users and access credentials per database. In 3.0, the users are stored in a central location in the `_system` database. To use the same user setup as in 2.8, it may be required to create extra users and/or adjust their permissions.

In order to do that, please connect to the 3.0 instance with an ArangoShell (this will connect to the `_system` database by default):

```
arangosh --server.username myuser --server.password mypasswd
```

Use the following commands to create a new user with some password and grant them access to a specific database

```
require("@arangodb/users").save(username, password, true);
require("@arangodb/users").grantDatabase(username, databaseName, "rw");
```

For example, to create a user `myuser` with password `mypasswd` and give them access to databases `mydb1` and `mydb2`, the commands would look as follows:

```
require("@arangodb/users").save("myuser", "mypasswd", true);
require("@arangodb/users").grantDatabase("myuser", "mydb1", "rw");
require("@arangodb/users").grantDatabase("myuser", "mydb2", "rw");
```

Existing users can also be updated, removed or listed using the following commands:

```
/* update user myuser with password mypasswd */
require("@arangodb/users").update("myuser", "mypasswd", true);

/* remove user myuser */
require("@arangodb/users").remove("myuser");

/* list all users */
require("@arangodb/users").all();
```

Foxx applications

The dump/restore procedure described above will not export and re-import Foxx applications. In order to move these from 2.8 to 3.0, Foxx applications should be exported as zip files via the 2.8 web interface.

The zip files can then be uploaded in the "Services" section in the ArangoDB 3.0 web interface. Applications may need to be adjusted manually to run in 3.0. Please consult the migration guide for Foxx apps.

An alternative way of moving Foxx apps into 3.0 is to copy the source directory of a 2.8 Foxx application manually into the 3.0 Foxx apps directory for the target database (which is normally `/var/lib/arangodb3-apps/_db/<dbname>/` but the exact location is platform-specific).

Upgrading to ArangoDB 2.8

Please read the following sections if you upgrade from a previous version to ArangoDB 2.8. Please be sure that you have checked the list of changes in 2.8 before upgrading.

Please note first that a database directory used with ArangoDB 2.8 cannot be used with earlier versions (e.g. ArangoDB 2.7) any more. Upgrading a database directory cannot be reverted. Therefore please make sure to create a full backup of your existing ArangoDB installation before performing an upgrade.

Database Directory Version Check and Upgrade

ArangoDB will perform a database version check at startup. When ArangoDB 2.8 encounters a database created with earlier versions of ArangoDB, it will refuse to start. This is intentional.

The output will then look like this:

```
2015-12-04T17:11:17Z [31432] ERROR In database '_system': Database directory version (20702) is lower than current versi
2015-12-04T17:11:17Z [31432] ERROR In database '_system': -------------------------------------------------------------
2015-12-04T17:11:17Z [31432] ERROR In database '_system': It seems like you have upgraded the ArangoDB binary.
2015-12-04T17:11:17Z [31432] ERROR In database '_system': If this is what you wanted to do, please restart with the
2015-12-04T17:11:17Z [31432] ERROR In database '_system':   --upgrade
2015-12-04T17:11:17Z [31432] ERROR In database '_system': option to upgrade the data in the database directory.
2015-12-04T17:11:17Z [31432] ERROR In database '_system': Normally you can use the control script to upgrade your databa
2015-12-04T17:11:17Z [31432] ERROR In database '_system':   /etc/init.d/arangodb stop
2015-12-04T17:11:17Z [31432] ERROR In database '_system':   /etc/init.d/arangodb upgrade
2015-12-04T17:11:17Z [31432] ERROR In database '_system':   /etc/init.d/arangodb start
2015-12-04T17:11:17Z [31432] ERROR In database '_system': -------------------------------------------------------------
2015-12-04T17:11:17Z [31432] FATAL Database '_system' needs upgrade. Please start the server with the --upgrade option
```

To make ArangoDB 2.8 start with a database directory created with an earlier ArangoDB version, you may need to invoke the upgrade procedure once. This can be done by running ArangoDB from the command line and supplying the `--upgrade` option.

Note: here the same database should be specified that is also specified when arangod is started regularly. Please do not run the `--upgrade` command on each individual database subfolder (named `database-<some number>`).

For example, if you regularly start your ArangoDB server with

```
unix> arangod mydatabasefolder
```

then running

```
unix> arangod mydatabasefolder --upgrade
```

will perform the upgrade for the whole ArangoDB instance, including all of its databases.

Starting with `--upgrade` will run a database version check and perform any necessary migrations. As usual, you should create a backup of your database directory before performing the upgrade.

The last line of the output should look like this:

```
2015-12-04T17:12:15Z [31558] INFO database upgrade passed
```

Please check the full output the `--upgrade` run. Upgrading may produce errors, which need to be fixed before ArangoDB can be used properly. If no errors are present or they have been resolved manually, you can start ArangoDB 2.8 regularly.

Upgrading a cluster planned in the web interface

A cluster of ArangoDB instances has to be upgraded as well. This involves upgrading all ArangoDB instances in the cluster, as well as running the version check on the whole running cluster in the end.

We have tried to make this procedure as painless and convenient for you. We assume that you planned, launched and administrated a cluster using the graphical front end in your browser. The upgrade procedure is then as follows:

1. First shut down your cluster using the graphical front end as usual.
2. Then upgrade all dispatcher instances on all machines in your cluster using the version check as described above and restart them.
3. Now open the cluster dash board in your browser by pointing it to the same dispatcher that you used to plan and launch the cluster in the graphical front end. In addition to the usual buttons "Relaunch", "Edit cluster plan" and "Delete cluster plan" you will see another button marked "Upgrade and relaunch cluster".
4. Hit this button, your cluster will be upgraded and launched and all is done for you behind the scenes. If all goes well, you will see the usual cluster dash board after a few seconds. If there is an error, you have to inspect the log files of your cluster ArangoDB instances. Please let us know if you run into problems.

There is an alternative way using the `ArangoDB` shell. Instead of steps 3. and 4. above you can launch `arangosh` , point it to the dispatcher that you have used to plan and launch the cluster using the option `--server.endpoint` , and execute

```
arangosh> require("org/arangodb/cluster").Upgrade("root","");
```

This upgrades the cluster and launches it, exactly as with the button above in the graphical front end. You have to replace `"root"` with a user name and `""` with a password that is valid for authentication with the cluster.

Upgrading Foxx apps generated by ArangoDB 2.7 and earlier

The implementation of the `require` function used to import modules in ArangoDB and Foxx has changed in order to improve compatibility with Node.js modules.

Given an app/service with the following layout:

- manifest.json
- controllers/
  - todos.js
- models/
  - todo.js
- repositories/
  - todos.js
- node_modules/
  - models/
    - todo.js

The file `controllers/todos.js` would previously contain the following `require` calls:

```
var _ = require('underscore');
var joi = require('joi');
var Foxx = require('org/arangodb/foxx');
var ArangoError = require('org/arangodb').ArangoError;
var Todos = require('repositories/todos'); // <-- !
var Todo = require('models/todo'); // <-- !
```

The require paths `repositories/todos` and `models/todo` were previously resolved locally as relative to the app root.

Starting with 2.8 these paths would instead be resolved as relative to the `node_modules` folder or the global ArangoDB module paths before being resolved locally as a fallback.

In the given example layout the app would break in 2.8 because the module name `models/todo` would always resolve to `node_modules/models/todo.js` (which previously would have been ignored) instead of the local `models/todo.js` .

In order to make sure the app still works in 2.8, the require calls in `controllers/todos.js` would need to be adjusted to look like this:

```
var _ = require('underscore');
var joi = require('joi');
var Foxx = require('org/arangodb/foxx');
var ArangoError = require('org/arangodb').ArangoError;
var Todos = require('../repositories/todos'); // <-- !
var Todo = require('../models/todo'); // <-- !
```

Note that the old "global" style require calls may still work in 2.8 but may break unexpectedly if modules with matching names are installed globally.

Upgrading to ArangoDB 2.6

Please read the following sections if you upgrade from a previous version to ArangoDB 2.6. Please be sure that you have checked the list of changes in 2.6 before upgrading.

Please note first that a database directory used with ArangoDB 2.6 cannot be used with earlier versions (e.g. ArangoDB 2.5) any more. Upgrading a database directory cannot be reverted. Therefore please make sure to create a full backup of your existing ArangoDB installation before performing an upgrade.

Database Directory Version Check and Upgrade

ArangoDB will perform a database version check at startup. When ArangoDB 2.6 encounters a database created with earlier versions of ArangoDB, it will refuse to start. This is intentional.

The output will then look like this:

```
2015-02-17T09:43:11Z [8302] ERROR In database '_system': Database directory version (20501) is lower than current versio
2015-02-17T09:43:11Z [8302] ERROR In database '_system': --------------------------------------------------------------
2015-02-17T09:43:11Z [8302] ERROR In database '_system': It seems like you have upgraded the ArangoDB binary.
2015-02-17T09:43:11Z [8302] ERROR In database '_system': If this is what you wanted to do, please restart with the
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   --upgrade
2015-02-17T09:43:11Z [8302] ERROR In database '_system': option to upgrade the data in the database directory.
2015-02-17T09:43:11Z [8302] ERROR In database '_system': Normally you can use the control script to upgrade your databas
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   /etc/init.d/arangodb stop
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   /etc/init.d/arangodb upgrade
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   /etc/init.d/arangodb start
2015-02-17T09:43:11Z [8302] ERROR In database '_system': --------------------------------------------------------------
2015-02-17T09:43:11Z [8302] FATAL Database '_system' needs upgrade. Please start the server with the --upgrade option
```

To make ArangoDB 2.6 start with a database directory created with an earlier ArangoDB version, you may need to invoke the upgrade procedure once. This can be done by running ArangoDB from the command line and supplying the `--upgrade` option.

Note: here the same database should be specified that is also specified when arangod is started regularly. Please do not run the `--upgrade` command on each individual database subfolder (named `database-<some number>`).

For example, if you regularly start your ArangoDB server with

```
unix> arangod mydatabasefolder
```

then running

```
unix> arangod mydatabasefolder --upgrade
```

will perform the upgrade for the whole ArangoDB instance, including all of its databases.

Starting with `--upgrade` will run a database version check and perform any necessary migrations. As usual, you should create a backup of your database directory before performing the upgrade.

The last line of the output should look like this:

```
2014-12-22T12:03:31Z [12026] INFO database upgrade passed
```

Please check the full output the `--upgrade` run. Upgrading may produce errors, which need to be fixed before ArangoDB can be used properly. If no errors are present or they have been resolved manually, you can start ArangoDB 2.6 regularly.

Upgrading a cluster planned in the web interface

A cluster of ArangoDB instances has to be upgraded as well. This involves upgrading all ArangoDB instances in the cluster, as well as running the version check on the whole running cluster in the end.

We have tried to make this procedure as painless and convenient for you. We assume that you planned, launched and administrated a cluster using the graphical front end in your browser. The upgrade procedure is then as follows:

1. First shut down your cluster using the graphical front end as usual.
2. Then upgrade all dispatcher instances on all machines in your cluster using the version check as described above and restart them.
3. Now open the cluster dash board in your browser by pointing it to the same dispatcher that you used to plan and launch the cluster in the graphical front end. In addition to the usual buttons "Relaunch", "Edit cluster plan" and "Delete cluster plan" you will see another button marked "Upgrade and relaunch cluster".
4. Hit this button, your cluster will be upgraded and launched and all is done for you behind the scenes. If all goes well, you will see the usual cluster dash board after a few seconds. If there is an error, you have to inspect the log files of your cluster ArangoDB instances. Please let us know if you run into problems.

There is an alternative way using the `ArangoDB` shell. Instead of steps 3. and 4. above you can launch `arangosh`, point it to the dispatcher that you have used to plan and launch the cluster using the option `--server.endpoint`, and execute

```
arangosh> require("org/arangodb/cluster").Upgrade("root","");
```

This upgrades the cluster and launches it, exactly as with the button above in the graphical front end. You have to replace `"root"` with a user name and `""` with a password that is valid for authentication with the cluster.

Upgrading to ArangoDB 2.5

Please read the following sections if you upgrade from a previous version to ArangoDB 2.5. Please be sure that you have checked the list of changes in 2.5 before upgrading.

Please note first that a database directory used with ArangoDB 2.5 cannot be used with earlier versions (e.g. ArangoDB 2.4) any more. Upgrading a database directory cannot be reverted. Therefore please make sure to create a full backup of your existing ArangoDB installation before performing an upgrade.

In 2.5 we have also changed the paths for Foxx applications. Please also make sure that you have a backup of all Foxx apps in your `javascript.app-path` and `javascript.dev-app-path` . It is sufficient to have the source files for Foxx somewhere else so you can reinstall them on error. To check that everything has worked during upgrade you could use the web-interface Applications tab or

```
unix> foxx-manager list
```

for all your databases. The listed apps should be identical before and after the upgrade.

Database Directory Version Check and Upgrade

ArangoDB will perform a database version check at startup. When ArangoDB 2.5 encounters a database created with earlier versions of ArangoDB, it will refuse to start. This is intentional.

The output will then look like this:

```
2015-02-17T09:43:11Z [8302] ERROR In database '_system': Database directory version (20401) is lower than current versio
2015-02-17T09:43:11Z [8302] ERROR In database '_system': ---------------------------------------------------------------
2015-02-17T09:43:11Z [8302] ERROR In database '_system': It seems like you have upgraded the ArangoDB binary.
2015-02-17T09:43:11Z [8302] ERROR In database '_system': If this is what you wanted to do, please restart with the
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   --upgrade
2015-02-17T09:43:11Z [8302] ERROR In database '_system': option to upgrade the data in the database directory.
2015-02-17T09:43:11Z [8302] ERROR In database '_system': Normally you can use the control script to upgrade your databas
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   /etc/init.d/arangodb stop
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   /etc/init.d/arangodb upgrade
2015-02-17T09:43:11Z [8302] ERROR In database '_system':   /etc/init.d/arangodb start
2015-02-17T09:43:11Z [8302] ERROR In database '_system': ---------------------------------------------------------------
2015-02-17T09:43:11Z [8302] FATAL Database '_system' needs upgrade. Please start the server with the --upgrade option
```

To make ArangoDB 2.5 start with a database directory created with an earlier ArangoDB version, you may need to invoke the upgrade procedure once. This can be done by running ArangoDB from the command line and supplying the `--upgrade` option. Note: We have changed Foxx folder structure and implemented an upgrade task to move your applications to the new structure. In order to tell this upgrade task to also move your development Foxx apps please make sure you give the dev-app-path as well. If you have not used development mode for Foxx apps you can drop the `--javascript.dev-app-path` . It is only possible to upgrade one dev-app-path together with one data folder.

```
unix> arangod data --upgrade --javascript.dev-app-path devapps
```

where `data` is ArangoDB's main data directory and `devapps` is the directory where you develop Foxx apps.

Note: here the same database should be specified that is also specified when arangod is started regularly. Please do not run the `--upgrade` command on each individual database subfolder (named `database-<some number>`).

For example, if you regularly start your ArangoDB server with

```
unix> arangod mydatabasefolder
```

then running

```
unix> arangod mydatabasefolder --upgrade
```

will perform the upgrade for the whole ArangoDB instance, including all of its databases.

Starting with `--upgrade` will run a database version check and perform any necessary migrations. As usual, you should create a backup of your database directory before performing the upgrade.

The last line of the output should look like this:

```
2014-12-22T12:03:31Z [12026] INFO database upgrade passed
```

Please check the full output the `--upgrade` run. Upgrading may produce errors, which need to be fixed before ArangoDB can be used properly. If no errors are present or they have been resolved manually, you can start ArangoDB 2.5 regularly.

Upgrading a cluster planned in the web interface

A cluster of ArangoDB instances has to be upgraded as well. This involves upgrading all ArangoDB instances in the cluster, as well as running the version check on the whole running cluster in the end.

We have tried to make this procedure as painless and convenient for you. We assume that you planned, launched and administrated a cluster using the graphical front end in your browser. The upgrade procedure is then as follows:

1. First shut down your cluster using the graphical front end as usual.
2. Then upgrade all dispatcher instances on all machines in your cluster using the version check as described above and restart them.
3. Now open the cluster dash board in your browser by pointing it to the same dispatcher that you used to plan and launch the cluster in the graphical front end. In addition to the usual buttons "Relaunch", "Edit cluster plan" and "Delete cluster plan" you will see another button marked "Upgrade and relaunch cluster".
4. Hit this button, your cluster will be upgraded and launched and all is done for you behind the scenes. If all goes well, you will see the usual cluster dash board after a few seconds. If there is an error, you have to inspect the log files of your cluster ArangoDB instances. Please let us know if you run into problems.

There is an alternative way using the `ArangoDB` shell. Instead of steps 3. and 4. above you can launch `arangosh`, point it to the dispatcher that you have used to plan and launch the cluster using the option `--server.endpoint`, and execute

```
arangosh> require("org/arangodb/cluster").Upgrade("root","");
```

This upgrades the cluster and launches it, exactly as with the button above in the graphical front end. You have to replace `"root"` with a user name and `""` with a password that is valid for authentication with the cluster.

Upgrading to ArangoDB 2.4

Please read the following sections if you upgrade from a previous version to ArangoDB 2.4. Please be sure that you have checked the list of changes in 2.4 before upgrading.

Please note first that a database directory used with ArangoDB 2.4 cannot be used with earlier versions (e.g. ArangoDB 2.3) any more. Upgrading a database directory cannot be reverted. Therefore please make sure to create a full backup of your existing ArangoDB installation before performing an upgrade.

Database Directory Version Check and Upgrade

ArangoDB will perform a database version check at startup. When ArangoDB 2.4 encounters a database created with earlier versions of ArangoDB, it will refuse to start. This is intentional.

The output will then look like this:

```
2014-12-22T12:02:28Z [12001] ERROR In database '_system': Database directory version (20302) is lower than current versi
2014-12-22T12:02:28Z [12001] ERROR In database '_system': --------------------------------------------------------------
2014-12-22T12:02:28Z [12001] ERROR In database '_system': It seems like you have upgraded the ArangoDB binary.
2014-12-22T12:02:28Z [12001] ERROR In database '_system': If this is what you wanted to do, please restart with the
2014-12-22T12:02:28Z [12001] ERROR In database '_system':   --upgrade
2014-12-22T12:02:28Z [12001] ERROR In database '_system': option to upgrade the data in the database directory.
2014-12-22T12:02:28Z [12001] ERROR In database '_system': Normally you can use the control script to upgrade your databa
2014-12-22T12:02:28Z [12001] ERROR In database '_system':   /etc/init.d/arangodb stop
2014-12-22T12:02:28Z [12001] ERROR In database '_system':   /etc/init.d/arangodb upgrade
2014-12-22T12:02:28Z [12001] ERROR In database '_system':   /etc/init.d/arangodb start
2014-12-22T12:02:28Z [12001] ERROR In database '_system': --------------------------------------------------------------
2014-12-22T12:02:28Z [12001] FATAL Database '_system' needs upgrade. Please start the server with the --upgrade option
```

To make ArangoDB 2.4 start with a database directory created with an earlier ArangoDB version, you may need to invoke the upgrade procedure once. This can be done by running ArangoDB from the command line and supplying the `--upgrade` option:

```
unix> arangod data --upgrade
```

where `data` is ArangoDB's main data directory.

Note: here the same database should be specified that is also specified when arangod is started regularly. Please do not run the `--upgrade` command on each individual database subfolder (named `database-<some number>`).

For example, if you regularly start your ArangoDB server with

```
unix> arangod mydatabasefolder
```

then running

```
unix> arangod mydatabasefolder --upgrade
```

will perform the upgrade for the whole ArangoDB instance, including all of its databases.

Starting with `--upgrade` will run a database version check and perform any necessary migrations. As usual, you should create a backup of your database directory before performing the upgrade.

The last line of the output should look like this:

```
2014-12-22T12:03:31Z [12026] INFO database upgrade passed
```

Please check the full output the `--upgrade` run. Upgrading may produce errors, which need to be fixed before ArangoDB can be used properly. If no errors are present or they have been resolved manually, you can start ArangoDB 2.4 regularly.

Upgrading a cluster planned in the web interface

A cluster of ArangoDB instances has to be upgraded as well. This involves upgrading all ArangoDB instances in the cluster, as well as running the version check on the whole running cluster in the end.

We have tried to make this procedure as painless and convenient for you. We assume that you planned, launched and administrated a cluster using the graphical front end in your browser. The upgrade procedure is then as follows:

1. First shut down your cluster using the graphical front end as usual.
2. Then upgrade all dispatcher instances on all machines in your cluster using the version check as described above and restart them.
3. Now open the cluster dash board in your browser by pointing it to the same dispatcher that you used to plan and launch the cluster in the graphical front end. In addition to the usual buttons "Relaunch", "Edit cluster plan" and "Delete cluster plan" you will see another button marked "Upgrade and relaunch cluster".
4. Hit this button, your cluster will be upgraded and launched and all is done for you behind the scenes. If all goes well, you will see the usual cluster dash board after a few seconds. If there is an error, you have to inspect the log files of your cluster ArangoDB instances. Please let us know if you run into problems.

There is an alternative way using the `ArangoDB` shell. Instead of steps 3. and 4. above you can launch `arangosh`, point it to the dispatcher that you have used to plan and launch the cluster using the option `--server.endpoint`, and execute

```
arangosh> require("org/arangodb/cluster").Upgrade("root","");
```

This upgrades the cluster and launches it, exactly as with the button above in the graphical front end. You have to replace `"root"` with a user name and `""` with a password that is valid for authentication with the cluster.

Upgrading to ArangoDB 2.3

Please read the following sections if you upgrade from a previous version to ArangoDB 2.3. Please be sure that you have checked the list of changes in 2.3 before upgrading.

Please note first that a database directory used with ArangoDB 2.3 cannot be used with earlier versions (e.g. ArangoDB 2.2) any more. Upgrading a database directory cannot be reverted. Therefore please make sure to create a full backup of your existing ArangoDB installation before performing an upgrade.

Database Directory Version Check and Upgrade

ArangoDB will perform a database version check at startup. When ArangoDB 2.3 encounters a database created with earlier versions of ArangoDB, it will refuse to start. This is intentional.

The output will then look like this:

```
2014-11-03T15:48:06Z [2694] ERROR In database '_system': Database directory version (2.2) is lower than current version
2014-11-03T15:48:06Z [2694] ERROR In database '_system': ---------------------------------------------------------------
2014-11-03T15:48:06Z [2694] ERROR In database '_system': It seems like you have upgraded the ArangoDB binary.
2014-11-03T15:48:06Z [2694] ERROR In database '_system': If this is what you wanted to do, please restart with the
2014-11-03T15:48:06Z [2694] ERROR In database '_system':   --upgrade
2014-11-03T15:48:06Z [2694] ERROR In database '_system': option to upgrade the data in the database directory.
2014-11-03T15:48:06Z [2694] ERROR In database '_system': Normally you can use the control script to upgrade your databas
2014-11-03T15:48:06Z [2694] ERROR In database '_system':   /etc/init.d/arangodb stop
2014-11-03T15:48:06Z [2694] ERROR In database '_system':   /etc/init.d/arangodb upgrade
2014-11-03T15:48:06Z [2694] ERROR In database '_system':   /etc/init.d/arangodb start
2014-11-03T15:48:06Z [2694] ERROR In database '_system': ---------------------------------------------------------------
2014-11-03T15:48:06Z [2694] FATAL Database '_system' needs upgrade. Please start the server with the --upgrade option
```

To make ArangoDB 2.3 start with a database directory created with an earlier ArangoDB version, you may need to invoke the upgrade procedure once. This can be done by running ArangoDB from the command line and supplying the `--upgrade` option:

```
unix> arangod data --upgrade
```

where `data` is ArangoDB's main data directory.

Note: here the same database should be specified that is also specified when arangod is started regularly. Please do not run the `--upgrade` command on each individual database subfolder (named `database-<some number>`).

For example, if you regularly start your ArangoDB server with

```
unix> arangod mydatabasefolder
```

then running

```
unix> arangod mydatabasefolder --upgrade
```

will perform the upgrade for the whole ArangoDB instance, including all of its databases.

Starting with `--upgrade` will run a database version check and perform any necessary migrations. As usual, you should create a backup of your database directory before performing the upgrade.

The output should look like this:

```
2014-11-03T15:48:47Z [2708] INFO In database '_system': Found 24 defined task(s), 5 task(s) to run
2014-11-03T15:48:47Z [2708] INFO In database '_system': state prod/standalone/upgrade, tasks updateUserModel, createStat
2014-11-03T15:48:48Z [2708] INFO In database '_system': upgrade successfully finished
2014-11-03T15:48:48Z [2708] INFO database upgrade passed
```

Please check the output the `--upgrade` run. It may produce errors, which need to be fixed before ArangoDB can be used properly. If no errors are present or they have been resolved, you can start ArangoDB 2.3 regularly.

Upgrading a cluster planned in the web interface

A cluster of ArangoDB instances has to be upgraded as well. This involves upgrading all ArangoDB instances in the cluster, as well as running the version check on the whole running cluster in the end.

We have tried to make this procedure as painless and convenient for you. We assume that you planned, launched and administrated a cluster using the graphical front end in your browser. The upgrade procedure is then as follows:

1. First shut down your cluster using the graphical front end as usual.
2. Then upgrade all dispatcher instances on all machines in your cluster using the version check as described above and restart them.
3. Now open the cluster dash board in your browser by pointing it to the same dispatcher that you used to plan and launch the cluster in the graphical front end. In addition to the usual buttons "Relaunch", "Edit cluster plan" and "Delete cluster plan" you will see another button marked "Upgrade and relaunch cluster".
4. Hit this button, your cluster will be upgraded and launched and all is done for you behind the scenes. If all goes well, you will see the usual cluster dash board after a few seconds. If there is an error, you have to inspect the log files of your cluster ArangoDB instances. Please let us know if you run into problems.

There is an alternative way using the `ArangoDB` shell. Instead of steps 3. and 4. above you can launch `arangosh`, point it to the dispatcher that you have used to plan and launch the cluster using the option `--server.endpoint`, and execute

```
arangosh> require("org/arangodb/cluster").Upgrade("root","");
```

This upgrades the cluster and launches it, exactly as with the button above in the graphical front end. You have to replace `"root"` with a user name and `""` with a password that is valid for authentication with the cluster.

Upgrading to ArangoDB 2.2

Please read the following sections if you upgrade from a previous version to ArangoDB 2.2.

Please note first that a database directory used with ArangoDB 2.2 cannot be used with earlier versions (e.g. ArangoDB 2.1) any more. Upgrading a database directory cannot be reverted. Therefore please make sure to create a full backup of your existing ArangoDB installation before performing an upgrade.

Database Directory Version Check and Upgrade

ArangoDB will perform a database version check at startup. When ArangoDB 2.2 encounters a database created with earlier versions of ArangoDB, it will refuse to start. This is intentional.

The output will then look like this:

```
2014-07-07T22:04:53Z [18675] ERROR In database '_system': Database directory version (2.1) is lower than server version
2014-07-07T22:04:53Z [18675] ERROR In database '_system': --------------------------------------------------------------
2014-07-07T22:04:53Z [18675] ERROR In database '_system': It seems like you have upgraded the ArangoDB binary.
2014-07-07T22:04:53Z [18675] ERROR In database '_system': If this is what you wanted to do, please restart with the
2014-07-07T22:04:53Z [18675] ERROR In database '_system':   --upgrade
2014-07-07T22:04:53Z [18675] ERROR In database '_system': option to upgrade the data in the database directory.
2014-07-07T22:04:53Z [18675] ERROR In database '_system': Normally you can use the control script to upgrade your databa
2014-07-07T22:04:53Z [18675] ERROR In database '_system':   /etc/init.d/arangodb stop
2014-07-07T22:04:53Z [18675] ERROR In database '_system':   /etc/init.d/arangodb upgrade
2014-07-07T22:04:53Z [18675] ERROR In database '_system':   /etc/init.d/arangodb start
2014-07-07T22:04:53Z [18675] ERROR In database '_system': --------------------------------------------------------------
2014-07-07T22:04:53Z [18675] FATAL Database version check failed for '_system'. Please start the server with the --upgra
```

To make ArangoDB 2.2 start with a database directory created with an earlier ArangoDB version, you may need to invoke the upgrade procedure once. This can be done by running ArangoDB from the command line and supplying the `--upgrade` option:

```
unix> arangod data --upgrade
```

where `data` is ArangoDB's main data directory.

Note: here the same database should be specified that is also specified when arangod is started regularly. Please do not run the `--upgrade` command on each individual database subfolder (named `database-<some number>`).

For example, if you regularly start your ArangoDB server with

```
unix> arangod mydatabasefolder
```

then running

```
unix> arangod mydatabasefolder --upgrade
```

will perform the upgrade for the whole ArangoDB instance, including all of its databases.

Starting with `--upgrade` will run a database version check and perform any necessary migrations. As usual, you should create a backup of your database directory before performing the upgrade.

The output should look like this:

```
2014-07-07T22:11:30Z [18867] INFO In database '_system': starting upgrade from version 2.1 to 2.2.0
2014-07-07T22:11:30Z [18867] INFO In database '_system': Found 19 defined task(s), 2 task(s) to run
2014-07-07T22:11:30Z [18867] INFO In database '_system': upgrade successfully finished
2014-07-07T22:11:30Z [18867] INFO database upgrade passed
```

Please check the output the `--upgrade` run. It may produce errors, which need to be fixed before ArangoDB can be used properly. If no errors are present or they have been resolved, you can start ArangoDB 2.2 regularly.

Upgrading a cluster planned in the web interface

A cluster of ArangoDB instances has to be upgraded as well. This involves upgrading all ArangoDB instances in the cluster, as well as running the version check on the whole running cluster in the end.

We have tried to make this procedure as painless and convenient for you. We assume that you planned, launched and administrated a cluster using the graphical front end in your browser. The upgrade procedure is then as follows:

1. First shut down your cluster using the graphical front end as usual.
2. Then upgrade all dispatcher instances on all machines in your cluster using the version check as described above and restart them.
3. Now open the cluster dash board in your browser by pointing it to the same dispatcher that you used to plan and launch the cluster in the graphical front end. In addition to the usual buttons "Relaunch", "Edit cluster plan" and "Delete cluster plan" you will see another button marked "Upgrade and relaunch cluster".
4. Hit this button, your cluster will be upgraded and launched and all is done for you behind the scenes. If all goes well, you will see the usual cluster dash board after a few seconds. If there is an error, you have to inspect the log files of your cluster ArangoDB instances. Please let us know if you run into problems.

There is an alternative way using the `ArangoDB` shell. Instead of steps 3. and 4. above you can launch `arangosh`, point it to the dispatcher that you have used to plan and launch the cluster using the option `--server.endpoint`, and execute

```
arangosh> require("org/arangodb/cluster").Upgrade("root","");
```

This upgrades the cluster and launches it, exactly as with the button above in the graphical front end. You have to replace `"root"` with a user name and `""` with a password that is valid for authentication with the cluster.

Auditing

This feature is available in the Enterprise Edition.

Auditing allows you to monitor access to the database in detail. In general audit logs are of the form

```
2016-01-01 12:00:00 | server | username | database | client-ip | authentication | text1 | text2 | ...
```

The *time-stamp* is in GMT. This allows to easily match log entries from servers in different time zones.

The name of the *server*. You can specify a custom name on startup. Otherwise the default hostname is used.

The *username* is the (authenticated or unauthenticated) name supplied by the client. A dash `-` is printed if no name was given by the client.

The *database* describes the database that was accessed. Please note that there are no database crossing queries. Each access is restricted to one database.

The *client-ip* describes the source of the request.

The *authentication* details the methods used to authenticate the user.

Details about the requests follow in the additional fields.

Audit Configuration

This feature is available in the Enterprise Edition.

Output

`--audit.output output`

Specifies the target of the audit log. Possible values are

`file://filename` where *filename* can be relative or absolute.

`syslog://facility` or `syslog://facility/application-name` to log into a syslog server.

The option can be specified multiple times in order to configure the output for multiple targets.

Hostname

`--audit.hostname name`

The name of the server used in audit log messages. By default the system hostname is used.

Audit Events

This feature is available in the Enterprise Edition.

Authentication

Unknown authentication methods

```
2016-10-03 15:44:23 | server1 | - | database1 | 127.0.0.1:61525 | - | unknown authentication method | /_api/version
```

Missing credentials

```
2016-10-03 15:39:49 | server1 | - | database1 | 127.0.0.1:61498 | - | credentials missing | /_api/version
```

Wrong credentials

```
2016-10-03 15:47:26 | server1 | user1 | database1 | 127.0.0.1:61528 | http basic | credentials wrong | /_api/version
```

Password change required

```
2016-10-03 16:18:53 | server1 | user1 | database1 | 127.0.0.1:62257 | - | password change required | /_api/version
```

JWT login succeeded

```
2016-10-03 17:21:22 | server1 | - | database1 | 127.0.0.1:64214 | http jwt | user 'root' authenticated | /_open/auth
```

Please note, that the user given as third part is the user that requested the login. In general, it will be empty.

JWT login failed

```
2016-10-03 17:21:22 | server1 | - | database1 | 127.0.0.1:64214 | http jwt | user 'root' wrong credentials  | /_open/aut
```

Please note, that the user given as third part is the user that requested the login. In general, it will be empty.

Authorization

User not authorized to access database

```
2016-10-03 16:20:52 | server1 | user1 | database2 | 127.0.0.1:62262 | http basic | not authorized | /_api/version
```

Databases

Create a database

```
2016-10-04 15:33:25 | server1 | user1 | database1 | 127.0.0.1:56920 | http basic | create database 'database1' | ok | /_
```

Drop a database

```
2016-10-04 15:33:25 | server1 | user1 | database1 | 127.0.0.1:56920 | http basic | delete database 'database1' | ok | /_
```

Collections

Create a collection

```
2016-10-05 17:35:57 | server1 | user1 | database1 | 127.0.0.1:51294 | http basic | create collection 'collection1' | ok
```

Truncate a collection

```
2016-10-05 17:36:08 | server1 | user1 | database1 | 127.0.0.1:51294 | http basic | truncate collection 'collection1' | o
```

Drop a collection

```
2016-10-05 17:36:30 | server1 | user1 | database1 | 127.0.0.1:51294 | http basic | delete collection 'collection1' | ok
```

Indexes

Create a index

```
2016-10-05 18:19:40 | server1 | user1 | database1 | 127.0.0.1:52467 | http basic | create index in 'collection1' | ok |
```

Drop a index

```
2016-10-05 18:18:28 | server1 | user1 | database1 | 127.0.0.1:52464 | http basic | drop index ':44051' | ok | /_api/inde
```

Documents

Reading a single document

```
2016-10-04 12:27:55 | server1 | user1 | database1 | 127.0.0.1:53699 | http basic | create document ok | /_api/document/c
```

Replacing a single document

```
2016-10-04 12:28:08 | server1 | user1 | database1 | 127.0.0.1:53699 | http basic | replace document ok | /_api/document/
```

Modifying a single document

```
2016-10-04 12:28:15 | server1 | user1 | database1 | 127.0.0.1:53699 | http basic | modify document ok | /_api/document/co
```

Deleting a single document

```
2016-10-04 12:28:23 | server1 | user1 | database1 | 127.0.0.1:53699 | http basic | delete document ok | /_api/document/co
```

For example, if someones tries to delete a non-existing document, it will be logged as

```
2016-10-04 12:28:26 | server1 | user1 | database1 | 127.0.0.1:53699 | http basic | delete document failed | /_api/documen
```

Queries

```
2016-10-06 12:12:10 | server1 | user1 | database1 | 127.0.0.1:54232 | http basic | query document | ok | for i in collec
```

Troubleshooting

Arangod

If the ArangoDB server does not start or if you cannot connect to it using *arangosh* or other clients, you can try to find the problem cause by executing the following steps. If the server starts up without problems you can skip this section.

- *Check the server log file*: If the server has written a log file you should check it because it might contain relevant error context information.
- *Check the configuration*: The server looks for a configuration file named *arangod.conf* on startup. The contents of this file will be used as a base configuration that can optionally be overridden with command-line configuration parameters. You should check the config file for the most relevant parameters such as:
  - *server.endpoint*: What IP address and port to bind to
  - *log parameters*: If and where to log
  - *database.directory*: Path the database files are stored in

  If the configuration reveals that something is not configured right the config file should be adjusted and the server be restarted.
- *Start the server manually and check its output*: Starting the server might fail even before logging is activated so the server will not produce log output. This can happen if the server is configured to write the logs to a file that the server has no permissions on. In this case the server cannot log an error to the specified log file but will write a startup error on stderr instead. Starting the server manually will also allow you to override specific configuration options, e.g. to turn on/off file or screen logging etc.
- *Check the TCP port*: If the server starts up but does not accept any incoming connections this might be due to firewall configuration between the server and any client(s). The server by default will listen on TCP port 8529. Please make sure this port is actually accessible by other clients if you plan to use ArangoDB in a network setup.

  When using hostnames in the configuration or when connecting, please make sure the hostname is actually resolvable. Resolving hostnames might invoke DNS, which can be a source of errors on its own.

  It is generally good advice to not use DNS when specifying the endpoints and connection addresses. Using IP addresses instead will rule out DNS as a source of errors. Another alternative is to use a hostname specified in the local */etc/hosts* file, which will then bypass DNS.
- *Test if* curl *can connect*: Once the server is started, you can quickly verify if it responds to requests at all. This check allows you to determine whether connection errors are client-specific or not. If at least one client can connect, it is likely that connection problems of other clients are not due to ArangoDB's configuration but due to client or in-between network configurations.

  You can test connectivity using a simple command such as:

  curl --dump - -X GET http://127.0.0.1:8529/_api/version && echo

  This should return a response with an *HTTP 200* status code when the server is running. If it does it also means the server is generally accepting connections. Alternative tools to check connectivity are *lynx* or *ab*.

Emergency Console

The ArangoDB database server has two modes of operation: As a server, where it will answer to client requests and as an emergency console, in which you can access the database directly. The latter - as the name suggests - should only be used in case of an emergency, for example, a corrupted collection. Using the emergency console allows you to issue all commands normally available in actions and transactions. When starting the server in emergency console mode, the server cannot handle any client requests.

You should never start more than one server using the same database directory, independent of the mode of operation. Normally, ArangoDB will prevent you from doing this by placing a lockfile in the database directory and not allowing a second ArangoDB instance to use the same database directory if a lockfile is already present.

In Case Of Disaster

The following command starts an emergency console.

Note: Never start the emergency console for a database which also has a server attached to it. In general, the ArangoDB shell is what you want.

```
> ./arangod --console --log error /tmp/vocbase
ArangoDB shell [V8 version 5.0.71.39, DB version 3.x.x]

arango> 1 + 2;
3

arango> var db = require("@arangodb").db; db.geo.count();
703
```

The emergency console provides a JavaScript console directly running in the arangod server process. This allows to debug and examine collections and documents as with the normal ArangoDB shell, but without client/server communication.

However, it is very likely that you will never need the emergency console unless you are an ArangoDB developer.

Datafile Debugger

In Case Of Disaster

AranagoDB uses append-only journals. Data corruption should only occur when the database server is killed. In this case, the corruption should only occur in the last object(s) that have being written to the journal.

If a corruption occurs within a normal datafile, then this can only happen if a hardware fault occurred.

If a journal or datafile is corrupt, shut down the database server and start the program

```
unix> arango-dfdb
```

in order to check the consistency of the datafiles and journals. This brings up

```
    ___      _         __ _ _           ___  ___    ___
   /   \__ _| |_ __ _ / _(_) | ___     /   \/ __\  / _ \
  / /\ / _` | __/ _` | |_| | |/ _ \   / /\ /__\// / /_\/
 / /_// (_| | || (_| |  _| | |  __/  / /_// \/  \/ /_\\
/___,' \__,_|\__\__,_|_| |_|_|\___| /___,'\_____/\____/

Available collections:
  0: _structures
  1: _users
  2: _routing
  3: _modules
  4: _graphs
  5: products
  6: prices
  *: all

Collection to check:
```

You can now select which database and collection you want to check. After you selected one or all of the collections, a consistency check will be performed.

```
Checking collection #1: _users

Database
  path: /usr/local/var/lib/arangodb

Collection
  name: _users
  identifier: 82343

Datafiles
  # of journals: 1
  # of compactors: 1
  # of datafiles: 0

Datafile
  path: /usr/local/var/lib/arangodb/collection-82343/journal-1065383.db
  type: journal
  current size: 33554432
  maximal size: 33554432
  total used: 256
  # of entries: 3
  status: OK
```

If there is a problem with one of the datafiles, then the database debugger will print it and prompt for whether to attempt to fix it.

```
WARNING: The journal was not closed properly, the last entries are corrupted.
         This might happen ArangoDB was killed and the last entries were not
         fully written to disk.

Wipe the last entries (Y/N)?
```

If you answer **Y**, the corrupted entry will be removed.

If you see a corruption in a datafile (and not a journal), then something is terribly wrong. These files are immutable and never changed by ArangoDB. A corruption in such file is an indication of a hard-disk failure.

Arangobench

Arangobench is ArangoDB's benchmark and test tool. It can be used to issue test requests to the database for performance and server function testing. It supports parallel querying and batch requests.

Related blog posts:

- Measuring ArangoDB insert performance
- Gain factor of 5 using batch requests

Startup options

- --*async*: Send asynchronous requests. The default value is *false*.
- --*batch-size*: Number of operations to send per batch. Use 0 to disable batching (this is the default).
- --*collection*: Name of collection to use in test (only relevant for tests that invoke collections).
- --*complexity*: Complexity value for test case (default: 1). Meaning depends on test case.
- --*concurrency*: Number of parallel threads that will issue requests (default: 1).
- --*configuration*: Read configuration from file.
- --*delay*: Use a startup delay. This is only necessary when run in series. The default value is *false*.
- --*keep-alive*: Use HTTP keep-alive (default: true).
- --*progress*: Show progress of benchmark, on every 20th request. Set to *false* to disable intermediate logging. The default value is *true*.
- --*requests*: Total number of requests to perform (default: 1000).
- --*server.endpoint*: Server endpoint to connect to, consisting of protocol, IP address and port. Defaults to *tcp://localhost:8529*.
- --*server.database*: Database name to use when connecting (default: "_system").
- --*server.username*: Username to use when connecting (default: "root").
- --*server.password*: Password to use when connecting. Don't specify this option to get a password prompt.
- --*server.authentication*: Wether or not to show the password prompt and use authentication when connecting to the server (default: true).
- --*test-case*: Name of test case to perform (default: "version"). Possible values:
  - version : requests /_api/version
  - document : creates documents
  - collection : creates collections
  - import-document : creates documents via the import API
  - hash : Create/Read/Update/Read documents indexed by a hash index
  - skiplist : Create/Read/Update/Read documents indexed by a skiplist
  - edge : Create/Read/Update edge documents
  - shapes : Create & Delete documents with heterogeneous attribute names
  - shapes-append : Create documents with heterogeneous attribute names
  - random-shapes : Create/Read/Delete heterogeneous documents with random values
  - crud : Create/Read/Update/Delete
  - crud-append : Create/Read/Update/Read again
  - crud-write-read : Create/Read Documents
  - aqltrx : AQL Transactions with deep nested AQL `FOR` - loops
  - counttrx : uses JS transactions to count the documents and insert the result again
  - multitrx : multiple transactions combining reads & writes from js

  - multi-collection : multiple transactions combining reads & writes from js on multiple collections
  - aqlinsert : insert documents via AQL
  - aqlv8 : execute AQL with V8 functions to insert random documents
- *--verbose*: Print out replies if the HTTP header indicates DB errors. (default: false).

Examples

```
arangobench
```

Starts Arangobench with the default user and server endpoint.

```
--test-case version --requests 1000 --concurrency 1
```

Runs the 'version' test case with 1000 requests, without concurrency.

```
--test-case document --requests 1000 --concurrency 2
```

Runs the 'document' test case with 2000 requests, with two concurrent threads.

```
--test-case document --requests 1000 --concurrency 2 --async true
```

Runs the 'document' test case with 2000 requests, with concurrency 2, with async requests.

```
--test-case document --requests 1000 --concurrency 2 --batch-size 10
```

Runs the 'document' test case with 2000 requests, with concurrency 2, using batch requests.

Architecture

AppendOnly/MVCC

Instead of overwriting existing documents, ArangoDB will create a new version of modified documents. This is even the case when a document gets deleted. The two benefits are:

- Objects can be stored coherently and compactly in the main memory.
- Objects are preserved, isolated writing and reading transactions allow accessing these objects for parallel operations.

The system collects obsolete versions as garbage, recognizing them as forsaken. Garbage collection is asynchronous and runs parallel to other processes.

Mostly Memory/Durability

Database documents are stored in memory-mapped files. Per default, these memory-mapped files are synced regularly but not instantly. This is often a good tradeoff between storage performance and durability. If this level of durability is too low for an application, the server can also sync all modifications to disk instantly. This will give full durability but will come with a performance penalty as each data modification will trigger a sync I/O operation.

Write-ahead log

Starting with version 2.2 ArangoDB stores all data-modification operation in its write-ahead log. The write-ahead log is sequence of append-only files containing all the write operations that were executed on the server.

It is used to run data recovery after a server crash, and can also be used in a replication setup when slaves need to replay the same sequence of operations as on the master.

By default, each write-ahead logfile is 32 MB big. This size is configurable via the option --*wal.logfile-size*.

When a write-ahead logfile is full, it is set to read-only, and following operations will be written into the next write-ahead logfile. By default, ArangoDB will reserve some spare logfiles in the background so switching logfiles should be fast. How many reserve logfiles ArangoDB will try to keep available in the background can be controlled by the configuration option --*wal.reserve-logfiles*.

Data contained in full datafiles will eventually be transferred into the journals or datafiles of collections. Only the "surviving" documents will be copied over. When all remaining operations from a write-ahead logfile have been copied over into the journals or datafiles of the collections, the write-ahead logfile can safely be removed if it is not used for replication.

Long-running transactions prevent write-ahead logfiles from being fully garbage-collected because it is unclear whether a transaction will commit or abort. Long-running transactions can thus block the garbage-collection progress and should therefore be avoided at all costs.

On a system that acts as a replication master, it is useful to keep a few of the already collected write-ahead logfiles so replication slaves still can fetch data from them if required. How many collected logfiles will be kept before they get deleted is configurable via the option --*wal.historic-logfiles*.

For all write-ahead log configuration options, please refer to the page Write-ahead log options.

Release Notes

Whats New

- Whats New in 3.1
- Whats New in 3.0
- Whats New in 2.8
- Whats New in 2.7
- Whats New in 2.6
- Whats New in 2.5
- Whats New in 2.4
- Whats New in 2.3
- Whats New in 2.2
- Whats New in 2.1

Incompatible changes

Also see Upgrading in the Administration chapter.

- Incompatible changes in 3.1
- Incompatible changes in 3.0
- Incompatible changes in 2.8
- Incompatible changes in 2.7
- Incompatible changes in 2.6
- Incompatible changes in 2.5
- Incompatible changes in 2.4
- Incompatible changes in 2.3

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 3.1. ArangoDB 3.1 also contains several bugfixes that are not listed here.

