ArangoDB Documentation

ArangoDB Documentation

Welcome to the ArangoDB documentation!

The documentation introduces ArangoDB for you as an user, developer and administrator and describes all of his functions in detail.

ArangoDB is a multi-purpose open-source database with a flexible data model for documents, graphs and key-values. You can easily build high performance applications using a convenient [SQL-like query language](#) or [JavaScript](#) and mruby extensions.

The database server [*arangod*](#) stores all documents and serves them using a REST interface. There are [drivers](#) for all major languages like Ruby, Python, PHP, JavaScript, and Perl. In the following sections we will use the JavaScript shell to communicate with the database and demonstrate some of ArangoDB's features using JavaScript.

Some of the features and programs of ArangoDB are:

- A powerful query language
- Open Source
- A database daemon
- An ArangoDB shell
- Flexible data modeling
- and many more!

In this documentation you can inform yourself about all the functions, features and programs ArangoDB provides for you.

If you want to test the shell go [here](#).

If you want to play with our query language, go to our [AQL Tutorial](#).

Community

If you have any questions don't hesitate to ask on:

- [github](#)
- [google groups](#)
- [stackoverflow](#)

We will respond as soon as possible.

Installing

This chapter describes how to install ArangoDB under various operation systems.

First of all download and install the corresponding RPM or Debian package or use homebrew on the MacOS X. You can find packages for various operation systems at our [download](#) section.

If you don't want to install ArangoDB at the beginning and just want to experiment with the features, you can use our [online demo](#).

In this Chapter you will also learn how to Compile ArangoDB from scratch.

You also get help if you want to update your ArangoDB Version to the newest one!

Linux

You can find binary packages for various Linux distributions [here](#).

We provide packages for:

- Centos
- Debian
- Fedora
- [Linux-Mint](#)
- Mandriva
- OpenSUSE
- RedHat RHEL
- SUSE SLE
- Ubuntu

Using a Package Manager to install ArangoDB

Follow the instructions on the [downloads](#) page to use your favorite package manager for the major distributions. After setting up the ArangoDB repository you can easily install ArangoDB using yum, aptitude, urpmi or zypper.

Gentoo

Please use the [portage](#) provided by @mgiken.

Debian sid

To use ArangoDB on Debian sid (the development version of Debian), a different version of ICU is required. User basir provided the following instructions for getting ArangoDB 2.0.7 to work on an x86_64:

[link to Github issue](#)

Other versions of ArangoDB or other architectures should work similarly.

Linux-Mint

Download and import GPG-PublicKey:

```
wget -O RPM-GPG-KEY-www.arangodb.org
http://www.arangodb.org/repositories/PublicKey
apt-key add RPM-GPG-KEY-www.arangodb.org
```

Add the corresponding repository in file `/etc/apt/sources.list`:

```
deb http://www.arangodb.org/repositories LinuxMint-13 main
```

Update the repository data:

```
aptitude update
```

Now you should be able to search for arangodb:

```
aptitude search arangodb
```

In order to install arangodb:

```
aptitude install arangodb
```

Using Vagrant and Chef

A Chef recipe is available from jbianquetti at:

```
https://github.com/jbianquetti/chef-arangodb
```

Mac OS X

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

There is also a version available in the [AppStore](), which comes with a nice graphical user interface to start and stop the server.

Homebrew

If you are using [homebrew,](#) then you can install the ArangoDB using *brew* as follows:

```
brew install arangodb
```

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

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

The ArangoDB shell will be installed as:

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

If you want to install the latest (unstable) version use:

```
brew install --HEAD arangodb
```

You can uninstall ArangoDB using:

```
brew uninstall arangodb
```

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

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

Then remove the LaunchAgent:

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

Apple's App Store

ArangoDB is available in Apple's App-Store. Please note, that it sometimes takes days or weeks until the latest versions are available.

Command-Line App

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

Choose *Mac OS X* and go to *Grab binary packages directly*. This allows you to install the application *ArangoDB-CLI* in your application folder.

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

```
ArangoDB server has been started

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

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

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

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


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

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

Windows

The default installation directory is *c:\Program Files\ArangoDB-1.x.y*. During the installation process you may change this. In the following description we will assume that ArangoDB has been installed in the location .

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

Installing for a single user: Select a different directory during installation. For example *C:/Users//arangodb* or *C:/ArangoDB*.

Installing for multiple users: Keep the default directory. After the installation edit the file */etc/arangodb/arangod.conf*. Adjust the *directory* and *app-path* so that these paths point into your home directory.

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

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

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

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

```
    cmd> arangod --install-service
    INFO: adding service 'ArangoDB - the multi-purpose database'
(internal 'ArangoDB')
    INFO: added service with command line '"C:\Program Files
(x86)\ArangoDB 1.4.4\bin\arangod.exe" --start-service'

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

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

Client, Server and Lock-Files

Please note that ArangoDB consists of a database server and client tools. If you start the server, it will place a (read-only) lock file to prevent accidental access to the data. The server will attempt to remove this lock file when it is started to see if the lock is still valid - this is in case the installation did not proceed correctly or if the server terminated unexpectedly.

Starting

To start an ArangoDB server instance with networking enabled, use the executable *arangod.exe* located in */bin*. This will use the configuration file *arangod.conf* located in */etc/arangodb*, which you can adjust to your needs and use the data directory */var/lib/arangodb*. This is the place where all your data (databases and collections) will be stored by default.

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

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

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

To check if your installation was successful, click the *Collection* tab and open the configuration. Select the *System* type. If the installation was successful, then the page should display a few system collections.

Try to add a new collection and then add some documents to this new collection. If you have succeeded in creating a new collection and inserting one or more documents, then your installation is working correctly.

Advanced Starting

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

Using the Client

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

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

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

32bit

If you have an EXISTING database, then please note that currently a 32 bit version of ArangoDB is NOT compatible with a 64 bit version. This means that if you have a database created with a 32 bit version of ArangoDB it may become corrupted if you execute a 64 bit version of ArangoDB against the same database, and vice versa.

Upgrading

To upgrade an EXISTING database created with a previous version of ArangoDB, please execute the server *arangod.exe* with the option *--upgrade*. Otherwise starting ArangoDB may fail with errors.

Note that there is no harm in running the upgrade. So you should run this batch file if you are unsure of the database version you are using.

You should always check the output for errors to see if the upgrade was completed successfully.

Uninstalling

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

Limitations for Cygwin

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

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

Compiling ArangoDB from scratch

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

There are the following possibilities:

- all-in-one: this version contains the source code of the ArangoDB, all generated files from the autotools, FLEX, and BISON as well as a version of V8, libev, and ICU.

- devel: this version contains the development version of the ArangoDB. Use this branch, if you want to make changes to ArangoDB source.

The devel version requires a complete development environment, while the all-in-one version allows you to compile the ArangoDB without installing all the prerequisites. The disadvantage is that it takes longer to compile and you cannot make changes to the flex or bison files.

Amazon Micro Instance

@sohgoh has reported that it is very easy to install ArangoDB on an Amazon Micro Instance:

```
amazon> sudo yum install readline-devel
amazon> ./configure
amazon> make
amazon> make install
```

For detailed instructions the following section.

All-In-One Version

Note: there are separate instructions for the **devel** version in the next section.

Basic System Requirements

Verify that your system contains:

- the GNU C/C++ compilers "gcc" and "g++" and the standard C/C++ libraries. You will compiler and library support for C++11. To be on the safe side with gcc/g++, you will need version number 4.8 or higher. For "clang" and "clang++", you will need at least version 3.4.
- the GNU make

In addition you will need the following library:

- the GNU readline library
- the OpenSSL library
- Go 1.2 (or higher)

Under Mac OS X you also need to install:

- Xcode
- scons

Download the Source

Download the latest source using GIT:

```
git clone git://github.com/triAGENS/ArangoDB.git
```

Note: if you only plan to compile ArangoDB locally and do not want to modify or push any changes, you can speed up cloning substantially by using the *--single-branch* and *--depth* parameters for the clone command as follws:

```
git clone --single-branch --depth 1
git://github.com/triAGENS/ArangoDB.git
```

Configure

Switch into the ArangoDB directory

```
cd ArangoDB
```

In order to configure the build environment execute

```
./configure --enable-all-in-one-v8 --enable-all-in-one-libev --
enable-all-in-one-icu
```

to setup the makefiles. This will check the various system characteristics and installed libraries.

Compile

Compile the program by executing

```
make
```

This will compile the ArangoDB and create a binary of the server in

```
./bin/arangod
```

Test

Create an empty directory

```
unix> mkdir /tmp/database-dir
```

Check the binary by starting it using the command line.

```
unix> ./bin/arangod -c etc/relative/arangod.conf --server.endpoint
tcp://127.0.0.1:12345 --server.disable-authentication true
/tmp/database-dir
```

This will start up the ArangoDB and listen for HTTP requests on port 12345 bound to IP address 127.0.0.1. You should see the startup messages similar to the following:

```
2013-10-14T12:47:29Z [29266] INFO ArangoDB xxx ... </br>
2013-10-14T12:47:29Z [29266] INFO using endpoint
'tcp://127.0.0.1.12345' for non-encrypted requests </br>
2013-10-14T12:47:30Z [29266] INFO Authentication is turned off
</br>
2013-10-14T12:47:30Z [29266] INFO ArangoDB (version xxx) is ready
for business. Have fun! </br>
```

If it fails with a message about the database directory, please make sure the database directory you specified exists and can be written into.

Use your favorite browser to access the URL

```
http://127.0.0.1:12345/_api/version
```

This should produce a JSON object like

```
{"server" : "arango", "version" : "..."}
```

as result.

Install

Install everything by executing

```
make install
```

You must be root to do this or at least have write permission to the corresponding directories.

The server will by default be installed in

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

The configuration file will be installed in

```
/usr/local/etc/arangodb/arangod.conf
```

The database will be installed in

```
/usr/local/var/lib/arangodb
```

The ArangoShell will be installed in

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

When upgrading from a previous version of ArangoDB, please make sure you inspect ArangoDB's log file after an upgrade. It may also be necessary to start ArangoDB with the *--upgrade* parameter once to perform required upgrade or initialisation tasks.

Devel Version

Basic System Requirements

Verify that your system contains

- the GNU C/C++ compilers "gcc" and "g++" and the standard C/C++ libraries. You will
- compiler and library support for C++11. To be on the safe side with gcc/g++, you will
- need version number 4.8 or higher. For "clang" and "clang++", you will need at least
- version 3.4.
- the GNU autotools (autoconf, automake)
- the GNU make
- the GNU scanner generator FLEX, at least version 2.3.35
- the GNU parser generator BISON, at least version 2.4
- Python, version 2 or 3
- Go, version 1.2 or higher

In addition you will need the following libraries

- libev in version 3 or 4 (only when configured with `--disable-all-in-one-libev`)
- Google's V8 engine (only when configured with `--disable-all-in-one-v8`)
- the ICU library (only when not configured with `--enable-all-in-one-icu`)
- the GNU readline library
- the OpenSSL library
- the Boost test framework library (boost_unit_test_framework)

To compile Google V8 yourself, you will also need Python 2 and SCons.

Some distributions, for example Centos 5, provide only very out-dated versions of compilers, FLEX, BISON, and the V8 engine. In that case you need to compile newer versions of the programs and/or libraries.

If necessary, install or download the prerequisites:

- GNU C/C++ 4.8 or higher (see http://gcc.gnu.org)
- Google's V8 engine (see http://code.google.com/p/v8)
- SCons for compiling V8 (see http://www.scons.org)
- libev (see http://software.schmorp.de/pkg/libev.html)
- OpenSSL (http://openssl.org/)
- Go (http://golang.org/)

Most linux systems already supply RPM or DEP for these packages. Please note that you have to install the development packages.

Download the Source

Download the latest source using GIT:

```
git clone git://github.com/triAGENS/ArangoDB.git
```

Setup

Switch into the ArangoDB directory

```
cd ArangoDB
```

The source tarball contains a pre-generated "configure" script. You can regenerate this script by using the GNU auto tools. In order to do so, execute

```
make setup
```

This will call aclocal, autoheader, automake, and autoconf in the correct order.

Configure

In order to configure the build environment please execute

```
unix> ./configure --enable-all-in-one-v8 --enable-all-in-one-libev
--enable-all-in-one-icu
```

to setup the makefiles. This will check for the various system characteristics and installed libraries.

Please note that it may be required to set the *--host* and *--target* variables when running the configure command. For example, if you compile on MacOS, you should add the following options to the configure command:

```
--host=x86_64-apple-darwin --target=x86_64-apple-darwin
```

The host and target values for other architectures vary.

If you also plan to make changes to the source code of ArangoDB, add the following option to the *configure* command: *--enable-maintainer-mode*. Using this option, you can make changes to the lexer and parser files and some other source files that will generate other files. Enabling this option will add extra dependencies to BISON, FLEX, and PYTHON. These external tools then need to be available in the correct versions on your system.

The following configuration options exist:

```
--enable-relative
```

This will make relative paths be used in the compiled binaries and scripts. It allows to run ArangoDB from the compile directory directly, without the need for a *make install* command and specifying much configuration parameters. When used, you can start ArangoDB using this command:

```
bin/arangod /tmp/database-dir
```

ArangoDB will then automatically use the configuration from file *etc/relative/arangod.conf*.

```
--enable-all-in-one-libev
```

This tells the build system to use the bundled version of libev instead of using the system version.

```
--disable-all-in-one-libev
```

This tells the build system to use the installed system version of libev instead of compiling the supplied version from the 3rdParty directory in the make run.

```
--enable-all-in-one-v8
```

This tells the build system to use the bundled version of V8 instead of using the system version.

```
--disable-all-in-one-v8
```

This tells the build system to use the installed system version of V8 instead of compiling the supplied version from the 3rdParty directory in the make run.

```
--enable-all-in-one-icu
```

This tells the build system to use the bundled version of ICU instead of using the system version.

```
--disable-all-in-one-icu
```

This tells the build system to use the installed system version of ICU instead of compiling the supplied version from the 3rdParty directory in the make run.

```
--enable-all-in-one-boost
```

This tells the build system to use the bundled version of Boost header files. This is the default and recommended.

```
--enable-all-in-one-etcd
```

This tells the build system to use the bundled version of ETCD. This is the default and recommended.

```
--enable-internal-go
```

This tells the build system to use Go binaries located in the 3rdParty directory. Note that ArangoDB does not ship with Go binaries, and that the Go binaries must be copied into this directory manually.

```
--enable-maintainer-mode
```

This tells the build system to use BISON and FLEX to regenerate the parser and scanner files. If disabled, the supplied files will be used so you cannot make changes to the parser and scanner files. You need at least BISON 2.4.1 and FLEX 2.5.35. This option also allows you to make changes to the error messages file, which is converted to js and C header files using Python. You will need Python 2 or 3 for this. Furthermore, this option enables additional test cases to be executed in a *make unittests* run. You also need to install the Boost test framework for this.

Compiling Go

Users F21 and duralog told us that some systems don't provide an update-to-date version of go. This seems to be the case for at least Ubuntu 12 and 13. To install go on these system, you may follow the instructions provided [here](). For other systems, you may follow the instructions [here]().

To make ArangoDB use a specific version of go, you may copy the go binaries into the 3rdParty/go-32 or 3rdParty/go-64 directories of ArangoDB (depending on your architecture), and then tell ArangoDB to use this specific go version by using the *--enable-internal-go* configure option.

User duralog provided some the following script to pull the latest release version of go into the ArangoDB source directory and build it:

```
cd ArangoDB
hg clone -u release https://code.google.com/p/go 3rdParty/go-64 && \
  cd 3rdParty/go-64/src && \
  ./all.bash

# now that go is installed, run your configure with --enable-internal-go
./configure\
  --enable-all-in-one-v8 \
```

```
--enable-all-in-one-libev \
--enable-internal-go
```

General Upgrade Information

Recommended upgrade procedure

To upgrade an existing ArangoDB database to a newer version of ArangoDB (e.g. 1.2 to 1.3, or 2.0 to 2.1), the following method is recommended:

- Check the *CHANGELOG* for API or other changes in the new version of ArangoDB and make sure your applications can deal with them
- Stop the "old" arangod service or binary
- Copy the entire "old" data directory to a safe place (that is, a backup)
- Install the new version of ArangoDB and start the server with the *--upgrade* option once. This might write to the logfile of ArangoDB, so you may want to check the logs for any issues before going on.
- Start the "new" arangod service or binary regularly and check the logs for any issues. When you're confident everything went well, you may want to check the database directory for any files with the ending *.old*. These files are created by ArangoDB during upgrades and can be safely removed manually later.

If anything goes wrong during or shortly after the upgrade:

- Stop the "new" arangod service or binary
- Revert to the "old" arangod binary and restore the "old" data directory
- Start the "old" version again

It is not supported to use datafiles created or modified by a newer version of ArangoDB with an older ArangoDB version. For example, it is unsupported and is likely to cause problems when using 1.4 datafiles with an ArangoDB 1.3 instance.

Set up your ArangoDB Cluster

Setting up a cluster can be intimidating task. You have to deal with firewalls, ports, different types of machines, and the like. ArangoDB is prepared to deal with all kinds of different setups and requirements.

However, in the following section we concentrate on a standard setup and show you, how to build a ArangoDB cluster within minutes. If you want to dive deeper into the nasty details, you should read about Sharding.

Development Scenario

While not really relevant for a production environment, a common setup for development is to create a cluster on a single machine. This is the easiest of all setups and you should be ready to play with a ArangoDB cluster in less than a minute. Even when developing it is convenient to create a cluster on a single machine instead of having to deal with a lot of servers.

Step 1: Enable the Cluster mode

In order to enable the cluster mode, edit the configuration file as root

```
vi /etc/arangodb/arangod.conf
```

and change the lines

```
[cluster]
disable-dispatcher-kickstarter = yes
disable-dispatcher-frontend = yes
```

to

```
[cluster]
disable-dispatcher-kickstarter = no
disable-dispatcher-frontend = no
```

Save and restart

```
/etc/init.d/arangodb restart
```

Step 2: Setup your Cluster

You can now configure your cluster. A cluster consists of a number of database server and coordinators. A database servers holds you precious data, while a coordinator talks to the outside worlds, takes requests from clients, distributes them to database server and assembles the result.

For this example, we assume that are creating three database servers and two coordinators.

Use your favorite web browser and go to

```
http://localhost:8529/
```

You will now see the cluster management frontend.

Select *Single Machine* scenario. The next page allows you to enter the number of coordinators and database servers.

Press *Launch Cluster* to fire up the cluster. That's it. Your cluster is up and running.

Step 3: Test your Cluster

Click on one of the coordinators (e. g. "Claus") to access your cluster. In order to create a sharded collection, use *Tools / JS Shell* and execute

```
JSH> db._create("users", { numberOfShards: 3 });
JSH> db.users.save({ _key: "cmeier", firstName: "Claus", lastName:
"Meier" });
```

Congratulations! You have created your first sharded collection and stored a document in it.

First Steps in ArangoDB

For installation instructions, please refer to the [Installation Manual](#).

As you know from the introduction ArangoDB is a multi-purpose open-source Database. Following you can see the Key features or look at the programs in the ArangoDB package.

Key features include:

- *Schema-free schemata*: Let you combine the space efficiency of MySQL with the performance power of NoSQL
- *Application server*: Use ArangoDB as an application server and fuse your application and database together for maximal throughput
- *JavaScript for all*: No language zoo, you can use one language from your browser to your back-end
- *Flexible data modeling*: Model your data as combination of key-value pairs, documents or graphs - perfect for social relations
- *Free index choice*: Use the correct index for your problem, may it be a skip list or a fulltext search
- *Configurable durability*: Let the application decide if it needs more durability or more performance
- *No-nonsense storage*: ArangoDB uses all of the power of modern storage hardware, like SSD and large caches
- *Powerful query language* (AQL) to retrieve data
- *Transactions*: Run queries on multiple documents or collections with optional transactional consistency and isolation
- *Replication*: Set up the database in a master-slave configuration
- It is open source (*Apache Licence 2.0*)

For more in-depth information:

- Read more on the [Design Goals](#) of ArangoDB
- [Watch the video](#): Martin Schönert, architect of ArangoDB, gives an introduction of what the ArangoDB project is about
- Or give it a [try](#)

ArangoDB programs

The ArangoDB package comes with the following programs:

- *arangod*: The ArangoDB database daemon. This server program is intended to run as daemon process and to server the various clients connection to the server via TCP / HTTP. See [Details about the ArangoDB Server](#)
- *arangosh*: The ArangoDB shell. A client that implements a read-eval-print loop (REPL) and provides functions to access and administrate the ArangoDB server. See [Details about the ArangoDB Shell](#).
- *arangoimp*: A bulk importer for the ArangoDB server See [Details about Arangoimp](#).
- *arangodump*: A tool to create backups of an ArangoDB database. See [Details about Arangodump](#).
- *arangorestore*: A tool to reload data from a backup into an ArangoDB database. See [Details about Arangorestore](#).
- *foxx-manager*: A shell script to administer Foxx applications. See [Foxx Manager](#)
- *arango-dfdb*: A datafile debugger for ArangoDB. It is intended to be used primarily during development of ArangoDB. See @ref DbaManualDatafileDebugger

Getting familiar with ArangoDB

First of all download and install the corresponding RPM or Debian package or use homebrew on the MacOS X. See the [installation manual](#) for more details. In case you just want to experiment with ArangoDB you can use the [online](#) demo without installing ArangoDB locally.

For Linux

- Visit the official [ArangoDB download page](#) and download the correct package for your Linux distribution
- Install the package using your favorite package manager
- Start up the database server, normally this is done by executing */etc/init.d/arangod start*. The exact command depends on your Linux distribution

For MacOS X

- Execute *brew install arangodb*
- And start the server using */usr/local/sbin/arangod &*

For Microsoft Windows

- Visit the official [ArangoDB download page](#) and download the installer for Windows
- Start up the database server

After these steps there should be a running instance of *arangod* - the ArangoDB database server.

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

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

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

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

Details about the ArangoDB Server

The ArangoDB database server has two modes of operation: As a server, where it will answer to client requests and as an emergency console, in which you can access the database directly. The latter - as the name suggests - should only be used in case of an emergency, for example, a corrupted collection. Using the emergency console allows you to issue all commands normally available in actions and transactions. When starting the server in emergency console mode, the server cannot handle any client requests.

You should never start more than one server using the same database directory, independent from the mode of operation. Normally ArangoDB will prevent you from doing this by placing a lockfile in the database directory and not allowing a second ArangoDB instance to use the same database directory if a lockfile is already present.

The following command starts the ArangoDB database in server mode. You will be able to access the server using HTTP requests on port 8529. Look here for a list of frequently used options – see here for a complete list.

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

After starting the server, point your favorite browser to:

```
http://localhost:8529/
```

to access the administration front-end.

Linux

To start the server at system boot time you should use one of the pre-rolled packages that will install the necessary start / stop scripts for ArangoDB. You can use the start script as follows:

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

To stop the server you can use the following command:

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

You may require root privileges to execute these commands.

If you compiled ArangoDB from source and did not use any installation package – or using non-default locations and/or multiple ArangoDB instances on the same host – you may want to start the server process manually. You can do so by invoking the arangod binary from the command line as shown before. To stop the database server gracefully, you can either press CTRL-C or by send the SIGINT signal to the server process. On many systems this can be achieved with the following command:

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

Frequently Used Options

The following command-line options are frequently used. For a full list of options see <u>here</u>.

```
database-directory
```

Uses the "database-directory" as base directory. There is an alternative version available for use in configuration files, see <u>here</u>.

```
--help
-h
```

Prints a list of the most common options available and then exists. In order to see all options use `--help-all`.

```
--log level
```

Allows the user to choose the level of information which is logged by the server. The "level" is specified as a string and can be one of the following values: fatal, error, warning, info, debug or trace. For more information see <u>here</u>.

`--server.endpoint endpoint` Specifies an endpoint for HTTP requests by clients. Endpoints have the following pattern:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6
- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption
- unix:///path/to/socket - Unix domain socket endpoint

If a TCP/IP endpoint is specified without a port number, then the default port (8529) will be used. If multiple endpoints need to be used, the option can be repeated multiple times.

Examples

```
unix> ./arangod --server.endpoint tcp://127.0.0.1:8529
--server.endpoint ssl://127.0.0.1:8530
--server.keyfile server.pem /tmp/vocbase
2012-07-26T07:07:47Z [8161] INFO using SSL protocol version 'TLSv1'
2012-07-26T07:07:48Z [8161] INFO using endpoint
'ssl://127.0.0.1:8530' for http ssl requests
2012-07-26T07:07:48Z [8161] INFO using endpoint
'tcp://127.0.0.1:8529' for http tcp requests
2012-07-26T07:07:49Z [8161] INFO ArangoDB (version 1.1.alpha) is
ready for business
2012-07-26T07:07:49Z [8161] INFO Have Fun!
Note that if you are using SSL-encrypted endpoints, you must also
supply the path to a server certificate using the --server.keyfile
option.
```

Endpoints can also be changed at runtime. Please refer to HTTP Interface for Endpoints for more details.

```
--server.disable-authentication
```

Setting value to true will turn off authentication on the server side so all clients can execute any action without authorization and privilege checks.

The default value is *false*.

```
--server.keep-alive-timeout
```

Allows to specify the timeout for HTTP keep-alive connections. The timeout value must be specified in seconds. Idle keep-alive connections will be closed by the server automatically when the timeout is reached. A keep-alive-timeout value 0 will disable the keep alive feature entirely.

```
--daemon
```

Runs the server as a "daemon" (as a background process).

ArangoDB Shell Introduction

The ArangoDB shell (*arangosh*) is a command-line tool that can be used for administration of ArangoDB, including running ad-hoc queries.

The *arangosh* binary is shipped with ArangoDB and can be invoked like so:

```
unix> arangosh
```

By default *arangosh* will try to connect to an ArangoDB server running on server *localhost* on port *8529*. It will use the username *root* and an empty password by default. Additionally it will connect to the default database (*_system*). All these defaults can be changed using the following command-line options:

- *--server.database* : name of the database to connect to
- *--server.endpoint* : endpoint to connect to
- *--server.username* : database username
- *--server.password* : password to use when connecting
- *--server.disable-authentication* : disable password prompt and authentication

For example, to connect to an ArangoDB server on IP *192.168.173.13* on port 8530 with the user *foo* and using the database *test*, use:

```
unix> arangosh  \
  --server.endpoint tcp://192.168.173.13:8530  \
  --server.username foo  \
  --server.database test  \
  --server.disable-authentication false
```

arangosh will then display a password prompt and try to connect to the server after the password was entered.

To change the current database after the connection has been made, you can use the `db._useDatabase()` command in arangosh:

```
arangosh> db._useDatabase("myapp");
```

To get a list of available commands, arangosh provides a *help()* function. Calling it will display helpful information.

arangosh also provides auto-completion. Additional information on available commands and methods is thus provided by typing the first few letters of a variable and then pressing the tab key. It is recommend to try this with entering *db.* (without pressing return) and then pressing tab.

By the way, *arangosh* provides the *db* object by default, and this object can be used for switching to a different database and managing collections inside the current database.

For a list of available methods for the *db* object, type

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

ArangoDB Shell Output

In general the ArangoDB shell prints its output to the standard output channel using the JSON stringifier.

```
arangosh> db.five.toArray();
[{ "_id" : "five/3665447", "_rev" : "3665447", "name" : "one" },
{ "_id" : "five/3730983", "_rev" : "3730983", "name" : "two" },
{ "_id" : "five/3862055", "_rev" : "3862055", "name" : "four" },
{ "_id" : "five/3993127", "_rev" : "3993127", "name" : "three" }]
```

start_pretty_print()

While the standard JSON stringifier is very concise it is hard to read. Calling the function *start_pretty_print* will enable the pretty printer which formats the output in a human-readable way.

```
arangosh> start_pretty_print();
using pretty printing
arangosh> db.five.toArray();
[
  {
    "_id" : "five/3665447",
    "_rev" : "3665447",
    "name" : "one"
  },
  {
    "_id" : "five/3730983",
    "_rev" : "3730983",
    "name" : "two"
  },
  {
    "_id" : "five/3862055",
    "_rev" : "3862055",
    "name" : "four"
  },
  {
    "_id" : "five/3993127",
    "_rev" : "3993127",
    "name" : "three"
  }
]
```

stop_pretty_print()

The function disables the pretty printer, switching back to the standard dense JSON output format.

