Table of Contents

ArangoDB vdevel 05. Jul 2018 Drivers Documentation

Official drivers

Name	Language	Repository	
ArangoDB-Java-Driver	Java	https://github.com/arangodb/arangodb-java-driver	Changelog
ArangoDB-Java-Driver-Async	Java	https://github.com/arangodb/arangodb-java-driver-async	Changelog
ArangoJS	JavaScript	https://github.com/arangodb/arangojs	Changelog
ArangoDB-PHP	PHP	https://github.com/arangodb/arangodb-php	Changelog
Go-Driver	Go	https://github.com/arangodb/go-driver	

Integrations

Name	Language	Repository	
Spring Data	Java	https://github.com/arangodb/spring-data	Changelog
ArangoDB-Spark-Connector	Scala, Java	https://github.com/arangodb/arangodb-spark-connector	Changelog

Community drivers

Please note that this list is not exhaustive.

Name	Language	Repository
ArangoDB-PHP-Core	PHP	https://github.com/frankmayer/ArangoDB-PHP-Core
ArangoDB-NET	.NET	https://github.com/yojimbo87/ArangoDB-NET
aranGO	Go	https://github.com/diegogub/aranGO
arangolite	Go	https://github.com/solher/arangolite
aranGoDriver	Go	https://github.com/TobiEiss/aranGoDriver
pyArango	Python	http://www.github.com/tariqdaouda/pyArango
python-arango	Python	https://github.com/Joowani/python-arango
Scarango	Scala	https://github.com/outr/scarango
ArangoRB	Ruby	https://github.com/StefanoMartin/ArangoRB

ArangoDB Java Driver

The official ArangoDB Java Driver.

- Getting Started
- Reference

See Also

- ChangeLog
- Examples
- Tutorial
- JavaDoc
- JavaDoc Velocy Pack

ArangoDB Java Driver - Getting Started

Supported versions

arangodb-java-driver	ArangoDB	network protocol	Java version
4.3.x	3.0.0+	Velocy Stream, HTTP	1.6+
4.2.x	3.0.0+	Velocy Stream, HTTP	1.6+
4.1.x	3.1.0+	Velocy Stream	1.6+
3.1.x	3.1.0+	HTTP	1.6+
3.0.x	3.0.x	HTTP	1.6+
2.7.4	2.7.x, 2.8.x	HTTP	1.6+

Note: Velocy Stream is only supported in ArangoDB 3.1 and above.

Maven

To add the driver to your project with maven, add the following code to your pom.xml (please use a driver with a version number compatible to your ArangoDB server's version):

ArangoDB 3.x.x

```
<dependencies>
  <dependency>
    <groupId>com.arangodb</groupId>
    <artifactId>arangodb-java-driver</artifactId>
    <version>4.3.0</version>
  </dependency>
</dependencies>
```

If you want to test with a snapshot version (e.g. 4.3.0-SNAPSHOT), add the staging repository of oss.sonatype.org to your pom.xml:

```
<repositories>
  <repository>
    <id>arangodb-snapshots</id>
    <url>https://oss.sonatype.org/content/groups/staging</url>
  </repository>
</repositories>
```

Compile the Java Driver

```
mvn clean install -DskipTests=true -Dgpg.skip=true -Dmaven.javadoc.skip=true -B
```

ArangoDB Java Driver - Reference

Driver setup

Setup with default configuration, this automatically loads a properties file arangodb.properties if exists in the classpath:

```
// this instance is thread-safe
ArangoDB arangoDB = new ArangoDB.Builder().build();
```

The driver is configured with some default values:

property-key	description	default value
arangodb.hosts	ArangoDB hosts	127.0.0.1:8529
arangodb.timeout	socket connect timeout(millisecond)	0
arangodb.user	Basic Authentication User	
arangodb.password	Basic Authentication Password	
arangodb.useSsl	use SSL connection	false
arangodb.chunksize	VelocyStream Chunk content-size(bytes)	30000
arangodb.connections.max	max number of connections	1 VST, 20 HTTP
arangodb.protocol	used network protocol	VST

To customize the configuration the parameters can be changed in the code...

```
ArangoDB arangoDB = new ArangoDB.Builder().host("192.168.182.50", 8888).build();
```

... or with a custom properties file (my.properties)

```
InputStream in = MyClass.class.getResourceAsStream("my.properties");
ArangoDB arangoDB = new ArangoDB.Builder().loadProperties(in).build();
```

Example for arangodb.properties:

```
arangodb.hosts=127.0.0.1:8529,127.0.0.1:8529
arangodb.user=root
arangodb.password=
```

Network protocol

The drivers default used network protocol is the binary protocol VelocyStream which offers the best performance within the driver. To use HTTP, you have to set the configuration `useProtocol` to `Protocol.HTTP_JSON` for HTTP with Json content or `Protocol.HTTP_VPACK` for HTTP with VelocyPack content.

```
ArangoDB arangoDB = new ArangoDB.Builder().useProtocol(Protocol.VST).build();
```

In addition to set the configuration for HTTP you have to add the apache httpclient to your classpath.

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.1</version>
</dependency>
```

Note: If you are using ArangoDB 3.0.x you have to set the protocol to `Protocol.HTTP_JSON` because it is the only one supported.

SSL

To use SSL, you have to set the configuration `useSsl` to `true` and set a `SSLContext` (see example code).

```
ArangoDB arangoDB = new ArangoDB.Builder().useSsl(true).sslContext(sc).build();
```

Connection Pooling

The driver supports connection pooling for VelocyStream with a default of 1 and HTTP with a default of 20 maximum connections. To change this value use the method `maxConnections(Integer)` in `ArangoDB.Builder`.

```
ArangoDB arangoDB = new ArangoDB.Builder().maxConnections(8).build();
```

The driver does not explicitly release connections. To avoid exhaustion of resources when no connection is needed, you can clear the connection pool (close all connections to the server) or use connection TTL.

```
arangoDB.shutdown();
```

Fallback hosts

The driver supports configuring multiple hosts. The first host is used to open a connection to. When this host is not reachable the next host from the list is used. To use this feature just call the method `host(String, int)` multiple times.

```
ArangoDB arangoDB = new ArangoDB.Builder().host("host1", 8529).host("host2", 8529).build();
```

Since version 4.3 the driver support acquiring a list of known hosts in a cluster setup or a single server setup with followers. For this the driver has to be able to successfully open a connection to at least one host to get the list of hosts. Then it can use this list when fallback is needed. To use this feature just pass `true` to the method `acquireHostList(boolean)`.

```
ArangoDB arangoDB = new ArangoDB.Builder().acquireHostList(true).build();
```

Load Balancing

Since version 4.3 the driver supports load balancing for cluster setups in two different ways.

The first one is a round robin load balancing where the driver iterates through a list of known hosts and performs every request on a different host than the request before. This load balancing strategy only work when the maximun of connections is greater 1.

```
ArangoDB arangoDB = new ArangoDB.Builder().loadBalancingStrategy(LoadBalancingStrategy.ROUND_ROBIN).maxConnections(8).build()
;
```

Just like the Fallback hosts feature the round robin load balancing strategy can use the `acquireHostList` configuration to acquire a list of all known hosts in the cluster. Do so only requires the manually configuration of only one host. Because this list is updated frequently it makes load balancing over the whole cluster very comfortable.

```
ArangoDB arangoDB = new ArangoDB.Builder().loadBalancingStrategy(LoadBalancingStrategy.ROUND_ROBIN).maxConnections(8).acquire
HostList(true).build();
```

The second load balancing strategy allows to pick a random host from the configured or acquired list of hosts and sticks to that host as long as the connection is open. This strategy is useful for an application - using the driver - which provides a session management where each session has its own instance of `ArangoDB` build from a global configured list of hosts. In this case it could be wanted that every sessions sticks with all its requests to the same host but not all sessions should use the same host. This load balancing strategy also works together with `acquireHostList`.

```
  ArangoDB arangoDB = new ArangoDB.Builder().loadBalancingStrategy(LoadBalancingStrategy.ONE_RANDOM).acquireHostList(true).buil
d();
```

Connection time to live

Since version 4.4 the driver supports setting a TTL for connections managed by the internal connection pool. Setting a TTL helps when using load balancing strategy `ROUND_ROBIN`, because as soon as a coordinator goes down, every open connection to that host will be closed and opened again with another target coordinator. As long as the driver does not have to open new connections (all connections in the pool are used) it will use only the coordinators which never went down. To use the downed coordinator again, when it is running again, the connections in the connection pool have to be closed and opened again with the target host mentioned by the load balancing strategy. To achieve this you can manually call `ArangoDB.shutdown` in your client code or use the TTL for connection so that a downed coordinator (which is then brought up again) will be used again after a certain time.

```
ArangoDB arango = new ArangoDB.Builder().connectionTtl(5 * 60 * 1000).build();
```

In this example all connections will be closed/reopened after 5 minutes.

Connection TTL can be disabled setting it to `null`:

```
.connectionTtl(null)
```

The default TTL is `null` (no automatic connection closure).

Manipulating databases

create database

```
// create database
arangoDB.createDatabase("myDatabase");
```

drop database

```
// drop database
arangoDB.db("myDatabase").drop();
```

Manipulating collections

create collection

```
// create collection
arangoDB.db("myDatabase").createCollection("myCollection", null);
```

drop collection

```
// delete collection
arangoDB.db("myDatabase").collection("myCollection").drop();
```

truncate collection

```
arangoDB.db("myDatabase").collection("myCollection").truncate();
```

Basic Document operations

Every document operations works with POJOs (e.g. MyObject), VelocyPack (VPackSlice) and Json (String).

For the next examples we use a small object:

```
public class MyObject {

  private String key;
  private String name;
  private int age;

  public MyObject(String name, int age) {
    this();
    this.name = name;
    this.age = age;
  }

  public MyObject() {
    super();
  }

  /*
   *  + getter and setter
   */

}
```

insert document

```
MyObject myObject = new MyObject("Homer", 38);
arangoDB.db("myDatabase").collection("myCollection").insertDocument(myObject);
```

When creating a document, the attributes of the object will be stored as key-value pair E.g. in the previous example the object was stored as follows:

```
"name" : "Homer"
"age" : "38"
```

delete document

```
arangoDB.db("myDatabase").collection("myCollection").deleteDocument(myObject.getKey());
```

update document

```
arangoDB.db("myDatabase").collection("myCollection").updateDocument(myObject.getKey(), myUpdatedObject);
```

replace document

```
arangoDB.db("myDatabase").collection("myCollection").replaceDocument(myObject.getKey(), myObject2);
```

read document as JavaBean

```
MyObject document = arangoDB.db("myDatabase").collection("myCollection").getDocument(myObject.getKey(), MyObject.class);
document.getName();
document.getAge();
```

read document as VelocyPack

```
VPackSlice document = arangoDB.db("myDatabase").collection("myCollection").getDocument(myObject.getKey(), VPackSlice.class);
document.get("name").getAsString();
document.get("age").getAsInt();
```

read document as Json

```
String json = arangoDB.db("myDatabase").collection("myCollection").getDocument(myObject.getKey(), String.class);
```

read document by key

```
arangoDB.db("myDatabase").collection("myCollection").getDocument("myKey", MyObject.class);
```

read document by id

```
arangoDB.db("myDatabase").getDocument("myCollection/myKey", MyObject.class);
```

Multi Document operations

insert documents

```
Collection<MyObject> documents = new ArrayList<>;
documents.add(myObject1);
documents.add(myObject2);
documents.add(myObject3);
arangoDB.db("myDatabase").collection("myCollection").insertDocuments(documents);
```

delete documents

```
Collection<String> keys = new ArrayList<>;
keys.add(myObject1.getKey());
keys.add(myObject2.getKey());
keys.add(myObject3.getKey());
arangoDB.db("myDatabase").collection("myCollection").deleteDocuments(keys);
```

update documents

```
Collection<MyObject> documents = new ArrayList<>;
documents.add(myObject1);
documents.add(myObject2);
documents.add(myObject3);
arangoDB.db("myDatabase").collection("myCollection").updateDocuments(documents);
```

replace documents

```
Collection<MyObject> documents = new ArrayList<>;
documents.add(myObject1);
documents.add(myObject2);
documents.add(myObject3);
arangoDB.db("myDatabase").collection("myCollection").replaceDocuments(documents);
```

AQL

Executing an AQL statement

Every AQL operations works with POJOs (e.g. MyObject), VelocyPack (VPackSlice) and Json (String).

E.g. get all Simpsons aged 3 or older in ascending order:

```
arangoDB.createDatabase("myDatabase");
ArangoDatabase db = arangoDB.db("myDatabase");

db.createCollection("myCollection");
ArangoCollection collection = db.collection("myCollection");

collection.insertDocument(new MyObject("Homer", 38));
collection.insertDocument(new MyObject("Marge", 36));
collection.insertDocument(new MyObject("Bart", 10));
collection.insertDocument(new MyObject("Lisa", 8));
collection.insertDocument(new MyObject("Maggie", 2));

Map<String, Object> bindVars = new HashMap<>();
bindVars.put("age", 3);

ArangoCursor<MyObject> cursor = db.query(query, bindVars, null, MyObject.class);

for(; cursor.hasNext;) {
  MyObject obj = cursor.next();
  System.out.println(obj.getName());
}
```

or return the AQL result as VelocyPack:

```
ArangoCursor<VPackSlice> cursor = db.query(query, bindVars, null, VPackSlice.class);

for(; cursor.hasNext;) {
  VPackSlice obj = cursor.next();
  System.out.println(obj.get("name").getAsString());
}
```

Note: The parameter `type` in `query()` has to match the result of the query, otherwise you get an VPackParserException. E.g. you set `type` to `BaseDocument` or a POJO and the query result is an array or simple type, you get an VPackParserException caused by VPackValueTypeException: Expecting type OBJECT.

Graphs

The driver supports the graph api.

Some of the basic graph operations are described in the following:

add graph

A graph consists of vertices and edges (stored in collections). Which collections are used within a graph is defined via edge definitions. A graph can contain more than one edge definition, at least one is needed.

```
Collection<EdgeDefinition> edgeDefinitions = new ArrayList<>();
EdgeDefinition edgeDefinition = new EdgeDefinition();
// define the edgeCollection to store the edges
edgeDefinition.collection("myEdgeCollection");
// define a set of collections where an edge is going out...
edgeDefinition.from("myCollection1", "myCollection2");

// repeat this for the collections where an edge is going into
edgeDefinition.to("myCollection1", "myCollection3");

edgeDefinitions.add(edgeDefinition);

// A graph can contain additional vertex collections, defined in the set of orphan collections
GraphCreateOptions options = new GraphCreateOptions();
options.orphanCollections("myCollection4", "myCollection5");

// now it's possible to create a graph
arangoDB.db("myDatabase").createGraph("myGraph", edgeDefinitions, options);
```

delete graph

A graph can be deleted by its name

```
arangoDB.db("myDatabase").graph("myGraph").drop();
```

add vertex

Vertices are stored in the vertex collections defined above.

```
MyObject myObject1 = new MyObject("Homer", 38);
MyObject myObject2 = new MyObject("Marge", 36);
arangoDB.db("myDatabase").graph("myGraph").vertexCollection("collection1").insertVertex(myObject1, null);
arangoDB.db("myDatabase").graph("myGraph").vertexCollection("collection3").insertVertex(myObject2, null);
```

add edge

Now an edge can be created to set a relation between vertices

```
arangoDB.db("myDatabase").graph("myGraph").edgeCollection("myEdgeCollection").insertEdge(myEdgeObject, null);
```

Foxx

call a service

```
Request request = new Request("mydb", RequestType.GET, "/my/foxx/service")
Response response = arangoDB.execute(request);
```

User management

If you are using authentication you can manage users with the driver.

add user

```
//username, password
arangoDB.createUser("myUser", "myPassword");
```

delete user

```
arangoDB.deleteUser("myUser");
```

list users

```
Collection<UserResult> users = arangoDB.getUsers();
for(UserResult user : users) {
  System.out.println(user.getUser())
}
```

grant user access

```
arangoDB.db("myDatabase").grantAccess("myUser");
```

revoke user access

```
arangoDB.db("myDatabase").revokeAccess("myUser");
```

Serialization

VelocyPack serialization

Since version `4.1.11` you can extend the VelocyPack serialization by registering additional `VPackModule`s on `ArangoDB.Builder`.

Java 8 types

Added support for:

- java.time.Instant
- java.time.LocalDate
- java.time.LocalDateTime
- java.util.Optional;
- java.util.OptionalDouble;
- java.util.OptionalInt;
- java.util.OptionalLong;

```
<dependencies>
  <dependency>
    <groupId>com.arangodb</groupId>
    <artifactId>velocypack-module-jdk8</artifactId>
    <version>1.0.2</version>
  </dependency>
</dependencies>
```

```
ArangoDB arangoDB = new ArangoDB.Builder().registerModule(new VPackJdk8Module()).build();
```

Scala types

Added support for:

- scala.Option
- scala.collection.immutable.List
- scala.collection.immutable.Map

```
<dependencies>
  <dependency>
    <groupId>com.arangodb</groupId>
    <artifactId>velocypack-module-scala</artifactId>
    <version>1.0.1</version>
  </dependency>
</dependencies>
```

```
val arangoDB: ArangoDB = new ArangoDB.Builder().registerModule(new VPackScalaModule).build
```

Joda-Time

Added support for:

- org.joda.time.DateTime;
- org.joda.time.Instant;
- org.joda.time.LocalDate;
- org.joda.time.LocalDateTime;

```
<dependencies>
  <dependency>
```

```
    <groupId>com.arangodb</groupId>
    <artifactId>velocypack-module-joda</artifactId>
    <version>1.0.0</version>
  </dependency>
</dependencies>
```

```
ArangoDB arangoDB = new ArangoDB.Builder().registerModule(new VPackJodaModule()).build();
```

Use of jackson as an alternative serializer

Since version 4.5.2, the driver supports alternative serializer to de-/serialize documents, edges and query results. One implementation is VelocyJack which is based on Jackson working with jackson-dataformat-velocypack.

Note: Any registered custom serializer/deserializer or module will be ignored.

custom serialization

```
ArangoDB arangoDB = new ArangoDB.Builder().registerModule(new VPackModule() {
  @Override
  public <C extends VPackSetupContext<C>> void setup(final C context) {
    context.registerDeserializer(MyObject.class, new VPackDeserializer<MyObject>() {
      @Override
      public MyObject deserialize(VPackSlice parent,VPackSlice vpack,
          VPackDeserializationContext context) throws VPackException {
        MyObject obj = new MyObject();
        obj.setName(vpack.get("name").getAsString());
        return obj;
      }
    });
    context.registerSerializer(MyObject.class, new VPackSerializer<MyObject>() {
      @Override
      public void serialize(VPackBuilder builder,String attribute,MyObject value,
          VPackSerializationContext context) throws VPackException {
        builder.add(attribute, ValueType.OBJECT);
        builder.add("name", value.getName());
        builder.close();
      }
    });
  }
}).build();
```

JavaBeans

The driver can serialize/deserialize JavaBeans. They need at least a constructor without parameter.

```
public class MyObject {

  private String name;
  private Gender gender;
  private int age;

  public MyObject() {
    super();
  }

}
```

internal fields

To use Arango-internal fields (like _id, _key, _rev, _from, _to) in your JavaBeans, use the annotation `DocumentField`.

```
public class MyObject {

  @DocumentField(Type.KEY)
  private String key;

  private String name;
  private Gender gender;
  private int age;

  public MyObject() {
    super();
  }

}
```

serialized fieldnames

To use a different serialized name for a field, use the annotation `SerializedName`.

```
public class MyObject {

  @SerializedName("title")
  private String name;

  private Gender gender;
  private int age;

  public MyObject() {
    super();
  }

}
```

ignore fields

To ignore fields at serialization/deserialization, use the annotation `Expose`

```
public class MyObject {

  @Expose
  private String name;
  @Expose(serialize = true, deserialize = false)
  private Gender gender;
  private int age;

  public MyObject() {
    super();
  }

}
```

custom serializer

```
ArangoDB arangoDB = new ArangoDB.Builder().registerModule(new VPackModule() {
  @Override
  public <C extends VPackSetupContext<C>> void setup(final C context) {
    context.registerDeserializer(MyObject.class, new VPackDeserializer<MyObject>() {
      @Override
      public MyObject deserialize(VPackSlice parent,VPackSlice vpack,
          VPackDeserializationContext context) throws VPackException {
        MyObject obj = new MyObject();
        obj.setName(vpack.get("name").getAsString());
        return obj;
      }
    });
```

```
        context.registerSerializer(MyObject.class, new VPackSerializer<MyObject>() {
          @Override
          public void serialize(VPackBuilder builder,String attribute,MyObject value,
              VPackSerializationContext context) throws VPackException {
            builder.add(attribute, ValueType.OBJECT);
            builder.add("name", value.getName());
            builder.close();
          }
        });
      }
    }).build();
```

manually serialization

To de-/serialize from and to VelocyPack before or after a database call, use the `ArangoUtil` from the method `util()` in `ArangoDB`, `ArangoDatabase`, `ArangoCollection`, `ArangoGraph`, `ArangoEdgeCollection` or `ArangoVertexCollection`.