SmartGraphs

ArangoDB 3.1 adds a first major enterprise only feature called SmartGraphs. SmartGraphs form an addition to the already existing graph features and allow to scale graphs beyond a single machine while keeping almost the same query performance. The SmartGraph feature is suggested for all graph database use cases that require a cluster of database servers for what ever reason. You can either have a graph that is too large to be stored on a single machine only. Or you can have a small graph, but at the same time need additional data with has to be sharded and you want to keep all of them in the same envirenment. Or you simply use the cluster for high-availability. In all the above cases SmartGraphs will significantly increase the performance of graph operations. For more detailed information read this manual section.

Data format

The format of the revision values stored in the `_rev` attribute of documents has been changed in 3.1. Up to 3.0 they were strings containing largish decimal numbers. With 3.1, revision values are still strings, but are actually encoded time stamps of the creation date of the revision of the document. The time stamps are acquired using a hybrid logical clock (HLC) on the DBserver that holds the revision (for the concept of a hybrid logical clock see this paper). See this manual section for details.

ArangoDB >= 3.1 can ArangoDB 3.0 database directories and will simply continue to use the old `_rev` attribute values. New revisions will be written with the new time stamps.

It is highly recommended to backup all your data before loading a database directory that was written by ArangoDB <= 3.0 into an ArangoDB >= 3.1.

Communication Layer

ArangoDB up to 3.0 used libev for the communication layer. ArangoDB starting from 3.1 uses Boost ASIO.

Starting with ArangoDB 3.1 we begin to provide the VelocyStream Protocol (vst) as a addition to the established http protocol.

A few options have changed concerning communication, please checkout Incompatible changes in 3.1.

Cluster

For its internal cluster communication a (bundled version) of curl is now being used. This enables asynchronous operation throughout the cluster and should improve general performance slightly.

Authentication is now supported within the cluster.

Document revisions cache

The ArangoDB server now provides an in-memory cache for frequently accessed document revisions. Documents that are accessed during read/write operations are loaded into the revisions cache automatically, and subsequently served from there.

The cache has a total target size, which can be controlled with the startup option `--database.revision-cache-target-size`. Once the cache reaches the target size, older entries may be evicted from the cache to free memory. Note that the target size currently is a high water mark that will trigger cache memory garbage collection if exceeded. However, if all cache chunks are still in use when the high water mark is reached, the cache may still grow and allocate more chunks until cache entries become unused and are allowed to be garbage-collected.

The cache is maintained on a per-collection basis, that is, memory for the cache is allocated on a per-collection basis in chunks. The size for the cache memory chunks can be controlled via the startup option `--database.revision-cache-chunk-size`. The default value is 4 MB per chunk. Bigger chunk sizes allow saving more documents per chunk, which can lead to more efficient chunk allocation and lookups, but will also lead to memory waste if many chunks are allocated and not fully used. The latter will be the case if there exist many small collections which all allocate their own chunks but not fully utilize them because of the low number of documents.

AQL

Functions added

The following AQL functions have been added in 3.1:

- *OUTERSECTION(array1, array2, ..., arrayn)*: returns the values that occur only once across all arrays specified.
- *DISTANCE(lat1, lon1, lat2, lon2)*: returns the distance between the two coordinates specified by *(lat1, lon1)* and *(lat2, lon2)*. The distance is calculated using the haversine formula.
- *JSON_STRINGIFY(value)*: returns a JSON string representation of the value.
- *JSON_PARSE(value)*: converts a JSON-encoded string into a regular object

Index usage in traversals

3.1 allows AQL traversals to use other indexes than just the edge index. Traversals with filters on edges can now make use of more specific indexes. For example, the query

```
FOR v, e, p IN 2 OUTBOUND @start @@edge
  FILTER p.edges[0].foo == "bar"
  RETURN [v, e, p]
```

may use a hash index on `["_from", "foo"]` instead of the edge index on just `["_from"]`.

Optimizer improvements

Make the AQL query optimizer inject filter condition expressions referred to by variables during filter condition aggregation. For example, in the following query

```
FOR doc IN collection
  LET cond1 = (doc.value == 1)
  LET cond2 = (doc.value == 2)
  FILTER cond1 || cond2
  RETURN { doc, cond1, cond2 }
```

the optimizer will now inject the conditions for `cond1` and `cond2` into the filter condition `cond1 || cond2`, expanding it to `(doc.value == 1) || (doc.value == 2)` and making these conditions available for index searching.

Note that the optimizer previously already injected some conditions into other conditions, but only if the variable that defined the condition was not used elsewhere. For example, the filter condition in the query

```
FOR doc IN collection
  LET cond = (doc.value == 1)
  FILTER cond
  RETURN { doc }
```

already got optimized before because `cond` was only used once in the query and the optimizer decided to inject it into the place where it was used.

This only worked for variables that were referred to once in the query. When a variable was used multiple times, the condition was not injected as in the following query

```
FOR doc IN collection
  LET cond = (doc.value == 1)
  FILTER cond
  RETURN { doc, cond }
```

3.1 allows using this condition so that the query can use an index on `doc.value` (if such index exists).

Miscellaneous improvements

The performance of the `[*]` operator was improved for cases in which this operator did not use any filters, projections and/or offset/limits.

The AQL query executor can now report the time required for loading and locking the collections used in an AQL query. When profiling is enabled, it will report the total loading and locking time for the query in the `loading collections` sub-attribute of the `extra.profile` value of the result. The loading and locking time can also be view in the AQL query editor in the web interface.

Audit Log

Audit logging has been added, see Auditing.

Client tools

Added option `--skip-lines` for arangoimp This allows skipping the first few lines from the import file in case the CSV or TSV import are used and some initial lines should be skipped from the input.

Web Admin Interface

The usability of the AQL editor significantly improved. In addition to the standard JSON output, the AQL Editor is now able to render query results as a graph preview or a table. Furthermore the AQL editor displays query profiling information.

Added a new Graph Viewer in order to exchange the technically obsolete version. The new Graph Viewer is based on Canvas but does also include a first WebGL implementation (limited functionality - will change in the future). The new Graph Viewer offers a smooth way to discover and visualize your graphs.

The shard view in cluster mode now displays a progress indicator while moving shards.

Authentication

Up to ArangoDB 3.0 authentication of client requests was only possible with HTTP basic authentication.

Starting with 3.1 it is now possible to also use a JSON Web Tokens (JWT) for authenticating incoming requests.

For details check the HTTP authentication chapter. Both authentication methods are valid and will be supported in the near future. Use whatever suits you best.

Foxx

GraphQL

It is now easy to get started with providing GraphQL APIs in Foxx, see Foxx GraphQL.

OAuth2

Foxx now officially provides a module for implementing OAuth2 clients, see Foxx OAuth2.

Per-route middleware

It's now possible to specify middleware functions for a route when defining a route handler. These middleware functions only apply to the single route and share the route's parameter definitions. Check out the Foxx Router documentation for more information.

Incompatible changes in ArangoDB 3.1

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 3.1, and adjust any client programs if necessary.

Communication Layer

The internal commication layer is now based on Boost ASIO. A few options regarding threads and communication have been changed.

There are no longer two different threads pools (`--scheduler.threads` and `--server.threads`). The option `--scheduler.threads` has been removed. The number of threads is now controlled by the option `--server.threads` only. By default `--server.threads` is set to the number of hyper-cores.

As a consequence of the change, the following (hidden) startup options have been removed:

- `--server.extra-threads`
- `--server.aql-threads`
- `--server.backend`
- `--server.show-backends`
- `--server.thread-affinity`

AQL

The behavior of the AQL array comparison operators has changed for empty arrays:

- `ALL` and `ANY` now always return `false` when the left-hand operand is an empty array. The behavior for non-empty arrays does not change:
  - `[] ALL == 1` will return `false`
  - `[1] ALL == 1` will return `true`
  - `[1, 2] ALL == 1` will return `false`
  - `[2, 2] ALL == 1` will return `false`
  - `[] ANY == 1` will return `false`
  - `[1] ANY == 1` will return `true`
  - `[1, 2] ANY == 1` will return `true`
  - `[2, 2] ANY == 1` will return `false`
- `NONE` now always returns `true` when the left-hand operand is an empty array. The behavior for non-empty arrays does not change:
  - `[] NONE == 1` will return `true`
  - `[1] NONE == 1` will return `false`
  - `[1, 2] NONE == 1` will return `false`
  - `[2, 2] NONE == 1` will return `true`

Data format changes

The attribute `maximalSize` has been renamed to `journalSize` in collection meta-data files ("parameter.json"). Files containing the `maximalSize` attribute will still be picked up correctly for not-yet adjusted collections.

The format of the revision values stored in the `_rev` attribute of documents has been changed in 3.1. Up to 3.0 they were strings containing largish decimal numbers. With 3.1, revision values are still strings, but are actually encoded time stamps of the creation date of the revision of the document. The time stamps are acquired using a hybrid logical clock (HLC) on the DBserver that holds the revision (for the concept of a hybrid logical clock see this paper). See this manual section for details.

ArangoDB >= 3.1 can ArangoDB 3.0 database directories and will simply continue to use the old `_rev` attribute values. New revisions will be written with the new time stamps.

It is highly recommended to backup all your data before loading a database directory that was written by ArangoDB <= 3.0 into an ArangoDB >= 3.1.

To change all your old `_rev` attributes into new style time stamps you have to use `arangodump` to dump all data out (using ArangoDB 3.0), and use `arangorestore` into the new ArangoDB 3.1, which is the safest way to upgrade.

The change also affects the return format of `_rev` values and other revision values in HTTP APIs (see below).

HTTP API changes

APIs added

The following HTTP REST APIs have been added for online loglevel adjustment of the server:

- GET `/_admin/log/level` returns the current loglevel settings
- PUT `/_admin/log/level` modifies the current loglevel settings

APIs changed

- the following REST APIs that return revision ids now make use of the new revision id format introduced in 3.1. All revision ids returned will be strings as in 3.0, but have a different internal format.

  The following APIs are affected:

  - GET /_api/collection/{collection}/checksum: `revision` attribute
  - GET /_api/collection/{collection}/revision: `revision` attribute
  - all other APIs that return documents, which may include the documents' `_rev` attribute

  Client applications should not try to interpret the internals of revision values, but only use revision values for checking whether two revision strings are identical.

- the replication REST APIs will now use the attribute name `journalSize` instead of `maximalSize` when returning information about collections.

- the default value for `keepNull` has been changed from `false` to `true` for the following partial update operations for vertices and edges in /_api/gharial:

  - PATCH /_api/gharial/{graph}/vertex/{collection}/{key}
  - PATCH /_api/gharial/{graph}/edge/{collection}/{key}

  The value for `keepNull` can still be set explicitly to `false` by setting the URL parameter `keepNull` to a value of `false`.

- the REST API for dropping collections (DELETE /_api/collection) now accepts an optional query string parameter `isSystem`, which can set to `true` in order to drop system collections. If the parameter is not set or not set to true, the REST API will refuse to drop system collections. In previous versions of ArangoDB, the `isSystem` parameter did not exist, and there was no distinction between system and non-system collections when dropping collections.

- the REST API for retrieving AQL query results (POST /_api/cursor) will now return an additional sub-attribute `loading collections` that will contain the total time required for loading and locking collections during the AQL query when profiling is enabled. The attribute can be found in the `extra` result attribute in sub-attribute `loading collections`. The attribute will only be set if profiling was enabled for the query.

Foxx Testing

The QUnit interface to Mocha has been removed. This affects the behaviour of the `suite`, `test`, `before`, `after`, `beforeEach` and `afterEach` functions in Foxx test suites. The `suite` and `test` functions are now provided by the TDD interface. The `before`, `after`, `beforeEach` and `afterEach` functions are now provided by the BDD interface.

This should not cause any problems with existing tests but may result in failures in test cases that previously passed for the wrong reasons. Specifically the execution order of the `before`, `after`, etc functions now follows the intended order and is no longer arbitrary.

For details on the expected behaviour of these functions see the testing chapter in the Foxx documentation.

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 3.0. ArangoDB 3.0 also contains several bugfixes that are not listed here.

Internal data format changes

ArangoDB now uses VelocyPack for storing documents, query results and temporarily computed values. Using a single data format removed the need for some data conversions in the core that slowed operations down previously.

The VelocyPack format is also quite compact, and reduces storage space requirements for "small" values such as boolean, integers, short strings. This can speed up several operations inside AQL queries.

VelocyPack document entries stored on disk are also self-contained, in the sense that each stored document will contain all of its data type and attribute name descriptions. While this may require a bit more space for storing the documents, it removes the overhead of fetching attribute names and document layout from shared structures as in previous versions of ArangoDB. It also simplifies the code paths for storing and reading documents.

AQL improvements

Syntax improvements

`LIKE` string-comparison operator

AQL now provides a `LIKE` operator and can be used to compare strings like this, for example inside filter conditions:

```
value LIKE search
```

This change makes `LIKE` an AQL keyword. Using `LIKE` as an attribute or collection name in AQL thus requires quoting the name from now on.

The `LIKE` operator is currently implemented by calling the already existing AQL function `LIKE`, which also remains operational in 3.0. Use the `LIKE` function in case you want to search case-insensitive (optional parameter), as the `LIKE` operator always compares case-sensitive.

AQL array comparison operators

All AQL comparison operators now also exist in an array variant. In the array variant, the operator is preceded with one of the keywords *ALL*, *ANY* or *NONE*. Using one of these keywords changes the operator behavior to execute the comparison operation for all, any, or none of its left hand argument values. It is therefore expected that the left hand argument of an array operator is an array.

Examples:

```
[ 1, 2, 3 ] ALL IN [ 2, 3, 4 ]   // false
[ 1, 2, 3 ] ALL IN [ 1, 2, 3 ]   // true
[ 1, 2, 3 ] NONE IN [ 3 ]        // false
[ 1, 2, 3 ] NONE IN [ 23, 42 ]   // true
[ 1, 2, 3 ] ANY IN [ 4, 5, 6 ]   // false
[ 1, 2, 3 ] ANY IN [ 1, 42 ]     // true
[ 1, 2, 3 ] ANY == 2             // true
[ 1, 2, 3 ] ANY == 4             // false
[ 1, 2, 3 ] ANY > 0              // true
[ 1, 2, 3 ] ANY <= 1             // true
[ 1, 2, 3 ] NONE < 99            // false
[ 1, 2, 3 ] NONE > 10            // true
[ 1, 2, 3 ] ALL > 2              // false
[ 1, 2, 3 ] ALL > 0              // true
[ 1, 2, 3 ] ALL >= 3             // false
["foo", "bar"] ALL != "moo"      // true
["foo", "bar"] NONE == "bar"     // false
["foo", "bar"] ANY == "foo"      // true
```

Regular expression string-comparison operators

AQL now supports the operators =~ and !~ for testing strings against regular expressions. =~ tests if a string value matches a regular expression, and !~ tests if a string value does not match a regular expression.

The two operators expect their left-hand operands to be strings, and their right-hand operands to be strings containing valid regular expressions as specified below.

The regular expressions may consist of literal characters and the following characters and sequences:

- `.` – the dot matches any single character except line terminators. To include line terminators, use `[\s\S]` instead to simulate `.` with *DOTALL* flag.
- `\d` – matches a single digit, equivalent to `[0-9]`
- `\s` – matches a single whitespace character
- `\S` – matches a single non-whitespace character
- `\t` – matches a tab character
- `\r` – matches a carriage return
- `\n` – matches a line-feed character
- `[xyz]` – set of characters. matches any of the enclosed characters (i.e. *x*, *y* or *z* in this case
- `[^xyz]` – negated set of characters. matches any other character than the enclosed ones (i.e. anything but *x*, *y* or *z* in this case)
- `[x-z]` – range of characters. Matches any of the characters in the specified range, e.g. `[0-9A-F]` to match any character in *0123456789ABCDEF*
- `[^x-z]` – negated range of characters. Matches any other character than the ones specified in the range
- `(xyz)` – defines and matches a pattern group
- `(x|y)` – matches either *x* or *y*
- `^` – matches the beginning of the string (e.g. `^xyz`)
- `$` – matches the end of the string (e.g. `xyz$`)

Note that the characters `.`, `*`, `?`, `[`, `]`, `(`, `)`, `{`, `}`, `^`, and `$` have a special meaning in regular expressions and may need to be escaped using a backslash (`\\`). A literal backslash should also be escaped using another backslash, i.e. `\\\\`.

Characters and sequences may optionally be repeated using the following quantifiers:

- `x*` – matches zero or more occurrences of *x*
- `x+` – matches one or more occurrences of *x*
- `x?` – matches one or zero occurrences of *x*
- `x{y}` – matches exactly *y* occurrences of *x*
- `x{y,z}` – matches between *y* and *z* occurrences of *x*
- `x{y,}` – matches at least *y* occurences of *x*

Enclosing identifiers in forward ticks

AQL identifiers can now optionally be enclosed in forward ticks in addition to using backward ticks. This allows convenient writing of AQL queries in JavaScript template strings (which are delimited with backticks themselves), e.g.

```
var q = `FOR doc IN ´collection´ RETURN doc.´name´`;
```

Functions added

The following AQL functions have been added in 3.0:

- *REGEX_TEST(value, regex)*: tests whether the string *value* matches the regular expression specified in *regex*. Returns *true* if it matches, and *false* otherwise.

  The syntax for regular expressions is the same as for the regular expression operators *=~* and *!~*.

- *HASH(value)*: Calculates a hash value for *value*. *value* is not required to be a string, but can have any data type. The calculated hash value will take the data type of *value* into account, so for example the number *1* and the string *"1"* will have different hash values. For arrays the hash values will be creared if the arrays contain exactly the same values (including value types) in the same order. For objects the same hash values will be created if the objects have exactly the same attribute names and values (including value types). The order in which attributes appear inside objects is not important for hashing. The hash value returned by this function is a number. The hash algorithm is not guaranteed to remain the same in future versions of ArangoDB. The hash values should therefore be used only for temporary calculations, e.g. to compare if two documents are the same, or for grouping values in queries.
- *TYPENAME(value)*: Returns the data type name of *value*. The data type name can be either *null*, *bool*, *number*, *string*, *array* or *object*.
- *LOG(value)*: Returns the natural logarithm of *value*. The base is Euler's constant (2.71828...).
- *LOG2(value)*: Returns the base 2 logarithm of *value*.
- *LOG10(value)*: Returns the base 10 logarithm of *value*.
- *EXP(value)*: Returns Euler's constant (2.71828...) raised to the power of *value*.
- *EXP2(value)*: Returns 2 raised to the power of *value*.
- *SIN(value)*: Returns the sine of *value*.
- *COS(value)*: Returns the cosine of *value*.
- *TAN(value)*: Returns the tangent of *value*.
- *ASIN(value)*: Returns the arcsine of *value*.
- *ACOS(value)*: Returns the arccosine of *value*.
- *ATAN(value)*: Returns the arctangent of *value*.
- *ATAN2(y, x)*: Returns the arctangent of the quotient of *y* and *x*.
- *RADIANS(value)*: Returns the angle converted from degrees to radians.
- *DEGREES(value)*: Returns the angle converted from radians to degrees.

Optimizer improvements

"inline-subqueries" rule

The AQL optimizer rule "inline-subqueries" has been added. This rule can pull out certain subqueries that are used as an operand to a `FOR` loop one level higher, eliminating the subquery completely. This reduces complexity of the query's execution plan and will likely enable further optimizations. For example, the query

```
FOR i IN (
    FOR j IN [1,2,3]
      RETURN j
  )
  RETURN i
```

will be transformed by the rule to:

```
FOR i IN [1,2,3]
  RETURN i
```

The query

```
FOR name IN (
  FOR doc IN _users
    FILTER doc.status == 1
    RETURN doc.name
  )
  LIMIT 2
  RETURN name
```

will be transformed into

```
FOR tmp IN _users
  FILTER tmp.status == 1
  LIMIT 2
  RETURN tmp.name
```

The rule will only fire when the subquery is used as an operand to a `FOR` loop, and if the subquery does not contain a `COLLECT` with an `INTO` variable.

"remove-unnecessary-calculations" rule

The AQL optimizer rule "remove-unnecessary-calculations" now fires in more cases than in previous versions. This rule removes calculations from execution plans, and by having less calculations done, a query may execute faster or requires less memory.

The rule will now remove calculations that are used exactly once in other expressions (e.g. `LET a = doc RETURN a.value`) and calculations, or calculations that are just references to other variables (e.g. `LET a = b`).

"optimize-traversals" rule

The AQL optimizer rule "merge-traversal-filter" was renamed to "optimize-traversals". The rule will remove unused edge and path result variables from the traversal in case they are specified in the `FOR` section of the traversal, but not referenced later in the query. This saves constructing edges and paths results that are not used later.

AQL now uses VelocyPack internally for storing intermediate values. For many value types it can now get away without extra memory allocations and less internal conversions. Values can be passed into internal AQL functions without copying them. This can lead to reduced query execution times for queries that use C++-based AQL functions.

"replace-or-with-in" and "use-index-for-sort" rules

These rules now fire in some additional cases, which allows simplifying index lookup conditions and removing SortNodes from execution plans.

Cluster state management

The cluster's internal state information is now also managed by ArangoDB instances. Earlier versions relied on third party software being installed for the storing the cluster state. The state is managed by dedicated ArangoDB instances, which can be started in a special *agency* mode. These instances can operate in a distributed fashion. They will automatically elect one of them to become their leader, being responsibile for storing the state changes sent from servers in the cluster. The other instances will automatically follow the leader and will transparently stand in should it become unavailable. The agency instances are also self-organizing: they will continuously probe each other and re-elect leaders. The communication between the agency instances use the consensus-based RAFT protocol.

The operations for storing and retrieving cluster state information are now much less expensive from an ArangoDB cluster node perspective, which in turn allows for faster cluster operations that need to fetch or update the overall cluster state.

`_from` and `_to` attributes of edges are updatable and usable in indexes

In ArangoDB prior to 3.0 the attributes `_from` and `_to` of edges were treated specially when loading or storing edges. That special handling led to these attributes being not as flexible as regular document attributes. For example, the `_from` and `_to` attribute values of an existing edge could not be updated once the edge was created. Now this is possible via the single-document APIs and via AQL.

Additionally, the `_from` and `_to` attributes could not be indexed in user-defined indexes, e.g. to make each combination of `_from` and `_to` unique. Finally, as `_from` and `_to` referenced the linked collections by collection id and not by collection name, their meaning became unclear once a referenced collection was dropped. The collection id stored in edges then became unusable, and when accessing such edge the collection name part of it was always translated to `_undefined` .

In ArangoDB 3.0, the `_from` and `_to` values of edges are saved as regular strings. This allows using `_from` and `_to` in user-defined indexes. Additionally, this allows to update the `_from` and `_to` values of existing edges. Furthermore, collections referenced by `_from` and `_to` values may be dropped and re-created later. Any `_from` and `_to` values of edges pointing to such dropped collection are unaffected by the drop operation now.

Unified APIs for CRUD operations

The CRUD APIs for documents and edge have been unified. Edges can now be inserted and modified via the same APIs as documents. `_from` and `_to` attribute values can be passed as regular document attributes now:

```
db.myedges.insert({ _from: "myvertices/some", _to: "myvertices/other", ... });
```

Passing `_from` and `_to` separately as it was required in earlier versions is not necessary anymore but will still work:

```
db.myedges.insert("myvertices/some", "myvertices/other", { ... });
```

The CRUD operations now also support batch variants that works on arrays of documents/edges, e.g.

```
db.myedges.insert([
  { _from: "myvertices/some", _to: "myvertices/other", ... },
  { _from: "myvertices/who", _to: "myvertices/friend", ... },
  { _from: "myvertices/one", _to: "myvertices/two", ... },
]);
```

The batch variants are also available in ArangoDB's HTTP API. They can be used to more efficiently carry out operations with multiple documents than their single-document equivalents, which required one HTTP request per operation. With the batch operations, the HTTP request/response overhead can be amortized across multiple operations.

Persistent indexes

ArangoDB 3.0 provides an experimental persistent index feature. Persistent indexes store the index values on disk instead of in-memory only. This means the indexes do not need to be rebuilt in-memory when a collection is loaded or reloaded, which should improve collection loading times.

The persistent indexes in ArangoDB are based on the RocksDB engine. To create a persistent index for a collection, create an index of type "rocksdb" as follows:

```
db.mycollection.ensureIndex({ type: "rocksdb", fields: [ "fieldname" ]});
```

The persistent indexes are sorted, so they allow equality lookups and range queries. Note that the feature is still highly experimental and has some known deficiencies. It will be finalized until the release of the 3.0 stable version.

Upgraded V8 version

The V8 engine that is used inside ArangoDB to execute JavaScript code has been upgraded from version 4.3.61 to 5.0.71.39. The new version makes several more ES6 features available by default, including

- arrow functions
- computed property names
- rest parameters
- array destructuring
- numeric and object literals

Web Admin Interface

The ArangoDB 3.0 web interface is significantly improved. It now comes with a more responsive design, making it easier to use on different devices. Navigation and menus have been simplified, and related items have been regrouped to stay closer together and allow tighter workflows.

The AQL query editor is now much easier to use. Multiple queries can be started and tracked in parallel, while results of earlier queries are still preserved. Queries still running can be canceled directly from the editor. The AQL query editor now allows the usage of bind parameters too, and provides a helper for finding collection names, AQL function names and keywords quickly.

The web interface now keeps track of whether the server is offline and of which server-side operations have been started and are still running. It now remains usable while such longer-running operations are ongoing. It also keeps more state about user's choices (e.g. windows sizes, whether the tree or the code view was last used in the document editor).

Cluster statistics are now integrated into the web interface as well. Additionally, a menu item "Help us" has been added to easily provide the ArangoDB team feedback about the product.

The frontend may now be mounted behind a reverse proxy on a different path. For this to work the proxy should send a X-Script-Name header containing the path.

A backend configuration for haproxy might look like this:

```
reqadd X-Script-Name:\ /arangodb
```

The frontend will recognize the subpath and produce appropriate links. ArangoDB will only accept paths from trusted frontend proxies. Trusted proxies may be added on startup:

```
--frontend.proxy-request-check true --frontend.trusted-proxy 192.168.1.117
```

--frontend.trusted-proxy may be any address or netmask.

To disable the check and blindly accept any x-script-name set --frontend.proxy-request-check to false.

Foxx improvements

The Foxx framework has been completely rewritten for 3.0 with a new, simpler and more familiar API. The most notable changes are:

- Legacy mode for 2.8 services

  Stuck with old code? You can continue using your 2.8-compatible Foxx services with 3.0 by adding `"engines": {"arangodb": "^2.8.0"}` (or similar version ranges that exclude 3.0 and up) to the service manifest.

- No more global variables and magical comments

  The `applicationContext` is now `module.context`. Instead of magical comments just use the `summary` and `description` methods to document your routes.

- Repository and Model have been removed

  Instead of repositories just use ArangoDB collections directly. For validation simply use the joi schemas (but wrapped in `joi.object()`) that previously lived inside the model. Collections and queries return plain JavaScript objects.

- Controllers have been replaced with nestable routers

  Create routers with `require('@arangodb/foxx/router')()` , attach them to your service with `module.context.use(router)` . Because routers are no longer mounted automagically, you can export and import them like any other object. Use `router.use('/path', subRouter)` to nest routers as deeply as you want.

- Routes can be named and reversed

  No more memorizing URLs: add a name to your route like `router.get('/hello/:name', function () {...}, 'hello')` and redirect to the full URL with `res.redirect(req.resolve('hello', {name: 'world'}))` .

- Simpler express-like middleware

  If you already know express, this should be familiar. Here's a request logger in three lines of code:

  ```
  router.use(function (req, res, next) {
    var start = Date.now();
    try {next();}
    finally {console.log(`${req.method} ${req.url} ${res.statusCode} ${Date.now() - start}ms`);}
  });
  ```

- Sessions and auth without dependencies

  To make it easier to get started, the functionality previously provided by the `simple-auth` , `oauth2` , `sessions-local` and `sessions-jwt` services have been moved into Foxx as the `@arangodb/foxx/auth` , `@arangodb/foxx/oauth2` and `@arangodb/foxx/sessions` modules.

Logging

ArangoDB's logging is now grouped into topics. The log verbosity and output files can be adjusted per log topic. For example

```
--log.level startup=trace --log.level queries=trace --log.level info
```

will log messages concerning startup at trace level, AQL queries at trace level and everything else at info level. `--log.level` can be specified multiple times at startup, for as many topics as needed.

Some relevant log topics available in 3.0 are:

- *collector*: information about the WAL collector's state
- *compactor*: information about the collection datafile compactor
- *datafiles*: datafile-related operations
- *mmap*: information about memory-mapping operations (including msync)
- *queries*: executed AQL queries, slow queries
- *replication*: replication-related info
- *requests*: HTTP requests
- *startup*: information about server startup and shutdown
- *threads*: information about threads

This also allows directing log output to different files based on topics. For example, to log all AQL queries to a file "queries.log" one can use the options:

```
--log.level queries=trace --log.output queries=file:///path/to/queries.log
```

To additionally log HTTP request to a file named "requests.log" add the options:

```
--log.level requests=info --log.output requests=file:///path/to/requests.log
```

Build system

ArangoDB now uses the cross-platform build system CMake for all its builds. Previous versions used two different build systems, making development and contributions harder than necessary. Now the build system is unified, and all targets (Linux, Windows, MacOS) are built from the same set of build instructions.

Documentation

The documentation has been enhanced and re-organized to be more intuitive.

A new introduction for beginners should bring you up to speed with ArangoDB in less than an hour. Additional topics have been introduced and will be extended with upcoming releases.

The topics AQL and HTTP API are now separated from the manual for better searchability and less confusion. A version switcher makes it easier to jump to the version of the docs you are interested in.

Incompatible changes in ArangoDB 3.0

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 3.0, and adjust any client programs if necessary.

Build system

Building ArangoDB 3.0 from source now requires CMake.

The pre-3.0 build system used a configure-based approach. The steps to build ArangoDB 2.8 from source code were:

```
make setup
./configure <options>
make
```

These steps will not work anymore, as ArangoDB 3.0 does not come with a configure script.

To build 3.0 on Linux, create a separate build directory first:

```
mkdir -p build
```

and then create the initial build scripts once using CMake:

```
(cd build && cmake <options> ..)
```

The above command will configure the build and check for the required dependencies. If everything works well the actual build can be started with

```
(cd build && make)
```

The binaries for the ArangoDB server and all client tools will then be created inside the `build` directory. To start ArangoDB locally from the `build` directory, use

```
build/bin/arangod <options>
```

Datafiles and datafile names

ArangoDB 3.0 uses a new VelocyPack-based format for storing data in WAL logfiles and collection datafiles. The file format is not compatible with the files used in prior versions of ArangoDB. That means datafiles written by ArangoDB 3.0 cannot be used in earlier versions and vice versa.

The pattern for collection directory names was changed in 3.0 to include a random id component at the end. The new pattern is `collection-<id>-<random>`, where `<id>` is the collection id and `<random>` is a random number. Previous versions of ArangoDB used a pattern `collection-<id>` without the random number.

User Management

Unlike ArangoDB 2.x, ArangoDB 3.0 users are now separated from databases, and you can grant one or more database permissions to a user.

If you want to mimic the behavior of ArangoDB, you should name your users like `username@dbname`.

Users that can access the *_system* database are allowed to manage users and permissions for all databases.

Edges and edges attributes

In ArangoDB prior to 3.0 the attributes `_from` and `_to` of edges were treated specially when loading or storing edges. That special handling led to these attributes being not as flexible as regular document attributes. For example, the `_from` and `_to` attribute values of an existing edge could not be updated once the edge was created. Additionally, the `_from` and `_to` attributes could not be indexed in user-defined indexes, e.g. to make each combination of `_from` and `_to` unique. Finally, as `_from` and `_to` referenced the linked collections by collection id and not by collection name, their meaning became unclear once a referenced collection was dropped. The collection id stored in edges then became unusable, and when accessing such edge the collection name part of it was always translated to `_undefined`.

In ArangoDB 3.0, the `_from` and `_to` values of edges are saved as regular strings. This allows using `_from` and `_to` in user-defined indexes. Additionally this allows updating the `_from` and `_to` values of existing edges. Furthermore, collections referenced by `_from` and `_to` values may be dropped and re-created later. Any `_from` and `_to` values of edges pointing to such dropped collection are unaffected by the drop operation now. Also note that renaming the collection referenced in `_from` and `_to` in ArangoDB 2.8 also relinked the edges. In 3.0 the edges are NOT automatically relinked to the new collection anymore.

Documents

Documents (in contrast to edges) cannot contain the attributes `_from` or `_to` on the main level in ArangoDB 3.0. These attributes will be automatically removed when saving documents (i.e. non-edges). `_from` and `_to` can be still used in sub-objects inside documents.

The `_from` and `_to` attributes will of course be preserved and are still required when saving edges.

AQL

Edges handling

When updating or replacing edges via AQL, any modifications to the `_from` and `_to` attributes of edges were ignored by previous versions of ArangoDB, without signaling any errors. This was due to the `_from` and `_to` attributes being immutable in earlier versions of ArangoDB.

From 3.0 on, the `_from` and `_to` attributes of edges are mutable, so any AQL queries that modify the `_from` or `_to` attribute values of edges will attempt to actually change these attributes. Clients should be aware of this change and should review their queries that modify edges to rule out unintended side-effects.

Additionally, when completely replacing the data of existing edges via the AQL `REPLACE` operation, it is now required to specify values for the `_from` and `_to` attributes, as `REPLACE` requires the entire new document to be specified. If either `_from` or `_to` are missing from the replacement document, an `REPLACE` operation will fail.

Graph functions

In version 3.0 all former graph related functions have been removed from AQL to be replaced by native AQL constructs. These constructs allow for more fine-grained filtering on several graph levels. Also this allows the AQL optimizer to automatically improve these queries by enhancing them with appropriate indexes. We have created recipes to upgrade from 2.8 to 3.0 when using these functions.

The functions:

- GRAPH_COMMON_NEIGHBORS
- GRAPH_COMMON_PROPERTIES
- GRAPH_DISTANCE_TO
- GRAPH_EDGES
- GRAPH_NEIGHBORS
- GRAPH_TRAVERSAL
- GRAPH_TRAVERSAL_TREE
- GRAPH_SHORTEST_PATH
- GRAPH_PATHS
- GRAPH_VERTICES

are covered in Migrating GRAPH_* Functions from 2.8 or earlier to 3.0

- GRAPH_ABSOLUTE_BETWEENNESS
- GRAPH_ABSOLUTE_CLOSENESS
- GRAPH_ABSOLUTE_ECCENTRICITY
- GRAPH_BETWEENNESS
- GRAPH_CLOSENESS
- GRAPH_DIAMETER
- GRAPH_ECCENTRICITY
- GRAPH_RADIUS

are covered in Migrating GRAPH_* Measurements from 2.8 or earlier to 3.0

- EDGES
- NEIGHBORS
- PATHS
- TRAVERSAL
- TRAVERSAL_TREE

are covered in Migrating anonymous graph functions from 2.8 or earlier to 3.0

Typecasting functions

The type casting applied by the `TO_NUMBER()` AQL function has changed as follows:

- string values that do not contain a valid numeric value are now converted to the number `0`. In previous versions of ArangoDB such string values were converted to the value `null`.
- array values with more than 1 member are now converted to the number `0`. In previous versions of ArangoDB such arrays were converted to the value `null`.
- objects / documents are now converted to the number `0`. In previous versions of ArangoDB objects / documents were converted to the value `null`.

Additionally, the `TO_STRING()` AQL function now converts `null` values into an empty string (`""`) instead of the string `"null"`, which is more in line with `LENGTH(null)` returning `0` and not `4` since v2.6.

The output of `TO_STRING()` has also changed for arrays and objects as follows:

- arrays are now converted into their JSON-stringify equivalents, e.g.
  - `[ ]` is now converted to `[]`
  - `[ 1, 2, 3 ]` is now converted to `[1,2,3]`
  - `[ "test", 1, 2 ] is now converted to `["test",1,2]`

  Previous versions of ArangoDB converted arrays with no members into the empty string, and non-empty arrays into a comma-separated list of member values, without the surrounding angular brackets. Additionally, string array members were not enclosed in quotes in the result string:

  - `[ ]` was converted to ``
  - `[ 1, 2, 3 ]` was converted to `1,2,3`
  - `[ "test", 1, 2 ] was converted to `test,1,2`
- objects are now converted to their JSON-stringify equivalents, e.g.
  - `{ }` is converted to `{}`
  - `{ a: 1, b: 2 }` is converted to `{"a":1,"b":2}`
  - `{ "test" : "foobar" }` is converted to `{"test":"foobar"}`

  Previous versions of ArangoDB always converted objects into the string `[object Object]`

This change also affects other parts in AQL that used `TO_STRING()` to implicitly cast operands to strings. It also affects the AQL functions `CONCAT()` and `CONCAT_SEPARATOR()` which treated array values differently. Previous versions of ArangoDB automatically flattened array values in the first level of the array, e.g. `CONCAT([1, 2, 3, [ 4, 5, 6 ]])` produced `1,2,3,4,5,6`. Now this will produce `[1,2,3,[4,5,6]]`. To flatten array members on the top level, you can now use the more explicit `CONCAT(FLATTEN([1, 2, 3, [4, 5, 6]], 1))`.

Arithmetic operators

As the arithmetic operations in AQL implicitly convert their operands to numeric values using `TO_NUMBER()` , their casting behavior has also changed as described above.

Some examples of the changed behavior:

- `"foo" + 1` produces `1` now. In previous versions this produced `null` .
- `[ 1, 2 ] + 1` produces `1` . In previous versions this produced `null` .
- `1 + "foo" + 1´ produces `2`now. In previous version this produced` 1`.

Attribute names and parameters

Previous versions of ArangoDB had some trouble with attribute names that contained the dot symbol (`.`). Some code parts in AQL used the dot symbol to split an attribute name into sub-components, so an attribute named `a.b` was not completely distinguishable from an attribute `a` with a sub-attribute `b` . This inconsistent behavior sometimes allowed "hacks" to work such as passing sub-attributes in a bind parameter as follows:

```
FOR doc IN collection
  FILTER doc.@name == 1
  RETURN doc
```

If the bind parameter `@name` contained the dot symbol (e.g. `@bind` = `a.b` , it was unclear whether this should trigger sub-attribute access (i.e. `doc.a.b`) or a access to an attribute with exactly the specified name (i.e. `doc["a.b"]`).

ArangoDB 3.0 now handles attribute names containing the dot symbol properly, and sending a bind parameter `@name` = `a.b` will now always trigger an access to the attribute `doc["a.b"]` , not the sub-attribute `b` of `a` in `doc` .

For users that used the "hack" of passing bind parameters containing dot symbol to access sub-attributes, ArangoDB 3.0 allows specifying the attribute name parts as an array of strings, e.g. `@name` = `[ "a", "b" ]` , which will be resolved to the sub-attribute access `doc.a.b` when the query is executed.

Keywords

`LIKE` is now a keyword in AQL. Using `LIKE` in either case as an attribute or collection name in AQL queries now requires quoting.

`SHORTEST_PATH` is now a keyword in AQL. Using `SHORTEST_PATH` in either case as an attribute or collection name in AQL queries now requires quoting.

Subqueries

Queries that contain subqueries that contain data-modification operations such as `INSERT` , `UPDATE` , `REPLACE` , `UPSERT` or `REMOVE` will now refuse to execute if the collection affected by the subquery's data-modification operation is read-accessed in an outer scope of the query.

For example, the following query will refuse to execute as the collection `myCollection` is modified in the subquery but also read-accessed in the outer scope:

```
FOR doc IN myCollection
  LET changes = (
    FOR what IN myCollection
      FILTER what.value == 1
      REMOVE what IN myCollection
  )
  RETURN doc
```

It is still possible to write to collections from which data is read in the same query, e.g.

```
FOR doc IN myCollection
  FILTER doc.value == 1
  REMOVE doc IN myCollection
```

and to modify data in different collection via subqueries.

Other changes

The AQL optimizer rule "merge-traversal-filter" that already existed in 3.0 was renamed to "optimize-traversals". This should be of no relevance to client applications except if they programatically look for applied optimizer rules in the explain out of AQL queries.

The order of results created by the AQL functions `VALUES()` and `ATTRIBUTES()` was never guaranteed and it only had the "correct" ordering by accident when iterating over objects that were not loaded from the database. As some of the function internals have changed, the "correct" ordering will not appear anymore, and still no result order is guaranteed by these functions unless the `sort` parameter is specified (for the `ATTRIBUTES()` function).

Upgraded V8 version

The V8 engine that is used inside ArangoDB to execute JavaScript code has been upgraded from version 4.3.61 to 5.0.71.39. The new version should be mostly compatible to the old version, but there may be subtle differences, including changes of error message texts thrown by the engine. Furthermore, some V8 startup parameters have changed their meaning or have been removed in the new version. This is only relevant when ArangoDB or ArangoShell are started with a custom value for the `--javascript.v8-options` startup option.

Among others, the following V8 options change in the new version of ArangoDB:

- `--es_staging` : in 2.8 it had the meaning `enable all completed harmony features`, in 3.0 the option means `enable test-worthy harmony features (for internal use only)`
- `--strong_this` : this option wasn't present in 2.8. In 3.0 it means `don't allow 'this' to escape from constructors` and defaults to true.
- `--harmony_regexps` : this options means `enable "harmony regular expression extensions"` and changes its default value from false to true
- `--harmony_proxies` : this options means `enable "harmony proxies"` and changes its default value from false to true
- `--harmony_reflect` : this options means `enable "harmony Reflect API"` and changes its default value from false to true
- `--harmony_sloppy` : this options means `enable "harmony features in sloppy mode"` and changes its default value from false to true
- `--harmony_tostring` : this options means `enable "harmony toString"` and changes its default value from false to true
- `--harmony_unicode_regexps` : this options means `enable "harmony unicode regexps"` and changes its default value from false to true
- `--harmony_arrays` , `--harmony_array_includes` , `--harmony_computed_property_names` , `--harmony_arrow_functions` , `--harmony_rest_parameters` , `--harmony_classes` , `--harmony_object_literals` , `--harmony_numeric_literals` , `--harmony_unicode` : these option have been removed in V8 5.

As a consequence of the upgrade to V8 version 5, the implementation of the JavaScript `Buffer` object had to be changed. JavaScript `Buffer` objects in ArangoDB now always store their data on the heap. There is no shared pool for small Buffer values, and no pointing into existing Buffer data when extracting slices. This change may increase the cost of creating Buffers with short contents or when peeking into existing Buffers, but was required for safer memory management and to prevent leaks.

JavaScript API changes

The following incompatible changes have been made to the JavaScript API in ArangoDB 3.0:

Foxx

The Foxx framework has been completely rewritten for 3.0 with a new, simpler and more familiar API. To make Foxx services developed for 2.8 or earlier ArangoDB versions run in 3.0, the service's manifest file needs to be edited.

To enable the legacy mode for a Foxx service, add `"engines": {"arangodb": "^2.8.0"}` (or similar version ranges that exclude 3.0 and up) to the service manifest file (named "manifest.json", located in the service's base directory).

Require

Modules shipped with ArangoDB can now be required using the pattern `@arangodb/<module>` instead of `org/arangodb/<module>` , e.g.

```
var cluster = require("@arangodb/cluster");
```

The old format can still be used for compatibility:

```
var cluster = require("org/arangodb/cluster");
```

ArangoDB prior to version 3.0 allowed a transparent use of CoffeeScript source files with the `require()` function. Files with a file name extension of `coffee` were automatically sent through a CoffeeScript parser and transpiled into JavaScript on-the-fly. This support is gone with ArangoDB 3.0. To run any CoffeeScript source files, they must be converted to JavaScript by the client application.

Response object

The `@arangodb/request` response object now stores the parsed JSON response body in a property `json` instead of `body` when the request was made using the `json` option. The `body` instead contains the response body as a string.

Edges API

When completely replacing an edge via a collection's `replace()` function the replacing edge data now needs to contain the `_from` and `_to` attributes for the new edge. Previous versions of ArangoDB did not require the edge data to contain `_from` and `_to` attributes when replacing an edge, since `_from` and `_to` values were immutable for existing edges.

For example, the following call worked in ArangoDB 2.8 but will fail in 3.0:

```
db.edgeCollection.replace("myKey", { value: "test" });
```

To make this work in ArangoDB 3.0, `_from` and `_to` need to be added to the replacement data:

```
db.edgeCollection.replace("myKey", { _from: "myVertexCollection/1", _to: "myVertexCollection/2", value: "test" });
```

Note that this only affects the `replace()` function but not `update()` , which will only update the specified attributes of the edge and leave all others intact.

Additionally, the functions `edges()` , `outEdges()` and `inEdges()` with an array of edge ids will now make the edge ids unique before returning the connected edges. This is probably desired anyway, as results will be returned only once per distinct input edge id. However, it may break client applications that rely on the old behavior.

Databases API

The `_listDatabases()` function of the `db` object has been renamed to `_databases()` , making it consistent with the `_collections()` function. Also the `_listEndpoints()` function has been renamed to `_endpoints()` .

Collection API

Example matching

The collection function `byExampleHash()` and `byExampleSkiplist()` have been removed in 3.0. Their functionality is provided by collection's `byExample()` function, which will automatically use a suitable index if present.

The collection function `byConditionSkiplist()` has been removed in 3.0. The same functionality can be achieved by issuing an AQL query with the target condition, which will automatically use a suitable index if present.

Revision id handling

The `exists()` method of a collection now throws an exception when the specified document exists but its revision id does not match the revision id specified. Previous versions of ArangoDB simply returned `false` if either no document existed with the specified key or when the revision id did not match. It was therefore impossible to distinguish these two cases from the return value alone. 3.0 corrects this. Additionally, `exists()` in previous versions always returned a boolean if only the document key was given. 3.0 now returns the document's meta-data, which includes the document's current revision id.

Given there is a document with key `test` in collection `myCollection` , then the behavior of 3.0 is as follows:

```
/* test if document exists. this returned true in 2.8 */
db.myCollection.exists("test");
{
  "_key" : "test",
  "_id" : "myCollection/test",
  "_rev" : "9758059"
}

/* test if document exists. this returned true in 2.8 */
db.myCollection.exists({ _key: "test" });
{
  "_key" : "test",
  "_id" : "myCollection/test",
  "_rev" : "9758059"
}

/* test if document exists. this also returned false in 2.8 */
db.myCollection.exists("foo");
false

/* test if document with a given revision id exists. this returned true in 2.8 */
db.myCollection.exists({ _key: "test", _rev: "9758059" });
{
  "_key" : "test",
  "_id" : "myCollection/test",
  "_rev" : "9758059"
}

/* test if document with a given revision id exists. this returned false in 2.8 */
db.myCollection.exists({ _key: "test", _rev: "1234" });
JavaScript exception: ArangoError 1200: conflict
```

Cap constraints

The cap constraints feature has been removed. This change has led to the removal of the collection operations `first()` and `last()` , which were internally based on data from cap constraints.

As cap constraints have been removed in ArangoDB 3.0 it is not possible to create an index of type "cap" with a collection's `ensureIndex()` function. The dedicated function `ensureCapConstraint()` has also been removed from the collection API.

Graph Blueprints JS Module

The deprecated module `graph-blueprints` has been deleted. All it's features are covered by the `general-graph` module.

General Graph Fluent AQL interface

The fluent interface has been removed from ArangoDB. It's features were completely overlapping with "aqb" which comes pre installed as well. Please switch to AQB instead.