ArangoDB Shell Configuration

arangosh will look for a user-defined startup script named *.arangosh.rc* in the user's home directory on startup. If the file is present *arangosh* will execute the contents of this file inside the global scope.

You can use this to define your own extra variables and functions that you need often. For example, you could put the following into the *.arangosh.rc* file in your home directory:

```
// var keyword omitted intentionally,
// otherwise "timed" would not survive the scope of this script
timed = function (cb) {
    var internal = require("internal");
  var start = internal.time();
  cb();
  internal.print("execution took: ", internal.time() - start);
};
```

This will make a function named *timed* available in *arangosh* in the global scope.

You can now start *arangosh* and invoke the function like this:

```
timed(function () {
  for (var i = 0; i < 1000; ++i) {
    db.test.save({ value: i });
  }
});
```

Please keep in mind that, if present, the *.arangosh.rc* file needs to contain valid JavaScript code. If you want any variables in the global scope to survive you need to omit the *var* keyword for them. Otherwise the variables will only be visible inside the script itself, but not outside.

Details about the ArangoDB Shell

After the server has been [started](#), you can use the ArangoDB shell (*arangosh*) to administrate the server. Without any arguments, the ArangoDB shell will try to contact the server on port 8529 on the localhost. For more information see the [ArangoDB Shell documentation](#). You might need to set additional options (endpoint, username and password) when connecting:

```
unix> ./arangosh --server.endpoint tcp://127.0.0.1:8529 --
server.username root
```

The shell will print its own version number and – if successfully connected to a server – the version number of the ArangoDB server.

Command-Line Options

Use `--help` to get a list of command-line options:

```
unix> ./arangosh --help
STANDARD options:
  --audit-log <string>          audit log file to save commands and
results to
  --configuration <string>      read configuration file
  --help                        help message
  --max-upload-size <uint64>    maximum size of import chunks (in
bytes) (default: 500000)
  --no-auto-complete            disable auto completion
  --no-colors                   deactivate color support
  --pager <string>              output pager (default: "less -X -R
-F -L")
  --pretty-print                pretty print values
  --quiet                       no banner
  --temp-path <string>          path for temporary files (default:
"/tmp/arangodb")
  --use-pager                   use pager

JAVASCRIPT options:
  --javascript.check <string>            syntax check code
Javascript code from file
  --javascript.execute <string>          execute Javascript
code from file
  --javascript.execute-string <string>   execute Javascript
code from string
  --javascript.startup-directory <string>    startup paths
containing the JavaScript files
  --javascript.unit-tests <string>       do not start as shell,
run unit tests instead
  --jslint <string>                      do not start as shell,
run jslint instead

LOGGING options:
  --log.level <string>    log level (default: "info")

CLIENT options:
  --server.connect-timeout <double>      connect timeout in
seconds (default: 3)
  --server.disable-authentication <bool>    disable authentication
(default: false)
  --server.endpoint <string>             endpoint to connect to,
use 'none' to start without a server (default:
"tcp://127.0.0.1:8529")
  --server.password <string>             password to use when
connecting (leave empty for prompt)
  --server.request-timeout <double>      request timeout in
seconds (default: 300)
  --server.username <string>             username to use when
connecting (default: "root")
```

Exploring Collections and Documents

ArangoDB is a database that serves documents to clients.

- A *document* contains zero or more attributes, each one of these attributes has a value. A value can either be an atomic type, i. e. integer, strings, boolean, a list or an embedded document. Documents are normally represented as JSON objects
- Documents are grouped into *collections*. A collection contains zero or more documents
- *Queries* are used to filter documents based on certain criteria. Queries can be as simple as a "query by example" or as complex as "joins" using many collections or graph structures
- *Cursors* are used to iterate over the result of a query
- *Indexes* are used to speed up of searches. There are various different types of indexes like Index Hash, Index Geo and Index BitArray

If you are familiar with RDBMS then it is safe to compare collections to tables and documents to rows. However, bringing structure to the "rows" has many advantages - as you will see later.

Starting the JavaScript shell

The easiest way to connect to the database is the JavaScript shell *arangosh*. You can either start it from the command-line or as an embedded version in the browser. Using the command-line tool has the advantage that you can use autocompletion.

```
unix> arangosh --server.password ""
                      _
   /__`|`__/__`|`__\ /__`|/__\/__| '__\
  | (_| | | | (_| | | | | (_| | (_) \__\ | | |
   \__,_|_|  \__,_|_| |_|\__, |\___/|___/_| |_|
              |___/

Welcome to arangosh 1.x.y. Copyright (c) 2012 triAGENS GmbH.
Using Google V8 3.9.4 JavaScript engine.
Using READLINE 6.1.

Connected to Arango DB 127.0.0.1:8529 Version 1.x.y

------------------------------------ Help ------------------------
-------------
Predefined objects:
  arango:                                ArangoConnection
  db:                                    ArangoDatabase
  fm:                                    FoxxManager
Example:
 > db._collections();                    list all collections
 > db._create(<name>)                    create a new collection
 > db._drop(<name>)                      drop a collection
 > db.<name>.toArray()                   list all documents
 > id = db.<name>.save({ ... })          save a document
 > db.<name>.remove(<_id>)               delete a document
 > db.<name>.document(<_id>)             retrieve a document
 > db.<name>.replace(<_id>, {...})       overwrite a document
 > db.<name>.update(<_id>, {...})        partially update a document
 > db.<name>.exists(<_id>)               check if document exists
 > db._query(<query>).toArray()          execute an AQL query
 > db._useDatabase(<name>)               switch database
 > db._createDatabase(<name>)            create a new database
 > db._listDatabases()                   list existing databases
 > help                                  show help pages
 > exit
arangosh>
```

This gives you a prompt where you can issue JavaScript commands.

The standard setup does not require a password. Depending on your setup you might need to specify the endpoint, username and password in order to run the shell on your system. You can use the options `--server.endpoint`, `--server.username` and `--server.password` for this.

```
unix> arangosh --server.endpoint tcp://127.0.0.1:8529 --
server.username root
```

A default configuration is normally installed under */etc/arangodb/arangosh.conf*. It contains a default endpoint and an empty password.

Troubleshooting

If the ArangoDB server does not start or if you cannot connect to it using *arangosh* or other clients, you can try to find the problem cause by executing the following steps. If the server starts up without problems you can skip this section.

- *Check the server log file*: If the server has written a log file you should check it because it might contain relevant error context information.

- *Check the configuration*: The server looks for a configuration file named *arangod.conf* on startup. The contents of this file will be used as a base configuration that can optionally be overridden with command-line configuration parameters. You should check the config file for the most relevant parameters such as:

  - *server.endpoint*: What IP address and port to bind to
  - *log parameters*: If and where to log
  - *database.directory*: Path the database files are stored in

  If the configuration reveals that something is not configured right the config file should be adjusted and the server be restarted.

- *Start the server manually and check its output*: Starting the server might fail even before logging is activated so the server will not produce log output. This can happen if the server is configured to write the logs to a file that the server has no permissions on. In this case the server cannot log an error to the specified log file but will write a startup error on stderr instead. Starting the server manually will also allow you to override specific configuration options, e.g. to turn on/off file or screen logging etc.

- *Check the TCP port*: If the server starts up but does not accept any incoming connections this might be due to firewall configuration between the server and any client(s). The server by default will listen on TCP port 8529. Please make sure this port is actually accessible by other clients if you plan to use ArangoDB in a network setup.

  When using hostnames in the configuration or when connecting, please make sure the hostname is actually resolvable. Resolving hostnames might invoke DNS, which can be a source of errors on its own.

  It is generally good advice to not use DNS when specifying the endpoints and connection addresses. Using IP addresses instead will rule out DNS as a source of errors. Another alternative is to use a hostname specified in the local */etc/hosts* file, which will then bypass DNS.

- *Test if* curl *can connect*: Once the server is started, you can quickly verify if it responds to requests at all. This check allows you to determine whether connection errors are client-specific or not. If at least one client can connect, it is likely that connection problems of other clients are not due to ArangoDB's configuration but due to client or in-between network configurations.

  You can test connectivity using a simple command such as:

  curl --dump - -X GET http://127.0.0.1:8529/_api/version && echo

  This should return a response with an *HTTP 200* status code when the server is running. If it does it also means the server is generally accepting connections. Alternative tools to check connectivity are *lynx* or *ab*.

Querying for Documents

All documents are stored in collections. All collections are stored in a database. The database object is accessible via the variable *db*.

Creating a collection is simple. You can use the *_create* method of the *db* variable.

```
arangosh> db._create("example");
[ArangoCollection 70628, "example" (status loaded)]
```

After the collection has been created you can easily access it using the path *db.example*. The collection currently shows as *loaded*, meaning that it's loaded into memory. If you restart the server and access the collection again it will now show as *unloaded*. You can also manually unload a collection.

```
arangosh> db.example.unload();
arangosh> db.example;
[ArangoCollection 70628, "example" (status unloaded)]
```

Whenever you use a collection ArangoDB will automatically load it into memory for you.

In order to create new documents in a collection use the *save* operation.

```
arangosh> db.example.save({ Hello : "World" });
{ error : false, _id : "example/1512420", _key : "1512420", _rev :
"1512420" }
arangosh> db.example.save({ name : "John Doe", age : 29 });
{ error : false, _id : "example/1774564", _key : "1774564", _rev :
"1774564" }
arangosh> db.example.save({ name : "Jane Smith", age : 31 });
{ error : false, _id : "example/1993214", _key : "1993214", _rev :
"1993214" }
```

Just storing documents would be no fun. We now want to select some of the stored documents again. In order to select all elements of a collection, one can use the *all* operator. Because this might return a lot of data, we switch on pretty printing before.

```
arangosh> start_pretty_print();
use pretty printing
```

The command *stop_pretty_print()* will switch off pretty printing again. Now extract all elements:

```
arangosh> db.example.all().toArray()
[
  {
_id : "example/6308263",
    _key : "1993214",
_rev : "1993214",
age : 31,
name : "Jane Smith"
  },
  {
_id : "example/6242727",
    _key : "1774564",
```

```
_rev : "1774564",
age : 29,
name : "John Doe"
   },
   {
_id : "example/5980583",
    _key : "1512420",
_rev : "1512420",
Hello : "World"
   }
]
```

The last document was a mistake – so let's delete it:

```
arangosh> db.example.remove("example/5980583")
true
arangosh> db.example.all().toArray()
[
   {
_id : "example/6308263",
    _key : "1993214",
_rev : "1993214",
age : 31,
name : "Jane Smith"
   },
   {
_id : "example/6242727",
    _key : "1774564",
_rev : "1774564",
age : 29,
name : "John Doe"
   }
]
```

Now we want to look for a person with a given name. We can use *byExample* for this. The method returns a list of documents matching a given example.

```
arangosh> db.example.byExample({ name: "Jane Smith" }).toArray()
[
   {
_id : "example/6308263",
    _key : "1993214",
_rev : "1993214",
age : 31,
name : "Jane Smith"
   }
]
```

While the *byExample* works very well for simple queries where you combine the conditions with an `and`. The syntax above becomes messy for *joins* and *or* conditions. Therefore ArangoDB also supports a full-blown query language.

```
arangosh> db._query('FOR user IN example FILTER user.name == "Jane
Smith" RETURN user').toArray()
[
   {
```

```
  _id : "example/6308263",
     _key : "1993214",
 _rev : "1993214",
 age : 31,
 name : "Jane Smith"
    }
]
```

Search for all persons over 30:

```
arangosh> db._query('FOR user IN example FILTER user.age > 30
RETURN user').toArray()
[
   {
 _id : "example/6308263",
     _key : "1993214",
 _rev : "1993214",
 age : 31,
 name : "Jane Smith"
    }
]
```

You can learn all about the query language Aql. Note that _query_ is a short-cut for _createStatement_ and _execute_. We will come back to these functions when we talk about cursors.

ArangoDB's Front-End

The ArangoDB server has a graphical front-end, which allows you to inspect the current state of the server from within your browser. You can use the front-end using the following URL:

```
http://localhost:8529/_admin/html/index.html
```

Unless you have loaded an application into the ArangoDB server – which remaps the paths – the front-end will also be available under

```
http://localhost:8529/
```

The front-end allows you to browse through the collections and documents. If you need to administrate the database, please use the ArangoDB shell described in the next section.

Accessing the Web Interface

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

```
http://localhost:8529
```

assuming you are using the standard port and no user routings. If you have any application installed, the home page might point to that application instead. In this case use

http://localhost:8529/_admin/aardvark/index.html

(note: *aardvark* is the web interface's internal name).

If no database name is specified in the URL, you will in most cases get routed to the web interface for the *_system* database. To access the web interface for any other ArangoDB database, put the database name into the request URI path as follows:

http://localhost:8529/_db/mydb/

The above will load the web interface for the database *mydb*.

To restrict access to the web interface, use ArangoDB's authentication feature.

Select Functionality provided by the Web Interface

The following sections provide a very brief overview of some features offered in the web interface. Please note that this is not a complete list of features.

Dashboard Tab

The *Dashboard* tab provides statistics which are polled regularly, in a configurable interval, from the ArangoDB server. To adjust which figures should be displayed, please use the "gear" icon in the top right. This allows enabling or disabling the display of the individual figures, and to adjust the statistics poll & update interval. Please note that you can use the magnifying glass icon on any of the small detail charts to make the chart be display at the top "in big".

The *Dashboard* tab also shows an overview of the current databases replication status, including:

- Whether or not replication is running
- The progress of the replication
- If there has been a replication error

Collections Tab

The *Collections* tab shows an overview of the loaded and unloaded collections present in ArangoDB. System collections (i.e. collections whose names start with an underscore) are not shown by default.

The list of collections can be restricted using the search bar or by using the filtering at the top. The filter can also be used to show or hide system collections.

Clicking on a collection will show the documents contained in it. Clicking the small icon on a collection's badge will bring up a dialog that allows loading/unloading, renaming and deleting the collection.

Please note that you should not change or delete system collections.

In the list of documents of a collection, you can click on the *Add document* line to add a new document to the collection. The document will be created instantly, with a system-defined key. The key and all other attributes of the document can be adjusted in the following view.

Applications Tab

The *Applications* tab provides a list of installed Foxx applications. The view is divided into lists of installed and applications that are available for installation.

Please note that ArangoDB's web interface (*aardvark*) is a Foxx application itself. Please also note that installed applications will be listed in both the *installed* and the *available* section. This is intentional because each application can be installed multiple times using different mount points.

Graphs Tab

The *Graphs* tab provides a viewer facility for graph data stored in ArangoDB. It allows browsing ArangoDB graphs stored in the *_graphs* system collection or a graph consisting of an arbitrary vertex and edge collection.

Please note that the graph viewer requires client-side SVG and that you need a browser capable of rendering that. Especially Internet Explorer browsers older than version 9 are likely to not support this.

AQL Editor Tab

The *AQL Editor* tab allows to execute ad-hoc AQL queries.

Type in a query in the bottom box and execute it by pressing the *Submit* button. The query result will be shown in the box at the top. The editor provides a few example queries that can be used as templates.

There is also the option to add own frequently used queries here. Note that own queries will be stored in the browser's local storage and the web interface has no control over when the browser's local storage is cleared.

JS Shell Tab

The *JS Shell* tab provides access to a JavaScript shell connection to the database server.

Any valid JavaScript code can be executed inside the shell. The code will be executed inside your browser. To contact the ArangoDB server you can use the *db* object, for example as follows:

```
JSH> db._create("mycollection");
JSH> db.mycollection.save({ _key: "test", value: "something"
});
```

Logs Tab

You can use the *Logs* tab to browse the most recent log entries provided by the ArangoDB database server.

Note that the server only keeps a limited number of log entries. For real log analyses write the logs to disk using syslog or a similar mechanism. ArangoDB provides several startup options for this.

The *Logs* tab will only be shown for the `_system` database, and is disabled for any other databases.

API Tab

The *API* tab provides an overview of ArangoDB's built-in HTTP REST API, with documentation and examples. It should be consulted when there is doubt about API URLs, parameters etc.

Handling Databases

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

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

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

Connecting to a specific database from arangosh is possible with the above command after arangosh has been started, but it is also possible to specify a database name when invoking arangosh. For this purpose, use the command-line parameter *--server.database*, e.g.

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

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

Working with Databases

Database Methods

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

Name

```
db._name()
```

Returns the name of the current database as a string.

ID

```
db._id()
```

Returns the id of the current database as a string.

Path

```
db._path()
```

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

isSystem

```
db._isSystem()
```

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

Use Database

```
db._useDatabase(name)
```

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

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

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

Is Database

```
db._listDatabases()
```

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

Create Database

```
db._createDatabase(name, options, users)
```

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

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

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

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

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

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

This method can only be used from within the *_system* database.

Drop Database

```
db._dropDatabase(name)
```

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

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

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

Notes about Databases

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

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

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

JavaScript Interface to Collections

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

The most import call is the call to create a new collection

Address of a Collection

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

Collection

db._collection(collection-name)

A collection is created by a ["db._create"](#) call.

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

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

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

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

Collection name

db.collection-name

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

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

Create

db._create(collection-name)

This call will create a new collection called *collection-name*.

Collection Methods

Drop

```
collection.drop()
```

Drops a *collection* and all its indexes.

Examples

```
arangosh> col = db.example;
[ArangoCollection 706896565, "example" (type document, status
loaded)]
arangosh> col.drop();
arangosh> col;
[ArangoCollection 706896565, "example" (type document, status
deleted)]
```

Truncate

```
collection.truncate()
```

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

Examples

Truncates a collection:

```
arangosh> col = db.example;
[ArangoCollection 140337845, "example" (type document, status
loaded)]
arangosh> col.save({ "Hello" : "World" });
{
  "error" : false,
  "_id" : "example/140665525",
  "_rev" : "140665525",
  "_key" : "140665525"
}
arangosh> col.count();
1
arangosh> col.truncate();
arangosh> col.count();
0
```

Properties

```
collection.properties()
```

Returns an object containing all collection properties.

- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.

- *journalSize* : The size of the journal in bytes.

- *isVolatile*: If *true* then the collection data will be kept in memory only and ArangoDB will not write or sync the data to disk.

- *keyOptions* (optional) additional options for key generation. This is a JSON array containing the following attributes (note: some of the attributes are optional):
    - *type*: the type of the key generator used for the collection.
    - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the _key attribute of a document. If set to *false*, then the key generator will solely be responsible for generating keys and supplying own key values in the _key attribute of documents is considered an error.
    - *increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
    - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.

In a cluster setup, the result will also contain the following attributes:

- *numberOfShards*: the number of shards of the collection.

- *shardKeys*: contains the names of document attributes that are used to determine the target shard for documents.

```
collection.properties(properties)
```

Changes the collection properties. *properties* must be a object with one or more of the following attribute(s):

- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.

- *journalSize* : The size of the journal in bytes.

Note: it is not possible to change the journal size after the journal or datafile has been created. Changing this parameter will only effect newly created journals. Also note that you cannot lower the journal size to less then size of the largest document already stored in the collection.

Note: some other collection properties, such as *type*, *isVolatile*, or *keyOptions* cannot be changed once the collection is created.

Examples

Read all properties

```
arangosh> db.example.properties();
{
  "doCompact" : true,
  "journalSize" : 1048576,
  "isSystem" : false,
  "isVolatile" : false,
  "waitForSync" : false,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  }
}
```

Change a property

```
arangosh> db.example.properties({ waitForSync : true });
{
  "doCompact" : true,
  "journalSize" : 1048576,
  "isSystem" : false,
```

```
    "isVolatile" : false,
    "waitForSync" : true,
    "keyOptions" : {
      "type" : "traditional",
      "allowUserKeys" : true
    }
}
```

Figures

```
collection.figures()
```

Returns an object containing all collection figures.

- *alive.count*: The number of living documents.
- *alive.size*: The total size in bytes used by all living documents.
- *dead.count*: The number of dead documents.
- *dead.size*: The total size in bytes used by all dead documents.
- *dead.deletion*: The total number of deletion markers.
- *datafiles.count*: The number of active datafiles.
- *datafiles.fileSize*: The total filesize of the active datafiles (in bytes).
- *journals.count*: The number of journal files.
- *journals.fileSize*: The total filesize of the journal files (in bytes).
- *compactors.count*: The number of compactor files.
- *compactors.fileSize*: The total filesize of the compactor files (in bytes).
- *shapefiles.count*: The number of shape files. This value is deprecated and kept for compatibility reasons only. The value will always be 0.
- *shapefiles.fileSize*: The total filesize of the shape files. This value is deprecated and kept for compatibility reasons only. The value will always be 0.
- *shapes.count*: The total number of shapes used in the collection. This includes shapes that are not in use anymore.
- *shapes.size*: The total size of all shapes (in bytes). This includes shapes that are not in use anymore.
- *attributes.count*: The total number of attributes used in the collection. Note: the value includes data of attributes that are not in use anymore.
- *attributes.size*: The total size of the attribute data (in bytes). Note: the value includes data of attributes that are not in use anymore.
- *indexes.count*: The total number of indexes defined for the collection, including the pre-defined indexes (e.g. primary index).
- *indexes.size*: The total memory allocated for indexes in bytes.
- *maxTick*: The tick of the last marker that was stored in a journal of the collection. This might be 0 if the collection does not yet have a journal.
- *uncollectedLogfileEntries*: The number of markers in the write-aheads for this collection that have not been transferred in datafiles / journals of the collection.

Examples

```
arangosh> db.demo.figures()
{
  "alive" : {
    "count" : 0,
    "size" : 0
  },
  "dead" : {
    "count" : 0,
    "size" : 0,
    "deletion" : 0
  },
```

```
   "datafiles" : {
     "count" : 0,
     "fileSize" : 0
   },
   "journals" : {
     "count" : 0,
     "fileSize" : 0
   },
   "compactors" : {
     "count" : 0,
     "fileSize" : 0
   },
   "shapefiles" : {
     "count" : 0,
     "fileSize" : 0
   },
   "shapes" : {
     "count" : 0,
     "size" : 0
   },
   "attributes" : {
     "count" : 0,
     "size" : 0
   },
   "indexes" : {
     "count" : 1,
     "size" : 2008
   },
   "lastTick" : "0",
   "uncollectedLogfileEntries" : 1
}
```

Load

```
collection.load()
```

Loads a collection into memory.

Examples

```
arangosh> col = db.example;
[ArangoCollection 329671349, "example" (type document, status
loaded)]
arangosh> col.load();
arangosh> col;
[ArangoCollection 329671349, "example" (type document, status
loaded)]
```

Reserve

```
collection.reserve( number)
```

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

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

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

Revision

```
collection.revision()
```

Returns the revision id of the collection

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

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

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

Checksum

```
collection.checksum(withRevisions, withData)
```

The *checksum* operation calculates a CRC32 checksum of the keys contained in collection *collection*.

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

If the optional argument *withData* is set to *true*, then the actual document data is also checksummed. Including the document data in checksumming will make the calculation slower, but is more accurate.

Note: this method is not available in a cluster.

Unload

```
collection.unload()
```

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

Examples

```
arangosh> col = db.example;
[ArangoCollection 51667637, "example" (type document, status
loaded)]
arangosh> col.unload();
arangosh> col;
[ArangoCollection 51667637, "example" (type document, status
unloaded)]
```

Rename

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

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

If renaming fails for any reason, an error is thrown.

Note: this method is not available in a cluster.

Examples

```
arangosh> c = db.example;
[ArangoCollection 616456885, "example" (type document, status
loaded)]
arangosh> c.rename("better-example");
arangosh> c;
[ArangoCollection 616456885, "better-example" (type document,
status loaded)]
```

Rotate

```
collection.rotate()
```

Rotates the current journal of a collection (i.e. makes the journal a read-only datafile). The purpose of the rotation is to include the datafile in a following compaction run and perform earlier garbage collection.

Note: this method is not available in a cluster.

Database Methods

Collection

```
db._collection(collection-name)
```

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

```
db._collection(collection-identifier)
```

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

Examples

Get a collection by name:

```
arangosh> db._collection("demo");
[ArangoCollection 25256629, "demo" (type document, status loaded)]
```

Get a collection by id:

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

Unknown collection:

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

Create

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

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

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

properties must be an object with the following attributes:

- *waitForSync* (optional, default *false*): If *true* creating a document will only return after the data was synced to disk.

- *journalSize* (optional, default is a configuration parameter: The maximal size of a journal or datafile. Note that this also limits the maximal size of a single object. Must be at least 1MB.

- *isSystem* (optional, default is *false*): If *true*, create a system collection. In this case *collection-name* should start with an underscore. End users should normally create non-system collections only. API implementors may be required to create system collections in very special occasions, but normally a regular collection will do.

- *isVolatile* (optional, default is *false*): If *true then the collection data is kept in-memory only and not made persistent. Unloading the collection will cause the collection data to be discarded.

Stopping or re-starting the server will also cause full loss of data in the collection. Setting this option will make the resulting collection be slightly faster than regular collections because ArangoDB does not enforce any synchronization to disk and does not calculate any CRC checksums for datafiles (as there are no datafiles).

- *keyOptions* (optional): additional options for key generation. If specified, then *keyOptions* should be a JSON array containing the following attributes (**note**: some of them are optional):
  - *type*: specifies the type of the key generator. The currently available generators are *traditional* and *autoincrement*.
  - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the *_key* attribute of a document. If set to *false*, then the key generator will solely be responsible for generating keys and supplying own key values in the *_key* attribute of documents is considered an error.
  - *{increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
  - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.

- *numberOfShards* (optional, default is *1*): in a cluster, this value determines the number of shards to create for the collection. In a single server setup, this option is meaningless.

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

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

Examples

With defaults:

```
arangosh> c = db._create("users");
[ArangoCollection 515465909, "users" (type document, status
loaded)]
arangosh> c.properties();
{
  "doCompact" : true,
  "journalSize" : 1048576,
  "isSystem" : false,
  "isVolatile" : false,
  "waitForSync" : false,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  }
}
```

With properties:

```
arangosh> c = db._create("users", { waitForSync : true, journalSize
: 1024 * 1204 });
[ArangoCollection 505176757, "users" (type document, status
loaded)]
arangosh> c.properties();
{
  "doCompact" : true,
  "journalSize" : 1232896,
  "isSystem" : false,
  "isVolatile" : false,
  "waitForSync" : true,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  }
}
```

With a key generator:

```
arangosh> db._create("users", { keyOptions: { type:
"autoincrement", offset: 10, increment: 5 } });
[ArangoCollection 98984629, "users" (type document, status loaded)]
arangosh> db.users.save({ name: "user 1" });
{
  "error" : false,
  "_id" : "users/10",
  "_rev" : "99312309",
  "_key" : "10"
}
arangosh> db.users.save({ name: "user 2" });
{
  "error" : false,
  "_id" : "users/15",
  "_rev" : "99508917",
  "_key" : "15"
}
arangosh> db.users.save({ name: "user 3" });
{
  "error" : false,
  "_id" : "users/20",
  "_rev" : "99705525",
  "_key" : "20"
}
```

With a special key option:

```
arangosh> db._create("users", { keyOptions: { allowUserKeys: false
} });
[ArangoCollection 636969653, "users" (type document, status
loaded)]
arangosh> db.users.save({ name: "user 1" });
{
  "error" : false,
  "_id" : "users/637297333",
  "_rev" : "637297333",
  "_key" : "637297333"
}
```

```
arangosh> db.users.save({ name: "user 2", _key: "myuser" });
[ArangoError 1222: collection does not allow using user-defined
keys]
arangosh> db.users.save({ name: "user 3" });
{
  "error" : false,
  "_id" : "users/637625013",
  "_rev" : "637625013",
  "_key" : "637625013"
}
```

All Collections

```
db._collections()
```

Returns all collections of the given database.