```
ArangoDB arangoDB = new ArangoDB.Builder();
VPackSlice vpack = arangoDB.util().serialize(myObj);
```

```
ArangoDB arangoDB = new ArangoDB.Builder();
MyObject myObj = arangoDB.util().deserialize(vpack, MyObject.class);
```

ArangoDB JavaScript Driver

The official ArangoDB low-level JavaScript client.

Note: if you are looking for the ArangoDB JavaScript API in Foxx (or the `arangosh` interactive shell) please refer to the documentation about the `@arangodb` module instead; specifically the `db` object exported by the `@arangodb` module. The JavaScript driver is **only** meant to be used when accessing ArangoDB from **outside** the database.

- Getting Started
- Reference
- Changelog

ArangoDB JavaScript Driver - Getting Started

Compatibility

ArangoJS is compatible with ArangoDB 3.0 and later. **For using ArangoJS with 2.8 or earlier see the upgrade note below.** ArangoJS is tested against the two most-recent releases of ArangoDB 3 (currently 3.2 and 3.3) as well as the most recent version of 2.8 and the latest development version.

The yarn/npm distribution of ArangoJS is compatible with Node.js versions 9.x (latest), 8.x (LTS) and 6.x (LTS). Node.js version support follows the official Node.js long-term support schedule.

The included browser build is compatible with Internet Explorer 11 and recent versions of all modern browsers (Edge, Chrome, Firefox and Safari).

Versions outside this range may be compatible but are not actively supported.

Upgrade note: If you want to use arangojs with ArangoDB 2.8 or earlier remember to set the appropriate `arangoVersion` option (e.g. `20800` for version 2.8.0). The current default value is `30000` (indicating compatibility with version 3.0.0 and newer). **The driver will behave differently depending on this value when using APIs that have changed between these versions.**

Upgrade note for 6.0.0: All asynchronous functions now return promises and support for node-style callbacks has been removed. If you are using a version of Node.js older than Node.js 6.x LTS ("Boron") make sure you replace the native `Promise` implementation with a substitute like bluebird to avoid a known memory leak in older versions of the V8 JavaScript engine.

Versions

The version number of this driver does not indicate supported ArangoDB versions!

This driver uses semantic versioning:

- A change in the bugfix version (e.g. X.Y.0 -> X.Y.1) indicates internal changes and should always be safe to upgrade.
- A change in the minor version (e.g. X.1.Z -> X.2.0) indicates additions and backwards-compatible changes that should not affect your code.
- A change in the major version (e.g. 1.Y.Z -> 2.0.0) indicates *breaking* changes that require changes in your code to upgrade.

If you are getting weird errors or functions seem to be missing, make sure you are using the latest version of the driver and following documentation written for a compatible version. If you are following a tutorial written for an older version of arangojs, you can install that version using the `<name>@<version>` syntax:

```
# for version 4.x.x
yarn add arangojs@4
# - or -
npm install --save arangojs@4
```

You can find the documentation for each version by clicking on the corresponding date on the left in the list of version tags.

Testing

Run the tests using the `yarn test` or `npm test` commands:

```
yarn test
# - or -
npm test
```

By default the tests will be run against a server listening on `http://localhost:8529` (using username `root` with no password). To override this, you can set the environment variable `TEST_ARANGODB_URL` to something different:

```
TEST_ARANGODB_URL=http://myserver.local:8530 yarn test
# - or -
TEST_ARANGODB_URL=http://myserver.local:8530 npm test
```

Install

With Yarn or NPM

```
yarn add arangojs
# - or -
npm install --save arangojs
```

With Bower

Starting with arangojs 6.0.0 Bower is no longer supported and the browser build is now included in the NPM release (see below).

From source

```
git clone https://github.com/arangodb/arangojs.git
cd arangojs
npm install
npm run dist
```

For browsers

For production use arangojs can be installed with Yarn or NPM like any other dependency. Just use arangojs like you would in your server code:

```
import { Database } from "arangojs";
// -- or --
var arangojs = require("arangojs");
```

Additionally the NPM release comes with a precompiled browser build:

```
var arangojs = require("arangojs/lib/web");
```

You can also use unpkg during development:

```
< !-- note the path includes the version number (e.g. 6.0.0) -- >
<script src="https://unpkg.com/arangojs@6.0.0/lib/web.js"></script>
<script>
var db = new arangojs.Database();
db.listCollections().then(function (collections) {
  alert("Your collections: " + collections.map(function (collection) {
    return collection.name;
  }).join(", "));
});
</script>
```

If you are targetting browsers older than Internet Explorer 11 you may want to use babel with a polyfill to provide missing functionality needed to use arangojs.

When loading the browser build with a script tag make sure to load the polyfill first:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-polyfill/6.26.0/polyfill.js"></script>
<script src="https://unpkg.com/arangojs@6.0.0/lib/web.js"></script>
```

Basic usage example

```
// Modern JavaScript
import { Database, aql } from "arangojs";
const db = new Database();
(async function() {
  const now = Date.now();
  try {
    const cursor = await db.query(aql`
      RETURN ${now}
    `);
    const result = await cursor.next();
    // ...
  } catch (err) {
    // ...
  }
})();

// or plain old Node-style
var arangojs = require("arangojs");
var db = new arangojs.Database();
var now = Date.now();
db
  .query({
    query: "RETURN @value",
    bindVars: { value: now }
  })
  .then(function(cursor) {
    return cursor.next().then(function(result) {
      // ...
    });
  })
  .catch(function(err) {
    // ...
  });

// Using different databases
const db = new Database({
  url: "http://localhost:8529"
});
db.useDatabase("pancakes");
db.useBasicAuth("root", "");
// The database can be swapped at any time
db.useDatabase("waffles");
db.useBasicAuth("admin", "maplesyrup");

// Using ArangoDB behind a reverse proxy
const db = new Database({
  url: "http://myproxy.local:8000",
  isAbsolute: true // don't automatically append database path to URL
});

// Trigger ArangoDB 2.8 compatibility mode
const db = new Database({
  arangoVersion: 20800
});
```

For AQL please check out the aql template tag for writing parametrized AQL queries without making your code vulnerable to injection attacks.

Error responses

If arangojs encounters an API error, it will throw an *ArangoError* with an *errorNum* as defined in the ArangoDB documentation as well as a *code* and *statusCode* property indicating the intended and actual HTTP status code of the response.

For any other error responses (4xx/5xx status code), it will throw an *HttpError* error with the status code indicated by the *code* and *statusCode* properties.

If the server response did not indicate an error but the response body could not be parsed, a *SyntaxError* may be thrown instead.

In all of these cases the error object will additionally have a *response* property containing the server response object.

If the request failed at a network level or the connection was closed without receiving a response, the underlying error will be thrown instead.

Examples

```
// Using async/await
try {
  const info = await db.createDatabase('mydb');
  // database created
} catch (err) {
  console.error(err.stack);
}

// Using promises with arrow functions
db.createDatabase('mydb')
.then(
  info => {
    // database created
  },
  err => console.error(err.stack)
);
```

Note: the examples in the remainder of this documentation use async/await and other modern language features like multi-line strings and template tags. When developing for an environment without support for these language features, just use promises instead as in the above example.

ArangoDB JavaScript Driver - Reference

Database API

new Database

`new Database([config]): Database`

Creates a new *Database* instance.

If *config* is a string, it will be interpreted as *config.url*.

Arguments

- **config**: `Object` (optional)

  An object with the following properties:

  - **url**: `string | Array<string>` (Default: `http://localhost:8529`)

    Base URL of the ArangoDB server or list of server URLs.

    Note: As of arangojs 6.0.0 it is no longer possible to pass the username or password from the URL.

    If you want to use ArangoDB with authentication, see *useBasicAuth* or *useBearerAuth* methods.

    If you need to support self-signed HTTPS certificates, you may have to add your certificates to the *agentOptions*, e.g.:

    ```
    agentOptions: {
      ca: [
        fs.readFileSync(".ssl/sub.class1.server.ca.pem"),
        fs.readFileSync(".ssl/ca.pem")
      ];
    }
    ```

  - **isAbsolute**: `boolean` (Default: `false`)

    If this option is explicitly set to `true`, the *url* will be treated as the absolute database path. This is an escape hatch to allow using arangojs with database APIs exposed with a reverse proxy and makes it impossible to switch databases with *useDatabase* or using *acquireHostList*.

  - **arangoVersion**: `number` (Default: `30000`)

    Value of the `x-arango-version` header. This should match the lowest version of ArangoDB you expect to be using. The format is defined as `XYYZZ` where `X` is the major version, `Y` is the two-digit minor version and `Z` is the two-digit bugfix version.

    Example: `30102` corresponds to version 3.1.2 of ArangoDB.

    Note: The driver will behave differently when using different major versions of ArangoDB to compensate for API changes. Some functions are not available on every major version of ArangoDB as indicated in their descriptions below (e.g. *collection.first*, *collection.bulkUpdate*).

  - **headers**: `Object` (optional)

    An object with additional headers to send with every request.

    Header names should always be lowercase. If an `"authorization"` header is provided, it will be overridden when using *useBasicAuth* or *useBearerAuth*.

  - **agent**: `Agent` (optional)

    An http Agent instance to use for connections.

    By default a new `http.Agent` (or https.Agent) instance will be created using the *agentOptions*.

    This option has no effect when using the browser version of arangojs.

  - **agentOptions**: `Object` (Default: see below)

An object with options for the agent. This will be ignored if *agent* is also provided.

Default: `{maxSockets: 3, keepAlive: true, keepAliveMsecs: 1000}`. Browser default: `{maxSockets: 3, keepAlive: false}`;

The option `maxSockets` can also be used to limit how many requests arangojs will perform concurrently. The maximum number of requests is equal to `maxSockets * 2` with `keepAlive: true` or equal to `maxSockets` with `keepAlive: false`.

In the browser version of arangojs this option can be used to pass additional options to the underlying calls of the `xhr` module.

- **loadBalancingStrategy**: `string` (Default: `"NONE"`)

  Determines the behaviour when multiple URLs are provided:

  - `NONE`: No load balancing. All requests will be handled by the first URL in the list until a network error is encountered. On network error, arangojs will advance to using the next URL in the list.
  - `ONE_RANDOM`: Randomly picks one URL from the list initially, then behaves like `NONE`.
  - `ROUND_ROBIN`: Every sequential request uses the next URL in the list.

Manipulating databases

These functions implement the HTTP API for manipulating databases.

database.acquireHostList

`async database.acquireHostList(): this`

Updates the URL list by requesting a list of all coordinators in the cluster and adding any endpoints not initially specified in the *url* configuration.

For long-running processes communicating with an ArangoDB cluster it is recommended to run this method repeatedly (e.g. once per hour) to make sure new coordinators are picked up correctly and can be used for fail-over or load balancing.

Note: This method can not be used when the arangojs instance was created with `isAbsolute: true`.

database.useDatabase

`database.useDatabase(databaseName): this`

Updates the *Database* instance and its connection string to use the given *databaseName*, then returns itself.

Note: This method can not be used when the arangojs instance was created with `isAbsolute: true`.

Arguments

- **databaseName**: `string`

  The name of the database to use.

Examples

```
const db = new Database();
db.useDatabase("test");
// The database instance now uses the database "test".
```

database.useBasicAuth

`database.useBasicAuth([username, [password]]): this`

Updates the *Database* instance's `authorization` header to use Basic authentication with the given *username* and *password*, then returns itself.

Arguments

- **username**: `string` (Default: `"root"`)

  The username to authenticate with.

- **password**: `string` (Default: `""`)

  The password to authenticate with.

Examples

```
const db = new Database();
db.useDatabase("test");
db.useBasicAuth("admin", "hunter2");
// The database instance now uses the database "test"
// with the username "admin" and password "hunter2".
```

database.useBearerAuth

`database.useBearerAuth(token): this`

Updates the *Database* instance's `authorization` header to use Bearer authentication with the given authentication token, then returns itself.

Arguments

- **token**: `string`

  The token to authenticate with.

Examples

```
const db = new Database();
db.useBearerAuth("keyboardcat");
// The database instance now uses Bearer authentication.
```

database.login

`async database.login([username, [password]]): string`

Validates the given database credentials and exchanges them for an authentication token, then uses the authentication token for future requests and returns it.

Arguments

- **username**: `string` (Default: `"root"`)

  The username to authenticate with.

- **password**: `string` (Default: `""`)

  The password to authenticate with.

Examples

```
const db = new Database();
db.useDatabase("test");
await db.login("admin", "hunter2");
// The database instance now uses the database "test"
// with an authentication token for the "admin" user.
```

database.version

`async database.version(): Object`

Fetches the ArangoDB version information for the active database from the server.

Examples

```
const db = new Database();
const version = await db.version();
// the version object contains the ArangoDB version information.
```

database.createDatabase

`async database.createDatabase(databaseName, [users]): Object`

Creates a new database with the given *databaseName*.

Arguments

- **databaseName**: `string`

  Name of the database to create.

- **users**: `Array<Object>` (optional)

  If specified, the array must contain objects with the following properties:

  - **username**: `string`

    The username of the user to create for the database.

  - **passwd**: `string` (Default: empty)

    The password of the user.

  - **active**: `boolean` (Default: `true`)

    Whether the user is active.

  - **extra**: `Object` (optional)

    An object containing additional user data.

Examples

```
const db = new Database();
const info = await db.createDatabase('mydb', [{username: 'root'}]);
// the database has been created
```

database.exists

`async database.exists(): boolean`

Checks whether the database exists.

Examples

```
const db = new Database();
const result = await db.exists();
// result indicates whether the database exists
```

database.get

`async database.get(): Object`

Fetches the database description for the active database from the server.

Examples

```
const db = new Database();
const info = await db.get();
// the database exists
```

database.listDatabases

`async database.listDatabases(): Array<string>`

Fetches all databases from the server and returns an array of their names.

Examples

```
const db = new Database();
const names = await db.listDatabases();
// databases is an array of database names
```

database.listUserDatabases

`async database.listUserDatabases(): Array<string>`

Fetches all databases accessible to the active user from the server and returns an array of their names.

Examples

```
const db = new Database();
const names = await db.listUserDatabases();
// databases is an array of database names
```

database.dropDatabase

`async database.dropDatabase(databaseName): Object`

Deletes the database with the given *databaseName* from the server.

```
const db = new Database();
await db.dropDatabase('mydb');
// database "mydb" no longer exists
```

database.truncate

`async database.truncate([excludeSystem]): Object`

Deletes **all documents in all collections** in the active database.

Arguments

- **excludeSystem**: `boolean` (Default: `true`)

  Whether system collections should be excluded. Note that this option will be ignored because truncating system collections is not supported anymore for some system collections.

Examples

```
const db = new Database();

await db.truncate();
// all non-system collections in this database are now empty
```

Accessing collections

These functions implement the HTTP API for accessing collections.

database.collection

`database.collection(collectionName): DocumentCollection`

Returns a *DocumentCollection* instance for the given collection name.

Arguments

- **collectionName**: `string`

  Name of the edge collection.

Examples

```
const db = new Database();
const collection = db.collection("potatos");
```

database.edgeCollection

`database.edgeCollection(collectionName): EdgeCollection`

Returns an *EdgeCollection* instance for the given collection name.

Arguments

- **collectionName**: `string`

  Name of the edge collection.

Examples

```
const db = new Database();
const collection = db.edgeCollection("potatos");
```

database.listCollections

`async database.listCollections([excludeSystem]): Array<Object>`

Fetches all collections from the database and returns an array of collection descriptions.

Arguments

- **excludeSystem**: `boolean` (Default: `true`)

  Whether system collections should be excluded.

Examples

```
const db = new Database();

const collections = await db.listCollections();
// collections is an array of collection descriptions
// not including system collections

// -- or --

const collections = await db.listCollections(false);
// collections is an array of collection descriptions
// including system collections
```

database.collections

`async database.collections([excludeSystem]): Array<Collection>`

Fetches all collections from the database and returns an array of *DocumentCollection* and *EdgeCollection* instances for the collections.

Arguments

- **excludeSystem**: `boolean` (Default: `true`)

  Whether system collections should be excluded.

Examples

```
const db = new Database();

const collections = await db.collections()
// collections is an array of DocumentCollection
// and EdgeCollection instances
// not including system collections

// -- or --

const collections = await db.collections(false)
// collections is an array of DocumentCollection
// and EdgeCollection instances
// including system collections
```

Queries

This function implements the HTTP API for single roundtrip AQL queries.

For collection-specific queries see simple queries.

database.query

`async database.query(query, [bindVars,] [opts]): Cursor`

Performs a database query using the given *query* and *bindVars*, then returns a new *Cursor* instance for the result list.

Arguments

- **query**: `string`

  An AQL query string or a query builder instance.

- **bindVars**: `Object` (optional)

  An object defining the variables to bind the query to.

- **opts**: `Object` (optional)

  Additional parameter object that will be passed to the query API. Possible keys are *count* and *options* (explained below)

If *opts.count* is set to `true`, the cursor will have a *count* property set to the query result count. Possible key options in *opts.options* include: *failOnWarning*, *cache*, profile or *skipInaccessibleCollections*. For a complete list of query settings please reference the arangodb.com documentation.

If *query* is an object with *query* and *bindVars* properties, those will be used as the values of the respective arguments instead.

Examples

```
const db = new Database();
const active = true;

// Using the aql template tag
const cursor = await db.query(aql`
  FOR u IN _users
  FILTER u.authData.active == ${active}
  RETURN u.user
`);
// cursor is a cursor for the query result

// -- or --

// Old-school JS with explicit bindVars:
db.query(
  'FOR u IN _users ' +
  'FILTER u.authData.active == @active ' +
  'RETURN u.user',
  {active: true}
).then(function (cursor) {
  // cursor is a cursor for the query result
});
```

aql

`aql(strings, ...args): Object`

Template string handler (aka template tag) for AQL queries. Converts a template string to an object that can be passed to `database.query` by converting arguments to bind variables.

Note: If you want to pass a collection name as a bind variable, you need to pass a *Collection* instance (e.g. what you get by passing the collection name to `db.collection`) instead. If you see the error `"array expected as operand to FOR loop"`, you're likely passing a collection name instead of a collection instance.

Examples

```
const userCollection = db.collection("_users");
const role = "admin";

const query = aql`
  FOR user IN ${userCollection}
  FILTER user.role == ${role}
  RETURN user
`;

// -- is equivalent to --
const query = {
  query: "FOR user IN @@value0 FILTER user.role == @value1 RETURN user",
  bindVars: { "@value0": userCollection.name, value1: role }
};
```

Note how the aql template tag automatically handles collection references (`@@value0` instead of `@value0`) for us so you don't have to worry about counting at-symbols.

Because the aql template tag creates actual bindVars instead of inlining values directly, it also avoids injection attacks via malicious parameters:

```
// malicious user input
const email = '" || (FOR x IN secrets REMOVE x IN secrets) || "';

// DON'T do this!
const query = `
  FOR user IN users
  FILTER user.email == "${email}"
  RETURN user
`;
// FILTER user.email == "" || (FOR x IN secrets REMOVE x IN secrets) || ""

// instead do this!
const query = aql`
  FOR user IN users
  FILTER user.email == ${email}
  RETURN user
`;
// FILTER user.email == @value0
```

Managing AQL user functions

These functions implement the HTTP API for managing AQL user functions.

database.listFunctions

`async database.listFunctions(): Array<Object>`

Fetches a list of all AQL user functions registered with the database.

Examples

```
const db = new Database();
const functions = db.listFunctions();
// functions is a list of function descriptions
```

database.createFunction

`async database.createFunction(name, code): Object`

Creates an AQL user function with the given *name* and *code* if it does not already exist or replaces it if a function with the same name already existed.

Arguments

- **name**: `string`

  A valid AQL function name, e.g.: `"myfuncs::accounting::calculate_vat"` .

- **code**: `string`

  A string evaluating to a JavaScript function (not a JavaScript function object).

Examples

```
const db = new Database();
await db.createFunction(
  'ACME::ACCOUNTING::CALCULATE_VAT',
  String(function (price) {
    return price * 0.19;
  })
);
// Use the new function in an AQL query with template handler:
const cursor = await db.query(aql`
  FOR product IN products
  RETURN MERGE(
    {vat: ACME::ACCOUNTING::CALCULATE_VAT(product.price)},
    product
  )
`);
// cursor is a cursor for the query result
```

database.dropFunction

`async database.dropFunction(name, [group]): Object`

Deletes the AQL user function with the given name from the database.

Arguments

- **name**: `string`

  The name of the user function to drop.

- **group**: `boolean` (Default: `false`)

  If set to `true`, all functions with a name starting with *name* will be deleted; otherwise only the function with the exact name will be deleted.

Examples

```
const db = new Database();
await db.dropFunction('ACME::ACCOUNTING::CALCULATE_VAT');
// the function no longer exists
```

Transactions

This function implements the HTTP API for transactions.

database.transaction

`async database.transaction(collections, action, [params, [options]]): Object`

Performs a server-side transaction and returns its return value.

Arguments

- **collections**: `Object`

  An object with the following properties:

  - **read**: `Array<string>` (optional)

    An array of names (or a single name) of collections that will be read from during the transaction.

  - **write**: `Array<string>` (optional)

    An array of names (or a single name) of collections that will be written to or read from during the transaction.

- **action**: `string`

  A string evaluating to a JavaScript function to be executed on the server.

  Note: For accessing the database from within ArangoDB, see the documentation for the `@arangodb` module in ArangoDB.

- **params**: `Object` (optional)

  Available as variable `params` when the *action* function is being executed on server. Check the example below.

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **lockTimeout**: `number` (optional)

    Determines how long the database will wait while attempting to gain locks on collections used by the transaction before timing out.

  - **waitForSync**: `boolean` (optional)

    Determines whether to force the transaction to write all data to disk before returning.

  - **maxTransactionSize**: `number` (optional)

    Determines the transaction size limit in bytes. Honored by the RocksDB storage engine only.

  - **intermediateCommitCount**: `number` (optional)

    Determines the maximum number of operations after which an intermediate commit is performed automatically. Honored by the RocksDB storage engine only.

  - **intermediateCommitSize**: `number` (optional)

    Determine the maximum total size of operations after which an intermediate commit is performed automatically. Honored by the RocksDB storage engine only.

If *collections* is an array or string, it will be treated as *collections.write*.

Please note that while *action* should be a string evaluating to a well-formed JavaScript function, it's not possible to pass in a JavaScript function directly because the function needs to be evaluated on the server and will be transmitted in plain text.

For more information on transactions, see the HTTP API documentation for transactions.

Examples

```
const db = new Database();
const action = String(function (params) {
  // This code will be executed inside ArangoDB!
  const db = require('@arangodb').db;
  return db._query(aql`
    FOR user IN _users
    FILTER user.age > ${params.age}
    RETURN u.user
  `).toArray();
});

const result = await db.transaction(
  {read: '_users'},
  action,
  {age: 12}
);
// result contains the return value of the action
```

Accessing graphs

These functions implement the HTTP API for accessing general graphs.

database.graph

`database.graph(graphName): Graph`

Returns a *Graph* instance representing the graph with the given graph name.

database.listGraphs

`async database.listGraphs(): Array<Object>`

Fetches all graphs from the database and returns an array of graph descriptions.

Examples

```
const db = new Database();
const graphs = await db.listGraphs();
// graphs is an array of graph descriptions
```

database.graphs

`async database.graphs(): Array<Graph>`

Fetches all graphs from the database and returns an array of *Graph* instances for the graphs.

Examples

```
const db = new Database();
const graphs = await db.graphs();
// graphs is an array of Graph instances
```

Managing Foxx services

database.listServices

`async database.listServices([excludeSystem]): Array<Object>`

Fetches a list of all installed service.

Arguments

- **excludeSystem**: `boolean` (Default: `true`)

  Whether system services should be excluded.

Examples

```
const services = await db.listServices();

// -- or --

const services = await db.listServices(false);
```

database.installService

`async database.installService(mount, source, [options]): Object`

Installs a new service.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **source**: `Buffer | Readable | File | string`

  The service bundle to install.

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **configuration**: `Object` (optional)

    An object mapping configuration option names to values.

  - **dependencies**: `Object` (optional)

    An object mapping dependency aliases to mount points.

  - **development**: `boolean` (Default: `false`)

    Whether the service should be installed in development mode.

  - **legacy**: `boolean` (Default: `false`)

    Whether the service should be installed in legacy compatibility mode.

    This overrides the `engines` option in the service manifest (if any).