Undocumented APIs

The undocumented functions `BY_EXAMPLE_HASH()` and `BY_EXAMPLE_SKIPLIST()` , `BY_CONDITION_SKIPLIST` , `CPP_NEIGHBORS` and `CPP_SHORTEST_PATH` have been removed. These functions were always hidden and not intended to be part of the public JavaScript API for collections.

HTTP API changes

CRUD operations

The following incompatible changes have been made to the HTTP API in ArangoDB 3.0:

General

The HTTP insert operations for single documents and edges (POST `/_api/document`) do not support the URL parameter "createCollection" anymore. In previous versions of ArangoDB this parameter could be used to automatically create a collection upon insertion of the first document. It is now required that the target collection already exists when using this API, otherwise it will return an HTTP 404 error. The same is true for the import API at POST `/_api/import`.

Collections can still be created easily via a separate call to POST `/_api/collection` as before.

The "location" HTTP header returned by ArangoDB when inserting a new document or edge now always contains the database name. This was also the default behavior in previous versions of ArangoDB, but it could be overridden by clients sending the HTTP header `x-arango-version: 1.4` in the request. Clients can continue to send this header to ArangoDB 3.0, but the header will not influence the location response headers produced by ArangoDB 3.0 anymore.

Additionally the CRUD operations APIs do not return an attribute "error" in the response body with an attribute value of "false" in case an operation succeeded.

Revision id handling

The operations for updating, replacing and removing documents can optionally check the revision number of the document to be updated, replaced or removed so the caller can ensure the operation works on a specific version of the document and there are no lost updates.

Previous versions of ArangoDB allowed passing the revision id of the previous document either in the HTTP header `If-Match` or in the URL parameter `rev`. For example, removing a document with a specific revision id could be achieved as follows:

```
curl -X DELETE \
     "http://127.0.0.1:8529/_api/document/myCollection/myKey?rev=123"
```

ArangoDB 3.0 does not support passing the revision id via the "rev" URL parameter anymore. Instead the previous revision id must be passed in the HTTP header `If-Match`, e.g.

```
curl -X DELETE \
     --header "If-Match: '123'" \
     "http://127.0.0.1:8529/_api/document/myCollection/myKey"
```

The URL parameter "policy" was also usable in previous versions of ArangoDB to control revision handling. Using it was redundant to specifying the expected revision id via the "rev" parameter or "If-Match" HTTP header and therefore support for the "policy" parameter was removed in 3.0.

In order to check for a previous revision id when updating, replacing or removing documents please use the `If-Match` HTTP header as described above. When no revision check if required the HTTP header can be omitted, and the operations will work on the current revision of the document, regardless of its revision id.

All documents API

The HTTP API for retrieving the ids, keys or URLs of all documents from a collection was previously located at GET `/_api/document?collection=...`. This API was moved to PUT `/_api/simple/all-keys` and is now executed as an AQL query. The name of the collection must now be passed in the HTTP request body instead of in the request URL. The same is true for the "type" parameter, which controls the type of the result to be created.

Calls to the previous API can be translated as follows:

- old: GET `/_api/document?collection=<collection>&type=<type>` without HTTP request body
- 3.0: PUT `/_api/simple/all-keys` with HTTP request body `{"collection":"<collection>","type":"id"}`

The result format of this API has also changed slightly. In previous versions calls to the API returned a JSON object with a `documents` attribute. As the functionality is based on AQL internally in 3.0, the API now returns a JSON object with a `result` attribute.

Edges API

CRUD operations

The API for documents and edges have been unified in ArangoDB 3.0. The CRUD operations for documents and edges are now handled by the same endpoint at `/_api/document`. For CRUD operations there is no distinction anymore between documents and edges API-wise.

That means CRUD operations concerning edges need to be sent to the HTTP endpoint `/_api/document` instead of `/_api/edge`. Sending requests to `/_api/edge` will result in an HTTP 404 error in 3.0. The following methods are available at `/_api/document` for documents and edge:

- HTTP POST: insert new document or edge
- HTTP GET: fetch an existing document or edge
- HTTP PUT: replace an existing document or edge
- HTTP PATCH: partially update an existing document or edge
- HTTP DELETE: remove an existing document or edge

When completely replacing an edge via HTTP PUT please note that the replacing edge data now needs to contain the `_from` and `_to` attributes for the edge. Previous versions of ArangoDB did not require sending `_from` and `_to` when replacing edges, as `_from` and `_to` values were immutable for existing edges.

The `_from` and `_to` attributes of edges now also need to be present inside the edges objects sent to the server:

```
curl -X POST \
     --data '{"value":1,"_from":"myVertexCollection/1","_to":"myVertexCollection/2"}' \
     "http://127.0.0.1:8529/_api/document?collection=myEdgeCollection"
```

Previous versions of ArangoDB required the `_from` and `_to` attributes of edges be sent separately in URL parameter `from` and `to`:

```
curl -X POST \
     --data '{"value":1}' \
     "http://127.0.0.1:8529/_api/edge?collection=e&from=myVertexCollection/1&to=myVertexCollection/2"
```

Querying connected edges

The REST API for querying connected edges at GET `/_api/edges/<collection>` will now make the edge ids unique before returning the connected edges. This is probably desired anyway as results will now be returned only once per distinct input edge id. However, it may break client applications that rely on the old behavior.

Graph API

Some data-modification operations in the named graphs API at `/_api/gharial` now return either HTTP 202 (Accepted) or HTTP 201 (Created) if the operation succeeds. Which status code is returned depends on the `waitForSync` attribute of the affected collection. In previous versions some of these operations return HTTP 200 regardless of the `waitForSync` value.

The deprecated graph API `/_api/graph` has been removed. All it's features can be replaced using `/_api/gharial` and AQL instead.

Simple queries API

The REST routes PUT `/_api/simple/first` and `/_api/simple/last` have been removed entirely. These APIs were responsible for returning the first-inserted and last-inserted documents in a collection. This feature was built on cap constraints internally, which have been removed in 3.0.

Calling one of these endpoints in 3.0 will result in an HTTP 404 error.

Indexes API

It is not supported in 3.0 to create an index with type `cap` (cap constraint) in 3.0 as the cap constraints feature has bee removed. Calling the index creation endpoint HTTP API POST `/_api/index?collection=...` with an index type `cap` will therefore result in an HTTP 400 error.

Log entries API

The REST route HTTP GET `/_admin/log` is now accessible from within all databases. In previous versions of ArangoDB, this route was accessible from within the `_system` database only, and an HTTP 403 (Forbidden) was thrown by the server for any access from within another database.

Figures API

The REST route HTTP GET `/_api/collection/<collection>/figures` will not return the following result attributes as they became meaningless in 3.0:

- shapefiles.count
- shapes.fileSize
- shapes.count
- shapes.size
- attributes.count
- attributes.size

Databases and Collections APIs

When creating a database via the API POST `/_api/database`, ArangoDB will now always return the HTTP status code 202 (created) if the operation succeeds. Previous versions of ArangoDB returned HTTP 202 as well, but this behavior was changable by sending an HTTP header `x-arango-version: 1.4`. When sending this header, previous versions of ArangoDB returned an HTTP status code 200 (ok). Clients can still send this header to ArangoDB 3.0 but this will not influence the HTTP status code produced by ArangoDB.

The "location" header produced by ArangoDB 3.0 will now always contain the database name. This was also the default in previous versions of ArangoDB, but the behavior could be overridden by sending the HTTP header `x-arango-version: 1.4`. Clients can still send the header, but this will not make the database name in the "location" response header disappear.

The result format for querying all collections via the API GET `/_api/collection` has been changed.

Previous versions of ArangoDB returned an object with an attribute named `collections` and an attribute named `names`. Both contained all available collections, but `collections` contained the collections as an array, and `names` contained the collections again, contained in an object in which the attribute names were the collection names, e.g.

```
{
  "collections": [
    {"id":"5874437","name":"test","isSystem":false,"status":3,"type":2},
    {"id":"17343237","name":"something","isSystem":false,"status":3,"type":2},
    ...
  ],
  "names": {
    "test": {"id":"5874437","name":"test","isSystem":false,"status":3,"type":2},
    "something": {"id":"17343237","name":"something","isSystem":false,"status":3,"type":2},
    ...
  }
}
```

This result structure was redundant, and therefore has been simplified to just

```
{
  "result": [
    {"id":"5874437","name":"test","isSystem":false,"status":3,"type":2},
    {"id":"17343237","name":"something","isSystem":false,"status":3,"type":2},
    ...
  ]
}
```

in ArangoDB 3.0.

Replication APIs

The URL parameter "failOnUnknown" was removed from the REST API GET `/_api/replication/dump`. This parameter controlled whether dumping or replicating edges should fail if one of the vertex collections linked in the edge's `_from` or `_to` attributes was not present anymore. In this case the `_from` and `_to` values could not be translated into meaningful ids anymore.

There were two ways for handling this:

- setting `failOnUnknown` to `true` caused the HTTP request to fail, leaving error handling to the user
- setting `failOnUnknown` to `false` caused the HTTP request to continue, translating the collection name part in the `_from` or `_to` value to `_unknown`.

In ArangoDB 3.0 this parameter is obsolete, as `_from` and `_to` are stored as self-contained string values all the time, so they cannot get invalid when referenced collections are dropped.

The result format of the API GET `/_api/replication/logger-follow` has changed slightly in the following aspects:

- documents and edges are reported in the same way. The type for document insertions/updates and edge insertions/updates is now always `2300`. Previous versions of ArangoDB returned a `type` value of `2300` for documents and `2301` for edges.
- records about insertions, updates or removals of documents and edges do not have the `key` and `rev` attributes on the top-level anymore. Instead, `key` and `rev` can be accessed by peeking into the `_key` and `_rev` attributes of the `data` sub-attributes of the change record.

The same is true for the collection-specific changes API GET `/_api/replication/dump`.

User management APIs

The REST API endpoint POST `/_api/user` for adding new users now requires the request to contain a JSON object with an attribute named `user`, containing the name of the user to be created. Previous versions of ArangoDB also checked this attribute, but additionally looked for an attribute `username` if the `user` attribute did not exist.

Undocumented APIs

The following undocumented HTTP REST endpoints have been removed from ArangoDB's REST API:

- `/_open/cerberus` and `/_system/cerberus`: these endpoints were intended for some ArangoDB-internal applications only
- PUT `/_api/simple/by-example-hash`, PUT `/_api/simple/by-example-skiplist` and PUT `/_api/simple/by-condition-skiplist`: these methods were documented in early versions of ArangoDB but have been marked as not intended to be called by end users since ArangoDB version 2.3. These methods should not have been part of any ArangoDB manual since version 2.4.
- `/_api/structure`: an older unfinished and unpromoted API for data format and type checks, superseded by Foxx applications.

Administration APIs

- `/_admin/shutdown` now needs to be called with the HTTP DELETE method

Handling of CORS requests

It can now be controlled in detail for which origin hosts CORS (Cross-origin resource sharing) requests with credentials will be allowed. ArangoDB 3.0 provides the startup option `--http.trusted-origin` that can be used to specify one or many origins from which CORS requests are treated as "trustworthy".

The option can be specified multiple times, once per trusted origin, e.g.

```
--http.trusted-origin http://127.0.0.1:8529 --http.trusted-origin https://127.0.0.1:8599
```

This will make the ArangoDB server respond to CORS requests from these origins with an `Access-Control-Allow-Credentials` HTTP header with a value of `true`. Web browsers can inspect this header and can allow passing ArangoDB web interface credentials (if stored in the browser) to the requesting site. ArangoDB will not forward or provide any credentials.

Setting this option is only required if applications on other hosts need to access the ArangoDB web interface or other HTTP REST APIs from a web browser with the same credentials that the user has entered when logging into the web interface. When a web browser finds the `Access-Control-Allow-Credentials` HTTP response header, it may forward the credentials entered into the browser for the ArangoDB web interface login to the other site.

This is a potential security issue, so there are no trusted origins by default. It may be required to set some trusted origins if you're planning to issue AJAX requests to ArangoDB from other sites from the browser, with the credentials entered during the ArangoDB interface login (i.e. single sign-on). If such functionality is not used, the option should not be set.

To specify a trusted origin, specify the option once per trusted origin as shown above. Note that the trusted origin values specified in this option will be compared bytewise with the `Origin` HTTP header value sent by clients, and only exact matches will pass.

There is also the wildcard `all` for enabling CORS access from all origins in a test or development setup:

```
--http.trusted-origin all
```

Setting this option will lead to the ArangoDB server responding with an `Access-Control-Allow-Credentials: true` HTTP header to all incoming CORS requests.

Command-line options

Quite a few startup options in ArangoDB 2 were double negations (like `--server.disable-authentication false`). In ArangoDB 3 these are now expressed as positives (e. g. `--server.authentication`). Also the options between the ArangoDB server and its client tools have being unified. For example, the logger options are now the same for the server and the client tools. Additionally many options have been moved into more appropriate topic sections.

Renamed options

The following options have been available before 3.0 and have changed their name in 3.0:

- `--server.disable-authentication` was renamed to `--server.authentication`. Note that the meaning of the option `--server.authentication` is the opposite of the previous `--server.disable-authentication`.
- `--server.disable-authentication-unix-sockets` was renamed to `--server.authentication-unix-sockets`. Note that the meaning of the option `--server.authentication-unix-sockets` is the opposite of the previous `--server.disable-authentication-unix-sockets`.
- `--server.authenticate-system-only` was renamed to `--server.authentication-system-only`. The meaning of the option in unchanged.
- `--server.disable-statistics` was renamed to `--server.statistics`. Note that the meaning of the option `--server.statistics` is the opposite of the previous `--server.disable-statistics`.
- `--server.cafile` was renamed to `--ssl.cafile`. The meaning of the option is unchanged.
- `--server.keyfile` was renamed to `--ssl.keyfile`. The meaning of the option is unchanged.
- `--server.ssl-cache` was renamed to `--ssl.session-cache`. The meaning of the option is unchanged.
- `--server.ssl-cipher-list` was renamed to `--ssl.cipher-list`. The meaning of the option is unchanged.
- `--server.ssl-options` was renamed to `--ssl.options`. The meaning of the option is unchanged.
- `--server.ssl-protocol` was renamed to `--ssl.protocol`. The meaning of the option is unchanged.
- `--server.backlog-size` was renamed to `--tcp.backlog-size`. The meaning of the option is unchanged.
- `--server.reuse-address` was renamed to `--tcp.reuse-address`. The meaning of the option is unchanged.
- `--server.disable-replication-applier` was renamed to `--database.replication-applier`. The meaning of the option `--database.replication-applier` is the opposite of the previous `--server.disable-replication-applier`.
- `--server.allow-method-override` was renamed to `--http.allow-method-override`. The meaning of the option is unchanged.
- `--server.hide-product-header` was renamed to `--http.hide-product-header`. The meaning of the option is unchanged.
- `--server.keep-alive-timeout` was renamed to `--http.keep-alive-timeout`. The meaning of the option is unchanged.
- `--server.foxx-queues` was renamed to `--foxx.queues`. The meaning of the option is unchanged.
- `--server.foxx-queues-poll-interval` was renamed to `--foxx.queues-poll-interval`. The meaning of the option is unchanged.
- `--no-server` was renamed to `--server.rest-server`. Note that the meaning of the option `--server.rest-server` is the opposite of the previous `--no-server`.
- `--database.query-cache-mode` was renamed to `--query.cache-mode`. The meaning of the option is unchanged.

- `--database.query-cache-max-results` was renamed to `--query.cache-entries`. The meaning of the option is unchanged.
- `--database.disable-query-tracking` was renamed to `--query.tracking`. The meaning of the option `--query.tracking` is the opposite of the previous `--database.disable-query-tracking`.
- `--log.tty` was renamed to `--log.foreground-tty`. The meaning of the option is unchanged.
- `--upgrade` has been renamed to `--database.auto-upgrade`. In contrast to 2.8 this option now requires a boolean parameter. To actually perform an automatic database upgrade at startup use `--database.auto-upgrade true`. To not perform it, use `--database.auto-upgrade false`.
- `--check-version` has been renamed to `--database.check-version`.
- `--temp-path` has been renamed to `--temp.path`.

Log verbosity, topics and output files

Logging now supports log topics. You can control these by specifying a log topic in front of a log level or an output. For example

```
--log.level startup=trace --log.level info
```

will log messages concerning startup at trace level, everything else at info level. `--log.level` can be specified multiple times at startup, for as many topics as needed.

Some relevant log topics available in 3.0 are:

- *collector*: information about the WAL collector's state
- *compactor*: information about the collection datafile compactor
- *datafiles*: datafile-related operations
- *mmap*: information about memory-mapping operations
- *performance*: some performance-related information
- *queries*: executed AQL queries
- *replication*: replication-related info
- *requests*: HTTP requests
- *startup*: information about server startup and shutdown
- *threads*: information about threads

The new log option `--log.output <definition>` allows directing the global or per-topic log output to different outputs. The output definition "" can be one of

- "-" for stdin
- "+" for stderr
- "syslog://"
- "syslog:///"
- "file://"

The option can be specified multiple times in order to configure the output for different log topics. To set up a per-topic output configuration, use `--log.output <topic>=<definition>`, e.g.

queries=file://queries.txt

logs all queries to the file "queries.txt".

The old option `--log.file` is still available in 3.0 for convenience reasons. In 3.0 it is a shortcut for the more general option `--log.output file://filename`.

The old option `--log.requests-file` is still available in 3.0. It is now a shortcut for the more general option `--log.output requests=file://...`.

The old option `--log.performance` is still available in 3.0. It is now a shortcut for the more general option `--log.level performance=trace`.

Removed options for logging

The options `--log.content-filter` and `--log.source-filter` have been removed. They have most been used during ArangoDB's internal development.

The syslog-related options `--log.application` and `--log.facility` have been removed. They are superseded by the more general `--log.output` option which can also handle syslog targets.

Removed other options

The option `--server.default-api-compatibility` was present in earlier version of ArangoDB to control various aspects of the server behavior, e.g. HTTP return codes or the format of HTTP "location" headers. Client applications could send an HTTP header "x-arango-version" with a version number to request the server behavior of a certain ArangoDB version.

This option was only honored in a handful of cases (described above) and was removed in 3.0 because the changes in server behavior controlled by this option were changed even before ArangoDB 2.0. This should have left enough time for client applications to adapt to the new behavior, making the option superfluous in 3.0.

Thread options

The options `--server.threads` and `--scheduler.threads` now have a default value of `0`. When `--server.threads` is set to `0` on startup, the suitable number of threads will be determined by ArangoDB by asking the OS for the number of available CPUs and using that as a baseline. If the number of CPUs is lower than 4, ArangoDB will still start 4 dispatcher threads. When `--scheduler.threads` is set to `0`, then ArangoDB will automatically determine the number of scheduler threads to start. This will normally create 2 scheduler threads.

If the exact number of threads needs to be set by the admin, then it is still possible to set `--server.threads` and `--scheduler.threads` to non-zero values. ArangoDB will use these values and start that many threads (note that some threads may be created lazily so they may not be present directly after startup).

The number of V8 JavaScript contexts to be created (`--javascript.v8-contexts`) now has a default value of `0` too, meaning that ArangoDB will create as many V8 contexts as there will be dispatcher threads (controlled by the `--server.threads` option). Setting this option to a non-zero value will create exactly as many V8 contexts as specified.

Setting these options explicitly to non-zero values may be beneficial in environments that have few resources (processing time, maximum thread count, available memory).

Authentication

The default value for `--server.authentication` is now `true` in the configuration files shipped with ArangoDB. This means the server will be started with authentication enabled by default, requiring all client connections to provide authentication data when connecting to ArangoDB APIs. Previous ArangoDB versions used the setting `--server.disable-authentication true`, effectively disabling authentication by default.

The default value for `--server.authentication-system-only` is now `true` in ArangoDB. That means that Foxx applications running in ArangoDB will be public accessible (at least they will not use ArangoDB's builtin authentication mechanism). Only requests to ArangoDB APIs at URL path prefixes `/_api/` and `/_admin` will require authentication. To change that, and use the builtin authentication mechanism for Foxx applications too, set `--server.authentication-system-only` to `false`, and make sure to have the option `--server.authentication` set to `true` as well.

Though enabling the authentication is recommended for production setups, it may be overkill in a development environment. To turn off authentication, the option `--server.authentication` can be set to `false` in ArangoDB's configuration file or on the command-line.

Web Admin Interface

The JavaScript shell has been removed from ArangoDB's web interface. The functionality the shell provided is still fully available in the ArangoShell (arangosh) binary shipped with ArangoDB.

ArangoShell and client tools

The ArangoShell (arangosh) and the other client tools bundled with ArangoDB can only connect to an ArangoDB server of version 3.0 or higher. They will not connect to an ArangoDB 2.8. This is because the server HTTP APIs have changed between 2.8 and 3.0, and all client tools uses these APIs.

In order to connect to earlier versions of ArangoDB with the client tools, an older version of the client tools needs to be kept installed.

The preferred name for the template string generator function `aqlQuery` is now `aql` and is automatically available in arangosh. Elsewhere, it can be loaded like `const aql = require('@arangodb').aql`.

Command-line options added

All client tools in 3.0 provide an option `--server.max-packet-size` for controlling the maximum size of HTTP packets to be handled by the client tools. The default value is 128 MB, as in previous versions of ArangoDB. In contrast to previous versions in which the value was hard-coded, the option is now configurable. It can be increased to make the client tools handle very large HTTP result messages sent by the server.

Command-line options changed

For all client tools, the option `--server.disable-authentication` was renamed to `--server.authentication`. Note that the meaning of the option `--server.authentication` is the opposite of the previous `--server.disable-authentication`.

The option `--server.ssl-protocol` was renamed to `--ssl.protocol`. The meaning of the option is unchanged.

The command-line option `--quiet` was removed from all client tools except arangosh because it had no effect in them.

Arangobench

In order to make its purpose more apparent the former `arangob` client tool has been renamed to `arangobench` in 3.0.

Miscellaneous changes

The checksum calculation algorithm for the `collection.checksum()` method and its corresponding REST API GET `/_api/collection/<collection</checksum` has changed in 3.0. Checksums calculated in 3.0 will differ from checksums calculated with 2.8 or before.

The ArangoDB server in 3.0 does not read a file `ENDPOINTS` containing a list of additional endpoints on startup. In 2.8 this file was automatically read if present in the database directory.

The names of the sub-threads started by ArangoDB have changed in 3.0. This is relevant on Linux only, where threads can be named and thread names may be visible to system tools such as *top* or monitoring solutions.

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.8. ArangoDB 2.8 also contains several bugfixes that are not listed here. For a list of bugfixes, please consult the CHANGELOG.

AQL improvements

AQL Graph Traversals / Pattern Matching

AQL offers a new feature to traverse over a graph without writing JavaScript functions but with all the other features you know from AQL. For this purpose, a special version of `FOR variableName IN expression` has been introduced.

This special version has the following format: `FOR vertex-variable, edge-variable, path-variable IN traversal-expression` , where `traversal-expression` has the following format: `[depth] direction start-vertex graph-definition` with the following input parameters:

- depth (optional): defines how many steps are executed. The value can either be an integer value (e.g. `3`) or a range of integer values (e.g. `1..5`). The default is 1.
- direction: defines which edge directions are followed. Can be either `OUTBOUND` , `INBOUND` or `ANY` .
- start-vertex: defines where the traversal is started. Must be an `_id` value or a document.
- graph-definition: defines which edge collections are used for the traversal. Must be either `GRAPH graph-name` for graphs created with the graph-module, or a list of edge collections `edge-col1, edge-col2, .. edge-colN` .

The three output variables have the following semantics:

- vertex-variable: The last visited vertex.
- edge-variable: The last visited edge (optional).
- path-variable: The complete path from start-vertex to vertex-variable (optional).

The traversal statement can be used in the same way as the original `FOR variableName IN expression` , and can be combined with filters and other AQL constructs.

As an example one can now find the friends of a friend for a certain user with this AQL statement:

```
FOR foaf, e, path IN 2 ANY @startUser GRAPH "relations"
  FILTER path.edges[0].type == "friend"
  FILTER path.edges[1].type == "friend"
  FILTER foaf._id != @startUser
  RETURN DISTINCT foaf
```

Optimizer rules have been implemented to gain performance of the traversal statement. These rules move filter statements into the traversal statement s.t. paths which can never pass the filter are not emitted to the variables.

As an example take the query above and assume there are edges that do not have `type == "friend"` . If in the first edge step there is such a non-friend edge the second steps will never be computed for these edges as they cannot fulfill the filter condition.

Array Indexes

Hash indexes and skiplist indexes can now optionally be defined for array values so that they index individual array members instead of the entire array value.

To define an index for array values, the attribute name is extended with the expansion operator `[*]` in the index definition.

Example:

```
db._create("posts");
db.posts.ensureHashIndex("tags[*]");
```

When given the following document

```
{
  "tags": [
    "AQL",
    "ArangoDB",
    "Index"
  ]
}
```

this index will now contain the individual values `"AQL"` , `"ArangoDB"` and `"Index"` .

Now the index can be used for finding all documents having `"ArangoDB"` somewhere in their `tags` array using the following AQL query:

```
FOR doc IN posts
  FILTER "ArangoDB" IN doc.tags[*]
  RETURN doc
```

It is also possible to create an index on sub-attributes of array values. This makes sense when the index attribute is an array of objects, e.g.

```
db._drop("posts");
db._create("posts");
db.posts.ensureIndex({ type: "hash", fields: [ "tags[*].name" ] });
db.posts.insert({ tags: [ { name: "AQL" }, { name: "ArangoDB" }, { name: "Index" } ] });
db.posts.insert({ tags: [ { name: "AQL" }, { name: "2.8" } ] });
```

The following query will then use the array index:

```
FOR doc IN posts
  FILTER 'AQL' IN doc.tags[*].name
  RETURN doc
```

Array values will automatically be de-duplicated before being inserted into an array index.

Please note that filtering using array indexes only works from within AQL queries and only if the query filters on the indexed attribute using the `IN` operator. The other comparison operators (`==` , `!=` , `>` , `>=` , `<` , `<=`) currently do not use array indexes.

Optimizer improvements

The AQL query optimizer can now use indexes if multiple filter conditions on attributes of the same collection are combined with logical ORs, and if the usage of indexes would completely cover these conditions.

For example, the following queries can now use two independent indexes on `value1` and `value2` (the latter query requires that the indexes are skiplist indexes due to usage of the `<` and `>` comparison operators):

```
FOR doc IN collection FILTER doc.value1 == 42 || doc.value2 == 23 RETURN doc
FOR doc IN collection FILTER doc.value1 < 42 || doc.value2 > 23 RETURN doc
```

The new optimizer rule "sort-in-values" can now pre-sort the right-hand side operand of `IN` and `NOT IN` operators so the operation can use a binary search with logarithmic complexity instead of a linear search. The rule will be applied when the right-hand side operand of an `IN` or `NOT IN` operator in a filter condition is a variable that is defined in a different loop/scope than the operator itself. Additionally, the filter condition must consist of solely the `IN` or `NOT IN` operation in order to avoid any side-effects.

The rule will kick in for a queries such as the following:

```
LET values = /* some runtime expression here */
FOR doc IN collection
  FILTER doc.value IN values
  RETURN doc
```

It will not be applied for the followig queries, because the right-hand side operand of the `IN` is either not a variable, or because the FILTER condition may have side effects:

```
FOR doc IN collection
  FILTER doc.value IN /* some runtime expression here */
  RETURN doc
```

```
LET values = /* some runtime expression here */
FOR doc IN collection
  FILTER FUNCTION(doc.values) == 23 && doc.value IN values
  RETURN doc
```

AQL functions added

The following AQL functions have been added in 2.8:

- `POW(base, exponent)` : returns the *base* to the exponent *exp*
- `UNSET_RECURSIVE(document, attributename, ...)` : recursively removes the attributes *attributename* (can be one or many) from *document* and its sub-documents. All other attributes will be preserved. Multiple attribute names can be specified by either passing multiple individual string argument names, or by passing an array of attribute names:

  ```
  UNSET_RECURSIVE(doc, '_id', '_key', 'foo', 'bar')
  UNSET_RECURSIVE(doc, [ '_id', '_key', 'foo', 'bar' ])
  ```

- `IS_DATESTRING(value)` : returns true if *value* is a string that can be used in a date function. This includes partial dates such as *2015* or *2015-10* and strings containing invalid dates such as *2015-02-31*. The function will return false for all non-string values, even if some of them may be usable in date functions.

Miscellaneous improvements

- the ArangoShell now provides the convenience function `db._explain(query)` for retrieving a human-readable explanation of AQL queries. This function is a shorthand for `require("org/arangodb/aql/explainer").explain(query)` .
- the AQL query optimizer now automatically converts `LENGTH(collection-name)` to an optimized expression that returns the number of documents in a collection. Previous versions of ArangoDB returned a warning when using this expression and also enumerated all documents in the collection, which was inefficient.
- improved performance of skipping over many documents in an AQL query when no indexes and no filters are used, e.g.

  ```
  FOR doc IN collection
    LIMIT 1000000, 10
    RETURN doc
  ```

- added cluster execution site info in execution plan explain output for AQL queries
- for 30+ AQL functions there is now an additional implementation in C++ that removes the need for internal data conversion when the function is called
- the AQL editor in the web interface now supports using bind parameters

Deadlock detection

ArangoDB 2.8 now has an automatic deadlock detection for transactions.

A deadlock is a situation in which two or more concurrent operations (user transactions or AQL queries) try to access the same resources (collections, documents) and need to wait for the others to finish, but none of them can make any progress.

In case of such a deadlock, there would be no progress for any of the involved transactions, and none of the involved transactions could ever complete. This is completely undesirable, so the new automatic deadlock detection mechanism in ArangoDB will automatically kick in and abort one of the transactions involved in such a deadlock. Aborting means that all changes done by the transaction will be rolled back and error 29 (`deadlock detected`) will be thrown.

Client code (AQL queries, user transactions) that accesses more than one collection should be aware of the potential of deadlocks and should handle the error 29 (`deadlock detected`) properly, either by passing the exception to the caller or retrying the operation.

Replication

The following improvements for replication have been made in 2.8 (note: most of them have been backported to ArangoDB 2.7 as well):

- added `autoResync` configuration parameter for continuous replication.

  When set to `true`, a replication slave will automatically trigger a full data re-synchronization with the master when the master cannot provide the log data the slave had asked for. Note that `autoResync` will only work when the option `requireFromPresent` is also set to `true` for the continuous replication, or when the continuous syncer is started and detects that no start tick is present.

  Automatic re-synchronization may transfer a lot of data from the master to the slave and may be expensive. It is therefore turned off by default. When turned off, the slave will never perform an automatic re-synchronization with the master.

- added `idleMinWaitTime` and `idleMaxWaitTime` configuration parameters for continuous replication.

  These parameters can be used to control the minimum and maximum wait time the slave will (intentionally) idle and not poll for master log changes in case the master had sent the full logs already. The `idleMaxWaitTime` value will only be used when `adapativePolling` is set to `true`. When `adaptivePolling` is disabled, only `idleMinWaitTime` will be used as a constant time span in which the slave will not poll the master for further changes. The default values are 0.5 seconds for `idleMinWaitTime` and 2.5 seconds for `idleMaxWaitTime`, which correspond to the hard-coded values used in previous versions of ArangoDB.

- added `initialSyncMaxWaitTime` configuration parameter for initial and continuous replication

  This option controls the maximum wait time (in seconds) that the initial synchronization will wait for a response from the master when fetching initial collection data. If no response is received within this time period, the initial synchronization will give up and fail. This option is also relevant for continuous replication in case *autoResync* is set to *true*, as then the continuous replication may trigger a full data re-synchronization in case the master cannot the log data the slave had asked for.

- HTTP requests sent from the slave to the master during initial synchronization will now be retried if they fail with connection problems.

- the initial synchronization now logs its progress so it can be queried using the regular replication status check APIs.

- added `async` attribute for `sync` and `syncCollection` operations called from the ArangoShell. Setthing this attribute to `true` will make the synchronization job on the server go into the background, so that the shell does not block. The status of the started asynchronous synchronization job can be queried from the ArangoShell like this:

  ```
  /* starts initial synchronization */
  var replication = require("org/arangodb/replication");
  var id = replication.sync({
    endpoint: "tcp://master.domain.org:8529",
    username: "myuser",
    password: "mypasswd",
    async: true
  });

  /* now query the id of the returned async job and print the status */
  print(replication.getSyncResult(id));
  ```

  The result of `getSyncResult()` will be `false` while the server-side job has not completed, and different to `false` if it has completed. When it has completed, all job result details will be returned by the call to `getSyncResult()`.

- the web admin interface dashboard now shows a server's replication status at the bottom of the page

Web Admin Interface

The following improvements have been made for the web admin interface:

- the AQL editor now has support for bind parameters. The bind parameter values can be edited in the web interface and saved with a query for future use.

- the AQL editor now allows canceling running queries. This can be used to cancel long-running queries without switching to the *query management* section.
- the dashboard now provides information about the server's replication status at the bottom of the page. This can be used to track either the status of a one-time synchronization or the continuous replication.
- the compaction status and some status internals about collections are now displayed in the detail view for a collection in the web interface. These data can be used for debugging compaction issues.
- unloading a collection via the web interface will now trigger garbage collection in all v8 contexts and force a WAL flush. This increases the chances of perfoming the unload faster.
- the status terminology for collections for which an unload request has been issued via the web interface was changed from `in the process of being unloaded` to `will be unloaded`. This is more accurate as the actual unload may be postponed until later if there are still references pointing to data in the collection.

Foxx improvements

- the module resolution used by `require` now behaves more like in node.js
- the `org/arangodb/request` module now returns response bodies for error responses by default. The old behavior of not returning bodies for error responses can be re-enabled by explicitly setting the option `returnBodyOnError` to `false`

Miscellaneous changes

The startup option `--server.hide-product-header` can be used to make the server not send the HTTP response header `"Server: ArangoDB"` in its HTTP responses. This can be used to conceal the server make from HTTP clients. By default, the option is turned off so the header is still sent as usual.

arangodump and arangorestore now have better error reporting. Additionally, arangodump will now fail by default when trying to dump edges that refer to already dropped collections. This can be circumvented by specifying the option `--force true` when invoking arangodump.

arangoimp now provides an option `--create-collection-type` to specify the type of the collection to be created when `--create-collection` is set to `true`. Previously `--create-collection` always created document collections and the creation of edge collections was not possible.

Incompatible changes in ArangoDB 2.8

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 2.8, and adjust any client programs if necessary.

AQL

Keywords added

The following AQL keywords were added in ArangoDB 2.8:

- `GRAPH`
- `OUTBOUND`
- `INBOUND`
- `ANY`
- `ALL`
- `NONE`
- `AGGREGATE`

Usage of these keywords for collection names, variable names or attribute names in AQL queries will not be possible without quoting. For example, the following AQL query will still work as it uses a quoted collection name and a quoted attribute name:

```
FOR doc IN `OUTBOUND`
  RETURN doc.`any`
```

Changed behavior

The AQL functions `NEAR` and `WITHIN` now have stricter validations for their input parameters `limit` , `radius` and `distance` . They may now throw exceptions when invalid parameters are passed that may have not led to exceptions in previous versions.

Additionally, the expansion (`[*]`) operator in AQL has changed its behavior when handling non-array values:

In ArangoDB 2.8, calling the expansion operator on a non-array value will always return an empty array. Previous versions of ArangoDB expanded non-array values by calling the `TO_ARRAY()` function for the value, which for example returned an array with a single value for boolean, numeric and string input values, and an array with the object's values for an object input value. This behavior was inconsistent with how the expansion operator works for the array indexes in 2.8, so the behavior is now unified:

- if the left-hand side operand of `[*]` is an array, the array will be returned as is when calling `[*]` on it
- if the left-hand side operand of `[*]` is not an array, an empty array will be returned by `[*]`

AQL queries that rely on the old behavior can be changed by either calling `TO_ARRAY` explicitly or by using the `[*]` at the correct position.

The following example query will change its result in 2.8 compared to 2.7:

```
LET values = "foo" RETURN values[*]
```

In 2.7 the query has returned the array `[ "foo" ]`, but in 2.8 it will return an empty array `[ ]`. To make it return the array `[ "foo" ]` again, an explicit `TO_ARRAY` function call is needed in 2.8 (which in this case allows the removal of the `[*]` operator altogether). This also works in 2.7:

```
LET values = "foo" RETURN TO_ARRAY(values)
```

Another example:

```
LET values = [ { name: "foo" }, { name: "bar" } ]
RETURN values[*].name[*]
```

The above returned `[ [ "foo" ], [ "bar" ] ] in 2.7. In 2.8 it will return` [[ ], [ ]] `, because the value of` name` is not an array. To change the results to the 2.7 style, the query can be changed to

```
LET values = [ { name: "foo" }, { name: "bar" } ]
RETURN values[* RETURN TO_ARRAY(CURRENT.name)]
```

The above also works in 2.7. The following types of queries won't change:

```
LET values = [ 1, 2, 3 ] RETURN values[*]
LET values = [ { name: "foo" }, { name: "bar" } ] RETURN values[*].name
LET values = [ { names: [ "foo", "bar" ] }, { names: [ "baz" ] } ] RETURN values[*].names[*]
LET values = [ { names: [ "foo", "bar" ] }, { names: [ "baz" ] } ] RETURN values[*].names[**]
```

Deadlock handling

Client applications should be prepared to handle error 29 (`deadlock detected`) that ArangoDB may now throw when it detects a deadlock across multiple transactions. When a client application receives error 29, it should retry the operation that failed.

The error can only occur for AQL queries or user transactions that involve more than a single collection.

Optimizer

The AQL execution node type `IndexRangeNode` was replaced with a new more capable execution node type `IndexNode` . That means in execution plan explain output there will be no more `IndexRangeNode` s but only `IndexNode` . This affects explain output that can be retrieved via `require("org/arangodb/aql/explainer").explain(query)` , `db._explain(query)` , and the HTTP query explain API.

The optimizer rule that makes AQL queries actually use indexes was also renamed from `use-index-range` to `use-indexes` . Again this affects explain output that can be retrieved via `require("org/arangodb/aql/explainer").explain(query)` , `db._explain(query)` , and the HTTP query explain API.

The query optimizer rule `remove-collect-into` was renamed to `remove-collect-variables` . This affects explain output that can be retrieved via `require("org/arangodb/aql/explainer").explain(query)` , `db._explain(query)` , and the HTTP query explain API.

HTTP API

When a server-side operation got canceled due to an explicit client cancel request via HTTP `DELETE /_api/job` , previous versions of ArangoDB returned an HTTP status code of 408 (request timeout) for the response of the canceled operation.

The HTTP return code 408 has caused problems with some client applications. Some browsers (e.g. Chrome) handled a 408 response by resending the original request, which is the opposite of what is desired when a job should be canceled.

Therefore ArangoDB will return HTTP status code 410 (gone) for canceled operations from version 2.8 on.

Foxx

Model and Repository

Due to compatibility issues the Model and Repository types are no longer implemented as ES2015 classes.

The pre-2.7 "extend" style subclassing is supported again and will not emit any deprecation warnings.

```
var Foxx = require('org/arangodb/foxx');
var MyModel = Foxx.Model.extend({
  // ...
  schema: {/* ... */}
});
```

Module resolution

The behavior of the JavaScript module resolution used by the `require` function has been modified to improve compatibility with modules written for Node.js.

Specifically

- absolute paths (e.g. `/some/absolute/path`) are now always interpreted as absolute file system paths, relative to the file system root
- global names (e.g. `global/name`) are now first intepreted as references to modules residing in a relevant `node_modules` folder, a built-in module or a matching document in the internal `_modules` collection, and only resolved to local file paths if no other match is found

Previously the two formats were treated interchangeably and would be resolved to local file paths first, leading to problems when local files used the same names as other modules (e.g. a local file `chai.js` would cause problems when trying to load the `chai` module installed in `node_modules`).

For more information see the blog announcement of this change and the upgrade guide.

Module `org/arangodb/request`

The module now always returns response bodies, even for error responses. In versions prior to 2.8 the module would silently drop response bodies if the response header indicated an error.

The old behavior of not returning bodies for error responses can be restored by explicitly setting the option `returnBodyOnError` to `false` :

```
let response = request({
  //...
  returnBodyOnError: false
});
```

Garbage collection

The V8 garbage collection strategy was slightly adjusted so that it eventually happens in all V8 contexts that hold V8 external objects (references to ArangoDB documents and collections). This enables a better cleanup of these resources and prevents other processes such as compaction being stalled while waiting for these resources to be released.

In this context the default value for the JavaScript garbage collection frequency (`--javascript.gc-frequency`) was also increased from 10 seconds to 15 seconds, as less internal operations in ArangoDB are carried out in JavaScript.

Client tools

arangodump will now fail by default when trying to dump edges that refer to already dropped collections. This can be circumvented by specifying the option `--force true` when invoking arangodump

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.7. ArangoDB 2.7 also contains several bugfixes that are not listed here. For a list of bugfixes, please consult the CHANGELOG.

Performance improvements

Index buckets

The primary indexes and hash indexes of collections can now be split into multiple index buckets. This option was available for edge indexes only in ArangoDB 2.6.

A bucket can be considered a container for a specific range of index values. For primary, hash and edge indexes, determining the responsible bucket for an index value is done by hashing the actual index value and applying a simple arithmetic operation on the hash.

Because an index value will be present in at most one bucket and buckets are independent, using multiple buckets provides the following benefits:

- initially building the in-memory index data can be parallelized even for a single index, with one thread per bucket (or with threads being responsible for more than one bucket at a time). This can help reducing the loading time for collections.
- resizing an index when it is about to run out of reserve space is performed per bucket. As each bucket only contains a fraction of the entire index, resizing and rehashing a bucket is much faster and less intrusive than resizing and rehashing the entire index.

When creating new collections, the default number of index buckets is `8` since ArangoDB 2.7. In previous versions, the default value was `1`. The number of buckets can also be adjusted for existing collections so they can benefit from the optimizations. The number of index buckets can be set for a collection at any time by using a collection's `properties` function:

```
db.collection.properties({ indexBuckets: 16 });
```

The number of index buckets must be a power of 2.

Please note that for building the index data for multiple buckets in parallel it is required that a collection contains a significant amount of documents because for a low number of documents the overhead of parallelization will outweigh its benefits. The current threshold value is 256k documents, but this value may change in future versions of ArangoDB. Additionally, the configuration option `--database.index-threads` will determine how many parallel threads may be used for building the index data.

Faster update and remove operations in non-unique hash indexes

The unique hash indexes in ArangoDB provided an amortized O(1) lookup, insert, update and remove performance. Non-unique hash indexes provided amortized O(1) insert performance, but had worse performance for update and remove operations for non-unique values. For documents with the same index value, they maintained a list of collisions. When a document was updated or removed, that exact document had to be found in the collisions list for the index value. While getting to the start of the collisions list was O(1), scanning the list had O(n) performance in the worst case (with n being the number of documents with the same index value). Overall, this made update and remove operations in non-unique hash indexes slow if the index contained many duplicate values.

This has been changed in ArangoDB 2.7 so that non-unique hash indexes now also provide update and remove operations with an amortized complexity of O(1), even if there are many duplicates.

Resizing non-unique hash indexes now also doesn't require looking into the document data (which may involve a disk access) because the index maintains some internal cache value per document. When resizing and rehashing the index (or an index bucket), the index will first compare only the cache values before peeking into the actual documents. This change can also lead to reduced index resizing times.

Throughput enhancements

The ArangoDB-internal implementations for dispatching requests, keeping statistics and assigning V8 contexts to threads have been improved in order to use less locks. These changes allow higher concurrency and throughput in these components, which can also make the server handle more requests in a given period of time.

What gains can be expected depends on which operations are executed, but there are real-world cases in which throughput increased by between 25 % and 70 % when compared to 2.6.

Madvise hints

The Linux variant for ArangoDB provides the OS with madvise hints about index memory and datafile memory. These hints can speed up things when memory is tight, in particular at collection load time but also for random accesses later. There is no formal guarantee that the OS actually uses the madvise hints provided by ArangoDB, but actual measurements have shown improvements for loading bigger collections.

AQL improvements

Additional date functions

ArangoDB 2.7 provides several extra AQL functions for date and time calculation and manipulation. These functions were contributed by GitHub users @CoDEmanX and @friday. A big thanks for their work!

The following extra date functions are available from 2.7 on:

- `DATE_DAYOFYEAR(date)` : Returns the day of year number of *date*. The return values range from 1 to 365, or 366 in a leap year respectively.
- `DATE_ISOWEEK(date)` : Returns the ISO week date of *date*. The return values range from 1 to 53. Monday is considered the first day of the week. There are no fractional weeks, thus the last days in December may belong to the first week of the next year, and the first days in January may be part of the previous year's last week.
- `DATE_LEAPYEAR(date)` : Returns whether the year of *date* is a leap year.
- `DATE_QUARTER(date)` : Returns the quarter of the given date (1-based):
  - 1: January, February, March
  - 2: April, May, June
  - 3: July, August, September
  - 4: October, November, December
- *DATE_DAYS_IN_MONTH(date)*: Returns the number of days in *date*'s month (28..31).
- `DATE_ADD(date, amount, unit)` : Adds *amount* given in *unit* to *date* and returns the calculated date.

  unit can be either of the following to specify the time unit to add or subtract (case-insensitive):

  - y, year, years
  - m, month, months
  - w, week, weeks
  - d, day, days
  - h, hour, hours
  - i, minute, minutes
  - s, second, seconds
  - f, millisecond, milliseconds

  amount is the number of *unit*s to add (positive value) or subtract (negative value).
- `DATE_SUBTRACT(date, amount, unit)` : Subtracts *amount* given in *unit* from *date* and returns the calculated date.

  It works the same as `DATE_ADD()` , except that it subtracts. It is equivalent to calling `DATE_ADD()` with a negative amount, except that `DATE_SUBTRACT()` can also subtract ISO durations. Note that negative ISO durations are not supported (i.e. starting with `-P` , like `-P1Y`).
- `DATE_DIFF(date1, date2, unit, asFloat)` : Calculate the difference between two dates in given time *unit*, optionally with decimal places. Returns a negative value if *date1* is greater than *date2*.
- `DATE_COMPARE(date1, date2, unitRangeStart, unitRangeEnd)` : Compare two partial dates and return true if they match, false otherwise. The parts to compare are defined by a range of time units.

The full range is: years, months, days, hours, minutes, seconds, milliseconds. Pass the unit to start from as *unitRangeStart*, and the unit to end with as *unitRangeEnd*. All units in between will be compared. Leave out *unitRangeEnd* to only compare *unitRangeStart*.