Examples

```
arangosh> db.example.load();
arangosh> var d = db.demo;
arangosh> db._collections();
[
  [ArangoCollection 4022965, "_aqlfunctions" (type document, status
loaded)],
  [ArangoCollection 1794741, "_modules" (type document, status
loaded)],
  [ArangoCollection 3564213, "_aal" (type document, status
loaded)],
  [ArangoCollection 691036853, "male" (type document, status
loaded)],
  [ArangoCollection 701260469, "germanHighway" (type edge, status
loaded)],
  [ArangoCollection 25256629, "demo" (type document, status
loaded)],
  [ArangoCollection 701653685, "internationalHighway" (type edge,
status loaded)],
  [ArangoCollection 1663669, "_graphs" (type document, status
loaded)],
  [ArangoCollection 26108597, "animals" (type document, status
loaded)],
  [ArangoCollection 2056885, "_cluster_kickstarter_plans" (type
document, status loaded)],
  [ArangoCollection 690905781, "female" (type document, status
loaded)],
  [ArangoCollection 701129397, "germanCity" (type document, status
loaded)],
  [ArangoCollection 497443509, "myEC1" (type edge, status loaded)],
  [ArangoCollection 284385973, "_statisticsRaw" (type document,
status loaded)],
  [ArangoCollection 701522613, "frenchHighway" (type edge, status
loaded)],
  [ArangoCollection 284779189, "_statistics" (type document, status
loaded)],
  [ArangoCollection 156341, "_users" (type document, status
loaded)],
  [ArangoCollection 4154037, "_configuration" (type document,
status loaded)],
```

```
   [ArangoCollection 1925813, "_routing" (type document, status
loaded)],
   [ArangoCollection 285172405, "_statistics15" (type document,
status loaded)],
   [ArangoCollection 616456885, "better-example" (type document,
status loaded)],
   [ArangoCollection 691167925, "relation" (type edge, status
loaded)],
   [ArangoCollection 701391541, "frenchCity" (type document, status
loaded)],
   [ArangoCollection 705782453, "example" (type document, status
loaded)]
]
```

Collection Name

`db.collection-name`

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

Examples

```
arangosh> db.example;
[ArangoCollection 317285045, "example" (type document, status
loaded)]
```

Drop

`db._drop(collection)`

Drops a *collection* and all its indexes.

`db._drop(collection-identifier)`

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

`db._drop(collection-name)`

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

Examples

Drops a collection:

```
arangosh> col = db.example;
[ArangoCollection 666985141, "example" (type document, status
loaded)]
arangosh> db._drop(col);
arangosh> col;
[ArangoCollection 666985141, "example" (type document, status
loaded)]
```

Drops a collection identified by name:

```
arangosh> col = db.example;
[ArangoCollection 478700213, "example" (type document, status
loaded)]
arangosh> db._drop("example");
arangosh> col;
[ArangoCollection 478700213, "example" (type document, status
deleted)]
```

Truncate

```
db._truncate(collection)
```

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

```
db._truncate(collection-identifier)
```

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

```
db._truncate(collection-name)
```

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

Examples

Truncates a collection:

```
arangosh> col = db.example;
[ArangoCollection 30630581, "example" (type document, status
loaded)]
arangosh> col.save({ "Hello" : "World" });
{
  "error" : false,
  "_id" : "example/30958261",
  "_rev" : "30958261",
  "_key" : "30958261"
}
arangosh> col.count();
1
arangosh> db._truncate(col);
arangosh> col.count();
0
```

Truncates a collection identified by name:

```
arangosh> col = db.example;
[ArangoCollection 311583413, "example" (type document, status
loaded)]
arangosh> col.save({ "Hello" : "World" });
{
  "error" : false,
  "_id" : "example/311911093",
  "_rev" : "311911093",
  "_key" : "311911093"
}
arangosh> col.count();
1
arangosh> db._truncate("example");
```

```
arangosh> col.count();
0
```

Documents, Identifiers, Handles

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

Documents in ArangoDB are JSON objects. These objects can be nested (to any depth) and may contain lists. Each document is uniquely identified by its document handle.

For example:

```
{
  "firstName" : "Hugo",
  "lastName" : "Schlonz",
  "address" : {
    "city" : "Hier",
    "street" : "Strasse 1"
  },
  "hobbies" : [
    "swimming",
    "biking",
    "programming"
  ],
  "_id" : "demo/schlonz",
  "_rev" : "13728680",
  "_key" : "schlonz"
}
```

All documents contain special attributes: the document handle in *_id*, the document's unique key in *_key* and and the ETag aka document revision in *_rev*. The value of the *_key* attribute can be specified by the user when creating a document. *_id* and *_key* values are immutable once the document has been created. The *_rev* value is maintained by ArangoDB autonomously.

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

As ArangoDB supports MVCC, documents can exist in more than one revision. The document revision is the MVCC token used to identify a particular revision of a document. It is a string value currently containing an integer number and is unique within the list of document revisions for a single document. Document revisions can be used to conditionally update, replace or delete documents in the database. In order to find a particular revision of a document, you need the document handle and the document revision.

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

Note: Revision ids have been returned as integers up to including ArangoDB 1.1

Document Etag: The document revision enclosed in double quotes. The revision is returned by several HTTP API methods in the Etag HTTP header.

Address and ETag of a Document

All documents in ArangoDB have a document handle. This handle uniquely defines a document and is managed by ArangoDB. The interface allows you to access the documents of a collection as:

```
db.collection.document("document-handle")
```

For example: Assume that the document handle, which is stored in the _id field of the document, is *demo/362549* and the document lives in a collection named *demo*, then that document can be accessed as:

```
db.demo.document("demo/362549736")
```

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

```
db._document("demo/362549736")
```

Each document also has a document revision or ETag which is returned in the *_rev* field when requesting a document. The document's key is returned in the *_key* attribute.

Collection Methods

All

```
collection.all()
```

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

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.five.all().toArray();
[
  {
    "_id" : "five/633692853",
    "_rev" : "633692853",
    "_key" : "633692853",
    "name" : "five"
  },
  {
    "_id" : "five/633299637",
    "_rev" : "633299637",
    "_key" : "633299637",
    "name" : "three"
  },
  {
    "_id" : "five/633496245",
    "_rev" : "633496245",
    "_key" : "633496245",
    "name" : "four"
  },
  {
    "_id" : "five/632906421",
    "_rev" : "632906421",
    "_key" : "632906421",
    "name" : "one"
  },
  {
    "_id" : "five/633103029",
    "_rev" : "633103029",
    "_key" : "633103029",
    "name" : "two"
  }
]
```

Use *limit* to restrict the documents:

```
arangosh> db.five.all().limit(2).toArray();
[
  {
    "_id" : "five/144007861",
    "_rev" : "144007861",
    "_key" : "144007861",
    "name" : "five"
```

```
    },
    {
      "_id" : "five/143418037",
      "_rev" : "143418037",
      "_key" : "143418037",
      "name" : "two"
    }
  ]
```

Query by example

```
collection.byExample(example)
```

Selects all documents of a collection that match the specified example and returns a cursor.

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

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

{ a : { c : 1 } }

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

{ a : { c : 1 }\, b : 1 }

will match, but the document

{ a : { c : 1\, b : 1 } }

will not.

However, if you use

{ a.c : 1 },

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

{ a : { c : 1 }\, b : 1 } and

{ a : { c : 1\, b : 1 } }

will match.

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

As alternative you can supply a list of paths and values.

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.users.all().toArray();
[
  {
    "_id" : "users/343433909",
    "_rev" : "343433909",
    "_key" : "343433909",
```

```
      "name" : "Gerhard"
   },
   {
      "_id" : "users/343892661",
      "_rev" : "343892661",
      "_key" : "343892661",
      "name" : "Angela"
   },
   {
      "_id" : "users/343696053",
      "_rev" : "343696053",
      "_key" : "343696053",
      "name" : "Helmut"
   }
]
arangosh> db.users.byExample({ "_id" : "users/20" }).toArray();
[ ]
arangosh> db.users.byExample({ "name" : "Gerhard" }).toArray();
[
   {
      "_id" : "users/343433909",
      "_rev" : "343433909",
      "_key" : "343433909",
      "name" : "Gerhard"
   }
]
arangosh> db.users.byExample({ "name" : "Helmut", "_id" :
"users/15" }).toArray();
[ ]
```

Use *next* to loop over all documents:

```
arangosh> var a = db.users.byExample( {"name" : "Angela" } );
arangosh> while (a.hasNext()) print(a.next());
{
   "_id" : "users/641229493",
   "_rev" : "641229493",
   "_key" : "641229493",
   "name" : "Angela"
}
```

First Example

collection.firstExample(example)

Returns the first document of a collection that matches the specified example or *null*. The example must be specified as paths and values. See *byExample* for details.

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

As alternative you can supply a list of paths and values.

Examples

```
arangosh> db.users.firstExample("name", "Angela");
{
   "_id" : "users/123822773",
```

```
  "_rev" : "123822773",
  "_key" : "123822773",
  "name" : "Angela"
}
```

Range

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

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

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

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

For range queries it is required that a skiplist index is present for the queried attribute. If no skiplist index is present on the attribute, an error will be thrown.

Examples

Use *toArray* to get all documents at once:

```
arangosh> db.old.range("age", 10, 30).toArray();
[
  {
    "_id" : "old/424960693",
    "_rev" : "424960693",
    "_key" : "424960693",
    "age" : 15
  },
  {
    "_id" : "old/425157301",
    "_rev" : "425157301",
    "_key" : "425157301",
    "age" : 25
  }
]
```

Any

```
collection.any()
```

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

Count

```
collection.count()
```

Returns the number of living documents in the collection.

Examples

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

toArray

```
collection.toArray()
```

Converts the collection into an array of documents. Never use this call in a production environment.

Document

```
collection.document(document)
```

The *document* method finds a document given its identifier or a document object containing the *_id* or *_key* attribute. The method returns the document if it can be found.

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

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

```
collection.document(document-handle)
```

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

Examples

Returns the document for a document-handle:

```
arangosh> db.example.document("example/2873916");
{
  "_id" : "example/2873916",
  "_rev" : "94986933",
  "_key" : "2873916"
}
```

An error is raised if the document is unknown:

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

An error is raised if the handle is invalid:

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

Exists

```
collection.exists(document)
```

The *exists* method determines whether a document exists given its identifier. Instead of returning the found document or an error, this method will return either *true* or *false*. It can thus be used for easy existence checks.

The *document* method finds a document given its identifier. It returns the document. Note that the returned document contains two pseudo-attributes, namely _id and _rev. _id contains the document-handle and _rev the revision of the document.

No error will be thrown if the sought document or collection does not exist. Still this method will throw an error if used improperly, e.g. when called with a non-document handle, a non-document, or when a cross-collection request is performed.

```
collection.exists(document-handle)
```

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

Save

```
collection.save(data)
```

Creates a new document in the *collection* from the given *data*. The *data* must be a hash array. It must not contain attributes starting with _.

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

```
collection.save(data, waitForSync)
```

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

Examples

```
arangosh> db.example.save({ Hello : "World" });
{
  "error" : false,
  "_id" : "example/409559733",
  "_rev" : "409559733",
  "_key" : "409559733"
}
arangosh> db.example.save({ Hello : "World" }, true);
{
  "error" : false,
  "_id" : "example/409756341",
  "_rev" : "409756341",
  "_key" : "409756341"
}
```

Replace

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

Replaces an existing *document*. The *document* must be a document in the current collection. This document is then replaced with the *data* given as second argument.

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

If there is a conflict, i. e. if the revision of the *document* does not match the revision in the collection, then an error is thrown.

```
collection.replace(document, data, true) or collection.replace(document,
data, overwrite: true)
```

As before, but in case of a conflict, the conflict is ignored and the old document is overwritten.

```
collection.replace(document, data, true, waitForSync) or
collection.replace(document, data, overwrite: true, waitForSync: true
or false)
```

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

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

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

Examples

Create and update a document:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/384983733",
  "_rev" : "384983733",
  "_key" : "384983733"
}
arangosh> a2 = db.example.replace(a1, { a : 2 });
{
  "error" : false,
  "_id" : "example/384983733",
  "_rev" : "385180341",
  "_key" : "384983733"
}
arangosh> a3 = db.example.replace(a1, { a : 3 });
[ArangoError 1200: precondition failed]
```

Use a document handle:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/357393077",
  "_rev" : "357393077",
  "_key" : "357393077"
}
arangosh> a2 = db.example.replace("example/3903044", { a : 2 });
{
  "error" : false,
  "_id" : "example/3903044",
  "_rev" : "357589685",
  "_key" : "3903044"
```

```
    }
```

Update

```
collection.update(document, data, overwrite, keepNull, waitForSync) or
collection.update(document, data, overwrite: true or false, keepNull:
true or false, waitForSync: true or false)
```

Updates an existing *document*. The *document* must be a document in the current collection. This document is then patched with the *data* given as second argument. The optional *overwrite* parameter can be used to control the behavior in case of version conflicts (see below). The optional *keepNull* parameter can be used to modify the behavior when handling *null* values. Normally, *null* values are stored in the database. By setting the *keepNull* parameter to *false*, this behavior can be changed so that all attributes in *data* with *null* values will be removed from the target document.

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

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

If there is a conflict, i. e. if the revision of the *document* does not match the revision in the collection, then an error is thrown.

```
collection.update(document, data, true)
```

As before, but in case of a conflict, the conflict is ignored and the old document is overwritten.

collection.update(document-handle, data)`

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

Examples

Create and update a document:

```
arangosh> a1 = db.example.save({"a" : 1});
{
  "error" : false,
  "_id" : "example/330261173",
  "_rev" : "330261173",
  "_key" : "330261173"
}
arangosh> a2 = db.example.update(a1, {"b" : 2, "c" : 3});
{
  "error" : false,
  "_id" : "example/330261173",
  "_rev" : "330654389",
  "_key" : "330261173"
}
arangosh> a3 = db.example.update(a1, {"d" : 4});
[ArangoError 1200: precondition failed]
arangosh> a4 = db.example.update(a2, {"e" : 5, "f" : 6 });
{
```

```
    "error" : false,
    "_id" : "example/330261173",
    "_rev" : "331309749",
    "_key" : "330261173"
}
arangosh> db.example.document(a4);
{
    "a" : 1,
    "b" : 2,
    "c" : 3,
    "e" : 5,
    "f" : 6,
    "_id" : "example/330261173",
    "_rev" : "331309749",
    "_key" : "330261173"
}
arangosh> a5 = db.example.update(a4, {"a" : 1, c : 9, e : 42 });
{
    "error" : false,
    "_id" : "example/330261173",
    "_rev" : "331571893",
    "_key" : "330261173"
}
arangosh> db.example.document(a5);
{
    "a" : 1,
    "b" : 2,
    "c" : 9,
    "e" : 42,
    "f" : 6,
    "_id" : "example/330261173",
    "_rev" : "331571893",
    "_key" : "330261173"
}
```

Use a document handle:

```
arangosh> a1 = db.example.save({"a" : 1});
{
    "error" : false,
    "_id" : "example/52454069",
    "_rev" : "52454069",
    "_key" : "52454069"
}
arangosh> a2 = db.example.update("example/18612115", { "x" : 1, "y"
: 2 });
{
    "error" : false,
    "_id" : "example/18612115",
    "_rev" : "52847285",
    "_key" : "18612115"
}
```

Use the keepNull parameter to remove attributes with null values:

```
arangosh> db.example.save({"a" : 1});
{
```

```
    "error" : false,
    "_id" : "example/152593077",
    "_rev" : "152593077",
    "_key" : "152593077"
}
arangosh> db.example.update("example/19988371", { "b" : null, "c" :
null, "d" : 3 });
{
    "error" : false,
    "_id" : "example/19988371",
    "_rev" : "153051829",
    "_key" : "19988371"
}
arangosh> db.example.document("example/19988371");
{
    "b" : null,
    "c" : null,
    "d" : 3,
    "_id" : "example/19988371",
    "_rev" : "153051829",
    "_key" : "19988371"
}
arangosh> db.example.update("example/19988371", { "a" : null },
false, false);
{
    "error" : false,
    "_id" : "example/19988371",
    "_rev" : "153313973",
    "_key" : "19988371"
}
arangosh> db.example.document("example/19988371");
{
    "b" : null,
    "c" : null,
    "d" : 3,
    "_id" : "example/19988371",
    "_rev" : "153313973",
    "_key" : "19988371"
}
arangosh> db.example.update("example/19988371", { "b" : null, "c":
null, "d" : null }, false, false);
{
    "error" : false,
    "_id" : "example/19988371",
    "_rev" : "153576117",
    "_key" : "19988371"
}
arangosh> db.example.document("example/19988371");
{
    "_id" : "example/19988371",
    "_rev" : "153576117",
    "_key" : "19988371"
}
```

Patching array values:

```
arangosh> db.example.save({"a" : { "one" : 1, "two" : 2, "three" :
3 }, "b" : { }});
```

```
{
  "error" : false,
  "_id" : "example/495805109",
  "_rev" : "495805109",
  "_key" : "495805109"
}
arangosh> db.example.update("example/20774803", {"a" : { "four" : 4
}, "b" : { "b1" : 1 }});
{
  "error" : false,
  "_id" : "example/20774803",
  "_rev" : "496329397",
  "_key" : "20774803"
}
arangosh> db.example.document("example/20774803");
{
  "a" : {
    "four" : 4
  },
  "b" : {
    "b1" : 1
  },
  "_id" : "example/20774803",
  "_rev" : "496329397",
  "_key" : "20774803"
}
arangosh> db.example.update("example/20774803", { "a" : { "one" :
null }, "b" : null }, false, false);
{
  "error" : false,
  "_id" : "example/20774803",
  "_rev" : "496657077",
  "_key" : "20774803"
}
arangosh> db.example.document("example/20774803");
{
  "a" : {
    "four" : 4
  },
  "_id" : "example/20774803",
  "_rev" : "496657077",
  "_key" : "20774803"
}
```

Remove

```
db._remove(document)
```

Removes a document. If there is revision mismatch, then an error is thrown.

```
db._remove(document, true)
```

Removes a document. If there is revision mismatch, then mismatch is ignored and document is deleted. The function returns *true* if the document existed and was deleted. It returns *false*, if the document was already deleted.

```
db._remove(document, true, waitForSync) or db._remove(document,
{overwrite: true or false, waitForSync: true or false})
```

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

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

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

Examples

Remove a document:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/316629685",
  "_rev" : "316629685",
  "_key" : "316629685"
}
arangosh> db._remove(a1);
true
arangosh> db._remove(a1);
[ArangoError 1202: document /_api/document/example/316629685 not
found]
arangosh> db._remove(a1, true);
false
```

Remove a document with a conflict:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/747528885",
  "_rev" : "747528885",
  "_key" : "747528885"
}
arangosh> a2 = db._replace(a1, { a : 2 });
{
  "error" : false,
  "_id" : "example/747528885",
  "_rev" : "747725493",
  "_key" : "747528885"
}
arangosh> db._delete(a1);
TypeError: Object [object Object] has no method '_delete'
arangosh> db._delete(a1, true);
TypeError: Object [object Object] has no method '_delete'
arangosh> db._document(a1);
[ArangoError 1200: precondition failed]
```

Remove a document using new signature:

```
arangosh> db.example.save({ a:  1 } );
{
```

```
   "error" : false,
   "_id" : "example/689529525",
   "_rev" : "689529525",
   "_key" : "689529525"
}
arangosh> db.example.remove("example/11265325374", {overwrite:
true, waitForSync: false})
false
```

Remove By Example

`collection.removeByExample(example)`

Removes all documents matching an example.

`collection.removeByExample(document, waitForSync)`

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

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

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

Examples

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

Replace By Example

`collection.replaceByExample(example, newValue)`

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

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

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

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

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

Examples

```
arangosh> db.example.replaceByExample({ Hello: "world" }, {Hello:
"mars"}, false, 5);
1
```

Update By Example

```
collection.updateByExample(example, newValue)
```

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

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

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

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

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

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

Examples

```
arangosh> db.example.updateByExample({ Hello: "world" }, { Hello:
"foo", Hello: "bar" }, false);
1
```

First

```
collection.first(count)
```

The *first* method returns the n first documents from the collection, in order of document insertion/update time.

If called with the *count* argument, the result is a list of up to *count* documents. If *count* is bigger than the number of documents in the collection, then the result will contain as many documents as there are in the collection. The result list is ordered, with the "oldest" documents being positioned at the beginning of the result list.

When called without an argument, the result is the first document from the collection. If the collection does not contain any documents, the result returned is *null*.

Note: this method is not supported in sharded collections with more than one shard.

Examples

```
arangosh> db.example.first(1);
[
  {
    "_id" : "example/639853237",
    "_rev" : "639853237",
    "_key" : "639853237",
    "Hello" : "world"
  }
]
```

```
arangosh> db.example.first();
{
  "_id" : "example/380723893",
  "_rev" : "380723893",
  "_key" : "380723893",
  "Hello" : "world"
}
```

Last

```
collection.last(count)
```

The *last* method returns the n last documents from the collection, in order of document insertion/update time.

If called with the *count* argument, the result is a list of up to *count* documents. If *count* is bigger than the number of documents in the collection, then the result will contain as many documents as there are in the collection. The result list is ordered, with the "latest" documents being positioned at the beginning of the result list.

When called without an argument, the result is the last document from the collection. If the collection does not contain any documents, the result returned is *null*.

Note: this method is not supported in sharded collections with more than one shard.

Examples

```
arangosh> db.example.last(2);
[
  {
    "_id" : "example/233464501",
    "_rev" : "233464501",
    "_key" : "233464501",
    "Foo" : "bar"
  },
  {
    "_id" : "example/233136821",
    "_rev" : "233136821",
    "_key" : "233136821",
    "Hello" : "world"
  }
]
```

```
arangosh> db.example.last(1);
[
```

```
    {
      "_id" : "example/402875061",
      "_rev" : "402875061",
      "_key" : "402875061",
      "Hello" : "world"
    }
]
```

Database Methods

Document

```
db._document(document)
```

This method finds a document given its identifier. It returns the document if the document exists. An error is throw if no document with the given identifier exists, or if the specified *_rev* value does not match the current revision of the document.

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

```
db._document(document-handle)
```

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

Examples

Returns the document:

```
arangosh> db._document("example/12345");
{
  "_id" : "example/12345",
  "_rev" : "746939061",
  "_key" : "12345"
}
```

Exists

```
db._exists(document)
```

This method determines whether a document exists given its identifier. Instead of returning the found document or an error, this method will return either *true* or *false*. It can thus be used for easy existence checks.

No error will be thrown if the sought document or collection does not exist. Still this method will throw an error if used improperly, e.g. when called with a non-document handle.

```
db._exists(document-handle)
```

As before, but instead of a document a document-handle can be passed.

Replace

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

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

If there is a conflict, i. e. if the revision of the *document* does not match the revision in the collection, then an error is thrown.

```
db._replace(document, data, true)
```

As before, but in case of a conflict, the conflict is ignored and the old document is overwritten.

```
db._replace(document, data, true, waitForSync)
```

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

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

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

Examples

Create and replace a document:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/276914869",
  "_rev" : "276914869",
  "_key" : "276914869"
}
arangosh> a2 = db._replace(a1, { a : 2 });
{
  "error" : false,
  "_id" : "example/276914869",
  "_rev" : "277111477",
  "_key" : "276914869"
}
arangosh> a3 = db._replace(a1, { a : 3 });
[ArangoError 1200: precondition failed]
```

Update

```
db._update(document, data, overwrite, keepNull, waitForSync)
```

Updates an existing *document*. The *document* must be a document in the current collection. This document is then patched with the *data* given as second argument. The optional *overwrite* parameter can be used to control the behavior in case of version conflicts (see below). The optional *keepNull* parameter can be used to modify the behavior when handling *null* values. Normally, *null* values are stored in the database. By setting the *keepNull* parameter to *false*, this behavior can be changed so that all attributes in *data* with *null* values will be removed from the target document.

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

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

If there is a conflict, i. e. if the revision of the *document* does not match the revision in the collection, then

an error is thrown.

```
db._update(document, data, true)
```

As before, but in case of a conflict, the conflict is ignored and the old document is overwritten.

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

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

Examples

Create and update a document:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/780493493",
  "_rev" : "780493493",
  "_key" : "780493493"
}
arangosh> a2 = db._update(a1, { b : 2 });
{
  "error" : false,
  "_id" : "example/780493493",
  "_rev" : "780821173",
  "_key" : "780493493"
}
arangosh> a3 = db._update(a1, { c : 3 });
[ArangoError 1200: precondition failed]
```

Remove

```
collection.remove(document)
```

Removes a document. If there is revision mismatch, then an error is thrown.

```
collection.remove(document, true)
```

Removes a document. If there is revision mismatch, then mismatch is ignored and document is deleted. The function returns *true* if the document existed and was deleted. It returns *false*, if the document was already deleted.

```
collection.remove(document, true, waitForSync)
```

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

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

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

Examples

Remove a document:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/638149301",
  "_rev" : "638149301",
  "_key" : "638149301"
}
arangosh> db.example.document(a1);
{
  "a" : 1,
  "_id" : "example/638149301",
  "_rev" : "638149301",
  "_key" : "638149301"
}
arangosh> db.example.remove(a1);
true
arangosh> db.example.document(a1);
[ArangoError 1202: document /_api/document/example/638149301 not
found]
```

Remove a document with a conflict:

```
arangosh> a1 = db.example.save({ a : 1 });
{
  "error" : false,
  "_id" : "example/122249909",
  "_rev" : "122249909",
  "_key" : "122249909"
}
arangosh> a2 = db.example.replace(a1, { a : 2 });
{
  "error" : false,
  "_id" : "example/122249909",
  "_rev" : "122446517",
  "_key" : "122249909"
}
arangosh> db.example.remove(a1);
[ArangoError 1200: precondition failed]
arangosh> db.example.remove(a1, true);
true
arangosh> db.example.document(a1);
[ArangoError 1202: document /_api/document/example/122249909 not
found]
```

Edges, Identifiers, Handles

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

Edges in ArangoDB are special documents. In addition to the internal attributes _key, _id and _rev, they have two attributes _from and _to, which contain document handles, namely the start-point and the end-point of the edge. The values of _from and _to are immutable once saved.

Edge collections are special collections that store edge documents. Edge documents are connection documents that reference other documents. The type of a collection must be specified when a collection is created and cannot be changed afterwards.

Working with Edges

Save

```
edge-collection.save(from, to, document)
```

Saves a new edge and returns the document-handle. *from* and *to* must be documents or document references.