  - **setup**: `boolean` (Default: `true`)

    Whether the setup script should be executed.

Examples

```
const source = fs.createReadStream('./my-foxx-service.zip');
const info = await db.installService('/hello', source);

// -- or --

const source = fs.readFileSync('./my-foxx-service.zip');
const info = await db.installService('/hello', source);

// -- or --

const element = document.getElementById('my-file-input');
const source = element.files[0];
const info = await db.installService('/hello', source);
```

database.replaceService

`async database.replaceService(mount, source, [options]): Object`

Replaces an existing service with a new service by completely removing the old service and installing a new service at the same mount point.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **source**: `Buffer | Readable | File | string`

  The service bundle to replace the existing service with.

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **configuration**: `Object` (optional)

    An object mapping configuration option names to values.

    This configuration will replace the existing configuration.

  - **dependencies**: `Object` (optional)

    An object mapping dependency aliases to mount points.

    These dependencies will replace the existing dependencies.

  - **development**: `boolean` (Default: `false`)

    Whether the new service should be installed in development mode.

  - **legacy**: `boolean` (Default: `false`)

    Whether the new service should be installed in legacy compatibility mode.

    This overrides the `engines` option in the service manifest (if any).

  - **teardown**: `boolean` (Default: `true`)

    Whether the teardown script of the old service should be executed.

  - **setup**: `boolean` (Default: `true`)

    Whether the setup script of the new service should be executed.

Examples

```
const source = fs.createReadStream('./my-foxx-service.zip');
const info = await db.replaceService('/hello', source);

// -- or --
```

```
const source = fs.readFileSync('./my-foxx-service.zip');
const info = await db.replaceService('/hello', source);

// -- or --

const element = document.getElementById('my-file-input');
const source = element.files[0];
const info = await db.replaceService('/hello', source);
```

database.upgradeService

`async database.upgradeService(mount, source, [options]): Object`

Replaces an existing service with a new service while retaining the old service's configuration and dependencies.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **source**: `Buffer | Readable | File | string`

  The service bundle to replace the existing service with.

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **configuration**: `Object` (optional)

    An object mapping configuration option names to values.

    This configuration will be merged into the existing configuration.

  - **dependencies**: `Object` (optional)

    An object mapping dependency aliases to mount points.

    These dependencies will be merged into the existing dependencies.

  - **development**: `boolean` (Default: `false`)

    Whether the new service should be installed in development mode.

  - **legacy**: `boolean` (Default: `false`)

    Whether the new service should be installed in legacy compatibility mode.

    This overrides the `engines` option in the service manifest (if any).

  - **teardown**: `boolean` (Default: `false`)

    Whether the teardown script of the old service should be executed.

  - **setup**: `boolean` (Default: `true`)

    Whether the setup script of the new service should be executed.

Examples

```
const source = fs.createReadStream('./my-foxx-service.zip');
const info = await db.upgradeService('/hello', source);

// -- or --

const source = fs.readFileSync('./my-foxx-service.zip');
const info = await db.upgradeService('/hello', source);

// -- or --
```

```
const element = document.getElementById('my-file-input');
const source = element.files[0];
const info = await db.upgradeService('/hello', source);
```

database.uninstallService

`async database.uninstallService(mount, [options]): void`

Completely removes a service from the database.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **teardown**: `boolean` (Default: `true`)

    Whether the teardown script should be executed.

Examples

```
await db.uninstallService('/my-service');
// service was uninstalled
```

database.getService

`async database.getService(mount): Object`

Retrieves information about a mounted service.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const info = await db.getService('/my-service');
// info contains detailed information about the service
```

database.getServiceConfiguration

`async database.getServiceConfiguration(mount, [minimal]): Object`

Retrieves an object with information about the service's configuration options and their current values.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **minimal**: `boolean` (Default: `false`)

  Only return the current values.

Examples

```
const config = await db.getServiceConfiguration('/my-service');
// config contains information about the service's configuration
```

database.replaceServiceConfiguration

`async database.replaceServiceConfiguration(mount, configuration, [minimal]): Object`

Replaces the configuration of the given service.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **configuration**: `Object`

  An object mapping configuration option names to values.

- **minimal**: `boolean` (Default: `false`)

  Only return the current values and warnings (if any).

  Note: when using ArangoDB 3.2.8 or older, enabling this option avoids triggering a second request to the database.

Examples

```
const config = {currency: 'USD', locale: 'en-US'};
const info = await db.replaceServiceConfiguration('/my-service', config);
// info.values contains information about the service's configuration
// info.warnings contains any validation errors for the configuration
```

database.updateServiceConfiguration

`async database.updateServiceConfiguration(mount, configuration, [minimal]): Object`

Updates the configuration of the given service my merging the new values into the existing ones.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **configuration**: `Object`

  An object mapping configuration option names to values.

- **minimal**: `boolean` (Default: `false`)

  Only return the current values and warnings (if any).

  Note: when using ArangoDB 3.2.8 or older, enabling this option avoids triggering a second request to the database.

Examples

```
const config = {locale: 'en-US'};
const info = await db.updateServiceConfiguration('/my-service', config);
// info.values contains information about the service's configuration
// info.warnings contains any validation errors for the configuration
```

database.getServiceDependencies

`async database.getServiceDependencies(mount, [minimal]): Object`

Retrieves an object with information about the service's dependencies and their current mount points.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **minimal**: `boolean` (Default: `false`)

  Only return the current values and warnings (if any).

Examples

```
const deps = await db.getServiceDependencies('/my-service');
// deps contains information about the service's dependencies
```

database.replaceServiceDependencies

`async database.replaceServiceDependencies(mount, dependencies, [minimal]): Object`

Replaces the dependencies for the given service.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **dependencies**: `Object`

  An object mapping dependency aliases to mount points.

- **minimal**: `boolean` (Default: `false`)

  Only return the current values and warnings (if any).

  Note: when using ArangoDB 3.2.8 or older, enabling this option avoids triggering a second request to the database.

Examples

```
const deps = {mailer: '/mailer-api', auth: '/remote-auth'};
const info = await db.replaceServiceDependencies('/my-service', deps);
// info.values contains information about the service's dependencies
// info.warnings contains any validation errors for the dependencies
```

database.updateServiceDependencies

`async database.updateServiceDependencies(mount, dependencies, [minimal]): Object`

Updates the dependencies for the given service by merging the new values into the existing ones.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **dependencies**: `Object`

  An object mapping dependency aliases to mount points.

- **minimal**: `boolean` (Default: `false`)

  Only return the current values and warnings (if any).

  Note: when using ArangoDB 3.2.8 or older, enabling this option avoids triggering a second request to the database.

Examples

```
const deps = {mailer: '/mailer-api'};
const info = await db.updateServiceDependencies('/my-service', deps);
```

```
// info.values contains information about the service's dependencies
// info.warnings contains any validation errors for the dependencies
```

database.enableServiceDevelopmentMode

`async database.enableServiceDevelopmentMode(mount): Object`

Enables development mode for the given service.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const info = await db.enableServiceDevelopmentMode('/my-service');
// the service is now in development mode
// info contains detailed information about the service
```

database.disableServiceDevelopmentMode

`async database.disableServiceDevelopmentMode(mount): Object`

Disabled development mode for the given service and commits the service state to the database.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const info = await db.disableServiceDevelopmentMode('/my-service');
// the service is now in production mode
// info contains detailed information about the service
```

database.listServiceScripts

`async database.listServiceScripts(mount): Object`

Retrieves a list of the service's scripts.

Returns an object mapping each name to a more readable representation.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const scripts = await db.listServiceScripts('/my-service');
// scripts is an object listing the service scripts
```

database.runServiceScript

`async database.runServiceScript(mount, name, [scriptArg]): any`

Runs a service script and returns the result.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

- **name**: `string`

  Name of the script to execute.

- **scriptArg**: `any`

  Value that will be passed as an argument to the script.

Examples

```
const result = await db.runServiceScript('/my-service', 'setup');
// result contains the script's exports (if any)
```

database.runServiceTests

`async database.runServiceTests(mount, [reporter]): any`

Runs the tests of a given service and returns a formatted report.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database

- **options**: `Object` (optional)

  An object with any of the following properties:

  - **reporter**: `string` (Default: `default`)

    The reporter to use to process the test results.

    As of ArangoDB 3.2 the following reporters are supported:

    - **stream**: an array of event objects
    - **suite**: nested suite objects with test results
    - **xunit**: JSONML representation of an XUnit report
    - **tap**: an array of TAP event strings
    - **default**: an array of test results
  - **idiomatic**: `boolean` (Default: `false`)

    Whether the results should be converted to the apropriate `string` representation:

    - **xunit** reports will be formatted as XML documents
    - **tap** reports will be formatted as TAP streams
    - **stream** reports will be formatted as JSON-LD streams

Examples

```
const opts = {reporter: 'xunit', idiomatic: true};
const result = await db.runServiceTests('/my-service', opts);
// result contains the XUnit report as a string
```

database.downloadService

`async database.downloadService(mount): Buffer | Blob`

Retrieves a zip bundle containing the service files.

Returns a `Buffer` in Node or `Blob` in the browser version.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const bundle = await db.downloadService('/my-service');
// bundle is a Buffer/Blob of the service bundle
```

database.getServiceReadme

`async database.getServiceReadme(mount): string?`

Retrieves the text content of the service's `README` or `README.md` file.

Returns `undefined` if no such file could be found.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const readme = await db.getServiceReadme('/my-service');
// readme is a string containing the service README's
// text content, or undefined if no README exists
```

database.getServiceDocumentation

`async database.getServiceDocumentation(mount): Object`

Retrieves a Swagger API description object for the service installed at the given mount point.

Arguments

- **mount**: `string`

  The service's mount point, relative to the database.

Examples

```
const spec = await db.getServiceDocumentation('/my-service');
// spec is a Swagger API description of the service
```

database.commitLocalServiceState

`async database.commitLocalServiceState([replace]): void`

Writes all locally available services to the database and updates any service bundles missing in the database.

Arguments

- **replace**: `boolean` (Default: `false`)

  Also commit outdated services.

  This can be used to solve some consistency problems when service bundles are missing in the database or were deleted manually.

Examples

```
await db.commitLocalServiceState();
// all services available on the coordinator have been written to the db

// -- or --

await db.commitLocalServiceState(true);
// all service conflicts have been resolved in favor of this coordinator
```

Arbitrary HTTP routes

database.route

`database.route([path,] [headers]): Route`

Returns a new *Route* instance for the given path (relative to the database) that can be used to perform arbitrary HTTP requests.

Arguments

- **path**: `string` (optional)

  The database-relative URL of the route.

- **headers**: `Object` (optional)

  Default headers that should be sent with each request to the route.

If *path* is missing, the route will refer to the base URL of the database.

For more information on *Route* instances see the *Route API* below.

Examples

```
const db = new Database();
const myFoxxService = db.route('my-foxx-service');
const response = await myFoxxService.post('users', {
  username: 'admin',
  password: 'hunter2'
});
// response.body is the result of
// POST /_db/_system/my-foxx-service/users
// with JSON request body '{"username": "admin", "password": "hunter2"}'
```

Collection API

These functions implement the HTTP API for manipulating collections.

The *Collection API* is implemented by all *Collection* instances, regardless of their specific type. I.e. it represents a shared subset between instances of *DocumentCollection*, *EdgeCollection*, *GraphVertexCollection* and *GraphEdgeCollection*.

Getting information about the collection

See the HTTP API documentation for details.

collection.exists

`async collection.exists(): boolean`

Checks whether the collection exists.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const result = await collection.exists();
// result indicates whether the collection exists
```

collection.get

`async collection.get(): Object`

Retrieves general information about the collection.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.get();
// data contains general information about the collection
```

collection.properties

`async collection.properties(): Object`

Retrieves the collection's properties.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.properties();
// data contains the collection's properties
```

collection.count

`async collection.count(): Object`

Retrieves information about the number of documents in a collection.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.count();
```

```
// data contains the collection's count
```

collection.figures

`async collection.figures(): Object`

Retrieves statistics for a collection.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.figures();
// data contains the collection's figures
```

collection.revision

`async collection.revision(): Object`

Retrieves the collection revision ID.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.revision();
// data contains the collection's revision
```

collection.checksum

`async collection.checksum([opts]): Object`

Retrieves the collection checksum.

Arguments

- **opts**: `Object` (optional)

  For information on the possible options see the HTTP API for getting collection information.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.checksum();
// data contains the collection's checksum
```

Manipulating the collection

These functions implement the HTTP API for modifying collections.

collection.create

`async collection.create([properties]): Object`

Creates a collection with the given *properties* for this collection's name, then returns the server response.

Arguments

- **properties**: `Object` (optional)

  For more information on the *properties* object, see the HTTP API documentation for creating collections.

Examples

```
const db = new Database();
const collection = db.collection('potatos');
await collection.create()
// the document collection "potatos" now exists

// -- or --

const collection = db.edgeCollection('friends');
await collection.create({
  waitForSync: true // always sync document changes to disk
});
// the edge collection "friends" now exists
```

collection.load

`async collection.load([count]): Object`

Tells the server to load the collection into memory.

Arguments

- **count**: `boolean` (Default: `true`)

  If set to `false`, the return value will not include the number of documents in the collection (which may speed up the process).

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
await collection.load(false)
// the collection has now been loaded into memory
```

collection.unload

`async collection.unload(): Object`

Tells the server to remove the collection from memory.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
await collection.unload()
// the collection has now been unloaded from memory
```

collection.setProperties

`async collection.setProperties(properties): Object`

Replaces the properties of the collection.

Arguments

- **properties**: `Object`

  For information on the *properties* argument see the HTTP API for modifying collections.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const result = await collection.setProperties({waitForSync: true})
assert.equal(result.waitForSync, true);
// the collection will now wait for data being written to disk
// whenever a document is changed
```

collection.rename

`async collection.rename(name): Object`

Renames the collection. The *Collection* instance will automatically update its name when the rename succeeds.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const result = await collection.rename('new-collection-name')
assert.equal(result.name, 'new-collection-name');
assert.equal(collection.name, result.name);
// result contains additional information about the collection
```

collection.rotate

`async collection.rotate(): Object`

Rotates the journal of the collection.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = await collection.rotate();
// data.result will be true if rotation succeeded
```

collection.truncate

`async collection.truncate(): Object`

Deletes **all documents** in the collection in the database.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
await collection.truncate();
// the collection "some-collection" is now empty
```

collection.drop

`async collection.drop([properties]): Object`

Deletes the collection from the database.

Arguments

- **properties**: `Object` (optional)

  An object with the following properties:

  - **isSystem**: `Boolean` (Default: `false`)

    Whether the collection should be dropped even if it is a system collection.

    This parameter must be set to `true` when dropping a system collection.

  For more information on the *properties* object, see the HTTP API documentation for dropping collections. **Examples**

```
const db = new Database();
const collection = db.collection('some-collection');
await collection.drop();
// the collection "some-collection" no longer exists
```

Manipulating documents

These functions implement the HTTP API for manipulating documents.

collection.replace

`async collection.replace(documentHandle, newValue, [opts]): Object`

Replaces the content of the document with the given *documentHandle* with the given *newValue* and returns an object containing the document's metadata.

Note: The *policy* option is not available when using the driver with ArangoDB 3.0 as it is redundant when specifying the *rev* option.

Arguments

- **documentHandle**: `string`

  The handle of the document to replace. This can either be the `_id` or the `_key` of a document in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **newValue**: `Object`

  The new data of the document.

- **opts**: `Object` (optional)

  If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until the document has been synced to disk. Default: `false`.

  - **rev**: `string` (optional)

    Only replace the document if it matches this revision.

  - **policy**: `string` (optional)

    Determines the behaviour when the revision is not matched:

    - if *policy* is set to `"last"`, the document will be replaced regardless of the revision.
    - if *policy* is set to `"error"` or not set, the replacement will fail with an error.

If a string is passed instead of an options object, it will be interpreted as the *rev* option.

For more information on the *opts* object, see the HTTP API documentation for working with documents.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const data = {number: 1, hello: 'world'};
const info1 = await collection.save(data);
const info2 = await collection.replace(info1, {number: 2});
assert.equal(info2._id, info1._id);
assert.notEqual(info2._rev, info1._rev);
const doc = await collection.document(info1);
assert.equal(doc._id, info1._id);
assert.equal(doc._rev, info2._rev);
assert.equal(doc.number, 2);
assert.equal(doc.hello, undefined);
```

collection.update

`async collection.update(documentHandle, newValue, [opts]): Object`

Updates (merges) the content of the document with the given *documentHandle* with the given *newValue* and returns an object containing the document's metadata.

Note: The *policy* option is not available when using the driver with ArangoDB 3.0 as it is redundant when specifying the *rev* option.

Arguments

- **documentHandle**: `string`

  Handle of the document to update. This can be either the `_id` or the `_key` of a document in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **newValue**: `Object`

  The new data of the document.

- **opts**: `Object` (optional)

  If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until document has been synced to disk.

  - **keepNull**: `boolean` (Default: `true`)

    If set to `false`, properties with a value of `null` indicate that a property should be deleted.

  - **mergeObjects**: `boolean` (Default: `true`)

    If set to `false`, object properties that already exist in the old document will be overwritten rather than merged. This does not affect arrays.

  - **returnOld**: `boolean` (Default: `false`)

    If set to `true`, return additionally the complete previous revision of the changed documents under the attribute `old` in the result.

  - **returnNew**: `boolean` (Default: `false`)

    If set to `true`, return additionally the complete new documents under the attribute `new` in the result.

  - **ignoreRevs**: `boolean` (Default: `true`)

    By default, or if this is set to true, the _rev attributes in the given documents are ignored. If this is set to false, then any _rev attribute given in a body document is taken as a precondition. The document is only updated if the current revision is the one specified.

  - **rev**: `string` (optional)

    Only update the document if it matches this revision.

  - **policy**: `string` (optional)

    Determines the behaviour when the revision is not matched:

    - if *policy* is set to `"last"`, the document will be replaced regardless of the revision.
    - if *policy* is set to `"error"` or not set, the replacement will fail with an error.

If a string is passed instead of an options object, it will be interpreted as the *rev* option.

For more information on the *opts* object, see the HTTP API documentation for working with documents.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const doc = {number: 1, hello: 'world'};
const doc1 = await collection.save(doc);
const doc2 = await collection.update(doc1, {number: 2});
assert.equal(doc2._id, doc1._id);
assert.notEqual(doc2._rev, doc1._rev);
```

```
const doc3 = await collection.document(doc2);
assert.equal(doc3._id, doc2._id);
assert.equal(doc3._rev, doc2._rev);
assert.equal(doc3.number, 2);
assert.equal(doc3.hello, doc.hello);
```

collection.bulkUpdate

`async collection.bulkUpdate(documents, [opts]): Object`

Updates (merges) the content of the documents with the given *documents* and returns an array containing the documents' metadata.

Note: This method is new in 3.0 and is available when using the driver with ArangoDB 3.0 and higher.

Arguments

- **documents**: `Array<Object>`

  Documents to update. Each object must have either the `_id` or the `_key` property.

- **opts**: `Object` (optional)

  If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until document has been synced to disk.

  - **keepNull**: `boolean` (Default: `true`)

    If set to `false`, properties with a value of `null` indicate that a property should be deleted.

  - **mergeObjects**: `boolean` (Default: `true`)

    If set to `false`, object properties that already exist in the old document will be overwritten rather than merged. This does not affect arrays.

  - **returnOld**: `boolean` (Default: `false`)

    If set to `true`, return additionally the complete previous revision of the changed documents under the attribute `old` in the result.

  - **returnNew**: `boolean` (Default: `false`)

    If set to `true`, return additionally the complete new documents under the attribute `new` in the result.

  - **ignoreRevs**: `boolean` (Default: `true`)

    By default, or if this is set to true, the _rev attributes in the given documents are ignored. If this is set to false, then any _rev attribute given in a body document is taken as a precondition. The document is only updated if the current revision is the one specified.

For more information on the *opts* object, see the HTTP API documentation for working with documents.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const doc1 = {number: 1, hello: 'world1'};
const info1 = await collection.save(doc1);
const doc2 = {number: 2, hello: 'world2'};
const info2 = await collection.save(doc2);
const result = await collection.bulkUpdate([
  {_key: info1._key, number: 3},
  {_key: info2._key, number: 4}
], {returnNew: true})
```

collection.remove

`async collection.remove(documentHandle, [opts]): Object`

Deletes the document with the given *documentHandle* from the collection.

Note: The *policy* option is not available when using the driver with ArangoDB 3.0 as it is redundant when specifying the *rev* option.

Arguments

- **documentHandle**: `string`

  The handle of the document to delete. This can be either the `_id` or the `_key` of a document in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **opts**: `Object` (optional)

  If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until document has been synced to disk.

  - **rev**: `string` (optional)

    Only update the document if it matches this revision.

  - **policy**: `string` (optional)

    Determines the behaviour when the revision is not matched:

    - if *policy* is set to `"last"`, the document will be replaced regardless of the revision.
    - if *policy* is set to `"error"` or not set, the replacement will fail with an error.

If a string is passed instead of an options object, it will be interpreted as the *rev* option.

For more information on the *opts* object, see the HTTP API documentation for working with documents.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

await collection.remove('some-doc');
// document 'some-collection/some-doc' no longer exists

// -- or --

await collection.remove('some-collection/some-doc');
// document 'some-collection/some-doc' no longer exists
```

collection.list

`async collection.list([type]): Array<string>`

Retrieves a list of references for all documents in the collection.

Arguments

- **type**: `string` (Default: `"id"`)

  The format of the document references:

  - if *type* is set to `"id"`, each reference will be the `_id` of the document.
  - if *type* is set to `"key"`, each reference will be the `_key` of the document.
  - if *type* is set to `"path"`, each reference will be the URI path of the document.

DocumentCollection API

The *DocumentCollection API* extends the *Collection API* with the following methods.

documentCollection.document

`async documentCollection.document(documentHandle): Object`

Retrieves the document with the given *documentHandle* from the collection.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve. This can be either the `_id` or the `_key` of a document in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const collection = db.collection('my-docs');

try {
  const doc = await collection.document('some-key');
  // the document exists
  assert.equal(doc._key, 'some-key');
  assert.equal(doc._id, 'my-docs/some-key');
} catch (err) {
  // something went wrong or
  // the document does not exist
}

// -- or --

try {
  const doc = await collection.document('my-docs/some-key');
  // the document exists
  assert.equal(doc._key, 'some-key');
  assert.equal(doc._id, 'my-docs/some-key');
} catch (err) {
  // something went wrong or
  // the document does not exist
}
```

documentCollection.save

`async documentCollection.save(data, [opts]): Object`

Creates a new document with the given *data* and returns an object containing the document's metadata.

Arguments

- **data**: `Object`

  The data of the new document, may include a `_key`.

- **opts**: `Object` (optional)

  If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until document has been synced to disk.