- `DATE_FORMAT(date, format)` : Format a date according to the given format string. It supports the following placeholders (case-insensitive):
  - %t: timestamp, in milliseconds since midnight 1970-01-01
  - %z: ISO date (0000-00-00T00:00:00.000Z)
  - %w: day of week (0..6)
  - %y: year (0..9999)
  - %yy: year (00..99), abbreviated (last two digits)
  - %yyyy: year (0000..9999), padded to length of 4
  - %yyyyyy: year (-009999 .. +009999), with sign prefix and padded to length of 6
  - %m: month (1..12)
  - %mm: month (01..12), padded to length of 2
  - %d: day (1..31)
  - %dd: day (01..31), padded to length of 2
  - %h: hour (0..23)
  - %hh: hour (00..23), padded to length of 2
  - %i: minute (0..59)
  - %ii: minute (00..59), padded to length of 2
  - %s: second (0..59)
  - %ss: second (00..59), padded to length of 2
  - %f: millisecond (0..999)
  - %fff: millisecond (000..999), padded to length of 3
  - %x: day of year (1..366)
  - %xxx: day of year (001..366), padded to length of 3
  - %k: ISO week date (1..53)
  - %kk: ISO week date (01..53), padded to length of 2
  - %l: leap year (0 or 1)
  - %q: quarter (1..4)
  - %a: days in month (28..31)
  - %mmm: abbreviated English name of month (Jan..Dec)
  - %mmmm: English name of month (January..December)
  - %www: abbreviated English name of weekday (Sun..Sat)
  - %wwww: English name of weekday (Sunday..Saturday)
  - %&: special escape sequence for rare occasions
  - %%: literal %
  - %: ignored

RETURN DISTINCT

To return unique values from a query, AQL now provides the `DISTINCT` keyword. It can be used as a modifier for `RETURN` statements, as a shorter alternative to the already existing `COLLECT` statement.

For example, the following query only returns distinct (unique) `status` attribute values from the collection:

```
FOR doc IN collection
  RETURN DISTINCT doc.status
```

`RETURN DISTINCT` is not allowed on the top-level of a query if there is no `FOR` loop in front of it. `RETURN DISTINCT` is allowed in subqueries.

`RETURN DISTINCT` ensures that the values returned are distinct (unique), but does not guarantee any order of results. In order to have certain result order, an additional `SORT` statement must be added to a query.

Shorthand object notation

AQL now provides a shorthand notation for object literals in the style of ES6 object literals:

```
LET name = "Peter"
LET age = 42
RETURN { name, age }
```

This is equivalent to the previously available canonical form, which is still available and supported:

```
LET name = "Peter"
LET age = 42
RETURN { name : name, age : age }
```

Array expansion improvements

The already existing *[*]* operator has been improved with optional filtering and projection and limit capabilities.

For example, consider the following example query that filters values from an array attribute:

```
FOR u IN users
  RETURN {
    name: u.name,
    friends: (
      FOR f IN u.friends
        FILTER f.age > u.age
        RETURN f.name
    )
  }
```

With the *[*]* operator, this query can be simplified to

```
FOR u IN users
  RETURN { name: u.name, friends: u.friends[* FILTER CURRENT.age > u.age].name }
```

The pseudo-variable *CURRENT* can be used to access the current array element. The `FILTER` condition can refer to `CURRENT` or any variables valid in the outer scope.

To return a projection of the current element, there can now be an inline `RETURN` :

```
FOR u IN users
  RETURN u.friends[* RETURN CONCAT(CURRENT.name, " is a friend of ", u.name)]
```

which is the simplified variant for:

```
FOR u IN users
  RETURN (
    FOR friend IN u.friends
      RETURN CONCAT(friend.name, " is a friend of ", u.name)
  )
```

Array contraction

In order to collapse (or flatten) results in nested arrays, AQL now provides the *[**]* operator. It works similar to the *[*]* operator, but additionally collapses nested arrays. How many levels are collapsed is determined by the amount of * characters used.

For example, consider the following query that produces a nested result:

```
FOR u IN users
  RETURN u.friends[*].name
```

The *[**]* operator can now be applied to get rid of the nested array and turn it into a flat array. We simply apply the *[**]* on the previous query result:

```
RETURN (
  FOR u IN users RETURN u.friends[*].name
)[**]
```

Template query strings

Assembling query strings in JavaScript has been error-prone when using simple string concatenation, especially because plain JavaScript strings do not have multiline-support, and because of potential parameter injection issues. While multiline query strings can be assembled with ES6 template strings since ArangoDB 2.5, and query bind parameters are there since ArangoDB 1.0 to prevent parameter injection, there was no JavaScript-y solution to combine these.

ArangoDB 2.7 now provides an ES6 template string generator function that can be used to easily and safely assemble AQL queries from JavaScript. JavaScript variables and expressions can be used easily using regular ES6 template string substitutions:

```
let name = 'test';
let attributeName = '_key';

let query = aqlQuery`FOR u IN users
  FILTER u.name == ${name}
  RETURN u.${attributeName}`;
db._query(query);
```

This is more legible than when using a plain JavaScript string and also does not require defining the bind parameter values separately:

```
let name = 'test';
let attributeName = '_key';

let query = "FOR u IN users " +
  "FILTER u.name == @name " +
  "RETURN u.@attributeName";
db._query(query, {
  name,
  attributeName
});
```

The `aqlQuery` template string generator will also handle collection objects automatically:

```
db._query(aqlQuery`FOR u IN ${ db.users } RETURN u.name`);
```

Note that while template strings are available in the JavaScript functions provided to build queries, they aren't a feature of AQL itself. AQL could always handle multiline query strings and provided bind parameters (`@...`) for separating the query string and the parameter values. The `aqlQuery` template string generator function will take care of this separation, too, but will do it *behind the scenes*.

AQL query result cache

The AQL query result cache can optionally cache the complete results of all or just selected AQL queries. It can be operated in the following modes:

- `off` : the cache is disabled. No query results will be stored
- `on` : the cache will store the results of all AQL queries unless their `cache` attribute flag is set to `false`
- `demand` : the cache will store the results of AQL queries that have their `cache` attribute set to `true` , but will ignore all others

The mode can be set at server startup using the `--database.query-cache-mode` configuration option and later changed at runtime. The default value is `off` , meaning that the query result cache is disabled. This is because the cache may consume additional memory to keep query results, and also because it must be invalidated when changes happen in collections for which results have been cached.

The query result cache may therefore have positive or negative effects on query execution times, depending on the workload: it will not make much sense turning on the cache in write-only or write-mostly scenarios, but the cache may be very beneficial in case workloads are read-only or read-mostly, and query are complex.

If the query cache is operated in `demand` mode, it can be controlled per query if the cache should be checked for a result.

Miscellaneous changes

Optimizer

The AQL optimizer rule `patch-update-statements` has been added. This rule can optimize certain AQL UPDATE queries that update documents in the a collection that they also iterate over.

For example, the following query reads documents from a collection in order to update them:

```
    FOR doc IN collection
      UPDATE doc WITH { newValue: doc.oldValue + 1 } IN collection
```

In this case, only a single collection is affected by the query, and there is no index lookup involved to find the to-be-updated documents. In this case, the UPDATE query does not require taking a full, memory-intensive snapshot of the collection, but it can be performed in small chunks. This can lead to memory savings when executing such queries.

Function call arguments optimization

This optimization will lead to arguments in function calls inside AQL queries not being copied but being passed by reference. This may speed up calls to functions with bigger argument values or queries that call AQL functions a lot of times.

Web Admin Interface

The web interface now has a new design.

The "Applications" tab in the web interfaces has been renamed to "Services".

The ArangoDB API documentation has been moved from the "Tools" menu to the "Links" menu. The new documentation is based on Swagger 2.0 and opens in a separate web page.

Foxx improvements

ES2015 Classes

All Foxx constructors have been replaced with ES2015 classes and can be extended using the class syntax. The `extend` method is still supported at the moment but will become deprecated in ArangoDB 2.8 and removed in ArangoDB 2.9.

Before:

```
var Foxx = require('org/arangodb/foxx');
var MyModel = Foxx.Model.extend({
  // ...
  schema: {/* ... */}
});
```

After:

```
var Foxx = require('org/arangodb/foxx');
class MyModel extends Foxx.Model {
  // ...
}
MyModel.prototype.schema = {/* ... */};
```

Confidential configuration

It is now possible to specify configuration options with the type `password`. The password type is equivalent to the text type but will be masked in the web frontend to prevent accidental exposure of confidential options like API keys and passwords when configuring your Foxx application.

Dependencies

The syntax for specifying dependencies in manifests has been extended to allow specifying optional dependencies. Unmet optional dependencies will not prevent an app from being mounted. The traditional shorthand syntax for specifying non-optional dependencies will still be supported in the upcoming versions of ArangoDB.

Before:

```
{
  ...
  "dependencies": {
    "notReallyNeeded": "users:^1.0.0",
    "totallyNecessary": "sessions:^1.0.0"
  }
}
```

After:

```
{
  "dependencies": {
    "notReallyNeeded": {
      "name": "users",
      "version": "^1.0.0",
      "required": false
    },
    "totallyNecessary": {
      "name": "sessions",
      "version": "^1.0.0"
    }
  }
}
```

Replication

The existing replication HTTP API has been extended with methods that replication clients can use to determine whether a given date, identified by a tick value, is still present on a master for replication. By calling these APIs, clients can make an informed decision about whether the master can still provide all missing data starting from the point up to which the client had already synchronized. This can be helpful in case a replication client is re-started after a pause.

Master servers now also track up the point up to which they have sent changes to clients for replication. This information can be used to determine the point of data that replication clients have received from the master, and if and how far approximately they lag behind.

Finally, restarting the replication applier on a slave server has been made more robust in case the applier was stopped while there were pending transactions on the master server, and re-starting the replication applier needs to restore the state of these transactions.

Client tools

The filenames in dumps created by arangodump now contain not only the name of the dumped collection, but also an additional 32-digit hash value. This is done to prevent overwriting dump files in case-insensitive file systems when there exist multiple collections with the same name (but with different cases).

For example, if a database had two collections *test* and *Test*, previous versions of arangodump created the following files:

- `test.structure.json` and `test.data.json` for collection *test*
- `Test.structure.json` and `Test.data.json` for collection *Test*

This did not work in case-insensitive filesystems, because the files for the second collection would have overwritten the files of the first. arangodump in 2.7 will create the unique files in this case, by appending the 32-digit hash value to the collection name in all case. These filenames will be unambiguous even in case-insensitive filesystems.

Miscellaneous changes

Better control-C support in arangosh

When CTRL-C is pressed in arangosh, it will now abort the locally running command (if any). If no command was running, pressing CTRL-C will print a `^C` first. Pressing CTRL-C again will then quit arangosh.

CTRL-C can also be used to reset the current prompt while entering complex nested objects which span multiple input lines.

CTRL-C support has been added to the ArangoShell versions built with Readline-support (Linux and MacOS only). The Windows version of ArangoDB uses a different library for handling input, and support for CTRL-C has not been added there yet.

Start / stop

Linux startup scripts and systemd configuration for arangod now try to adjust the NOFILE (number of open files) limits for the process. The limit value is set to 131072 (128k) when ArangoDB is started via start/stop commands.

This will prevent arangod running out of available file descriptors in case of many parallel HTTP connections or large collections with many datafiles.

Additionally, when ArangoDB is started/stopped manually via the start/stop commands, the main process will wait for up to 10 seconds after it forks the supervisor and arangod child processes. If the startup fails within that period, the start/stop script will fail with a non-zero exit code, allowing any invoking scripts to handle this error. Previous versions always returned an exit code of 0, even when arangod couldn't be started.

If the startup of the supervisor or arangod is still ongoing after 10 seconds, the main program will still return with exit code 0 in order to not block any scripts. The limit of 10 seconds is arbitrary because the time required for an arangod startup is not known in advance.

Non-sparse logfiles

WAL logfiles and datafiles created by arangod are now non-sparse. This prevents SIGBUS signals being raised when a memory-mapped region backed by a sparse datafile was accessed and the memory region was not actually backed by disk, for example because the disk ran out of space.

arangod now always fully allocates the disk space required for a logfile or datafile when it creates one, so the memory region can always be backed by disk, and memory can be accessed without SIGBUS being raised.

Incompatible changes in ArangoDB 2.7

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 2.7, and adjust any client programs if necessary.

AQL changes

`DISTINCT` is now a keyword in AQL.

AQL queries that use `DISTINCT` (in lower, upper or mixed case) as an identifier (i.e. as a variable, a collection name or a function name) will stop working. To make such queries working again, each occurrence of `DISTINCT` in an AQL query should be enclosed in backticks. This will turn `DISTINCT` from a keyword into an identifier again.

The AQL function `SKIPLIST()` has been removed in ArangoDB 2.7. This function was deprecated in ArangoDB 2.6. It was a left-over from times when the query optimizer wasn't able to use skiplist indexes together with filters, skip and limit values. Since this issue been fixed since version 2.3, there is no AQL replacement function for `SKIPLIST`. Queries that use the `SKIPLIST` function can be fixed by using the usual combination of `FOR`, `FILTER` and `LIMIT`, e.g.

```
    FOR doc IN @@collection
      FILTER doc.value >= @value
      SORT doc.value DESC
      LIMIT 1
      RETURN doc
```

Foxx changes

Bundling and compilation

The `assets` property is no longer supported in Foxx manifests and is scheduled to be removed in a future version of ArangoDB. The `files` property can still be used to serve static assets but it is recommended to use separate tooling to compile and bundle your assets.

Manifest scripts

The properties `setup` and `teardown` have been moved into the `scripts` property map:

Before:

```
{
  ...
  "setup": "scripts/setup.js",
  "teardown": "scripts/teardown.js"
}
```

After:

```
{
  ...
  "scripts": {
    "setup": "scripts/setup.js",
    "teardown": "scripts/teardown.js"
  }
}
```

Foxx Queues

Function-based Foxx Queue job types are no longer supported. To learn about how you can use the new script-based job types follow the updated recipe in the cookbook.

Foxx Sessions

The `jwt` and `type` options have been removed from the `activateSessions` API.

If you want to replicate the behavior of the `jwt` option you can use the JWT functions in the `crypto` module. A JWT-based session storage that doesn't write sessions to the database is available as the sessions-jwt app in the Foxx app store.

The session type is now inferred from the presence of the `cookie` or `header` options (allowing you to enable support for both). If you want to use the default settings for `cookie` or `header` you can pass the value `true` instead.

The `sessionStorageApp` option has been removed in favour of the `sessionStorage` option.

Before:

```
var Foxx = require('org/arangodb/foxx');
var ctrl = new Foxx.Controller(applicationContext);

ctrl.activateSessions({
  sessionStorageApp: 'some-sessions-app',
  type: 'cookie'
});
```

After:

```
ctrl.activateSessions({
  sessionStorage: applicationContext.dependencies.sessions.sessionStorage,
  cookie: true
});
```

Request module

The module `org/arangodb/request` uses an internal library function for sending HTTP requests. This library functionally unconditionally set an HTTP header `Accept-Encoding: gzip` in all outgoing HTTP requests, without client code having to set this header explicitly.

This has been fixed in 2.7, so `Accept-Encoding: gzip` is not set automatically anymore. Additionally the header `User-Agent: ArangoDB` is not set automatically either. If client applications rely on these headers being sent, they are free to add it when constructing requests using the request module.

The `internal.download()` function is also affected by this change. Again, the header can be added here if required by passing it via a `headers` sub-attribute in the third parameter (`options`) to this function.

arangodump / backups

The filenames in dumps created by arangodump now contain not only the name of the dumped collection, but also an additional 32-digit hash value. This is done to prevent overwriting dump files in case-insensitive file systems when there exist multiple collections with the same name (but with different cases).

This change leads to changed filenames in dumps created by arangodump. If any client scripts depend on the filenames in the dump output directory being equal to the collection name plus one of the suffixes `.structure.json` and `.data.json` , they need to be adjusted.

Starting with ArangoDB 2.7, the file names will contain an underscore plus the 32-digit MD5 value (represented in hexadecimal notation) of the collection name.

For example, when arangodump dumps data of two collections *test* and *Test*, the filenames in previous versions of ArangoDB were:

- `test.structure.json` (definitions for collection *test*)
- `test.data.json` (data for collection *test*)
- `Test.structure.json` (definitions for collection *Test*)
- `Test.data.json` (data for collection *Test*)

In 2.7, the filenames will be:

- `test_098f6bcd4621d373cade4e832627b4f6.structure.json` (definitions for collection *test*)
- `test_098f6bcd4621d373cade4e832627b4f6.data.json` (data for collection *test*)
- `Test_0cbc6611f5540bd0809a388dc95a615b.structure.json` (definitions for collection *Test*)
- `Test_0cbc6611f5540bd0809a388dc95a615b.data.json` (data for collection *Test*)

Starting / stopping

When starting arangod, the server will now drop the process privileges to the specified values in options `--server.uid` and `--server.gid` instantly after parsing the startup options.

That means when either `--server.uid` or `--server.gid` are set, the privilege change will happen earlier. This may prevent binding the server to an endpoint with a port number lower than 1024 if the arangodb user has no privileges for that. Previous versions of ArangoDB changed the privileges later, so some startup actions were still carried out under the invoking user (i.e. likely *root* when started via init.d or system scripts) and especially binding to low port numbers was still possible there.

The default privileges for user *arangodb* will not be sufficient for binding to port numbers lower than 1024. To have an ArangoDB 2.7 bind to a port number lower than 1024, it needs to be started with either a different privileged user, or the privileges of the *arangodb* user have to raised manually beforehand.

Additionally, Linux startup scripts and systemd configuration for arangod now will adjust the NOFILE (number of open files) limits for the process. The limit value is set to 131072 (128k) when ArangoDB is started via start/stop commands. The goal of this change is to prevent arangod from running out of available file descriptors for socket connections and datafiles.

Connection handling

arangod will now actually close lingering client connections when idle for at least the duration specified in the `--server.keep-alive-timeout` startup option.

In previous versions of ArangoDB, idle connections were not closed by the server when the timeout was reached and the client was still connected. Now the connection is properly closed by the server in case of timeout. Client applications relying on the old behavior may now need to reconnect to the server when their idle connections time out and get closed (note: connections being idle for a long time may be closed by the OS or firewalls anyway - client applications should be aware of that and try to reconnect).

Option changes

Configure options removed

The following options for `configure` have been removed because they were unused or exotic:

- `--enable-timings`
- `--enable-figures`

Startup options added

The following configuration options have been added in 2.7:

- `--database.query-cache-max-results` : sets the maximum number of results in AQL query result cache per database
- `--database.query-cache-mode` : sets the mode for the AQL query results cache. Possible values are `on` , `off` and `demand` . The default value is `off`

Miscellaneous changes

Simple queries

Many simple queries provide a `skip()` function that can be used to skip over a certain number of documents in the result. This function allowed specifying negative offsets in previous versions of ArangoDB. Specifying a negative offset led to the query result being iterated in reverse order, so skipping was performed from the back of the result. As most simple queries do not provide a guaranteed result order, skipping from the back of a result with unspecific order seems a rather exotic use case and was removed to increase consistency with AQL, which also does not provide negative skip values.

Negative skip values were deprecated in ArangoDB 2.6.

Tasks API

The undocumented function `addJob()` has been removed from the `org/arangodb/tasks` module in ArangoDB 2.7.

Runtime endpoints manipulation API

The following HTTP REST API methods for runtime manipulation of server endpoints have been removed in ArangoDB 2.7:

- POST `/_api/endpoint`: to dynamically add an endpoint while the server was running
- DELETE `/_api/endpoint`: to dynamically remove an endpoint while the server was running

This change also affects the equivalent JavaScript endpoint manipulation methods available in Foxx. The following functions have been removed in ArangoDB 2.7:

- `db._configureEndpoint()`
- `db._removeEndpoint()`

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.6. ArangoDB 2.6 also contains several bugfixes that are not listed here. For a list of bugfixes, please consult the CHANGELOG.

APIs added

Batch document removal and lookup commands

The following commands have been added for `collection` objects:

- collection.lookupByKeys(keys)
- collection.removeByKeys(keys)

These commands can be used to perform multi-document lookup and removal operations efficiently from the ArangoShell. The argument to these operations is an array of document keys.

These commands can also be used via the HTTP REST API. Their endpoints are:

- PUT /_api/simple/lookup-by-keys
- PUT /_api/simple/remove-by-keys

Collection export HTTP REST API

ArangoDB now provides a dedicated collection export API, which can take snapshots of entire collections more efficiently than the general-purpose cursor API. The export API is useful to transfer the contents of an entire collection to a client application. It provides optional filtering on specific attributes.

The export API is available at endpoint `POST /_api/export?collection=...` . The API has the same return value structure as the already established cursor API (`POST /_api/cursor`).

An introduction to the export API is given in this blog post: http://jsteemann.github.io/blog/2015/04/04/more-efficient-data-exports/

AQL improvements

EDGES AQL Function

The AQL function EDGES got a new fifth optional parameter, which must be an object if specified. Right now only one option is available for it:

- `includeVertices` this is a boolean parameter that allows to modify the result of `EDGES()` . The default value for `includeVertices` is `false` , which does not have any effect. Setting it to `true` will modify the result, such that also the connected vertices are returned along with the edges:

  ```
  { vertex: <vertexDocument>, edge: <edgeDocument> }
  ```

Subquery optimizations for AQL queries

This optimization avoids copying intermediate results into subqueries that are not required by the subquery.

A brief description can be found here: http://jsteemann.github.io/blog/2015/05/04/subquery-optimizations/

Return value optimization for AQL queries

This optimization avoids copying the final query result inside the query's main `ReturnNode` .

A brief description can be found here: http://jsteemann.github.io/blog/2015/05/04/return-value-optimization-for-aql/

Speed up AQL queries containing big `IN` lists for index lookups

`IN` lists used for index lookups had performance issues in previous versions of ArangoDB. These issues have been addressed in 2.6 so using bigger `IN` lists for filtering is much faster.

A brief description can be found here: http://jsteemann.github.io/blog/2015/05/07/in-list-improvements/

Added alternative implementation for AQL COLLECT

The alternative method uses a hash table for grouping and does not require its input elements to be sorted. It will be taken into account by the optimizer for `COLLECT` statements that do not use an `INTO` clause.

In case a `COLLECT` statement can use the hash table variant, the optimizer will create an extra plan for it at the beginning of the planning phase. In this plan, no extra `SORT` node will be added in front of the `COLLECT` because the hash table variant of `COLLECT` does not require sorted input. Instead, a `SORT` node will be added after it to sort its output. This `SORT` node may be optimized away again in later stages. If the sort order of the result is irrelevant to the user, adding an extra `SORT null` after a hash `COLLECT` operation will allow the optimizer to remove the sorts altogether.

In addition to the hash table variant of `COLLECT`, the optimizer will modify the original plan to use the regular `COLLECT` implementation. As this implementation requires sorted input, the optimizer will insert a `SORT` node in front of the `COLLECT`. This `SORT` node may be optimized away in later stages.

The created plans will then be shipped through the regular optimization pipeline. In the end, the optimizer will pick the plan with the lowest estimated total cost as usual. The hash table variant does not require an up-front sort of the input, and will thus be preferred over the regular `COLLECT` if the optimizer estimates many input elements for the `COLLECT` node and cannot use an index to sort them.

The optimizer can be explicitly told to use the regular *sorted* variant of `COLLECT` by suffixing a `COLLECT` statement with `OPTIONS { "method" : "sorted" }`. This will override the optimizer guesswork and only produce the *sorted* variant of `COLLECT`.

A blog post on the new `COLLECT` implementation can be found here: http://jsteemann.github.io/blog/2015/04/22/collecting-with-a-hash-table/

Simplified return value syntax for data-modification AQL queries

ArangoDB 2.4 since version allows to return results from data-modification AQL queries. The syntax for this was quite limited and verbose:

```
FOR i IN 1..10
  INSERT { value: i } IN test
  LET inserted = NEW
  RETURN inserted
```

The `LET inserted = NEW RETURN inserted` was required literally to return the inserted documents. No calculations could be made using the inserted documents.

This is now more flexible. After a data-modification clause (e.g. `INSERT`, `UPDATE`, `REPLACE`, `REMOVE`, `UPSERT`) there can follow any number of `LET` calculations. These calculations can refer to the pseudo-values `OLD` and `NEW` that are created by the data-modification statements.

This allows returning projections of inserted or updated documents, e.g.:

```
FOR i IN 1..10
  INSERT { value: i } IN test
  RETURN { _key: NEW._key, value: i }
```

Still not every construct is allowed after a data-modification clause. For example, no functions can be called that may access documents.

More information can be found here: http://jsteemann.github.io/blog/2015/03/27/improvements-for-data-modification-queries/

Added AQL `UPSERT` statement

This adds an `UPSERT` statement to AQL that is a combination of both `INSERT` and `UPDATE` / `REPLACE` . The `UPSERT` will search for a matching document using a user-provided example. If no document matches the example, the *insert* part of the `UPSERT` statement will be executed. If there is a match, the *update* / *replace* part will be carried out:

```
UPSERT { page: 'index.html' }                 /* search example */
INSERT { page: 'index.html', pageViews: 1 } /* insert part */
UPDATE { pageViews: OLD.pageViews + 1 }     /* update part */
IN pageViews
```

`UPSERT` can be used with an `UPDATE` or `REPLACE` clause. The `UPDATE` clause will perform a partial update of the found document, whereas the `REPLACE` clause will replace the found document entirely. The `UPDATE` or `REPLACE` parts can refer to the pseudo-value `OLD` , which contains all attributes of the found document.

`UPSERT` statements can optionally return values. In the following query, the return attribute `found` will return the found document before the `UPDATE` was applied. If no document was found, `found` will contain a value of `null` . The `updated` result attribute will contain the inserted / updated document:

```
UPSERT { page: 'index.html' }                 /* search example */
INSERT { page: 'index.html', pageViews: 1 } /* insert part */
UPDATE { pageViews: OLD.pageViews + 1 }     /* update part */
IN pageViews
RETURN { found: OLD, updated: NEW }
```

A more detailed description of `UPSERT` can be found here: http://jsteemann.github.io/blog/2015/03/27/preview-of-the-upsert-command/

Miscellaneous changes

When errors occur inside AQL user functions, the error message will now contain a stacktrace, indicating the line of code in which the error occurred. This should make debugging AQL user functions easier.

Web Admin Interface

ArangoDB's built-in web interface now uses sessions. Session information is stored in cookies, so clients using the web interface must accept cookies in order to use it.

The new startup option `--server.session-timeout` can be used for adjusting the session lifetime.

The AQL editor in the web interface now provides an *explain* functionality, which can be used for inspecting and performance-tuning AQL queries. The query execution time is now also displayed in the AQL editor.

Foxx apps that require configuration or are missing dependencies are now indicated in the app overview and details.

Foxx improvements

Configuration and Dependencies

Foxx app manifests can now define configuration options, as well as dependencies on other Foxx apps.

An introduction to Foxx configurations can be found in the blog: https://www.arangodb.com/2015/05/reusable-foxx-apps-with-configurations/

And the blog post on Foxx dependencies can be found here: https://www.arangodb.com/2015/05/foxx-dependencies-for-more-composable-foxx-apps/

Mocha Tests

You can now write tests for your Foxx apps using the Mocha testing framework: https://www.arangodb.com/2015/04/testing-foxx-mocha/

A recipe for writing tests for your Foxx apps can be found in the cookbook: https://docs.arangodb.com/2.8/cookbook/FoxxTesting.html

API Documentation

The API documentation has been updated to Swagger 2. You can now also mount API documentation in your own Foxx apps.

Also see the blog post introducing this feature: https://www.arangodb.com/2015/05/document-your-foxx-apps-with-swagger-2/

Custom Scripts and Foxx Queue

In addition to the existing *setup* and *teardown* scripts you can now define custom scripts in your Foxx manifest and invoke these using the web admin interface or the Foxx manager CLI. These scripts can now also take positional arguments and export return values.

Job types for the Foxx Queue can now be defined as a script name and app mount path allowing the use of Foxx scripts as job types. The pre-2.6 job types are known to cause issues when restarting the server and are error-prone; we strongly recommended converting any existing job types to the new format.

Client tools

The default configuration value for the option `--server.request-timeout` was increased from 300 to 1200 seconds for all client tools (arangosh, arangoimp, arangodump, arangorestore).

The default configuration value for the option `--server.connect-timeout` was increased from 3 to 5 seconds for client tools (arangosh, arangoimp, arangodump, arangorestore).

Arangorestore

The option `--create-database` was added for arangorestore.

Setting this option to `true` will now create the target database if it does not exist. When creating the target database, the username and passwords passed to arangorestore will be used to create an initial user for the new database.

The default value for this option is `false`.

Arangoimp

Arangoimp can now optionally update or replace existing documents, provided the import data contains documents with `_key` attributes.

Previously, the import could be used for inserting new documents only, and re-inserting a document with an existing key would have failed with a *unique key constraint violated* error.

The behavior of arangoimp (insert, update, replace on duplicate key) can now be controlled with the option `--on-duplicate`. The option can have one of the following values:

- `error`: when a unique key constraint error occurs, do not import or update the document but report an error. This is the default.
- `update`: when a unique key constraint error occurs, try to (partially) update the existing document with the data specified in the import. This may still fail if the document would violate secondary unique indexes. Only the attributes present in the import data will be updated and other attributes already present will be preserved. The number of updated documents will be reported in the `updated` attribute of the HTTP API result.
- `replace`: when a unique key constraint error occurs, try to fully replace the existing document with the data specified in the import. This may still fail if the document would violate secondary unique indexes. The number of replaced documents will be reported in the `updated` attribute of the HTTP API result.
- `ignore`: when a unique key constraint error occurs, ignore this error. There will be no insert, update or replace for the particular document. Ignored documents will be reported separately in the `ignored` attribute of the HTTP API result.

The default value is `error`.

A few examples for using arangoimp with the `--on-duplicate` option can be found here: http://jsteemann.github.io/blog/2015/04/14/updating-documents-with-arangoimp/

Miscellaneous changes

- Some Linux-based ArangoDB packages are now using tcmalloc for memory allocator.
- Upgraded ICU library to version 54. This increases performance in many places.
- Allow to split an edge index into buckets which are resized individually. The default value is `1`, resembling the pre-2.6 behavior. Using multiple buckets will lead to the index entries being distributed to the individual buckets, with each bucket being responsible only for a fraction of the total index entries. Using multiple buckets may lead to more frequent but much faster index bucket resizes, and is recommended for bigger edge collections.
- Default configuration value for option `--server.backlog-size` was changed from 10 to 64.
- Default configuration value for option `--database.ignore-datafile-errors` was changed from `true` to `false`
- Document keys can now contain `@` and `.` characters
- Fulltext index can now index text values contained in direct sub-objects of the indexed attribute.

  Previous versions of ArangoDB only indexed the attribute value if it was a string. Sub-attributes of the index attribute were ignored when fulltext indexing.

  Now, if the index attribute value is an object, the object's values will each be included in the fulltext index if they are strings. If the index attribute value is an array, the array's values will each be included in the fulltext index if they are strings.

  For example, with a fulltext index present on the `translations` attribute, the following text values will now be indexed:

  ```
  var c = db._create("example");
  c.ensureFulltextIndex("translations");
  c.insert({ translations: { en: "fox", de: "Fuchs", fr: "renard", ru: "лиса" } });
  c.insert({ translations: "Fox is the English translation of the German word Fuchs" });
  c.insert({ translations: [ "ArangoDB", "document", "database", "Foxx" ] });

  c.fulltext("translations", "лиса").toArray();       // returns only first document
  c.fulltext("translations", "Fox").toArray();        // returns first and second documents
  c.fulltext("translations", "prefix:Fox").toArray(); // returns all three documents
  ```

- Added configuration option `--server.foxx-queues-poll-interval`

  This startup option controls the frequency with which the Foxx queues manager is checking the queue (or queues) for jobs to be executed.

  The default value is `1` second. Lowering this value will result in the queue manager waking up and checking the queues more frequently, which may increase CPU usage of the server. When not using Foxx queues, this value can be raised to save some CPU time.

- Added configuration option `--server.foxx-queues`

  This startup option controls whether the Foxx queue manager will check queue and job entries in the `_system` database only. Restricting the Foxx queue manager to the `_system` database will lead to the queue manager having to check only the queues collection of a single database, whereas making it check the queues of all databases might result in more work to be done and more CPU time to be used by the queue manager.

Incompatible changes in ArangoDB 2.6

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 2.6, and adjust any client programs if necessary.

Requirements

ArangoDB's built-in web interface now uses cookies for session management. Session information ids are stored in cookies, so clients using the web interface must accept cookies in order to log in and use it.

Foxx changes

Foxx Queues

Foxx Queue job type definitions were previously based on functions and had to be registered before use. Due to changes in 2.5 this resulted in problems when restarting the server or defining job types incorrectly.

Function-based job types have been deprecated in 2.6 and will be removed entirely in 2.7.

In order to convert existing function-based job types to the new script-based job types, create custom scripts in your Foxx app and reference them by their name and the mount point of the app they are defined in. Official job types from the Foxx app store can be upgraded by upgrading from the 1.x version to the 2.x version of the same app.

In order to upgrade queued jobs to the new job types, you need to update the `type` property of the affected jobs in the database's `_jobs` system collection. In order to see the collection in the web interface you need to enable the collection type "System" in the collection list options.

Example:

Before: `"type": "mailer.postmark"`

After: `"type": {"name": "mailer", "mount": "/my-postmark-mailer"}`

Foxx Sessions

The options `jwt` and `type` of the controller method `controller.activateSessions` have been deprecated in 2.6 and will be removed entirely in 2.7.

If you want to use pure JWT sessions, you can use the `sessions-jwt` Foxx app from the Foxx app store.

If you want to use your own JWT-based sessions, you can use the JWT functions in the `crypto` module directly.

Instead of using the `type` option you can just use the `cookie` and `header` options on their own, which both now accept the value `true` to enable them with their default configurations.

The option `sessionStorageApp` has been renamed to `sessionStorage` and now also accepts session storages directly. The old option `sessionStorageApp` will be removed entirely in 2.7.

Libraries

The bundled version of the `joi` library used in Foxx was upgraded to version 6.0.8. This may affect Foxx applications that depend on the library.

AQL changes

AQL LENGTH function

The return value of the AQL `LENGTH` function was changed if `LENGTH` is applied on `null` or a boolean value:

- `LENGTH(null)` now returns `0` . In previous versions of ArangoDB, this returned `4` .
- `LENGTH(false)` now returns `0` . In previous versions of ArangoDB, the return value was `5` .
- `LENGTH(true)` now returns `1` . In previous versions of ArangoDB, the return value was `4` .

AQL graph functions

In 2.6 the graph functions did undergo a performance lifting. During this process we had to adopt the result format and the options for some of them. Many graph functions now have an option `includeData` which allows to trigger if the result of this function should contain fully extracted documents `includeData: true` or only the `_id` values `includeData: false` . In most use cases the `_id` is sufficient to continue and the extraction of data is an unnecessary operation. The AQL functions supporting this additional option are:

- SHORTEST_PATH
- NEIGHBORS
- GRAPH_SHORTEST_PATH
- GRAPH_NEIGHBORS
- GRAPH_EDGES

Furthermore the result `SHORTEST_PATH` has changed. The old format returned a list of all vertices on the path. Optionally it could include each sub-path for these vertices. All of the documents were fully extracted. Example:

```
[
  {
    vertex: {
      _id: "vertex/1",
      _key: "1",
      _rev: "1234"
      name: "Alice"
    },
    path: {
      vertices: [
        {
          _id: "vertex/1",
          _key: "1",
          _rev: "1234"
          name: "Alice"
        }
      ],
      edges: []
    }
  },
  {
    vertex: {
      _id: "vertex/2",
      _key: "2",
      _rev: "5678"
      name: "Bob"
    },
    path: {
      vertices: [
        {
          _id: "vertex/1",
          _key: "1",
          _rev: "1234"
          name: "Alice"
        }, {
          _id: "vertex/2",
          _key: "2",
          _rev: "5678"
          name: "Bob"
        }
      ],
      edges: [
        {
          _id: "edge/1",
          _key: "1",
          _rev: "9876",
          type: "loves"
        }
      ]
    }
  }
]
```

The new version is more compact. Each `SHORTEST_PATH` will only return one document having the attributes `vertices` , `edges` , `distance` . The `distance` is computed taking into account the given weight. Optionally the documents can be extracted with `includeData: true` Example:

```
{
  vertices: [
    "vertex/1",
    "vertex/2"
  ],
  edges: [
    "edge/1"
  ],
  distance: 1
}
```

The next function that returns a different format is `NEIGHBORS` . Since 2.5 it returned an object with `edge` and `vertex` for each connected edge. Example:

```
[
  {
    vertex: {
      _id: "vertex/2",
      _key: "2",
      _rev: "5678"
      name: "Bob"
    },
    edge: {
      _id: "edge/1",
      _key: "1",
      _rev: "9876",
      type: "loves"
    }
  }
]
```

With 2.6 it will only return the vertex directly, again using `includeData: true`. By default it will return a distinct set of neighbors, using the option `distinct: false` will include the same vertex for each edge pointing to it.

Example:

```
[
  "vertex/2"
]
```

Function and API changes

Graph measurements functions

All graph measurements functions in JavaScript module `general-graph` that calculated a single figure previously returned an array containing just the figure. Now these functions will return the figure directly and not put it inside an array.

The affected functions are:

- `graph._absoluteEccentricity`
- `graph._eccentricity`
- `graph._absoluteCloseness`
- `graph._closeness`
- `graph._absoluteBetweenness`
- `graph._betweenness`
- `graph._radius`
- `graph._diameter`

Client programs calling these functions should be adjusted so they process the scalar value returned by the function instead of the previous array value.

Cursor API

A batchSize value `0` is now disallowed when calling the cursor API via HTTP `POST /_api/cursor`.

The HTTP REST API `POST /_api/cursor` does not accept a `batchSize` parameter value of `0` any longer. A batch size of 0 never made much sense, but previous versions of ArangoDB did not check for this value. Now creating a cursor using a `batchSize` value 0 will result in an HTTP 400 error response.

Document URLs returned

The REST API method GET `/_api/document?collection=...` (that method will return partial URLs to all documents in the collection) will now properly prefix document address URLs with the current database name.

Previous versions of ArangoDB returned the URLs starting with `/_api/` but without the current database name, e.g. `/_api/document/mycollection/mykey`. Starting with 2.6, the response URLs will include the database name as well, e.g. `/_db/_system/_api/document/mycollection/mykey`.

Fulltext indexing

Fulltext indexes will now also index text values contained in direct sub-objects of the indexed attribute.

Previous versions of ArangoDB only indexed the attribute value if it was a string. Sub-attributes of the index attribute were ignored when fulltext indexing.

Now, if the index attribute value is an object, the object's values will each be included in the fulltext index if they are strings. If the index attribute value is an array, the array's values will each be included in the fulltext index if they are strings.

Deprecated server functionality

Simple queries

The following simple query functions are now deprecated:

- collection.near
- collection.within
- collection.geo
- collection.fulltext
- collection.range
- collection.closedRange

This also lead to the following REST API methods being deprecated from now on:

- PUT /_api/simple/near
- PUT /_api/simple/within
- PUT /_api/simple/fulltext
- PUT /_api/simple/range

It is recommended to replace calls to these functions or APIs with equivalent AQL queries, which are more flexible because they can be combined with other operations:

```
FOR doc IN NEAR(@@collection, @latitude, @longitude, @limit)
  RETURN doc

FOR doc IN WITHIN(@@collection, @latitude, @longitude, @radius, @distanceAttributeName)
  RETURN doc

FOR doc IN FULLTEXT(@@collection, @attributeName, @queryString, @limit)
  RETURN doc

FOR doc IN @@collection
  FILTER doc.value >= @left && doc.value < @right
  LIMIT @skip, @limit
  RETURN doc`
```

The above simple query functions and REST API methods may be removed in future versions of ArangoDB.

Using negative values for `SimpleQuery.skip()` is also deprecated. This functionality will be removed in future versions of ArangoDB.

AQL functions

The AQL `SKIPLIST` function has been deprecated because it is obsolete.

The function was introduced in older versions of ArangoDB with a less powerful query optimizer to retrieve data from a skiplist index using a `LIMIT` clause.

Since 2.3 the same goal can be achieved by using regular AQL constructs, e.g.

```
FOR doc IN @@collection
  FILTER doc.value >= @value
  SORT doc.value
  LIMIT 1
  RETURN doc
```

Startup option changes

Options added

The following configuration options have been added in 2.6:

- `--server.session-timeout` : allows controlling the timeout of user sessions in the web interface. The value is specified in seconds.
- `--server.foxx-queues` : controls whether the Foxx queue manager will check queue and job entries. Disabling this option can reduce server load but will prevent jobs added to Foxx queues from being processed at all.

  The default value is `true` , enabling the Foxx queues feature.
- `--server.foxx-queues-poll-interval` : allows adjusting the frequency with which the Foxx queues manager is checking the queue (or queues) for jobs to be executed.

  The default value is `1` second. Lowering this value will result in the queue manager waking up and checking the queues more frequently, which may increase CPU usage of the server.

  Note: this option only has an effect when `--server.foxx-queues` is not set to `false` .

Options removed

The following configuration options have been removed in 2.6.:

- `--log.severity` : the docs for `--log.severity` mentioned lots of severities (e.g. `exception` , `technical` , `functional` , `development`) but only a few severities (e.g. `all` , `human`) were actually used, with `human` being the default and `all` enabling the additional logging of incoming requests.

  The option pretended to control a lot of things which it actually didn't. Additionally, the option `--log.requests-file` was around for a long time already, also controlling request logging.

  Because the `--log.severity` option effectively did not control that much, it was removed. A side effect of removing the option is that 2.5 installations started with option `--log.severity all` will not log requests after the upgrade to 2.6. This can be adjusted by setting the `--log.requests-file` option instead.

Default values changed

The default values for the following options have changed in 2.6:

- `--database.ignore-datafile-errors` : the default value for this option was changed from `true` to `false` .

  If the new default value of `false` is used, then arangod will refuse loading collections that contain datafiles with CRC mismatches or other errors. A collection with datafile errors will then become unavailable. This prevents follow up errors from happening.

  The only way to access such collection is to use the datafile debugger (arango-dfdb) and try to repair or truncate the datafile with it.
- `--server.request-timeout` : the default value was increased from 300 to 1200 seconds for all client tools (arangosh, arangoimp, arangodump, arangorestore).
- `--server.connect-timeout` : the default value was increased from 3 to 5 seconds for all client tools (arangosh, arangoimp, arangodump, arangorestore).

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.5. ArangoDB 2.5 also contains several bugfixes that are not listed here. For a list of bugfixes, please consult the CHANGELOG.

V8 version upgrade

The built-in version of V8 has been upgraded from 3.29.54 to 3.31.74.1. This allows activating additional ES6 (also dubbed *Harmony* or *ES.next*) features in ArangoDB, both in the ArangoShell and the ArangoDB server. They can be used for scripting and in server-side actions such as Foxx routes, traversals etc.

The following additional ES6 features become available in ArangoDB 2.5 by default:

- iterators and generators
- template strings
- enhanced object literals
- enhanced numeric literals
- block scoping with `let` and constant variables using `const` (note: constant variables require using strict mode, too)
- additional string methods (such as `startsWith` , `repeat` etc.)

Index improvements

Sparse hash and skiplist indexes

Hash and skiplist indexes can optionally be made sparse. Sparse indexes exclude documents in which at least one of the index attributes is either not set or has a value of `null` .

As such documents are excluded from sparse indexes, they may contain fewer documents than their non-sparse counterparts. This enables faster indexing and can lead to reduced memory usage in case the indexed attribute does occur only in some, but not all documents of the collection. Sparse indexes will also reduce the number of collisions in non-unique hash indexes in case non-existing or optional attributes are indexed.

In order to create a sparse index, an object with the attribute `sparse` can be added to the index creation commands:

```
db.collection.ensureHashIndex(attributeName, { sparse: true });
db.collection.ensureHashIndex(attributeName1, attributeName2, { sparse: true });
db.collection.ensureUniqueConstraint(attributeName, { sparse: true });
db.collection.ensureUniqueConstraint(attributeName1, attributeName2, { sparse: true });

db.collection.ensureSkiplist(attributeName, { sparse: true });
db.collection.ensureSkiplist(attributeName1, attributeName2, { sparse: true });
db.collection.ensureUniqueSkiplist(attributeName, { sparse: true });
db.collection.ensureUniqueSkiplist(attributeName1, attributeName2, { sparse: true });
```

Note that in place of the above specialized index creation commands, it is recommended to use the more general index creation command `ensureIndex` :

```
db.collection.ensureIndex({ type: "hash", sparse: true, unique: true, fields: [ attributeName ] });
db.collection.ensureIndex({ type: "skiplist", sparse: false, unique: false, fields: [ "a", "b" ] });
```

When not explicitly set, the `sparse` attribute defaults to `false` for new hash or skiplist indexes.

This causes a change in behavior when creating a unique hash index without specifying the sparse flag: in 2.4, unique hash indexes were implicitly sparse, always excluding `null` values. There was no option to control this behavior, and sparsity was neither supported for non-unique hash indexes nor skiplists in 2.4. This implicit sparsity of unique hash indexes was considered an inconsistency, and therefore the behavior was cleaned up in 2.5. As of 2.5, indexes will only be created sparse if sparsity is explicitly requested. Existing unique hash indexes from 2.4 or before will automatically be migrated so they are still sparse after the upgrade to 2.5.

Geo indexes are implicitly sparse, meaning documents without the indexed location attribute or containing invalid location coordinate values will be excluded from the index automatically. This is also a change when compared to pre-2.5 behavior, when documents with missing or invalid coordinate values may have caused errors on insertion when the geo index' `unique` flag was set and its `ignoreNull` flag was not. This was confusing and has been rectified in 2.5. The method `ensureGeoConstraint()` now does the same as `ensureGeoIndex()` . Furthermore, the attributes `constraint` , `unique` , `ignoreNull` and `sparse` flags are now completely ignored when creating geo indexes.

The same is true for fulltext indexes. There is no need to specify non-uniqueness or sparsity for geo or fulltext indexes.

As sparse indexes may exclude some documents, they cannot be used for every type of query. Sparse hash indexes cannot be used to find documents for which at least one of the indexed attributes has a value of `null` . For example, the following AQL query cannot use a sparse index, even if one was created on attribute `attr` :

```
FOR doc In collection
  FILTER doc.attr == null
  RETURN doc
```

If the lookup value is non-constant, a sparse index may or may not be used, depending on the other types of conditions in the query. If the optimizer can safely determine that the lookup value cannot be `null` , a sparse index may be used. When uncertain, the optimizer will not make use of a sparse index in a query in order to produce correct results.

For example, the following queries cannot use a sparse index on `attr` because the optimizer will not know beforehand whether the comparison values for `doc.attr` will include `null` :

```
FOR doc In collection
  FILTER doc.attr == SOME_FUNCTION(...)
  RETURN doc

FOR other IN otherCollection
  FOR doc In collection
    FILTER doc.attr == other.attr
    RETURN doc
```

Sparse skiplist indexes can be used for sorting if the optimizer can safely detect that the index range does not include `null` for any of the index attributes.

Selectivity estimates

Indexes of type `primary` , `edge` and `hash` now provide selectivity estimates. These will be used by the AQL query optimizer when deciding about index usage. Using selectivity estimates can lead to faster query execution when more selective indexes are used.

The selectivity estimates are also returned by the `GET /_api/index` REST API method in a sub-attribute `selectivityEstimate` for each index that supports it. This attribute will be omitted for indexes that do not provide selectivity estimates. If provided, the selectivity estimate will be a numeric value between 0 and 1.

Selectivity estimates will also be reported in the result of `collection.getIndexes()` for all indexes that support this. If no selectivity estimate can be determined for an index, the attribute `selectivityEstimate` will be omitted here, too.

The web interface also shows selectivity estimates for each index that supports this.

Currently the following index types can provide selectivity estimates:

- primary index
- edge index
- hash index (unique and non-unique)

No selectivity estimates will be provided for indexes when running in cluster mode.