```
edge-collection.save(from, to, document, waitForSync)
```

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

Examples

```
arangosh> v1 = db.vertex.save({ name : "vertex 1" });
{
  "error" : false,
  "_id" : "vertex/389571253",
  "_rev" : "389571253",
  "_key" : "389571253"
}
arangosh> v2 = db.vertex.save({ name : "vertex 2" });
{
  "error" : false,
  "_id" : "vertex/389767861",
  "_rev" : "389767861",
  "_key" : "389767861"
}
arangosh> e1 = db.relation.save(v1, v2, { label : "knows" });
{
  "error" : false,
  "_id" : "relation/390095541",
  "_rev" : "390095541",
  "_key" : "390095541"
}
arangosh> db._document(e1);
{
  "label" : "knows",
  "_id" : "relation/390095541",
  "_rev" : "390095541",
  "_key" : "390095541",
  "_from" : "vertex/389571253",
  "_to" : "vertex/389767861"
}
```

Edges

```
edge-collection.edges(vertex)
```

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

```
edge-collection.edges(vertices)
```

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

Examples

```
arangosh> db.relation.edges("vertex/1593622");
[ArangoError 1203: collection not found]
```

InEdges

```
edge-collection.inEdges(vertex)
```

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

```
edge-collection.inEdges(vertices)
```

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

Examples

```
arangosh> db.relation.inEdges("vertex/1528086");
[ArangoError 1203: collection not found]
arangosh> db.relation.inEdges("vertex/1593622");
[ArangoError 1203: collection not found]
```

OutEdges

```
edge-collection.outEdges(vertex)
```

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

```
edge-collection.outEdges(vertices)
```

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

Examples

```
arangosh> db.relation.inEdges("vertex/1528086");
[ArangoError 1203: collection not found]
arangosh> db.relation.inEdges("vertex/1593622");
[ArangoError 1203: collection not found]
```

Simple Queries

Simple queries can be used if the query condition is straight forward, i.e., a document reference, all documents, a query-by-example, or a simple geo query. In a simple query you can specify exactly one collection and one query criteria. In the following sections we describe the JavaScript shell interface for simple queries, which you can use within the ArangoDB shell and within actions and transactions. For other languages see the corresponding language API documentation.

If a query returns a cursor, then you can use *hasNext* and *next* to iterate over the result set or *toArray* to convert it to an array.

If the number of query results is expected to be big, it is possible to limit the amount of documents transferred between the server and the client to a specific value. This value is called *batchSize*. The *batchSize* can optionally be set before or when a simple query is executed. If the server has more documents than should be returned in a single batch, the server will set the *hasMore* attribute in the result. It will also return the id of the server-side cursor in the *id* attribute in the result. This id can be used with the cursor API to fetch any outstanding results from the server and dispose the server-side cursor afterwards.

The initial *batchSize* value can be set using the *setBatchSize* method that is available for each type of simple query, or when the simple query is executed using its *execute* method. If no *batchSize* value is specified, the server will pick a reasonable default value.

You can find a list of queries at [Collection Methods](#).

Geo Queries

The ArangoDB allows to select documents based on geographic coordinates. In order for this to work, a geo-spatial index must be defined. This index will use a very elaborate algorithm to lookup neighbors that is a magnitude faster than a simple R* index.

In general a geo coordinate is a pair of latitude and longitude. This can either be a list with two elements like *[-10, +30]* (latitude first, followed by longitude) or an object like `{lon: -10, lat: +30}`. In order to find all documents within a given radius around a coordinate use the *within* operator. In order to find all documents near a given document use the *near* operator.

It is possible to define more than one geo-spatial index per collection. In this case you must give a hint using the *geo* operator which of indexes should be used in a query.

Near

```
collection.near(latitude, longitude)
```

The returned list is sorted according to the distance, with the nearest document to the coordinate (*latitude, longitude*) coming first. If there are near documents of equal distance, documents are chosen randomly from this set until the limit is reached. It is possible to change the limit using the *limit* operator.

In order to use the *near* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* operator to select a particular index.

Note: *near* does not support negative skips. However, you can still use *limit* followed to *skip*.

```
collection.near(latitude, longitude).limit(limit)
```

Limits the result to limit documents instead of the default 100.

Note: Unlike with multiple explicit limits, limit} will raise the implicit default limit imposed by *within*.

```
collection.near(latitude, longitude).distance()
```

This will add an attribute *distance* to all documents returned, which contains the distance between the given point and the document in meter.

```
collection.near(latitude, longitude).distance(name)
```

This will add an attribute *name* to all documents returned, which contains the distance between the given point and the document in meter.

Examples

To get the nearst two locations:

```
arangosh> db.geo.near(0,0).limit(2).toArray();
TypeError: Cannot call method 'near' of undefined
```

If you need the distance as well, then you can use the *distance* operator:

```
arangosh> db.geo.near(0,0).distance().limit(2).toArray();
TypeError: Cannot call method 'near' of undefined
```

Within

```
collection.within(latitude, longitude, radius)
```

This will find all documents within a given radius around the coordinate (*latitude*, *longitude*). The returned list is sorted by distance, beginning with the nearest document.

In order to use the *within* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* operator to select a particular index.

```
collection.within(latitude, longitude, radius).distance()
```

This will add an attribute *_distance* to all documents returned, which contains the distance between the given point and the document in meter.

```
collection.within(latitude, longitude, radius).distance(name)
```

This will add an attribute *name* to all documents returned, which contains the distance between the given point and the document in meter.

Examples

To find all documents within a radius of 2000 km use:

```
arangosh> db.geo.within(0, 0, 2000 * 1000).distance().toArray();
TypeError: Cannot call method 'within' of undefined
```

Geo

```
collection.geo(location-attribute)
```

Looks up a geo index defined on attribute *location-attribute*.

Returns a geo index object if an index was found. The *near* or *within* operators can then be used to execute a geo-spatial query on this particular index.

This is useful for collections with multiple defined geo indexes.

```
collection.geo(location-attribute, true)
```

Looks up a geo index on a compound attribute *location-attribute*.

Returns a geo index object if an index was found. The *near* or *within* operators can then be used to execute a geo-spatial query on this particular index.

```
collection.geo(latitude-attribute, longitude-attribute)
```

Looks up a geo index defined on the two attributes *latitude-attribute* and *longitude-attribute*.

Returns a geo index object if an index was found. The *near* or *within* operators can then be used to execute a geo-spatial query on this particular index.

Examples

Assume you have a location stored as list in the attribute *home* and a destination stored in the attribute *work*. Then you can use the *geo* operator to select which geo-spatial attributes (and thus which index) to use in a near query.

@TINYEXAMPLE{simple-query-geo,use a specific index}

Fulltext queries

ArangoDB allows to run queries on text contained in document attributes. To use this, a fulltext index must be defined for the attribute of the collection that contains the text. Creating the index will parse the text in the specified attribute for all documents of the collection. Only documents will be indexed that contain a textual value in the indexed attribute. For such documents, the text value will be parsed, and the individual words will be inserted into the fulltext index.

When a fulltext index exists, it can be queried using a fulltext query.

Fulltext

```
collection.FULLTEXT(index-handle, query)
```

The *FULLTEXT* operator performs a fulltext search using the specified index and the specified *query*.

query must contain a comma-separated list of words to look for. Each word can optionally be prefixed with one of the following command literals:

- *prefix*: perform a prefix-search for the word following
- *substring*: perform substring-matching for the word following. This option is only supported for fulltext indexes that have been created with the *indexSubstrings* option
- *complete*: only match the complete following word (this is the default)

Examples

```
arangosh> db.emails.FULLTEXT("emails/1632537", "word");
TypeError: Cannot call method 'FULLTEXT' of undefined
```

Fulltext Syntax:

In the simplest form, a fulltext query contains just the sought word. If multiple search words are given in a query, they should be separated by commas. All search words will be combined with a logical AND by default, and only such documents will be returned that contain all search words. This default behavior can be changed by providing the extra control characters in the fulltext query, which are:

- +: logical AND (intersection)
- |: logical OR (union)
- -: negation (exclusion)

Examples:

- *"banana"*: searches for documents containing "banana"
- *"banana,apple"*: searches for documents containing both "banana" *AND* "apple"
- *"banana,|orange"*: searches for documents containing either "banana" *OR* "orange" *OR* both
- *"banana,-apple"*: searches for documents that contains "banana" but *NOT* "apple".

Logical operators are evaluated from left to right.

Each search word can optionally be prefixed with *complete*: or *prefix*:, with *complete*: being the default. This allows searching for complete words or for word prefixes. Suffix searches or any other forms are partial-word matching are currently not supported.

Examples:

- *"complete:banana"*: searches for documents containing the exact word "banana"
- *"prefix:head"*: searches for documents with words that start with prefix "head"
- *"prefix:head,banana"*: searches for documents contain words starting with prefix "head" and that

also contain the exact word "banana".

Complete match and prefix search options can be combined with the logical operators.

Please note that only words with a minimum length will get indexed. This minimum length can be defined when creating the fulltext index. For words tokenisation, the libicu text boundary analysis is used, which takes into account the default as defined at server startup (*--server.default-language* startup option). Generally, the word boundary analysis will filter out punctuation but will not do much more.

Especially no Word normalization, stemming, or similarity analysis will be performed when indexing or searching. If any of these features is required, it is suggested that the user does the text normalization on the client side, and provides for each document an extra attribute containing just a comma-separated list of normalized words. This attribute can then be indexed with a fulltext index, and the user can send fulltext queries for this index, with the fulltext queries also containing the stemmed or normalized versions of words as required by the user.

Pagination

If, for example, you display the result of a user search, then you are in general not interested in the completed result set, but only the first 10 or so documents. Or maybe the next 10 documents for the second page. In this case, you can the *skip* and *limit* operators. These operators work like LIMIT in MySQL.

skip used together with *limit* can be used to implement pagination. The *skip* operator skips over the first n documents. So, in order to create result pages with 10 result documents per page, you can use *skip(n 10).limit(10)** to access the 10 documents on the n.th page. This result should be sorted, so that the pagination works in a predicable way.

Limit

```
query.limit(number)
```

Limits a result to the first *number* documents. Specifying a limit of *0* returns no documents at all. If you do not need a limit, just do not add the limit operator. The limit must be non-negative.

In general the input to *limit* should be sorted. Otherwise it will be unclear which documents are used in the result set.

Examples

```
arangosh> db.five.all().toArray();
[
  {
    "_id" : "five/375612085",
    "_rev" : "375612085",
    "_key" : "375612085",
    "name" : "two"
  },
  {
    "_id" : "five/375808693",
    "_rev" : "375808693",
    "_key" : "375808693",
    "name" : "three"
  },
  {
    "_id" : "five/376201909",
    "_rev" : "376201909",
    "_key" : "376201909",
    "name" : "five"
  },
  {
    "_id" : "five/375415477",
    "_rev" : "375415477",
    "_key" : "375415477",
    "name" : "one"
  },
  {
    "_id" : "five/376005301",
    "_rev" : "376005301",
    "_key" : "376005301",
    "name" : "four"
  }
]
arangosh> db.five.all().limit(2).toArray();
```

```
[
  {
    "_id" : "five/375612085",
    "_rev" : "375612085",
    "_key" : "375612085",
    "name" : "two"
  },
  {
    "_id" : "five/375808693",
    "_rev" : "375808693",
    "_key" : "375808693",
    "name" : "three"
  }
]
```

Skip

```
query.skip(number)
```

Skips the first *number* documents. If *number* is positive, then skip the number of documents. If *number* is negative, then the total amount N of documents must be known and the results starts at position (N + *number*).

In general the input to *limit* should be sorted. Otherwise it will be unclear which documents are used in the result set.

Examples

```
arangosh> db.five.all().toArray();
[
  {
    "_id" : "five/142238389",
    "_rev" : "142238389",
    "_key" : "142238389",
    "name" : "four"
  },
  {
    "_id" : "five/141845173",
    "_rev" : "141845173",
    "_key" : "141845173",
    "name" : "two"
  },
  {
    "_id" : "five/142041781",
    "_rev" : "142041781",
    "_key" : "142041781",
    "name" : "three"
  },
  {
    "_id" : "five/141648565",
    "_rev" : "141648565",
    "_key" : "141648565",
    "name" : "one"
  },
  {
    "_id" : "five/142434997",
    "_rev" : "142434997",
    "_key" : "142434997",
```

```
    "name" : "five"
  }
]
arangosh> db.five.all().skip(3).toArray();
[
  {
    "_id" : "five/141648565",
    "_rev" : "141648565",
    "_key" : "141648565",
    "name" : "one"
  },
  {
    "_id" : "five/142434997",
    "_rev" : "142434997",
    "_key" : "142434997",
    "name" : "five"
  }
]
```

Sequential Access and Cursors

Has Next

```
cursor.hasNext()
```

The *hasNext* operator returns *true*, then the cursor still has documents. In this case the next document can be accessed using the *next* operator, which will advance the cursor.

Examples

```
arangosh> var a = db.five.all();
arangosh> while (a.hasNext()) print(a.next());
{
  "_id" : "five/514941621",
  "_rev" : "514941621",
  "_key" : "514941621",
  "name" : "four"
}
{
  "_id" : "five/514548405",
  "_rev" : "514548405",
  "_key" : "514548405",
  "name" : "two"
}
{
  "_id" : "five/515138229",
  "_rev" : "515138229",
  "_key" : "515138229",
  "name" : "five"
}
{
  "_id" : "five/514351797",
  "_rev" : "514351797",
  "_key" : "514351797",
  "name" : "one"
}
{
  "_id" : "five/514745013",
  "_rev" : "514745013",
  "_key" : "514745013",
  "name" : "three"
}
```

Next

```
cursor.next()
```

If the *hasNext* operator returns *true*, then the underlying cursor of the simple query still has documents. In this case the next document can be accessed using the *next* operator, which will advance the underlying cursor. If you use *next* on an exhausted cursor, then *undefined* is returned.

Examples

```
arangosh> db.five.all().next();
{
```

```
    "_id" : "five/234054325",
    "_rev" : "234054325",
    "_key" : "234054325",
    "name" : "one"
}
```

Set Batch size

```
cursor.setBatchSize(number)
```

Sets the batch size for queries. The batch size determines how many results are at most transferred from the server to the client in one chunk.

Get Batch size

```
cursor.getBatchSize()
```

Returns the batch size for queries. If the returned value is undefined, the server will determine a sensible batch size for any following requests.

Execute Query

```
query.execute(batchSize)
```

Executes a simple query. If the optional batchSize value is specified, the server will return at most batchSize values in one roundtrip. The batchSize cannot be adjusted after the query is first executed.

Note: There is no need to explicitly call the execute method if another means of fetching the query results is chosen. The following two approaches lead to the same result:

```
arangosh> result = db.users.all().toArray();
[
  {
    "_id" : "users/544105141",
    "_rev" : "544105141",
    "_key" : "544105141",
    "name" : "Gerhard"
  },
  {
    "_id" : "users/544367285",
    "_rev" : "544367285",
    "_key" : "544367285",
    "name" : "Helmut"
  },
  {
    "_id" : "users/544563893",
    "_rev" : "544563893",
    "_key" : "544563893",
    "name" : "Angela"
  }
]
arangosh> q = db.users.all(); q.execute(); result = [ ]; while
(q.hasNext()) { result.push(q.next()); }
SimpleQueryAll(users)
```

The following two alternatives both use a batchSize and return the same result:

```
arangosh> q = db.users.all(); q.setBatchSize(20); q.execute();
while (q.hasNext()) { print(q.next()); }
{
  "_id" : "users/510157493",
  "_rev" : "510157493",
  "_key" : "510157493",
  "name" : "Gerhard"
}
{
  "_id" : "users/510419637",
  "_rev" : "510419637",
  "_key" : "510419637",
  "name" : "Helmut"
}
{
  "_id" : "users/510616245",
  "_rev" : "510616245",
  "_key" : "510616245",
  "name" : "Angela"
}
SimpleQueryAll(users)
arangosh> q = db.users.all(); q.execute(20); while (q.hasNext()) {
print(q.next()); }
{
  "_id" : "users/510157493",
  "_rev" : "510157493",
  "_key" : "510157493",
  "name" : "Gerhard"
}
{
  "_id" : "users/510419637",
  "_rev" : "510419637",
  "_key" : "510419637",
  "name" : "Helmut"
}
{
  "_id" : "users/510616245",
  "_rev" : "510616245",
  "_key" : "510616245",
  "name" : "Angela"
}
SimpleQueryAll(users)
```

Dispose

`cursor.dispose()`

If you are no longer interested in any further results, you should call *dispose* in order to free any resources associated with the cursor. After calling *dispose* you can no longer access the cursor.

Count

`cursor.count()`

The *count* operator counts the number of document in the result set and returns that number. The *count* operator ignores any limits and returns the total number of documents found.

Note: Not all simple queries support counting. In this case *null* is returned.

```
cursor.count(true)
```

If the result set was limited by the *limit* operator or documents were skiped using the *skip* operator, the *count* operator with argument *true* will use the number of elements in the final result set - after applying *limit* and *skip*.

Note: Not all simple queries support counting. In this case *null* is returned.

Examples

Ignore any limit:

```
arangosh> db.five.all().limit(2).count();
null
```

Counting any limit or skip:

```
arangosh> db.five.all().limit(2).count(true);
2
```

Modification Queries

ArangoDB also allows removing, replacing, and updating documents based on an example document. Every document in the collection will be compared against the specified example document and be deleted/replaced/ updated if all attributes match.

These method should be used with caution as they are intended to remove or modify lots of documents in a collection.

All methods can optionally be restricted to a specific number of operations. However, if a limit is specific but is less than the number of matches, it will be undefined which of the matching documents will get removed/modified. Remove by Example, Replace by Example and Update by Example are described with examples in the subchapter Collection Methods.

Transactions

Starting with version 1.3, ArangoDB provides support for user-definable transactions.

Transactions in ArangoDB are atomic, consistent, isolated, and durable (*ACID*).

These *ACID* properties provide the following guarantees:

- The *atomicity* principle makes transactions either complete in their entirety or have no effect at all.
- The *consistency* principle ensures that no constraints or other invariants will be violated during or after any transaction.
- The *isolation* property will hide the modifications of a transaction from other transactions until the transaction commits.
- Finally, the *durability* proposition makes sure that operations from transactions that have committed will be made persistent. The amount of transaction durability is configurable in ArangoDB, as is the durability on collection level.

Transaction invocation

ArangoDB transactions are different from transactions in SQL.

In SQL, transactions are started with explicit *BEGIN* or *START TRANSACTION* command. Following any series of data retrieval or modification operations, an SQL transaction is finished with a *COMMIT* command, or rolled back with a *ROLLBACK* command. There may be client/server communication between the start and the commit/rollback of an SQL transaction.

In ArangoDB, a transaction is always a server-side operation, and is executed on the server in one go, without any client interaction. All operations to be executed inside a transaction need to be known by the server when the transaction is started.

There are no individual *BEGIN*, *COMMIT* or *ROLLBACK* transaction commands in ArangoDB. Instead, a transaction in ArangoDB is started by providing a description of the transaction to the *db._executeTransaction* Javascript function:

```
db._executeTransaction(description);
```

This function will then automatically start a transaction, execute all required data retrieval and/or modification operations, and at the end automatically commit the transaction. If an error occurs during transaction execution, the transaction is automatically aborted, and all changes are rolled back.

Declaration of collections

All collections which are to participate in a transaction need to be declared beforehand. This is a necessity to ensure proper locking and isolation.

Collections can be used in a transaction in write mode or in read-only mode.

If any data modification operations are to be executed, the collection must be declared for use in write mode. The write mode allows modifying and reading data from the collection during the transaction (i.e. the write mode includes the read mode).

Contrary, using a collection in read-only mode will only allow performing read operations on a collection. Any attempt to write into a collection used in read-only mode will make the transaction fail.

Collections for a transaction are declared by providing them in the *collections* attribute of the object passed to the *_executeTransaction* function. The *collections* attribute has the sub-attributes *read* and *write*:

```
db._executeTransaction({
  collections: {
    write: [ "users", "logins" ],
    read: [ "recommendations" ]
  },
  ...
});
```

read and *write* are optional attributes, and only need to be specified if the operations inside the transactions demand for it.

The contents of *read* or *write* can each be lists with collection names or a single collection name (as a string):

```
db._executeTransaction({
  collections: {
```

```
        write: "users",
        read: "recommendations"
    },
    ...
});
```

Note that it is currently optional to specify collections for read-only access. Even without specifying them, it is still possible to read from such collections from within a transaction, but with relaxed isolation. Please refer to [Transactions Locking](#) for more details.

!SUBSECTION Declaration of data modification and retrieval operations

All data modification and retrieval operations that are to be executed inside the transaction need to be specified in a Javascript function, using the *action* attribute:

```
db._executeTransaction({
    collections: {
        write: "users"
    },
    action: function () {
        // all operations go here
    }
});
```

Any valid Javascript code is allowed inside *action* but the code may only access the collections declared in *collections*. *action* may be a Javascript function as shown above, or a string representation of a Javascript function:

```
db._executeTransaction({
    collections: {
        write: "users"
    },
    action: "function () { doSomething(); }"
});
```

Please note that any operations specified in *action* will be executed on the server, in a separate scope. Variables will be bound late. Accessing any Javascript variables defined on the client-side or in some other server context from inside a transaction may not work. Instead, any variables used inside *action* should be defined inside *action* itself:

```
db._executeTransaction({
    collections: {
        write: "users"
    },
    action: function () {
        var db = require(...).db;
        db.users.save({ ... });
    }
});
```

When the code inside the *action* attribute is executed, the transaction is already started and all required locks have been acquired. When the code inside the *action* attribute finishes, the transaction will automatically commit. There is no explicit commit command.

To make a transaction abort and roll back all changes, an exception needs to be thrown and not caught

inside the transaction:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: function () {
    var db = require("internal").db;
    db.users.save({ _key: "hello" });

    // will abort and roll back the transaction
    throw "doh!";
  }
});
```

There is no explicit abort or roll back command.

As mentioned earlier, a transaction will commit automatically when the end of the *action* function is reached and no exception has been thrown. In this case, the user can return any legal Javascript value from the function:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  action: function () {
    var db = require("internal").db;
    db.users.save({ _key: "hello" });

    // will commit the transaction and return the value "hello"
    return "hello";
  }
});
```

!SUBSECTION Examples

The first example will write 3 documents into a collection named *c1*. The *c1* collection needs to be declared in the *write* attribute of the *collections* attribute passed to the *executeTransaction* function.

The *action* attribute contains the actual transaction code to be executed. This code contains all data modification operations (3 in this example).

```
// setup
db._create("c1");

db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("internal").db;
    db.c1.save({ _key: "key1" });
    db.c1.save({ _key: "key2" });
    db.c1.save({ _key: "key3" });
  }
});
    db.c1.count(); // 3
```

Aborting the transaction by throwing an exception in the *action* function will revert all changes, so as if the transaction never happened:

```
// setup
db._create("c1");

db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("internal").db;
    db.c1.save({ _key: "key1" });
    db.c1.count(); // 1

    db.c1.save({ _key: "key2" });
    db.c1.count(); // 2

    throw "doh!";
  }
});

db.c1.count(); // 0
```

The automatic rollback is also executed when an internal exception is thrown at some point during transaction execution:

```
// setup
db._create("c1");

db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("internal").db;
    db.c1.save({ _key: "key1" });

    // will throw duplicate a key error, not explicitly requested
by the user
    db.c1.save({ _key: "key1" });

    // we'll never get here...
  }
});

db.c1.count(); // 0
```

As required by the *consistency* principle, aborting or rolling back a transaction will also restore secondary indexes to the state at transaction start. The following example using a cap constraint should illustrate that:

```
// setup
db._create("c1");

// limit the number of documents to 3
```

```
db.c1.ensureCapConstraint(3);

// insert 3 documents
db.c1.save({ _key: "key1" });
db.c1.save({ _key: "key2" });
db.c1.save({ _key: "key3" });

// this will push out key1
// we now have these keys: [ "key1", "key2", "key3" ]
db.c1.save({ _key: "key4" });


db._executeTransaction({
  collections: {
    write: [ "c1" ]
  },
  action: function () {
    var db = require("internal").db;
    // this will push out key2. we now have keys [ "key3", "key4",
"key5" ]
    db.c1.save({ _key: "key5" });

    // will abort the transaction
    throw "doh!"
  }
});

// we now have these keys back: [ "key2", "key3", "key4" ]
```

Cross-collection transactions

There's also the possibility to run a transaction across multiple collections. In this case, multiple collections need to be declared in the *collections* attribute, e.g.:

```
// setup
db._create("c1");
db._create("c2");

db._executeTransaction({
  collections: {
    write: [ "c1", "c2" ]
  },
  action: function () {
    var db = require("internal").db;
    db.c1.save({ _key: "key1" });
    db.c2.save({ _key: "key2" });
  }
});

db.c1.count(); // 1
db.c2.count(); // 1
```

Again, throwing an exception from inside the *action* function will make the transaction abort and roll back all changes in all collections:

```
// setup
db._create("c1");
```

```
db._create("c2");

db._executeTransaction({
  collections: {
    write: [ "c1", "c2" ]
  },
  action: function () {
    var db = require("internal").db;
    for (var i = 0; i < 100; ++i) {
      db.c1.save({ _key: "key" + i });
      db.c2.save({ _key: "key" + i });
    }

    db.c1.count(); // 100
    db.c2.count(); // 100

    // abort
    throw "doh!"
  }
});

db.c1.count(); // 0
db.c2.count(); // 0
```

Passing parameters to transactions

Arbitrary parameters can be passed to transactions by setting the *params* attribute when declaring the transaction. This feature is handy to re-use the same transaction code for multiple calls but with different parameters.

A basic example:

```
db._executeTransaction({
  collections: { },
  action: "function (params) { return params[1]; }",
  params: [ 1, 2, 3 ]
});
```

The above example will return *1*.

Some example that uses collections:

```
db._executeTransaction({
  collections: {
    write: "users",
    read: [ "c1", "c2" ]
  },
  action: "function (params) { var db = require('internal').db; var
doc = db.c1.document(params['c1Key']); db.users.save(doc); doc =
db.c2.document(params['c2Key']); db.users.save(doc);}",
  params: {
    c1Key: "foo",
    c2Key: "bar"
  }
});
```

Disallowed operations

Some operations are not allowed inside ArangoDB transactions:

- creation and deletion of collections (*db._create()*, *db._drop()*, *db._rename()*)
- creation and deletion of indexes (*db.ensure...Index()*, *db.dropIndex()*)

If an attempt is made to carry out any of these operations during a transaction, ArangoDB will abort the transaction with error code *1653 (disallowed operation inside transaction)*.

Locking and Isolation

All collections specified in the *collections* attribute are locked in the requested mode (read or write) at transaction start. Locking of multiple collections is performed in alphabetical order. When a transaction commits or rolls back, all locks are released in reverse order. The locking order is deterministic to avoid deadlocks.

While locks are held, modifications by other transactions to the collections participating in the transaction are prevented. A transaction will thus see a consistent view of the participating collections' data.

Additionally, a transaction will not be interrupted or interleaved with any other ongoing operations on the same collection. This means each transaction will run in isolation. A transaction should never see uncommitted or rolled back modifications by other transactions. Additionally, reads inside a transaction are repeatable.

Note that the above is true only for all collections that are declared in the *collections* attribute of the transaction.

There might be situations when declaring all collections a priori is not possible, for example, because further collections are determined by a dynamic AQL query inside the transaction. In this case, it would be impossible to know beforehand which collection to lock, and thus it is legal to not declare collections that will be accessed in the transaction in read-only mode. Accessing a non-declared collection in read-only mode during a transaction will add the collection to the transaction lazily, and fetch data from the collection as usual. However, as the collection ie added lazily, there is no isolation from other concurrent operations or transactions. Reads from such collections are potentially non-repeatable.

@EXAMPLES

```
db._executeTransaction({
  collections: {
    read: "users"
  },
  action: function () {
    // execute an AQL query that traverses a graph starting at a
"users" vertex.
    // it is yet unknown into which other collections the query
will traverse
    db._createStatement({
      query: "FOR t IN TRAVERSAL(users, connections, "users/1234",
"any", { }) RETURN t"
    }).execute().toArray().forEach(function (d) {
      // ...
    });
  }
});
```

This automatic lazy addition of collections to a transaction also introduces the possibility of deadlocks. Deadlocks may occur if there are concurrent transactions that try to acquire locks on the same collections lazily.

To recover from a deadlock state, ArangoDB will give up waiting for a collection after a configurable amount of time. The wait time can be specified per transaction using the optional*lockTimeout*attribute. If no value is specified, some default value will be applied.

If ArangoDB was waited at least *lockTimeout* seconds during lock acquisition, it will give up and rollback the transaction. Note that the *lockTimeout* is used per lock acquisition in a transaction, and not just once per transaction. There will be at least as many lock acquisition attempts as there are collections used in the transaction. The total lock wait time may thus be much higher than the value of *lockTimeout*.

To avoid both deadlocks and non-repeatable reads, all collections used in a transaction should always be specified if known in advance.

Durability

Transactions are executed in main memory first until there is either a rollback or a commit. On rollback, no data will be written to disk, but the operations from the transaction will be reversed in memory.

On commit, all modifications done in the transaction will be written to the collection datafiles. These writes will be synchronized to disk if any of the modified collections has the *waitForSync* property set to *true*, or if any individual operation in the transaction was executed with the *waitForSync* attribute. Additionally, transactions that modify data in more than one collection are automatically synchronized to disk. This synchronization is done to not only ensure durability, but to also ensure consistency in case of a server crash.

That means if you only modify data in a single collection, and that collection has its *waitForSync* property set to *false*, the whole transaction will not be synchronized to disk instantly, but with a small delay.

There is thus the potential risk of losing data between the commit of the transaction and the actual (delayed) disk synchronization. This is the same as writing into collections that have the *waitForSync* property set to *false* outside of a transaction. In case of a crash with *waitForSync* set to false, the operations performed in the transaction will either be visible completely or not at all, depending on whether the delayed synchronization had kicked in or not.

To ensure durability of transactions on a collection that have the *waitForSync* property set to *false*, you can set the *waitForSync* attribute of the object that is passed to *executeTransaction*. This will force a synchronization of the transaction to disk even for collections that have *waitForSync* set to *false*:

```
db._executeTransaction({
  collections: {
    write: "users"
  },
  waitForSync: true,
  action: function () { ... }
});
```

An alternative is to perform an operation with an explicit *sync* request in a transaction, e.g.