  - **returnNew**: `boolean` (Default: `false`)

If set to `true`, return additionally the complete new documents under the attribute `new` in the result.

- **returnOld**: `boolean` (Default: `false`)

  If set to `true`, return additionally the complete old documents under the attribute `old` in the result.

- **silent**: `boolean` (Default: `false`)

  If set to true, an empty object will be returned as response. No meta-data will be returned for the created document. This option can be used to save some network traffic.

- **overwrite**: `boolean` (Default: `false`)

  If set to true, the insert becomes a replace-insert. If a document with the same _key already exists the new document is not rejected with unique constraint violated but will replace the old document.

If a boolean is passed instead of an options object, it will be interpreted as the *returnNew* option.

For more information on the *opts* object, see the HTTP API documentation for working with documents.

Examples

```
const db = new Database();
const collection = db.collection('my-docs');
const data = {some: 'data'};
const info = await collection.save(data);
assert.equal(info._id, 'my-docs/' + info._key);
const doc2 = await collection.document(info)
assert.equal(doc2._id, info._id);
assert.equal(doc2._rev, info._rev);
assert.equal(doc2.some, data.some);

// -- or --

const db = new Database();
const collection = db.collection('my-docs');
const data = {some: 'data'};
const opts = {returnNew: true};
const doc = await collection.save(data, opts)
assert.equal(doc1._id, 'my-docs/' + doc1._key);
assert.equal(doc1.new.some, data.some);
```

EdgeCollection API

The *EdgeCollection API* extends the *Collection API* with the following methods.

edgeCollection.edge

`async edgeCollection.edge(documentHandle): Object`

Retrieves the edge with the given *documentHandle* from the collection.

Arguments

- **documentHandle**: `string`

  The handle of the edge to retrieve. This can be either the `_id` or the `_key` of an edge in the collection, or an edge (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const collection = db.edgeCollection('edges');

const edge = await collection.edge('some-key');
// the edge exists
assert.equal(edge._key, 'some-key');
assert.equal(edge._id, 'edges/some-key');

// -- or --

const edge = await collection.edge('edges/some-key');
// the edge exists
assert.equal(edge._key, 'some-key');
assert.equal(edge._id, 'edges/some-key');
```

edgeCollection.save

`async edgeCollection.save(data, [fromId, toId]): Object`

Creates a new edge between the documents *fromId* and *toId* with the given *data* and returns an object containing the edge's metadata.

Arguments

- **data**: `Object`

  The data of the new edge. If *fromId* and *toId* are not specified, the *data* needs to contain the properties `_from` and `_to`.

- **fromId**: `string` (optional)

  The handle of the start vertex of this edge. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **toId**: `string` (optional)

  The handle of the end vertex of this edge. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **opts**: `Object` (optional)

  If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until document has been synced to disk.

  - **returnNew**: `boolean` (Default: `false`)

If set to `true`, return additionally the complete new documents under the attribute `new` in the result.

- **returnOld**: `boolean` (Default: `false`)

  If set to `true`, return additionally the complete old documents under the attribute `old` in the result.

- **silent**: `boolean` (Default: `false`)

  If set to true, an empty object will be returned as response. No meta-data will be returned for the created document. This option can be used to save some network traffic.

- **overwrite**: `boolean` (Default: `false`)

  If set to true, the insert becomes a replace-insert. If a document with the same _key already exists the new document is not rejected with unique constraint violated but will replace the old document.

If a boolean is passed instead of an options object, it will be interpreted as the *returnNew* option.

Examples

```
const db = new Database();
const collection = db.edgeCollection('edges');
const data = {some: 'data'};

const info = await collection.save(
  data,
  'vertices/start-vertex',
  'vertices/end-vertex'
);
assert.equal(info._id, 'edges/' + info._key);
const edge = await collection.edge(edge)
assert.equal(edge._key, info._key);
assert.equal(edge._rev, info._rev);
assert.equal(edge.some, data.some);
assert.equal(edge._from, 'vertices/start-vertex');
assert.equal(edge._to, 'vertices/end-vertex');

// -- or --

const info = await collection.save({
  some: 'data',
  _from: 'verticies/start-vertex',
  _to: 'vertices/end-vertex'
});
// ...
```

edgeCollection.edges

`async edgeCollection.edges(documentHandle): Array<Object>`

Retrieves a list of all edges of the document with the given *documentHandle*.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve the edges of. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const collection = db.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/a', 'vertices/c'],
  ['z', 'vertices/d', 'vertices/a']
])
const edges = await collection.edges('vertices/a');
assert.equal(edges.length, 3);
```

```
assert.deepEqual(edges.map(edge => edge._key), ['x', 'y', 'z']);
```

edgeCollection.inEdges

`async edgeCollection.inEdges(documentHandle): Array<Object>`

Retrieves a list of all incoming edges of the document with the given *documentHandle*.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve the edges of. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const collection = db.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/a', 'vertices/c'],
  ['z', 'vertices/d', 'vertices/a']
]);
const edges = await collection.inEdges('vertices/a');
assert.equal(edges.length, 1);
assert.equal(edges[0]._key, 'z');
```

edgeCollection.outEdges

`async edgeCollection.outEdges(documentHandle): Array<Object>`

Retrieves a list of all outgoing edges of the document with the given *documentHandle*.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve the edges of. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const collection = db.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/a', 'vertices/c'],
  ['z', 'vertices/d', 'vertices/a']
]);
const edges = await collection.outEdges('vertices/a');
assert.equal(edges.length, 2);
assert.deepEqual(edges.map(edge => edge._key), ['x', 'y']);
```

edgeCollection.traversal

`async edgeCollection.traversal(startVertex, opts): Object`

Performs a traversal starting from the given *startVertex* and following edges contained in this edge collection.

Arguments

- **startVertex**: `string`

The handle of the start vertex. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **opts**: `Object`

  See the HTTP API documentation for details on the additional arguments.

  Please note that while *opts.filter*, *opts.visitor*, *opts.init*, *opts.expander* and *opts.sort* should be strings evaluating to well-formed JavaScript code, it's not possible to pass in JavaScript functions directly because the code needs to be evaluated on the server and will be transmitted in plain text.

Examples

```
const db = new Database();
const collection = db.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/b', 'vertices/c'],
  ['z', 'vertices/c', 'vertices/d']
]);
const result = await collection.traversal('vertices/a', {
  direction: 'outbound',
  visitor: 'result.vertices.push(vertex._key);',
  init: 'result.vertices = [];'
});
assert.deepEqual(result.vertices, ['a', 'b', 'c', 'd']);
```

Manipulating indexes

These functions implement the HTTP API for manipulating indexes.

collection.createIndex

`async collection.createIndex(details): Object`

Creates an arbitrary index on the collection.

Arguments

- **details**: `Object`

  For information on the possible properties of the *details* object, see the HTTP API for manipulating indexes.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const index = await collection.createIndex({type: 'hash', fields: ['a', 'a.b']});
// the index has been created with the handle `index.id`
```

collection.createHashIndex

`async collection.createHashIndex(fields, [opts]): Object`

Creates a hash index on the collection.

Arguments

- **fields**: `Array<string>`

  An array of names of document fields on which to create the index. If the value is a string, it will be wrapped in an array automatically.

- **opts**: `Object` (optional)

  Additional options for this index. If the value is a boolean, it will be interpreted as *opts.unique*.

For more information on hash indexes, see the HTTP API for hash indexes.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

const index = await collection.createHashIndex('favorite-color');
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['favorite-color']);

// -- or --

const index = await collection.createHashIndex(['favorite-color']);
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['favorite-color']);
```

collection.createSkipList

`async collection.createSkipList(fields, [opts]): Object`

Creates a skiplist index on the collection.

Arguments

- **fields**: `Array<string>`

  An array of names of document fields on which to create the index. If the value is a string, it will be wrapped in an array automatically.

- **opts**: `Object` (optional)

  Additional options for this index. If the value is a boolean, it will be interpreted as *opts.unique*.

For more information on skiplist indexes, see the HTTP API for skiplist indexes.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

const index = await collection.createSkipList('favorite-color')
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['favorite-color']);

// -- or --

const index = await collection.createSkipList(['favorite-color'])
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['favorite-color']);
```

collection.createGeoIndex

`async collection.createGeoIndex(fields, [opts]): Object`

Creates a geo-spatial index on the collection.

Arguments

- **fields**: `Array<string>`

  An array of names of document fields on which to create the index. Currently, geo indexes must cover exactly one field. If the value is a string, it will be wrapped in an array automatically.

- **opts**: `Object` (optional)

  An object containing additional properties of the index.

For more information on the properties of the *opts* object see the HTTP API for manipulating geo indexes.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

const index = await collection.createGeoIndex(['latitude', 'longitude']);
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['longitude', 'latitude']);

// -- or --

const index = await collection.createGeoIndex('location', {geoJson: true});
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['location']);
```

collection.createFulltextIndex

`async collection.createFulltextIndex(fields, [minLength]): Object`

Creates a fulltext index on the collection.

Arguments

- **fields**: `Array<string>`

  An array of names of document fields on which to create the index. Currently, fulltext indexes must cover exactly one field. If the value is a string, it will be wrapped in an array automatically.

- **minLength** (optional):

  Minimum character length of words to index. Uses a server-specific default value if not specified.

For more information on fulltext indexes, see the HTTP API for fulltext indexes.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

const index = await collection.createFulltextIndex('description');
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['description']);

// -- or --

const index = await collection.createFulltextIndex(['description']);
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['description']);
```

collection.createPersistentIndex

`async collection.createPersistentIndex(fields, [opts]): Object`

Creates a Persistent index on the collection. Persistent indexes are similarly in operation to skiplist indexes, only that these indexes are in disk as opposed to in memory. This reduces memory usage and DB startup time, with the trade-off being that it will always be orders of magnitude slower than in-memory indexes.

Arguments

- **fields**: `Array<string>`

  An array of names of document fields on which to create the index.

- **opts**: `Object` (optional)

  An object containing additional properties of the index.

For more information on the properties of the *opts* object see the HTTP API for manipulating Persistent indexes.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

const index = await collection.createPersistentIndex(['name', 'email']);
// the index has been created with the handle `index.id`
assert.deepEqual(index.fields, ['name', 'email']);
```

collection.index

`async collection.index(indexHandle): Object`

Fetches information about the index with the given *indexHandle* and returns it.

Arguments

- **indexHandle**: `string`

The handle of the index to look up. This can either be a fully-qualified identifier or the collection-specific key of the index. If the value is an object, its *id* property will be used instead.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const index = await collection.createFulltextIndex('description');
const result = await collection.index(index.id);
assert.equal(result.id, index.id);
// result contains the properties of the index

// -- or --

const result = await collection.index(index.id.split('/')[1]);
assert.equal(result.id, index.id);
// result contains the properties of the index
```

collection.indexes

`async collection.indexes(): Array<Object>`

Fetches a list of all indexes on this collection.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
await collection.createFulltextIndex('description')
const indexes = await collection.indexes();
assert.equal(indexes.length, 1);
// indexes contains information about the index
```

collection.dropIndex

`async collection.dropIndex(indexHandle): Object`

Deletes the index with the given *indexHandle* from the collection.

Arguments

- **indexHandle**: `string`

  The handle of the index to delete. This can either be a fully-qualified identifier or the collection-specific key of the index. If the value is an object, its *id* property will be used instead.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');
const index = await collection.createFulltextIndex('description');
await collection.dropIndex(index.id);
// the index has been removed from the collection

// -- or --

await collection.dropIndex(index.id.split('/')[1]);
// the index has been removed from the collection
```

collection.createCapConstraint

`async collection.createCapConstraint(size): Object`

Creates a cap constraint index on the collection.

Note: This method is not available when using the driver with ArangoDB 3.0 and higher as cap constraints are no longer supported.

Arguments

- **size**: `Object`

  An object with any of the following properties:

  - **size**: `number` (optional)

    The maximum number of documents in the collection.

  - **byteSize**: `number` (optional)

    The maximum size of active document data in the collection (in bytes).

If *size* is a number, it will be interpreted as *size.size*.

For more information on the properties of the *size* object see the HTTP API for creating cap constraints.

Examples

```
const db = new Database();
const collection = db.collection('some-collection');

const index = await collection.createCapConstraint(20)
// the index has been created with the handle `index.id`
assert.equal(index.size, 20);

// -- or --

const index = await collection.createCapConstraint({size: 20})
// the index has been created with the handle `index.id`
assert.equal(index.size, 20);
```

Simple queries

These functions implement the HTTP API for simple queries.

collection.all

`async collection.all([opts]): Cursor`

Performs a query to fetch all documents in the collection. Returns a new *Cursor* instance for the query results.

Arguments

- **opts**: `Object` (optional)

  For information on the possible options see the HTTP API for returning all documents.

collection.any

`async collection.any(): Object`

Fetches a document from the collection at random.

collection.first

`async collection.first([opts]): Array<Object>`

Performs a query to fetch the first documents in the collection. Returns an array of the matching documents.

Note: This method is not available when using the driver with ArangoDB 3.0 and higher as the corresponding API method has been removed.

Arguments

- **opts**: `Object` (optional)

  For information on the possible options see the HTTP API for returning the first document of a collection.

  If *opts* is a number it is treated as *opts.count*.

collection.last

`async collection.last([opts]): Array<Object>`

Performs a query to fetch the last documents in the collection. Returns an array of the matching documents.

Note: This method is not available when using the driver with ArangoDB 3.0 and higher as the corresponding API method has been removed.

Arguments

- **opts**: `Object` (optional)

  For information on the possible options see the HTTP API for returning the last document of a collection.

  If *opts* is a number it is treated as *opts.count*.

collection.byExample

`async collection.byExample(example, [opts]): Cursor`

Performs a query to fetch all documents in the collection matching the given *example*. Returns a new *Cursor* instance for the query results.

Arguments

- **example**: *Object*

  An object representing an example for documents to be matched against.

- **opts**: *Object* (optional)

  For information on the possible options see the HTTP API for fetching documents by example.

collection.firstExample

```
async collection.firstExample(example): Object
```

Fetches the first document in the collection matching the given *example*.

Arguments

- **example**: *Object*

  An object representing an example for documents to be matched against.

collection.removeByExample

```
async collection.removeByExample(example, [opts]): Object
```

Removes all documents in the collection matching the given *example*.

Arguments

- **example**: *Object*

  An object representing an example for documents to be matched against.

- **opts**: *Object* (optional)

  For information on the possible options see the HTTP API for removing documents by example.

collection.replaceByExample

```
async collection.replaceByExample(example, newValue, [opts]): Object
```

Replaces all documents in the collection matching the given *example* with the given *newValue*.

Arguments

- **example**: *Object*

  An object representing an example for documents to be matched against.

- **newValue**: *Object*

  The new value to replace matching documents with.

- **opts**: *Object* (optional)

  For information on the possible options see the HTTP API for replacing documents by example.

collection.updateByExample

```
async collection.updateByExample(example, newValue, [opts]): Object
```

Updates (patches) all documents in the collection matching the given *example* with the given *newValue*.

Arguments

- **example**: *Object*

  An object representing an example for documents to be matched against.

- **newValue**: *Object*

  The new value to update matching documents with.

- **opts**: *Object* (optional)

  For information on the possible options see the HTTP API for updating documents by example.

collection.lookupByKeys

```
async collection.lookupByKeys(keys): Array<Object>
```

Fetches the documents with the given *keys* from the collection. Returns an array of the matching documents.

Arguments

- **keys**: *Array*

  An array of document keys to look up.

collection.removeByKeys

```
async collection.removeByKeys(keys, [opts]): Object
```

Deletes the documents with the given *keys* from the collection.

Arguments

- **keys**: *Array*

  An array of document keys to delete.

- **opts**: *Object* (optional)

  For information on the possible options see the HTTP API for removing documents by keys.

collection.fulltext

```
async collection.fulltext(fieldName, query, [opts]): Cursor
```

Performs a fulltext query in the given *fieldName* on the collection.

Arguments

- **fieldName**: *String*

  Name of the field to search on documents in the collection.

- **query**: *String*

  Fulltext query string to search for.

- **opts**: *Object* (optional)

  For information on the possible options see the HTTP API for fulltext queries.

Bulk importing documents

This function implements the HTTP API for bulk imports.

collection.import

`async collection.import(data, [opts]): Object`

Bulk imports the given *data* into the collection.

Arguments

- **data**: `Array<Array<any>> | Array<Object>`

  The data to import. This can be an array of documents:

  ```
  [
    {key1: value1, key2: value2}, // document 1
    {key1: value1, key2: value2}, // document 2
    ...
  ]
  ```

  Or it can be an array of value arrays following an array of keys.

  ```
  [
    ['key1', 'key2'], // key names
    [value1, value2], // document 1
    [value1, value2], // document 2
    ...
  ]
  ```

- **opts**: `Object` (optional) If *opts* is set, it must be an object with any of the following properties:

  - **waitForSync**: `boolean` (Default: `false`)

    Wait until the documents have been synced to disk.

  - **details**: `boolean` (Default: `false`)

    Whether the response should contain additional details about documents that could not be imported.false*.

  - **type**: `string` (Default: `"auto"`)

    Indicates which format the data uses. Can be `"documents"`, `"array"` or `"auto"`.

If *data* is a JavaScript array, it will be transmitted as a line-delimited JSON stream. If *opts.type* is set to `"array"`, it will be transmitted as regular JSON instead. If *data* is a string, it will be transmitted as it is without any processing.

For more information on the *opts* object, see the HTTP API documentation for bulk imports.

Examples

```
const db = new Database();
const collection = db.collection('users');

// document stream
const result = await collection.import([
  {username: 'admin', password: 'hunter2'},
  {username: 'jcd', password: 'bionicman'},
  {username: 'jreyes', password: 'amigo'},
  {username: 'ghermann', password: 'zeitgeist'}
]);
assert.equal(result.created, 4);

// -- or --
```

```
// array stream with header
const result = await collection.import([
  ['username', 'password'], // keys
  ['admin', 'hunter2'], // row 1
  ['jcd', 'bionicman'], // row 2
  ['jreyes', 'amigo'],
  ['ghermann', 'zeitgeist']
]);
assert.equal(result.created, 4);

// -- or --

// raw line-delimited JSON array stream with header
const result = await collection.import([
  '["username", "password"]',
  '["admin", "hunter2"]',
  '["jcd", "bionicman"]',
  '["jreyes", "amigo"]',
  '["ghermann", "zeitgeist"]'
].join('\r\n') + '\r\n');
assert.equal(result.created, 4);
```

Cursor API

Cursor instances provide an abstraction over the HTTP API's limitations. Unless a method explicitly exhausts the cursor, the driver will only fetch as many batches from the server as necessary. Like the server-side cursors, *Cursor* instances are incrementally depleted as they are read from.

```
const db = new Database();
const cursor = await db.query('FOR x IN 1..5 RETURN x');
// query result list: [1, 2, 3, 4, 5]
const value = await cursor.next();
assert.equal(value, 1);
// remaining result list: [2, 3, 4, 5]
```

cursor.count

`cursor.count: number`

The total number of documents in the query result. This is only available if the `count` option was used.

cursor.all

`async cursor.all(): Array<Object>`

Exhausts the cursor, then returns an array containing all values in the cursor's remaining result list.

Examples

```
const cursor = await db._query('FOR x IN 1..5 RETURN x');
const result = await cursor.all()
// result is an array containing the entire query result
assert.deepEqual(result, [1, 2, 3, 4, 5]);
assert.equal(cursor.hasNext(), false);
```

cursor.next

`async cursor.next(): Object`

Advances the cursor and returns the next value in the cursor's remaining result list. If the cursor has already been exhausted, returns `undefined` instead.

Examples

```
// query result list: [1, 2, 3, 4, 5]
const val = await cursor.next();
assert.equal(val, 1);
// remaining result list: [2, 3, 4, 5]

const val2 = await cursor.next();
assert.equal(val2, 2);
// remaining result list: [3, 4, 5]
```

cursor.hasNext

`cursor.hasNext(): boolean`

Returns `true` if the cursor has more values or `false` if the cursor has been exhausted.

Examples

```
await cursor.all(); // exhausts the cursor
assert.equal(cursor.hasNext(), false);
```

cursor.each

`async cursor.each(fn): any`

Advances the cursor by applying the function *fn* to each value in the cursor's remaining result list until the cursor is exhausted or *fn* explicitly returns `false`.

Returns the last return value of *fn*.

Equivalent to *Array.prototype.forEach* (except async).

Arguments

- **fn**: `Function`

  A function that will be invoked for each value in the cursor's remaining result list until it explicitly returns `false` or the cursor is exhausted.

  The function receives the following arguments:

  - **value**: `any`

    The value in the cursor's remaining result list.

  - **index**: `number`

    The index of the value in the cursor's remaining result list.

  - **cursor**: `Cursor`

    The cursor itself.

Examples

```
const results = [];
function doStuff(value) {
  const VALUE = value.toUpperCase();
  results.push(VALUE);
  return VALUE;
}

const cursor = await db.query('FOR x IN ["a", "b", "c"] RETURN x')
const last = await cursor.each(doStuff);
assert.deepEqual(results, ['A', 'B', 'C']);
assert.equal(cursor.hasNext(), false);
assert.equal(last, 'C');
```

cursor.every

`async cursor.every(fn): boolean`

Advances the cursor by applying the function *fn* to each value in the cursor's remaining result list until the cursor is exhausted or *fn* returns a value that evaluates to `false`.

Returns `false` if *fn* returned a value that evaluates to `false`, or `true` otherwise.

Equivalent to *Array.prototype.every* (except async).

Arguments

- **fn**: `Function`

  A function that will be invoked for each value in the cursor's remaining result list until it returns a value that evaluates to `false` or the cursor is exhausted.

The function receives the following arguments:

- **value**: `any`

  The value in the cursor's remaining result list.

- **index**: `number`

  The index of the value in the cursor's remaining result list.

- **cursor**: `Cursor`

  The cursor itself.