AQL Optimizer improvements

Sort removal

The AQL optimizer rule "use-index-for-sort" will now remove sorts also in case a non-sorted index (e.g. a hash index) is used for only equality lookups and all sort attributes are covered by the equality lookup conditions.

For example, in the following query the extra sort on `doc.value` will be optimized away provided there is an index on `doc.value`):

```
FOR doc IN collection
  FILTER doc.value == 1
  SORT doc.value
  RETURN doc
```

The AQL optimizer rule "use-index-for-sort" now also removes sort in case the sort criteria excludes the left-most index attributes, but the left-most index attributes are used by the index for equality-only lookups.

For example, in the following query with a skiplist index on `value1`, `value2`, the sort can be optimized away:

```
FOR doc IN collection
  FILTER doc.value1 == 1
  SORT doc.value2
  RETURN doc
```

Constant attribute propagation

The new AQL optimizer rule `propagate-constant-attributes` will look for attributes that are equality-compared to a constant value, and will propagate the comparison value into other equality lookups. This rule will only look inside `FILTER` conditions, and insert constant values found in `FILTER`s, too.

For example, the rule will insert `42` instead of `i.value` in the second `FILTER` of the following query:

```
FOR i IN c1
  FOR j IN c2
    FILTER i.value == 42
    FILTER j.value == i.value
    RETURN 1
```

Interleaved processing

The optimizer will now inspect AQL data-modification queries and detect if the query's data-modification part can run in lockstep with the data retrieval part of the query, or if the data retrieval part must be executed and completed first before the data-modification can start.

Executing both data retrieval and data-modification in lockstep allows using much smaller buffers for intermediate results, reducing the memory usage of queries. Not all queries are eligible for this optimization, and the optimizer will only apply the optimization when it can safely detect that the data-modification part of the query will not modify data to be found by the retrieval part.

Query execution statistics

The `filtered` attribute was added to AQL query execution statistics. The value of this attribute indicates how many documents were filtered by `FilterNode`s in the AQL query. Note that `IndexRangeNode`s can also filter documents by selecting only the required ranges from the index. The `filtered` value will not include the work done by `IndexRangeNode`s, but only the work performed by `FilterNode`s.

Language improvements

Dynamic attribute names in AQL object literals

This change allows using arbitrary expressions to construct attribute names in object literals specified in AQL queries. To disambiguate expressions and other unquoted attribute names, dynamic attribute names need to be enclosed in brackets (`[` and `]`).

Example:

```
FOR i IN 1..100
  RETURN { [ CONCAT('value-of-', i) ] : i }
```

AQL functions

The following AQL functions were added in 2.5:

- `MD5(value)` : generates an MD5 hash of `value`
- `SHA1(value)` : generates an SHA1 hash of `value`
- `RANDOM_TOKEN(length)` : generates a random string value of the specified length

Simplify Foxx usage

Thanks to our user feedback we learned that Foxx is a powerful, yet rather complicated concept. With 2.5 we made it less complicated while keeping all its strength. That includes a rewrite of the documentation as well as some code changes as follows:

Moved Foxx applications to a different folder.

Until 2.4 foxx apps were stored in the following folder structure: `<app-path>/databases/<dbname>/<appname>:<appversion>` . This caused some trouble as apps where cached based on name and version and updates did not apply. Also the path on filesystem and the app's access URL had no relation to one another. Now the path on filesystem is identical to the URL (except the appended APP): `<app-path>/_db/<dbname>/<mointpoint>/APP`

Rewrite of Foxx routing

The routing of Foxx has been exposed to major internal changes we adjusted because of user feedback. This allows us to set the development mode per mountpoint without having to change paths and hold apps at separate locations.

Foxx Development mode

The development mode used until 2.4 is gone. It has been replaced by a much more mature version. This includes the deprecation of the javascript.dev-app-path parameter, which is useless since 2.5. Instead of having two separate app directories for production and development, apps now reside in one place, which is used for production as well as for development. Apps can still be put into development mode, changing their behavior compared to production mode. Development mode apps are still reread from disk at every request, and still they ship more debug output.

This change has also made the startup options `--javascript.frontend-development-mode` and `--javascript.dev-app-path` obsolete. The former option will not have any effect when set, and the latter option is only read and used during the upgrade to 2.5 and does not have any effects later.

Foxx install process

Installing Foxx apps has been a two step process: import them into ArangoDB and mount them at a specific mountpoint. These operations have been joined together. You can install an app at one mountpoint, that's it. No fetch, mount, unmount, purge cycle anymore. The commands have been simplified to just:

- install: get your Foxx app up and running
- uninstall: shut it down and erase it from disk

Foxx error output

Until 2.4 the errors produced by Foxx were not optimal. Often, the error message was just `unable to parse manifest` and contained only an internal stack trace. In 2.5 we made major improvements there, including a much more fine grained error output that helps you debug your Foxx apps. The error message printed is now much closer to its source and should help you track it down.

Also we added the default handlers for unhandled errors in Foxx apps:

- You will get a nice internal error page whenever your Foxx app is called but was not installed due to any error

- You will get a proper error message when having an uncaught error appears in any app route

In production mode the messages above will NOT contain any information about your Foxx internals and are safe to be exposed to third party users. In development mode the messages above will contain the stacktrace (if available), making it easier for your in-house devs to track down errors in the application.

Foxx console

We added a `console` object to Foxx apps. All Foxx apps now have a console object implementing the familiar Console API in their global scope, which can be used to log diagnostic messages to the database. This console also allows to read the error output of one specific foxx.

Foxx requests

We added `org/arangodb/request` module, which provides a simple API for making HTTP requests to external services. This is enables Foxx to be directly part of a micro service architecture.

Incompatible changes in ArangoDB 2.5

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 2.5, and adjust any client programs if necessary.

Changed behavior

V8

The V8 version shipped with ArangoDB was upgraded from 3.29.59 to 3.31.74.1. This leads to additional ECMAScript 6 (ES6 or "harmony") features being enabled by default in ArangoDB's scripting environment.

Apart from that, a change in the interpretation of command-line options by V8 may affect users. ArangoDB passes the value of the command-line option `--javascript.v8-options` to V8 and leaves interpretation of the contents to V8. For example, the ArangoDB option `--javascript.v8-options="--harmony"` could be used to tell V8 to enable its harmony features.

In ArangoDB 2.4, the following harmony options were made available by V8:

- --harmony_scoping (enable harmony block scoping)
- --harmony_modules (enable harmony modules (implies block scoping))
- --harmony_proxies (enable harmony proxies)
- --harmony_generators (enable harmony generators)
- --harmony_numeric_literals (enable harmony numeric literals (0o77, 0b11))
- --harmony_strings (enable harmony string)
- --harmony_arrays (enable harmony arrays)
- --harmony_arrow_functions (enable harmony arrow functions)
- --harmony_classes (enable harmony classes)
- --harmony_object_literals (enable harmony object literal extensions)
- --harmony (enable all harmony features (except proxies))

There was the option `--harmony`, which turned on almost all harmony features.

In ArangoDB 2.5, V8 provides the following harmony-related options:

- --harmony (enable all completed harmony features)
- --harmony_shipping (enable all shipped harmony features)
- --harmony_modules (enable "harmony modules (implies block scoping)" (in progress))
- --harmony_arrays (enable "harmony array methods" (in progress))
- --harmony_array_includes (enable "harmony Array.prototype.includes" (in progress))
- --harmony_regexps (enable "harmony regular expression extensions" (in progress))
- --harmony_arrow_functions (enable "harmony arrow functions" (in progress))
- --harmony_proxies (enable "harmony proxies" (in progress))
- --harmony_sloppy (enable "harmony features in sloppy mode" (in progress))
- --harmony_unicode (enable "harmony unicode escapes" (in progress))
- --harmony_tostring (enable "harmony toString")
- --harmony_numeric_literals (enable "harmony numeric literals")
- --harmony_strings (enable "harmony string methods")
- --harmony_scoping (enable "harmony block scoping")
- --harmony_classes (enable "harmony classes (implies block scoping & object literal extension)")
- --harmony_object_literals (enable "harmony object literal extensions")
- --harmony_templates (enable "harmony template literals")

Note that there are extra options for better controlling the dedicated features, and especially that the meaning of the `--harmony` option has changed from enabling **all** harmony features to **all completed** harmony features!

Users should adjust the value of `--javascript.v8-options` accordingly.

Please note that incomplete harmony features are subject to change in future V8 releases.

Sparse indexes

Hash indexes and skiplist indexes can now be created in a sparse variant. When not explicitly set, the `sparse` attribute defaults to `false` for new indexes.

This causes a change in behavior when creating a unique hash index without specifying the sparse flag. The unique hash index will be created in a non-sparse variant in ArangoDB 2.5.

In 2.4 and before, unique hash indexes were implicitly sparse, always excluding `null` values from the index. There was no option to control this behavior, and sparsity was neither supported for non-unique hash indexes nor skiplists in 2.4. This implicit sparsity of just unique hash indexes was considered an inconsistency, and therefore the behavior was cleaned up in 2.5.

As of 2.5, hash and skiplist indexes will only be created sparse if sparsity is explicitly requested. This may require a change in index-creating client code, but only if the client code creates unique hash indexes and if they are still intended to be sparse. In this case, the client code should explicitly set the `sparse` flag to `true` when creating a unique hash index.

Existing unique hash indexes from 2.4 or before will automatically be migrated so they are still sparse after the upgrade to 2.5. For these indexes, the `sparse` attribute will be populated automatically with a value of `true` .

Geo indexes are implicitly sparse, meaning documents without the indexed location attribute or containing invalid location coordinate values will be excluded from the index automatically. This is also a change when compared to pre-2.5 behavior, when documents with missing or invalid coordinate values may have caused errors on insertion when the geo index' `unique` flag was set and its `ignoreNull` flag was not.

This was confusing and has been rectified in 2.5. The method `ensureGeoConstraint()` now does the same as `ensureGeoIndex()` . Furthermore, the attributes `constraint` , `unique` , `ignoreNull` and `sparse` flags are now completely ignored when creating geo indexes. Client index creation code therefore does not need to set the `ignoreNull` or `constraint` attributes when creating a geo index.

The same is true for fulltext indexes. There is no need to specify non-uniqueness or sparsity for geo or fulltext indexes. They will always be non-unique and sparse.

Moved Foxx applications to a different folder.

Until 2.4 foxx apps were stored in the following folder structure: `<app-path>/databases/<dbname>/<appname>:<appversion>` . This caused some trouble as apps where cached based on name and version and updates did not apply. Also the path on filesystem and the app's access URL had no relation to one another. Now the path on filesystem is identical to the URL (except the appended APP): `<app-path>/_db/<dbname>/<mointpoint>/APP`

Foxx Development mode

The development mode used until 2.4 is gone. It has been replaced by a much more mature version. This includes the deprecation of the javascript.dev-app-path parameter, which is useless since 2.5. Instead of having two separate app directories for production and development, apps now reside in one place, which is used for production as well as for development. Apps can still be put into development mode, changing their behavior compared to production mode. Development mode apps are still reread from disk at every request, and still they ship more debug output.

This change has also made the startup options `--javascript.frontend-development-mode` and `--javascript.dev-app-path` obsolete. The former option will not have any effect when set, and the latter option is only read and used during the upgrade to 2.5 and does not have any effects later.

Foxx install process

Installing Foxx apps has been a two step process: import them into ArangoDB and mount them at a specific mountpoint. These operations have been joined together. You can install an app at one mountpoint, that's it. No fetch, mount, unmount, purge cycle anymore. The commands have been simplified to just:

- install: get your Foxx app up and running
- uninstall: shut it down and erase it from disk

Deprecated features

- Foxx: method `Model#toJSONSchema(id)` is deprecated, it will raise a warning if you use it. Please use `Foxx.toJSONSchema(id, model)` instead.

Removed features

- Startup switch `--javascript.frontend-development-mode` : Its major purpose was internal development anyway. Now the web frontend can be set to development mode similar to any other foxx app.
- Startup switch `--javascript.dev-app-path` : Was used for the development mode of Foxx. This is integrated with the normal app-path now and can be triggered on app level. The second app-path is superfluous.
- Foxx: `controller.collection` : Please use `appContext.collection` instead.
- Foxx: `FoxxRepository.modelPrototype` : Please use `FoxxRepository.model` instead.
- Foxx: `Model.extend({}, {attributes: {}})` : Please use `Model.extend({schema: {}})` instead.
- Foxx: `requestContext.bodyParam(paramName, description, Model)` : Please use `requestContext.bodyParam(paramName, options)` instead.
- Foxx: `requestContext.queryParam({type: string})` : Please use `requestContext.queryParam({type: joi})` instead.
- Foxx: `requestContext.pathParam({type: string})` : Please use `requestContext.pathParam({type: joi})` instead.
- Graph: The modules `org/arangodb/graph` and `org/arangodb/graph-blueprint` : Please use module `org/arangodb/general-graph` instead. NOTE: This does not mean we do not support blueprints any more. General graph covers everything the graph--blueprint did, plus many more features.
- General-Graph: In the module `org/arangodb/general-graph` the functions `_undirectedRelation` and `_directedRelation` are no longer available. Both functions have been unified to `_relation` .

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.4. ArangoDB 2.4 also contains several bugfixes that are not listed here. For a list of bugfixes, please consult the CHANGELOG.

V8 version upgrade

The built-in version of V8 has been upgraded from 3.16.14 to 3.29.59. This activates several ES6 (also dubbed *Harmony* or *ES.next*) features in ArangoDB, both in the ArangoShell and the ArangoDB server. They can be used for scripting and in server-side actions such as Foxx routes, traversals etc.

The following ES6 features are available in ArangoDB 2.4 by default:

- iterators
- the `of` operator
- symbols
- predefined collections types (Map, Set etc.)
- typed arrays

Many other ES6 features are disabled by default, but can be made available by starting arangod or arangosh with the appropriate options:

- arrow functions
- proxies
- generators
- String, Array, and Number enhancements
- constants
- enhanced object and numeric literals

To activate all these ES6 features in arangod or arangosh, start it with the following options:

```
arangosh --javascript.v8-options="--harmony --harmony_generators"
```

More details on the available ES6 features can be found in this blog.

FoxxGenerator

ArangoDB 2.4 is shipped with FoxxGenerator, a framework for building standardized Hypermedia APIs easily. The generated APIs can be consumed with client tools that understand Siren.

Hypermedia is the simple idea that our HTTP APIs should have links between their endpoints in the same way that our web sites have links between them. FoxxGenerator is based on the idea that you can represent an API as a statechart: Every endpoint is a state and the links are the transitions between them. Using your description of states and transitions, it can then create an API for you.

The FoxxGenerator can create APIs based on a semantic description of entities and transitions. A blog series on the use cases and how to use the Foxx generator is here:

- part 1
- part 2
- part 3

A cookbook recipe for getting started with FoxxGenerator is here.

AQL improvements

Optimizer improvements

The AQL optimizer has been enhanced to use of indexes in queries in several additional cases. Filters containing the `IN` operator can now make use of indexes, and multiple OR- or AND-combined filter conditions can now also use indexes if the filter conditions refer to the same indexed attribute.

Here are a few examples of queries that can now use indexes but couldn't before:

```
FOR doc IN collection
  FILTER doc.indexedAttribute == 1 || doc.indexedAttribute > 99
  RETURN doc

FOR doc IN collection
  FILTER doc.indexedAttribute IN [ 3, 42 ] || doc.indexedAttribute > 99
  RETURN doc

FOR doc IN collection
  FILTER (doc.indexedAttribute > 2 && doc.indexedAttribute < 10) ||
         (doc.indexedAttribute > 23 && doc.indexedAttribute < 42)
  RETURN doc
```

Additionally, the optimizer rule `remove-filter-covered-by-index` has been added. This rule removes FilterNodes and CalculationNodes from an execution plan if the filter condition is already covered by a previous IndexRangeNode. Removing the filter's CalculationNode and the FilterNode itself will speed up query execution because the query requires less computation.

Furthermore, the new optimizer rule `remove-sort-rand` will remove a `SORT RAND()` statement and move the random iteration into the appropriate `EnumerateCollectionNode` . This is usually more efficient than individually enumerating and sorting.

Data-modification queries returning documents

`INSERT` , `REMOVE` , `UPDATE` or `REPLACE` queries now can optionally return the documents inserted, removed, updated, or replaced. This is helpful for tracking the auto-generated attributes (e.g. `_key` , `_rev`) created by an `INSERT` and in a lot of other situations.

In order to return documents from a data-modification query, the statement must immediately be immediately followed by a `LET` statement that assigns either the pseudo-value `NEW` or `OLD` to a variable. This `LET` statement must be followed by a `RETURN` statement that returns the variable introduced by `LET` :

```
FOR i IN 1..100
  INSERT { value: i } IN test LET inserted = NEW RETURN inserted

FOR u IN users
  FILTER u.status == 'deleted'
  REMOVE u IN users LET removed = OLD RETURN removed

FOR u IN users
  FILTER u.status == 'not active'
  UPDATE u WITH { status: 'inactive' } IN users LET updated = NEW RETURN updated
```

`NEW` refers to the inserted or modified document revision, and `OLD` refers to the document revision before update or removal. `INSERT` statements can only refer to the `NEW` pseudo-value, and `REMOVE` operations only to `OLD` . `UPDATE` and `REPLACE` can refer to either.

In all cases the full documents will be returned with all their attributes, including the potentially auto-generated attributes such as `_id` , `_key` , or `_rev` and the attributes not specified in the update expression of a partial update.

Language improvements

`COUNT` clause

An optional `COUNT` clause was added to the `COLLECT` statement. The `COUNT` clause allows for more efficient counting of values.

In previous versions of ArangoDB one had to write the following to count documents:

```
RETURN LENGTH (
  FOR doc IN collection
  FILTER ...some condition...
  RETURN doc
)
```

With the `COUNT` clause, the query can be modified to

```
FOR doc IN collection
  FILTER ...some condition...
  COLLECT WITH COUNT INTO length
  RETURN length
```

The latter query will be much more efficient because it will not produce any intermediate results with need to be shipped from a subquery into the `LENGTH` function.

The `COUNT` clause can also be used to count the number of items in each group:

```
FOR doc IN collection
  FILTER ...some condition...
  COLLECT group = doc.group WITH COUNT INTO length
  return { group: group, length: length }
```

`COLLECT` modifications

In ArangoDB 2.4, `COLLECT` operations can be made more efficient if only a small fragment of the group values is needed later. For these cases, `COLLECT` provides an optional conversion expression for the `INTO` clause. This expression controls the value that is inserted into the array of group values. It can be used for projections.

The following query only copies the `dateRegistered` attribute of each document into the groups, potentially saving a lot of memory and computation time compared to copying `doc` completely:

```
FOR doc IN collection
  FILTER ...some condition...
  COLLECT group = doc.group INTO dates = doc.dateRegistered
  return { group: group, maxDate: MAX(dates) }
```

Compare this to the following variant of the query, which was the only way to achieve the same result in previous versions of ArangoDB:

```
FOR doc IN collection
  FILTER ...some condition...
  COLLECT group = doc.group INTO dates
  return { group: group, maxDate: MAX(dates[*].doc.dateRegistered) }
```

The above query will need to copy the full `doc` attribute into the `lengths` variable, whereas the new variant will only copy the `dateRegistered` attribute of each `doc` .

Subquery syntax

In previous versions of ArangoDB, subqueries required extra parentheses around them, and this caused confusion when subqueries were used as function parameters. For example, the following query did not work:

```
LET values = LENGTH(
  FOR doc IN collection RETURN doc
)
```

but had to be written as follows:

```
LET values = LENGTH((
  FOR doc IN collection RETURN doc
))
```

This was unintuitive and is fixed in version 2.4 so that both variants of the query are accepted and produce the same result.

Web interface

The `Applications` tab for Foxx applications in the web interface has got a complete redesign.

It will now only show applications that are currently running in ArangoDB. For a selected application, a new detailed view has been created. This view provides a better overview of the app, e.g.:

- author
- license
- version
- contributors
- download links
- API documentation

Installing a new Foxx application on the server is made easy using the new `Add application` button. The `Add application` dialog provides all the features already available in the `foxx-manager` console application plus some more:

- install a Foxx application from Github
- install a Foxx application from a zip file
- install a Foxx application from ArangoDB's application store
- create a new Foxx application from scratch: this feature uses a generator to create a Foxx application with pre-defined CRUD methods for a given list of collections. The generated Foxx app can either be downloaded as a zip file or be installed on the server. Starting with a new Foxx app has never been easier.

Miscellaneous improvements

Default endpoint is 127.0.0.1

The default endpoint for the ArangoDB server has been changed from `0.0.0.0` to `127.0.0.1`. This will make new ArangoDB installations unaccessible from clients other than localhost unless the configuration is changed. This is a security precaution measure that has been requested as a feature a lot of times.

If you are the development option `--enable-relative`, the endpoint will still be `0.0.0.0`.

System collections in replication

By default, system collections are now included in replication and all replication API return values. This will lead to user accounts and credentials data being replicated from master to slave servers. This may overwrite slave-specific database users.

If this is undesired, the `_users` collection can be excluded from replication easily by setting the `includeSystem` attribute to `false` in the following commands:

- replication.sync({ includeSystem: false });
- replication.applier.properties({ includeSystem: false });

This will exclude all system collections (including `_aqlfunctions`, `_graphs` etc.) from the initial synchronization and the continuous replication.

If this is also undesired, it is also possible to specify a list of collections to exclude from the initial synchronization and the continuous replication using the `restrictCollections` attribute, e.g.:

```
require("org/arangodb/replication").applier.properties({
  includeSystem: true,
  restrictType: "exclude",
  restrictCollections: [ "_users", "_graphs", "foo" ]
});
```

Incompatible changes in ArangoDB 2.4

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 2.4, and adjust any client programs if necessary.

Changed behavior

V8 upgrade

The bundled V8 version has been upgraded from 3.16.14 to 3.29.59.

The new version provides better error checking, which can lead to subtle changes in the execution of JavaScript code.

The following code, though nonsense, runs without error in 2.3 and 2.4 when strict mode is not enabled:

```
(function () {
  a = true;
  a.foo = 1;
})();
```

When enabling strict mode, the function will throw an error in 2.4 but not in 2.3:

```
(function () {
  "use strict";
  a = true;
  a.foo = 1;
})();

TypeError: Cannot assign to read only property 'foo' of true
```

Though this is a change in behavior it can be considered an improvement. The new version actually uncovers an error that went undetected in the old version.

Error messages have also changed slightly in the new version. Applications that rely on the exact error messages of the JavaScript engine may need to be adjusted so they look for the updated error messages.

Default endpoint

The default endpoint for arangod is now `127.0.0.1`.

This change will modify the IP address ArangoDB listens on to 127.0.0.1 by default. This will make new ArangoDB installations unaccessible from clients other than localhost unless the configuration is changed. This is a security feature.

To make ArangoDB accessible from any client, change the server's configuration (`--server.endpoint`) to either `tcp://0.0.0.0:8529` or the server's publicly visible IP address.

Replication

System collections are now included in the replication and all replication API return values by default.

This will lead to user accounts and credentials data being replicated from master to slave servers. This may overwrite slave-specific database users.

This may be considered a feature or an anti-feature, so it is configurable.

If replication of system collections is undesired, they can be excluded from replication by setting the `includeSystem` attribute to `false` in the following commands:

- initial synchronization: `replication.sync({ includeSystem: false })`
- continuous replication: `replication.applier.properties({ includeSystem: false })`

This will exclude all system collections (including `_aqlfunctions` , `_graphs` etc.) from the initial synchronization and the continuous replication.

If this is also undesired, it is also possible to specify a list of collections to exclude from the initial synchronization and the continuous replication using the `restrictCollections` attribute, e.g.:

```
require("org/arangodb/replication").applier.properties({
  includeSystem: true,
  restrictType: "exclude",
  restrictCollections: [ "_users", "_graphs", "foo" ]
});
```

The above example will in general include system collections, but will exclude the specified three collections from continuous replication.

The HTTP REST API methods for fetching the replication inventory and for dumping collections also support the `includeSystem` control flag via a URL parameter of the same name.

Build process changes

Several options for the `configure` command have been removed in 2.4. The options

- `--enable-all-in-one-v8`
- `--enable-all-in-one-icu`
- `--enable-all-in-one-libev`
- `--with-libev=DIR`
- `--with-libev-lib=DIR`
- `--with-v8=DIR`
- `--with-v8-lib=DIR`
- `--with-icu-config=FILE`

are not available anymore because the build process will always use the bundled versions of the libraries.

When building ArangoDB from source in a directory that already contained a pre-2.4 version, it will be necessary to run a `make superclean` command once and a full rebuild afterwards:

```
git pull
make superclean
make setup
./configure <options go here>
make
```

Miscellaneous changes

As a consequence of global renaming in the codebase, the option `mergeArrays` has been renamed to `mergeObjects` . This option controls whether JSON objects will be merged on an update operation or overwritten. The default has been, and still is, to merge. Not specifying the parameter will lead to a merge, as it has been the behavior in ArangoDB ever since.

This affects the HTTP REST API method PATCH `/_api/document/collection/key` . Its optional URL parameter `mergeArrays` for the option has been renamed to `mergeObjects` .

The AQL `UPDATE` statement is also affected, as its option `mergeArrays` has also been renamed to `mergeObjects` . The 2.3 query

```
UPDATE doc IN collection WITH { ... } IN collection OPTIONS { mergeArrays: false }
```

should thus be rewritten to the following in 2.4:

```
UPDATE doc IN collection WITH { ... } IN collection OPTIONS { mergeObjects: false }
```

Deprecated features

For `FoxxController` objects, the method `collection()` is deprecated and will be removed in future version of ArangoDB. Using this method will issue a warning. Please use `applicationContext.collection()` instead.

For `FoxxRepository` objects, the property `modelPrototype` is now deprecated. Using it will issue a warning. Please use `FoxxRepository.model` instead.

In `FoxxController` / `RequestContext` , calling method `bodyParam()` with three arguments is deprecated. Please use `.bodyParam(paramName, options)` instead.

In `FoxxController` / `RequestContext` calling method `queryParam({type: string})` is deprecated. Please use `requestContext.queryParam({type: joi})` instead.

In `FoxxController` / `RequestContext` calling method `pathParam({type: string})` is deprecated. Please use `requestContext.pathParam({type: joi})` instead.

For `FoxxModel` , calling `Model.extend({}, {attributes: {}})` is deprecated. Please use `Model.extend({schema: {}})` instead.

In module `org/arangodb/general-graph` , the functions `_undirectedRelation()` and `_directedRelation()` are deprecated and will be removed in a future version of ArangoDB. Both functions have been unified to `_relation()` .

The modules `org/arangodb/graph` and `org/arangodb/graph-blueprint` are deprecated. Please use module `org/arangodb/general-graph` instead.

The HTTP REST API `_api/graph` and all its methods are deprecated. Please use the general graph API `_api/gharial` instead.

Removed features

The following replication-related JavaScript methods became obsolete in ArangoDB 2.2 and have been removed in ArangoDB 2.4:

- `require("org/arangodb/replication").logger.start()`
- `require("org/arangodb/replication").logger.stop()`
- `require("org/arangodb/replication").logger.properties()`

The REST API methods for these functions have also been removed in ArangoDB 2.4:

- HTTP PUT `/_api/replication/logger-start`
- HTTP PUT `/_api/replication/logger-stop`
- HTTP GET `/_api/replication/logger-config`
- HTTP PUT `/_api/replication/logger-config`

Client applications that call one of these methods should be adjusted by removing the calls to these methods. This shouldn't be problematic as these methods have been no-ops since ArangoDB 2.2 anyway.

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.3. ArangoDB 2.3 also contains several bugfixes that are not listed here.

AQL improvements

Framework improvements

AQL queries are now sent through a query optimizer framework before execution. The query optimizer framework will first convert the internal representation of the query, the abstract syntax tree, into an initial execution plan.

The execution plan is then send through optimizer rules that may directly modify the plan in place or create a new variant of the plan. New plans might again be optimized, allowing the optimizer to carry out several optimizations.

After creating plans, the optimizer will estimate the costs for each plan and pick the plan with the lowest cost (termed the *optimal plan*) for the actual query execution.

With the `explain()` method of `ArangoStatement` users can check which execution plan the optimizer pick or retrieve a list of other plans that optimizer did not choose. The plan will reveal many details about which indexes are used etc. `explain()` will also return the of optimizer rules applied so users can validate whether or not a query allows using a specific optimization.

Execution of AQL queries has been rewritten in C++, allowing many queries to avoid the conversion of documents between ArangoDB's internal low-level data structure and the V8 object representation format.

The framework for optimizer rules is now also generally cluster-aware, allowing specific optimizations for queries that run in a cluster. Additionally, the optimizer was designed to be extensible in order to add more optimizations in the future.

Language improvements

Alternative operator syntax

ArangoDB 2.3 allows to use the following alternative forms for the logical operators:

- `AND` : logical and
- `OR` : logical or
- `NOT` : negation

This new syntax is just an alternative to the old syntax, allowing easier migration from SQL. The old syntax is still fully supported and will be:

- `&&` : logical and
- `||` : logical or
- `!` : negation

`NOT IN` operator

AQL now has a dedicated `NOT IN` operator.

Previously, a `NOT IN` was only achievable by writing a negated `IN` condition:

```
FOR i IN ... FILTER ! (i IN [ 23, 42 ]) ...
```

In ArangoDB 2.3, the same result can now alternatively be achieved by writing the more intuitive variant:

```
FOR i IN ... FILTER i NOT IN [ 23, 42 ] ...
```

Improvements of built-in functions

The following AQL string functions have been added:

- `LTRIM(value, characters)` : left-trims a string value
- `RTRIM(value, characters)` : right-trims a string value
- `FIND_FIRST(value, search, start, end)` : finds the first occurrence of a search string
- `FIND_LAST(value, search, start, end)` : finds the last occurrence of a search string
- `SPLIT(value, separator, limit)` : splits a string into an array, using a separator
- `SUBSTITUTE(value, search, replace, limit)` : replaces characters or strings inside another

The following other AQL functions have been added:

- `VALUES(document)` : returns the values of an object as a array (this is the counterpart to the already existing `ATTRIBUTES` function)
- `ZIP(attributes, values)` : returns an object constructed from attributes and values passed in separate parameters
- `PERCENTILE(values, n, method)` : returns the nths percentile of the values provided, using rank or interpolation method

The already existing functions `CONCAT` and `CONCAT_SEPARATOR` now support array arguments, e.g.:

```
/* "foobarbaz" */
CONCAT([ 'foo', 'bar', 'baz'])

/* "foo,bar,baz" */
CONCAT_SEPARATOR(", ", [ 'foo', 'bar', 'baz'])
```

AQL queries throw less exceptions

In previous versions of ArangoDB, AQL queries aborted with an exception in many situations and threw a runtime exception. For example, exceptions were thrown when trying to find a value using the `IN` operator in a non-array element, when trying to use non-boolean values with the logical operands `&&` or `||` or `!` , when using non-numeric values in arithmetic operations, when passing wrong parameters into functions etc.

The fact that many AQL operators could throw exceptions led to a lot of questions from users, and a lot of more-verbose-than-necessary queries. For example, the following query failed when there were documents that did not have a `topics` attribute at all:

```
FOR doc IN mycollection
  FILTER doc.topics IN [ "something", "whatever" ]
  RETURN doc
```

This forced users to rewrite the query as follows:

```
FOR doc IN mycollection
  FILTER IS_LIST(doc.topics) && doc.topics IN [ "something", "whatever" ]
  RETURN doc
```

In ArangoDB 2.3 this has been changed to make AQL easier to use. The change provides an extra benefit, and that is that non-throwing operators allow the query optimizer to perform much more transformations in the query without changing its overall result.

Here is a summary of changes:

- when a non-array value is used on the right-hand side of the `IN` operator, the result will be `false` in ArangoDB 2.3, and no exception will be thrown.
- the boolean operators `&&` and `||` do not throw in ArangoDB 2.3 if any of the operands is not a boolean value. Instead, they will perform an implicit cast of the values to booleans. Their result will be as follows:
  - `lhs && rhs` will return `lhs` if it is `false` or would be `false` when converted into a boolean. If `lhs` is `true` or would be `true` when converted to a boolean, `rhs` will be returned.
  - `lhs || rhs` will return `lhs` if it is `true` or would be `true` when converted into a boolean. If `lhs` is `false` or would be `false` when converted to a boolean, `rhs` will be returned.
  - `! value` will return the negated value of `value` converted into a boolean
- the arithmetic operators (`+` , `-` , `*` , `/` , `%`) can be applied to any value and will not throw exceptions when applied to non-numeric values. Instead, any value used in these operators will be casted to a numeric value implicitly. If no numeric result can be produced by an arithmetic operator, it will return `null` in ArangoDB 2.3. This is also true for division by zero.
- passing arguments of invalid types into AQL functions does not throw a runtime exception in most cases, but may produce runtime

warnings. Built-in AQL functions that receive invalid arguments will then return `null` .

Performance improvements

Non-unique hash indexes

The performance of insertion into *non-unique* hash indexes has been improved significantly. This fixes performance problems in case attributes were indexes that contained only very few distinct values, or when most of the documents did not even contain the indexed attribute. This also fixes problems when loading collections with such indexes.

The insertion time now scales linearly with the number of documents regardless of the cardinality of the indexed attribute.

Reverse iteration over skiplist indexes

AQL queries can now use a sorted skiplist index for reverse iteration. This allows several queries to run faster than in previous versions of ArangoDB.

For example, the following AQL query can now use the index on `doc.value` :

```
FOR doc IN mycollection
  FILTER doc.value > 23
  SORT doc.values DESC
  RETURN doc
```

Previous versions of ArangoDB did not use the index because of the descending (`DESC`) sort.

Additionally, the new AQL optimizer can use an index for sorting now even if the AQL query does not contain a `FILTER` statement. This optimization was not available in previous versions of ArangoDB.

Added basic support for handling binary data in Foxx

Buffer objects can now be used when setting the response body of any Foxx action. This allows Foxx actions to return binary data.

Requests with binary payload can be processed in Foxx applications by using the new method `res.rawBodyBuffer()` . This will return the unparsed request body as a Buffer object.

There is now also the method `req.requestParts()` available in Foxx to retrieve the individual components of a multipart HTTP request. That can be used for example to process file uploads.

Additionally, the `res.send()` method has been added as a convenience method for returning strings, JSON objects or Buffers from a Foxx action. It provides some auto-detection based on its parameter value:

```
res.send("<p>some HTML</p>");  // returns an HTML string
res.send({ success: true });   // returns a JSON object
res.send(new Buffer("some binary data"));  // returns binary data
```

The convenience method `res.sendFile()` can now be used to return the contents of a file from a Foxx action. They file may contain binary data:

```
res.sendFile(applicationContext.foxxFilename("image.png"));
```

The filesystem methods `fs.write()` and `fs.readBuffer()` can be used to work with binary data, too:

`fs.write()` will perform an auto-detection of its second parameter's value so it works with Buffer objects:

```
fs.write(filename, "some data");  // saves a string value in file
fs.write(filename, new Buffer("some binary data"));  // saves (binary) contents of a buffer
```

`fs.readBuffer()` has been added as a method to read the contents of an arbitrary file into a Buffer object.

Web interface

Batch document removal and move functionality has been added to the web interface, making it easier to work with multiple documents at once. Additionally, basic JSON import and export tools have been added.

Command-line options added

The command-line option `--javascript.v8-contexts` was added to arangod to provide better control over the number of V8 contexts created in arangod.

Previously, the number of V8 contexts arangod created at startup was equal to the number of server threads (as specified by option `--server.threads`).

In some situations it may be more sensible to create different amounts of threads and V8 contexts. This is because each V8 contexts created will consume memory and requires CPU resources for periodic garbage collection. Contrary, server threads do not have such high memory or CPU footprint.

If the option `--javascript.v8-contexts` is not specified, the number of V8 contexts created at startup will remain equal to the number of server threads. Thus no change in configuration is required to keep the same behavior as in previous ArangoDB versions.

The command-line option `--log.use-local-time` was added to print dates and times in ArangoDB's log in the server-local timezone instead of UTC. If it is not set, the timezone will default to UTC.

The option `--backslash-escape` has been added to arangoimp. Specifying this option will use the backslash as the escape character for literal quotes when parsing CSV files. The escape character for literal quotes is still the double quote character.

Miscellaneous improvements

ArangoDB's built-in HTTP server now supports HTTP pipelining.

The ArangoShell tutorial from the arangodb.com website is now integrated into the ArangoDB shell.

Powerful Foxx Enhancements

With the new **job queue** feature you can run async jobs to communicate with external services, **Foxx queries** make writing complex AQL queries much easier and **Foxx sessions** will handle the authentication and session hassle for you.

Foxx Queries

Writing long AQL queries in JavaScript can quickly become unwieldy. As of 2.3 ArangoDB bundles the ArangoDB Query Builder module that provides a JavaScript API for writing complex AQL queries without string concatenation. All built-in functions that accept AQL strings now support query builder instances directly. Additionally Foxx provides a method `Foxx.createQuery` for creating parametrized queries that can return Foxx models or apply arbitrary transformations to the query results.

Foxx Sessions

The session functionality in Foxx has been completely rewritten. The old `activateAuthentication` API is still supported but may be deprecated in the future. The new `activateSessions` API supports cookies or configurable headers, provides optional JSON Web Token and cryptographic signing support and uses the new sessions Foxx app.

ArangoDB 2.3 provides Foxx apps for user management and salted hash-based authentication which can be replaced with or supplemented by alternative implementations. For an example app using both the built-in authentication and OAuth2 see the Foxx Sessions Example app.

Foxx Queues

Foxx now provides async workers via the Foxx Queues API. Jobs enqueued in a job queue will be executed asynchronously outside of the request/response cycle of Foxx controllers and can be used to communicate with external services or perform tasks that take a long time to complete or may require multiple attempts.

Jobs can be scheduled in advance or set to be executed immediately, the number of retry attempts, the retry delay as well as success and failure handlers can be defined for each job individually. Job types that integrate various external services for transactional e-mails, logging and user tracking can be found in the Foxx app registry.

Misc

The request and response objects in Foxx controllers now provide methods for reading and writing raw cookies and signed cookies.

Mounted Foxx apps will now be loaded when arangod starts rather than at the first database request. This may result in slightly slower start up times (but a faster response for the first request).

Incompatible changes in ArangoDB 2.3

It is recommended to check the following list of incompatible changes **before** upgrading to ArangoDB 2.3, and adjust any client programs if necessary.

Configuration file changes

Threads and contexts

The number of server threads specified is now the minimum of threads started. There are situation in which threads are waiting for results of distributed database servers. In this case the number of threads is dynamically increased.

With ArangoDB 2.3, the number of server threads can be configured independently of the number of V8 contexts. The configuration option `--javascript.v8-contexts` was added to arangod to provide better control over the number of V8 contexts created in arangod.

Previously, the number of V8 contexts arangod created at startup was equal to the number of server threads (as specified by option `--server.threads`).

In some situations it may be more sensible to create different amounts of threads and V8 contexts. This is because each V8 contexts created will consume memory and requires CPU resources for periodic garbage collection. Contrary, server threads do not have such high memory or CPU footprint.

If the option `--javascript.v8-contexts` is not specified, the number of V8 contexts created at startup will remain equal to the number of server threads. Thus no change in configuration is required to keep the same behavior as in previous ArangoDB versions.

If you are using the default config files or merge them with your local config files, please review if the default number of server threads is okay in your environment. Additionally you should verify that the number of V8 contexts created (as specified in option `--javascript.v8-contexts`) is okay.

Syslog

The command-line option `--log.syslog` was used in previous versions of ArangoDB to turn logging to syslog on or off: when setting to a non-empty string, syslog logging was turned on, otherwise turned off. When syslog logging was turned on, logging was done with the application name specified in `--log.application`, which defaulted to `triagens`. There was also a command-line option `--log.hostname` which could be set but did not have any effect.

This behavior turned out to be unintuitive and was changed in 2.3 as follows:

- the command-line option `--log.syslog` is deprecated and does not have any effect when starting ArangoDB.
- to turn on syslog logging in 2.3, the option `--log.facility` has to be set to a non-empty string. The value for `facility` is OS-dependent (possible values can be found in `/usr/include/syslog.h` or the like - `user` should be available on many systems).
- the default value for `--log.application` has been changed from `triagens` to `arangod`.
- the command-line option `--log.hostname` is deprecated and does not have any effect when starting ArangoDB. Instead, the host name will be set by syslog automatically.
- when logging to syslog, ArangoDB now omits the datetime prefix and the process id, because they'll be added by syslog automatically.

AQL

AQL queries throw less exceptions

ArangoDB 2.3 contains a completely rewritten AQL query optimizer and execution engine. This means that AQL queries will be executed with a different engine than in ArangoDB 2.2 and earlier. Parts of AQL queries might be executed in different order than before because the AQL optimizer has more freedom to move things around in a query.

In previous versions of ArangoDB, AQL queries aborted with an exception in many situations and threw a runtime exception. Exceptions were thrown when trying to find a value using the `IN` operator in a non-array element, when trying to use non-boolean values with the logical operands `&&` or `||` or `!`, when using non-numeric values in arithmetic operations, when passing wrong parameters into functions etc.

In ArangoDB 2.3 this has been changed in many cases to make AQL more user-friendly and to allow the optimization to perform much more query optimizations.

Here is a summary of changes:

- when a non-array value is used on the right-hand side of the `IN` operator, the result will be `false` in ArangoDB 2.3, and no exception will be thrown.
- the boolean operators `&&` and `||` do not throw in ArangoDB 2.3 if any of the operands is not a boolean value. Instead, they will perform an implicit cast of the values to booleans. Their result will be as follows:
    - `lhs && rhs` will return `lhs` if it is `false` or would be `false` when converted into a boolean. If `lhs` is `true` or would be `true` when converted to a boolean, `rhs` will be returned.
    - `lhs || rhs` will return `lhs` if it is `true` or would be `true` when converted into a boolean. If `lhs` is `false` or would be `false` when converted to a boolean, `rhs` will be returned.
    - `! value` will return the negated value of `value` converted into a boolean
- the arithmetic operators (`+` , `-` , `*` , `/` , `%`) can be applied to any value and will not throw exceptions when applied to non-numeric values. Instead, any value used in these operators will be casted to a numeric value implicitly. If no numeric result can be produced by an arithmetic operator, it will return `null` in ArangoDB 2.3. This is also true for division by zero.
- passing arguments of invalid types into AQL functions does not throw a runtime exception in most cases, but may produce runtime warnings. Built-in AQL functions that receive invalid arguments will then return `null` .

Nested FOR loop execution order

The query optimizer in 2.3 may permute the order of nested `FOR` loops in AQL queries, provided that exchanging the loops will not alter a query result. However, a change in the order of returned values is allowed because no sort order is guaranteed by AQL (and was never) unless an explicit `SORT` statement is used in a query.

Changed return values of ArangoQueryCursor.getExtra()

The return value of `ArangoQueryCursor.getExtra()` has been changed in ArangoDB 2.3. It now contains a `stats` attribute with statistics about the query previously executed. It also contains a `warnings` attribute with warnings that happened during query execution. The return value structure has been unified in 2.3 for both read-only and data-modification queries.

The return value looks like this for a read-only query:

```
arangosh> stmt = db._createStatement("FOR i IN mycollection RETURN i"); stmt.execute().getExtra()
{
  "stats" : {
    "writesExecuted" : 0,
    "writesIgnored" : 0,
    "scannedFull" : 2600,
    "scannedIndex" : 0
  },
  "warnings" : [ ]
}
```

For data-modification queries, ArangoDB 2.3 returns a result with the same structure:

```
arangosh> stmt = db._createStatement("FOR i IN xx REMOVE i IN xx"); stmt.execute().getExtra()
{
  "stats" : {
    "writesExecuted" : 2600,
    "writesIgnored" : 0,
    "scannedFull" : 2600,
    "scannedIndex" : 0
  },
  "warnings" : [ ]
}
```

In ArangoDB 2.2, the return value of `ArangoQueryCursor.getExtra()` was empty for read-only queries and contained an attribute `operations` with two sub-attributes for data-modification queries:

```
arangosh> stmt = db._createStatement("FOR i IN mycollection RETURN i"); stmt.execute().getExtra()
{
}
```

```
arangosh> stmt = db._createStatement("FOR i IN mycollection REMOVE i IN mycollection"); stmt.execute().getExtra()
{
  "operations" : {
    "executed" : 2600,
    "ignored" : 0
  }
}
```

Changed return values in HTTP method `POST /_api/cursor`

The previously mentioned change also leads to the statistics being returned in the HTTP REST API method `POST /_api/cursor`. Previously, the return value contained an optional `extra` attribute that was filled only for data-modification queries and in some other cases as follows:

```
{
  "result" : [ ],
  "hasMore" : false,
  "extra" : {
    "operations" : {
      "executed" : 2600,
      "ignored" : 0
    }
  }
}
```

With the changed result structure in ArangoDB 2.3, the `extra` attribute in the result will look like this:

```
{
  "result" : [],
  "hasMore" : false,
  "extra" : {
    "stats" : {
      "writesExecuted" : 2600,
      "writesIgnored" : 0,
      "scannedFull" : 0,
      "scannedIndex" : 0
    },
    "warnings" : [ ]
  }
}
```

If the query option `fullCount` is requested, the `fullCount` result value will also be returned inside the `stats` attribute of the `extra` attribute, and not directly as an attribute inside the `extra` attribute as in 2.2. Note that a `fullCount` will only be present in `extra`.`stats` if it was requested as an option for the query.

The result in ArangoDB 2.3 will also contain a `warnings` attribute with the array of warnings that happened during query execution.

Changed return values in ArangoStatement.explain()

The return value of `ArangoStatement.explain()` has changed significantly in ArangoDB 2.3. The new return value structure is not compatible with the structure returned by 2.2.

In ArangoDB 2.3, the full execution plan for an AQL query is returned alongside all applied optimizer rules, optimization warnings etc. It is also possible to have the optimizer return all execution plans. This required a new data structure.

Client programs that use `ArangoStatement.explain()` or the HTTP REST API method `POST /_api/explain` may need to be adjusted to use the new return format.

The return value of `ArangoStatement.parse()` has been extended in ArangoDB 2.3. In addition to the existing attributes, ArangoDB 2.3 will also return an `ast` attribute containing the abstract syntax tree of the statement. This extra attribute can safely be ignored by client programs.

Variables not updatable in queries

Previous versions of ArangoDB allowed the modification of variables inside AQL queries, e.g.

```
LET counter = 0
FOR i IN 1..10
  LET counter = counter + 1
  RETURN counter
```

While this is admittedly a convenient feature, the new query optimizer design did not allow to keep it. Additionally, updating variables inside a query would prevent a lot of optimizations to queries that we would like the optimizer to make. Additionally, updating variables in queries that run on different nodes in a cluster would like cause non-deterministic behavior because queries are not executed linearly.

Changed return value of `TO_BOOL`

The AQL function `TO_BOOL` now always returns *true* if its argument is a array or an object. In previous versions of ArangoDB, the function returned *false* for empty arrays or for objects without attributes.

Changed return value of `TO_NUMBER`

The AQL function `TO_NUMBER` now returns *null* if its argument is an object or an array with more than one member. In previous version of ArangoDB, the return value in these cases was 0. `TO_NUMBER` will return 0 for empty array, and the numeric equivalent of the array member's value for arrays with a single member.