```
db.users.save({ _key: "1234" }, true);
```

In this case, the *true* value will make the whole transaction be synchronized to disk at the commit.

In any case, ArangoDB will give users the choice of whether or not they want full durability for single collection transactions. Using the delayed synchronization (i.e. *waitForSync* with a value of *false*) will potentially increase throughput and performance of transactions, but will introduce the risk of losing the last committed transactions in the case of a crash.

In contrast, transactions that modify data in more than one collection are automatically synchronized to disk. This comes at the cost of several disk sync For a multi-collection transaction, the call to the *_executeTransaction* function will only return only after the data of all modified collections has been synchronized to disk and the transaction has been made fully durable. This not only reduces the risk of losing data in case of a crash but also ensures consistency after a restart.

In case of a server crash, any multi-collection transactions that were not yet committed or in preparation to be committed will be rolled back on server restart.

For multi-collection transactions, there will be at least one disk sync operation per modified collection. Multi-collection transactions thus have a potentially higher cost than single collection transactions. There is no configuration to turn off disk synchronization for multi-collection transactions in ArangoDB. The

disk sync speed of the system will thus be the most important factor for the performance of multi-collection transactions.

Limitations

Transactions in ArangoDB have been designed with particular use cases in mind. They will be mainly useful for short and small data retrieval and/or modification operations.

The implementation is not optimized for very long-running or very voluminous operations, and may not be usable for these cases.

A major limitation is that a transaction must entirely fit into main memory. This includes all data that is created, updated, or deleted during a transaction, plus management overhead.

Transactions should thus be kept as small as possible, and big operations should be split into multiple smaller transactions if they are too big to fit into one transaction.

Additionally, transactions in ArangoDB cannot be nested, i.e. a transaction must not call any other transaction. If an attempt is made to call a transaction from inside a running transaction, the server will throw error *1651 (nested transactions detected)*.

It is also disallowed to execute user transaction on some of ArangoDB's own system collections. This shouldn't be a problem for regular usage as system collections will not contain user data and there is no need to access them from within a user transaction.

Finally, all collections that may be modified during a transaction must be declared beforehand, i.e. using the *collections* attribute of the object passed to the *_executeTransaction* function. If any attempt is made to carry out a data modification operation on a collection that was not declared in the *collections* attribute, the transaction will be aborted and ArangoDB will throw error *1652 unregistered collection used in transaction*. It is legal to not declare read-only collections, but this should be avoided if possible to reduce the probability of deadlocks and non-repeatable reads.

Please refer to Locking and Isolation for more details.

Introduction

The ArangoDB query language (AQL) can be used to retrieve data that is stored in ArangoDB. The general workflow when executing a query is as follows:

- A client application ships an AQL query to the ArangoDB server. The query text contains everything ArangoDB needs to compile the result set
- ArangoDB will parse the query, execute it and compile the results. If the query is invalid or cannot be executed, the server will return an error that the client can process and react to. If the query can be executed successfully, the server will return the query results to the client

AQL is mainly a declarative language, meaning that in a query it is expressed what result should be achieved and not how. AQL aims to be human- readable and therefore uses keywords from the English language. Another design goal of AQL was client independency, meaning that the language and syntax are the same for all clients, no matter what programming language the clients might use. Further design goals of AQL were to support complex query patterns, and to support the different data models ArangoDB offers.

In its purpose, AQL is similar to the Structured Query Language (SQL), but the two languages have major syntactic differences. Furthermore, to avoid any confusion between the two languages, the keywords in AQL have been chosen to be different from the keywords used in SQL.

AQL currently supports reading data only. That means you can use the language to issue read-requests on your database, but modifying data via AQL is currently not supported.

For some example queries, please refer to the page AQL Examples.

How to invoke AQL

You can run AQL queries from your application via the HTTP REST API. The full API description is available at Http Interface for AQL Query Cursor.

You can also run AQL queries from arangosh. To do so, first create an ArangoStatement object as follows:

```
stmt = db._createStatement( { "query": "FOR i IN [ 1, 2 ] RETURN i
* 2" } );
[object ArangoQueryCursor]
```

To execute the query, use the *execute* method:

```
arangosh> c = stmt.execute();
[object ArangoQueryCursor]
```

This has executed the query. The query results are available in a cursor now. The cursor can return all its results at once using the *toArray* method. This is a short-cut that you can use if you want to access the full result set without iterating over it yourself.

```
arangosh> c.toArray();
[2, 4]
```

Cursors can also be used to iterate over the result set document-by-document. To do so, use the *hasNext* and *next* methods of the cursor:

```
arangosh> while (c.hasNext()) {
require("internal").print(c.next()); }
2
4
```

Please note that you can iterate over the results of a cursor only once, and that the cursor will be empty when you have fully iterated over it. To iterate over the results again, the query needs to be re-executed.

Additionally, the iteration can be done in a forward-only fashion. There is no backwards iteration or random access to elements in a cursor.

To execute an AQL query using bind parameters, you need to create a statement first and then bind the parameters to it before execution:

```
arangosh> stmt = db._createStatement( { "query": "FOR i IN [ @one,
@two ] RETURN i * 2" } );
[object ArangoStatement]
arangosh> stmt.bind("one", 1);
arangosh> stmt.bind("two", 2);
arangosh> c = stmt.execute();
[object ArangoQueryCursor]
```

The cursor results can then be dumped or iterated over as usual, e.g.:

```
arangosh> c.toArray();
[2, 4]
```

or

```
arangosh> while (c.hasNext()) {
require("internal").print(c.next()); }
2
4
```

Please note that bind variables can also be passed into the *_createStatement* method directly, making it a bit more convenient:

```
arangosh> stmt = db._createStatement( {
   "query": "FOR i IN [ @one, @two ] RETURN i * 2",
   "bindVars": {
     "one": 1,
     "two": 2
   }
} );
```

Cursors also optionally provide the total number of results. By default, they do not. To make the server return the total number of results, you may set the *count* attribute to *true* when creating a statement:

```
arangosh> stmt = db._createStatement( { "query": "FOR i IN [ 1, 2,
3, 4 ] RETURN i", "count": true } );
```

After executing this query, you can use the *count* method of the cursor to get the number of total results from the result set:

```
arangosh> c = stmt.execute();
[object ArangoQueryCursor]
arangosh> c.count();
4
```

Please note that the *count* method returns nothing if you did not specify the *count* attribute when creating the query.

This is intentional so that the server may apply optimizations when executing the query and construct the result set incrementally. Incremental creating of the result sets would not be possible if the total number of results needs to be shipped to the client anyway. Therefore, the client has the choice to specify *count* and retrieve the total number of results for a query (and disable potential incremental result set creation on the server), or to not retrieve the total number of results and allow the server to apply optimizations.

Please note that at the moment the server will always create the full result set for each query so specifying or omitting the *count* attribute currently does not have any impact on query execution. This might change in the future. Future versions of ArangoDB might create result sets incrementally on the server-side and might be able to apply optimizations if a result set is not fully fetched by a client.

AQL Examples

This page contains some examples how to write queries in AQL. For better understandability the query results are also included directly below each query.

Simple queries

Following is a query that returns a string value. The result string is contained in a list because the result of every valid query is a list:

```
RETURN "this will be returned"
[
  "this will be returned"
]
```

Here is a query that creates the cross products of two lists and runs a projection on it, using a few of AQL's built-in functions:

```
FOR year in [ 2011, 2012, 2013 ]
  FOR quarter IN [ 1, 2, 3, 4 ]
    RETURN {
      "y" : "year",
      "q" : quarter,
      "nice" : CONCAT(TO_STRING(quarter), "/", TO_STRING(year))
    }
[
  { "y" : "year", "q" : 1, "nice" : "1/2011" },
  { "y" : "year", "q" : 2, "nice" : "2/2011" },
  { "y" : "year", "q" : 3, "nice" : "3/2011" },
  { "y" : "year", "q" : 4, "nice" : "4/2011" },
  { "y" : "year", "q" : 1, "nice" : "1/2012" },
  { "y" : "year", "q" : 2, "nice" : "2/2012" },
  { "y" : "year", "q" : 3, "nice" : "3/2012" },
  { "y" : "year", "q" : 4, "nice" : "4/2012" },
  { "y" : "year", "q" : 1, "nice" : "1/2013" },
  { "y" : "year", "q" : 2, "nice" : "2/2013" },
  { "y" : "year", "q" : 3, "nice" : "3/2013" },
  { "y" : "year", "q" : 4, "nice" : "4/2013" }
]
```

Collection-based queries

Normally you would want to run queries on data stored in collections. This section will provide several examples for that.

Example data

Some of the following example queries are executed on a collection *users* with the following initial data:

```
[
   { "id" : 100, "name" : "John", "age" : 37, "active" : true,
"gender" : "m" },
   { "id" : 101, "name" : "Fred", "age" : 36, "active" : true,
"gender" : "m" },
   { "id" : 102, "name" : "Jacob", "age" : 35, "active" : false,
"gender" : "m" },
   { "id" : 103, "name" : "Ethan", "age" : 34, "active" : false,
"gender" : "m" },
   { "id" : 104, "name" : "Michael", "age" : 33, "active" : true,
"gender" : "m" },
   { "id" : 105, "name" : "Alexander", "age" : 32, "active" : true,
"gender" : "m" },
   { "id" : 106, "name" : "Daniel", "age" : 31, "active" : true,
"gender" : "m" },
   { "id" : 107, "name" : "Anthony", "age" : 30, "active" : true,
"gender" : "m" },
   { "id" : 108, "name" : "Jim", "age" : 29, "active" : true,
"gender" : "m" },
   { "id" : 109, "name" : "Diego", "age" : 28, "active" : true,
"gender" : "m" },
   { "id" : 200, "name" : "Sophia", "age" : 37, "active" : true,
"gender" : "f" },
   { "id" : 201, "name" : "Emma", "age" : 36, "active" : true,
"gender" : "f" },
   { "id" : 202, "name" : "Olivia", "age" : 35, "active" : false,
"gender" : "f" },
   { "id" : 203, "name" : "Madison", "age" : 34, "active" : true,
"gender": "f" },
   { "id" : 204, "name" : "Chloe", "age" : 33, "active" : true,
"gender" : "f" },
   { "id" : 205, "name" : "Eva", "age" : 32, "active" : false,
"gender" : "f" },
   { "id" : 206, "name" : "Abigail", "age" : 31, "active" : true,
"gender" : "f" },
   { "id" : 207, "name" : "Isabella", "age" : 30, "active" : true,
"gender" : "f" },
   { "id" : 208, "name" : "Mary", "age" : 29, "active" : true,
"gender" : "f" },
   { "id" : 209, "name" : "Mariah", "age" : 28, "active" : true,
"gender" : "f" }
]
```

For some of the examples, we'll also use a collection *relations* to store relationships between users. The example data for *relations* are as follows:

```
[
```

```
      { "from" : 209,  "to" : 205,  "type" : "friend" },
      { "from" : 206,  "to" : 108,  "type" : "friend" },
      { "from" : 202,  "to" : 204,  "type" : "friend" },
      { "from" : 200,  "to" : 100,  "type" : "friend" },
      { "from" : 205,  "to" : 101,  "type" : "friend" },
      { "from" : 209,  "to" : 203,  "type" : "friend" },
      { "from" : 200,  "to" : 203,  "type" : "friend" },
      { "from" : 100,  "to" : 208,  "type" : "friend" },
      { "from" : 101,  "to" : 209,  "type" : "friend" },
      { "from" : 206,  "to" : 102,  "type" : "friend" },
      { "from" : 104,  "to" : 100,  "type" : "friend" },
      { "from" : 104,  "to" : 108,  "type" : "friend" },
      { "from" : 108,  "to" : 209,  "type" : "friend" },
      { "from" : 206,  "to" : 106,  "type" : "friend" },
      { "from" : 204,  "to" : 105,  "type" : "friend" },
      { "from" : 208,  "to" : 207,  "type" : "friend" },
      { "from" : 102,  "to" : 108,  "type" : "friend" },
      { "from" : 207,  "to" : 203,  "type" : "friend" },
      { "from" : 203,  "to" : 106,  "type" : "friend" },
      { "from" : 202,  "to" : 108,  "type" : "friend" },
      { "from" : 201,  "to" : 203,  "type" : "friend" },
      { "from" : 105,  "to" : 100,  "type" : "friend" },
      { "from" : 100,  "to" : 109,  "type" : "friend" },
      { "from" : 207,  "to" : 109,  "type" : "friend" },
      { "from" : 103,  "to" : 203,  "type" : "friend" },
      { "from" : 208,  "to" : 104,  "type" : "friend" },
      { "from" : 105,  "to" : 104,  "type" : "friend" },
      { "from" : 103,  "to" : 208,  "type" : "friend" },
      { "from" : 203,  "to" : 107,  "type" : "boyfriend" },
      { "from" : 107,  "to" : 203,  "type" : "girlfriend" },
      { "from" : 208,  "to" : 109,  "type" : "boyfriend" },
      { "from" : 109,  "to" : 208,  "type" : "girlfriend" },
      { "from" : 106,  "to" : 205,  "type" : "girlfriend" },
      { "from" : 205,  "to" : 106,  "type" : "boyfriend" },
      { "from" : 103,  "to" : 209,  "type" : "girlfriend" },
      { "from" : 209,  "to" : 103,  "type" : "boyfriend" },
      { "from" : 201,  "to" : 102,  "type" : "boyfriend" },
      { "from" : 102,  "to" : 201,  "type" : "girlfriend" },
      { "from" : 206,  "to" : 100,  "type" : "boyfriend" },
      { "from" : 100,  "to" : 206,  "type" : "girlfriend" }
]
```

Things to consider when running queries on collections

Note that all documents created in the two collections will automatically get the following server-generated attributes:

- _id: An unique id, consisting of collection name and a server-side sequence value
- _key: The server sequence value
- _rev: The document's revision id

Whenever you run queries on the documents in the two collections, don't be surprised if these additional attributes are returned as well.

Please also note that with real-world data, you might want to create additional indexes on the data (left out here for brevity). Adding indexes on attributes that are used in *FILTER* statements may considerably speed up queries. Furthermore, instead of using attributes such as *id*, *from* and *to*, you might want to use the built-in _id, _from and to attributes. Finally, edge collections provide a nice way of establishing references / links between documents. These features have been left out here for brevity as well.

Projections and Filters

Returning unaltered documents

To return three complete documents from collection *users*, the following query can be used:

```
FOR u IN users
  LIMIT 0, 3
  RETURN u

[
  {
    "_id" : "users/229886047207520",
    "_rev" : "229886047207520",
    "_key" : "229886047207520",
    "active" : true,
    "id" : 206,
    "age" : 31,
    "gender" : "f",
    "name" : "Abigail"
  },
  {
    "_id" : "users/229886045175904",
    "_rev" : "229886045175904",
    "_key" : "229886045175904",
    "active" : true,
    "id" : 101,
    "age" : 36,
    "name" : "Fred",
    "gender" : "m"
  },
  {
    "_id" : "users/229886047469664",
    "_rev" : "229886047469664",
    "_key" : "229886047469664",
    "active" : true,
    "id" : 208,
    "age" : 29,
    "name" : "Mary",
    "gender" : "f"
  }
]
```

Note that there is a *LIMIT* clause but no *SORT* clause. In this case it is not guaranteed which of the user documents are returned. Effectively the document return order is unspecified if no *SORT* clause is used, and you should not rely on the order in such queries.

Projections

To return a projection from the collection *users* use a modified *RETURN* instruction:

```
FOR u IN users
  LIMIT 0, 3
  RETURN {
    "user" : {
      "isActive" : u.active ? "yes" : "no",
```

```
        "name" : u.name
      }
    }

 [
   {
     "user" : {
       "isActive" : "yes",
       "name" : "John"
     }
   },
   {
     "user" : {
       "isActive" : "yes",
       "name" : "Anthony"
     }
   },
   {
     "user" : {
       "isActive" : "yes",
       "name" : "Fred"
     }
   }
 ]
```

Filters

To return a filtered projection from collection *users*, you can use the *FILTER* keyword. Additionally, a *SORT* clause is used to have the result returned in a specific order:

```
FOR u IN users
   FILTER u.active == true && u.age >= 30
   SORT u.age DESC
   LIMIT 0, 5
   RETURN {
     "age" : u.age,
     "name" : u.name
   }

 [
   {
     "age" : 37,
       "name" : "Sophia"
   },
   {
     "age" : 37,
     "name" : "John"
   },
   {
     "age" : 36,
     "name" : "Emma"
   },
   {
     "age" : 36,
     "name" : "Fred"
   },
   {
     "age" : 34,
```

```
      "name" : "Madison"
  }
]
```

Joins

So far we have only dealt with one collection (*users*) at a time. We also have a collection *relations* that stores relationships between users. We will now use this extra collection to create a result from two collections.

First of all, we'll query a few users together with their friends' ids. For that, we'll use all *relations* that have a value of *friend* in their *type* attribute. Relationships are established by using the *from* and *to* attributes in the *relations* collection, which point to the *id* values in the *users* collection.

Join tuples

We'll start with a SQL-ish result set and return each tuple (user name, friend id) separately. The AQL query to generate such result is:

```
FOR u IN users
  FILTER u.active == true
  LIMIT 0, 4
  FOR f IN relations
    FILTER f.type == "friend" && f.from == u.id
    RETURN {
      "user" : u.name,
      "friendId" : f.to
    }

[
  {
    "user" : "Abigail",
    "friendId" : 108
  },
  {
    "user" : "Abigail",
    "friendId" : 102
  },
  {
    "user" : "Abigail",
    "friendId" : 106
  },
  {
    "user" : "Fred",
    "friendId" : 209
  },
  {
    "user" : "Mary",
    "friendId" : 207
  },
  {
    "user" : "Mary",
    "friendId" : 104
  },
  {
    "user" : "Mariah",
    "friendId" : 203
  },
  {
    "user" : "Mariah",
    "friendId" : 205
  }
```

```
]
```

Horizontal lists

Note that in the above result, a user might be returned multiple times. This is the SQL way of returning data. If this is not desired, the friends' ids of each user can be returned in a horizontal list. This will return each user at most once.

The AQL query for doing so is:

```
FOR u IN users
  FILTER u.active == true LIMIT 0, 4
  RETURN {
    "user" : u.name,
    "friendIds" : (
      FOR f IN relations
        FILTER f.from == u.id && f.type == "friend"
        RETURN f.to
    )
  }

[
  {
    "user" : "Abigail",
    "friendIds" : [
      108,
      102,
      106
    ]
  },
  {
    "user" : "Fred",
    "friendIds" : [
      209
    ]
  },
  {
    "user" : "Mary",
    "friendIds" : [
      207,
      104
    ]
  },
  {
    "user" : "Mariah",
    "friendIds" : [
      203,
      205
    ]
  }
]
```

In this query we are still iterating over the users in the *users* collection and for each matching user we are executing a sub-query to create the matching list of related users.

Self joins

To not only return friend ids but also the names of friends, we could "join" the *users* collection once more (something like a "self join"):

```
FOR u IN users
  FILTER u.active == true
  LIMIT 0, 4
  RETURN {
    "user" : u.name,
    "friendIds" : (
      FOR f IN relations
        FILTER f.from == u.id && f.type == "friend"
        FOR u2 IN users
          FILTER f.to == u2.id
          RETURN u2.name
    )
  }

[
  {
    "user" : "Abigail",
    "friendIds" : [
      "Jim",
      "Jacob",
      "Daniel"
    ]
  },
  {
    "user" : "Fred",
    "friendIds" : [
      "Mariah"
    ]
  },
  {
    "user" : "Mary",
    "friendIds" : [
      "Isabella",
      "Michael"
    ]
  },
  {
    "user" : "Mariah",
    "friendIds" : [
      "Madison",
      "Eva"
    ]
  }
]
```

Grouping

To group results by arbitrary criteria, AQL provides the *COLLECT* keyword. *COLLECT* will perform a grouping, but no aggregation. Aggregation can still be added in the query if required.

Grouping by criteria

To group users by age, and result the names of the users with the highest ages, we'll issue a query like this:

```
FOR u IN users
  FILTER u.active == true
  COLLECT age = u.age INTO usersByAge
  SORT age DESC LIMIT 0, 5
  RETURN {
    "age" : age,
    "users" : usersByAge[*].u.name
  }

[
  {
    "age" : 37,
    "users" : [
      "John",
      "Sophia"
    ]
  },
  {
    "age" : 36,
    "users" : [
      "Fred",
      "Emma"
    ]
  },
  {
    "age" : 34,
    "users" : [
      "Madison"
    ]
  },
  {
    "age" : 33,
    "users" : [
      "Chloe",
      "Michael"
    ]
  },
  {
    "age" : 32,
    "users" : [
      "Alexander"
    ]
  }
]
```

The query will put all users together by their *age* attribute. There will be one result document per distinct *age* value (let aside the *LIMIT*). For each group, we have access to the matching document via the *usersByAge* variable introduced in the *COLLECT* statement.

list expander

The *usersByAge* variable contains the full documents found, and as we're only interested in user names, we'll use the list expander (*[\]*) to extract just the name* attribute of all user documents in each group.

The *[\]* *expander is just a handy short-cut. Instead of* usersByAge[*].u.name* we could also write:

```
FOR temp IN usersByAge
  RETURN temp.u.name
```

Grouping by multiple criteria

To group by multiple criteria, we'll use multiple arguments in the *COLLECT* clause. For example, to group users by *ageGroup* (a derived value we need to calculate first) and then by *gender*, we'll do:

```
FOR u IN users
  FILTER u.active == true
  COLLECT ageGroup = FLOOR(u.age / 5) * 5,
          gender = u.gender INTO group
  SORT ageGroup DESC
  RETURN {
    "ageGroup" : ageGroup,
    "gender" : gender
  }

[
  {
    "ageGroup" : 35,
    "gender" : "f"
  },
  {
    "ageGroup" : 35,
    "gender" : "m"
  },
  {
    "ageGroup" : 30,
    "gender" : "f"
  },
  {
    "ageGroup" : 30,
    "gender" : "m"
  },
  {
    "ageGroup" : 25,
    "gender" : "f"
  },
  {
    "ageGroup" : 25,
    "gender" : "m"
  }
]
```

Aggregation

So far we only grouped data without aggregation. Adding aggregation is simple in AQL, as all that needs to be done is to run an aggregate function on the list created by the *INTO* clause of a *COLLECT* statement:

```
FOR u IN users
  FILTER u.active == true
  COLLECT ageGroup = FLOOR(u.age / 5) * 5,
          gender = u.gender INTO group
  SORT ageGroup DESC
  RETURN {
    "ageGroup" : ageGroup,
    "gender" : gender,
    "numUsers" : LENGTH(group)
  }

[
  {
    "ageGroup" : 35,
    "gender" : "f",
    "numUsers" : 2
  },
  {
    "ageGroup" : 35,
    "gender" : "m",
    "numUsers" : 2
  },
  {
    "ageGroup" : 30,
    "gender" : "f",
    "numUsers" : 4
  },
  {
    "ageGroup" : 30,
    "gender" : "m",
    "numUsers" : 4
  },
  {
    "ageGroup" : 25,
    "gender" : "f",
    "numUsers" : 2
  },
  {
    "ageGroup" : 25,
    "gender" : "m",
    "numUsers" : 2
  }
]
```

We have used the function *LENGTH* (returns the length of a list) here. This is the equivalent to SQL's *SELECT g, COUNT()* FROM ... GROUP BY g. *In addition to* LENGTH *AQL also provides* MAX, MIN, SUM *and* AVERAGE* as basic aggregation functions.

In AQL all aggregation functions can be run on lists only. If an aggregation function is run on anything that is not a list, an error will be occur and the query will fail.

Post-filtering aggregated data

To filter on the results of a grouping or aggregation operation (i.e. something similar to *HAVING* in SQL), simply add another *FILTER* clause after the *COLLECT* statement.

For example, to get the 3 *ageGroup*s with the most users in them:

```
FOR u IN users
  FILTER u.active == true
  COLLECT ageGroup = FLOOR(u.age / 5) * 5 INTO group
  LET numUsers = LENGTH(group)
  FILTER numUsers > 2 // group must contain at least 3 users in
order to qualify
  SORT numUsers DESC
  LIMIT 0, 3
  RETURN {
    "ageGroup" : ageGroup,
    "numUsers" : numUsers,
    "users" : group[*].u.name
  }

[
  {
    "ageGroup" : 30,
    "numUsers" : 8,
    "users" : [
      "Abigail",
      "Madison",
      "Anthony",
      "Alexander",
      "Isabella",
      "Chloe",
      "Daniel",
      "Michael"
    ]
  },
  {
    "ageGroup" : 25,
    "numUsers" : 4,
    "users" : [
      "Mary",
      "Mariah",
      "Jim",
      "Diego"
    ]
  },
  {
    "ageGroup" : 35,
    "numUsers" : 4,
    "users" : [
      "Fred",
      "John",
      "Emma",
      "Sophia"
    ]
  }
]
```

To increase readability, the repeated expression *LENGTH(group)* was put into a variable *numUsers*. The *FILTER* on *numUsers* is the SQL HAVING equivalent.

Language basics

Whitespace

Whitespace can be used in the query text to increase its readability. However, for the parser any whitespace (spaces, carriage returns, line feeds, and tab stops) does not have any special meaning except that it separates individual tokens in the query. Whitespace within strings or names must be enclosed in quotes in order to be preserved.

Comments

Comments can be embedded at any position in a query. The text contained in the comment is ignored by the AQL parser. Comments cannot be nested, meaning the comment text may not contain another comment.

AQL supports two types of comments:

- Single line comments: These start with a double forward slash and end at the end of the line, or the end of the query string (whichever is first).
- Multi line comments: These start with a forward slash and asterisk, and end with an asterisk and a following forward slash. They can span as many lines as necessary.

```
/* this is a comment */ RETURN 1
/* these */ RETURN /* are */ 1 /* multiple */ + /* comments */ 1
/* this is
   a multi line
   comment */
// a single line comment
```

Keywords

On the top level, AQL offers the following operations:

- FOR: list iteration
- RETURN: results projection
- FILTER: results filtering
- SORT: result sorting
- LIMIT: result slicing
- LET: variable assignment
- COLLECT: result grouping

Each of the above operations can be initiated in a query by using a keyword of the same name. An AQL query can (and typically does) consist of multiple of the above operations.

An example AQL query might look like this:

```
FOR u IN users
    FILTER u.type == "newbie" && u.active == true
    RETURN u.name
```

In this example query, the terms *FOR*, *FILTER*, and *RETURN* initiate the higher-level operation according to their name. These terms are also keywords, meaning that they have a special meaning in the language.

For example, the query parser will use the keywords to find out which high-level operations to execute. That also means keywords can only be used at certain locations in a query. This also makes all keywords reserved words that must not be used for other purposes than they are intended for.

For example, it is not possible to use a keyword as a collection or attribute name. If a collection or attribute need to have the same name as a keyword, the collection or attribute name needs to be quoted.

Keywords are case-insensitive, meaning they can be specified in lower, upper, or mixed case in queries. In this documentation, all keywords are written in upper case to make them distinguishable from other query parts.

In addition to the higher-level operations keywords, there are other keywords. The current list of keywords is:

- FOR
- RETURN
- FILTER
- SORT
- LIMIT
- LET
- COLLECT
- ASC
- DESC
- IN
- INTO
- NULL
- TRUE
- FALSE

Additional keywords might be added in future versions of ArangoDB.

Names

In general, names are used to identify objects (collections, attributes, variables, and functions) in AQL queries.

The maximum supported length of any name is 64 bytes. Names in AQL are always case-sensitive.

Keywords must not be used as names. If a reserved keyword should be used as a name, the name must be enclosed in backticks. Enclosing a name in backticks allows using otherwise-reserved keywords as names. An example for this is:

```
FOR f IN `filter`
  RETURN f.`sort`
```

Due to the backticks, *filter* and *sort* are interpreted as names and not as keywords here.

Collection names

Collection names can be used in queries as they are. If a collection happens to have the same name as a keyword, the name must be enclosed in backticks.

Please refer to the [Naming Conventions in ArangoDB](#) about collection naming conventions.

Attribute names

When referring to attributes of documents from a collection, the fully qualified attribute name must be used. This is because multiple collections with ambiguous attribute names might be used in a query. To avoid any ambiguity, it is not allowed to refer to an unqualified attribute name.

Please refer to the [Naming Conventions in ArangoDB](#) for more information about the attribute naming conventions.