```
const even = value => value % 2 === 0;

const cursor = await db.query('FOR x IN 2..5 RETURN x');
const result = await cursor.every(even);
assert.equal(result, false); // 3 is not even
assert.equal(cursor.hasNext(), true);

const value = await cursor.next();
assert.equal(value, 4); // next value after 3
```

cursor.some

`async cursor.some(fn): boolean`

Advances the cursor by applying the function *fn* to each value in the cursor's remaining result list until the cursor is exhausted or *fn* returns a value that evaluates to `true`.

Returns `true` if *fn* returned a value that evalutes to `true`, or `false` otherwise.

Equivalent to *Array.prototype.some* (except async).

Examples

```
const even = value => value % 2 === 0;

const cursor = await db.query('FOR x IN 1..5 RETURN x');
const result = await cursor.some(even);
assert.equal(result, true); // 2 is even
assert.equal(cursor.hasNext(), true);

const value = await cursor.next();
assert.equal(value, 3); // next value after 2
```

cursor.map

`cursor.map(fn): Array<any>`

Advances the cursor by applying the function *fn* to each value in the cursor's remaining result list until the cursor is exhausted.

Returns an array of the return values of *fn*.

Equivalent to *Array.prototype.map* (except async).

Note: This creates an array of all return values. It is probably a bad idea to do this for very large query result sets.

Arguments

- **fn**: `Function`

  A function that will be invoked for each value in the cursor's remaining result list until the cursor is exhausted.

  The function receives the following arguments:

  - **value**: `any`

The value in the cursor's remaining result list.

- **index**: `number`

  The index of the value in the cursor's remaining result list.

- **cursor**: `Cursor`

  The cursor itself.

Examples

```
const square = value => value * value;
const cursor = await db.query('FOR x IN 1..5 RETURN x');
const result = await cursor.map(square);
assert.equal(result.length, 5);
assert.deepEqual(result, [1, 4, 9, 16, 25]);
assert.equal(cursor.hasNext(), false);
```

cursor.reduce

`cursor.reduce(fn, [accu]): any`

Exhausts the cursor by reducing the values in the cursor's remaining result list with the given function *fn*. If *accu* is not provided, the first value in the cursor's remaining result list will be used instead (the function will not be invoked for that value).

Equivalent to *Array.prototype.reduce* (except async).

Arguments

- **fn**: `Function`

  A function that will be invoked for each value in the cursor's remaining result list until the cursor is exhausted.

  The function receives the following arguments:

  - **accu**: `any`

    The return value of the previous call to *fn*. If this is the first call, *accu* will be set to the *accu* value passed to *reduce* or the first value in the cursor's remaining result list.

  - **value**: `any`

    The value in the cursor's remaining result list.

  - **index**: `number`

    The index of the value in the cursor's remaining result list.

  - **cursor**: `Cursor`

    The cursor itself.

Examples

```
const add = (a, b) => a + b;
const baseline = 1000;

const cursor = await db.query('FOR x IN 1..5 RETURN x');
const result = await cursor.reduce(add, baseline)
assert.equal(result, baseline + 1 + 2 + 3 + 4 + 5);
assert.equal(cursor.hasNext(), false);

// -- or --

const result = await cursor.reduce(add);
assert.equal(result, 1 + 2 + 3 + 4 + 5);
assert.equal(cursor.hasNext(), false);
```

Graph API

These functions implement the HTTP API for manipulating graphs.

graph.exists

`async graph.exists(): boolean`

Checks whether the graph exists.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const result = await graph.exists();
// result indicates whether the graph exists
```

graph.get

`async graph.get(): Object`

Retrieves general information about the graph.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const data = await graph.get();
// data contains general information about the graph
```

graph.create

`async graph.create(properties): Object`

Creates a graph with the given *properties* for this graph's name, then returns the server response.

Arguments

- **properties**: `Object`

  For more information on the *properties* object, see the HTTP API documentation for creating graphs.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const info = await graph.create({
  edgeDefinitions: [{
    collection: 'edges',
    from: ['start-vertices'],
    to: ['end-vertices']
  }]
});
// graph now exists
```

graph.drop

`async graph.drop([dropCollections]): Object`

Deletes the graph from the database.

Arguments

- **dropCollections**: `boolean` (optional)

  If set to `true`, the collections associated with the graph will also be deleted.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
await graph.drop();
// the graph "some-graph" no longer exists
```

Manipulating vertices

graph.vertexCollection

`graph.vertexCollection(collectionName): GraphVertexCollection`

Returns a new *GraphVertexCollection* instance with the given name for this graph.

Arguments

- **collectionName**: `string`

  Name of the vertex collection.

Examples

```
const db = new Database();
const graph = db.graph("some-graph");
const collection = graph.vertexCollection("vertices");
assert.equal(collection.name, "vertices");
// collection is a GraphVertexCollection
```

graph.listVertexCollections

`async graph.listVertexCollections([excludeOrphans]): Array<Object>`

Fetches all vertex collections from the graph and returns an array of collection descriptions.

Arguments

- **excludeOrphans**: `boolean` (Default: `false`)

  Whether orphan collections should be excluded.

Examples

```
const graph = db.graph('some-graph');

const collections = await graph.listVertexCollections();
// collections is an array of collection descriptions
// including orphan collections

// -- or --

const collections = await graph.listVertexCollections(true);
// collections is an array of collection descriptions
// not including orphan collections
```

graph.vertexCollections

`async graph.vertexCollections([excludeOrphans]): Array<Collection>`

Fetches all vertex collections from the database and returns an array of *GraphVertexCollection* instances for the collections.

Arguments

- **excludeOrphans**: `boolean` (Default: `false`)

  Whether orphan collections should be excluded.

Examples

```
const graph = db.graph('some-graph');
```

```
const collections = await graph.vertexCollections()
// collections is an array of GraphVertexCollection
// instances including orphan collections

// -- or --

const collections = await graph.vertexCollections(true)
// collections is an array of GraphVertexCollection
// instances not including orphan collections
```

graph.addVertexCollection

`async graph.addVertexCollection(collectionName): Object`

Adds the collection with the given *collectionName* to the graph's vertex collections.

Arguments

- **collectionName**: `string`

  Name of the vertex collection to add to the graph.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
await graph.addVertexCollection('vertices');
// the collection "vertices" has been added to the graph
```

graph.removeVertexCollection

`async graph.removeVertexCollection(collectionName, [dropCollection]): Object`

Removes the vertex collection with the given *collectionName* from the graph.

Arguments

- **collectionName**: `string`

  Name of the vertex collection to remove from the graph.

- **dropCollection**: `boolean` (optional)

  If set to `true`, the collection will also be deleted from the database.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
await graph.removeVertexCollection('vertices')
// collection "vertices" has been removed from the graph

// -- or --

await graph.removeVertexCollection('vertices', true)
// collection "vertices" has been removed from the graph
// the collection has also been dropped from the database
// this may have been a bad idea
```

Manipulating edges

graph.edgeCollection

`graph.edgeCollection(collectionName): GraphEdgeCollection`

Returns a new *GraphEdgeCollection* instance with the given name bound to this graph.

Arguments

- **collectionName**: `string`

  Name of the edge collection.

Examples

```
const db = new Database();
// assuming the collections "edges" and "vertices" exist
const graph = db.graph("some-graph");
const collection = graph.edgeCollection("edges");
assert.equal(collection.name, "edges");
// collection is a GraphEdgeCollection
```

graph.addEdgeDefinition

`async graph.addEdgeDefinition(definition): Object`

Adds the given edge definition *definition* to the graph.

Arguments

- **definition**: `Object`

  For more information on edge definitions see the HTTP API for managing graphs.

Examples

```
const db = new Database();
// assuming the collections "edges" and "vertices" exist
const graph = db.graph('some-graph');
await graph.addEdgeDefinition({
  collection: 'edges',
  from: ['vertices'],
  to: ['vertices']
});
// the edge definition has been added to the graph
```

graph.replaceEdgeDefinition

`async graph.replaceEdgeDefinition(collectionName, definition): Object`

Replaces the edge definition for the edge collection named *collectionName* with the given *definition*.

Arguments

- **collectionName**: `string`

  Name of the edge collection to replace the definition of.

- **definition**: `Object`

  For more information on edge definitions see the HTTP API for managing graphs.

Examples

```
const db = new Database();
// assuming the collections "edges", "vertices" and "more-vertices" exist
const graph = db.graph('some-graph');
await graph.replaceEdgeDefinition('edges', {
  collection: 'edges',
  from: ['vertices'],
  to: ['more-vertices']
});
// the edge definition has been modified
```

graph.removeEdgeDefinition

`async graph.removeEdgeDefinition(definitionName, [dropCollection]): Object`

Removes the edge definition with the given *definitionName* form the graph.

Arguments

- **definitionName**: `string`

  Name of the edge definition to remove from the graph.

- **dropCollection**: `boolean` (optional)

  If set to `true`, the edge collection associated with the definition will also be deleted from the database.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');

await graph.removeEdgeDefinition('edges')
// the edge definition has been removed

// -- or --

await graph.removeEdgeDefinition('edges', true)
// the edge definition has been removed
// and the edge collection "edges" has been dropped
// this may have been a bad idea
```

graph.traversal

`async graph.traversal(startVertex, opts): Object`

Performs a traversal starting from the given *startVertex* and following edges contained in any of the edge collections of this graph.

Arguments

- **startVertex**: `string`

  The handle of the start vertex. This can be either the `_id` of a document in the graph or a document (i.e. an object with an `_id` property).

- **opts**: `Object`

  See the HTTP API documentation for details on the additional arguments.

  Please note that while *opts.filter*, *opts.visitor*, *opts.init*, *opts.expander* and *opts.sort* should be strings evaluating to well-formed JavaScript functions, it's not possible to pass in JavaScript functions directly because the functions need to be evaluated on the server and will be transmitted in plain text.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');
```

```
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/b', 'vertices/c'],
  ['z', 'vertices/c', 'vertices/d']
])
const result = await graph.traversal('vertices/a', {
  direction: 'outbound',
  visitor: 'result.vertices.push(vertex._key);',
  init: 'result.vertices = [];'
});
assert.deepEqual(result.vertices, ['a', 'b', 'c', 'd']);
```

GraphVertexCollection API

The *GraphVertexCollection API* extends the *Collection API* with the following methods.

graphVertexCollection.remove

`async graphVertexCollection.remove(documentHandle): Object`

Deletes the vertex with the given *documentHandle* from the collection.

Arguments

- **documentHandle**: `string`

  The handle of the vertex to retrieve. This can be either the `_id` or the `_key` of a vertex in the collection, or a vertex (i.e. an object with an `_id` or `_key` property).

Examples

```
const graph = db.graph('some-graph');
const collection = graph.vertexCollection('vertices');

await collection.remove('some-key')
// document 'vertices/some-key' no longer exists

// -- or --

await collection.remove('vertices/some-key')
// document 'vertices/some-key' no longer exists
```

graphVertexCollection.vertex

`async graphVertexCollection.vertex(documentHandle): Object`

Retrieves the vertex with the given *documentHandle* from the collection.

Arguments

- **documentHandle**: `string`

  The handle of the vertex to retrieve. This can be either the `_id` or the `_key` of a vertex in the collection, or a vertex (i.e. an object with an `_id` or `_key` property).

Examples

```
const graph = db.graph('some-graph');
const collection = graph.vertexCollection('vertices');

const doc = await collection.vertex('some-key');
// the vertex exists
assert.equal(doc._key, 'some-key');
assert.equal(doc._id, 'vertices/some-key');

// -- or --

const doc = await collection.vertex('vertices/some-key');
// the vertex exists
assert.equal(doc._key, 'some-key');
assert.equal(doc._id, 'vertices/some-key');
```

graphVertexCollection.save

`async graphVertexCollection.save(data): Object`

Creates a new vertex with the given *data*.

Arguments

- **data**: `Object`

  The data of the vertex.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.vertexCollection('vertices');
const doc = await collection.save({some: 'data'});
assert.equal(doc._id, 'vertices/' + doc._key);
assert.equal(doc.some, 'data');
```

GraphEdgeCollection API

The *GraphEdgeCollection API* extends the *Collection API* with the following methods.

graphEdgeCollection.remove

`async graphEdgeCollection.remove(documentHandle): Object`

Deletes the edge with the given *documentHandle* from the collection.

Arguments

- **documentHandle**: `string`

  The handle of the edge to retrieve. This can be either the `_id` or the `_key` of an edge in the collection, or an edge (i.e. an object with an `_id` or `_key` property).

Examples

```
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');

await collection.remove('some-key')
// document 'edges/some-key' no longer exists

// -- or --

await collection.remove('edges/some-key')
// document 'edges/some-key' no longer exists
```

graphEdgeCollection.edge

`async graphEdgeCollection.edge(documentHandle): Object`

Retrieves the edge with the given *documentHandle* from the collection.

Arguments

- **documentHandle**: `string`

  The handle of the edge to retrieve. This can be either the `_id` or the `_key` of an edge in the collection, or an edge (i.e. an object with an `_id` or `_key` property).

Examples

```
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');

const edge = await collection.edge('some-key');
// the edge exists
assert.equal(edge._key, 'some-key');
assert.equal(edge._id, 'edges/some-key');

// -- or --

const edge = await collection.edge('edges/some-key');
// the edge exists
assert.equal(edge._key, 'some-key');
assert.equal(edge._id, 'edges/some-key');
```

graphEdgeCollection.save

`async graphEdgeCollection.save(data, [fromId, toId]): Object`

Creates a new edge between the vertices *fromId* and *toId* with the given *data*.

Arguments

- **data**: `Object`

  The data of the new edge. If *fromId* and *toId* are not specified, the *data* needs to contain the properties **from_ and** to_.

- **fromId**: `string` (optional)

  The handle of the start vertex of this edge. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **toId**: `string` (optional)

  The handle of the end vertex of this edge. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');
const edge = await collection.save(
  {some: 'data'},
  'vertices/start-vertex',
  'vertices/end-vertex'
);
assert.equal(edge._id, 'edges/' + edge._key);
assert.equal(edge.some, 'data');
assert.equal(edge._from, 'vertices/start-vertex');
assert.equal(edge._to, 'vertices/end-vertex');
```

graphEdgeCollection.edges

`async graphEdgeCollection.edges(documentHandle): Array<Object>`

Retrieves a list of all edges of the document with the given *documentHandle*.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve the edges of. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/a', 'vertices/c'],
  ['z', 'vertices/d', 'vertices/a']
]);
const edges = await collection.edges('vertices/a');
assert.equal(edges.length, 3);
assert.deepEqual(edges.map(edge => edge._key), ['x', 'y', 'z']);
```

graphEdgeCollection.inEdges

`async graphEdgeCollection.inEdges(documentHandle): Array<Object>`

Retrieves a list of all incoming edges of the document with the given *documentHandle*.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve the edges of. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/a', 'vertices/c'],
  ['z', 'vertices/d', 'vertices/a']
]);
const edges = await collection.inEdges('vertices/a');
assert.equal(edges.length, 1);
assert.equal(edges[0]._key, 'z');
```

graphEdgeCollection.outEdges

`async graphEdgeCollection.outEdges(documentHandle): Array<Object>`

Retrieves a list of all outgoing edges of the document with the given *documentHandle*.

Arguments

- **documentHandle**: `string`

  The handle of the document to retrieve the edges of. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/a', 'vertices/c'],
  ['z', 'vertices/d', 'vertices/a']
]);
const edges = await collection.outEdges('vertices/a');
assert.equal(edges.length, 2);
assert.deepEqual(edges.map(edge => edge._key), ['x', 'y']);
```

graphEdgeCollection.traversal

`async graphEdgeCollection.traversal(startVertex, opts): Object`

Performs a traversal starting from the given *startVertex* and following edges contained in this edge collection.

Arguments

- **startVertex**: `string`

  The handle of the start vertex. This can be either the `_id` of a document in the database, the `_key` of an edge in the collection, or a document (i.e. an object with an `_id` or `_key` property).

- **opts**: `Object`

  See the HTTP API documentation for details on the additional arguments.

Please note that while *opts.filter*, *opts.visitor*, *opts.init*, *opts.expander* and *opts.sort* should be strings evaluating to well-formed JavaScript code, it's not possible to pass in JavaScript functions directly because the code needs to be evaluated on the server and will be transmitted in plain text.

Examples

```
const db = new Database();
const graph = db.graph('some-graph');
const collection = graph.edgeCollection('edges');
await collection.import([
  ['_key', '_from', '_to'],
  ['x', 'vertices/a', 'vertices/b'],
  ['y', 'vertices/b', 'vertices/c'],
  ['z', 'vertices/c', 'vertices/d']
]);
const result = await collection.traversal('vertices/a', {
  direction: 'outbound',
  visitor: 'result.vertices.push(vertex._key);',
  init: 'result.vertices = [];'
});
assert.deepEqual(result.vertices, ['a', 'b', 'c', 'd']);
```

Route API

Route instances provide access for arbitrary HTTP requests. This allows easy access to Foxx services and other HTTP APIs not covered by the driver itself.

route.route

`route.route([path], [headers]): Route`

Returns a new *Route* instance for the given path (relative to the current route) that can be used to perform arbitrary HTTP requests.

Arguments

- **path**: `string` (optional)

  The relative URL of the route.

- **headers**: `Object` (optional)

  Default headers that should be sent with each request to the route.

If *path* is missing, the route will refer to the base URL of the database.

Examples

```
const db = new Database();
const route = db.route("my-foxx-service");
const users = route.route("users");
// equivalent to db.route('my-foxx-service/users')
```

route.get

`async route.get([path,] [qs]): Response`

Performs a GET request to the given URL and returns the server response.

Arguments

- **path**: `string` (optional)

  The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

- **qs**: `string` (optional)

  The query string for the request. If *qs* is an object, it will be translated to a query string.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.get();
// response.body is the response body of calling
// GET _db/_system/my-foxx-service

// -- or --

const response = await route.get('users');
// response.body is the response body of calling
// GET _db/_system/my-foxx-service/users

// -- or --

const response = await route.get('users', {group: 'admin'});
// response.body is the response body of calling
// GET _db/_system/my-foxx-service/users?group=admin
```

route.post

`async route.post([path,] [body, [qs]]): Response`

Performs a POST request to the given URL and returns the server response.

Arguments

- **path**: `string` (optional)

  The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

- **body**: `string` (optional)

  The response body. If *body* is an object, it will be encoded as JSON.

- **qs**: `string` (optional)

  The query string for the request. If *qs* is an object, it will be translated to a query string.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.post()
// response.body is the response body of calling
// POST _db/_system/my-foxx-service

// -- or --

const response = await route.post('users')
// response.body is the response body of calling
// POST _db/_system/my-foxx-service/users

// -- or --

const response = await route.post('users', {
  username: 'admin',
  password: 'hunter2'
});
// response.body is the response body of calling
// POST _db/_system/my-foxx-service/users
// with JSON request body {"username": "admin", "password": "hunter2"}

// -- or --

const response = await route.post('users', {
  username: 'admin',
  password: 'hunter2'
}, {admin: true});
// response.body is the response body of calling
// POST _db/_system/my-foxx-service/users?admin=true
// with JSON request body {"username": "admin", "password": "hunter2"}
```

route.put

`async route.put([path,] [body, [qs]]): Response`

Performs a PUT request to the given URL and returns the server response.

Arguments

- **path**: `string` (optional)

  The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

- **body**: `string` (optional)

  The response body. If *body* is an object, it will be encoded as JSON.

- **qs**: `string` (optional)

The query string for the request. If *qs* is an object, it will be translated to a query string.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.put();
// response.body is the response body of calling
// PUT _db/_system/my-foxx-service

// -- or --

const response = await route.put('users/admin');
// response.body is the response body of calling
// PUT _db/_system/my-foxx-service/users

// -- or --

const response = await route.put('users/admin', {
  username: 'admin',
  password: 'hunter2'
});
// response.body is the response body of calling
// PUT _db/_system/my-foxx-service/users/admin
// with JSON request body {"username": "admin", "password": "hunter2"}

// -- or --

const response = await route.put('users/admin', {
  username: 'admin',
  password: 'hunter2'
}, {admin: true});
// response.body is the response body of calling
// PUT _db/_system/my-foxx-service/users/admin?admin=true
// with JSON request body {"username": "admin", "password": "hunter2"}
```

route.patch

`async route.patch([path,] [body, [qs]]): Response`

Performs a PATCH request to the given URL and returns the server response.

Arguments

- **path**: `string` (optional)

  The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

- **body**: `string` (optional)

  The response body. If *body* is an object, it will be encoded as JSON.

- **qs**: `string` (optional)

  The query string for the request. If *qs* is an object, it will be translated to a query string.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.patch();
// response.body is the response body of calling
// PATCH _db/_system/my-foxx-service

// -- or --

const response = await route.patch('users/admin');
// response.body is the response body of calling
// PATCH _db/_system/my-foxx-service/users

// -- or --
```

```
const response = await route.patch('users/admin', {
  password: 'hunter2'
});
// response.body is the response body of calling
// PATCH _db/_system/my-foxx-service/users/admin
// with JSON request body {"password": "hunter2"}

// -- or --

const response = await route.patch('users/admin', {
  password: 'hunter2'
}, {admin: true});
// response.body is the response body of calling
// PATCH _db/_system/my-foxx-service/users/admin?admin=true
// with JSON request body {"password": "hunter2"}
```

route.delete

`async route.delete([path,] [qs]): Response`

Performs a DELETE request to the given URL and returns the server response.

Arguments

- **path**: `string` (optional)

  The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

- **qs**: `string` (optional)

  The query string for the request. If *qs* is an object, it will be translated to a query string.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.delete()
// response.body is the response body of calling
// DELETE _db/_system/my-foxx-service

// -- or --

const response = await route.delete('users/admin')
// response.body is the response body of calling
// DELETE _db/_system/my-foxx-service/users/admin

// -- or --

const response = await route.delete('users/admin', {permanent: true})
// response.body is the response body of calling
// DELETE _db/_system/my-foxx-service/users/admin?permanent=true
```

route.head

`async route.head([path,] [qs]): Response`

Performs a HEAD request to the given URL and returns the server response.

Arguments

- **path**: `string` (optional)

  The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

- **qs**: `string` (optional)

  The query string for the request. If *qs* is an object, it will be translated to a query string.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.head();
// response is the response object for
// HEAD _db/_system/my-foxx-service
```

route.request

`async route.request([opts]): Response`

Performs an arbitrary request to the given URL and returns the server response.

Arguments

- **opts**: `Object` (optional)

  An object with any of the following properties:

  - **path**: `string` (optional)

    The route-relative URL for the request. If omitted, the request will be made to the base URL of the route.

  - **absolutePath**: `boolean` (Default: `false`)

    Whether the *path* is relative to the connection's base URL instead of the route.

  - **body**: `string` (optional)

    The response body. If *body* is an object, it will be encoded as JSON.

  - **qs**: `string` (optional)

    The query string for the request. If *qs* is an object, it will be translated to a query string.

  - **headers**: `Object` (optional)

    An object containing additional HTTP headers to be sent with the request.

  - **method**: `string` (Default: `"GET"`)

    HTTP method of this request.

Examples

```
const db = new Database();
const route = db.route('my-foxx-service');
const response = await route.request({
  path: 'hello-world',
  method: 'POST',
  body: {hello: 'world'},
  qs: {admin: true}
});
// response.body is the response body of calling
// POST _db/_system/my-foxx-service/hello-world?admin=true
// with JSON request body '{"hello": "world"}'
```

Spring Data ArangoDB

- Getting Started
- Reference

Learn more

- ArangoDB
- Demo
- JavaDoc 1.0.0
- JavaDoc 2.0.0
- JavaDoc Java driver
- Changelog

Spring Data ArangoDB - Getting Started

Supported versions

Spring Data ArangoDB	Spring Data	ArangoDB
1.0.0	1.13.x	3.0*, 3.1, 3.2
2.0.0	2.0.x	3.0*, 3.1, 3.2

Spring Data ArangoDB requires ArangoDB 3.0 or higher - which you can download here - and Java 8 or higher.