New AQL keywords

The following keywords have been added to AQL in ArangoDB 2.3:

- *NOT*
- *AND*
- *OR*

Unquoted usage of these keywords for attribute names in AQL queries will likely fail in ArangoDB 2.3. If any such attribute name needs to be used in a query, it should be enclosed in backticks to indicate the usage of a literal attribute name.

Removed features

Bitarray indexes

Bitarray indexes were only half-way documented and integrated in previous versions of ArangoDB so their benefit was limited. The support for bitarray indexes has thus been removed in ArangoDB 2.3. It is not possible to create indexes of type "bitarray" with ArangoDB 2.3.

When a collection is opened that contains a bitarray index definition created with a previous version of ArangoDB, ArangoDB will ignore it and log the following warning:

```
index type 'bitarray' is not supported in this version of ArangoDB and is ignored
```

Future versions of ArangoDB may automatically remove such index definitions so the warnings will eventually disappear.

Other removed features

The HTTP REST API method at `POST /_admin/modules/flush` has been removed.

Known issues

In ArangoDB 2.3.0, AQL queries containing filter conditions with an IN expression will not yet use an index:

```
FOR doc IN collection FILTER doc.indexedAttribute IN [ ... ] RETURN doc

FOR doc IN collection
  FILTER doc.indexedAttribute IN [ ... ]
  RETURN doc
```

We’re currently working on getting the IN optimizations done, and will ship them in a 2.3 maintenance release soon (e.g. 2.3.1 or 2.3.2).

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.2. ArangoDB 2.2 also contains several bugfixes that are not listed here.

AQL improvements

Data modification AQL queries

Up to including version 2.1, AQL supported data retrieval operations only. Starting with ArangoDB version 2.2, AQL also supports the following data modification operations:

- INSERT: insert new documents into a collection
- UPDATE: partially update existing documents in a collection
- REPLACE: completely replace existing documents in a collection
- REMOVE: remove existing documents from a collection

Data-modification operations are normally combined with other AQL statements such as *FOR* loops and *FILTER* conditions to determine the set of documents to operate on. For example, the following query will find all documents in collection *users* that match a specific condition and set their *status* variable to *inactive*:

```
FOR u IN users
  FILTER u.status == 'not active'
  UPDATE u WITH { status: 'inactive' } IN users
```

The following query copies all documents from collection *users* into collection *backup*:

```
FOR u IN users
  INSERT u IN backup
```

And this query removes documents from collection *backup*:

```
FOR doc IN backup
  FILTER doc.lastModified < DATE_NOW() - 3600
  REMOVE doc IN backup
```

For more information on data-modification queries, please refer to Data modification queries.

Updatable variables

Previously, the value of a variable assigned in an AQL query with the `LET` keyword was not updatable in an AQL query. This prevented statements like the following from being executable:

```
LET sum = 0
FOR v IN values
  SORT v.year
  LET sum = sum + v.value
  RETURN { year: v.year, value: v.value, sum: sum }
```

Other AQL improvements

- added AQL TRANSLATE function

  This function can be used to perform lookups from static objects, e.g.

```
LET countryNames = { US: "United States", UK: "United Kingdom", FR: "France" }
RETURN TRANSLATE("FR", countryNames)

LET lookup = { foo: "foo-replacement", bar: "bar-replacement", baz: "baz-replacement" }
RETURN TRANSLATE("foobar", lookup, "not contained!")
```

Write-ahead log

All write operations in an ArangoDB server will now be automatically logged in the server's write-ahead log. The write-ahead log is a set of append-only logfiles, and it is used in case of a crash recovery and for replication.

Data from the write-ahead log will eventually be moved into the journals or datafiles of collections, allowing the server to remove older write-ahead logfiles.

Cross-collection transactions in ArangoDB should benefit considerably by this change, as less writes than in previous versions are required to ensure the data of multiple collections are atomically and durably committed. All data-modifying operations inside transactions (insert, update, remove) will write their operations into the write-ahead log directly now. In previous versions, such operations were buffered until the commit or rollback occurred. Transactions with multiple operations should therefore require less physical memory than in previous versions of ArangoDB.

The data in the write-ahead log can also be used in the replication context. In previous versions of ArangoDB, replicating from a master required turning on a special replication logger on the master. The replication logger caused an extra write operation into the *_replication* system collection for each actual write operation. This extra write is now superfluous. Instead, slaves can read directly from the master's write-ahead log to get informed about most recent data changes. This removes the need to store data-modification operations in the *_replication* collection altogether.

For the configuration of the write-ahead log, please refer to Write-ahead log options.

The introduction of the write-ahead log also removes the need to configure and start the replication logger on a master. Though the replication logger object is still available in ArangoDB 2.2 to ensure API compatibility, starting, stopping, or configuring it will have no effect.

Performance improvements

- Removed sorting of attribute names when in collection shaper

  In previous versions of ArangoDB, adding a document with previously not-used attribute names caused a full sort of all attribute names used in the collection. The sorting was done to ensure fast comparisons of attribute names in some rare edge cases, but it considerably slowed down inserts into collections with many different or even unique attribute names.

- Specialized primary index implementation to allow faster hash table rebuilding and reduce lookups in datafiles for the actual value of `_key`. This also reduces the amount of random memory accesses for primary index inserts.

- Reclamation of index memory when deleting last document in collection

  Deleting documents from a collection did not lead to index sizes being reduced. Instead, the index memory was kept allocated and re-used later when a collection was refilled with new documents. Now, index memory of primary indexes and hash indexes is reclaimed instantly when the last document in a collection is removed.

- Prevent buffering of long print results in arangosh's and arangod's print command

  This change will emit buffered intermediate print results and discard the output buffer to quickly deliver print results to the user, and to prevent constructing very large buffers for large results.

Miscellaneous improvements

- Added `insert` method as an alias for `save`. Documents can now be inserted into a collection using either method:

  ```
  db.test.save({ foo: "bar" });
  db.test.insert({ foo: "bar" });
  ```

- Cleanup of options for data-modification operations

  Many of the data-modification operations had signatures with many optional bool parameters, e.g.:

  ```
  db.test.update("foo", { bar: "baz" }, true, true, true)
  db.test.replace("foo", { bar: "baz" }, true, true)
  db.test.remove("foo", true, true)
  db.test.save({ bar: "baz" }, true)
  ```

  Such long parameter lists were unintuitive and hard to use when only one of the optional parameters should have been set.

  To make the APIs more usable, the operations now understand the following alternative signature:

  ```
  collection.update(key, update-document, options)
  collection.replace(key, replacement-document, options)
  collection.remove(key, options)
  collection.save(document, options)
  ```

  Examples:

  ```
  db.test.update("foo", { bar: "baz" }, { overwrite: true, keepNull: true, waitForSync: true })
  db.test.replace("foo", { bar: "baz" }, { overwrite: true, waitForSync: true })
  db.test.remove("foo", { overwrite: true, waitForSync: true })
  db.test.save({ bar: "baz" }, { waitForSync: true })
  ```

- Added `--overwrite` option to arangoimp

  This allows removing all documents in a collection before importing into it using arangoimp.

- Honor startup option `--server.disable-statistics` when deciding whether or not to start periodic statistics collection jobs

  Previously, the statistics collection jobs were started even if the server was started with the `--server.disable-statistics` flag being set to `true`. Now if the option is set to `true`, no statistics will be collected on the server.

- Disallow storing of JavaScript objects that contain JavaScript native objects of type `Date`, `Function`, `RegExp` or `External`, e.g.

  ```
  db.test.save({ foo: /bar/ });
  db.test.save({ foo: new Date() });
  ```

  This will now print

  ```
  Error: <data> cannot be converted into JSON shape: could not shape document
  ```

  Previously, objects of these types were silently converted into an empty object (i.e. `{ }`) and no warning was issued.

  To store such objects in a collection, explicitly convert them into strings like this:

  ```
  db.test.save({ foo: String(/bar/) });
  db.test.save({ foo: String(new Date()) });
  ```

Removed features

MRuby integration for arangod

ArangoDB had an experimental MRuby integration in some of the publish builds. This wasn't continuously developed, and so it has been removed in ArangoDB 2.2.

This change has led to the following startup options being superfluous:

- `--ruby.gc-interval`
- `--ruby.action-directory`

- `--ruby.modules-path`
- `--ruby.startup-directory`

Specifying these startup options will do nothing in ArangoDB 2.2, so using these options should be avoided from now on as they might be removed in a future version of ArangoDB.

Removed startup options

The following startup options have been removed in ArangoDB 2.2. Specifying them in the server's configuration file will not produce an error to make migration easier. Still, usage of these options should be avoided as they will not have any effect and might fully be removed in a future version of ArangoDB:

- `--database.remove-on-drop`
- `--database.force-sync-properties`
- `--random.no-seed`
- `--ruby.gc-interval`
- `--ruby.action-directory`
- `--ruby.modules-path`
- `--ruby.startup-directory`
- `--server.disable-replication-logger`

Multi Collection Graphs

ArangoDB is a multi model database with native graph support. In version 2.2 the features for graphs have been improved by integration of a new graph module. All graphs created with the old module are automatically migrated into the new module but can still be used by the old module.

New graph module

Up to including version 2.1, ArangoDB offered a module for graphs and graph operations. This module allowed you to use exactly one edge collection together with one vertex collection in a graph. With ArangoDB version 2.2 this graph module is deprecated and a new graph module is offered. This new module allows to combine an arbitrary number of vertex collections and edge collections in the same graph. For each edge collection a list of collections containing source vertices and a list of collections containing target vertices can be defined. If an edge is stored ArangoDB checks if this edge is valid in this collection. Furthermore if a vertex is removed from one of the collections all connected edges will be removed as well, giving the guarantee of no loose ends in the graphs. The layout of the graph can be modified at runtime by adding or removing collections and changing the definitions for edge collections. All operations on the graph level are transactional by default.

Graphs in AQL

Multi collection graphs have been added to AQL as well. Basic functionality (getting vertices, edges, neighbors) can be executed using the entire graph. Also more advanced features like shortest path calculations, characteristic factors of the graph or traversals have been integrated into AQL. For these functions all graphs created with the graph module can be used.

Features and Improvements

The following list shows in detail which features have been added or improved in ArangoDB 2.1. ArangoDB 2.1 also contains several bugfixes that are not listed here.

New Edges Index

The edges index (used to store connections between nodes in a graph) internally uses a new data structure. This data structure improves the performance when populating the edge index (i.e. when loading an edge collection). For large graphs loading can be 20 times faster than with ArangoDB 2.0.

Additionally, the new index fixes performance problems that occurred when many duplicate `_from` or `_to` values were contained in the index. Furthermore, the new index supports faster removal of edges.

Finally, when loading an existing collection and building the edges index for the collection, less memory re-allocations will be performed.

Overall, this should considerably speed up loading edge collections.

The new index type replaces the old edges index type automatically, without any changes being required by the end user.

The API of the new index is compatible with the API of the old index. Still it is possible that the new index returns edges in a different order than the old index. This is still considered to be compatible because the old index had never guaranteed any result order either.

AQL Improvements

AQL offers functionality to work with dates. Dates are no data types of their own in AQL (neither they are in JSON, which is often used as a format to ship data into and out of ArangoDB). Instead, dates in AQL are internally represented by either numbers (timestamps) or strings. The date functions in AQL provide mechanisms to convert from a numeric timestamp to a string representation and vice versa.

There are two date functions in AQL to create dates for further use:

- `DATE_TIMESTAMP(date)` Creates a UTC timestamp value from `date`
- `DATE_TIMESTAMP(year, month, day, hour, minute, second, millisecond)` : Same as before, but allows specifying the individual date components separately. All parameters after `day` are optional.
- `DATE_ISO8601(date)` : Returns an ISO8601 datetime string from `date` . The datetime string will always use UTC time, indicated by the `Z` at its end.
- `DATE_ISO8601(year, month, day, hour, minute, second, millisecond)` : same as before, but allows specifying the individual date components separately. All parameters after `day` are optional.

These two above date functions accept the following input values:

- numeric timestamps, indicating the number of milliseconds elapsed since the UNIX epoch (i.e. January 1st 1970 00:00:00 UTC). An example timestamp value is `1399472349522` , which translates to `2014-05-07T14:19:09.522Z` .
- datetime strings in formats `YYYY-MM-DDTHH:MM:SS.MMM` , `YYYY-MM-DD HH:MM:SS.MMM` , or `YYYY-MM-DD` . Milliseconds are always optional.

  A timezone difference may optionally be added at the end of the string, with the hours and minutes that need to be added or subtracted to the datetime value. For example, `2014-05-07T14:19:09+01:00` can be used to specify a one hour offset, and `2014-05-07T14:19:09+07:30` can be specified for seven and half hours offset. Negative offsets are also possible. Alternatively to an offset, a `Z` can be used to indicate UTC / Zulu time.

  An example value is `2014-05-07T14:19:09.522Z` meaning May 7th 2014, 14:19:09 and 522 milliseconds, UTC / Zulu time. Another example value without time component is `2014-05-07Z` .

  Please note that if no timezone offset is specified in a datestring, ArangoDB will assume UTC time automatically. This is done to ensure portability of queries across servers with different timezone settings, and because timestamps will always be UTC-based.

- individual date components as separate function arguments, in the following order:
  - year
  - month
  - day
  - hour
  - minute
  - second
  - millisecond

  All components following `day` are optional and can be omitted. Note that no timezone offsets can be specified when using separate date components, and UTC / Zulu time will be used.

The following calls to `DATE_TIMESTAMP` are equivalent and will all return `1399472349522` :

```
DATE_TIMESTAMP("2014-05-07T14:19:09.522")
DATE_TIMESTAMP("2014-05-07T14:19:09.522Z")
DATE_TIMESTAMP("2014-05-07 14:19:09.522")
DATE_TIMESTAMP("2014-05-07 14:19:09.522Z")
DATE_TIMESTAMP(2014, 5, 7, 14, 19, 9, 522)
DATE_TIMESTAMP(1399472349522)
```

The same is true for calls to `DATE_ISO8601` that also accepts variable input formats:

```
DATE_ISO8601("2014-05-07T14:19:09.522Z")
DATE_ISO8601("2014-05-07 14:19:09.522Z")
DATE_ISO8601(2014, 5, 7, 14, 19, 9, 522)
DATE_ISO8601(1399472349522)
```

The above functions are all equivalent and will return `"2014-05-07T14:19:09.522Z"` .

The following date functions can be used with dates created by `DATE_TIMESTAMP` and `DATE_ISO8601` :

- `DATE_DAYOFWEEK(date)` : Returns the weekday number of `date` . The return values have the following meanings:
  - 0: Sunday
  - 1: Monday
  - 2: Tuesday
  - 3: Wednesday
  - 4: Thursday
  - 5: Friday
  - 6: Saturday
- `DATE_YEAR(date)` : Returns the year part of `date` as a number.
- `DATE_MONTH(date)` : Returns the month part of `date` as a number.
- `DATE_DAY(date)` : Returns the day part of `date` as a number.
- `DATE_HOUR(date)` : Returns the hour part of `date` as a number.
- `DATE_MINUTE(date)` : Returns the minute part of `date` as a number.
- `DATE_SECOND(date)` : Returns the seconds part of `date` as a number.
- `DATE_MILLISECOND(date)` : Returns the milliseconds part of `date` as a number.

The following other date functions are also available:

- `DATE_NOW()` : Returns the current time as a timestamp.

  Note that this function is evaluated on every invocation and may return different values when invoked multiple times in the same query.

The following other AQL functions have been added in ArangoDB 2.1:

- `FLATTEN` : this function can turn an array of sub-arrays into a single flat array. All array elements in the original array will be expanded recursively up to a configurable depth. The expanded values will be added to the single result array.

Example:

```
FLATTEN([ 1, 2, [ 3, 4 ], 5, [ 6, 7 ], [ 8, [ 9, 10 ] ])
```

will expand the sub-arrays on the first level and produce:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, [ 9, 10 ] ]
```

To fully flatten the array, the maximum depth can be specified (e.g. with a value of `2`):

```
FLATTEN([ 1, 2, [ 3, 4 ], 5, [ 6, 7 ], [ 8, [ 9, 10 ] ], 2)
```

This will fully expand the sub-arrays and produce:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

- `CURRENT_DATABASE` : this function will return the name of the database the current query is executed in.
- `CURRENT_USER` : this function returns the name of the current user that is executing the query. If authorization is turned off or the query is executed outside of a request context, no user is present and the function will return `null` .

Cluster Dump and Restore

The dump and restore tools, *arangodump* and *arangorestore*, can now be used to dump and restore collections in a cluster. Additionally, a collection dump from a standalone ArangoDB server can be imported into a cluster, and vice versa.

Web Interface Improvements

The web interface in version 2.1 has a more compact dashboard. It provides charts with time-series for incoming requests, HTTP transfer volume and some server resource usage figures.

Additionally it provides trend indicators (e.g. 15 min averages) and distribution charts (aka histogram) for some figures.

Foxx Improvements

To easily access a file inside the directory of a Foxx application from within Foxx, Foxx's `applicationContext` now provides the `foxxFilename()` function. It can be used to assemble the full filename of a file inside the application's directory. The `applicationContext` can be accessed as global variable from any module within a Foxx application.

The filename can be used inside Foxx actions or setup / teardown scripts, e.g. to populate a Foxx application's collection with data.

The `require` function now also prefers local modules when used from inside a Foxx application. This allows putting modules inside the Foxx application directory and requiring them easily. It also allows using application-specific versions of libraries that are bundled with ArangoDB (such as underscore.js).

Windows Installer

The Windows installer shipped with ArangoDB now supports installation of ArangoDB for the current user or all users, with the required privileges. It also supports the installation of ArangoDB as a service.

Fixes for 32 bit systems

Several issues have been fixed that occurred only when using ArangoDB on a 32 bits operating system, specifically:

- a crash in a third party component used to manage cluster data

- a third party library that failed to initialize on 32 bit Windows, making arangod and arangosh crash immediately.
- overflows of values used for nanosecond-precision timeouts: these overflows have led to invalid values being passed to socket operations, making them fail and re-try too often

Updated drivers

Several drivers for ArangoDB have been checked for compatibility with 2.1. The current list of drivers with compatibility notes can be found online here.

C++11 usage

We have moved several files from C to C++, allowing more code reuse and reducing the need for shipping data between the two. We have also decided to require C++11 support for ArangoDB, which allows us to use some of the simplifications, features and guarantees that this standard has in stock.

That also means a compiler with C++11 support is required to build ArangoDB from source. For instance GNU CC of at least version 4.8.

Miscellaneous Improvements

- Cancelable asynchronous jobs: several potentially long-running jobs can now be canceled via an explicit cancel operation. This allows stopping long-running queries, traversals or scripts without shutting down the complete ArangoDB process. Job cancelation is provided for asynchronously executed jobs as is described in @ref HttpJobCancel.
- Server-side periodic task management: an ArangoDB server now provides functionality to register and unregister periodic tasks. Tasks are user-defined JavaScript actions that can be run periodically and automatically, independent of any HTTP requests.

  The following task management functions are provided:

  - require("org/arangodb/tasks").register(): registers a periodic task
  - require("org/arangodb/tasks").unregister(): unregisters and removes a periodic task
  - require("org/arangodb/tasks").get(): retrieves a specific tasks or all existing tasks

  An example task (to be executed every 15 seconds) can be registered like this:

  ```
  var tasks = require("org/arangodb/tasks");
  tasks.register({
    name: "this is an example task with parameters",
    period: 15,
    command: function (params) {
      var greeting = params.greeting;
      var data = JSON.stringify(params.data);
      require('console').log('%s from parameter task: %s', greeting, data);
    },
    params: { greeting: "hi", data: "how are you?" }
  });
  ```

  Please refer to the section @ref Tasks for more details.

- The `figures` method of a collection now returns data about the collection's index memory consumption. The returned value `indexes.size` will contain the total amount of memory acquired by all indexes of the collection. This figure can be used to assess the memory impact of indexes.
- Capitalized HTTP response headers: from version 2.1, ArangoDB will return capitalized HTTP headers by default, e.g. `Content-Length` instead of `content-length`. Though the HTTP specification states that headers field name are case-insensitive, several older client tools rely on a specific case in HTTP response headers. This changes make ArangoDB a bit more compatible with those.
- Simplified usage of `db._createStatement()`: to easily run an AQL query, the method `db._createStatement` now allows passing the AQL query as a string. Previously it required the user to pass an object with a `query` attribute (which then contained the query string).

ArangoDB now supports both versions:

```
db._createStatement(queryString);
db._createStatement({ query: queryString });
```

Appendix

- References: Brief overviews over interfaces and objects
  - db: the `db` object
  - collection: the `collection` object
- JavaScript Modules: List of built-in and supported JS modules
- Deprecated: Features that are considered obsolete and may get removed eventually
- Error codes and meanings: List of all possible errors that can be encountered
- Glossary: Disambiguation page

References

The "db" Object

The `db` object is available in arangosh by default, and can also be imported and used in Foxx services.

db.name returns a collection object for the collection *name*.

The following methods exists on the _db object:

Database

- db._createDatabase(name, options, users)
- db._databases()
- db._dropDatabase(name, options, users)
- db._useDatabase(name)

Indexes

- db._index(index)
- db._dropIndex(index)

Properties

- db._id()
- db._isSystem()
- db._name()
- db._path()
- db._version()

Collection

- db._collection(name)
- db._create(name)
- db._drop(name)
- db._truncate(name)

AQL

- db._createStatement(query)
- db._query(query)
- db._explain(query)

Document

- db._document(object)
- db._exists(object)
- db._remove(selector)
- db._replace(selector,data)
- db._update(selector,data)

The "collection" Object

The following methods exist on the collection object (returned by *db.name*):

Collection

- collection.checksum()
- collection.count()
- collection.drop()
- collection.figures()
- collection.load()
- collection.properties()
- collection.reserve()
- collection.revision()
- collection.rotate()
- collection.toArray()
- collection.truncate()
- collection.type()
- collection.unload()

Indexes

- collection.dropIndex(index)
- collection.ensureIndex(description)
- collection.getIndexes(name)
- collection.index(index)

Document

- collection.all()
- collection.any()
- collection.closedRange(attribute, left, right)
- collection.document(object)
- collection.documents(keys)
- collection.edges(vertex-id)
- collection.exists(object)
- collection.firstExample(example)
- collection.inEdges(vertex-id)
- collection.insert(data)
- collection.edges(vertices)
- collection.iterate(iterator,options)
- collection.outEdges(vertex-id)
- collection.queryByExample(example)
- collection.range(attribute, left, right)
- collection.remove(selector)
- collection.removeByKeys(keys)
- collection.rename()
- collection.replace(selector, data)
- collection.replaceByExample(example, data)
- collection.save(data)
- collection.update(selector, data)
- collection.updateByExample(example, data)

JavaScript Modules

ArangoDB uses a Node.js compatible module system. You can use the function *require* in order to load a module or library. It returns the exported variables and functions of the module.

The global variables `global` , `process` , `console` , `Buffer` , `__filename` and `__dirname` are available throughout ArangoDB and Foxx.

Node compatibility modules

ArangoDB supports a number of modules for compatibility with Node.js, including:

- assert implements basic assertion and testing functions.
- buffer implements a binary data type for JavaScript.
- console is a well known logging facility to all the JavaScript developers. ArangoDB implements most of the Console API, with the exceptions of *profile* and *count*.
- events implements an event emitter.
- fs provides a file system API for the manipulation of paths, directories, files, links, and the construction of file streams. ArangoDB implements most Filesystem/A functions.
- module provides direct access to the module system.
- path implements functions dealing with filenames and paths.
- punycode implements conversion functions for punycode encoding.
- querystring provides utilities for dealing with query strings.
- stream provides a streaming interface.
- string_decoder implements logic for decoding buffers into strings.
- url provides utilities for URL resolution and parsing.
- util provides general utility functions like `format` and `inspect` .

Additionally ArangoDB provides partial implementations for the following modules:

- `net` : only `isIP` , `isIPv4` and `isIPv6` .
- `process` : only `env` and `cwd` ; stubs for `argv` , `stdout.isTTY` , `stdout.write` , `nextTick` .
- `timers` : stubs for `setImmediate` , `setTimeout` , `setInterval` , `clearImmediate` , `clearTimeout` , `clearInterval` and `ref` .
- `tty` : only `isatty` (always returns `false`).
- `vm` : only `runInThisContext` .

The following Node.js modules are not available at all: `child_process` , `cluster` , `constants` , `crypto` (but see `@arangodb/crypto` below), `dgram` , `dns` , `domain` , `http` , `https` , `os` , `sys` , `tls` , `v8` , `zlib` .

ArangoDB Specific Modules

There are a large number of ArangoDB-specific modules using the `@arangodb` namespace, mostly for internal use by ArangoDB itself. The following however are noteworthy:

- @arangodb/crypto provides various cryptography functions including hashing algorithms.
- @arangodb/foxx is the namespace providing the various building blocks of the Foxx microservice framework.

Bundled NPM Modules

The following NPM modules are preinstalled:

- aqb is the ArangoDB Query Builder and can be used to construct AQL queries with a chaining JavaScript API.
- chai is a full-featured assertion library for writing JavaScript tests.
- dedent is a simple utility function for formatting multi-line strings.
- error-stack-parser parses stacktraces into a more useful format.
- graphql-sync is an ArangoDB-compatible GraphQL server/schema implementation.
- highlight.js is an HTML syntax highlighter.
- i (inflect) is a utility library for inflecting (e.g. pluralizing) words.
- iconv-lite is a utility library for converting between character encodings
- joi is a validation library that is supported throughout the Foxx framework.
- js-yaml is a JavaScript implementation of the YAML data format (a partial superset of JSON).
- lodash is a utility belt for JavaScript providing various useful helper functions.
- minimatch is a glob matcher for matching wildcards in file paths.
- qs provides utilities for dealing with query strings using a different format than the **querystring** module.
- semver is a utility library for handling semver version numbers.
- sinon is a mocking library for writing test stubs, mocks and spies.
- timezone is a library for converting date time values between formats and timezones.

Console Module

`require('console')`

The implementation follows the CommonJS specification Console.

console.assert

`console.assert(expression, format, argument1, ...)`

Tests that an expression is *true*. If not, logs a message and throws an exception.

Examples

```
console.assert(value === "abc", "expected: value === abc, actual:", value);
```

console.debug

`console.debug(format, argument1, ...)`

Formats the arguments according to *format* and logs the result as debug message. Note that debug messages will only be logged if the server is started with log levels *debug* or *trace*.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Examples

```
console.debug("%s", "this is a test");
```

console.dir

`console.dir(object)`

Logs a listing of all properties of the object.

Example usage:

```
console.dir(myObject);
```

console.error

`console.error(format, argument1, ...)`

Formats the arguments according to @FA{format} and logs the result as error message.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Example usage:

```
console.error("error '%s': %s", type, message);
```

console.getline

`console.getline()`

Reads in a line from the console and returns it as string.

console.group

`console.group(format, argument1, ...)`

Formats the arguments according to *format* and logs the result as log message. Opens a nested block to indent all future messages sent. Call *groupEnd* to close the block. Representation of block is up to the platform, it can be an interactive block or just a set of indented sub messages.

Example usage:

```
console.group("user attributes");
console.log("name", user.name);
console.log("id", user.id);
console.groupEnd();
```

console.groupCollapsed

`console.groupCollapsed(format, argument1, ...)`

Same as *console.group*.

console.groupEnd

`console.groupEnd()`

Closes the most recently opened block created by a call to *group*.

console.info

`console.info(format, argument1, ...)`

Formats the arguments according to *format* and logs the result as info message.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Example usage:

```
console.info("The %s jumped over %d fences", animal, count);
```

console.log

`console.log(format, argument1, ...)`

Formats the arguments according to *format* and logs the result as log message. This is an alias for *console.info*.

console.time

`console.time(name)`

Creates a new timer under the given name. Call *timeEnd* with the same name to stop the timer and log the time elapsed.

Example usage:

```
console.time("mytimer");
...
console.timeEnd("mytimer"); // this will print the elapsed time
```

console.timeEnd

`console.timeEnd(name)`

Stops a timer created by a call to *time* and logs the time elapsed.

console.timeEnd

`console.trace()`

Logs a stack trace of JavaScript execution at the point where it is called.

console.warn

`console.warn(format, argument1, ...)`

Formats the arguments according to *format* and logs the result as warn message.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Crypto Module

`const crypto = require('@arangodb/crypto')`

The crypto module provides implementations of various hashing algorithms as well as cryptography related functions.

Nonces

These functions deal with cryptographic nonces.

createNonce

`crypto.createNonce(): string`

Creates a cryptographic nonce.

Returns the created nonce.

checkAndMarkNonce

`crypto.checkAndMarkNonce(nonce): void`

Checks and marks a nonce.

Arguments

- **nonce**: `string`

  The nonce to check and mark.

Returns nothing.

Random values

The following functions deal with generating random values.

rand

`crypto.rand(): number`

Generates a random integer that may be positive, negative or even zero.

Returns the generated number.

genRandomAlphaNumbers

`crypto.genRandomAlphaNumbers(length): string`

Generates a string of random alpabetical characters and digits.

Arguments

- **length**: `number`

  The length of the string to generate.

Returns the generated string.

genRandomNumbers

`crypto.genRandomNumbers(length): string`

Generates a string of random digits.

Arguments

- **length**: `number`

  The length of the string to generate.

Returns the generated string.

genRandomSalt

`crypto.genRandomSalt(length): string`

Generates a string of random (printable) ASCII characters.

Arguments

- **length**: `number`

  The length of the string to generate.

Returns the generated string.

JSON Web Tokens (JWT)

These methods implement the JSON Web Token standard.

jwtEncode

`crypto.jwtEncode(key, message, algorithm): string`

Generates a JSON Web Token for the given message.

Arguments

- **key**: `string | null`

  The secret cryptographic key to be used to sign the message using the given algorithm. Note that this function will raise an error if the key is omitted but the algorithm expects a key, and also if the algorithm does not expect a key but a key is provided (e.g. when using `"none"`).

- **message**: `string`

  Message to be encoded as JWT. Note that the message will only be base64-encoded and signed, not encrypted. Do not store sensitive information in tokens unless they will only be handled by trusted parties.

- **algorithm**: `string`

  Name of the algorithm to use for signing the message, e.g. `"HS512"` .

Returns the JSON Web Token.

jwtDecode

`crypto.jwtDecode(key, token, noVerify): string | null`

Arguments

- **key**: `string | null`

  The secret cryptographic key that was used to sign the message using the algorithm indicated by the token. Note that this function will raise an error if the key is omitted but the algorithm expects a key.

  If the algorithm does not expect a key but a key is provided, the token will fail to verify.

- **token**: `string`

  The token to decode.

Note that the function will raise an error if the token is malformed (e.g. does not have exactly three segments).

- **noVerify**: `boolean` (Default: `false`)

  Whether verification should be skipped. If this is set to `true` the signature of the token will not be verified. Otherwise the function will raise an error if the signature can not be verified using the given key.

Returns the decoded JSON message or `null` if no token is provided.

jwtAlgorithms

A helper object containing the supported JWT algorithms. Each attribute name corresponds to a JWT `alg` and the value is an object with `sign` and `verify` methods.

jwtCanonicalAlgorithmName

`crypto.jwtCanonicalAlgorithmName(name): string`

A helper function that translates a JWT `alg` value found in a JWT header into the canonical name of the algorithm in `jwtAlgorithms` . Raises an error if no algorithm with a matching name is found.

Arguments

- **name**: `string`

  Algorithm name to look up.

Returns the canonical name for the algorithm.

Hashing algorithms

md5

`crypto.md5(message): string`

Hashes the given message using the MD5 algorithm.

Arguments

- **message**: `string`

  The message to hash.

Returns the cryptographic hash.

sha1

`crypto.sha1(message): string`

Hashes the given message using the SHA-1 algorithm.

Arguments

- **message**: `string`

  The message to hash.

Returns the cryptographic hash.

sha224

`crypto.sha224(message): string`

Hashes the given message using the SHA-224 algorithm.

Arguments

- **message**: `string`

  The message to hash.

Returns the cryptographic hash.

sha256

`crypto.sha256(message): string`

Hashes the given message using the SHA-256 algorithm.

Arguments

- **message**: `string`

  The message to hash.

Returns the cryptographic hash.

sha384

`crypto.sha384(message): string`

Hashes the given message using the SHA-384 algorithm.

Arguments

- **message**: `string`

  The message to hash.

Returns the cryptographic hash.

sha512

`crypto.sha512(message): string`

Hashes the given message using the SHA-512 algorithm.

Arguments

- **message**: `string`

  The message to hash.

Returns the cryptographic hash.

Miscellaneous

constantEquals

`crypto.constantEquals(str1, str2): boolean`

Compares two strings. This function iterates over the entire length of both strings and can help making certain timing attacks harder.

Arguments

- **str1**: `string`

  The first string to compare.

- **str2**: `string`

  The second string to compare.

Returns `true` if the strings are equal, `false` otherwise.

pbkdf2

`crypto.pbkdf2(salt, password, iterations, keyLength): string`

Generates a PBKDF2-HMAC-SHA1 hash of the given password.

Arguments

- **salt**: `string`

  The cryptographic salt to hash the password with.

- **password**: `string`

  The message or password to hash.

- **iterations**: `number`

  The number of iterations. This should be a very high number. OWASP recommended 64000 iterations in 2012 and recommends doubling that number every two years.

  When using PBKDF2 for password hashes it is also recommended to add a random value (typically between 0 and 32000) to that number that is different for each user.

- **keyLength**: `number`

  The key length.

Returns the cryptographic hash.

hmac

`crypto.hmac(key, message, algorithm): string`

Generates an HMAC hash of the given message.

Arguments

- **key**: `string`

  The cryptographic key to use to hash the message.

- **message**: `string`

  The message to hash.

- **algorithm**: `string`

  The name of the algorithm to use.

Returns the cryptographic hash.

Filesystem Module

`require('fs')`

The implementation tries to follow the CommonJS specification where possible. Filesystem/A/0.

Single File Directory Manipulation

exists

checks if a file of any type or directory exists `fs.exists(path)`

Returns true if a file (of any type) or a directory exists at a given path. If the file is a broken symbolic link, returns false.

isFile

tests if path is a file `fs.isFile(path)`

Returns true if the *path* points to a file.

isDirectory

tests if path is a directory `fs.isDirectory(path)`

Returns true if the *path* points to a directory.

size

gets the size of a file `fs.size(path)`

Returns the size of the file specified by *path*.

mtime

gets the last modification time of a file `fs.mtime(filename)`

Returns the last modification date of the specified file. The date is returned as a Unix timestamp (number of seconds elapsed since January 1 1970).

pathSeparator

`fs.pathSeparator`

If you want to combine two paths you can use fs.pathSeparator instead of */* or **.

join

`fs.join(path, filename)`

The function returns the combination of the path and filename, e.g. fs.join(Hello/World, foo.bar) would return Hello/World/foo.bar.

getTempFile

returns the name for a (new) temporary file `fs.getTempFile(directory, createFile)`

Returns the name for a new temporary file in directory *directory*. If *createFile* is *true*, an empty file will be created so no other process can create a file of the same name.

Note: The directory *directory* must exist.

getTempPath

returns the temporary directory `fs.getTempPath()`

Returns the absolute path of the temporary directory

makeAbsolute

makes a given path absolute `fs.makeAbsolute(path)`

Returns the given string if it is an absolute path, otherwise an absolute path to the same location is returned.

chmod

sets file permissions of specified files (non windows only) `fs.exists(path)`

Returns true on success.

list

returns the directory listing `fs.list(path)`

The functions returns the names of all the files in a directory, in lexically sorted order. Throws an exception if the directory cannot be traversed (or path is not a directory).

Note: this means that list("x") of a directory containing "a" and "b" would return ["a", "b"], not ["x/a", "x/b"].

listTree

returns the directory tree `fs.listTree(path)`

The function returns an array that starts with the given path, and all of the paths relative to the given path, discovered by a depth first traversal of every directory in any visited directory, reporting but not traversing symbolic links to directories. The first path is always *""*, the path relative to itself.

makeDirectory

creates a directory `fs.makeDirectory(path)`

Creates the directory specified by *path*.

makeDirectoryRecursive

creates a directory `fs.makeDirectoryRecursive(path)`

Creates the directory hierarchy specified by *path*.

remove

removes a file `fs.remove(filename)`

Removes the file *filename* at the given path. Throws an exception if the path corresponds to anything that is not a file or a symbolic link. If "path" refers to a symbolic link, removes the symbolic link.

removeDirectory

removes an empty directory `fs.removeDirectory(path)`

Removes a directory if it is empty. Throws an exception if the path is not an empty directory.

removeDirectoryRecursive

removes a directory `fs.removeDirectoryRecursive(path)`

Removes a directory with all subelements. Throws an exception if the path is not a directory.

File IO

read

reads in a file `fs.read(filename)`

Reads in a file and returns the content as string. Please note that the file content must be encoded in UTF-8.

read64

reads in a file as base64 `fs.read64(filename)`

Reads in a file and returns the content as string. The file content is Base64 encoded.

readBuffer

reads in a file `fs.readBuffer(filename)`

Reads in a file and returns its content in a Buffer object.

readFileSync

`fs.readFileSync(filename, encoding)`

Reads the contents of the file specified in `filename`. If `encoding` is specified, the file contents will be returned as a string. Supported encodings are:

- `utf8` or `utf-8`
- `ascii`
- `base64`
- `ucs2` or `ucs-2`
- `utf16le` or `utf16be`
- `hex`

If no `encoding` is specified, the file contents will be returned in a Buffer object.

write

`fs.write(filename, content)`

Writes the content into a file. Content can be a string or a Buffer object. If the file already exists, it is truncated.

writeFileSync

`fs.writeFileSync(filename, content)`

This is an alias for `fs.write(filename, content)`.

append

`fs.append(filename, content)`

Writes the content into a file. Content can be a string or a Buffer object. If the file already exists, the content is appended at the end.

Recursive Manipulation

copyRecursive

copies a directory structure `fs.copyRecursive(source, destination)`

Copies *source* to *destination*. Exceptions will be thrown on:

- Failure to copy the file
- specifying a directory for destination when source is a file
- specifying a directory as source and destination

CopyFile

copies a file into a target file `fs.copyFile(source, destination)`

Copies *source* to destination. If Destination is a directory, a file of the same name will be created in that directory, else the copy will get the specified filename.

move

renames a file `fs.move(source, destination)`

Moves *source* to destination. Failure to move the file, or specifying a directory for destination when source is a file will throw an exception. Likewise, specifying a directory as source and destination will fail.

ZIP

unzipFile

unzips a file `fs.unzipFile(filename, outpath, skipPaths, overwrite, password)`

Unzips the zip file specified by *filename* into the path specified by *outpath*. Overwrites any existing target files if *overwrite* is set to *true*.

Returns *true* if the file was unzipped successfully.

zipFile

zips a file `fs.zipFile(filename, chdir, files, password)`

Stores the files specified by *files* in the zip file *filename*. If the file *filename* already exists, an error is thrown. The list of input files *files* must be given as a list of absolute filenames. If *chdir* is not empty, the *chdir* prefix will be stripped from the filename in the zip file, so when it is unzipped filenames will be relative. Specifying a password is optional.

Returns *true* if the file was zipped successfully.

Module "request"

The request module provides the functionality for making HTTP requests.

`require('@arangodb/request')`

Making HTTP requests

HTTP method helpers

In addition to the *request* function convenience shorthands are available for each HTTP method in the form of, i.e.:

- `request.head(url, options)`
- `request.get(url, options)`
- `request.post(url, options)`
- `request.put(url, options)`
- `request.delete(url, options)`
- `request.patch(url, options)`

These are equivalent to using the *request* function directly, i.e.:

```
request[method](url, options)
// is equivalent to
request({method, url, ...options});
```

For example:

```
const request = require('@arangodb/request');

request.get('http://localhost', {headers: {'x-session-id': 'keyboardcat'}});
// is equivalent to
request({
  method: 'get',
  url: 'http://localhost',
  headers: {'x-session-id': 'keyboardcat'}
});
```

The request function

The request function can be used to make HTTP requests.

`request(options)`

Performs an HTTP request and returns a *Response* object.

Parameter

The request function takes the following options:

- *url* or *uri*: the fully-qualified URL or a parsed URL from `url.parse` .
- *qs* (optional): object containing querystring values to be appended to the URL.
- *useQuerystring*: if `true` , use `querystring` module to handle querystrings, otherwise use `qs` module. Default: `false` .
- *method* (optional): HTTP method (case-insensitive). Default: `"GET"` .
- *headers* (optional): HTTP headers (case-insensitive). Default: `{}` .
- *body* (optional): request body. Must be a string or `Buffer` , or a JSON serializable value if *json* is `true` .
- *json*: if `true` , *body* will be serialized to a JSON string and the *Content-Type* header will be set to `"application/json"` . Additionally the response body will also be parsed as JSON (unless *encoding* is set to `null`). Default: `false` .
- *form* (optional): when set to a string or object and no *body* has been set, *body* will be set to a querystring representation of that value and the *Content-Type* header will be set to `"application/x-www-form-urlencoded"` . Also see *useQuerystring*.
- *auth* (optional): an object with the properties *username* and *password* for HTTP Basic authentication or the property *bearer* for

HTTP Bearer token authentication.
- *followRedirect*: whether HTTP 3xx redirects should be followed. Default: `true`.
- *maxRedirects*: the maximum number of redirects to follow. Default: `10`.
- *encoding*: encoding to be used for the response body. If set to `null`, the response body will be returned as a `Buffer`. Default: `"utf-8"`.
- *timeout*: number of milliseconds to wait for a response before aborting the request.
- *returnBodyOnError*: whether the response body should be returned even when the server response indicates an error. Default: `true`.

The function returns a *Response* object with the following properties:

- *rawBody*: the raw response body as a `Buffer`.
- *body*: the parsed response body. If *encoding* was not set to `null`, this is a string. If additionally *json* was set to `true` and the response body is well-formed JSON, this is the parsed JSON data.
- *headers*: an object containing the response headers. Otherwise this is identical to *rawBody*.
- *statusCode* and *status*: the HTTP status code of the response, e.g. `404`.
- *message*: the HTTP status message of the response, e.g. `Not Found`.

Forms

The request module supports `application/x-www-form-urlencoded` (URL encoded) form uploads:

```
const request = require('@arangodb/request');

var res = request.post('http://service.example/upload', {form: {key: 'value'}});
// or
var res = request.post({url: 'http://service.example/upload', form: {key: 'value'}});
// or
var res = request({
  method: 'post',
  url: 'http://service.example/upload',
  form: {key: 'value'}
});
```

Form data will be encoded using the qs module by default.

If you want to use the querystring module instead, simply use the *useQuerystring* option.

JSON

If you want to submit JSON-serializable values as request bodies, just set the *json* option:

```
const request = require('@arangodb/request');

var res = request.post('http://service.example/notify', {body: {key: 'value'}, json: true});
// or
var res = request.post({url: 'http://service.example/notify', body: {key: 'value'}, json: true});
// or
var res = request({
  method: 'post',
  url: 'http://service.example/notify',
  body: {key: 'value'},
  json: true
});
```

HTTP authentication

The request module supports both *HTTP Basic* authentication. Just pass the credentials via the *auth* option:

```
const request = require('@arangodb/request');

var res = request.get(
  'http://service.example/secret',
  {auth: {username: 'jcd', password: 'bionicman'}}
);
// or
var res = request.get({
  url: 'http://service.example/secret',
  auth: {username: 'jcd', password: 'bionicman'}
});
// or
var res = request({
  method: 'get',
  url: 'http://service.example/secret',
  auth: {username: 'jcd', password: 'bionicman'}
});
```

Alternatively you can supply the credentials via the URL:

```
const request = require('@arangodb/request');

var username = 'jcd';
var password = 'bionicman';
var res = request.get(
  'http://' +
  encodeURIComponent(username) +
  ':' +
  encodeURIComponent(password) +
  '@service.example/secret'
);
```

You can also use *Bearer* token authentication:

```
const request = require('@arangodb/request');

var res = request.get(
  'http://service.example/secret',
  {auth: {bearer: 'keyboardcat'}}
);
// or
var res = request.get({
  url: 'http://service.example/secret',
  auth: {bearer: 'keyboardcat'}
});
// or
var res = request({
  method: 'get',
  url: 'http://service.example/secret',
  auth: {bearer: 'keyboardcat'}
});
```

Module "actions"

The action module provides the infrastructure for defining low-level HTTP actions.

If you want to define HTTP endpoints in ArangoDB you should probably use the Foxx microservice framework instead.

Basics

Error message

`actions.getErrorMessage(code)`

Returns the error message for an error code.

Standard HTTP Result Generators

`actions.defineHttp(options)`

Defines a new action. The *options* are as follows:

`options.url`

The URL, which can be used to access the action. This path might contain slashes. Note that this action will also be called, if a url is given such that *options.url* is a prefix of the given url and no longer definition matches.

`options.prefix`

If *false*, then only use the action for exact matches. The default is *true*.

`options.callback(request, response)`

The request argument contains a description of the request. A request parameter *foo* is accessible as *request.parametrs.foo*. A request header *bar* is accessible as *request.headers.bar*. Assume that the action is defined for the url */foo/bar* and the request url is */foo/bar/hugo/egon*. Then the suffix parts *["hugo", "egon"]* are availible in *request.suffix*.

The callback must define fill the *response*.

- *response.responseCode*: the response code
- *response.contentType*: the content type of the response
- *response.body*: the body of the response

You can use the functions *ResultOk* and *ResultError* to easily generate a response.

Result ok

`actions.resultOk(req, res, code, result, headers)`

The function defines a response. *code* is the status code to return. *result* is the result object, which will be returned as JSON object in the body. *headers* is an array of headers to returned. The function adds the attribute *error* with value *false* and *code* with value *code* to the *result*.

Result bad

`actions.resultBad(req, res, error-code, msg, headers)`

The function generates an error response.

Result not found

`actions.resultNotFound(req, res, code, msg, headers)`

The function generates an error response.

Result unsupported

`actions.resultUnsupported(req, res, headers)`

The function generates an error response.

Result error

*actions.resultError(*req, res, code, errorNum, errorMessage, headers, keyvals)*

The function generates an error response. The response body is an array with an attribute *errorMessage* containing the error message *errorMessage*, *error* containing *true*, *code* containing *code*, *errorNum* containing *errorNum*, and *errorMessage* containing the error message *errorMessage*. *keyvals* are mixed into the result.

Result not Implemented

`actions.resultNotImplemented(req, res, msg, headers)`

The function generates an error response.

Result permanent redirect

`actions.resultPermanentRedirect(req, res, options, headers)`

The function generates a redirect response.

Result temporary redirect

`actions.resultTemporaryRedirect(req, res, options, headers)`

The function generates a redirect response.

ArangoDB Result Generators

Collection not found

`actions.collectionNotFound(req, res, collection, headers)`

The function generates an error response.

Index not found

`actions.indexNotFound(req, res, collection, index, headers)`

The function generates an error response.

Result exception

`actions.resultException(req, res, err, headers, verbose)`

The function generates an error response. If @FA{verbose} is set to *true* or not specified (the default), then the error stack trace will be included in the error message if available. If @FA{verbose} is a string it will be prepended before the error message and the stacktrace will also be included.

Module "queries"

The query module provides the infrastructure for working with currently running AQL queries via arangosh.

Properties

`queries.properties()` Returns the servers current query tracking configuration; we change the slow query threshold to get better results:

```
arangosh> var queries = require("@arangodb/aql/queries");
arangosh> queries.properties();
arangosh> queries.properties({slowQueryThreshold: 1});
```

show execution results

Currently running queries

We create a task that spawns queries, so we have nice output. Since this task uses resources, you may want to increase `period` (and not forget to remove it... afterwards):

```
arangosh> var theQuery = 'FOR sleepLoooong IN 1..5 LET sleepLoooonger = SLEEP(1000) RETURN
arangosh> var tasks = require("@arangodb/tasks");
arangosh> tasks.register({
........>  id: "mytask-1",
........>  name: "this is a sample task to spawn a slow aql query",
........>  command: "require('@arangodb').db._query('" + theQuery + "');"
........> });
arangosh> queries.current();
```

show execution results
The function returns the currently running AQL queries as an array.

Slow queries

The function returns the last AQL queries that exceeded the slow query threshold as an array:

```
arangosh> queries.slow();
[ ]
```

Clear slow queries

Clear the list of slow AQL queries:

```
arangosh> queries.clearSlow();
{
  "code" : 200
}
arangosh> queries.slow();
[ ]
```

Kill

Kill a running AQL query:

```
arangosh> var runningQueries = queries.current().filter(function(query) {
........>   return query.query === theQuery;
........> });
arangosh> queries.kill(runningQueries[0].id);
{
  "code" : 200
}
```

Write-ahead log

This module provides functionality for administering the write-ahead logs.

Configuration

retrieves the configuration of the write-ahead log `internal.wal.properties()`

Retrieves the configuration of the write-ahead log. The result is a JSON array with the following attributes:

- *allowOversizeEntries*: whether or not operations that are bigger than a single logfile can be executed and stored
- *logfileSize*: the size of each write-ahead logfile
- *historicLogfiles*: the maximum number of historic logfiles to keep
- *reserveLogfiles*: the maximum number of reserve logfiles that ArangoDB allocates in the background
- *syncInterval*: the interval for automatic synchronization of not-yet synchronized write-ahead log data (in milliseconds)
- *throttleWait*: the maximum wait time that operations will wait before they get aborted if case of write-throttling (in milliseconds)
- *throttleWhenPending*: the number of unprocessed garbage-collection operations that, when reached, will activate write-throttling. A value of *0* means that write-throttling will not be triggered.

Examples

```
arangosh> require("internal").wal.properties();
```

show execution results
configures the write-ahead log `internal.wal.properties(properties)`

Configures the behavior of the write-ahead log. *properties* must be a JSON JSON object with the following attributes:

- *allowOversizeEntries*: whether or not operations that are bigger than a single logfile can be executed and stored
- *logfileSize*: the size of each write-ahead logfile
- *historicLogfiles*: the maximum number of historic logfiles to keep
- *reserveLogfiles*: the maximum number of reserve logfiles that ArangoDB allocates in the background
- *throttleWait*: the maximum wait time that operations will wait before they get aborted if case of write-throttling (in milliseconds)
- *throttleWhenPending*: the number of unprocessed garbage-collection operations that, when reached, will activate write-throttling. A value of *0* means that write-throttling will not be triggered.

Specifying any of the above attributes is optional. Not specified attributes will be ignored and the configuration for them will not be modified.

Examples

```
arangosh> require("internal").wal.properties({
........>    allowOverSizeEntries: true,
........> logfileSize: 32 * 1024 * 1024 });
```

show execution results

Flushing

flushes the currently open WAL logfile `internal.wal.flush(waitForSync, waitForCollector)`

Flushes the write-ahead log. By flushing the currently active write-ahead logfile, the data in it can be transferred to collection journals and datafiles. This is useful to ensure that all data for a collection is present in the collection journals and datafiles, for example, when dumping the data of a collection.

The *waitForSync* option determines whether or not the operation should block until the not-yet synchronized data in the write-ahead log was synchronized to disk.

The *waitForCollector* operation can be used to specify that the operation should block until the data in the flushed log has been collected by the write-ahead log garbage collector. Note that setting this option to *true* might block for a long time if there are long-running transactions and the write-ahead log garbage collector cannot finish garbage collection.

Examples

```
arangosh> require("internal").wal.flush();
```

Task Management

Introduction to Task Management in ArangoDB

ArangoDB can execute user-defined JavaScript functions as one-shot or periodic tasks. This functionality can be used to implement timed or recurring jobs in the database.

Tasks in ArangoDB consist of a JavaScript snippet or function that is executed when the task is scheduled. A task can be a one-shot task (meaning it is run once and not repeated) or a periodic task (meaning that it is re-scheduled after each execution). Tasks can have optional parameters, which are defined at task setup time. The parameters specified at task setup time will be passed as arguments to the task whenever it gets executed. Periodic Tasks have an execution frequency that needs to be specified when the task is set up. One-shot tasks have a configurable delay after which they'll get executed.

Tasks will be executed on the server they have been set up on. Tasks will not be shipped around in a cluster. A task will be executed in the context of the database it was created in. However, when dropping a database, any tasks that were created in the context of this database will remain active. It is therefore sensible to first unregister all active tasks for a database before dropping the database.

Tasks registered in ArangoDB will be executed until the server gets shut down or restarted. After a restart of the server, any user-defined one-shot or periodic tasks will be lost.

Commands for Working with Tasks

ArangoDB provides the following commands for working with tasks. All commands can be accessed via the *tasks* module, which can be loaded like this:

`require("@arangodb/tasks")`

Please note that the *tasks* module is available inside the ArangoDB server only. It cannot be used from the ArangoShell or ArangoDB's web interface.

Register a task

To register a task, the JavaScript snippet or function needs to be specified in addition to the execution frequency. Optionally, a task can have an id and a name. If no id is specified, it will be auto-assigned for a new task. The task id is also the means to access or unregister a task later. Task names are informational only. They can be used to make a task distinguishable from other tasks also running on the server.

The following server-side commands register a task. The command to be executed is a JavaScript string snippet which prints a message to the server's logfile:

```
const tasks = require("@arangodb/tasks");

tasks.register({
  id: "mytask-1",
  name: "this is a snippet task",
  period: 15,
  command: "require('console').log('hello from snippet task');"
});
```

The above has register a task with id *mytask-1*, which will be executed every 15 seconds on the server. The task will write a log message whenever it is invoked.

Tasks can also be set up using a JavaScript callback function like this:

```
const tasks = require("@arangodb/tasks");

tasks.register({
  id: "mytask-2",
  name: "this is a function task",
  period: 15,
  command: function () {
    require('console').log('hello from function task');
  }
});
```

It is important to note that the callback function is late bound and will be executed in a different context than in the creation context. The callback function must therefore not access any variables defined outside of its own scope. The callback function can still define and use its own variables.

To pass parameters to a task, the *params* attribute can be set when registering a task. Note that the parameters are limited to data types usable in JSON (meaning no callback functions can be passed as parameters into a task):

```
const tasks = require("@arangodb/tasks");

tasks.register({
  id: "mytask-3",
  name: "this is a parameter task",
  period: 15,
  command: function (params) {
    var greeting = params.greeting;
    var data = JSON.stringify(params.data);
    require('console').log('%s from parameter task: %s', greeting, data);
  },
  params: { greeting: "hi", data: "how are you?" }
});
```

Registering a one-shot task works the same way, except that the *period* attribute must be omitted. If *period* is omitted, then the task will be executed just once. The task invocation delay can optionally be specified with the *offset* attribute:

```
const tasks = require("@arangodb/tasks");

tasks.register({
  id: "mytask-once",
  name: "this is a one-shot task",
  offset: 10,
  command: function (params) {
    require('console').log('you will see me just once!');
  }
});
```

Note: When specifying an *offset* value of 0, ArangoDB will internally add a very small value to the offset so will be slightly greater than zero.

Unregister a task

After a task has been registered, it can be unregistered using its id:

```
const tasks = require("@arangodb/tasks");
tasks.unregister("mytask-1");
```

Note that unregistering a non-existing task will throw an exception.

List all tasks

To get an overview of which tasks are registered, there is the *get* method. If the *get* method is called without any arguments, it will return an array of all tasks:

```
const tasks = require("@arangodb/tasks");
tasks.get();
```

If *get* is called with a task id argument, it will return information about this particular task:

```
const tasks = require("@arangodb/tasks");
tasks.get("mytask-3");
```

The *created* attribute of a task reveals when a task was created. It is returned as a Unix timestamp.

Deprecated

Simple Queries

Simple queries can be used if the query condition is straight forward, i.e., a document reference, all documents, a query-by-example, or a simple geo query. In a simple query you can specify exactly one collection and one query criteria. In the following sections we describe the JavaScript shell interface for simple queries, which you can use within the ArangoDB shell and within actions and transactions. For other languages see the corresponding language API documentation.

You can find a list of queries at Collection Methods.

Sequential Access and Cursors

If a query returns a cursor, then you can use *hasNext* and *next* to iterate over the result set or *toArray* to convert it to an array.

If the number of query results is expected to be big, it is possible to limit the amount of documents transferred between the server and the client to a specific value. This value is called *batchSize*. The *batchSize* can optionally be set before or when a simple query is executed. If the server has more documents than should be returned in a single batch, the server will set the *hasMore* attribute in the result. It will also return the id of the server-side cursor in the *id* attribute in the result. This id can be used with the cursor API to fetch any outstanding results from the server and dispose the server-side cursor afterwards.

The initial *batchSize* value can be set using the *setBatchSize* method that is available for each type of simple query, or when the simple query is executed using its *execute* method. If no *batchSize* value is specified, the server will pick a reasonable default value.

Has Next

checks if the cursor is exhausted `cursor.hasNext()`

The *hasNext* operator returns *true*, then the cursor still has documents. In this case the next document can be accessed using the *next* operator, which will advance the cursor.

Examples

```
arangosh> var a = db.five.all();
arangosh> while (a.hasNext()) print(a.next());
```

show execution results

Next

returns the next result document `cursor.next()`

If the *hasNext* operator returns *true*, then the underlying cursor of the simple query still has documents. In this case the next document can be accessed using the *next* operator, which will advance the underlying cursor. If you use *next* on an exhausted cursor, then *undefined* is returned.

Examples

```
arangosh> db.five.all().next();
```

show execution results

Set Batch size

sets the batch size for any following requests `cursor.setBatchSize(number)`

Sets the batch size for queries. The batch size determines how many results are at most transferred from the server to the client in one chunk.

Get Batch size

returns the batch size `cursor.getBatchSize()`

Returns the batch size for queries. If the returned value is undefined, the server will determine a sensible batch size for any following requests.

Execute Query

executes a query `query.execute(batchSize)`

Executes a simple query. If the optional batchSize value is specified, the server will return at most batchSize values in one roundtrip. The batchSize cannot be adjusted after the query is first executed.

Note: There is no need to explicitly call the execute method if another means of fetching the query results is chosen. The following two approaches lead to the same result:

```
arangosh> result = db.users.all().toArray();
arangosh> q = db.users.all(); q.execute(); result = [ ]; while (q.hasNext()) { result.push
```

show execution results
The following two alternatives both use a batchSize and return the same result:

```
arangosh> q = db.users.all(); q.setBatchSize(20); q.execute(); while (q.hasNext()) { print
arangosh> q = db.users.all(); q.execute(20); while (q.hasNext()) { print(q.next()); }
```

show execution results

Dispose

disposes the result `cursor.dispose()`

If you are no longer interested in any further results, you should call *dispose* in order to free any resources associated with the cursor. After calling *dispose* you can no longer access the cursor.

Count

counts the number of documents `cursor.count()`

The *count* operator counts the number of document in the result set and returns that number. The *count* operator ignores any limits and returns the total number of documents found.

Note: Not all simple queries support counting. In this case *null* is returned.

`cursor.count(true)`

If the result set was limited by the *limit* operator or documents were skiped using the *skip* operator, the *count* operator with argument *true* will use the number of elements in the final result set - after applying *limit* and *skip*.

Note: Not all simple queries support counting. In this case *null* is returned.

Examples

Ignore any limit:

```
arangosh> db.five.all().limit(2).count();
null
```

Counting any limit or skip:

```
arangosh> db.five.all().limit(2).count(true);
2
```

Pagination

If, for example, you display the result of a user search, then you are in general not interested in the completed result set, but only the first 10 or so documents. Or maybe the next 10 documents for the second page. In this case, you can the *skip* and *limit* operators. These operators work like LIMIT in MySQL.

skip used together with *limit* can be used to implement pagination. The *skip* operator skips over the first n documents. So, in order to create result pages with 10 result documents per page, you can use *skip(n * 10).limit(10)* to access the 10 documents on the *n*th page. This result should be sorted, so that the pagination works in a predicable way.

Limit

limit `query.limit(number)`

Limits a result to the first *number* documents. Specifying a limit of *0* will return no documents at all. If you do not need a limit, just do not add the limit operator. The limit must be non-negative.

In general the input to *limit* should be sorted. Otherwise it will be unclear which documents will be included in the result set.

Examples

```
arangosh> db.five.all().toArray();
arangosh> db.five.all().limit(2).toArray();
```

show execution results

Skip

skip `query.skip(number)`

Skips the first *number* documents. If *number* is positive, then this number of documents are skipped before returning the query results.

In general the input to *skip* should be sorted. Otherwise it will be unclear which documents will be included in the result set.

Note: using negative *skip* values is **deprecated** as of ArangoDB 2.6 and will not be supported in future versions of ArangoDB.

Examples

```
arangosh> db.five.all().toArray();
arangosh> db.five.all().skip(3).toArray();
```

show execution results

Modification Queries

ArangoDB also allows removing, replacing, and updating documents based on an example document. Every document in the collection will be compared against the specified example document and be deleted/replaced/ updated if all attributes match.

These method should be used with caution as they are intended to remove or modify lots of documents in a collection.

All methods can optionally be restricted to a specific number of operations. However, if a limit is specific but is less than the number of matches, it will be undefined which of the matching documents will get removed/modified. Remove by Example, Replace by Example and Update by Example are described with examples in the subchapter Collection Methods.

Geo Queries

The ArangoDB allows to select documents based on geographic coordinates. In order for this to work, a geo-spatial index must be defined. This index will use a very elaborate algorithm to lookup neighbors that is a magnitude faster than a simple R* index.

In general a geo coordinate is a pair of latitude and longitude, which must both be specified as numbers. A geo index can be created on coordinates that are stored in a single list attribute with two elements like *[-10, +30]* (latitude first, followed by longitude) or on coordinates stored in two separate attributes.

For example, to index the following documents, an index can be created on the *position* attribute of the documents:

```
db.test.save({ position: [ -10, 30 ] });
db.test.save({ position: [ 10, 45.5 ] });

db.test.ensureIndex({ type: "geo", fields: [ "position" ] });
```

If coordinates are stored in two distinct attributes, the index must be created on the two attributes:

```
db.test.save({ latitude: -10, longitude: 30 });
db.test.save({ latitude: 10, longitude: 45.5 });

db.test.ensureIndex({ type: "geo", fields: [ "latitude", "longitude" ] });
```

In order to find all documents within a given radius around a coordinate use the *within* operator. In order to find all documents near a given document use the *near* operator.

It is possible to define more than one geo-spatial index per collection. In this case you must give a hint using the *geo* operator which of indexes should be used in a query.

Near

constructs a near query for a collection `collection.near(latitude, longitude)` The returned list is sorted according to the distance, with the nearest document to the coordinate (*latitude*, *longitude*) coming first. If there are near documents of equal distance, documents are chosen randomly from this set until the limit is reached. It is possible to change the limit using the *limit* operator. In order to use the *near* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* operator to select a particular index. *Note*: `near` does not support negative skips. // However, you can still use `limit` followed to skip. `collection.near(latitude, longitude).limit(limit)` Limits the result to limit documents instead of the default 100. *Note*: Unlike with multiple explicit limits, `limit` will raise the implicit default limit imposed by `within`. `collection.near(latitude, longitude).distance()` This will add an attribute `distance` to all documents returned, which contains the distance between the given point and the document in meters. `collection.near(latitude, longitude).distance(name)` This will add an attribute *name* to all documents returned, which contains the distance between the given point and the document in meters. Note: the *near* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for retrieving documents from a collection using the near operator is to use the AQL *NEAR* function in an AQL query as follows:

```
FOR doc IN NEAR(@@collection, @latitude, @longitude, @limit)
    RETURN doc
```

Examples

To get the nearest two locations:

```
arangosh> db.geo.ensureIndex({ type: "geo", fields: [ "loc" ] });
arangosh> for (var i = -90;  i <= 90;  i += 10) {
........>   for (var j = -180; j <= 180; j += 10) {
........>     db.geo.save({
........>        name : "Name/" + i + "/" + j,
........>        loc: [ i, j ] });
........> } }
arangosh> db.geo.near(0, 0).limit(2).toArray();
```

show execution results

If you need the distance as well, then you can use the `distance` operator:

```
arangosh> db.geo.ensureIndex({ type: "geo", fields: [ "loc" ] });
arangosh> for (var i = -90;  i <= 90;  i += 10) {
........>  for (var j = -180; j <= 180; j += 10) {
........>     db.geo.save({
........>         name : "Name/" + i + "/" + j,
........>         loc: [ i, j ] });
........> } }
arangosh> db.geo.near(0, 0).distance().limit(2).toArray();
```

show execution results

Within

constructs a within query for a collection `collection.within(latitude, longitude, radius)` This will find all documents within a given radius around the coordinate (*latitude*, *longitude*). The returned array is sorted by distance, beginning with the nearest document. In order to use the *within* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the `geo` operator to select a particular index. `collection.within(latitude, longitude, radius).distance()` This will add an attribute `_distance` to all documents returned, which contains the distance between the given point and the document in meters. `collection.within(latitude, longitude, radius).distance(name)` This will add an attribute *name* to all documents returned, which contains the distance between the given point and the document in meters. Note: the *within* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for retrieving documents from a collection using the within operator is to use the AQL *WITHIN* function in an AQL query as follows:

```
FOR doc IN WITHIN(@@collection, @latitude, @longitude, @radius, @distanceAttributeName)
    RETURN doc
```

Examples

To find all documents within a radius of 2000 km use:

```
arangosh> for (var i = -90;  i <= 90;  i += 10) {
........>  for (var j = -180; j <= 180; j += 10) {
........> db.geo.save({ name : "Name/" + i + "/" + j, loc: [ i, j ] }); } }
arangosh> db.geo.within(0, 0, 2000 * 1000).distance().toArray();
```

show execution results

Geo

constructs a geo index selection `collection.geo(location-attribute)` Looks up a geo index defined on attribute *location_attribute*. Returns a geo index object if an index was found. The `near` or `within` operators can then be used to execute a geo-spatial query on this particular index. This is useful for collections with multiple defined geo indexes. `collection.geo(location_attribute, true)`

Looks up a geo index on a compound attribute *location_attribute*. Returns a geo index object if an index was found. The `near` or `within` operators can then be used to execute a geo-spatial query on this particular index. `collection.geo(latitude_attribute, longitude_attribute)` Looks up a geo index defined on the two attributes *latitude_attribute* and *longitude-attribute*. Returns a geo index object if an index was found. The `near` or `within` operators can then be used to execute a geo-spatial query on this particular index. Note: the *geo* simple query helper function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for running geo queries is to use their AQL equivalents.

Examples

Assume you have a location stored as list in the attribute *home* and a destination stored in the attribute *work*. Then you can use the `geo` operator to select which geo-spatial attributes (and thus which index) to use in a `near` query.

```
arangosh> for (i = -90;  i <= 90;  i += 10) {
........>  for (j = -180;  j <= 180;  j += 10) {
........>    db.complex.save({ name : "Name/" + i + "/" + j,
........>                      home : [ i, j ],
........>                      work : [ -i, -j ] });
........>  }
........> }
........>
arangosh> db.complex.near(0, 170).limit(5);
arangosh> db.complex.ensureIndex({ type: "geo", fields: [ "home" ] });
arangosh> db.complex.near(0, 170).limit(5).toArray();
arangosh> db.complex.geo("work").near(0, 170).limit(5);
arangosh> db.complex.ensureIndex({ type: "geo", fields: [ "work" ] });
arangosh> db.complex.geo("work").near(0, 170).limit(5).toArray();
```

show execution results

Related topics

Other ArangoDB geographic features are described in:

- AQL Geo functions
- Geo indexes

Fulltext queries

ArangoDB allows to run queries on text contained in document attributes. To use this, a fulltext index must be defined for the attribute of the collection that contains the text. Creating the index will parse the text in the specified attribute for all documents of the collection. Only documents will be indexed that contain a textual value in the indexed attribute. For such documents, the text value will be parsed, and the individual words will be inserted into the fulltext index.

When a fulltext index exists, it can be queried using a fulltext query.

Fulltext

queries the fulltext index `collection.fulltext(attribute, query)`

The *fulltext* simple query functions performs a fulltext search on the specified *attribute* and the specified *query*.

Details about the fulltext query syntax can be found below.

Note: the *fulltext* simple query function is **deprecated** as of ArangoDB 2.6. The function may be removed in future versions of ArangoDB. The preferred way for executing fulltext queries is to use an AQL query using the *FULLTEXT* AQL function as follows:

```
FOR doc IN FULLTEXT(@@collection, @attributeName, @queryString, @limit)
  RETURN doc
```

Examples

```
arangosh> db.emails.ensureFulltextIndex("content");
arangosh> db.emails.save({ content:
........> "Hello Alice, how are you doing? Regards, Bob"});
arangosh> db.emails.save({ content:
........> "Hello Charlie, do Alice and Bob know about it?"});
arangosh> db.emails.save({ content: "I think they don't know. Regards, Eve" });
arangosh> db.emails.fulltext("content", "charlie,|eve").toArray();
```

show execution results

Fulltext Syntax:

In the simplest form, a fulltext query contains just the sought word. If multiple search words are given in a query, they should be separated by commas. All search words will be combined with a logical AND by default, and only such documents will be returned that contain all search words. This default behavior can be changed by providing the extra control characters in the fulltext query, which are:

- +: logical AND (intersection)
- |: logical OR (union)
- -: negation (exclusion)

Examples:

- *"banana"*: searches for documents containing "banana"
- *"banana,apple"*: searches for documents containing both "banana" *AND* "apple"
- *"banana,|orange"*: searches for documents containing either "banana" *OR* "orange" *OR* both
- *"banana,-apple"*: searches for documents that contains "banana" but *NOT* "apple".

Logical operators are evaluated from left to right.

Each search word can optionally be prefixed with *complete*: or *prefix*:, with *complete*: being the default. This allows searching for complete words or for word prefixes. Suffix searches or any other forms are partial-word matching are currently not supported.

Examples:

- *"complete:banana"*: searches for documents containing the exact word "banana"

- *"prefix:head"*: searches for documents with words that start with prefix "head"
- *"prefix:head,banana"*: searches for documents contain words starting with prefix "head" and that also contain the exact word "banana".

Complete match and prefix search options can be combined with the logical operators.

Please note that only words with a minimum length will get indexed. This minimum length can be defined when creating the fulltext index. For words tokenization, the libicu text boundary analysis is used, which takes into account the default as defined at server startup (*--server.default-language* startup option). Generally, the word boundary analysis will filter out punctuation but will not do much more.

Especially no word normalization, stemming, or similarity analysis will be performed when indexing or searching. If any of these features is required, it is suggested that the user does the text normalization on the client side, and provides for each document an extra attribute containing just a comma-separated list of normalized words. This attribute can then be indexed with a fulltext index, and the user can send fulltext queries for this index, with the fulltext queries also containing the stemmed or normalized versions of words as required by the user.

ArangoDB's Actions

Introduction to User Actions

In some ways the communication layer of the ArangoDB server behaves like a Web server. Unlike a Web server, it normally responds to HTTP requests by delivering JSON objects. Remember, documents in the database are just JSON objects. So, most of the time the HTTP response will contain a JSON document from the database as body. You can extract the documents stored in the database using HTTP *GET*. You can store documents using HTTP *POST*.

However, there is something more. You can write small snippets - so called actions - to extend the database. The idea of actions is that sometimes it is better to store parts of the business logic within ArangoDB.

The simplest example is the age of a person. Assume you store information about people in your database. It is an anti-pattern to store the age, because it changes every now and then. Therefore, you normally store the birthday and let the client decide what to do with it. However, if you have many different clients, it might be easier to enrich the person document with the age using actions once on the server side.

Or, for instance, if you want to apply some statistics to large data-sets and you cannot easily express this as query. You can define a action instead of transferring the whole data to the client and do the computation on the client.

Actions are also useful if you want to restrict and filter data according to some complex permission system.

The ArangoDB server can deliver all kinds of information, JSON being only one possible format. You can also generate HTML or images. However, a Web server is normally better suited for the task as it also implements various caching strategies, language selection, compression and so on. Having said that, there are still situations where it might be suitable to use the ArangoDB to deliver HTML pages - static or dynamic. A simple example is the built-in administration interface. You can access it using any modern browser and there is no need for a separate Apache or IIS.

In general you will use Foxx to easily extend the database with business logic. Foxx provides an simple to use interface to actions.

The following sections will explain the low-level actions within ArangoDB on which Foxx is built and show how to define them. The examples start with delivering static HTML pages - even if this is not the primary use-case for actions. The later sections will then show you how to code some pieces of your business logic and return JSON objects.

The interface is loosely modeled after the JavaScript classes for HTTP request and responses found in node.js and the middleware/routing aspects of connect.js and express.js.

Note that unlike node.js, ArangoDB is multi-threaded and there is no easy way to share state between queries inside the JavaScript engine. If such state information is required, you need to use the database itself.

A Hello World Example

The client API or browser sends a HTTP request to the ArangoDB server and the server returns a HTTP response to the client. A HTTP request consists of a method, normally *GET* or *POST* when using a browser, and a request path like */hello/world*. For a real Web server there are a zillion of other thing to consider, we will ignore this for the moment. The HTTP response contains a content type, describing how to interpret the returned data, and the data itself.

In the following example, we want to define an action in ArangoDB, so that the server returns the HTML document

```
<html>
  <body>
   Hello World
  </body>
</html>
```

if asked *GET /hello/world*.

The server needs to know what function to call or what document to deliver if it receives a request. This is called routing. All the routing information of ArangoDB is stored in a collection *_routing*. Each entry in this collections describes how to deal with a particular request path.

For the above example, add the following document to the _routing collection:

```
arangosh> db._routing.save({
........>  url: {
........>    match: "/hello/world"
........>  },
........>  content: {
........>    contentType: "text/html",
........>    body: "<html><body>Hello World</body></html>"
........>  }
........> });
```

show execution results
In order to activate the new routing, you must either restart the server or call the internal reload function.

```
arangosh> require("internal").reloadRouting()
```

Now use the browser and access http:// localhost:8529/hello/world

You should see the *Hello World* in our browser:

```
shell> curl --dump - http://localhost:8529/hello/world

HTTP/1.1 200 OK
content-type: text/html

"Hello World"
```

Matching a URL

There are a lot of options for the *url* attribute. If you define different routing for the same path, then the following simple rule is applied in order to determine which match wins: If there are two matches, then the more specific wins. I. e, if there is a wildcard match and an exact match, the exact match is preferred. If there is a short and a long match, the longer match wins.

Exact Match

If the definition is

```
{
  url: {
    match: "/hello/world"
  }
}
```

then the match must be exact. Only the request for */hello/world* will match, everything else, e. g. */hello/world/my* or */hello/world2*, will not match.

The following definition is a short-cut for an exact match.

```
{
  url: "/hello/world"
}
```

Note: While the two definitions will result in the same URL matching, there is a subtle difference between them:

The former definition (defining *url* as an object with a *match* attribute) will result in the URL being accessible via all supported HTTP methods (e.g. *GET*, *POST*, *PUT*, *DELETE*, ...), whereas the latter definition (providing a string *url* attribute) will result in the URL being accessible via HTTP *GET* and HTTP *HEAD* only, with all other HTTP methods being disabled. Calling a URL with an unsupported or disabled HTTP method will result in an HTTP 501 (not implemented) error.

Prefix Match

If the definition is

```
{
  url: {
    match: "/hello/world/*"
  }
}
```

then the match can be a prefix match. The requests for */hello/world*, */hello/world/my*, and */hello/world/how/are/you* will all match. However */hello/world2* does not match. Prefix matches within a URL part, i. e. */hello/world**, are not allowed. The wildcard must occur at the end, i. e.

```
/hello/*/world
```

is also disallowed.

If you define two routes

```
{ url: { match: "/hello/world/*" } }
{ url: { match: "/hello/world/emil" } }
```

then the second route will be used for */hello/world/emil* because it is more specific.

Parameterized Match

A parameterized match is similar to a prefix match, but the parameters are also allowed inside the URL path.

If the definition is

```
{
  url: {
    match: "/hello/:name/world"
  }
}
```

then the URL must have three parts, the first part being *hello* and the third part *world*. For example, */hello/emil/world* will match, while */hello/emil/meyer/world* will not.

Constraint Match

A constraint match is similar to a parameterized match, but the parameters can carry constraints.

If the definition is

```
{
  url: {
    match: "/hello/:name/world",
    constraint: {
      name: "/[a-z]+/"
    }
  }
}
```

then the URL must have three parts, the first part being *hello* and the third part *world*. The second part must be all lowercase.

It is possible to use more then one constraint for the same URL part.

```
{
  url: {
    match: "/hello/:name|:id/world",
    constraint: {
      name: "/[a-z]+/", id: "/[0-9]+/"
    }
  }
}
```

Optional Match

An optional match is similar to a parameterized match, but the last parameter is optional.

If the definition is

```
{
  url: {
    match: "/hello/:name?",
    constraint: {
      name: "/[a-z]+/"
    }
  }
}
```

then the URL */hello* and */hello/emil* will match.

If the definitions are

```
{ url: { match: "/hello/world" } }
{ url: { match: "/hello/:name", constraint: { name: "/[a-z]+/" } } }
{ url: { match: "/hello/*" } }
```

then the URL */hello/world* will be matched by the first route, because it is the most specific. The URL */hello/you* will be matched by the second route, because it is more specific than the prefix match.

Method Restriction

You can restrict the match to specific HTTP methods.

If the definition is

```
{
  url: {
    match: "/hello/world",
    methods: [ "post", "put" ]
  }
}
```

then only HTTP *POST* and *PUT* requests will match. Calling with a different HTTP method will result in an HTTP 501 error.

Please note that if *url* is defined as a simple string, then only the HTTP methods *GET* and *HEAD* will be allowed, an all other methods will be disabled:

```
{
  url: "/hello/world"
}
```

More on Matching

Remember that the more specific match wins.

- A match without parameter or wildcard is more specific than a match with parameters or wildcard.
- A match with parameter is more specific than a match with a wildcard.
- If there is more than one parameter, specificity is applied from left to right.

Consider the following definitions

```
arangosh> db._routing.save({
........>  url: { match: "/hello/world" },
........> content: { contentType: "text/plain", body: "Match No 1"} });
arangosh> db._routing.save({
........>  url: { match: "/hello/:name", constraint: { name: "/[a-z]+/" } },
........> content: { contentType: "text/plain", body: "Match No 2"} });
arangosh> db._routing.save({
........>  url: { match: "/:something/world" },
........> content: { contentType: "text/plain", body: "Match No 3"} });
arangosh> db._routing.save({
........>  url: { match: "/hi/*" },
........> content: { contentType: "text/plain", body: "Match No 4"} });
arangosh> require("internal").reloadRouting()
```

show execution results

Then

```
shell> curl --dump - http://localhost:8529/hello/world

HTTP/1.1 200 OK
content-type: text/plain

"Match No 1"
shell> curl --dump - http://localhost:8529/hello/emil

HTTP/1.1 200 OK
content-type: text/plain

"Match No 2"
shell> curl --dump - http://localhost:8529/your/world

HTTP/1.1 200 OK
content-type: text/plain

"Match No 3"
shell> curl --dump - http://localhost:8529/hi/you

HTTP/1.1 200 OK
content-type: text/plain

"Match No 4"
```

You can write the following document into the *_routing* collection to test the above examples.

```
{
  routes: [
{ url: { match: "/hello/world" }, content: "route 1" },
{ url: { match: "/hello/:name|:id", constraint: { name: "/[a-z]+/", id: "/[0-9]+/" } }, content: "route 2" },
{ url: { match: "/:something/world" }, content: "route 3" },
{ url: { match: "/hello/*" }, content: "route 4" },
  ]
}
```

A Hello World Example for JSON

If you change the example slightly, then a JSON object will be delivered.

```
arangosh> db._routing.save({
........>  url: "/hello/json",
........>  content: {
........>  contentType: "application/json",
........>    body: '{"hello" : "world"}'
........>  }
........> });
arangosh> require("internal").reloadRouting()
```

show execution results
Again check with your browser or cURL http://localhost:8529/hello/json

Depending on your browser and installed add-ons you will either see the JSON object or a download dialog. If your browser wants to open an external application to display the JSON object, you can change the *contentType* to *"text/plain"* for the example. This makes it easier to check the example using a browser. Or use *curl* to access the server.

```
shell> curl --dump - http://localhost:8529/hello/json

HTTP/1.1 200 OK
content-type: application/json

{
  "hello" : "world"
}
```

Delivering Content

There are a lot of different ways on how to deliver content. We have already seen the simplest one, where static content is delivered. The fun, however, starts when delivering dynamic content.

Static Content

You can specify a body and a content-type.

```
arangosh> db._routing.save({
........>  url: "/hello/contentType",
........>  content: {
........>    contentType: "text/html",
........>    body: "<html><body>Hello World</body></html>"
........>  }
........> });
arangosh> require("internal").reloadRouting()
```

show execution results

```
shell> curl --dump - http://localhost:8529/hello/contentType

HTTP/1.1 200 OK
content-type: text/html

"Hello World"
```

If the content type is *text/plain* then you can use the short-cut

```
{
  content: "Hello World"
}
```

A Simple Action

The simplest dynamic action is:

```
{
  action: {
    do: "@arangodb/actions/echoRequest"
  }
}
```

It is not advisable to store functions directly in the routing table. It is better to call functions defined in modules. In the above example the function can be accessed from JavaScript as:

```
require("@arangodb/actions").echoRequest
```

The function *echoRequest* is pre-defined. It takes the request objects and echos it in the response.

The signature of such a function must be

```
function (req, res, options, next)
```

Examples

```
arangosh> db._routing.save({
........>    url: "/hello/echo",
........>    action: {
........>    do: "@arangodb/actions/echoRequest"
........>  }
........> });
```

show execution results
Reload the routing and check http:// 127.0.0.1:8529/hello/echo

You should see something like

```
arangosh> arango.GET("/hello/echo")
```

show execution results

The request might contain *path*, *prefix*, *suffix*, and *urlParameters* attributes. *path* is the complete path as supplied by the user and always available. If a prefix was matched, then this prefix is stored in the attribute *prefix* and the remaining URL parts are stored as an array in *suffix*. If one or more parameters were matched, then the parameter values are stored in *urlParameters*.

For example, if the url description is

```
{
  url: {
    match: "/hello/:name/:action"
  }
}
```

and you request the path */hello/emil/jump*, then the request object will contain the following attribute

```
urlParameters: {
  name: "emil",
  action: "jump"
}
```

Action Controller

As an alternative to the simple action, you can use controllers. A controller is a module, defines the function *get*, *put*, *post*, *delete*, *head*, *patch*. If a request of the corresponding type is matched, the function will be called.

Examples

```
arangosh> db._routing.save({
........>  url: "/hello/echo",
........>  action: {
........>    controller: "@arangodb/actions/echoController"
........>  }
........> });
```

show execution results
Reload the routing and check http:// 127.0.0.1:8529/hello/echo:

```
arangosh> arango.GET("/hello/echo")
```

show execution results

```

```

Prefix Action Controller

The controller is selected when the definition is read. There is a more flexible, but slower and maybe insecure variant, the prefix controller.

Assume that the url is a prefix match

```
{
  url: {
    match: /hello/*"
  }
}
```

You can use

```
{
  action: {
    prefixController: "@arangodb/actions"
  }
}
```

to define a prefix controller. If the URL */hello/echoController* is given, then the module *@arangodb/actions/echoController* is used.

If you use a prefix controller, you should make certain that no unwanted actions are available under the prefix.

The definition

```
{
  action: "@arangodb/actions"
}
```

is a short-cut for a prefix controller definition.

Function Action

You can also store a function directly in the routing table.

Examples

```
arangosh> db._routing.save({
........>  url: "/hello/echo",
........>  action: {
........>    callback: "function(req,res) {res.statusCode=200; res.body='Hello'}"
........>  }
........> });
```

show execution results

```
arangosh> arango.GET("hello/echo")
arangosh> db._query("FOR route IN _routing FILTER route.url == '/hello/echo' REMOVE route
[object ArangoQueryCursor, count: 0, hasMore: false]
arangosh> require("internal").reloadRouting()
```

Requests and Responses

The controller must define handler functions which take a request object and fill the response object.

A very simple example is the function *echoRequest* defined in the module *@arangodb/actions*.

```
function (req, res, options, next) {
  var result;

  result = { request: req, options: options };

  res.responseCode = exports.HTTP_OK;
  res.contentType = "application/json";
  res.body = JSON.stringify(result);
}
```

Install it via:

```
arangosh> db._routing.save({
........>  url: "/echo",
........>  action: {
........>    do: "@arangodb/actions/echoRequest"
........>  }
........> })
```

show execution results

Reload the routing and check http:// 127.0.0.1:8529/hello/echo

You should see something like

```
arangosh> arango.GET("/hello/echo")
arangosh> db._query("FOR route IN _routing FILTER route.url == '/hello/echo' REMOVE route
arangosh> require("internal").reloadRouting()
```

show execution results

You may also pass options to the called function:

```
arangosh> db._routing.save({
........>  url: "/echo",
........>  action: {
........>    do: "@arangodb/actions/echoRequest",
........>    options: {
........>      "Hello": "World"
........>    }
........>  }
........> });
```

show execution results

You now see the options in the result:

```
arangosh> arango.GET("/echo")
arangosh> db._query("FOR route IN _routing FILTER route.url == '/echo' REMOVE route in _ro
arangosh> require("internal").reloadRouting()
```

show execution results

Modifying Request and Response

As we've seen in the previous examples, actions get called with the request and response objects (named *req* and *res* in the examples) passed as parameters to their handler functions.

The *req* object contains the incoming HTTP request, which might or might not have been modified by a previous action (if actions were chained).

A handler can modify the request object in place if desired. This might be useful when writing middleware (see below) that is used to intercept incoming requests, modify them and pass them to the actual handlers.

While modifying the request object might not be that relevant for non-middleware actions, modifying the response object definitely is. Modifying the response object is an action's only way to return data to the caller of the action.

We've already seen how to set the HTTP status code, the content type, and the result body. The *res* object has the following properties for these:

- *contentType*: MIME type of the body as defined in the HTTP standard (e.g. *text/html*, *text/plain*, *application/json*, ...)
- *responsecode*: the HTTP status code of the response as defined in the HTTP standard. Common values for actions that succeed are *200* or *201*. Please refer to the HTTP standard for more information.
- *body*: the actual response data

To set or modify arbitrary headers of the response object, the *headers* property can be used. For example, to add a user-defined header to the response, the following code will do:

```
res.headers = res.headers || { }; // headers might or might not be present
res.headers['X-Test'] = 'someValue'; // set header X-Test to "someValue"
```

This will set the additional HTTP header *X-Test* to value *someValue*. Other headers can be set as well. Note that ArangoDB might change the case of the header names to lower case when assembling the overall response that is sent to the caller.

It is not necessary to explicitly set a *Content-Length* header for the response as ArangoDB will calculate the content length automatically and add this header itself. ArangoDB might also add a *Connection* header itself to handle HTTP keep-alive.

ArangoDB also supports automatic transformation of the body data to another format. Currently, the only supported transformations are base64-encoding and base64-decoding. Using the transformations, an action can create a base64 encoded body and still let ArangoDB send the non-encoded version, for example:

```
res.body = 'VGhpcyBpcyBhIHRlc3Q=';
res.transformations = res.transformations || [ ]; // initialize
res.transformations.push('base64decode'); // will base64 decode the response body
```

When ArangoDB processes the response, it will base64-decode what's in *res.body* and set the HTTP header *Content-Encoding: binary*. The opposite can be achieved with the *base64encode* transformation: ArangoDB will then automatically base64-encode the body and set a *Content-Encoding: base64* HTTP header.

Writing dynamic action handlers

To write your own dynamic action handlers, you must put them into modules.

Modules are a means of organizing action handlers and making them loadable under specific names.

To start, we'll define a simple action handler in a module */ownTest*:

```
arangosh> db._modules.save({
........>  path: "/db:/ownTest",
........>  content:
........>     "exports.do = function(req, res, options, next) {"+
........>     "  res.body = 'test';" +
........>     "  res.responseCode = 200;" +
........>     "  res.contentType = 'text/plain';" +
........>     "};"
........> });
```

show execution results

This does nothing but register a do action handler in a module */ownTest*. The action handler is not yet callable, but must be mapped to a route first. To map the action to the route */ourtest*, execute the following command:

```
arangosh> db._routing.save({
........>  url: "/ourtest",
........>  action: {
........>    controller: "db://ownTest"
........>  }
........> });
arangosh> require("internal").reloadRouting()
```

show execution results

Now use the browser or cURL and access http://localhost:8529/ourtest :

```
shell> curl --dump - http://localhost:8529/ourtest

HTTP/1.1 200 OK
content-type: text/plain

"test"
```

You will see that the module's do function has been executed.

A Word about Caching

Sometimes it might seem that your change do not take effect. In this case the culprit could be the routing caches:

The routing cache stores the routing information computed from the *_routing* collection. Whenever you change this collection manually, you need to call

```
arangosh> require("internal").reloadRouting()
```

in order to rebuild the cache.

Advanced Usages

For detailed information see the reference manual.

Redirects

Use the following for a permanent redirect:

```
arangosh> db._routing.save({
........>  url: "/redirectMe",
........>  action: {
........>    do: "@arangodb/actions/redirectRequest",
........>    options: {
........>      permanently: true,
........>      destination: "/somewhere.else/"
........>    }
........>  }
........> });
arangosh> require("internal").reloadRouting()
```

show execution results

```
shell> curl --dump - http://localhost:8529/redirectMe

HTTP/1.1 301 Moved Permanently
content-type: text/html
location: /somewhere.else/

"<html><head><title>Moved</title></head><body><h1>Moved</h1><p>This page has moved to <a h
```

Routing Bundles

Instead of adding all routes for package separately, you can specify a bundle:

```
arangosh> db._routing.save({
........>  routes: [
........>    {
........>      url: "/url1",
........>      content: "route 1"
........>    },
........>    {
........>      url: "/url2",
........>      content: "route 2"
........>    },
........>    {
........>      url: "/url3",
........>      content: "route 3"
........>    }
........>  ]
........> });
arangosh> require("internal").reloadRouting()
```

show execution results

```
shell> curl --dump - http://localhost:8529/url2

HTTP/1.1 200 OK
content-type: text/plain

"route 2"
shell> curl --dump - http://localhost:8529/url3

HTTP/1.1 200 OK
content-type: text/plain

"route 3"
```

The advantage is, that you can put all your routes into one document and use a common prefix.