```
FOR u IN users
   FOR f IN friends
FILTER u.active == true && f.active == true && u.id == f.userId
RETURN u.name
```

In the above example, the attribute names *active*, *name*, *id*, and *userId* are qualified using the collection names they belong to (*u* and *f* respectively).

Variable names

AQL offers the user to assign values to additional variables in a query. All variables that are assigned a value must have a name that is unique within the context of the query. Variable names must be different from the names of any collection name used in the same query.

```
FOR u IN users
   LET friends = u.friends
   RETURN { "name" : u.name, "friends" : friends }
```

In the above query, *users* is a collection name, and both *u* and *friends* are variable names. This is because the *FOR* and *LET* operations need target variables to store their intermediate results.

Allowed characters in variable names are the letters *a* to *z* (both in lower and upper case), the numbers *0* to *9* and the underscore (_) symbol. A variable name must not start with a number. If a variable name starts with the underscore character, it must also contain at least one letter (a-z or A-Z).

Data types

AQL supports both primitive and compound data types. The following types are available:

- Primitive types: Consisting of exactly one value
  - null: An empty value, also: The absence of a value
  - bool: Boolean truth value with possible values *false* and *true*
  - number: Signed (real) number
  - string: UTF-8 encoded text value
- Compound types: Consisting of multiple values
  - list: Sequence of values, referred to by their positions
  - document: Sequence of values, referred to by their names

Numeric literals

Numeric literals can be integers or real values. They can optionally be signed using the + or - symbols. The scientific notation is also supported.

```
1
42
-1
-42
1.23
-99.99
0.1
-4.87e103
```

All numeric values are treated as 64-bit double-precision values internally. The internal format used is IEEE 754.

String literals

String literals must be enclosed in single or double quotes. If the used quote character is to be used itself within the string literal, it must be escaped using the backslash symbol. Backslash literals themselves also be escaped using a backslash.

```
"yikes!"
"don't know"
"this is a \"quoted\" word"
"this is a longer string."
"the path separator on Windows is \\"

'yikes!'
'don\'t know'
'this is a longer string."
'the path separator on Windows is \\'
```

All string literals must be UTF-8 encoded. It is currently not possible to use arbitrary binary data if it is not UTF-8 encoded. A workaround to use binary data is to encode the data using base64 or other algorithms on the application side before storing, and decoding it on application side after retrieval.

Lists

AQL supports two compound types:

- lists: A composition of unnamed values, each accessible by their positions
- documents: A composition of named values, each accessible by their names

The first supported compound type is the list type. Lists are effectively sequences of (unnamed/anonymous) values. Individual list elements can be accessed by their positions. The order of elements in a list is important.

A *list-declaration* starts with the *[* symbol and ends with the *]* symbol. A *list-declaration* contains zero or many *expression*s, separated from each other with the *,* symbol.

In the easiest case, a list is empty and thus looks like:

```
[ ]
```

List elements can be any legal *expression* values. Nesting of lists is supported.

```
[ 1, 2, 3 ]
[ -99, "yikes!", [ true, [ "no"], [ ] ], 1 ]
[ [ "fox", "marshal" ] ]
```

Individual list values can later be accesses by their positions using the *[]* accessor. The position of the accessed element must be a numeric value. Positions start at 0. It is also possible to use negative index values to access list values starting from the end of the list. This is convenient if the length of the list is unknown and access to elements at the end of the list is required.

```
// access 1st list element (element start at index 0)
u.friends[0]

// access 3rd list element
u.friends[2]
```

```
// access last list element
u.friends[-1]

// access second last list element
u.friends[-2]
```

Documents

The other supported compound type is the document type. Documents are a composition of zero to many attributes. Each attribute is a name/value pair. Document attributes can be accessed individually by their names.

Document declarations start with the *{* symbol and end with the *}* symbol. A document contains zero to many attribute declarations, separated from each other with the *,* symbol. In the simplest case, a document is empty. Its declaration would then be:

```
{ }
```

Each attribute in a document is a name/value pair. Name and value of an attribute are separated using the *:* symbol.

The attribute name is mandatory and must be specified as a quoted or unquoted string. If a keyword is to be used as an attribute name, the name must be quoted.

Any valid expression can be used as an attribute value. That also means nested documents can be used as attribute values

```
{ name : "Peter" }
{ "name" : "Vanessa", "age" : 15 }
{ "name" : "John", likes : [ "Swimming", "Skiing" ], "address" : {
"street" : "Cucumber lane", "zip" : "94242" } }
```

Individual document attributes can later be accesses by their names using the . accessor. If a non-existing attribute is accessed, the result is *null*.

```
u.address.city.name
u.friends[0].name.first
```

Bind parameters

AQL supports the usage of bind parameters, thus allowing to separate the query text from literal values used in the query. It is good practice to separate the query text from the literal values because this will prevent (malicious) injection of keywords and other collection names into an existing query. This injection would be dangerous because it might change the meaning of an existing query.

Using bind parameters, the meaning of an existing query cannot be changed. Bind parameters can be used everywhere in a query where literals can be used.

The syntax for bind parameters is *@nameparameter* where *nameparameter* is the actual parameter name. The bind parameter values need to be passed along with the query when it is executed, but not as part of the query text itself. Please refer to the @ref HttpCursorHttp manual section for information about how to pass the bind parameter values to the server.

```
FOR u IN users
  FILTER u.id == @id && u.name == @nameparameter
  RETURN u
```

Bind parameter names must start with any of the letters *a* to *z* (both in lower and upper case) or a digit (*0* to *9*), and can be followed by any letter, digit or the underscore symbol.

A special type of bind parameter exists for injecting collection names. This type of bind parameter has a name prefixed with an additional @ symbol (thus when using the bind parameter in a query, two @ symbols must be used).

```
FOR u IN @@collection
  FILTER u.active == true
RETURN u
```

Type and value order

When checking for equality or inequality or when determining the sort order of values, AQL uses a deterministic algorithm that takes both the data types and the actual values into account.

The compared operands are first compared by their data types, and only by their data values if the operands have the same data types.

The following type order is used when comparing data types:

```
null < bool  < number < string < list < document
```

This means *null* is the smallest type in AQL and *document* is the type with the highest order. If the compared operands have a different type, then the comparison result is determined and the comparison is finished.

For example, the boolean *true* value will always be less than any numeric or string value, any list (even an empty list) or any document. Additionally, any string value (even an empty string) will always be greater than any numeric value, a boolean value, *true* or *false*.

```
null < false
null < true
null < 0
null < ''
null < ' '
null < '0'
null < 'abc'
null < [ ]
null < { }

false < true
false < 0
false < ''
false < ' '
false < '0'
false < 'abc'
false < [ ]
false < { }

true < 0
```

```
true < ''
true < ' '
true < '0'
true < 'abc'
true < [ ]
true < { }

0 < ''
0 < ' '
0 < '0'
0 < 'abc'
0 < [ ]
0 < { }

'' < ' '
'' < '0'
'' < 'abc'
'' < [ ]
'' < { }

[ ] < { }
```

If the two compared operands have the same data types, then the operands values are compared. For the primitive types (null, boolean, number, and string), the result is defined as follows:

- null: *null* is equal to *null*
- boolean: *false* is less than *true*
- number: numeric values are ordered by their cardinal value
- string: string values are ordered using a localized comparison, see @ref CommandLineDefaultLanguage "--default-language"

Note: unlike in SQL, *null* can be compared to any value, including *null* itself, without the result being converted into *null* automatically.

For compound, types the following special rules are applied:

Two list values are compared by comparing their individual elements position by position, starting at the first element. For each position, the element types are compared first. If the types are not equal, the comparison result is determined, and the comparison is finished. If the types are equal, then the values of the two elements are compared. If one of the lists is finished and the other list still has an element at a compared position, then *null* will be used as the element value of the fully traversed list.

If a list element is itself a compound value (a list or a document), then the comparison algorithm will check the element's sub values recursively. The element's sub elements are compared recursively.

```
[ ] < [ 0 ]
[ 1 ] < [ 2 ]
[ 1, 2 ] < [ 2 ]
[ 99, 99 ] < [ 100 ]
[ false ] < [ true ]
[ false, 1 ] < [ false, '' ]
```

Two documents operands are compared by checking attribute names and value. The attribute names are compared first. Before attribute names are compared, a combined list of all attribute names from both operands is created and sorted lexicographically. This means that the order in which attributes are declared in a document is not relevant when comparing two documents.

The combined and sorted list of attribute names is then traversed, and the respective attributes from the

two compared operands are then looked up. If one of the documents does not have an attribute with the sought name, its attribute value is considered to be *null*. Finally, the attribute value of both documents is compared using the before mentioned data type and value comparison. The comparisons are performed for all document attributes until there is an unambiguous comparison result. If an unambiguous comparison result is found, the comparison is finished. If there is no unambiguous comparison result, the two compared documents are considered equal.

```
{ } < { "a" : 1 }
{ } < { "a" : null }
{ "a" : 1 } < { "a" : 2 }
{ "b" : 1 } < { "a" : 0 }
{ "a" : { "c" : true } } < { "a" : { "c" : 0 } }
{ "a" : { "c" : true, "a" : 0 } } < { "a" : { "c" : false, "a" : 1
} }

{ "a" : 1, "b" : 2 } == { "b" : 2, "a" : 1 }
```

Accessing data from collections

Collection data can be accessed by specifying a collection name in a query. A collection can be understood as a list of documents, and that is how they are treated in AQL. Documents from collections are normally accessing using the *FOR* keyword. Note that when iterating over documents from a collection, the order of documents is undefined. To traverse documents in an explicit and deterministic order, the *SORT* keyword should be used in addition.

Data in collections is stored in documents, with each document potentially having different attributes than other documents. This is true even for documents of the same collection.

It is therefore quite normal to encounter documents that do not have some or all of the attributes that are queried in an AQL query. In this case, the non-existing attributes in the document will be treated as if they would exist with a value of *null*. That means that comparing a document attribute to *null* will return true if the document has the particular attribute and the attribute has a value of *null*, or that the document does not have the particular attribute at all.

For example, the following query will return all documents from the collection *users* that have a value of *null* in the attribute *name*, plus all documents from *users* that do not have the *name* attribute at all:

```
FOR u IN users
   FILTER u.name == null
   RETURN u
```

Furthermore, *null* is less than any other value (excluding *null* itself). That means documents with non-existing attributes might be included in the result when comparing attribute values with the less than or less equal operators.

For example, the following query will return all documents from the collection *users* that have an attribute *age* with a value less than *39*, but also all documents from the collection that do not have the attribute *age* at all.

```
FOR u IN users
   FILTER u.age < 39
   RETURN u
```

This behavior should always be taken into account when writing queries.

Functions

AQL supports functions to allow more complex computations. Functions can be called at any query position where an expression is allowed. The general function call syntax is:

```
FUNCTIONNAME(arguments)
```

where *FUNCTIONNAME* is the name of the function to be called, and *arguments* is a comma-separated list of function arguments. If a function does not need any arguments, the argument list can be left empty. However, even if the argument list is empty the parentheses around it are still mandatory to make function calls distinguishable from variable names.

Some example function calls:

```
HAS(user, "name")
LENGTH(friends)
COLLECTIONS()
```

In contrast to collection and variable names, function names are case-insensitive, i.e. *LENGTH(foo)* and *length(foo)* are equivalent.

Extending AQL

Since ArangoDB 1.3, it is possible to extend AQL with user-defined functions. These functions need to be written in Javascript, and be registered before usage in a query.

Please refer to [Extending AQL](#) for more details on this.

By default, any function used in an AQL query will be sought in the built-in function namespace *_aql*. This is the default namespace that contains all AQL functions that are shipped with ArangoDB. To refer to a user-defined AQL function, the function name must be fully qualified to also include the user-defined namespace. The *::* symbol is used as the namespace separator:

```
MYGROUP::MYFUNC()

MYFUNCTIONS::MATH::RANDOM()
```

As all AQL function names, user function names are also case-insensitive.

Type cast functions

As mentioned before, some of the operators expect their operands to have a certain data type. For example, the logical operators expect their operands to be boolean values, and the arithmetic operators expect their operands to be numeric values. If an operation is performed with operands of an unexpected type, the operation will fail with an error. To avoid such failures, value types can be converted explicitly in a query. This is called type casting.

In an AQL query, type casts are performed only upon request and not implicitly. This helps avoiding unexpected results. All type casts have to be performed by invoking a type cast function. AQL offers several type cast functions for this task. Each of the these functions takes an operand of any data type and returns a result value of type corresponding to the function name (e.g. *TO_NUMBER()* will return a number value):

- *TO_BOOL(value)*: Takes an input *value* of any type and converts it into the appropriate boolean

value as follows:

  - *null* is converted to *false*.
  - Numbers are converted to *true* if they are unequal to 0, and to *false* otherwise.
  - Strings are converted to *true* if they are non-empty, and to *false* otherwise.
  - Lists are converted to *true* if they are non-empty, and to *false* otherwise.
  - Documents are converted to *true* if they are non-empty, and to *false* otherwise.

- *TO_NUMBER(value)*: Takes an input *value* of any type and converts it into a numeric value as follows:

  - *null*, *false*, lists, and documents are converted to the value *0*.
  - *true* is converted to *1*.
  - Strings are converted to their numeric equivalent if the full string content is is a valid number, and to *0* otherwise.

- *TO_STRING(value)*: Takes an input *value* of any type and converts it into a string value as follows:

  - *null* is converted to the string *"null"*
  - *false* is converted to the string *"false"*, *true* to the string *"true"*
  - Numbers, lists and documents are converted to their string equivalents.

- *TO_LIST(value)*: Takes an input *value* of any type and converts it into a list value as follows:

  - *null* is converted to an empty list
  - Boolean values, numbers and strings are converted to a list containing the original value as its single element
  - Documents are converted to a list containing their attribute values as list elements

Type check functions

AQL also offers functions to check the data type of a value at runtime. The following type check functions are available. Each of these functions takes an argument of any data type and returns true if the value has the type that is checked for, and false otherwise.

The following type check functions are available:

- *IS_NULL(value)*: Checks whether *value* is a *null* value

- *IS_BOOL(value)*: Checks whether *value* is a *boolean* value

- *IS_NUMBER(value)*: Checks whether *value* is a *numeric* value

- *IS_STRING(value)*: Checks whether *value* is a *string* value

- *IS_LIST(value)*: Checks whether *value* is a *list* value

- *IS_DOCUMENT(value)*: Checks whether *value* is a *document* value

String functions

For string processing, AQL offers the following functions:

- *CONCAT(value1, value2, ... valuen)*: Concatenate the strings passed as in *value1* to *valuen*. *null* values are ignored

- *CONCAT_SEPARATOR(separator, value1, value2, ... valuen)*: Concatenate the strings passed as arguments *value1* to *valuen* using the *separator* string. *null* values are ignored

- *CHAR_LENGTH(value)*: Return the number of characters in *value*. This is a synonym for LENGTH(value)*

- *LOWER(value)*: Lower-case *value*

- *UPPER(value)*: Upper-case *value*

- *SUBSTRING(value, offset, length)*: Return a substring of *value*, starting at *offset* and with a maximum length of *length* characters. Offsets start at position 0

- *LEFT(value, LENGTH)*: Returns the *LENGTH* leftmost characters of the string *value*

- *RIGHT(value, LENGTH)*: Returns the *LENGTH* rightmost characters of the string *value*

- *TRIM(value, type)*: Returns the string *value* with whitespace stripped from the start and/or end. The optional *type* parameter specifies from which parts of the string the whitespace is stripped:

   - *type* 0 will strip whitespace from the start and end of the string
   - *type* 1 will strip whitespace from the start of the string only
   - *type* 2 will strip whitespace from the end of the string only

- *REVERSE(value)*: Returns the reverse of the string *value*

- *CONTAINS(text, search, return-index)*: Checks whether the string *search* is contained in the string *text*. By default, this function returns *true* if *search* is contained in *text*, and *false* otherwise. By passing *true* as the third function parameter *return-index*, the function will return the position of the first occurrence of *search* within *text*, starting at offset 0, or *-1* if *search* is not contained in *text*.

   The string matching performed by *CONTAINS* is case-sensitive.

- *LIKE(text, search, case-insensitive)*: Checks whether the pattern *search* is contained in the string *text*, using wildcard matching. Returns *true* if the pattern is contained in *text*, and *false* otherwise. The *pattern* string can contain the wildcard characters *%* (meaning any sequence of characters) and _ (any single character).

   The string matching performed by *LIKE* is case-sensitive by default, but by passing *true* as the third parameter, the matching will be case-insensitive.

   The value for *search* cannot be a variable or a document attribute. The actual value must be present at query parse time already.

Numeric functions

AQL offers some numeric functions for calculations. The following functions are supported:

- *FLOOR(value)*: Returns the integer closest but not greater to *value*

- *CEIL(value)*: Returns the integer closest but not less than *value*

- *ROUND(value)*: Returns the integer closest to *value*

- *ABS(value)*: Returns the absolute part of *value*

- *SQRT(value)*: Returns the square root of *value*

- *RAND()*: Returns a pseudo-random number between 0 and 1

Date functions

AQL offers functionality to work with dates. Dates are no datatypes of their own in AQL (neither they are

in JSON, which is often used as a format to ship data into and out of ArangoDB). Instead, dates in AQL are internally represented by either numbers (timestamps) or strings. The date functions in AQL provide mechanisms to convert from a numeric timestamp to a string representation and vice versa.

There are two date functions in AQL to create dates for further use:

- *DATE_TIMESTAMP(date)*: Creates a UTC timestamp value from *date*.

- *DATE_TIMESTAMP(year, month, day, hour, minute, second, millisecond)*: Same as before, but allows specifying the individual date components separately. All parameters after *day* are optional.

- *DATE_ISO8601(date)*: Returns an ISO8601 date time string from *date*. The date time string will always use UTC time, indicated by the *Z* at its end.

- *DATE_ISO8601(year, month, day, hour, minute, second, millisecond)*: same as before, but allows specifying the individual date components separately. All parameters after *day* are optional.

These two above date functions accept the following input values:

- numeric timestamps, indicating the number of milliseconds elapsed since the UNIX epoch (i.e. January 1st 1970 00:00:00 UTC). An example timestamp value is *1399472349522*, which translates to *2014-05-07T14:19:09.522Z*.

- date time strings in formats *YYYY-MM-DDTHH:MM:SS.MMM*, *YYYY-MM-DD HH:MM:SS.MMM*, or *YYYY-MM-DD* Milliseconds are always optional. A timezone difference may optionally be added at the end of the string, with the hours and minutes that need to be added or subtracted to the date time value. For example, *2014-05-07T14:19:09+01:00* can be used to specify a one hour offset, and *2014-05-07T14:19:09+07:30* can be specified for seven and half hours offset. Negative offsets are also possible. Alternatively to an offset, a *Z* can be used to indicate UTC / Zulu time.

  An example value is *2014-05-07T14:19:09.522Z* meaning May 7th 2014, 14:19:09 and 522 milliseconds, UTC / Zulu time. Another example value without time component is *2014-05-07Z*.

  Please note that if no timezone offset is specified in a datestring, ArangoDB will assume UTC time automatically. This is done to ensure portability of queries across servers with different timezone settings, and because timestamps will always be UTC-based.

- individual date components as separate function arguments, in the following order:

  - year
  - month
  - day
  - hour
  - minute
  - second
  - millisecond

  All components following *day* are optional and can be omitted. Note that no timezone offsets can be specified when using separate date components, and UTC / Zulu time will be used.

The following calls to *DATE_TIMESTAMP* are equivalent and will all return *1399472349522*:

```
DATE_TIMESTAMP("2014-05-07T14:19:09.522")
DATE_TIMESTAMP("2014-05-07T14:19:09.522Z")
DATE_TIMESTAMP("2014-05-07 14:19:09.522")
DATE_TIMESTAMP("2014-05-07 14:19:09.522Z")
DATE_TIMESTAMP(2014, 5, 7, 14, 19, 9, 522)
DATE_TIMESTAMP(1399472349522)
```

The same is true for calls to *DATE_ISO8601* that also accepts variable input formats:

```
DATE_ISO8601("2014-05-07T14:19:09.522Z")
DATE_ISO8601("2014-05-07 14:19:09.522Z")
DATE_ISO8601(2014, 5, 7, 14, 19, 9, 522)
DATE_ISO8601(1399472349522)
```

The above functions are all equivalent and will return *"2014-05-07T14:19:09.522Z"*.

The following date functions can be used with dates created by *DATE_TIMESTAMP* and *DATE_ISO8601*:

- *DATE_DAYOFWEEK(date)*: Returns the weekday number of *date*. The return values have the following meanings:

    - 0: Sunday
    - 1: Monday
    - 2: Tuesday
    - 3: Wednesday
    - 4: Thursday
    - 5: Friday
    - 6: Saturday

- *DATE_YEAR(date)*: Returns the year part of *date* as a number.

- *DATE_MONTH(date)*: Returns the month part of *date* as a number.

- *DATE_DAY(date)*: Returns the day part of *date* as a number.

- *DATE_HOUR(date)*: Returns the hour part of *date* as a number.

- *DATE_MINUTE(date)*: Returns the minute part of *date* as a number.

- *DATE_SECOND(date)*: Returns the seconds part of *date* as a number.

- *DATE_MILLISECOND(date)*: Returns the milliseconds part of *date* as a number.

The following other date functions are also available:

- *DATE_NOW()*: Returns the current time as a timestamp.

    Note that this function is evaluated on every invocation and may return different values when invoked multiple times in the same query.

List functions

AQL supports the following functions to operate on list values:

- *LENGTH(list)*: Returns the length (number of list elements) of *list*. If *list* is a document, returns the number of attribute keys of the document, regardless of their values.

- *FLATTEN(list), depth)*: Turns a list of lists into a flat list. All list elements in *list* will be expanded in the result list. Non-list elements are added as they are. The function will recurse into sub-lists up to a depth of *depth*. *depth* has a default value of 1.

Examples

```
FLATTEN([ 1, 2, [ 3, 4 ], 5, [ 6, 7 ], [ 8, [ 9, 10 ] ])
```

will produce:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, [ 9, 10 ] ]
```

To fully flatten the list, use a *depth* of 2:

```
FLATTEN([ 1, 2, [ 3, 4 ], 5, [ 6, 7 ], [ 8, [ 9, 10 ] ], 2)
```

This will produce:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

- *MIN(list)*: Returns the smallest element of *list*. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *MAX(list)*: Returns the greatest element of *list*. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *AVERAGE(list)*: Returns the average (arithmetic mean) of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *SUM(list)*: Returns the sum of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored.

- *MEDIAN(list)*: Returns the median value of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *VARIANCE_POPULATION(list)*: Returns the population variance of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *VARIANCE_SAMPLE(list)*: Returns the sample variance of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *STDDEV_POPULATION(list)*: Returns the population standard deviation of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *STDDEV_SAMPLE(list)*: Returns the sample standard deviation of the values in *list*. This requires the elements in *list* to be numbers. *null* values are ignored. If the list is empty or only *null* values are contained in the list, the function will return *null*.

- *REVERSE(list)*: Returns the elements in *list* in reversed order.

- *FIRST(list)*: Returns the first element in *list* or *null* if the list is empty.

- *LAST(list)*: Returns the last element in *list* or *null* if the list is empty.

- *NTH(list, position)*: Returns the list element at position *position*. Positions start at 0. If *position* is negative or beyond the upper bound of the list specified by *list*, then *null* will be returned.

- *POSITION(list, search, return-index)*: Returns the position of the element *search* in list *list*. Positions start at 0. If the element is not found, then *-1* is returned. If *return-index* is *false*, then instead of the position only *true* or *false* are returned, depending on whether the sought element

is contained in the list.

- *SLICE(list, start, length)*: Extracts a slice of the list specified by *list*. The extraction will start at list element with position *start*. Positions start at 0. Up to *length* elements will be extracted. If *length* is not specified, all list elements starting at *start* will be returned. If *start* is negative, it can be used to indicate positions from the end of the list.

Examples

```
SLICE([ 1, 2, 3, 4, 5 ], 0, 1)
```

will return *[1]*

```
SLICE([ 1, 2, 3, 4, 5 ], 1, 2)
```

will return *[2, 3]*

```
SLICE([ 1, 2, 3, 4, 5 ], 3)
```

will return *[4, 5]*

```
SLICE([ 1, 2, 3, 4, 5 ], 1, -1)
```

will return *[2, 3, 4]*

```
SLICE([ 1, 2, 3, 4, 5 ], 0, -2)
```

will return *[1, 2, 3]*

- *UNIQUE(list)*: Returns all unique elements in *list*. To determine uniqueness, the function will use the comparison order defined in @ref AqlTypeOrder. Calling this function might return the unique elements in any order.

- *UNION(list1, list2, ...)*: Returns the union of all lists specified. The function expects at least two list values as its arguments. The result is a list of values in an undefined order.

  Note: No duplicates will be removed. In order to remove duplicates, please use either *UNION_DISTINCT* function or apply the *UNIQUE* on the result of *union*.

Examples

```
RETURN UNION(
  [ 1, 2, 3 ],
  [ 1, 2 ]
)
```

will produce:

```
[ [ 1, 2, 3, 1, 2 ] ]
```

with duplicate removal:

```
   RETURN UNIQUE(
     UNION(
       [ 1, 2, 3 ],
       [ 1, 2 ]
     )
   )
```

will produce:

```
   [ [ 1, 2, 3 ] ]
```

- *UNION_DISTINCT(list1, list2, ...)*: Returns the union of distinct values of all lists specified. The function expects at least two list values as its arguments. The result is a list of values in an undefined order.

- *MINUS(list1, list2, ...)*: Returns the difference of all lists specified. The function expects at least two list values as its arguments. The result is a list of values that occur in the first list but not in any of the subsequent lists. The order of the result list is undefined and should not be relied on. Note: duplicates will be removed.

- *INTERSECTION(list1, list2, ...)*: Returns the intersection of all lists specified. The function expects at least two list values as its arguments. The result is a list of values that occur in all arguments. The order of the result list is undefined and should not be relied on.

  Note: Duplicates will be removed.

Apart from these functions, AQL also offers several language constructs (e.g. *FOR*, *SORT*, *LIMIT*, *COLLECT*) to operate on lists.

Document functions

AQL supports the following functions to operate on document values:

- *MATCHES(document, examples, return-index)*: Compares the document *document* against each example document provided in the list *examples*. If *document* matches one of the examples, *true* is returned, and if there is no match *false* will be returned. The default return value type can be changed by passing *true* as the third function parameter *return-index*. Setting this flag will return the index of the example that matched (starting at offset 0), or *-1* if there was no match.

  The comparisons will be started with the first example. All attributes of the example will be compared against the attributes of *document*. If all attributes match, the comparison stops and the result is returned. If there is a mismatch, the function will continue the comparison with the next example until there are no more examples left.

  The *examples* must be a list of 1..n example documents, with any number of attributes each. Note: specifying an empty list of examples is not allowed.

  @EXAMPLE

```
  RETURN MATCHES(
    { "test" : 1 }, [
      { "test" : 1, "foo" : "bar" },
      { "foo" : 1 },
      { "test : 1 }
    ], true)
```

This will return *2*, because the third example matches, and because the *return-index* flag is set to *true*.

- *MERGE(document1, document2, ... documentn)*: *Merges the documents in* document1 *to* documentn* into a single document. If document attribute keys are ambiguous, the merged result will contain the values of the documents contained later in the argument list.