Note: ArangoDB 3.0 does not support the default transport protocol VelocyStream. A manual switch to HTTP is required. See chapter configuration. Also ArangoDB 3.0 does not support geospatial queries.

Maven

To use Spring Data ArangoDB in your project, your build automation tool needs to be configured to include and use the Spring Data ArangoDB dependency. Example with Maven:

```
<dependency>
  <groupId>com.arangodb</groupId>
  <artifactId>arangodb-spring-data</artifactId>
  <version>{version}</version>
</dependency>
```

There is a demonstration app, which contains common use cases and examples of how to use Spring Data ArangoDB's functionality.

Configuration

You can use Java to configure your Spring Data environment as show below. Setting up the underlying driver (`ArangoDB.Builder`) with default configuration automatically loads a properties file `arangodb.properties` , if it exists in the classpath.

```
@Configuration
@EnableArangoRepositories(basePackages = { "com.company.mypackage" })
public class MyConfiguration extends AbstractArangoConfiguration {

  @Override
  public ArangoDB.Builder arango() {
    return new ArangoDB.Builder();
  }

  @Override
  public String database() {
    // Name of the database to be used
    return "example-database";
  }

}
```

The driver is configured with some default values:

property-key	description	default value
arangodb.host	ArangoDB host	127.0.0.1
arangodb.port	ArangoDB port	8529
arangodb.timeout	socket connect timeout(millisecond)	0
arangodb.user	Basic Authentication User	

arangodb.password	Basic Authentication Password	
arangodb.useSsl	use SSL connection	false

To customize the configuration, the parameters can be changed in the Java code.

```
@Override
public ArangoDB.Builder arango() {
  ArangoDB.Builder arango = new ArangoDB.Builder()
    .host("127.0.0.1")
    .port(8429)
    .user("root");
  return arango;
}
```

In addition you can use the *arangodb.properties* or a custom properties file to supply credentials to the driver.

Properties file

```
arangodb.host=127.0.0.1
arangodb.port=8529
# arangodb.hosts=127.0.0.1:8529 could be used instead
arangodb.user=root
arangodb.password=
```

Custom properties file

```
@Override
public ArangoDB.Builder arango() {
  InputStream in = MyClass.class.getResourceAsStream("my.properties");
  ArangoDB.Builder arango = new ArangoDB.Builder()
    .loadProperties(in);
  return arango;
}
```

Note: When using ArangoDB 3.0 it is required to set the transport protocol to HTTP and fetch the dependency `org.apache.httpcomponents:httpclient`.

```
@Override
public ArangoDB.Builder arango() {
  ArangoDB.Builder arango = new ArangoDB.Builder()
    .useProtocol(Protocol.HTTP_JSON);
  return arango;
}
```

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.1</version>
</dependency>
```

Spring Data ArangoDB - Reference

Template

With `ArangoTemplate` Spring Data ArangoDB offers a central support for interactions with the database over a rich feature set. It mostly offers the features from the ArangoDB Java driver with additional exception translation from the drivers exceptions to the Spring Data access exceptions inheriting the `DataAccessException` class. The `ArangoTemplate` class is the default implementation of the operations interface `ArangoOperations` which developers of Spring Data are encouraged to code against.

Repositories

Introduction

Spring Data Commons provides a composable repository infrastructure which Spring Data ArangoDB is built on. These allow for interface-based composition of repositories consisting of provided default implementations for certain interfaces (like `CrudRepository`) and custom implementations for other methods.

Instantiating

Instances of a Repository are created in Spring beans through the auto-wired mechanism of Spring.

```
public class MySpringBean {

  @Autowired
  private MyRepository rep;

}
```

Return types

The method return type for single results can be a primitive type, a domain class, `Map<String, Object>`, `BaseDocument`, `BaseEdgeDocument`, `Optional<Type>`, `GeoResult<Type>`. The method return type for multiple results can additionally be `ArangoCursor<Type>`, `Iterable<Type>`, `Collection<Type>`, `List<Type>`, `Set<Type>`, `Page<Type>`, `Slice<Type>`, `GeoPage<Type>`, `GeoResults<Type>` where Type can be everything a single result can be.

Query methods

Queries using ArangoDB Query Language (AQL) can be supplied with the `@Query` annotation on methods. `AqlQueryOptions` can also be passed to the driver, as an argument anywhere in the method signature.

There are three ways of passing bind parameters to the query in the query annotation.

Using number matching, arguments will be substituted into the query in the order they are passed to the query method.

```
public interface MyRepository extends Repository<Customer, String>{

  @Query("FOR c IN customers FILTER c.name == @0 AND c.surname == @2 RETURN c")
  ArangoCursor<Customer> query(String name, AqlQueryOptions options, String surname);

}
```

With the `@Param` annotation, the argument will be placed in the query at the place corresponding to the value passed to the `@Param` annotation.

```
public interface MyRepository extends Repository<Customer, String>{

  @Query("FOR c IN customers FILTER c.name == @name AND c.surname == @surname RETURN c")
  ArangoCursor<Customer> query(@Param("name") String name, @Param("surname") String surname);

}
```

In addition you can use a parameter of type `Map<String, Object>` annotated with `@BindVars` as your bind parameters. You can then fill the map with any parameter used in the query. (see here for more Information about Bind Parameters).

```
public interface MyRepository extends Repository<Customer, String>{

  @Query("FOR c IN customers FILTER c.name == @name AND c.surname = @surname RETURN c")
  ArangoCursor<Customer> query(@BindVars Map<String, Object> bindVars);

}
```

A mixture of any of these methods can be used. Parameters with the same name from an `@Param` annotation will override those in the `bindVars`.

```
public interface MyRepository extends Repository<Customer, String>{

  @Query("FOR c IN customers FILTER c.name == @name AND c.surname = @surname RETURN c")
  ArangoCursor<Customer> query(@BindVars Map<String, Object> bindVars, @Param("name") String name);

}
```

Named queries

An alternative to using the `@Query` annotation on methods is specifying them in a separate `.properties` file. The default path for the file is `META-INF/arango-named-queries.properties` and can be changed with the `EnableArangoRepositories#namedQueriesLocation()` setting. The entries in the properties file must adhere to the following convention: `{simple entity name}.{method name} = {query}`. Let's assume we have the following repository interface:

```
package com.arangodb.repository;

public interface CustomerRepository extends ArangoRepository<Customer> {
    Customer findByUsername(@Param("username") String username);
}
```

The corresponding `arango-named-queries.properties` file looks like this:

```
Customer.findByUsername = FOR c IN customers FILTER c.username == @username RETURN c
```

The queries specified in the properties file are no different than the queries that can be defined with the `@Query` annotation. The only difference is that the queries are in one place. If there is a `@Query` annotation present and a named query defined, the query in the `@Query` annotation takes precedence.

Derived queries

Spring Data ArangoDB supports queries derived from methods names by splitting it into its semantic parts and converting into AQL. The mechanism strips the prefixes `find..By`, `get..By`, `query..By`, `read..By`, `stream..By`, `count..By`, `exists..By`, `delete..By`, `remove..By` from the method and parses the rest. The By acts as a separator to indicate the start of the criteria for the query to be built. You can define conditions on entity properties and concatenate them with `And` and `Or`.

The complete list of part types for derived methods is below, where doc is a document in the database

Keyword	Sample	Predicate
IsGreaterThan, GreaterThan, After	findByAgeGreaterThan(int age)	doc.age > age
IsGreaterThanEqual, GreaterThanEqual	findByAgeIsGreaterThanEqual(int age)	doc.age >= age
IsLessThan, LessThan, Before	findByAgeIsLessThan(int age)	doc.age < age
IsLessThanEqualLessThanEqual	findByAgeLessThanEqual(int age)	doc.age <= age
IsBetween, Between	findByAgeBetween(int lower, int upper)	lower < doc.age < upper
IsNotNull, NotNull	findByNameNotNull()	doc.name != null
IsNull, Null	findByNameNull()	doc.name == null
IsLike, Like	findByNameLike(String name)	doc.name LIKE name
IsNotLike, NotLike	findByNameNotLike(String name)	NOT(doc.name LIKE name)
IsStartingWith, StartingWith, StartsWith	findByNameStartsWith(String prefix)	doc.name LIKE prefix
IsEndingWith, EndingWith, EndsWith	findByNameEndingWith(String suffix)	doc.name LIKE suffix
Regex, MatchesRegex, Matches	findByNameRegex(String pattern)	REGEX_TEST(doc.name, name, ignoreCase)
(No Keyword)	findByFirstName(String name)	doc.name == name
IsTrue, True	findByActiveTrue()	doc.active == true
IsFalse, False	findByActiveFalse()	doc.active == false
Is, Equals	findByAgeEquals(int age)	doc.age == age
IsNot, Not	findByAgeNot(int age)	doc.age != age
IsIn, In	findByNameIn(String[] names)	doc.name IN names
IsNotIn, NotIn	findByNameIsNotIn(String[] names)	doc.name NOT IN names
IsContaining, Containing, Contains	findByFriendsContaining(String name)	name IN doc.friends
IsNotContaining, NotContaining, NotContains	findByFriendsNotContains(String name)	name NOT IN doc.friends
Exists	findByFriendNameExists()	HAS(doc.friend, name)

```
public interface MyRepository extends Repository<Customer, String> {

  // FOR c IN customers FILTER c.name == @0 RETURN c
  ArangoCursor<Customer> findByName(String name);
  ArangoCursor<Customer> getByName(String name);

  // FOR c IN customers
  // FILTER c.name == @0 && c.age == @1
  // RETURN c
  ArangoCursor<Customer> findByNameAndAge(String name, int age);

  // FOR c IN customers
  // FILTER c.name == @0 || c.age == @1
  // RETURN c
  ArangoCursor<Customer> findByNameOrAge(String name, int age);
}
```

You can apply sorting for one or multiple sort criteria by appending `OrderBy` to the method and `Asc` or `Desc` for the directions.

```
public interface MyRepository extends Repository<Customer, String> {

  // FOR c IN customers
```

```
  // FILTER c.name == @0
  // SORT c.age DESC RETURN c
  ArangoCursor<Customer> getByNameOrderByAgeDesc(String name);

  // FOR c IN customers
  // FILTER c.name = @0
  // SORT c.name ASC, c.age DESC RETURN c
  ArangoCursor<Customer> findByNameOrderByNameAscAgeDesc(String name);

}
```

Geospatial queries

Geospatial queries are a subsection of derived queries. To use a geospatial query on a collection, a geo index must exist on that collection. A geo index can be created on a field which is a two element array, corresponding to latitude and longitude coordinates.

As a subsection of derived queries, geospatial queries support all the same return types, but also support the three return types `GeoPage, GeoResult and Georesults`. These types must be used in order to get the distance of each document as generated by the query.

There are two kinds of geospatial query, Near and Within. Near sorts documents by distance from the given point, while within both sorts and filters documents, returning those within the given distance range or shape.

```
public interface MyRepository extends Repository<City, String> {

    GeoResult<City> getByLocationNear(Point point);

    GeoResults<City> findByLocationWithinOrLocationWithin(Box box, Polygon polygon);

    //Equivalent queries
    GeoResults<City> findByLocationWithinOrLocationWithin(Point point, int distance);
    GeoResults<City> findByLocationWithinOrLocationWithin(Point point, Distance distance);
    GeoResults<City> findByLocationWithinOrLocationWithin(Circle circle);

}
```

Property expression

Property expressions can refer only to direct and nested properties of the managed domain class. The algorithm checks the domain class for the entire expression as the property. If the check fails, the algorithm splits up the expression at the camel case parts from the right and tries to find the corresponding property.

```
@Document("customers")
public class Customer {
  private Address address;
}

public class Address {
  private ZipCode zipCode;
}

public interface MyRepository extends Repository<Customer, String> {

  // 1. step: search domain class for a property "addressZipCode"
  // 2. step: search domain class for "addressZip.code"
  // 3. step: search domain class for "address.zipCode"
  ArangoCursor<Customer> findByAddressZipCode(ZipCode zipCode);
}
```

It is possible for the algorithm to select the wrong property if the domain class also has a property which matches the first split of the expression. To resolve this ambiguity you can use _ as a separator inside your method-name to define traversal points.

```
@Document("customers")
public class Customer {
  private Address address;
  private AddressZip addressZip;
}
```

```
public class Address {
  private ZipCode zipCode;
}

public class AddressZip {
  private String code;
}

public interface MyRepository extends Repository<Customer, String> {

  // 1. step: search domain class for a property "addressZipCode"
  // 2. step: search domain class for "addressZip.code"
  // creates query with "x.addressZip.code"
  ArangoCursor<Customer> findByAddressZipCode(ZipCode zipCode);

  // 1. step: search domain class for a property "addressZipCode"
  // 2. step: search domain class for "addressZip.code"
  // 3. step: search domain class for "address.zipCode"
  // creates query with "x.address.zipCode"
  ArangoCursor<Customer> findByAddress_ZipCode(ZipCode zipCode);

}
```

Special parameter handling

Bind parameters

AQL supports the usage of bind parameters which you can define with a method parameter annotated with `@BindVars` of type `Map<String, Object>` .

```
public interface MyRepository extends Repository<Customer, String> {

  @Query("FOR c IN customers FILTER c[@field] == @value RETURN c")
  ArangoCursor<Customer> query(Map<String, Object> bindVars);

}

Map<String, Object> bindVars = new HashMap<String, Object>();
bindVars.put("field", "name");
bindVars.put("value", "john";

// will execute query "FOR c IN customers FILTER c.name == "john" RETURN c"
ArangoCursor<Customer> cursor = myRepo.query(bindVars);
```

AQL query options

You can set additional options for the query and the created cursor over the class `AqlQueryOptions` which you can simply define as a method parameter without a specific name. AqlQuery options can also be defined with the `@QueryOptions` annotation, as shown below. AqlQueryOptions from an annotation and those from an argument are merged if both exist, with those in the argument taking precedence.

The `AqlQueryOptions` allows you to set the cursor time-to-live, batch-size, caching flag and several other settings. This special parameter works with both query-methods and finder-methods. Keep in mind that some options, like time-to-live, are only effective if the method return type is `ArangoCursor<T>` or `Iterable<T>` .

```
public interface MyRepository extends Repository<Customer, String> {


  @Query("FOR c IN customers FILTER c.name == @0 RETURN c")
  Iterable<Customer> query(String name, AqlQueryOptions options);


  Iterable<Customer> findByName(String name, AqlQueryOptions options);


  @QueryOptions(maxPlans = 1000, ttl = 128)
  ArangoCursor<Customer> findByAddressZipCode(ZipCode zipCode);
```

```
  @Query("FOR c IN customers FILTER c[@field] == @value RETURN c")
  @QueryOptions(cache = true, ttl = 128)
  ArangoCursor<Customer> query(Map<String, Object> bindVars, AqlQueryOptions options);

}
```

Paging and sorting

Spring Data ArangoDB supports Spring Data's `Pageable` and `Sort` parameters for repository query methods. If these parameters are used together with a native query, either through `@Query` annotation or named queries, a placeholder must be specified:

- `#pageable` for `Pageable` parameter
- `#sort` for `Sort` parameter

Sort properties or paths are attributes separated by dots (e.g. `customer.age`). Some rules apply for them:

- they must not begin or end with a dot (e.g. `.customer.age`)
- dots in attributes are supported, but the whole attribute must be enclosed by backticks (e.g. `customer.`attr.with.dots``
- backticks in attributes are supported, but they must be escaped with a backslash (e.g. `customer.attr_with\``)
- any backslashes (that do not escape a backtick) are escaped (e.g. `customer\` => `customer\\`)

```
just.`some`.`attributes.that`.`form\``.a path\`.\  is converted to
`just`.`some`.`attributes.that`.`form\``.`a path\``.`\\`
```

Native queries example:

```
public interface CustomerRepository extends ArangoRepository<Customer> {

  @Query("FOR c IN customer FILTER c.name == @1 #pageable RETURN c")
  Page<Customer> findByNameNative(Pageable pageable, String name);

  @Query("FOR c IN customer FILTER c.name == @1 #sort RETURN c")
  List<Customer> findByNameNative(Sort sort, String name);
}

// don't forget to specify the var name of the document
final Pageable page = PageRequest.of(1, 10, Sort.by("c.age"));
repository.findByNameNative(page, "Matt");

final Sort sort = Sort.by(Direction.DESC, "c.age");
repository.findByNameNative(sort, "Tony");
```

Derived queries example:

```
public interface CustomerRepository extends ArangoRepository<Customer> {

  Page<Customer> findByName(Pageable pageable, String name);

  List<Customer> findByName(Sort sort, String name);
}

// no var name is necessary for derived queries
final Pageable page = PageRequest.of(1, 10, Sort.by("age"));
repository.findByName(page, "Matt");

final Sort sort = Sort.by(Direction.DESC, "age");
repository.findByName(sort, "Tony");
```

Mapping

Introduction

In this section we will describe the features and conventions for mapping Java objects to documents and how to override those conventions with annotation based mapping metadata.

Conventions

- The Java class name is mapped to the collection name
- The non-static fields of a Java object are used as fields in the stored document
- The Java field name is mapped to the stored document field name
- All nested Java object are stored as nested objects in the stored document
- The Java class needs a constructor which meets the following criteria:
  - in case of a single constructor:
    - a non-parameterized constructor or
    - a parameterized constructor
  - in case of multiple constructors:
    - a non-parameterized constructor or
    - a parameterized constructor annotated with `@PersistenceConstructor`

Type conventions

ArangoDB uses VelocyPack as it's internal storage format which supports a large number of data types. In addition Spring Data ArangoDB offers - with the underlying Java driver - built-in converters to add additional types to the mapping.

Java type	VelocyPack type
java.lang.String	string
java.lang.Boolean	bool
java.lang.Integer	signed int 4 bytes, smallint
java.lang.Long	signed int 8 bytes, smallint
java.lang.Short	signed int 2 bytes, smallint
java.lang.Double	double
java.lang.Float	double
java.math.BigInteger	signed int 8 bytes, unsigned int 8 bytes
java.math.BigDecimal	double
java.lang.Number	double
java.lang.Character	string
java.util.Date	string (date-format ISO 8601)
java.sql.Date	string (date-format ISO 8601)
java.sql.Timestamp	string (date-format ISO 8601)
java.util.UUID	string
java.lang.byte[]	string (Base64)

Type mapping

As collections in ArangoDB can contain documents of various types, a mechanism to retrieve the correct Java class is required. The type information of properties declared in a class may not be enough to restore the original class (due to inheritance). If the declared complex type and the actual type do not match, information about the actual type is stored together with the document. This is necessary to restore the correct type when reading from the DB. Consider the following example:

```
public class Person {
    private String name;
    private Address homeAddress;
    // ...

    // getters and setters omitted
}

public class Employee extends Person {
    private Address workAddress;
    // ...

    // getters and setters omitted
}

public class Address {
    private final String street;
    private final String number;
    // ...

    public Address(String street, String number) {
        this.street = street;
        this.number = number;
    }

    // getters omitted
}

@Document
public class Company {
    @Key
    private String key;
    private Person manager;

    // getters and setters omitted
}

Employee manager = new Employee();
manager.setName("Jane Roberts");
manager.setHomeAddress(new Address("Park Avenue", "432/64"));
manager.setWorkAddress(new Address("Main Street",  "223"));
Company comp = new Company();
comp.setManager(manager);
```

The serialized document for the DB looks like this:

```
{
  "manager": {
    "name": "Jane Roberts",
    "homeAddress": {
        "street": "Park Avenue",
        "number": "432/64"
    },
    "workAddress": {
        "street": "Main Street",
        "number": "223"
    },
    "_class": "com.arangodb.Employee"
  },
  "_class": "com.arangodb.Company"
}
```

Type hints are written for top-level documents (as a collection can contain different document types) as well as for every value if it's a complex type and a sub-type of the property type declared. `Map`s and `Collection`s are excluded from type mapping. Without the additional information about the concrete classes used, the document couldn't be restored in Java. The type information of the `manager` property is not enough to determine the `Employee` type. The `homeAddress` and `workAddress` properties have the same actual and defined type, thus no type hint is needed.

Customizing type mapping

By default, the fully qualified class name is stored in the documents as a type hint. A custom type hint can be set with the `@TypeAlias("my-alias")` annotation on an entity. Make sure that it is an unique identifier across all entities. If we would add a `TypeAlias("employee")` annotation to the `Employee` class above, it would be persisted as `"_class": "employee"`.

The default type key is `_class` and can be changed by overriding the `typeKey()` method of the `AbstractArangoConfiguration` class.

If you need to further customize the type mapping process, the `arangoTypeMapper()` method of the configuration class can be overridden. The included `DefaultArangoTypeMapper` can be customized by providing a list of `TypeInformationMapper`s that create aliases from types and vice versa.

In order to fully customize the type mapping process you can provide a custom type mapper implementation by extending the `DefaultArangoTypeMapper` class.

Deactivating type mapping

To deactivate the type mapping process, you can return `null` from the `typeKey()` method of the `AbstractArangoConfiguration` class. No type hints are stored in the documents with this setting. If you make sure that each defined type corresponds to the actual type, you can disable the type mapping, otherwise it can lead to exceptions when reading the entities from the DB.

Annotations

Annotation overview

annotation	level	description
@Document	class	marks this class as a candidate for mapping
@Edge	class	marks this class as a candidate for mapping
@Id	field	stores the field as the system field _id
@Key	field	stores the field as the system field _key
@Rev	field	stores the field as the system field _rev
@Field("alt-name")	field	stores the field with an alternative name
@Ref	field	stores the _id of the referenced document and not the nested document
@From	field	stores the _id of the referenced document as the system field _from
@To	field	stores the _id of the referenced document as the system field _to
@Relations	field	vertices which are connected over edges
@Transient	field, method, annotation	marks a field to be transient for the mapping framework, thus the property will not be persisted and not further inspected by the mapping framework
@PersistenceConstructor	constructor	marks a given constructor - even a package protected one - to use when instantiating the object from the database
@TypeAlias("alias")	class	set a type alias for the class when persisted to the DB
@HashIndex	class	describes a hash index
@HashIndexed	field	describes how to index the field
@SkiplistIndex	class	describes a skiplist index
@SkiplistIndexed	field	describes how to index the field
@PersistentIndex	class	describes a persistent index
@PersistentIndexed	field	describes how to index the field
@GeoIndex	class	describes a geo index
@GeoIndexed	field	describes how to index the field

@FulltextIndex	class	describes a fulltext index
@FulltextIndexed	field	describes how to index the field

Document

The annotations `@Document` applied to a class marks this class as a candidate for mapping to the database. The most relevant parameter is `value` to specify the collection name in the database. The annotation `@Document` specifies the collection type to `DOCUMENT` .