```
arangosh> db._routing.save({
........>  urlPrefix: "/test",
........>  routes: [
........>    {
........>      url: "/url1",
........>      content: "route 1"
........>    },
........>    {
........>      url: "/url2",
........>      content: "route 2"
........>    },
........>    {
........>      url: "/url3",
........>      content: "route 3"
........>    }
........>  ]
........> });
arangosh> require("internal").reloadRouting()
```

show execution results
will define the URL */test/url1*, */test/url2*, and */test/url3*:

```
shell> curl --dump - http://localhost:8529/test/url1

HTTP/1.1 200 OK
content-type: text/plain

"route 1"
shell> curl --dump - http://localhost:8529/test/url2

HTTP/1.1 200 OK
content-type: text/plain

"route 2"
shell> curl --dump - http://localhost:8529/test/url3

HTTP/1.1 200 OK
content-type: text/plain

"route 3"
```

Writing Middleware

Assume, you want to log every request in your namespace to the console. *(if ArangoDB is running as a daemon, this will end up in the logfile)*. In this case you can easily define an action for the URL */subdirectory*. This action simply logs the requests, calls the next in line, and logs the response:

```
arangosh> db._modules.save({
........>  path: "/db:/OwnMiddlewareTest",
........>  content:
........>     "exports.logRequest = function (req, res, options, next) {" +
........>     "    console = require('console'); " +
........>     "    console.log('received request: %s', JSON.stringify(req));" +
........>     "    next();" +
........>     "    console.log('produced response: %s', JSON.stringify(res));" +
........>     "};"
........> });
```

show execution results
This function will now be available as *db://OwnMiddlewareTest/logRequest*. You need to tell ArangoDB that it is should use a prefix match and that the shortest match should win in this case:

```
arangosh> db._routing.save({
........>  middleware: [
........>    {
........>      url: {
........>        match: "/subdirectory/*"
........>      },
........>      action: {
........>        do: "db://OwnMiddlewareTest/logRequest"
........>      }
........>    }
........>  ]
........> });
```

show execution results

When you call *next()* in that action, the next specific routing will be used for the original URL. Even if you modify the URL in the request object *req*, this will not cause the *next()* to jump to the routing defined for this next URL. If proceeds occurring the origin URL. However, if you use *next(true)*, the routing will stop and request handling is started with the new URL. You must ensure that *next(true)* is never called without modifying the URL in the request object *req*. Otherwise an endless loop will occur.

Now we add some other simple routings to test all this:

```
arangosh> db._routing.save({
........>    url: "/subdirectory/ourtest/1",
........>    action: {
........>      do: "@arangodb/actions/echoRequest"
........>    }
........> });
arangosh> db._routing.save({
........>    url: "/subdirectory/ourtest/2",
........>    action: {
........>      do: "@arangodb/actions/echoRequest"
........>    }
........> });
arangosh> db._routing.save({
........>    url: "/subdirectory/ourtest/3",
........>    action: {
........>      do: "@arangodb/actions/echoRequest"
........>    }
........> });
arangosh> require("internal").reloadRouting()
```

show execution results
Then we send some curl requests to these sample routes:

```
shell> curl --dump - http://localhost:8529/subdirectory/ourtest/1

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
```

show response body
and the console (and / or the logfile) will show requests and replies. *Note that logging doesn't warrant the sequence in which these lines will appear.*

Application Deployment

Using single routes or bundles can be become a bit messy in large applications. Kaerus has written a deployment tool in node.js.

Note that there is also Foxx for building applications with ArangoDB.

Common Pitfalls when using Actions

Caching

If you made any changes to the routing but the changes does not have any effect when calling the modified actions URL, you might have been hit by some caching issues.

After any modification to the routing or actions, it is thus recommended to make the changes "live" by calling the following functions from within arangosh:

You might also be affected by client-side caching. Browsers tend to cache content and also redirection URLs. You might need to clear or disable the browser cache in some cases to see your changes in effect.

Data types

When processing the request data in an action, please be aware that the data type of all query parameters is *string*. This is because the whole URL is a string and when the individual parts are extracted, they will also be strings.

For example, when calling the URL http:// localhost:8529/hello/world?value=5

the parameter *value* will have a value of (string) *5*, not (number) *5*. This might be troublesome if you use JavaScript's === operator when checking request parameter values.

The same problem occurs with incoming HTTP headers. When sending the following header from a client to ArangoDB

```
X-My-Value: 5
```

then the header *X-My-Value* will have a value of (string) *5* and not (number) *5*.

404 Not Found

If you defined a URL in the routing and the URL is accessible fine via HTTP *GET* but returns an HTTP 501 (not implemented) for other HTTP methods such as *POST*, *PUT* or *DELETE*, then you might have been hit by some defaults.

By default, URLs defined like this (simple string *url* attribute) are accessible via HTTP *GET* and *HEAD* only. To make such URLs accessible via other HTTP methods, extend the URL definition with the *methods* attribute.

For example, this definition only allows access via *GET* and *HEAD*:

```
{
  url: "/hello/world"
}
```

whereas this definition allows HTTP *GET*, *POST*, and *PUT*:

```
arangosh> db._routing.save({
........>     url: {
........>       match: "/hello/world",
........>       methods: [ "get", "post", "put" ]
........>     },
........>     action: {
........>       do: "@arangodb/actions/echoRequest"
........>     }
........> });
arangosh> require("internal").reloadRouting()
```

show execution results

```
shell> curl --dump - http://localhost:8529/hello/world

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
```

show response body

The former definition (defining *url* as an object with a *match* attribute) will result in the URL being accessible via all supported HTTP methods (e.g. *GET*, *POST*, *PUT*, *DELETE*, ...), whereas the latter definition (providing a string *url* attribute) will result in the URL being accessible via HTTP *GET* and HTTP *HEAD* only, with all other HTTP methods being disabled. Calling a URL with an unsupported or disabled HTTP method will result in an HTTP 404 error.

Error codes and meanings

General errors

- **0 - ERROR_NO_ERROR**
  No error has occurred.
- **1 - ERROR_FAILED**
  Will be raised when a general error occurred.
- **2 - ERROR_SYS_ERROR**
  Will be raised when operating system error occurred.
- **3 - ERROR_OUT_OF_MEMORY**
  Will be raised when there is a memory shortage.
- **4 - ERROR_INTERNAL**
  Will be raised when an internal error occurred.
- **5 - ERROR_ILLEGAL_NUMBER**
  Will be raised when an illegal representation of a number was given.
- **6 - ERROR_NUMERIC_OVERFLOW**
  Will be raised when a numeric overflow occurred.
- **7 - ERROR_ILLEGAL_OPTION**
  Will be raised when an unknown option was supplied by the user.
- **8 - ERROR_DEAD_PID**
  Will be raised when a PID without a living process was found.
- **9 - ERROR_NOT_IMPLEMENTED**
  Will be raised when hitting an unimplemented feature.
- **10 - ERROR_BAD_PARAMETER**
  Will be raised when the parameter does not fulfill the requirements.
- **11 - ERROR_FORBIDDEN**
  Will be raised when you are missing permission for the operation.
- **12 - ERROR_OUT_OF_MEMORY_MMAP**
  Will be raised when there is a memory shortage.
- **13 - ERROR_CORRUPTED_CSV**
  Will be raised when encountering a corrupt csv line.
- **14 - ERROR_FILE_NOT_FOUND**
  Will be raised when a file is not found.
- **15 - ERROR_CANNOT_WRITE_FILE**
  Will be raised when a file cannot be written.
- **16 - ERROR_CANNOT_OVERWRITE_FILE**
  Will be raised when an attempt is made to overwrite an existing file.
- **17 - ERROR_TYPE_ERROR**
  Will be raised when a type error is unencountered.
- **18 - ERROR_LOCK_TIMEOUT**
  Will be raised when there's a timeout waiting for a lock.
- **19 - ERROR_CANNOT_CREATE_DIRECTORY**
  Will be raised when an attempt to create a directory fails.

- **20 - ERROR_CANNOT_CREATE_TEMP_FILE**
  Will be raised when an attempt to create a temporary file fails.
- **21 - ERROR_REQUEST_CANCELED**
  Will be raised when a request is canceled by the user.
- **22 - ERROR_DEBUG**
  Will be raised intentionally during debugging.
- **25 - ERROR_IP_ADDRESS_INVALID**
  Will be raised when the structure of an IP address is invalid.
- **27 - ERROR_FILE_EXISTS**
  Will be raised when a file already exists.
- **28 - ERROR_LOCKED**
  Will be raised when a resource or an operation is locked.
- **29 - ERROR_DEADLOCK**
  Will be raised when a deadlock is detected when accessing collections.
- **30 - ERROR_SHUTTING_DOWN**
  Will be raised when a call cannot succeed because a server shutdown is already in progress.
- **31 - ERROR_ONLY_ENTERPRISE**
  Will be raised when an enterprise-feature is requested from the community edition.

HTTP error status codes

- **400 - ERROR_HTTP_BAD_PARAMETER**
  Will be raised when the HTTP request does not fulfill the requirements.
- **401 - ERROR_HTTP_UNAUTHORIZED**
  Will be raised when authorization is required but the user is not authorized.
- **403 - ERROR_HTTP_FORBIDDEN**
  Will be raised when the operation is forbidden.
- **404 - ERROR_HTTP_NOT_FOUND**
  Will be raised when an URI is unknown.
- **405 - ERROR_HTTP_METHOD_NOT_ALLOWED**
  Will be raised when an unsupported HTTP method is used for an operation.
- **412 - ERROR_HTTP_PRECONDITION_FAILED**
  Will be raised when a precondition for an HTTP request is not met.
- **500 - ERROR_HTTP_SERVER_ERROR**
  Will be raised when an internal server is encountered.

HTTP processing errors

- **600 - ERROR_HTTP_CORRUPTED_JSON**
  Will be raised when a string representation of a JSON object is corrupt.
- **601 - ERROR_HTTP_SUPERFLUOUS_SUFFICES**
  Will be raised when the URL contains superfluous suffices.

Internal ArangoDB storage errors

For errors that occur because of a programming error.

- **1000 - ERROR_ARANGO_ILLEGAL_STATE**
  Internal error that will be raised when the datafile is not in the required state.
- **1002 - ERROR_ARANGO_DATAFILE_SEALED**
  Internal error that will be raised when trying to write to a datafile.
- **1004 - ERROR_ARANGO_READ_ONLY**
  Internal error that will be raised when trying to write to a read-only datafile or collection.
- **1005 - ERROR_ARANGO_DUPLICATE_IDENTIFIER**
  Internal error that will be raised when a identifier duplicate is detected.
- **1006 - ERROR_ARANGO_DATAFILE_UNREADABLE**
  Internal error that will be raised when a datafile is unreadable.
- **1007 - ERROR_ARANGO_DATAFILE_EMPTY**
  Internal error that will be raised when a datafile is empty.
- **1008 - ERROR_ARANGO_RECOVERY**
  Will be raised when an error occurred during WAL log file recovery.
- **1009 - ERROR_ARANGO_DATAFILE_STATISTICS_NOT_FOUND**
  Will be raised when a required datafile statistics object was not found.

External ArangoDB storage errors

For errors that occur because of an outside event.

- **1100 - ERROR_ARANGO_CORRUPTED_DATAFILE**
  Will be raised when a corruption is detected in a datafile.
- **1101 - ERROR_ARANGO_ILLEGAL_PARAMETER_FILE**
  Will be raised if a parameter file is corrupted or cannot be read.
- **1102 - ERROR_ARANGO_CORRUPTED_COLLECTION**
  Will be raised when a collection contains one or more corrupted data files.
- **1103 - ERROR_ARANGO_MMAP_FAILED**
  Will be raised when the system call mmap failed.
- **1104 - ERROR_ARANGO_FILESYSTEM_FULL**
  Will be raised when the filesystem is full.
- **1105 - ERROR_ARANGO_NO_JOURNAL**
  Will be raised when a journal cannot be created.
- **1106 - ERROR_ARANGO_DATAFILE_ALREADY_EXISTS**
  Will be raised when the datafile cannot be created or renamed because a file of the same name already exists.
- **1107 - ERROR_ARANGO_DATADIR_LOCKED**
  Will be raised when the database directory is locked by a different process.
- **1108 - ERROR_ARANGO_COLLECTION_DIRECTORY_ALREADY_EXISTS**
  Will be raised when the collection cannot be created because a directory of the same name already exists.
- **1109 - ERROR_ARANGO_MSYNC_FAILED**
  Will be raised when the system call msync failed.
- **1110 - ERROR_ARANGO_DATADIR_UNLOCKABLE**
  Will be raised when the server cannot lock the database directory on startup.
- **1111 - ERROR_ARANGO_SYNC_TIMEOUT**
  Will be raised when the server waited too long for a datafile to be synced to disk.

General ArangoDB storage errors

For errors that occur when fulfilling a user request.

- **1200 - ERROR_ARANGO_CONFLICT**
  Will be raised when updating or deleting a document and a conflict has been detected.
- **1201 - ERROR_ARANGO_DATADIR_INVALID**
  Will be raised when a non-existing database directory was specified when starting the database.
- **1202 - ERROR_ARANGO_DOCUMENT_NOT_FOUND**
  Will be raised when a document with a given identifier or handle is unknown.
- **1203 - ERROR_ARANGO_COLLECTION_NOT_FOUND**
  Will be raised when a collection with a given identifier or name is unknown.
- **1204 - ERROR_ARANGO_COLLECTION_PARAMETER_MISSING**
  Will be raised when the collection parameter is missing.
- **1205 - ERROR_ARANGO_DOCUMENT_HANDLE_BAD**
  Will be raised when a document handle is corrupt.
- **1206 - ERROR_ARANGO_MAXIMAL_SIZE_TOO_SMALL**
  Will be raised when the maximal size of the journal is too small.
- **1207 - ERROR_ARANGO_DUPLICATE_NAME**
  Will be raised when a name duplicate is detected.
- **1208 - ERROR_ARANGO_ILLEGAL_NAME**
  Will be raised when an illegal name is detected.
- **1209 - ERROR_ARANGO_NO_INDEX**
  Will be raised when no suitable index for the query is known.
- **1210 - ERROR_ARANGO_UNIQUE_CONSTRAINT_VIOLATED**
  Will be raised when there is a unique constraint violation.
- **1212 - ERROR_ARANGO_INDEX_NOT_FOUND**
  Will be raised when an index with a given identifier is unknown.
- **1213 - ERROR_ARANGO_CROSS_COLLECTION_REQUEST**
  Will be raised when a cross-collection is requested.
- **1214 - ERROR_ARANGO_INDEX_HANDLE_BAD**
  Will be raised when a index handle is corrupt.
- **1216 - ERROR_ARANGO_DOCUMENT_TOO_LARGE**
  Will be raised when the document cannot fit into any datafile because of it is too large.
- **1217 - ERROR_ARANGO_COLLECTION_NOT_UNLOADED**
  Will be raised when a collection should be unloaded, but has a different status.
- **1218 - ERROR_ARANGO_COLLECTION_TYPE_INVALID**
  Will be raised when an invalid collection type is used in a request.
- **1219 - ERROR_ARANGO_VALIDATION_FAILED**
  Will be raised when the validation of an attribute of a structure failed.
- **1220 - ERROR_ARANGO_ATTRIBUTE_PARSER_FAILED**
  Will be raised when parsing an attribute name definition failed.
- **1221 - ERROR_ARANGO_DOCUMENT_KEY_BAD**
  Will be raised when a document key is corrupt.
- **1222 - ERROR_ARANGO_DOCUMENT_KEY_UNEXPECTED**
  Will be raised when a user-defined document key is supplied for collections with auto key generation.

- **1224 - ERROR_ARANGO_DATADIR_NOT_WRITABLE**
  Will be raised when the server's database directory is not writable for the current user.
- **1225 - ERROR_ARANGO_OUT_OF_KEYS**
  Will be raised when a key generator runs out of keys.
- **1226 - ERROR_ARANGO_DOCUMENT_KEY_MISSING**
  Will be raised when a document key is missing.
- **1227 - ERROR_ARANGO_DOCUMENT_TYPE_INVALID**
  Will be raised when there is an attempt to create a document with an invalid type.
- **1228 - ERROR_ARANGO_DATABASE_NOT_FOUND**
  Will be raised when a non-existing database is accessed.
- **1229 - ERROR_ARANGO_DATABASE_NAME_INVALID**
  Will be raised when an invalid database name is used.
- **1230 - ERROR_ARANGO_USE_SYSTEM_DATABASE**
  Will be raised when an operation is requested in a database other than the system database.
- **1231 - ERROR_ARANGO_ENDPOINT_NOT_FOUND**
  Will be raised when there is an attempt to delete a non-existing endpoint.
- **1232 - ERROR_ARANGO_INVALID_KEY_GENERATOR**
  Will be raised when an invalid key generator description is used.
- **1233 - ERROR_ARANGO_INVALID_EDGE_ATTRIBUTE**
  will be raised when the _from or _to values of an edge are undefined or contain an invalid value.
- **1234 - ERROR_ARANGO_INDEX_DOCUMENT_ATTRIBUTE_MISSING**
  Will be raised when an attempt to insert a document into an index is caused by in the document not having one or more attributes which the index is built on.
- **1235 - ERROR_ARANGO_INDEX_CREATION_FAILED**
  Will be raised when an attempt to create an index has failed.
- **1236 - ERROR_ARANGO_WRITE_THROTTLE_TIMEOUT**
  Will be raised when the server is write-throttled and a write operation has waited too long for the server to process queued operations.
- **1237 - ERROR_ARANGO_COLLECTION_TYPE_MISMATCH**
  Will be raised when a collection has a different type from what has been expected.
- **1238 - ERROR_ARANGO_COLLECTION_NOT_LOADED**
  Will be raised when a collection is accessed that is not yet loaded.
- **1239 - ERROR_ARANGO_DOCUMENT_REV_BAD**
  Will be raised when a document revision is corrupt or is missing where needed.

Checked ArangoDB storage errors

For errors that occur but are anticipated.

- **1300 - ERROR_ARANGO_DATAFILE_FULL**
  Will be raised when the datafile reaches its limit.
- **1301 - ERROR_ARANGO_EMPTY_DATADIR**
  Will be raised when encountering an empty server database directory.

ArangoDB replication errors

- **1400 - ERROR_REPLICATION_NO_RESPONSE**
  Will be raised when the replication applier does not receive any or an incomplete response from the master.
- **1401 - ERROR_REPLICATION_INVALID_RESPONSE**
  Will be raised when the replication applier receives an invalid response from the master.
- **1402 - ERROR_REPLICATION_MASTER_ERROR**
  Will be raised when the replication applier receives a server error from the master.
- **1403 - ERROR_REPLICATION_MASTER_INCOMPATIBLE**
  Will be raised when the replication applier connects to a master that has an incompatible version.
- **1404 - ERROR_REPLICATION_MASTER_CHANGE**
  Will be raised when the replication applier connects to a different master than before.
- **1405 - ERROR_REPLICATION_LOOP**
  Will be raised when the replication applier is asked to connect to itself for replication.
- **1406 - ERROR_REPLICATION_UNEXPECTED_MARKER**
  Will be raised when an unexpected marker is found in the replication log stream.
- **1407 - ERROR_REPLICATION_INVALID_APPLIER_STATE**
  Will be raised when an invalid replication applier state file is found.
- **1408 - ERROR_REPLICATION_UNEXPECTED_TRANSACTION**
  Will be raised when an unexpected transaction id is found.
- **1410 - ERROR_REPLICATION_INVALID_APPLIER_CONFIGURATION**
  Will be raised when the configuration for the replication applier is invalid.
- **1411 - ERROR_REPLICATION_RUNNING**
  Will be raised when there is an attempt to perform an operation while the replication applier is running.
- **1412 - ERROR_REPLICATION_APPLIER_STOPPED**
  Special error code used to indicate the replication applier was stopped by a user.
- **1413 - ERROR_REPLICATION_NO_START_TICK**
  Will be raised when the replication applier is started without a known start tick value.
- **1414 - ERROR_REPLICATION_START_TICK_NOT_PRESENT**
  Will be raised when the replication applier fetches data using a start tick, but that start tick is not present on the logger server anymore.

ArangoDB cluster errors

- **1450 - ERROR_CLUSTER_NO_AGENCY**
  Will be raised when none of the agency servers can be connected to.
- **1451 - ERROR_CLUSTER_NO_COORDINATOR_HEADER**
  Will be raised when a DB server in a cluster receives a HTTP request without a coordinator header.
- **1452 - ERROR_CLUSTER_COULD_NOT_LOCK_PLAN**
  Will be raised when a coordinator in a cluster cannot lock the Plan hierarchy in the agency.
- **1453 - ERROR_CLUSTER_COLLECTION_ID_EXISTS**
  Will be raised when a coordinator in a cluster tries to create a collection and the collection ID already exists.
- **1454 - ERROR_CLUSTER_COULD_NOT_CREATE_COLLECTION_IN_PLAN**
  Will be raised when a coordinator in a cluster cannot create an entry for a new collection in the Plan hierarchy in the agency.
- **1455 - ERROR_CLUSTER_COULD_NOT_READ_CURRENT_VERSION**
  Will be raised when a coordinator in a cluster cannot read the Version entry in the Current hierarchy in the agency.

- **1456 - ERROR_CLUSTER_COULD_NOT_CREATE_COLLECTION**
  Will be raised when a coordinator in a cluster notices that some DBServers report problems when creating shards for a new collection.
- **1457 - ERROR_CLUSTER_TIMEOUT**
  Will be raised when a coordinator in a cluster runs into a timeout for some cluster wide operation.
- **1458 - ERROR_CLUSTER_COULD_NOT_REMOVE_COLLECTION_IN_PLAN**
  Will be raised when a coordinator in a cluster cannot remove an entry for a collection in the Plan hierarchy in the agency.
- **1459 - ERROR_CLUSTER_COULD_NOT_REMOVE_COLLECTION_IN_CURRENT**
  Will be raised when a coordinator in a cluster cannot remove an entry for a collection in the Current hierarchy in the agency.
- **1460 - ERROR_CLUSTER_COULD_NOT_CREATE_DATABASE_IN_PLAN**
  Will be raised when a coordinator in a cluster cannot create an entry for a new database in the Plan hierarchy in the agency.
- **1461 - ERROR_CLUSTER_COULD_NOT_CREATE_DATABASE**
  Will be raised when a coordinator in a cluster notices that some DBServers report problems when creating databases for a new cluster wide database.
- **1462 - ERROR_CLUSTER_COULD_NOT_REMOVE_DATABASE_IN_PLAN**
  Will be raised when a coordinator in a cluster cannot remove an entry for a database in the Plan hierarchy in the agency.
- **1463 - ERROR_CLUSTER_COULD_NOT_REMOVE_DATABASE_IN_CURRENT**
  Will be raised when a coordinator in a cluster cannot remove an entry for a database in the Current hierarchy in the agency.
- **1464 - ERROR_CLUSTER_SHARD_GONE**
  Will be raised when a coordinator in a cluster cannot determine the shard that is responsible for a given document.
- **1465 - ERROR_CLUSTER_CONNECTION_LOST**
  Will be raised when a coordinator in a cluster loses an HTTP connection to a DBserver in the cluster whilst transferring data.
- **1466 - ERROR_CLUSTER_MUST_NOT_SPECIFY_KEY**
  Will be raised when a coordinator in a cluster finds that the _key attribute was specified in a sharded collection the uses not only _key as sharding attribute.
- **1467 - ERROR_CLUSTER_GOT_CONTRADICTING_ANSWERS**
  Will be raised if a coordinator in a cluster gets conflicting results from different shards, which should never happen.
- **1468 - ERROR_CLUSTER_NOT_ALL_SHARDING_ATTRIBUTES_GIVEN**
  Will be raised if a coordinator tries to find out which shard is responsible for a partial document, but cannot do this because not all sharding attributes are specified.
- **1469 - ERROR_CLUSTER_MUST_NOT_CHANGE_SHARDING_ATTRIBUTES**
  Will be raised if there is an attempt to update the value of a shard attribute.
- **1470 - ERROR_CLUSTER_UNSUPPORTED**
  Will be raised when there is an attempt to carry out an operation that is not supported in the context of a sharded collection.
- **1471 - ERROR_CLUSTER_ONLY_ON_COORDINATOR**
  Will be raised if there is an attempt to run a coordinator-only operation on a different type of node.
- **1472 - ERROR_CLUSTER_READING_PLAN_AGENCY**
  Will be raised if a coordinator or DBserver cannot read the Plan in the agency.
- **1473 - ERROR_CLUSTER_COULD_NOT_TRUNCATE_COLLECTION**
  Will be raised if a coordinator cannot truncate all shards of a cluster collection.
- **1474 - ERROR_CLUSTER_AQL_COMMUNICATION**
  Will be raised if the internal communication of the cluster for AQL produces an error.
- **1475 - ERROR_ARANGO_DOCUMENT_NOT_FOUND_OR_SHARDING_ATTRIBUTES_CHANGED**
  Will be raised when a document with a given identifier or handle is unknown, or if the sharding attributes have been changed in a REPLACE operation in the cluster.

- **1476 - ERROR_CLUSTER_COULD_NOT_DETERMINE_ID**
  Will be raised if a cluster server at startup could not determine its own ID from the local info provided.
- **1477 - ERROR_CLUSTER_ONLY_ON_DBSERVER**
  Will be raised if there is an attempt to run a DBserver-only operation on a different type of node.
- **1478 - ERROR_CLUSTER_BACKEND_UNAVAILABLE**
  Will be raised if a required db server can't be reached.
- **1479 - ERROR_CLUSTER_UNKNOWN_CALLBACK_ENDPOINT**
  An endpoint couldn't be found
- **1480 - ERROR_CLUSTER_AGENCY_STRUCTURE_INVALID**
  The structure in the agency is invalid

ArangoDB query errors

- **1500 - ERROR_QUERY_KILLED**
  Will be raised when a running query is killed by an explicit admin command.
- **1501 - ERROR_QUERY_PARSE**
  Will be raised when query is parsed and is found to be syntactically invalid.
- **1502 - ERROR_QUERY_EMPTY**
  Will be raised when an empty query is specified.
- **1503 - ERROR_QUERY_SCRIPT**
  Will be raised when a runtime error is caused by the query.
- **1504 - ERROR_QUERY_NUMBER_OUT_OF_RANGE**
  Will be raised when a number is outside the expected range.
- **1510 - ERROR_QUERY_VARIABLE_NAME_INVALID**
  Will be raised when an invalid variable name is used.
- **1511 - ERROR_QUERY_VARIABLE_REDECLARED**
  Will be raised when a variable gets re-assigned in a query.
- **1512 - ERROR_QUERY_VARIABLE_NAME_UNKNOWN**
  Will be raised when an unknown variable is used or the variable is undefined the context it is used.
- **1521 - ERROR_QUERY_COLLECTION_LOCK_FAILED**
  Will be raised when a read lock on the collection cannot be acquired.
- **1522 - ERROR_QUERY_TOO_MANY_COLLECTIONS**
  Will be raised when the number of collections in a query is beyond the allowed value.
- **1530 - ERROR_QUERY_DOCUMENT_ATTRIBUTE_REDECLARED**
  Will be raised when a document attribute is re-assigned.
- **1540 - ERROR_QUERY_FUNCTION_NAME_UNKNOWN**
  Will be raised when an undefined function is called.
- **1541 - ERROR_QUERY_FUNCTION_ARGUMENT_NUMBER_MISMATCH**
  Will be raised when the number of arguments used in a function call does not match the expected number of arguments for the function.
- **1542 - ERROR_QUERY_FUNCTION_ARGUMENT_TYPE_MISMATCH**
  Will be raised when the type of an argument used in a function call does not match the expected argument type.
- **1543 - ERROR_QUERY_INVALID_REGEX**
  Will be raised when an invalid regex argument value is used in a call to a function that expects a regex.
- **1550 - ERROR_QUERY_BIND_PARAMETERS_INVALID**
  Will be raised when the structure of bind parameters passed has an unexpected format.

- **1551 - ERROR_QUERY_BIND_PARAMETER_MISSING**
  Will be raised when a bind parameter was declared in the query but the query is being executed with no value for that parameter.
- **1552 - ERROR_QUERY_BIND_PARAMETER_UNDECLARED**
  Will be raised when a value gets specified for an undeclared bind parameter.
- **1553 - ERROR_QUERY_BIND_PARAMETER_TYPE**
  Will be raised when a bind parameter has an invalid value or type.
- **1560 - ERROR_QUERY_INVALID_LOGICAL_VALUE**
  Will be raised when a non-boolean value is used in a logical operation.
- **1561 - ERROR_QUERY_INVALID_ARITHMETIC_VALUE**
  Will be raised when a non-numeric value is used in an arithmetic operation.
- **1562 - ERROR_QUERY_DIVISION_BY_ZERO**
  Will be raised when there is an attempt to divide by zero.
- **1563 - ERROR_QUERY_ARRAY_EXPECTED**
  Will be raised when a non-array operand is used for an operation that expects an array argument operand.
- **1569 - ERROR_QUERY_FAIL_CALLED**
  Will be raised when the function FAIL() is called from inside a query.
- **1570 - ERROR_QUERY_GEO_INDEX_MISSING**
  Will be raised when a geo restriction was specified but no suitable geo index is found to resolve it.
- **1571 - ERROR_QUERY_FULLTEXT_INDEX_MISSING**
  Will be raised when a fulltext query is performed on a collection without a suitable fulltext index.
- **1572 - ERROR_QUERY_INVALID_DATE_VALUE**
  Will be raised when a value cannot be converted to a date.
- **1573 - ERROR_QUERY_MULTI_MODIFY**
  Will be raised when an AQL query contains more than one data-modifying operation.
- **1574 - ERROR_QUERY_INVALID_AGGREGATE_EXPRESSION**
  Will be raised when an AQL query contains an invalid aggregate expression.
- **1575 - ERROR_QUERY_COMPILE_TIME_OPTIONS**
  Will be raised when an AQL data-modification query contains options that cannot be figured out at query compile time.
- **1576 - ERROR_QUERY_EXCEPTION_OPTIONS**
  Will be raised when an AQL data-modification query contains an invalid options specification.
- **1577 - ERROR_QUERY_COLLECTION_USED_IN_EXPRESSION**
  Will be raised when a collection is used as an operand in an AQL expression.
- **1578 - ERROR_QUERY_DISALLOWED_DYNAMIC_CALL**
  Will be raised when a dynamic function call is made to a function that cannot be called dynamically.
- **1579 - ERROR_QUERY_ACCESS_AFTER_MODIFICATION**
  Will be raised when collection data are accessed after a data-modification operation.

AQL user function errors

- **1580 - ERROR_QUERY_FUNCTION_INVALID_NAME**
  Will be raised when a user function with an invalid name is registered.
- **1581 - ERROR_QUERY_FUNCTION_INVALID_CODE**
  Will be raised when a user function is registered with invalid code.
- **1582 - ERROR_QUERY_FUNCTION_NOT_FOUND**
  Will be raised when a user function is accessed but not found.

- **1583 - ERROR_QUERY_FUNCTION_RUNTIME_ERROR**
  Will be raised when a user function throws a runtime exception.

AQL query registry errors

- **1590 - ERROR_QUERY_BAD_JSON_PLAN**
  Will be raised when an HTTP API for a query got an invalid JSON object.
- **1591 - ERROR_QUERY_NOT_FOUND**
  Will be raised when an Id of a query is not found by the HTTP API.
- **1592 - ERROR_QUERY_IN_USE**
  Will be raised when an Id of a query is found by the HTTP API but the query is in use.

ArangoDB cursor errors

- **1600 - ERROR_CURSOR_NOT_FOUND**
  Will be raised when a cursor is requested via its id but a cursor with that id cannot be found.
- **1601 - ERROR_CURSOR_BUSY**
  Will be raised when a cursor is requested via its id but a concurrent request is still using the cursor.

ArangoDB transaction errors

- **1650 - ERROR_TRANSACTION_INTERNAL**
  Will be raised when a wrong usage of transactions is detected. this is an internal error and indicates a bug in ArangoDB.
- **1651 - ERROR_TRANSACTION_NESTED**
  Will be raised when transactions are nested.
- **1652 - ERROR_TRANSACTION_UNREGISTERED_COLLECTION**
  Will be raised when a collection is used in the middle of a transaction but was not registered at transaction start.
- **1653 - ERROR_TRANSACTION_DISALLOWED_OPERATION**
  Will be raised when a disallowed operation is carried out in a transaction.
- **1654 - ERROR_TRANSACTION_ABORTED**
  Will be raised when a transaction was aborted.

User management errors

- **1700 - ERROR_USER_INVALID_NAME**
  Will be raised when an invalid user name is used.
- **1701 - ERROR_USER_INVALID_PASSWORD**
  Will be raised when an invalid password is used.
- **1702 - ERROR_USER_DUPLICATE**
  Will be raised when a user name already exists.
- **1703 - ERROR_USER_NOT_FOUND**
  Will be raised when a user name is updated that does not exist.
- **1704 - ERROR_USER_CHANGE_PASSWORD**
  Will be raised when the user must change his password.

Service management errors (legacy)

These have been superceded by the Foxx management errors in public APIs.

- **1750 - ERROR_SERVICE_INVALID_NAME**
  Will be raised when an invalid service name is specified.
- **1751 - ERROR_SERVICE_INVALID_MOUNT**
  Will be raised when an invalid mount is specified.
- **1752 - ERROR_SERVICE_DOWNLOAD_FAILED**
  Will be raised when a service download from the central repository failed.
- **1753 - ERROR_SERVICE_UPLOAD_FAILED**
  Will be raised when a service upload from the client to the ArangoDB server failed.

Task errors

- **1850 - ERROR_TASK_INVALID_ID**
  Will be raised when a task is created with an invalid id.
- **1851 - ERROR_TASK_DUPLICATE_ID**
  Will be raised when a task id is created with a duplicate id.
- **1852 - ERROR_TASK_NOT_FOUND**
  Will be raised when a task with the specified id could not be found.

Graph / traversal errors

- **1901 - ERROR_GRAPH_INVALID_GRAPH**
  Will be raised when an invalid name is passed to the server.
- **1902 - ERROR_GRAPH_COULD_NOT_CREATE_GRAPH**
  Will be raised when an invalid name, vertices or edges is passed to the server.
- **1903 - ERROR_GRAPH_INVALID_VERTEX**
  Will be raised when an invalid vertex id is passed to the server.
- **1904 - ERROR_GRAPH_COULD_NOT_CREATE_VERTEX**
  Will be raised when the vertex could not be created.
- **1905 - ERROR_GRAPH_COULD_NOT_CHANGE_VERTEX**
  Will be raised when the vertex could not be changed.
- **1906 - ERROR_GRAPH_INVALID_EDGE**
  Will be raised when an invalid edge id is passed to the server.
- **1907 - ERROR_GRAPH_COULD_NOT_CREATE_EDGE**
  Will be raised when the edge could not be created.
- **1908 - ERROR_GRAPH_COULD_NOT_CHANGE_EDGE**
  Will be raised when the edge could not be changed.
- **1909 - ERROR_GRAPH_TOO_MANY_ITERATIONS**
  Will be raised when too many iterations are done in a graph traversal.
- **1910 - ERROR_GRAPH_INVALID_FILTER_RESULT**
  Will be raised when an invalid filter result is returned in a graph traversal.
- **1920 - ERROR_GRAPH_COLLECTION_MULTI_USE**
  an edge collection may only be used once in one edge definition of a graph.,
- **1921 - ERROR_GRAPH_COLLECTION_USE_IN_MULTI_GRAPHS**
  is already used by another graph in a different edge definition.,

- **1922 - ERROR_GRAPH_CREATE_MISSING_NAME**
  a graph name is required to create a graph.,
- **1923 - ERROR_GRAPH_CREATE_MALFORMED_EDGE_DEFINITION**
  the edge definition is malformed. It has to be an array of objects.,
- **1924 - ERROR_GRAPH_NOT_FOUND**
  a graph with this name could not be found.,
- **1925 - ERROR_GRAPH_DUPLICATE**
  a graph with this name already exists.,
- **1926 - ERROR_GRAPH_VERTEX_COL_DOES_NOT_EXIST**
  the specified vertex collection does not exist or is not part of the graph.,
- **1927 - ERROR_GRAPH_WRONG_COLLECTION_TYPE_VERTEX**
  the collection is not a vertex collection.,
- **1928 - ERROR_GRAPH_NOT_IN_ORPHAN_COLLECTION**
  Vertex collection not in orphan collection of the graph.,
- **1929 - ERROR_GRAPH_COLLECTION_USED_IN_EDGE_DEF**
  The collection is already used in an edge definition of the graph.,
- **1930 - ERROR_GRAPH_EDGE_COLLECTION_NOT_USED**
  The edge collection is not used in any edge definition of the graph.,
- **1931 - ERROR_GRAPH_NOT_AN_ARANGO_COLLECTION**
  The collection is not an ArangoCollection.,
- **1932 - ERROR_GRAPH_NO_GRAPH_COLLECTION**
  collection _graphs does not exist.,
- **1933 - ERROR_GRAPH_INVALID_EXAMPLE_ARRAY_OBJECT_STRING**
  Invalid example type. Has to be String, Array or Object.,
- **1934 - ERROR_GRAPH_INVALID_EXAMPLE_ARRAY_OBJECT**
  Invalid example type. Has to be Array or Object.,
- **1935 - ERROR_GRAPH_INVALID_NUMBER_OF_ARGUMENTS**
  Invalid number of arguments. Expected: ,
- **1936 - ERROR_GRAPH_INVALID_PARAMETER**
  Invalid parameter type.,
- **1937 - ERROR_GRAPH_INVALID_ID**
  Invalid id,
- **1938 - ERROR_GRAPH_COLLECTION_USED_IN_ORPHANS**
  The collection is already used in the orphans of the graph.,
- **1939 - ERROR_GRAPH_EDGE_COL_DOES_NOT_EXIST**
  the specified edge collection does not exist or is not part of the graph.,
- **1940 - ERROR_GRAPH_EMPTY**
  The requested graph has no edge collections.

Session errors

- **1950 - ERROR_SESSION_UNKNOWN**
  Will be raised when an invalid/unknown session id is passed to the server.
- **1951 - ERROR_SESSION_EXPIRED**
  Will be raised when a session is expired.

Simple Client errors

- **2000 - SIMPLE_CLIENT_UNKNOWN_ERROR**
  This error should not happen.
- **2001 - SIMPLE_CLIENT_COULD_NOT_CONNECT**
  Will be raised when the client could not connect to the server.
- **2002 - SIMPLE_CLIENT_COULD_NOT_WRITE**
  Will be raised when the client could not write data.
- **2003 - SIMPLE_CLIENT_COULD_NOT_READ**
  Will be raised when the client could not read data.

Communicator errors

COMMUNICATOR_REQUEST_ABORTED,2100,"Request aborted", "Request was aborted."

Foxx management errors

- **3000 - ERROR_MALFORMED_MANIFEST_FILE**
  The service manifest file is not well-formed JSON.
- **3001 - ERROR_INVALID_SERVICE_MANIFEST**
  The service manifest contains invalid values.
- **3004 - ERROR_INVALID_FOXX_OPTIONS**
  The service options contain invalid values.
- **3007 - ERROR_INVALID_MOUNTPOINT**
  The service mountpath contains invalid characters.
- **3009 - ERROR_SERVICE_NOT_FOUND**
  No service found at the given mountpath.
- **3010 - ERROR_SERVICE_NEEDS_CONFIGURATION**
  The service is missing configuration or dependencies.
- **3011 - ERROR_SERVICE_MOUNTPOINT_CONFLICT**
  A service already exists at the given mountpath.
- **3012 - ERROR_SERVICE_MANIFEST_NOT_FOUND**
  The service directory does not contain a manifest file.
- **3013 - ERROR_SERVICE_OPTIONS_MALFORMED**
  The service options are not well-formed JSON.
- **3014 - ERROR_SERVICE_SOURCE_NOT_FOUND**
  The source path does not match a file or directory.
- **3015 - ERROR_SERVICE_SOURCE_ERROR**
  The source path could not be resolved.
- **3016 - ERROR_SERVICE_UNKNOWN_SCRIPT**
  The service does not have a script with this name.

JavaScript module loader errors

- **3100 - ERROR_MODULE_NOT_FOUND**
  The module path could not be resolved.

- **3103 - ERROR_MODULE_FAILURE**
  Failed to invoke the module in its context.

Enterprise errors

- **4000 - ERROR_NO_SMART_COLLECTION**
  The requested collection needs to be smart, but it ain't
- **4001 - ERROR_NO_SMART_GRAPH_ATTRIBUTE**
  The given document does not have the smart graph attribute set.
- **4002 - ERROR_CANNOT_DROP_SMART_COLLECTION**
  This smart collection cannot be dropped, it dictates sharding in the graph.
- **4003 - ERROR_KEY_MUST_BE_PREFIXED_WITH_SMART_GRAPH_ATTRIBUTE**
  In a smart vertex collection _key must be prefixed with the value of the smart graph attribute.
- **4004 - ERROR_ILLEGAL_SMART_GRAPH_ATTRIBUTE**
  The given smartGraph attribute is illegal and connot be used for sharding. All system attributes are forbidden.

Agency errors

- **20011 - ERROR_AGENCY_INFORM_MUST_BE_OBJECT**
  The inform message in the agency must be an object.
- **20012 - ERROR_AGENCY_INFORM_MUST_CONTAIN_TERM**
  The inform message in the agency must contain a uint parameter 'term'.
- **20013 - ERROR_AGENCY_INFORM_MUST_CONTAIN_ID**
  The inform message in the agency must contain a string parameter 'id'.
- **20014 - ERROR_AGENCY_INFORM_MUST_CONTAIN_ACTIVE**
  The inform message in the agency must contain an array 'active'.
- **20015 - ERROR_AGENCY_INFORM_MUST_CONTAIN_POOL**
  The inform message in the agency must contain an object 'pool'.

Dispatcher errors

- **21001 - ERROR_DISPATCHER_IS_STOPPING**
  Will be returned if a shutdown is in progress.
- **21002 - ERROR_QUEUE_UNKNOWN**
  Will be returned if a queue with this name does not exist.
- **21003 - ERROR_QUEUE_FULL**
  Will be returned if a queue with this name is full.

Glossary

Collection

A collection consists of documents. It is uniquely identified by its collection identifier. It also has a unique name that clients should use to identify and access it. Collections can be renamed. It will change the collection name, but not the collection identifier. Collections contain documents of a specific type. There are currently two types: document (default) and edge. The type is specified by the user when the collection is created, and cannot be changed later.

Collection Identifier

A collection identifier identifies a collection in a database. It is a string value and is unique within the database. Up to including ArangoDB 1.1, the collection identifier has been a client's primary means to access collections. Starting with ArangoDB 1.2, clients should instead use a collection's unique name to access a collection instead of its identifier.

ArangoDB currently uses 64bit unsigned integer values to maintain collection ids internally. When returning collection ids to clients, ArangoDB will put them into a string to ensure the collection id is not clipped by clients that do not support big integers. Clients should treat the collection ids returned by ArangoDB as opaque strings when they store or use it locally.

Collection Name

A collection name identifies a collection in a database. It is a string and is unique within the database. Unlike the collection identifier it is supplied by the creator of the collection. The collection name must consist of letters, digits, and the _ (underscore) and - (dash) characters only. Please refer to NamingConventions for more information on valid collection names.

Database

ArangoDB can handle multiple databases in the same server instance. Databases can be used to logically group and separate data. An ArangoDB database consists of collections and dedicated database-specific worker processes.

A database contains its own collections (which cannot be accessed from other databases), Foxx applications, and replication loggers and appliers. Each ArangoDB database contains its own system collections (e.g. _users, _replication, ...).

There will always be at least one database in ArangoDB. This is the default database, named _system. This database cannot be dropped, and provides special operations for creating, dropping, and enumerating databases. Users can create additional databases and give them unique names to access them later. Database management operations cannot be initiated from out of user-defined databases.

When ArangoDB is accessed via its HTTP REST API, the database name is read from the first part of the request URI path (e.g. /_db/_system/...). If the request URI does not contain a database name, the database name is automatically derived from the endpoint. Please refer to DatabaseEndpoint for more information.

Database Name

A single ArangoDB instance can handle multiple databases in parallel. When multiple databases are used, each database must be given a unique name. This name is used to uniquely identify a database. The default database in ArangoDB is named _system.

The database name is a string consisting of only letters, digits and the _ (underscore) and - (dash) characters. User-defined database names must always start with a letter. Database names is case-sensitive.

Database Organization

A single ArangoDB instance can handle multiple databases in parallel. By default, there will be at least one database, which is named _system.

Databases are physically stored in separate sub-directories underneath the database directory, which itself resides in the instance's data directory.

Each database has its own sub-directory, named database-. The database directory contains sub-directories for the collections of the database, and a file named parameter.json. This file contains the database id and name.

In an example ArangoDB instance which has two databases, the filesystem layout could look like this:

```
data/                     # the instance's data directory
  databases/              # sub-directory containing all databases' data
    database-<id>/        # sub-directory for a single database
      parameter.json      # file containing database id and name
      collection-<id>/    # directory containing data about a collection
    database-<id>/        # sub-directory for another database
      parameter.json      # file containing database id and name
      collection-<id>/    # directory containing data about a collection
      collection-<id>/    # directory containing data about a collection
```

Foxx applications are also organized in database-specific directories inside the application path. The filesystem layout could look like this:

```
apps/                   # the instance's application directory
  system/               # system applications (can be ignored)
  _db/                  # sub-directory containing database-specific applications
    <database-name>/    # sub-directory for a single database
      <mountpoint>/APP  # sub-directory for a single application
      <mountpoint>/APP  # sub-directory for a single application
    <database-name>/    # sub-directory for another database
      <mountpoint>/APP  # sub-directory for a single application
```

Document

Documents in ArangoDB are JSON objects. These objects can be nested (to any depth) and may contain arrays. Each document is uniquely identified by its document handle.

Document ETag

The document revision (`_rev` value) enclosed in double quotes. The revision is returned by several HTTP API methods in the Etag HTTP header.

Document Handle

A document handle uniquely identifies a document in the database. It is a string and consists of the collection's name and the document key (`_key` attribute) separated by /. The document handle is stored in a document's `_id` attribute.

Document Key

A document key is a string that uniquely identifies a document in a given collection. It can and should be used by clients when specific documents are searched. Document keys are stored in the `_key` attribute of documents. The key values are automatically indexed by ArangoDB in a collection's primary index. Thus looking up a document by its key is regularly a fast operation. The `_key` value of a document is immutable once the document has been created.

By default, ArangoDB will auto-generate a document key if no `_key` attribute is specified, and use the user-specified `_key` value otherwise.

This behavior can be changed on a per-collection level by creating collections with the `keyOptions` attribute.

Using `keyOptions` it is possible to disallow user-specified keys completely, or to force a specific regime for auto-generating the `_key` values.

There are some restrictions for user-defined keys (see NamingConventions for document keys).

Document Revision

As ArangoDB supports MVCC, documents can exist in more than one revision. The document revision is the MVCC token used to identify a particular revision of a document. It is a string value currently containing an integer number and is unique within the list of document revisions for a single document. Document revisions can be used to conditionally update, replace or delete documents in the database. In order to find a particular revision of a document, you need the document handle and the document revision.

The document revision is stored in the `_rev` attribute of a document, and is set and updated by ArangoDB automatically. The `_rev` value cannot be set from the outside.

ArangoDB currently uses 64bit unsigned integer values to maintain document revisions internally. When returning document revisions to clients, ArangoDB will put them into a string to ensure the revision id is not clipped by clients that do not support big integers. Clients should treat the revision id returned by ArangoDB as an opaque string when they store or use it locally. This will allow ArangoDB to change the format of revision ids later if this should be required. Clients can use revisions ids to perform simple equality/non-equality comparisons (e.g. to check whether a document has changed or not), but they should not use revision ids to perform greater/less than comparisons with them to check if a document revision is older than one another, even if this might work for some cases.

Edge

Edges are special documents used for connecting other documents into a graph. An edge describes the connection between two documents using the internal attributes: `_from` and `_to`. These contain document handles, namely the start-point and the end-point of the edge.

Edge Collection

Edge collections are collections that store edges.

Edge Definition

Edge definitions are parts of the definition of `named graphs`. They describe which edge collections connect which vertex collections.

General Graph

Module maintaining graph setup in the `_graphs` collection - aka `named graphs`. Configures which edge collections relate to which vertex collections. Ensures graph consistency in modification queries.

Named Graphs

Named graphs enforce consistency between edge collections and vertex collections, so if you remove a vertex, edges pointing to it will be removed too.

Index

Indexes are used to allow fast access to documents in a collection. All collections have a primary index, which is the document's _key attribute. This index cannot be dropped or changed.

Edge collections will also have an automatically created edges index, which cannot be modified. This index provides quick access to documents via the `_from` and `_to` attributes.

Most user-land indexes can be created by defining the names of the attributes which should be indexed. Some index types allow indexing just one attribute (e.g. fulltext index) whereas other index types allow indexing multiple attributes.

Indexing the system attribute `_id` in user-defined indexes is not supported by any index type.

Edges Index

An edges index is automatically created for edge collections. It contains connections between vertex documents and is invoked when the connecting edges of a vertex are queried. There is no way to explicitly create or delete edges indexes.

Fulltext Index

A fulltext index can be used to find words, or prefixes of words inside documents. A fulltext index can be defined on one attribute only, and will include all words contained in documents that have a textual value in the index attribute. Since ArangoDB 2.6 the index will also include words from the index attribute if the index attribute is an array of strings, or an object with string value members.

For example, given a fulltext index on the `translations` attribute and the following documents, then searching for `лиса` using the fulltext index would return only the first document. Searching for the index for the exact string `Fox` would return the first two documents, and searching for `prefix:Fox` would return all three documents:

```
{ translations: { en: "fox", de: "Fuchs", fr: "renard", ru: "лиса" } }
{ translations: "Fox is the English translation of the German word Fuchs" }
{ translations: [ "ArangoDB", "document", "database", "Foxx" ] }
```

If the index attribute is neither a string, an object or an array, its contents will not be indexed. When indexing the contents of an array attribute, an array member will only be included in the index if it is a string. When indexing the contents of an object attribute, an object member value will only be included in the index if it is a string. Other data types are ignored and not indexed.

Only words with a (specifiable) minimum length are indexed. Word tokenization is done using the word boundary analysis provided by libicu, which is taking into account the selected language provided at server start. Words are indexed in their lower-cased form. The index supports complete match queries (full words) and prefix queries.

Geo Index

A geo index is used to find places on the surface of the earth fast.

Index Handle

An index handle uniquely identifies an index in the database. It is a string and consists of a collection name and an index identifier separated by /.

Hash Index

A hash index is used to find documents based on examples. A hash index can be created for one or multiple document attributes.

A hash index will only be used by queries if all indexed attributes are present in the example or search query, and if all attributes are compared using the equality (== operator). That means the hash index does not support range queries.

A unique hash index has an amortized complexity of O(1) for lookup, insert, update, and remove operations. The non-unique hash index is similar, but amortized lookup performance is O(n), with n being the number of index entries with the same lookup value.

Skiplist Index

A skiplist is a sorted index type that can be used to find ranges of documents.

Anonymous Graphs

You may use edge collections with vertex collections without the graph management facilities. However, graph consistency is not enforced by these. If you remove vertices, you have to ensure by yourselves edges pointing to this vertex are removed. Anonymous graphs may not be browsed using graph viewer in the webinterface. This may be faster in some scenarios.