  For example, two documents with distinct attribute names can easily be merged into one:

```
RETURN MERGE(
   { "user1" : { "name" : "J" } },
   { "user2" : { "name" : "T" } }
)


[
   { "user1" : { "name" : "J" },
"user2" : { "name" : "T" } }
]
```

  When merging documents with identical attribute names, the attribute values of the latter documents will be used in the end result:

```
RETURN MERGE(
   { "users" : { "name" : "J" } },
   { "users" : { "name" : "T" } }
)


[
   { "users" : { "name" : "T" } }
]
```

  Please note that merging will only be done for top-level attributes. If you wish to merge sub-attributes, you should consider using *MERGE_RECURSIVE* instead.

- *MERGE_RECURSIVE(document1, document2, ... documentn)*: Recursively merges the documents in *document1* to *documentn* into a single document. If document attribute keys are ambiguous, the merged result will contain the values of the documents contained later in the argument list.

  For example, two documents with distinct attribute names can easily be merged into one:

```
RETURN MERGE_RECURSIVE(
   { "user-1" : { "name" : "J", "livesIn" : { "city" : "LA" } }
},
   { "user-1" : { "age" : 42, "livesIn" : { "state" : "CA" } }
}
)


[
   { "user-1" : { "name" : "J", "livesIn" : { "city" : "LA",
"state" : "CA" }, "age" : 42 } }
]
```

- *TRANSLATE(value, lookup, defaultValue)*: Looks up the value *value* in the *lookup* document. If *value* is a key in *lookup*, then *value* will be replaced with the lookup value found. If *value* is not present in *lookup*, then *defaultValue* will be returned if specified. If no *defaultValue* is specified,

value will be returned:

```
RETURN TRANSLATE("FR", { US: "United States", UK: "United
Kingdom", FR: "France" })

"France"

RETURN TRANSLATE(42, { foo: "bar", bar: "baz" }, "not found!")

"not found!"
```

- *HAS(document, attributename)*: Returns *true* if *document* has an attribute named *attributename*, and *false* otherwise.

- *ATTRIBUTES(document,* removeInternal, sort)*: Returns the attribute names of the document* document as a list. If *removeInternal* is set to *true*, then all internal attributes (such as *_id*, *_key* etc.) are removed from the result. If *sort* is set to *true*, then the attribute names in the result will be sorted. Otherwise they will be returned in any order.

- *UNSET(document, attributename, ...)*: Removes the attributes *attributename* (can be one or many) from *document*. All other attributes will be preserved. Multiple attribute names can be specified by either passing multiple individual string argument names, or by passing a list of attribute names:

```
RETURN UNSET(doc, '_id', '_key', [ 'foo', 'bar' ])
```

- *KEEP(document, attributename, ...)*: Keeps only the attributes *attributename* (can be one or many) from *document*. All other attributes will be removed from the result. Multiple attribute names can be specified by either passing multiple individual string argument names, or by passing a list of attribute names:

```
RETURN KEEP(doc, 'firstname', 'name', 'likes')
```

- *PARSE_IDENTIFIER(document-handle)*: Parses the document handle specified in *document-handle* and returns a the handle's individual parts a separate attributes. This function can be used to easily determine the collection name and key from a given document. The *document-handle* can either be a regular document from a collection, or a document identifier string (e.g. *_users/1234*). Passing either a non-string or a non-document or a document without an *_id* attribute will result in an error.

```
RETURN PARSE_IDENTIFIER('_users/my-user')

[
  { "collection" : "_users", "key" : "my-user" }
]

RETURN PARSE_IDENTIFIER({ "_id" : "mycollection/mykey",
"value" : "some value" })

[
  { "collection" : "mycollection", "key" : "mykey" }
]
```

Geo functions

AQL offers the following functions to filter data based on geo indexes:

- *NEAR(collection, latitude, longitude, limit, distancename)*: Returns at most *limit* documents from collection *collection* that are near *latitude* and *longitude*. The result contains at most *limit* documents, returned in any order. If more than *limit* documents qualify, it is undefined which of the qualifying documents are returned. Optionally, the distances between the specified coordinate (*latitude* and *longitude*) and the document coordinates can be returned as well. To make use of that, an attribute name for the distance result has to be specified in the *distancename* argument. The result documents will contain the distance value in an attribute of that name. *limit* is an optional parameter since ArangoDB 1.3. If it is not specified or null, a limit value of 100 will be applied.

- *WITHIN(collection, latitude, longitude, radius, distancename)*: Returns all documents from collection *collection* that are within a radius of *radius* around that specified coordinate (*latitude* and *longitude*). The order in which the result documents are returned is undefined. Optionally, the distance between the coordinate and the document coordinates can be returned as well. To make use of that, an attribute name for the distance result has to be specified in the *distancename* argument. The result documents will contain the distance value in an attribute of that name.

Note: these functions require the collection *collection* to have at least one geo index. If no geo index can be found, calling this function will fail with an error.

Fulltext functions

AQL offers the following functions to filter data based on fulltext indexes:

- *FULLTEXT(collection, attribute, query)*: Returns all documents from collection *collection* for which the attribute *attribute* matches the fulltext query *query*. *query* is a comma-separated list of sought words (or prefixes of sought words). To distinguish between prefix searches and complete-match searches, each word can optionally be prefixed with either the *prefix:* or *complete:* qualifier. Different qualifiers can be mixed in the same query. Not specifying a qualifier for a search word will implicitly execute a complete-match search for the given word:

    - *FULLTEXT(emails, "body", "banana")* Will look for the word *banana* in the attribute *body* of the collection *collection*.

    - *FULLTEXT(emails, "body", "banana,orange")* Will look for boths the words *banana* and *orange* in the mentioned attribute. Only those documents will be returned that contain both words.

    - *FULLTEXT(emails, "body", "prefix:head")* Will look for documents that contain any words starting with the prefix *head*.

    - *FULLTEXT(emails, "body", "prefix:head,complete:aspirin")* Will look for all documents that contain a word starting with the prefix *head* and that also contain the (complete) word *aspirin*. Note: specifying *complete* is optional here.

    - *FULLTEXT(emails, "body", "prefix:cent,prefix:subst")* Will look for all documents that contain a word starting with the prefix *cent* and that also contain a word starting with the prefix *subst*.

    If multiple search words (or prefixes) are given, then by default the results will be AND-combined, meaning only the logical intersection of all searches will be returned. It is also possible to combine partial results with a logical OR, and with a logical NOT:

    - *FULLTEXT(emails, "body", "+this,+text,+document")* Will return all documents that contain all the mentioned words. Note: specifying the + symbols is optional here.

    - *FULLTEXT(emails, "body", "banana,|apple")* Will return all documents that contain either (or both) words *banana* or *apple*.

- *FULLTEXT(emails, "body", "banana,-apple")* Will return all documents that contain the word *banana* but do not contain the word *apple*.

- *FULLTEXT(emails, "body", "banana,pear,-cranberry")* Will return all documents that contain both the words *banana* and *pear* but do not contain the word *cranberry*.

No precedence of logical operators will be honored in a fulltext query. The query will simply be evaluated from left to right.

Note: the *FULLTEXT* function requires the collection *collection* to have a fulltext index on *attribute*. If no fulltext index is available, this function will fail with an error.

Graph functions

AQL has the following functions to traverse graphs:

Warning Deprecated

This query is deprecated and will be removed soon. Please use Graph operations instead.

- *PATHS(vertexcollection, edgecollection, direction, followcycles)*: returns a list of paths through the graph defined by the nodes in the collection *vertexcollection* and edges in the collection *edgecollection*. For each vertex in *vertexcollection*, it will determine the paths through the graph depending on the value of *direction*:

  - *"outbound"*: Follow all paths that start at the current vertex and lead to another vertex
  - *"inbound"*: Follow all paths that lead from another vertex to the current vertex
  - *"any"*: Combination of *"outbound"* and *"inbound"* The default value for *direction* is *"outbound"*. If *followcycles* is true, cyclic paths will be followed as well. This is turned off by default.

  The result of the function is a list of paths. Paths of length 0 will also be returned. Each path is a document consisting of the following attributes:

  - *vertices*: list of vertices visited along the path
  - *edges*: list of edges visited along the path (might be empty)
  - *source*: start vertex of path
  - *destination*: destination vertex of path

Examples

```
PATHS(friends, friendrelations, "outbound", false)

FOR p IN PATHS(friends, friendrelations, "outbound")
  FILTER p.source._id == "123456/123456" && LENGTH(p.edges) == 2
  RETURN p.vertices[*].name
```

Warning Deprecated

This query is deprecated and will be removed soon. Please use Graph operations instead.

- *TRAVERSAL(vertexcollection, edgecollection, startVertex, direction, options)*: Traverses the graph described by *vertexcollection* and *edgecollection*, starting at the vertex identified by id *startVertex*. Vertex connectivity is specified by the *direction* parameter:

  - *"outbound"*: Vertices are connected in *_from* to *_to* order
  - *"inbound"*: Vertices are connected in *_to* to *_from* order
  - *"any"*: Vertices are connected in both *_to* to *_from* and in *_from* to *_to* order

Additional options for the traversal can be provided via the *options* document:

- *strategy*: Defines the traversal strategy. Possible values are *depthfirst* and *breadthfirst*. Defaults to *depthfirst*
- *order*: Defines the traversal order: Possible values are *preorder* and *postorder*. Defaults to *preorder*
- *itemOrder*: Defines the level item order. Can be *forward* or *backward*. Defaults to *forward*
- *minDepth*: Minimum path depths for vertices to be included. This can be used to include only vertices in the result that are found after a certain minimum depth. Defaults to 0
- *maxIterations*: Maximum number of iterations in each traversal. This number can be set to prevent endless loops in traversal of cyclic graphs. When a traversal performs as many iterations as the *maxIterations* value, the traversal will abort with an error. If *maxIterations* is not set, a server-defined value may be used
- *maxDepth*: Maximum path depth for sub-edges expansion. This can be used to limit the depth of the traversal to a sensible amount. This should especially be used for big graphs to limit the traversal to some sensible amount, and for graphs containing cycles to prevent infinite traversals. The maximum depth defaults to 256, with the chance of this value being non-sensical. For several graphs, a much lower maximum depth is sensible, whereas for other, more list-oriented graphs a higher depth should be used
- *paths*: If *true*, the paths encountered during the traversal will also be returned along with each traversed vertex. If *false*, only the encountered vertices will be returned.
- *uniqueness*: An optional document with the following attributes:
  - *vertices*:
    - *none*: No vertex uniqueness is enforced
    - *global*: A vertex may be visited at most once. This is the default.
    - *path*: A vertex is visited only if not already contained in the current traversal path
  - *edges*:
    - *none*: No edge uniqueness is enforced
    - *global*: An edge may be visited at most once. This is the default
    - *path*: An edge is visited only if not already contained in the current traversal path

- *followEdges*: An optional list of example edge documents that the traversal will expand into. If no examples are given, the traversal will follow all edges. If one or many edge examples are given, the traversal will only follow an edge if it matches at least one of the specified examples. *followEdges* can also be a string with the name of an AQL user-defined function that should be responsible for checking if an edge should be followed. In this case, the AQL function will is expected to have the following signature:

```
function (config, vertex, edge, path)
```

The function is expected to return a boolean value. If it returns *true*, the edge will be followed. If *false* is returned, the edge will be ignored.

- *filterVertices*: An optional list of example vertex documents that the traversal will treat specially. If no examples are given, the traversal will handle all encountered vertices equally. If one or many vertex examples are given, the traversal will exclude any non-matching vertex from the result and/or not descend into it. Optionally, *filterVertices* can contain the name of a user-defined AQL function that should be responsible for filtering. If so, the AQL function is expected to have the following signature:

```
function (config, vertex, path)
```

If a custom AQL function is used, it is expected to return one of the following values:

- **[ ]**: Include the vertex in the result and descend into its connected edges
- **["prune"]**: Will include the vertex in the result but not descend into its connected edges
- **["exclude"]**: Will not include the vertex in the result but descend into its connected edges
- **["prune", "exclude"]**: Will completely ignore the vertex and its connected edges

- *vertexFilterMethod*: Only useful in conjunction with *filterVertices* and if no user-defined AQL function is used. If specified, it will influence how vertices are handled that don't match the examples in *filterVertices*:

  - **["prune"]**: Will include non-matching vertices in the result but not descend into them
  - **["exclude"]**: Will not include non-matching vertices in the result but descend into them
  - **["prune", "exclude"]**: Will neither include non-matching vertices in the result nor descend into them

The result of the TRAVERSAL function is a list of traversed points. Each point is a document consisting of the following attributes:

- *vertex*: The vertex at the traversal point
- *path*: The path history for the traversal point. The path is a document with the attributes *vertices* and *edges*, which are both lists. Note that *path* is only present in the result if the *paths* attribute is set in the *options*

Examples

```
TRAVERSAL(friends, friendrelations, "friends/john", "outbound", {
  strategy: "depthfirst",
  order: "postorder",
  itemOrder: "backward",
  maxDepth: 6,
  paths: true
})

// filtering on specific edges (by specifying example edges)
TRAVERSAL(friends, friendrelations, "friends/john", "outbound", {
  strategy: "breadthfirst",
  order: "preorder",
  itemOrder: "forward",
  followEdges: [ { type: "knows" }, { state: "FL" } ]
})

// filtering on specific edges and vertices
TRAVERSAL(friends, friendrelations, "friends/john", "outbound", {
  strategy: "breadthfirst",
  order: "preorder",
  itemOrder: "forward",
  followEdges: [ { type: "knows" }, { state: "FL" } ],
  filterVertices: [ { isActive: true }, { isDeleted: false } ],
  vertexFilterMethod: [ "prune", "exclude" ]
})

// using user-defined AQL functions for edge and vertex filtering
TRAVERSAL(friends, friendrelations, "friends/john", "outbound", {
  followEdges: "myfunctions::checkedge",
  filterVertices: "myfunctions::checkvertex"
```

```
  })

  // to register the custom AQL functions, execute something in the
fashion of the
  // following commands in arangosh once:
  var aqlfunctions = require("org/arangodb/aql/functions");

  // these are the actual filter functions
  aqlfunctions.register("myfunctions::checkedge", function (config,
vertex, edge, path) {
    return (edge.type !== 'dislikes'); // don't follow these edges
  }, false);

  aqlfunctions.register("myfunctions::checkvertex", function
(config, vertex, path) {
    if (vertex.isDeleted || ! vertex.isActive) {
      return [ "prune", "exclude" ]; // exclude these and don't
follow them
    }
    return [ ]; // include everything else
  }, false);
```

Warning Deprecated

This query is deprecated and will be removed soon. Please use [Graph operations](#) instead.

- *TRAVERSAL_TREE(vertexcollection, edgecollection, startVertex, direction, connectName, options)*: Traverses the graph described by *vertexcollection* and *edgecollection*, starting at the vertex identified by id *startVertex* and creates a hierarchical result. Vertex connectivity is establish by inserted an attribute which has the name specified via the *connectName* parameter. Connected vertices will be placed in this attribute as a list.

  The *options* are the same as for the *TRAVERSAL* function, except that the result will be set up in a way that resembles a depth-first, pre-order visitation result. Thus, the *strategy* and *order* attributes of the *options* attribute will be ignored.

Examples

```
  TRAVERSAL_TREE(friends, friendrelations, "friends/john",
"outbound", "likes", {
    itemOrder: "forward"
  })
```

When using one of AQL's graph functions please make sure that the graph does not contain cycles, or that you at least specify some maximum depth or uniqueness criteria for a traversal.

If no bounds are set, a traversal might run into an endless loop in a cyclic graph or sub-graph, and even in a non-cyclic graph, traversing far into the graph might consume a lot of processing time and memory for the result set.

Warning Deprecated

This query is deprecated and will be removed soon. Please use [Graph operations](#) instead.

- *SHORTEST_PATH(vertexcollection, edgecollection, startVertex, endVertex, direction, options)*: Determines the first shortest path from the *startVertex* to the *endVertex*. Both vertices must be present in the vertex collection specified in *vertexcollection*, and any connecting edges must be present in the collection specified by *edgecollection*. Vertex connectivity is specified by the

direction parameter:

- *"outbound"*: Vertices are connected in *_from* to *_to* order
- *"inbound"*: Vertices are connected in *_to* to *_from* order
- *"any"*: Vertices are connected in both *_to* to *_from* and in *_from* to *_to* order The search is aborted when a shortest path is found. Only the first shortest path will be returned. Any vertex will be visited at most once by the search.

Additional options for the traversal can be provided via the *options* document:

- *maxIterations*: Maximum number of iterations in the search. This number can be set to bound long-running searches. When a search performs as many iterations as the *maxIterations* value, the search will abort with an error. If *maxIterations* is not set, a server-defined value may be used.
- *paths*: If *true*, the result will not only contain the vertices along the shortest path, but also the connecting edges. If *false*, only the encountered vertices will be returned.
- *distance*: An optional custom function to be used when calculating the distance between a vertex and a neighboring vertex. The expected function signature is:

```
function (config, vertex1, vertex2, edge)
```

  Both vertices and the connecting edge will be passed into the function. The function is expected to return a numeric value that expresses the distance between the two vertices. Higher values will mean higher distances, giving the connection a lower priority in further analysis. If no custom distance function is specified, all vertices are assumed to have the same distance (1) to each other. If a function name is specified, it must have been registered as a regular user-defined AQL function.

- *followEdges*: An optional list of example edge documents that the search will expand into. If no examples are given, the search will follow all edges. If one or many edge examples are given, the search will only follow an edge if it matches at least one of the specified examples. *followEdges* can also be a string with the name of an AQL user-defined function that should be responsible for checking if an edge should be followed. In this case, the AQL function will is expected to have the following signature:

```
function (config, vertex, edge, path)
```

  The function is expected to return a boolean value. If it returns *true*, the edge will be followed. If *false* is returned, the edge will be ignored.

- *filterVertices*: An optional list of example vertex documents that the search will treat specially. If no examples are given, the search will handle all encountered vertices equally. If one or many vertex examples are given, the search will exclude the vertex from the result and/or not descend into it. Optionally, *filterVertices* can contain the name of a user-defined AQL function that should be responsible for filtering. If so, the AQL function is expected to have the following signature:

```
function (config, vertex, path)
```

  If a custom AQL function is used, it is expected to return one of the following values:

  - *[ ]*: Include the vertex in the result and descend into its connected edges
  - *["prune"]*: Will include the vertex in the result but not descend into its connected edges
  - *["exclude"]*: Will not include the vertex in the result but descend into its

connected edges
- *["prune", "exclude"]*: Will completely ignore the vertex and its connected edges

The result of the SHORTEST_PATH function is a list with the components of the shortest path. Each component is a document consisting of the following attributes:

- *vertex*: The vertex at the traversal point
- *path*: The path history for the traversal point. The path is a document with the attributes *vertices* and *edges*, which are both lists. Note that *path* is only present in the result if the *paths* attribute is set in the *options*.

Examples

```
  SHORTEST_PATH(cities, motorways, "cities/CGN", "cities/MUC",
"outbound", {
    paths: true
  })

  // using a user-defined distance function
  SHORTEST_PATH(cities, motorways, "cities/CGN", "cities/MUC",
"outbound", {
    paths: true,
    distance: "myfunctions::citydistance"
  })

  // using a user-defined function to filter edges
  SHORTEST_PATH(cities, motorways, "cities/CGN", "cities/MUC",
"outbound", {
    paths: true,
    followEdges: "myfunctions::checkedge"
  })

  // to register a custom AQL distance function, execute something
in the fashion of the
  // following commands in arangosh once:
  var aqlfunctions = require("org/arangodb/aql/functions");

  // this is the actual distance function
  aqlfunctions.register("myfunctions::distance", function (config,
vertex1, vertex2, edge) {
    return Math.sqrt(Math.pow(vertex1.x - vertex2.x) +
Math.pow(vertex1.y - vertex2.y));
  }, false);

  // this is the filter function for the edges
  aqlfunctions.register("myfunctions::checkedge", function (config,
vertex, edge, path) {
    return (edge.underConstruction === false); // don't follow
these edges
  }, false);
```

- *EDGES(edgecollection, startvertex, direction, edgeexamples)*: Return all edges connected to the vertex *startvertex* as a list. The possible values for *direction* are:

  - *outbound*: Return all outbound edges
  - *inbound*: Return all inbound edges
  - *any*: Return outbound and inbound edges

The *edgeexamples* parameter can optionally be used to restrict the results to specific edge connections only. The matching is then done via the *MATCHES* function. To not restrict the result to specific connections, *edgeexamples* should be left unspecified.

Examples

```
  EDGES(friendrelations, "friends/john", "outbound")
  EDGES(friendrelations, "friends/john", "any", [ { "$label":
"knows" } ])
```

Warning Deprecated

This query is deprecated and will be removed soon. Please use Graph operations instead.

- *NEIGHBORS(vertexcollection, edgecollection, startvertex, direction, edgeexamples)*: Return all neighbors that are directly connected to the vertex *startvertex* as a list. The possible values for *direction* are:

  - *outbound*: Return all outbound edges
  - *inbound*: Return all inbound edges
  - *any*: Return outbound and inbound edges

  The *edgeexamples* parameter can optionally be used to restrict the results to specific edge connections only. The matching is then done via the *MATCHES* function. To not restrict the result to specific connections, *edgeexamples* should be left unspecified.

Examples

```
  NEIGHBORS(friends, friendrelations, "friends/john", "outbound")
  NEIGHBORS(users, usersrelations, "users/john", "any", [ {
"$label": "recommends" } ] )
```

Control flow functions

AQL offers the following functions to let the user control the flow of operations:

- *NOT_NULL(alternative, ...)*: Returns the first alternative that is not *null*, and *null* if all alternatives are *null* themselves

- *FIRST_LIST(alternative, ...)*: Returns the first alternative that is a list, and *null* if none of the alternatives is a list

- *FIRST_DOCUMENT(alternative, ...)*: Returns the first alternative that is a document, and *null* if none of the alternatives is a document

Miscellaneous functions

Finally, AQL supports the following functions that do not belong to any of the other function categories:

- *COLLECTIONS()*: Returns a list of collections. Each collection is returned as a document with attributes *name* and *_id*

- *CURRENT_USER()*: Returns the name of the current user. The current user is the user account name that was specified in the *Authorization* HTTP header of the request. It will only be populated if authentication on the server is turned on, and if the query was executed inside a request context. Otherwise, the return value of this function will be *null*.

- *DOCUMENT(collection, id)*: Returns the document which is uniquely identified by the *id*. ArangoDB will try to find the document using the *_id* value of the document in the specified collection. If there is a mismatch between the *collection* passed and the collection specified in *id*, then *null* will be returned. Additionally, if the *collection* matches the collection value specified in *id* but the document cannot be found, *null* will be returned. This function also allows *id* to be a list of ids. In this case, the function will return a list of all documents that could be found.

Examples

```
DOCUMENT(users, "users/john")
DOCUMENT(users, "john")

DOCUMENT(users, [ "users/john", "users/amy" ])
DOCUMENT(users, [ "john", "amy" ])
```

Note: The *DOCUMENT* function is polymorphic since ArangoDB 1.4. It can now be used with a single parameter *id* as follows:

- *DOCUMENT(id)*: In this case, *id* must either be a document handle string (consisting of collection name and document key) or a list of document handle strings, e.g.

```
DOCUMENT("users/john")
DOCUMENT([ "users/john", "users/amy" ])
```

- *SKIPLIST(collection, condition, skip, limit)*: Return all documents from a skiplist index on collection *collection* that match the specified *condition*. This is a shortcut method to use a skiplist index for retrieving specific documents in indexed order. The skiplist index supports equality and less than/greater than queries. The *skip* and *limit* parameters are optional but can be specified to further limit the results:

```
SKIPLIST(test, { created: [[ '>', 0 ]] }, 0, 100)
SKIPLIST(test, { age: [[ '>', 25 ], [ '<=', 65 ]] })
SKIPLIST(test, { a: [[ '==', 10 ]], b: [[ '==', 25 ]] }
```

The *condition* document must contain an entry for each attribute that is contained in the index. It is not allowed to specify just a subset of attributes that are present in an index. Additionally the attributes in the *condition* document must be specified in the same order as in the index. If no suitable skiplist index is found, an error will be raised and the query will be aborted.

Query results

Result sets

The result of an AQL query is a list of values. The individual values in the result list may or may not have a homogeneous structure, depending on what is actually queried.

For example, when returning data from a collection with inhomogeneous documents (the individual documents in the collection have different attribute names) without modification, the result values will as well have an inhomogeneous structure. Each result value itself is a document:

```
FOR u IN users
  RETURN u

[ { "id" : 1, "name" : "John", "active" : false },
  { "age" : 32, "id" : 2, "name" : "Vanessa" },
  { "friends" : [ "John", "Vanessa" ], "id" : 3, "name" : "Amy" } ]
```

However, if a fixed set of attributes from the collection is queried, then the query result values will have a homogeneous structure. Each result value is still a document:

```
FOR u IN users
  RETURN { "id" : u.id, "name" : u.name }

[ { "id" : 1, "name" : "John" },
  { "id" : 2, "name" : "Vanessa" },
  { "id" : 3, "name" : "Amy" } ]
```

It is also possible to query just scalar values. In this case, the result set is a list of scalars, and each result value is a scalar value:

```
FOR u IN users
  RETURN u.id

[ 1, 2, 3 ]
```

If a query does not produce any results because no matching data can be found, it will produce an empty result list:

```
[ ]
```

Errors

Issuing an invalid query to the server will result in a parse error if the query is syntactically invalid. ArangoDB will detect such errors during query inspection and abort further processing. Instead, the error number and an error message are returned so that the errors can be fixed.

If a query passes the parsing stage, all collections referenced in the query will be opened. If any of the referenced collections is not present, query execution will again be aborted and an appropriate error message will be returned.

Under some circumstances, executing a query might also produce run-time errors that cannot be predicted from inspecting the query text alone. This is because queries might use data from collections that might

also be inhomogeneous. Some examples that will cause run-time errors are:

- Division by zero: Will be triggered when an attempt is made to use the value *0* as the divisor in an arithmetic division or modulus operation
- Invalid operands for arithmetic operations: Will be triggered when an attempt is made to use any non-numeric values as operands in arithmetic operations. This includes unary (unary minus, unary plus) and binary operations (plus, minus, multiplication, division, and modulus)
- Invalid operands for logical operations: Will be triggered when an attempt is made to use any non-boolean values as operand(s) in logical operations. This includes unary (logical not/negation), binary (logical and, logical or), and the ternary operators

Please refer to the [Arango Errors](#) page for a list of error codes and meanings.

Operators

AQL supports a number of operators that can be used in expressions. There are comparison, logical, arithmetic, and the ternary operator.

Comparison operators

Comparison (or relational) operators compare two operands. They can be used with any input data types, and will return a boolean result value.

The following comparison operators are supported:

- $==$ equality
- $!=$ inequality
- $<$ less than
- $<=$ less or equal
- $>$ greater than
- $>=$ greater or equal
- *in* test if a value is contained in a list

The *in* operator expects the second operand to be of type list. All other operators accept any data types for the first and second operands.

Each of the comparison operators returns a boolean value if the comparison can be evaluated and returns *true* if the comparison evaluates to true, and *false* otherwise.

Some examples for comparison operations in AQL:

```
1 > 0
true != null
45 <= "yikes!"
65 != "65"
65 == 65
1.23 < 1.32
1.5 IN [ 2, 3, 1.5 ]
```

Logical operators

Logical operators combine two boolean operands in a logical operation and return a boolean result value.

The following logical operators are supported:

- *&&* logical and operator
- || logical or operator
- *!* logical not/negation operator

Some examples for logical operations in AQL:

```
u.age > 15 && u.address.city != ""
true || false
!u.isInvalid
```

The *&&*, ||, and *!* operators expect their input operands to be boolean values each. If a non-boolean operand is used, the operation will fail with an error. In case all operands are valid, the result of each logical operator is a boolean value.

Both the *&&* and *||* operators use short-circuit evaluation and only evaluate the second operand if the result of the operation cannot be determined by checking the first operand alone.

Arithmetic operators

Arithmetic operators perform an arithmetic operation on two numeric operands. The result of an arithmetic operation is again a numeric value. Operators are supported.