```
@Document(value="persons")
public class Person {
  ...
}
```

Edge

The annotations `@Edge` applied to a class marks this class as a candidate for mapping to the database. The most relevant parameter is `value` to specify the collection name in the database. The annotation `@Edge` specifies the collection type to `EDGE` .

```
@Edge("relations")
public class Relation {
  ...
}
```

Reference

With the annotation `@Ref` applied on a field the nested object isn't stored as a nested object in the document. The `_id` field of the nested object is stored in the document and the nested object has to be stored as a separate document in another collection described in the `@Document` annotation of the nested object class. To successfully persist an instance of your object the referencing field has to be null or it's instance has to provide a field with the annotation `@Id` including a valid id.

```
@Document(value="persons")
public class Person {
  @Ref
  private Address address;
}

@Document("addresses")
public class Address {
  @Id
  private String id;
  private String country;
  private String street;
}
```

The database representation of `Person` in collection *persons* looks as follow:

```
{
  "_key" : "123",
  "_id" : "persons/123",
  "address" : "addresses/456"
}
```

and the representation of `Address` in collection *addresses*:

```
{
  "_key" : "456",
  "_id" : "addresses/456",
  "country" : "...",
  "street" : "..."
}
```

Without the annotation `@Ref` at the field `address` , the stored document would look:

```
{
  "_key" : "123",
  "_id" : "persons/123",
  "address" : {
    "country" : "...",
     "street" : "..."
  }
}
```

Relations

With the annotation `@Relations` applied on a collection or array field in a class annotated with `@Document` the nested objects are fetched from the database over a graph traversal with your current object as the starting point. The most relevant parameter is `edge`. With `edge` you define the edge collection - which should be used in the traversal - using the class type. With the parameter `depth` you can define the maximal depth for the traversal (default 1) and the parameter `direction` defines whether the traversal should follow outgoing or incoming edges (default Direction.ANY).

```
@Document(value="persons")
public class Person {
  @Relations(edge=Relation.class, depth=1, direction=Direction.ANY)
  private List<Person> friends;
}

@Edge(name="relations")
public class Relation {

}
```

Document with From and To

With the annotations `@From` and `@To` applied on a collection or array field in a class annotated with `@Document` the nested edge objects are fetched from the database. Each of the nested edge objects has to be stored as separate edge document in the edge collection described in the `@Edge` annotation of the nested object class with the *_id* of the parent document as field *_from* or *_to*.

```
@Document("persons")
public class Person {
  @From
  private List<Relation> relations;
}

@Edge(name="relations")
public class Relation {
  ...
}
```

The database representation of `Person` in collection *persons* looks as follow:

```
{
  "_key" : "123",
  "_id" : "persons/123"
}
```

and the representation of `Relation` in collection *relations*:

```
{
  "_key" : "456",
  "_id" : "relations/456",
  "_from" : "persons/123"
  "_to" : ".../..."
}
{
  "_key" : "789",
  "_id" : "relations/456",
  "_from" : "persons/123"
  "_to" : ".../..."
}
```

```
...
```

Edge with From and To

With the annotations `@From` and `@To` applied on a field in a class annotated with `@Edge` the nested object is fetched from the database. The nested object has to be stored as a separate document in the collection described in the `@Document` annotation of the nested object class. The *_id* field of this nested object is stored in the fields `_from` or `_to` within the edge document.

```
@Edge("relations")
public class Relation {
  @From
  private Person c1;
  @To
  private Person c2;
}

@Document(value="persons")
public class Person {
  @Id
  private String id;
}
```

The database representation of `Relation` in collection *relations* looks as follow:

```
{
  "_key" : "123",
  "_id" : "relations/123",
  "_from" : "persons/456",
  "_to" : "persons/789"
}
```

and the representation of `Person` in collection *persons*:

```
{
  "_key" : "456",
  "_id" : "persons/456",
}
{
  "_key" : "789",
  "_id" : "persons/789",
}
```

Note: If you want to save an instance of `Relation`, both `Person` objects (from & to) already have to be persisted and the class `Person` needs a field with the annotation `@Id` so it can hold the persisted `_id` from the database.

Index and Indexed annotations

With the `@<IndexType>Indexed` annotations user defined indexes can be created at a collection level by annotating single fields of a class.

Possible `@<IndexType>Indexed` annotations are:

- `@HashIndexed`
- `@SkiplistIndexed`
- `@PersistentIndexed`
- `@GeoIndexed`
- `@FulltextIndexed`

The following example creates a hash index on the field `name` and a separate hash index on the field `age`:

```
public class Person {
  @HashIndexed
  private String name;

  @HashIndexed
  private int age;
```

```
}
```

With the `@<IndexType>Indexed` annotations different indexes can be created on the same field.

The following example creates a hash index and also a skiplist index on the field `name`:

```
public class Person {
  @HashIndexed
  @SkiplistIndexed
  private String name;
}
```

If the index should include multiple fields the `@<IndexType>Index` annotations can be used on the type instead.

Possible `@<IndexType>Index` annotations are:

- `@HashIndex`
- `@SkiplistIndex`
- `@PersistentIndex`
- `@GeoIndex`
- `@FulltextIndex`

The following example creates a single hash index on the fields `name` and `age`, note that if a field is renamed in the database with @Field, the new field name must be used in the index declaration:

```
@HashIndex(fields = {"fullname", "age"})
public class Person {
  @Field("fullname")
  private String name;

  private int age;
}
```

The `@<IndexType>Index` annotations can also be used to create an index on a nested field.

The following example creates a single hash index on the fields `name` and `address.country`:

```
@HashIndex(fields = {"name", "address.country"})
public class Person {
  private String name;

  private Address address;
}
```

The `@<IndexType>Index` annotations and the `@<IndexType>Indexed` annotations can be used at the same time in one class.

The following example creates a hash index on the fields `name` and `age` and a separate hash index on the field `age`:

```
@HashIndex(fields = {"name", "age"})
public class Person {
  private String name;

  @HashIndexed
  private int age;
}
```

The `@<IndexType>Index` annotations can be used multiple times to create more than one index in this way.

The following example creates a hash index on the fields `name` and `age` and a separate hash index on the fields `name` and `gender`:

```
@HashIndex(fields = {"name", "age"})
@HashIndex(fields = {"name", "gender"})
public class Person {
  private String name;

  private int age;
```

```
  private Gender gender
}
```

ArangoDB-PHP - A PHP client for ArangoDB

The official ArangoDB PHP Driver.

- Getting Started
- Tutorial
- Changelog

More information

- Check the ArangoDB PHP client on github.com regularly for new releases and updates: https://github.com/arangodb/arangodb-php
- More example code, containing some code to create, delete and rename collections, is provided in the examples subdirectory that is provided with the library.
- PHPDoc documentation for the complete library is in the library's docs subdirectory. Point your browser at this directory to get a click-through version of the documentation after cloning the repository.
- Follow us on Twitter @arangodbphp to receive updates on the PHP driver

ArangoDB-PHP - Getting Started

Description

This PHP client allows REST-based access to documents on the server. The *DocumentHandler* class should be used for these purposes. There is an example for REST-based documents access in the file examples/document.php.

Furthermore, the PHP client also allows to issue more AQL complex queries using the *Statement* class. There is an example for this kind of statements in the file examples/select.php.

To use the PHP client, you must include the file autoloader.php from the main directory. The autoloader will care about loading additionally required classes on the fly. The autoloader can be nested with other autoloaders.

The ArangoDB PHP client is an API that allows you to send and retrieve documents from ArangoDB from out of your PHP application. The client library itself is written in PHP and has no further dependencies but just plain PHP 5.6 (or higher).

The client library provides document and collection classes you can use to work with documents and collections in an OO fashion. When exchanging document data with the server, the library internally will use the HTTP REST interface of ArangoDB. The library user does not have to care about this fact as all the details of the REST interface are abstracted by the client library.

Requirements

- PHP version 5.6 or higher (Travis-tested with PHP 5.6, 7.0, 7.1 and hhvm)

Note on PHP version support:

This driver will cease to support old PHP versions as soon as they have reached end-of-life status. Support will be removed with the next minor or patch version of the driver to be released.

In general, it is recommended to always use the latest PHP versions (currently those in the PHP 7 line) in order to take advantage of all the improvements (especially in performance).

Important version information on ArangoDB-PHP

The ArangoDB-PHP driver version has to match with the ArangoDB version:

- ArangoDB-PHP 3.1.x is on par with the functionality of ArangoDB 3.1.x
- ArangoDB-PHP 3.2.x is on par with the functionality of ArangoDB 3.2.x
- ArangoDB-PHP 3.3.x is on par with the functionality of ArangoDB 3.3.x

etc...

Installing the PHP client

To get started you need PHP 5.6 or higher plus an ArangoDB server running on any host that you can access.

There are two alternative ways to get the ArangoDB PHP client:

- Using Composer
- Cloning the git repository

Alternative 1: Using Composer

```
composer require triagens/arangodb
```

Alternative 2: Cloning the git repository

When preferring this alternative, you need to have a git client installed. To clone the ArangoDB PHP client repository from github, execute the following command in your project directory:

```
git clone "https://github.com/arangodb/arangodb-php.git"
```

This will create a subdirectory arangodb-php in your current directory. It contains all the files of the client library. It also includes a dedicated autoloader that you can use for autoloading the client libraries class files. To invoke this autoloader, add the following line to your PHP files that will use the library:

```
require 'arangodb-php/autoload.php';
```

The ArangoDB PHP client's autoloader will only care about its own class files and will not handle any other files. That means it is fully nestable with other autoloaders.

Alternative 3: Invoking the autoloader directly

If you do not wish to include autoload.php to load and setup the autoloader, you can invoke the autoloader directly:

```
require 'arangodb-php/lib/ArangoDBClient/autoloader.php';
\ArangoDBClient\Autoloader::init();
```

ArangoDB-PHP - Tutorial

Setting up the connection options

In order to use ArangoDB, you need to specify the connection options. We do so by creating a PHP array $connectionOptions. Put this code into a file named test.php in your current directory:

```
// use the following line when using Composer
// require __DIR__ . '/vendor/composer/autoload.php';

// use the following line when using git
require __DIR__ . '/arangodb-php/autoload.php';

// set up some aliases for less typing later
use ArangoDBClient\Collection as ArangoCollection;
use ArangoDBClient\CollectionHandler as ArangoCollectionHandler;
use ArangoDBClient\Connection as ArangoConnection;
use ArangoDBClient\ConnectionOptions as ArangoConnectionOptions;
use ArangoDBClient\DocumentHandler as ArangoDocumentHandler;
use ArangoDBClient\Document as ArangoDocument;
use ArangoDBClient\Exception as ArangoException;
use ArangoDBClient\Export as ArangoExport;
use ArangoDBClient\ConnectException as ArangoConnectException;
use ArangoDBClient\ClientException as ArangoClientException;
use ArangoDBClient\ServerException as ArangoServerException;
use ArangoDBClient\Statement as ArangoStatement;
use ArangoDBClient\UpdatePolicy as ArangoUpdatePolicy;

// set up some basic connection options
$connectionOptions = [
    // database name
    ArangoConnectionOptions::OPTION_DATABASE => '_system',
    // server endpoint to connect to
    ArangoConnectionOptions::OPTION_ENDPOINT => 'tcp://127.0.0.1:8529',
    // authorization type to use (currently supported: 'Basic')
    ArangoConnectionOptions::OPTION_AUTH_TYPE => 'Basic',
    // user for basic authorization
    ArangoConnectionOptions::OPTION_AUTH_USER => 'root',
    // password for basic authorization
    ArangoConnectionOptions::OPTION_AUTH_PASSWD => '',
    // connection persistence on server. can use either 'Close' (one-time connections) or 'Keep-Alive' (re-used connections)
    ArangoConnectionOptions::OPTION_CONNECTION => 'Keep-Alive',
    // connect timeout in seconds
    ArangoConnectionOptions::OPTION_TIMEOUT => 3,
    // whether or not to reconnect when a keep-alive connection has timed out on server
    ArangoConnectionOptions::OPTION_RECONNECT => true,
    // optionally create new collections when inserting documents
    ArangoConnectionOptions::OPTION_CREATE => true,
    // optionally create new collections when inserting documents
    ArangoConnectionOptions::OPTION_UPDATE_POLICY => ArangoUpdatePolicy::LAST,
];


// turn on exception logging (logs to whatever PHP is configured)
ArangoException::enableLogging();


    $connection = new ArangoConnection($connectionOptions);
```

This will make the client connect to ArangoDB

- running on localhost (OPTION_HOST)
- on the default port 8529 (OPTION_PORT)
- with a connection timeout of 3 seconds (OPTION_TIMEOUT)

When creating new documents in a collection that does not yet exist, you have the following choices:

- auto-generate a new collection: if you prefer that, set OPTION_CREATE to true

- fail with an error: if you prefer this behavior, set OPTION_CREATE to false

When updating a document that was previously/concurrently updated by another user, you can select between the following behaviors:

- last update wins: if you prefer this, set OPTION_UPDATE_POLICY to last
- fail with a conflict error: if you prefer that, set OPTION_UPDATE_POLICY to conflict

Setting up active failover

By default the PHP client will connect to a single endpoint only, by specifying a string value for the endpoint in the `ConnectionOptions`, e.g.

```
$connectionOptions = [
    ArangoConnectionOptions::OPTION_ENDPOINT => 'tcp://127.0.0.1:8529'
];
```

To set up multiple servers to connect to, it is also possible to specify an array of servers instead:

```
$connectionOptions = [
    ConnectionOptions::OPTION_ENDPOINT    => [ 'tcp://localhost:8531', 'tcp://localhost:8532', 'tcp://localhost:8530' ]
];
```

Using this option requires ArangoDB 3.3 or higher and the database running in active failover mode.

The driver will by default try to connect to the first server endpoint in the endpoints array, and only try the following servers if no connection can be established. If no connection can be made to any server, the driver will throw an exception.

As it is unknown to the driver which server from the array is the current leader, the driver will connect to the specified servers in array order by default. However, to spare a few unnecessary connection attempts to failed servers, it is possible to set up caching (using Memcached) for the server list. The cached value will contain the last working server first, so that as few connection attempts as possible will need to be made.

In order to use this caching, it is required to install the Memcached module for PHP, and to set up the following relevant options in the `ConnectionOptions`:

```
$connectionOptions = [
    // memcached persistent id (will be passed to Memcached::__construct)
    ConnectionOptions::OPTION_MEMCACHED_PERSISTENT_ID => 'arangodb-php-pool',

    // memcached servers to connect to (will be passed to Memcached::addServers)
    ConnectionOptions::OPTION_MEMCACHED_SERVERS       => [ [ '127.0.0.1', 11211 ] ],

    // memcached options (will be passed to Memcached::setOptions)
    ConnectionOptions::OPTION_MEMCACHED_OPTIONS       => [ ],

    // key to store the current endpoints array under
    ConnectionOptions::OPTION_MEMCACHED_ENDPOINTS_KEY => 'arangodb-php-endpoints'

    // time-to-live for the endpoints array stored in memcached
    ConnectionOptions::OPTION_MEMCACHED_TTL           => 600
];
```

Creating a collection

This is just to show how a collection is created. For these examples it is not needed to create a collection prior to inserting a document, as we set ArangoConnectionOptions::OPTION_CREATE to true.

So, after we get the settings, we can start with creating a collection. We will create a collection named "users".

The below code will first set up the collection locally in a variable name $user, and then push it to the server and return the collection id created by the server:

```
    $collectionHandler = new ArangoCollectionHandler($connection);
```

```
// clean up first
if ($collectionHandler->has('users')) {
    $collectionHandler->drop('users');
}
if ($collectionHandler->has('example')) {
    $collectionHandler->drop('example');
}

// create a new collection
$userCollection = new ArangoCollection();
$userCollection->setName('users');
$id = $collectionHandler->create($userCollection);

// print the collection id created by the server
var_dump($id);
// check if the collection exists
$result = $collectionHandler->has('users');
var_dump($result);
```

Creating a document

After we created the collection, we can start with creating an initial document. We will create a user document in a collection named "users". This collection does not need to exist yet. The first document we'll insert in this collection will create the collection on the fly. This is because we have set OPTION_CREATE to true in $connectionOptions.

The below code will first set up the document locally in a variable name $user, and then push it to the server and return the document id created by the server:

```
$handler = new ArangoDocumentHandler($connection);

// create a new document
$user = new ArangoDocument();

// use set method to set document properties
$user->set('name', 'John');
$user->set('age', 25);
$user->set('thisIsNull', null);

// use magic methods to set document properties
$user->likes = ['fishing', 'hiking', 'swimming'];

// send the document to the server
$id = $handler->save('users', $user);

// check if a document exists
$result = $handler->has('users', $id);
var_dump($result);

// print the document id created by the server
var_dump($id);
var_dump($user->getId());
```

Document properties can be set by using the set() method, or by directly manipulating the document properties.

As you can see, sending a document to the server is achieved by calling the save() method on the client library's *DocumentHandler* class. It needs the collection name ("users" in this case") plus the document object to be saved. save() will return the document id as created by the server. The id is a numeric value that might or might not fit in a PHP integer.

Adding exception handling

The above code will work but it does not check for any errors. To make it work in the face of errors, we'll wrap it into some basic exception handlers

```
try {
     $handler = new ArangoDocumentHandler($connection);
```

```
        // create a new document
        $user = new ArangoDocument();

        // use set method to set document properties
        $user->set('name', 'John');
        $user->set('age', 25);

        // use magic methods to set document properties
        $user->likes = ['fishing', 'hiking', 'swimming'];

        // send the document to the server
        $id = $handler->save('users', $user);

        // check if a document exists
        $result = $handler->has('users', $id);
        var_dump($result);

        // print the document id created by the server
        var_dump($id);
        var_dump($user->getId());
} catch (ArangoConnectException $e) {
  print 'Connection error: ' . $e->getMessage() . PHP_EOL;
} catch (ArangoClientException $e) {
  print 'Client error: ' . $e->getMessage() . PHP_EOL;
} catch (ArangoServerException $e) {
  print 'Server error: ' . $e->getServerCode() . ':' . $e->getServerMessage() . ' ' . $e->getMessage() . PHP_EOL;
}
```

Retrieving a document

To retrieve a document from the server, the get() method of the *DocumentHandler* class can be used. It needs the collection name plus a document id. There is also the getById() method which is an alias for get().

```
    // get the document back from the server
    $userFromServer = $handler->get('users', $id);
    var_dump($userFromServer);

/*
The result of the get() method is a Document object that you can use in an OO fashion:

object(ArangoDBClient\Document)##6 (4) {
    ["_id":"ArangoDBClient\Document":private]=>
    string(15) "2377907/4818344"
    ["_rev":"ArangoDBClient\Document":private]=>
    int(4818344)
    ["_values":"ArangoDBClient\Document":private]=>
    array(3) {
        ["age"]=>
        int(25)
        ["name"]=>
        string(4) "John"
        ["likes"]=>
        array(3) {
            [0]=>
            string(7) "fishing"
            [1]=>
            string(6) "hiking"
            [2]=>
            string(8) "swimming"
        }
    }
    ["_changed":"ArangoDBClient\Document":private]=>
    bool(false)
}
*/
```

Whenever the document id is yet unknown, but you want to fetch a document from the server by any of its other properties, you can use the CollectionHandler->byExample() method. It allows you to provide an example of the document that you are looking for. The example should either be a Document object with the relevant properties set, or, a PHP array with the propeties that you are looking for:

```
// get a document list back from the server, using a document example
$cursor = $collectionHandler->byExample('users', ['name' => 'John']);
var_dump($cursor->getAll());
```

This will return all documents from the specified collection (here: "users") with the properties provided in the example (here: that have an attribute "name" with a value of "John"). The result is a cursor which can be iterated sequentially or completely. We have chosen to get the complete result set above by calling the cursor's getAll() method. Note that CollectionHandler->byExample() might return multiple documents if the example is ambigious.

Updating a document

To update an existing document, the update() method of the *DocumentHandler* class can be used. In this example we want to

- set state to 'ca'
- change the `likes` array.

```
// update a document
$userFromServer->likes = ['fishing', 'swimming'];
$userFromServer->state = 'CA';

$result = $handler->update($userFromServer);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);
```

To remove an attribute using the update() method, an option has to be passed telling it to not keep attributes with null values. In this example we want to

- remove the `age`

```
// update a document removing an attribute,
// The 'keepNull'=>false option will cause ArangoDB to
// remove all attributes in the document,
// that have null as their value - not only the ones defined here

$userFromServer->likes = ['fishing', 'swimming'];
$userFromServer->state = 'CA';
$userFromServer->age   = null;

$result = $handler->update($userFromServer, ['keepNull' => false]);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);
```

To completely replace an existing document, the replace() method of the *DocumentHandler* class can be used. In this example we want to remove the `state` attribute.

```
// replace a document (notice that we are using the previously fetched document)
// In this example we are removing the state attribute
unset($userFromServer->state);

$result = $handler->replace($userFromServer);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);
```

The document that is replaced using the previous example must have been fetched from the server before. If you want to update a document without having fetched it from the server before, use updateById():

```
// replace a document, identified by collection and document id
$user        = new ArangoDocument();
```

```
$user->name  = 'John';
$user->likes = ['Running', 'Rowing'];
$userFromServer->state = 'CA';

// Notice that for the example we're getting the existing
// document id via a method call. Normally we would use the known id
$result = $handler->replaceById('users', $userFromServer->getId(), $user);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);
```

Deleting a document

To remove an existing document on the server, the remove() method of the *DocumentHandler* class will do. remove() just needs the document to be removed as a parameter:

```
// remove a document on the server, using a document object
$result = $handler->remove($userFromServer);
var_dump($result);
```

Note that the document must have been fetched from the server before. If you haven't fetched the document from the server before, use the removeById() method. This requires just the collection name (here: "users") and the document id.

```
// remove a document on the server, using a collection id and document id
// In this example, we are using the id of the document we deleted in the previous example,
// so it will throw an exception here. (we are catching it though, in order to continue)

try {
    $result = $handler->removeById('users', $userFromServer->getId());
} catch (\ArangoDBClient\ServerException $e) {
    $e->getMessage();
}
```

Running an AQL query

To run an AQL query, use the *Statement* class.

The method Statement::execute creates a Cursor object which can be used to iterate over the query's result set.

```
// create a statement to insert 1000 test users
$statement = new ArangoStatement(
    $connection, [
                   'query' => 'FOR i IN 1..1000 INSERT { _key: CONCAT("test", i) } IN users'
               ]
);

// execute the statement
$cursor = $statement->execute();


// now run another query on the data, using bind parameters
$statement = new ArangoStatement(
    $connection, [
                   'query' => 'FOR u IN @@collection FILTER u.name == @name RETURN u',
                   'bindVars' => [
                       '@collection' => 'users',
                       'name' => 'John'
                   ]
               ]
);

// executing the statement returns a cursor
$cursor = $statement->execute();

// easiest way to get all results returned by the cursor
var_dump($cursor->getAll());
```

```
// to get statistics for the query, use Cursor::getExtra();
var_dump($cursor->getExtra());
```

Note: by default the Statement object will create a Cursor that converts each value into a Document object. This is normally the intended behavior for AQL queries that return entire documents. However, an AQL query can also return projections or any other data that cannot be converted into Document objects.

In order to suppress the conversion into Document objects, the Statement must be given the `_flat` attribute. This allows processing the results of arbitrary AQL queries:

```
// run an AQL query that does not return documents but scalars
// we need to set the _flat attribute of the Statement in order for this to work
$statement = new ArangoStatement(
    $connection, [
                   'query' => 'FOR i IN 1..1000 RETURN i',
                   '_flat' => true
                 ]
);

// executing the statement returns a cursor
$cursor = $statement->execute();

// easiest way to get all results returned by the cursor
// note that now the results won't be converted into Document objects
var_dump($cursor->getAll());
```

Exporting data

To export the contents of a collection to PHP, use the *Export* class. The *Export* class will create a light-weight cursor over all documents of the specified collection. The results can be transferred to PHP in chunks incrementally. This is the most efficient way of iterating over all documents in a collection.

```
// creates an export object for collection users
$export = new ArangoExport($connection, 'users', []);

// execute the export. this will return a special, forward-only cursor
$cursor = $export->execute();

// now we can fetch the documents from the collection in blocks
while ($docs = $cursor->getNextBatch()) {
    // do something with $docs
    var_dump($docs);
}

// the export can also be restricted to just a few attributes per document:
$export = new ArangoExport(
    $connection, 'users', [
                   '_flat' => true,
                   'restrict' => [
                       'type' => 'include',
                       'fields' => ['_key', 'likes']
                   ]
               ]
);

// now fetch just the configured attributes for each document
while ($docs = $cursor->getNextBatch()) {
    // do something with $docs
    var_dump($docs);
}
```

Bulk document handling

The ArangoDB-PHP driver provides a mechanism to easily fetch multiple documents from the same collection with a single request. All that needs to be provided is an array of document keys:

```
$exampleCollection = new ArangoCollection();
$exampleCollection->setName('example');
$id = $collectionHandler->create($exampleCollection);

// create a statement to insert 100 example documents
$statement = new ArangoStatement(
    $connection, [
                    'query' => 'FOR i IN 1..100 INSERT { _key: CONCAT("example", i), value: i } IN example'
                ]
);
$statement->execute();

// later on, we can assemble a list of document keys
$keys = [];
for ($i = 1; $i <= 100; ++$i) {
    $keys[] = 'example' . $i;
}
// and fetch all the documents at once
$documents = $collectionHandler->lookupByKeys('example', $keys);
var_dump($documents);

// we can also bulk-remove them:
$result = $collectionHandler->removeByKeys('example', $keys);

var_dump($result);
```

Dropping a collection

To drop an existing collection on the server, use the drop() method of the *CollectionHandler* class. drop() just needs the name of the collection name to be dropped:

```
// drop a collection on the server, using its name,
$result = $collectionHandler->drop('users');
var_dump($result);

// drop the other one we created, too
$collectionHandler->drop('example');
```

Custom Document class

If you want to use custom document class you can pass it's name to DocumentHandler or CollectionHandler using method `setDocumentClass`. Remember that Your class must extend `\ArangoDBClient\Document`.

```
$ch = new CollectionHandler($connection);
$ch->setDocumentClass('\AppBundle\Entity\Product');
$cursor = $ch->all('product');
// All returned documents will be \AppBundle\Entity\Product instances


$dh = new DocumentHandler($connection);
$dh->setDocumentClass('\AppBundle\Entity\Product');
$product = $dh->get('products', 11231234);
// Product will be \AppBundle\Entity\Product instance
```

See file examples/customDocumentClass.php for more details.

Logging exceptions

The driver provides a simple logging mechanism that is turned off by default. If it is turned on, the driver will log all its exceptions using PHP's standard `error_log` mechanism. It will call PHP's `error_log()` function for this. It depends on the PHP configuration if and where exceptions will be logged. Please consult your php.ini settings for further details.

To turn on exception logging in the driver, set a flag on the driver's Exception base class, from which all driver exceptions are subclassed:

```
use ArangoDBClient\Exception as ArangoException;

ArangoException::enableLogging();
```

To turn logging off, call its `disableLogging` method:

```
use ArangoDBClient\Exception as ArangoException;

ArangoException::disableLogging();
```

Putting it all together

Here's the full code that combines all the pieces outlined above:

```
// use the following line when using Composer
// require __DIR__ . '/vendor/composer/autoload.php';

// use the following line when using git
require __DIR__ . '/autoload.php';

// set up some aliases for less typing later
use ArangoDBClient\Collection as ArangoCollection;
use ArangoDBClient\CollectionHandler as ArangoCollectionHandler;
use ArangoDBClient\Connection as ArangoConnection;
use ArangoDBClient\ConnectionOptions as ArangoConnectionOptions;
use ArangoDBClient\DocumentHandler as ArangoDocumentHandler;
use ArangoDBClient\Document as ArangoDocument;
use ArangoDBClient\Exception as ArangoException;
use ArangoDBClient\Export as ArangoExport;
use ArangoDBClient\ConnectException as ArangoConnectException;
use ArangoDBClient\ClientException as ArangoClientException;
use ArangoDBClient\ServerException as ArangoServerException;
use ArangoDBClient\Statement as ArangoStatement;
use ArangoDBClient\UpdatePolicy as ArangoUpdatePolicy;

// set up some basic connection options
$connectionOptions = [
    // database name
    ArangoConnectionOptions::OPTION_DATABASE => '_system',
    // server endpoint to connect to
    ArangoConnectionOptions::OPTION_ENDPOINT => 'tcp://127.0.0.1:8529',
    // authorization type to use (currently supported: 'Basic')
    ArangoConnectionOptions::OPTION_AUTH_TYPE => 'Basic',
    // user for basic authorization
    ArangoConnectionOptions::OPTION_AUTH_USER => 'root',
    // password for basic authorization
    ArangoConnectionOptions::OPTION_AUTH_PASSWD => '',
    // connection persistence on server. can use either 'Close' (one-time connections) or 'Keep-Alive' (re-used connections)
    ArangoConnectionOptions::OPTION_CONNECTION => 'Keep-Alive',
    // connect timeout in seconds
    ArangoConnectionOptions::OPTION_TIMEOUT => 3,
    // whether or not to reconnect when a keep-alive connection has timed out on server
    ArangoConnectionOptions::OPTION_RECONNECT => true,
    // optionally create new collections when inserting documents
    ArangoConnectionOptions::OPTION_CREATE => true,
    // optionally create new collections when inserting documents
    ArangoConnectionOptions::OPTION_UPDATE_POLICY => ArangoUpdatePolicy::LAST,
];


// turn on exception logging (logs to whatever PHP is configured)
ArangoException::enableLogging();

try {
    $connection = new ArangoConnection($connectionOptions);

    $collectionHandler = new ArangoCollectionHandler($connection);

    // clean up first
    if ($collectionHandler->has('users')) {
```

```
    $collectionHandler->drop('users');
}
if ($collectionHandler->has('example')) {
    $collectionHandler->drop('example');
}

// create a new collection
$userCollection = new ArangoCollection();
$userCollection->setName('users');
$id = $collectionHandler->create($userCollection);

// print the collection id created by the server
var_dump($id);

// check if the collection exists
$result = $collectionHandler->has('users');
var_dump($result);

$handler = new ArangoDocumentHandler($connection);

// create a new document
$user = new ArangoDocument();

// use set method to set document properties
$user->set('name', 'John');
$user->set('age', 25);
$user->set('thisIsNull', null);

// use magic methods to set document properties
$user->likes = ['fishing', 'hiking', 'swimming'];

// send the document to the server
$id = $handler->save('users', $user);

// check if a document exists
$result = $handler->has('users', $id);
var_dump($result);

// print the document id created by the server
var_dump($id);
var_dump($user->getId());


// get the document back from the server
$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);

// get a document list back from the server, using a document example
$cursor = $collectionHandler->byExample('users', ['name' => 'John']);
var_dump($cursor->getAll());


// update a document
$userFromServer->likes = ['fishing', 'swimming'];
$userFromServer->state = 'CA';

$result = $handler->update($userFromServer);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);


// update a document removing an attribute,
// The 'keepNull'=>false option will cause ArangoDB to
// remove all attributes in the document,
// that have null as their value - not only the ones defined here

$userFromServer->likes = ['fishing', 'swimming'];
$userFromServer->state = 'CA';
$userFromServer->age   = null;

$result = $handler->update($userFromServer, ['keepNull' => false]);
var_dump($result);

$userFromServer = $handler->get('users', $id);
```

```
var_dump($userFromServer);


// replace a document (notice that we are using the previously fetched document)
// In this example we are removing the state attribute
unset($userFromServer->state);

$result = $handler->replace($userFromServer);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);


// replace a document, identified by collection and document id
$user                  = new ArangoDocument();
$user->name            = 'John';
$user->likes           = ['Running', 'Rowing'];
$userFromServer->state = 'CA';

// Notice that for the example we're getting the existing
// document id via a method call. Normally we would use the known id
$result = $handler->replaceById('users', $userFromServer->getId(), $user);
var_dump($result);

$userFromServer = $handler->get('users', $id);
var_dump($userFromServer);


// remove a document on the server
$result = $handler->remove($userFromServer);
var_dump($result);


// remove a document on the server, using a collection id and document id
// In this example, we are using the id of the document we deleted in the previous example,
// so it will throw an exception here. (we are catching it though, in order to continue)

try {
    $result = $handler->removeById('users', $userFromServer->getId());
} catch (\ArangoDBClient\ServerException $e) {
    $e->getMessage();
}



// create a statement to insert 1000 test users
$statement = new ArangoStatement(
    $connection, [
                   'query' => 'FOR i IN 1..1000 INSERT { _key: CONCAT("test", i) } IN users'
               ]
);

// execute the statement
$cursor = $statement->execute();


// now run another query on the data, using bind parameters
$statement = new ArangoStatement(
    $connection, [
                   'query' => 'FOR u IN @@collection FILTER u.name == @name RETURN u',
                   'bindVars' => [
                       '@collection' => 'users',
                       'name' => 'John'
                   ]
               ]
);

// executing the statement returns a cursor
$cursor = $statement->execute();

// easiest way to get all results returned by the cursor
var_dump($cursor->getAll());

// to get statistics for the query, use Cursor::getExtra();
var_dump($cursor->getExtra());
```

```
    // creates an export object for collection users
    $export = new ArangoExport($connection, 'users', []);

    // execute the export. this will return a special, forward-only cursor
    $cursor = $export->execute();

    // now we can fetch the documents from the collection in blocks
    while ($docs = $cursor->getNextBatch()) {
        // do something with $docs
        var_dump($docs);
    }

    // the export can also be restricted to just a few attributes per document:
    $export = new ArangoExport(
        $connection, 'users', [
                        '_flat' => true,
                        'restrict' => [
                            'type' => 'include',
                            'fields' => ['_key', 'likes']
                        ]
                    ]
    );

    // now fetch just the configured attributes for each document
    while ($docs = $cursor->getNextBatch()) {
        // do something with $docs
        var_dump($docs);
    }


    $exampleCollection = new ArangoCollection();
    $exampleCollection->setName('example');
    $id = $collectionHandler->create($exampleCollection);

    // create a statement to insert 100 example documents
    $statement = new ArangoStatement(
        $connection, [
                        'query' => 'FOR i IN 1..100 INSERT { _key: CONCAT("example", i), value: i } IN example'
                    ]
    );
    $statement->execute();

    // later on, we can assemble a list of document keys
    $keys = [];
    for ($i = 1; $i <= 100; ++$i) {
        $keys[] = 'example' . $i;
    }
    // and fetch all the documents at once
    $documents = $collectionHandler->lookupByKeys('example', $keys);
    var_dump($documents);

    // we can also bulk-remove them:
    $result = $collectionHandler->removeByKeys('example', $keys);

    var_dump($result);


    // drop a collection on the server, using its name,
    $result = $collectionHandler->drop('users');
    var_dump($result);

    // drop the other one we created, too
    $collectionHandler->drop('example');
} catch (ArangoConnectException $e) {
    print 'Connection error: ' . $e->getMessage() . PHP_EOL;
} catch (ArangoClientException $e) {
    print 'Client error: ' . $e->getMessage() . PHP_EOL;
} catch (ArangoServerException $e) {
    print 'Server error: ' . $e->getServerCode() . ': ' . $e->getServerMessage() . ' - ' . $e->getMessage() . PHP_EOL;
}
```

ArangoDB GO Driver

The official ArangoDB GO Driver

ArangoDB GO Driver - Getting Started

Supported versions

- ArangoDB versions 3.1 and up.
  - Single server & cluster setups
  - With or without authentication
- Go 1.7 and up.

Go dependencies

- None (Additional error libraries are supported).

Configuration

To use the driver, first fetch the sources into your GOPATH.

```
go get github.com/arangodb/go-driver
```

Using the driver, you always need to create a `Client`. The following example shows how to create a `Client` for a single server running on localhost.

```
import (
    "fmt"

    driver "github.com/arangodb/go-driver"
    "github.com/arangodb/go-driver/http"
)

...

conn, err := http.NewConnection(http.ConnectionConfig{
    Endpoints: []string{"http://localhost:8529"},
})
if err != nil {
    // Handle error
}
c, err := driver.NewClient(driver.ClientConfig{
    Connection: conn,
})
if err != nil {
    // Handle error
}
```

Once you have a `Client` you can access/create databases on the server, access/create collections, graphs, documents and so on.

The following example shows how to open an existing collection in an existing database and create a new document in that collection.

```
// Open "examples_books" database
db, err := c.Database(nil, "examples_books")
if err != nil {
    // Handle error
}

// Open "books" collection
col, err := db.Collection(nil, "books")
if err != nil {
    // Handle error
}

// Create document
```

```
book := Book{
    Title:   "ArangoDB Cookbook",
    NoPages: 257,
}
meta, err := col.CreateDocument(nil, book)
if err != nil {
    // Handle error
}
fmt.Printf("Created document in collection '%s' in database '%s'\n", col.Name(), db.Name())
```

API design

Concurrency

All functions of the driver are stricly synchronous. They operate and only return a value (or error) when they're done.

If you want to run operations concurrently, use a go routine. All objects in the driver are designed to be used from multiple concurrent go routines, except `Cursor`.

All database objects (except `Cursor`) are considered static. After their creation they won't change. E.g. after creating a `Collection` instance you can remove the collection, but the (Go) instance will still be there. Calling functions on such a removed collection will of course fail.

Structured error handling & wrapping

All functions of the driver that can fail return an `error` value. If that value is not `nil`, the function call is considered to be failed. In that case all other return values are set to their `zero` values.

All errors are structured using error checking functions named `Is<SomeErrorCategory>`. E.g. `IsNotFound(error)` return true if the given error is of the category "not found". There can be multiple internal error codes that all map onto the same category.

All errors returned from any function of the driver (either internal or exposed) wrap errors using the `WithStack` function. This can be used to provide detail stack trackes in case of an error. All error checking functions use the `Cause` function to get the cause of an error instead of the error wrapper.

Note that `WithStack` and `Cause` are actually variables to you can implement it using your own error wrapper library.

If you for example use https://github.com/pkg/errors, you want to initialize to go driver like this:

```
import (
    driver "github.com/arangodb/go-driver"
    "github.com/pkg/errors"
)

func init() {
    driver.WithStack = errors.WithStack
    driver.Cause = errors.Cause
}
```

Context aware

All functions of the driver that involve some kind of long running operation or support additional options not given as function arguments, have a `context.Context` argument. This enables you cancel running requests, pass timeouts/deadlines and pass additional options.

In all methods that take a `context.Context` argument you can pass `nil` as value. This is equivalent to passing `context.Background()`.

Many functions support 1 or more optional (and infrequently used) additional options. These can be used with a `With<OptionName>` function. E.g. to force a create document call to wait until the data is synchronized to disk, use a prepared context like this:

```
ctx := driver.WithWaitForSync(parentContext)
collection.CreateDocument(ctx, yourDocument)
```

ArangoDB GO Driver - Example requests

Connecting to ArangoDB

```
conn, err := http.NewConnection(http.ConnectionConfig{
    Endpoints: []string{"http://localhost:8529"},
    TLSConfig: &tls.Config{ /*...*/ },
})
if err != nil {
    // Handle error
}
c, err := driver.NewClient(driver.ClientConfig{
    Connection: conn,
    Authentication: driver.BasicAuthentication("user", "password"),
})
if err != nil {
    // Handle error
}
```

Opening a database

```
ctx := context.Background()
db, err := client.Database(ctx, "myDB")
if err != nil {
    // handle error
}
```

Opening a collection

```
ctx := context.Background()
col, err := db.Collection(ctx, "myCollection")
if err != nil {
    // handle error
}
```

Checking if a collection exists

```
ctx := context.Background()
found, err := db.CollectionExists(ctx, "myCollection")
if err != nil {
    // handle error
}
```

Creating a collection

```
ctx := context.Background()
options := &driver.CreateCollectionOptions{ /* ... */ }
col, err := db.CreateCollection(ctx, "myCollection", options)
if err != nil {
    // handle error
}
```

Reading a document from a collection

```
var doc MyDocument
ctx := context.Background()
meta, err := col.ReadDocument(ctx, myDocumentKey, &doc)
if err != nil {
    // handle error
}
```

Reading a document from a collection with an explicit revision

```
var doc MyDocument
revCtx := driver.WithRevision(ctx, "mySpecificRevision")
meta, err := col.ReadDocument(revCtx, myDocumentKey, &doc)
if err != nil {
    // handle error
}
```

Creating a document

```
doc := MyDocument{
    Name: "jan",
    Counter: 23,
}
ctx := context.Background()
meta, err := col.CreateDocument(ctx, doc)
if err != nil {
    // handle error
}
fmt.Printf("Created document with key '%s', revision '%s'\n", meta.Key, meta.Rev)
```

Removing a document

```
ctx := context.Background()
err := col.RemoveDocument(revCtx, myDocumentKey)
if err != nil {
    // handle error
}
```

Removing a document with an explicit revision

```
revCtx := driver.WithRevision(ctx, "mySpecificRevision")
err := col.RemoveDocument(revCtx, myDocumentKey)
if err != nil {
    // handle error
}
```

Updating a document

```
ctx := context.Background()
patch := map[string]interface{}{
    "Name": "Frank",
}
meta, err := col.UpdateDocument(ctx, myDocumentKey, patch)
if err != nil {
    // handle error
}
```

Querying documents, one document at a time

```
ctx := context.Background()
query := "FOR d IN myCollection LIMIT 10 RETURN d"
cursor, err := db.Query(ctx, query, nil)
if err != nil {
    // handle error
}
defer cursor.Close()
for {
    var doc MyDocument
    meta, err := cursor.ReadDocument(ctx, &doc)
    if driver.IsNoMoreDocuments(err) {
        break
    } else if err != nil {
        // handle other errors
    }
    fmt.Printf("Got doc with key '%s' from query\n", meta.Key)
}
```

Querying documents, fetching total count

```
ctx := driver.WithQueryCount(context.Background())
query := "FOR d IN myCollection RETURN d"
cursor, err := db.Query(ctx, query, nil)
if err != nil {
    // handle error
}
defer cursor.Close()
fmt.Printf("Query yields %d documents\n", cursor.Count())
```

Querying documents, with bind variables

```
ctx := context.Background()
query := "FOR d IN myCollection FILTER d.Name == @name RETURN d"
bindVars := map[string]interface{}{
    "name": "Some name",
}
cursor, err := db.Query(ctx, query, bindVars)
if err != nil {
    // handle error
}
defer cursor.Close()
...
```

ArangoDB GO Driver - Connection Management

Failover

The driver supports multiple endpoints to connect to. All request are in principle send to the same endpoint until that endpoint fails to respond. In that case a new endpoint is chosen and the operation is retried.

The following example shows how to connect to a cluster of 3 servers.

```
conn, err := http.NewConnection(http.ConnectionConfig{
    Endpoints: []string{"http://server1:8529", "http://server2:8529", "http://server3:8529"},
})
if err != nil {
    // Handle error
}
c, err := driver.NewClient(driver.ClientConfig{
    Connection: conn,
})
if err != nil {
    // Handle error
}
```

Note that a valid endpoint is an URL to either a standalone server, or a URL to a coordinator in a cluster.

Failover: Exact behavior

The driver monitors the request being send to a specific server (endpoint). As soon as the request has been completely written, failover will no longer happen. The reason for that is that several operations cannot be (safely) retried. E.g. when a request to create a document has been send to a server and a timeout occurs, the driver has no way of knowing if the server did or did not create the document in the database.

If the driver detects that a request has been completely written, but still gets an error (other than an error response from Arango itself), it will wrap the error in a `ResponseError`. The client can test for such an error using `IsResponseError`.

If a client received a `ResponseError`, it can do one of the following:

- Retry the operation and be prepared for some kind of duplicate record / unique constraint violation.
- Perform a test operation to see if the "failed" operation did succeed after all.
- Simply consider the operation failed. This is risky, since it can still be the case that the operation did succeed.

Failover: Timeouts

To control the timeout of any function in the driver, you must pass it a context configured with `context.WithTimeout` (or `context.WithDeadline`).

In the case of multiple endpoints, the actual timeout used for requests will be shorter than the timeout given in the context. The driver will divide the timeout by the number of endpoints with a maximum of 3. This ensures that the driver can try up to 3 different endpoints (in case of failover) without being canceled due to the timeout given by the client. E.g.

- With 1 endpoint and a given timeout of 1 minute, the actual request timeout will be 1 minute.
- With 3 endpoints and a given timeout of 1 minute, the actual request timeout will be 20 seconds.
- With 8 endpoints and a given timeout of 1 minute, the actual request timeout will be 20 seconds.

For most requests you want a actual request timeout of at least 30 seconds.

Secure connections (SSL)

The driver supports endpoints that use SSL using the `https` URL scheme.

The following example shows how to connect to a server that has a secure endpoint using a self-signed certificate.

```
conn, err := http.NewConnection(http.ConnectionConfig{
    Endpoints: []string{"https://localhost:8529"},
    TLSConfig: &tls.Config{InsecureSkipVerify: true},
})
if err != nil {
    // Handle error
}
c, err := driver.NewClient(driver.ClientConfig{
    Connection: conn,
})
if err != nil {
    // Handle error
}
```