AQL supports the following arithmetic operators:

- *+* addition
- *-* subtraction
- *** multiplication
- */* division
- *%* modulus

These operators work with numeric operands only. Invoking any of the operators with non-numeric operands will result in an error. An error will also be raised for some other edge cases as division by zero, numeric over- or underflow etc. If both operands are numeric and the computation result is also valid, the result will be returned as a numeric value.

The unary plus and unary minus are supported as well.

Some example arithmetic operations:

```
1 + 1
33 - 99
12.4 * 4.5
13.0 / 0.1
23 % 7
-15
+9.99
```

Ternary operator

AQL also supports a ternary operator that can be used for conditional evaluation. The ternary operator expects a boolean condition as its first operand, and it returns the result of the second operand if the condition evaluates to true, and the third operand otherwise.

Examples

```
u.age > 15 || u.active == true ? u.userId : null
```

Range operator

AQL supports expressing simple numeric ranges with the .. operator. This operator can be used to easily iterate over a sequence of numeric values.

The .. operator will produce a list of values in the defined range, with both bounding values included.

Examples

```
2010..2013
```

will produce the following result:

```
[ 2010, 2011, 2012, 2013 ]
```

Operator precedence

The operator precedence in AQL is as follows (lowest precedence first):

- *? :* ternary operator
- *||* logical or
- *&&* logical and
- *==, !=* equality and inequality
- *in* in operator
- *<, <=, >=, >* less than, less equal, greater equal, greater than
- *+, -* addition, subtraction
- ***, /, %** multiplication, division, modulus
- *!, +, -* logical negation, unary plus, unary minus
- *[]** expansion
- *()* function call
- *.* member access
- *[]* indexed value access

The parentheses *(* and *)* can be used to enforce a different operator evaluation order.

High-level operations

FOR

The *FOR* keyword can be to iterate over all elements of a list. The general syntax is:

```
FOR variable-name IN expression
```

Each list element returned by *expression* is visited exactly once. It is required that *expression* returns a list in all cases. The empty list is allowed, too. The current list element is made available for further processing in the variable specified by *variable-name*.

```
FOR u IN users
   RETURN u
```

This will iterate over all elements from the list *users* (note: this list consists of all documents from the collection named "users" in this case) and make the current list element available in variable *u*. *u* is not modified in this example but simply pushed into the result using the *RETURN* keyword.

Note: When iterating over collection-based lists as shown here, the order of documents is undefined unless an explicit sort order is defined using a *SORT* statement.

The variable introduced by *FOR* is available until the scope the *FOR* is placed in is closed.

Another example that uses a statically declared list of values to iterate over:

```
FOR year IN [ 2011, 2012, 2013 ]
   RETURN { "year" : year, "isLeapYear" : year % 4 == 0 && (year %
100 != 0 || year % 400 == 0) }
```

Nesting of multiple *FOR* statements is allowed, too. When *FOR* statements are nested, a cross product of the list elements returned by the individual *FOR* statements will be created.

```
FOR u IN users
   FOR l IN locations
RETURN { "user" : u, "location" : l }
```

In this example, there are two list iterations: an outer iteration over the list *users* plus an inner iteration over the list *locations*. The inner list is traversed as many times as there are elements in the outer list. For each iteration, the current values of *users* and *locations* are made available for further processing in the variable *u* and *l*.

RETURN

The *RETURN* statement can (and must) be used to produce the result of a query. It is mandatory to specify a *RETURN* statement at the end of each block in a query, otherwise the query result would be undefined.

The general syntax for *return* is:

```
RETURN expression
```

The *expression* returned by *RETURN* is produced for each iteration the *RETURN* statement is placed in.

That means the result of a *RETURN* statement is always a list (this includes the empty list). To return all elements from the currently iterated list without modification, the following simple form can be used:

```
FOR variable-name IN expression
  RETURN variable-name
```

As *RETURN* allows specifying an expression, arbitrary computations can be performed to calculate the result elements. Any of the variables valid in the scope the *RETURN* is placed in can be used for the computations.

Note: Return will close the current scope and eliminate all local variables in it.

FILTER

The *FILTER* statement can be used to restrict the results to elements that match an arbitrary logical condition. The general syntax is:

```
FILTER condition
```

condition must be a condition that evaluates to either *false* or *true*. If the condition result is false, the current element is skipped, so it will not be processed further and not be part of the result. If the condition is true, the current element is not skipped and can be further processed.

```
FOR u IN users
  FILTER u.active == true && u.age < 39
  RETURN u
```

In the above example, all list elements from *users* will be included that have an attribute *active* with value *true* and that have an attribute *age* with a value less than *39*. All other elements from *users* will be skipped and not be included the result produced by *RETURN*.

It is allowed to specify multiple *FILTER* statements in a query, and even in the same block. If multiple *FILTER* statements are used, their results will be combined with a logical and, meaning all filter conditions must be true to include an element.

```
FOR u IN users
  FILTER u.active == true
  FILTER u.age < 39
  RETURN u
```

SORT

The *SORT* statement will force a sort of the list of already produced intermediate results in the current block. *SORT* allows specifying one or multiple sort criteria and directions. The general syntax is:

```
SORT expression direction
```

Specifying the *direction* is optional. The default (implicit) direction for a sort is the ascending order. To explicitly specify the sort direction, the keywords *ASC* (ascending) and *DESC* can be used. Multiple sort criteria can be separated using commas.

Note: when iterating over collection-based lists, the order of documents is always undefined unless an explicit sort order is defined using *SORT*.

```
FOR u IN users
   SORT u.lastName, u.firstName, u.id DESC
   RETURN u
```

LIMIT

The *LIMIT* statement allows slicing the list of result documents using an offset and a count. It reduces the number of elements in the result to at most the specified number. Two general forms of *LIMIT* are followed:

```
LIMIT count
LIMIT offset, count
```

The first form allows specifying only the *count* value whereas the second form allows specifying both *offset* and *count*. The first form is identical using the second form with an *offset* value of *0*.

The *offset* value specifies how many elements from the result shall be discarded. It must be 0 or greater. The *count* value specifies how many elements should be at most included in the result.

```
FOR u IN users
   SORT u.firstName, u.lastName, u.id DESC
   LIMIT 0, 5
   RETURN u
```

LET

The *LET* statement can be used to assign an arbitrary value to a variable. The variable is then introduced in the scope the *LET* statement is placed in. The general syntax is:

```
LET variable-name = expression
```

LET statements are mostly used to declare complex computations and to avoid repeated computations of the same value at multiple parts of a query.

```
FOR u IN users
   LET numRecommendations = LENGTH(u.recommendations)
   RETURN { "user" : u, "numRecommendations" : numRecommendations,
"isPowerUser" : numRecommendations >= 10 }
```

In the above example, the computation of the number of recommendations is factored out using a *LET* statement, thus avoiding computing the value twice in the *RETURN* statement.

Another use case for *LET* is to declare a complex computation in a subquery, making the whole query more readable.

```
FOR u IN users
   LET friends = (
FOR f IN friends
   FILTER u.id == f.userId
   RETURN f
   )
   LET memberships = (
```

```
FOR m IN memberships
   FILTER u.id == m.userId
   RETURN m
   )
   RETURN { "user" : u, "friends" : friends, "numFriends" :
LENGTH(friends), "memberShips" : memberships }
```

COLLECT

The *COLLECT* keyword can be used to group a list by one or multiple group criteria. The two general syntaxes for *COLLECT* are:

```
COLLECT variable-name = expression
COLLECT variable-name = expression INTO groups
```

The first form only groups the result by the defined group criteria defined by *expression*. In order to further process the results produced by *COLLECT*, a new variable (specified by *variable-name*) is introduced. This variable contains the group value.

The second form does the same as the first form, but additionally introduces a variable (specified by *groups*) that contains all elements that fell into the group. Specifying the *INTO* clause is optional-

```
FOR u IN users
   COLLECT city = u.city INTO g
   RETURN { "city" : city, "users" : g }
```

In the above example, the list of *users* will be grouped by the attribute *city*. The result is a new list of documents, with one element per distinct *city* value. The elements from the original list (here: *users*) per city are made available in the variable *g*. This is due to the *INTO* clause.

COLLECT also allows specifying multiple group criteria. Individual group criteria can be separated by commas.

```
FOR u IN users
   COLLECT first = u.firstName, age = u.age INTO g
   RETURN { "first" : first, "age" : age, "numUsers" : LENGTH(g) }
```

In the above example, the list of *users* is grouped by first names and ages first, and for each distinct combination of first name and age, the number of users found is returned.

Note: The *COLLECT* statement eliminates all local variables in the current scope. After *COLLECT* only the variables introduced by *COLLECT* itself are available.

Graph operations

This chapter describes graph related AQL functions.

Short explanation of the example parameter

A lot of the following functions accept a vertex (or edge) example as parameter. This can contain the following:

- {} : Returns all possible vertices for this graph
- *idString* : Returns the vertex/edge with the id *idString*
- {*key1* : *value1*, *key2* : *value2*} : Returns the vertices/edges that match this example, which means that both have *key1* and *key2* with the corresponding attributes
- {*key1.key2* : *value1*, *key3* : *value2*} : It is possible to chain keys which means that a document *{ key1 : {key2 : value1}, key3 : value2}* would be a match
- [{*key1* : *value1*}, {*key2* : *value2*}] : Returns the vertices/edges that match one of the examples, which means that either *key1* or *key2* are set with the corresponding value

The complexity of the shortest path algorithms

Most of the functions described in this chapter calculate the shortest paths for subsets of the graphs vertices. Hence the complexity of these functions depends of the chosen algorithm for this task. For Floyd-Warshall it is **O(n^3)** with *n* being the amount of vertices in the graph. For Dijkstra it would be **O(x*y*n^2)** with *n* being the amount of vertices in the graph, *x* the amount of start vertices and *y* the amount of target vertices. Hence a suggestion might be to use Dijkstra when x*y < n and the functions supports choosing your algorithm.

Edges and Vertices related functions

This section describes various AQL functions which can be used to receive information about the graph's vertices, edges, neighbor relationship and shared properties.

GRAPH_EDGES

```
GRAPH_EDGES (graphName, vertexExample, options)
```

The GRAPH_EDGES function returns all edges of the graph connected to the vertices defined by the example.

The complexity of this method is **O(n*m^x)** with *n* being the vertices defined by the parameter vertexExamplex, *m* the average amount of edges of a vertex and *x* the maximal depths. Hence the default call would have a complexity of **O(n*m)**;

Parameters

- *graphName* : The name of the graph as a string.
- *vertexExample* : An example for the desired vertices (see example).
- *options* (optional) : An object containing the following options::
  - *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or multiple edge collection names. Only edges from these collections will be considered for the path.
  - *startVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as start vertex of a path.
  - *endVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as end vertex of a path.
  - *edgeExamples* : A filter example for the edges (see example).
  - *neighborExamples* : An example for the desired neighbors (see example).

- ○ *minDepth* : Defines the minimal length of a path from an edge to a vertex (default is 1, which means only the edges directly connected to a vertex would be returned).
- ○ *maxDepth* : Defines the maximal length of a path from an edge to a vertex (default is 1, which means only the edges directly connected to a vertex would be returned).

Examples

A route planner example, all edges to locations with a distance of either 700 or 600.:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_EDGES("
........> +"'routeplanner', {}, {edgeExamples : [{distance: 600},
{distance: 700}]}) RETURN e"
........> ).toArray();
[
  {
    "_id" : "internationalHighway/431448757",
    "_rev" : "431448757",
    "_key" : "431448757",
    "_from" : "germanCity/Cologne",
    "_to" : "frenchCity/Lyon",
    "distance" : 700
  },
  {
    "_id" : "internationalHighway/431448757",
    "_rev" : "431448757",
    "_key" : "431448757",
    "_from" : "germanCity/Cologne",
    "_to" : "frenchCity/Lyon",
    "distance" : 700
  }
]
```

A route planner example, all outbound edges of Hamburg with a maximal depth of 2 :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_EDGES("
........> +"'routeplanner', 'germanCity/Hamburg', {direction :
'outbound', maxDepth : 2}) RETURN e"
........> ).toArray();
[
  {
    "_id" : "germanHighway/436101813",
    "_rev" : "436101813",
    "_key" : "436101813",
    "_from" : "germanCity/Hamburg",
    "_to" : "germanCity/Cologne",
    "distance" : 500
  },
  {
    "_id" : "germanHighway/436101813",
    "_rev" : "436101813",
    "_key" : "436101813",
    "_from" : "germanCity/Hamburg",
```

```
      "_to" : "germanCity/Cologne",
      "distance" : 500
    },
    {
      "_id" : "internationalHighway/437543605",
      "_rev" : "437543605",
      "_key" : "437543605",
      "_from" : "germanCity/Cologne",
      "_to" : "frenchCity/Lyon",
      "distance" : 700
    },
    {
      "_id" : "germanHighway/436101813",
      "_rev" : "436101813",
      "_key" : "436101813",
      "_from" : "germanCity/Hamburg",
      "_to" : "germanCity/Cologne",
      "distance" : 500
    },
    {
      "_id" : "internationalHighway/437740213",
      "_rev" : "437740213",
      "_key" : "437740213",
      "_from" : "germanCity/Cologne",
      "_to" : "frenchCity/Paris",
      "distance" : 550
    },
    {
      "_id" : "internationalHighway/437150389",
      "_rev" : "437150389",
      "_key" : "437150389",
      "_from" : "germanCity/Hamburg",
      "_to" : "frenchCity/Paris",
      "distance" : 900
    },
    {
      "_id" : "internationalHighway/437150389",
      "_rev" : "437150389",
      "_key" : "437150389",
      "_from" : "germanCity/Hamburg",
      "_to" : "frenchCity/Paris",
      "distance" : 900
    },
    {
      "_id" : "frenchHighway/436429493",
      "_rev" : "436429493",
      "_key" : "436429493",
      "_from" : "frenchCity/Paris",
      "_to" : "frenchCity/Lyon",
      "distance" : 550
    },
    {
      "_id" : "internationalHighway/437346997",
      "_rev" : "437346997",
      "_key" : "437346997",
      "_from" : "germanCity/Hamburg",
      "_to" : "frenchCity/Lyon",
      "distance" : 1300
    }
]
```

GRAPH_VERTICES

The GRAPH_VERTICES function returns all vertices.

```
GRAPH_VERTICES (graphName, vertexExample, options)
```

According to the optional filters it will only return vertices that have outbound, inbound or any (default) edges.

Parameters

- *graphName* : The name of the graph as a string.
- *vertexExample* : An example for the desired vertices (see [example](#)).
- *options* (optional) : An object containing the following options::
  - *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
  - *vertexCollectionRestriction* : One or multiple vertex collections that should be considered.

Examples

A route planner example, all vertices of the graph

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_VERTICES("
........> +"'routeplanner', {}) RETURN e").toArray();
[
  {
    "_id" : "germanCity/Berlin",
    "_rev" : "405693109",
    "_key" : "Berlin",
    "isCapital" : true,
    "population" : 3000000
  },
  {
    "_id" : "germanCity/Hamburg",
    "_rev" : "406086325",
    "_key" : "Hamburg",
    "isCapital" : false,
    "population" : 1000000
  },
  {
    "_id" : "germanCity/Cologne",
    "_rev" : "405889717",
    "_key" : "Cologne",
    "isCapital" : false,
    "population" : 1000000
  },
  {
    "_id" : "frenchCity/Paris",
    "_rev" : "406676149",
    "_key" : "Paris",
    "isCapital" : true,
    "population" : 4000000
  },
  {
```

```
      "_id" : "frenchCity/Lyon",
      "_rev" : "406479541",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    }
]
```

A route planner example, all vertices from collection *germanCity*.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_VERTICES("
........> +"'routeplanner', {}, {direction : 'any',
vertexCollectionRestriction" +
........> " : 'germanCity'}) RETURN e").toArray();
[
    {
      "_id" : "germanCity/Berlin",
      "_rev" : "399008437",
      "_key" : "Berlin",
      "isCapital" : true,
      "population" : 3000000
    },
    {
      "_id" : "germanCity/Hamburg",
      "_rev" : "399401653",
      "_key" : "Hamburg",
      "isCapital" : false,
      "population" : 1000000
    },
    {
      "_id" : "germanCity/Cologne",
      "_rev" : "399205045",
      "_key" : "Cologne",
      "isCapital" : false,
      "population" : 1000000
    }
]
```

GRAPH_NEIGHBORS

The GRAPH_NEIGHBORS function returns all neighbors of vertices.

```
GRAPH_NEIGHBORS (graphName, vertexExample, options)
```

By default only the direct neighbors (path length equals 1) are returned, but one can define the range of the path length to the neighbors with the options *minDepth* and *maxDepth*. The complexity of this method is $O(n*m^x)$ with *n* being the vertices defined by the parameter vertexExamplex, *m* the average amount of neighbors and *x* the maximal depths. Hence the default call would have a complexity of $O(n*m)$;

Parameters

- *graphName* : The name of the graph as a string.
- *vertexExample* : An example for the desired vertices (see example).
- *options* : An object containing the following options::
    - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any*

(default).

- *edgeExamples* : A filter example for the edges to the neighbors (see [example](#)).
- *neighborExamples* : An example for the desired neighbors (see [example](#)).
- *edgeCollectionRestriction* : One or multiple edge collection names. Only edges from these collections will be considered for the path.
- *vertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as neighbor.
- *minDepth* : Defines the minimal depth a path to a neighbor must have to be returned (default is 1).
- *maxDepth* : Defines the maximal depth a path to a neighbor must have to be returned (default is 1).

Examples

A route planner example, all neighbors of locations with a distance of either 700 or 600.:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_NEIGHBORS("
........> +"'routeplanner', {}, {edgeExamples : [{distance: 600},
{distance: 700}]}) RETURN e"
........> ).toArray();
[
  {
    "vertex" : {
      "_id" : "frenchCity/Lyon",
      "_rev" : "337994421",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    "path" : {
      "edges" : [
        {
          "_id" : "internationalHighway/340353717",
          "_rev" : "340353717",
          "_key" : "340353717",
          "_from" : "germanCity/Cologne",
          "_to" : "frenchCity/Lyon",
          "distance" : 700
        }
      ],
      "vertices" : [
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "337404597",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "337994421",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
```

```
        },
        "startVertex" : "germanCity/Cologne"
    },
    {
        "vertex" : {
          "_id" : "germanCity/Cologne",
          "_rev" : "337404597",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        "path" : {
          "edges" : [
            {
              "_id" : "internationalHighway/340353717",
              "_rev" : "340353717",
              "_key" : "340353717",
              "_from" : "germanCity/Cologne",
              "_to" : "frenchCity/Lyon",
              "distance" : 700
            }
          ],
          "vertices" : [
            {
              "_id" : "frenchCity/Lyon",
              "_rev" : "337994421",
              "_key" : "Lyon",
              "isCapital" : false,
              "population" : 80000
            },
            {
              "_id" : "germanCity/Cologne",
              "_rev" : "337404597",
              "_key" : "Cologne",
              "isCapital" : false,
              "population" : 1000000
            }
          ]
        },
        "startVertex" : "frenchCity/Lyon"
    }
]
```

A route planner example, all outbound neighbors of Hamburg with a maximal depth of 2 :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_NEIGHBORS("
........> +"'routeplanner', 'germanCity/Hamburg', {direction :
'outbound', maxDepth : 2}) RETURN e"
........> ).toArray();
[
  {
    "vertex" : {
      "_id" : "germanCity/Cologne",
      "_rev" : "238052021",
      "_key" : "Cologne",
```

```
        "isCapital" : false,
        "population" : 1000000
      },
      "path" : {
        "edges" : [
          {
            "_id" : "germanHighway/239559349",
            "_rev" : "239559349",
            "_key" : "239559349",
            "_from" : "germanCity/Hamburg",
            "_to" : "germanCity/Cologne",
            "distance" : 500
          }
        ],
        "vertices" : [
          {
            "_id" : "germanCity/Hamburg",
            "_rev" : "238248629",
            "_key" : "Hamburg",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "germanCity/Cologne",
            "_rev" : "238052021",
            "_key" : "Cologne",
            "isCapital" : false,
            "population" : 1000000
          }
        ]
      },
      "startVertex" : "germanCity/Hamburg"
    },
    {
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "238641845",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      },
      "path" : {
        "edges" : [
          {
            "_id" : "germanHighway/239559349",
            "_rev" : "239559349",
            "_key" : "239559349",
            "_from" : "germanCity/Hamburg",
            "_to" : "germanCity/Cologne",
            "distance" : 500
          },
          {
            "_id" : "internationalHighway/241001141",
            "_rev" : "241001141",
            "_key" : "241001141",
            "_from" : "germanCity/Cologne",
            "_to" : "frenchCity/Lyon",
            "distance" : 700
          }
        ],
```

```
      "vertices" : [
        {
          "_id" : "germanCity/Hamburg",
          "_rev" : "238248629",
          "_key" : "Hamburg",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "238052021",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "238641845",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
    },
    "startVertex" : "germanCity/Hamburg"
  },
  {
    "vertex" : {
      "_id" : "frenchCity/Paris",
      "_rev" : "238838453",
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000
    },
    "path" : {
      "edges" : [
        {
          "_id" : "germanHighway/239559349",
          "_rev" : "239559349",
          "_key" : "239559349",
          "_from" : "germanCity/Hamburg",
          "_to" : "germanCity/Cologne",
          "distance" : 500
        },
        {
          "_id" : "internationalHighway/241197749",
          "_rev" : "241197749",
          "_key" : "241197749",
          "_from" : "germanCity/Cologne",
          "_to" : "frenchCity/Paris",
          "distance" : 550
        }
      ],
      "vertices" : [
        {
          "_id" : "germanCity/Hamburg",
          "_rev" : "238248629",
          "_key" : "Hamburg",
          "isCapital" : false,
          "population" : 1000000
```

```
        },
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "238052021",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "238838453",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      ]
    },
    "startVertex" : "germanCity/Hamburg"
  },
  {
    "vertex" : {
      "_id" : "frenchCity/Paris",
      "_rev" : "238838453",
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000
    },
    "path" : {
      "edges" : [
        {
          "_id" : "internationalHighway/240607925",
          "_rev" : "240607925",
          "_key" : "240607925",
          "_from" : "germanCity/Hamburg",
          "_to" : "frenchCity/Paris",
          "distance" : 900
        }
      ],
      "vertices" : [
        {
          "_id" : "germanCity/Hamburg",
          "_rev" : "238248629",
          "_key" : "Hamburg",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "238838453",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      ]
    },
    "startVertex" : "germanCity/Hamburg"
  },
  {
    "vertex" : {
      "_id" : "frenchCity/Lyon",
```

```
      "_rev" : "238641845",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    "path" : {
      "edges" : [
        {
          "_id" : "internationalHighway/240607925",
          "_rev" : "240607925",
          "_key" : "240607925",
          "_from" : "germanCity/Hamburg",
          "_to" : "frenchCity/Paris",
          "distance" : 900
        },
        {
          "_id" : "frenchHighway/239887029",
          "_rev" : "239887029",
          "_key" : "239887029",
          "_from" : "frenchCity/Paris",
          "_to" : "frenchCity/Lyon",
          "distance" : 550
        }
      ],
      "vertices" : [
        {
          "_id" : "germanCity/Hamburg",
          "_rev" : "238248629",
          "_key" : "Hamburg",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "238838453",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "238641845",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
    },
    "startVertex" : "germanCity/Hamburg"
  },
  {
    "vertex" : {
      "_id" : "frenchCity/Lyon",
      "_rev" : "238641845",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    "path" : {
      "edges" : [
```

```
            {
              "_id" : "internationalHighway/240804533",
              "_rev" : "240804533",
              "_key" : "240804533",
              "_from" : "germanCity/Hamburg",
              "_to" : "frenchCity/Lyon",
              "distance" : 1300
            }
          ],
          "vertices" : [
            {
              "_id" : "germanCity/Hamburg",
              "_rev" : "238248629",
              "_key" : "Hamburg",
              "isCapital" : false,
              "population" : 1000000
            },
            {
              "_id" : "frenchCity/Lyon",
              "_rev" : "238641845",
              "_key" : "Lyon",
              "isCapital" : false,
              "population" : 80000
            }
          ]
        },
        "startVertex" : "germanCity/Hamburg"
      }
  ]
```

GRAPH_COMMON_NEIGHBORS

The GRAPH_COMMON_NEIGHBORS function returns all common neighbors of the vertices defined by the examples.

```
GRAPH_COMMON_NEIGHBORS (graphName, vertex1Example, vertex2Examples,
optionsVertex1, optionsVertex2)
```

This function returns the intersection of *GRAPH_NEIGHBORS(vertex1Example, optionsVertex1)* and *GRAPH_NEIGHBORS(vertex2Example, optionsVertex2)*. The complexity of this method is **O(n*m^x)** with *n* being the maximal amount of vertices defined by the parameters vertexExamples, *m* the average amount of neighbors and *x* the maximal depths. Hence the default call would have a complexity of **O(n*m)**;

For parameter documentation read the documentation of [GRAPH_NEIGHBORS](#).

Examples

A route planner example, all common neighbors of capitals.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_COMMON_NEIGHBORS("
........> +"'routeplanner', {isCapital : true}, {isCapital : true})
RETURN e"
........> ).toArray();
[
  {
```

```
      "germanCity/Berlin" : {
        "frenchCity/Paris" : [
          {
            "_id" : "germanCity/Cologne",
            "_rev" : "696345269",
            "_key" : "Cologne",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "germanCity/Hamburg",
            "_rev" : "696541877",
            "_key" : "Hamburg",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "frenchCity/Lyon",
            "_rev" : "696935093",
            "_key" : "Lyon",
            "isCapital" : false,
            "population" : 80000
          }
        ]
      }
    },
    {
      "frenchCity/Paris" : {
        "germanCity/Berlin" : [
          {
            "_id" : "germanCity/Cologne",
            "_rev" : "696345269",
            "_key" : "Cologne",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "germanCity/Hamburg",
            "_rev" : "696541877",
            "_key" : "Hamburg",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "frenchCity/Lyon",
            "_rev" : "696935093",
            "_key" : "Lyon",
            "isCapital" : false,
            "population" : 80000
          }
        ]
      }
    }
  ]
```

A route planner example, all common outbound neighbors of Hamburg with any other location which have a maximal depth of 2 :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_COMMON_NEIGHBORS("
........> +"'routeplanner', 'germanCity/Hamburg', {}, {direction :
'outbound', maxDepth : 2}, "+
........> "{direction : 'outbound', maxDepth : 2}) RETURN e"
........> ).toArray();
[
  {
    "germanCity/Hamburg" : {
      "germanCity/Berlin" : [
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "784032437",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "784622261",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "784818869",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      ],
      "germanCity/Cologne" : [
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "784622261",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "784818869",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      ],
      "frenchCity/Paris" : [
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "784622261",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
```

```
        }
    }
]
```

GRAPH_COMMON_PROPERTIES

```
GRAPH_COMMON_PROPERTIES (graphName, vertex1Example, vertex2Examples,
options)
```

The GRAPH_COMMON_PROPERTIES function returns a list of objects which have the id of the vertices defined by *vertex1Example* as keys and a list of vertices defined by *vertex21Example*, that share common properties as value. Notice that only the vertex id and the matching attributes are returned in the result.

The complexity of this method is **O(n)** with *n* being the maximal amount of vertices defined by the parameters vertexExamples.

Parameters

- *graphName* : The name of the graph as a string.
- *vertex1Example* : An example for the desired vertices (see example).
- *vertex2Example* : An example for the desired vertices (see example).
- *options* (optional) : An object containing the following options::
  - *vertex1CollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered.
  - *vertex2CollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered.
  - *ignoreProperties* : One or multiple attributes of a document that should be ignored, either a string or an array..

Examples

A route planner example, all locations with the same properties:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_COMMON_PROPERTIES("
........> +"'routeplanner', {}, {}) RETURN e"
........> ).toArray();
[
  {
    "germanCity/Berlin" : [
      {
        "_id" : "frenchCity/Paris",
        "isCapital" : true
      }
    ]
  },
  {
    "frenchCity/Paris" : [
      {
        "_id" : "germanCity/Berlin",
        "isCapital" : true
      }
    ]
  },
  {
    "germanCity/Hamburg" : [
      {
```

```
        "_id" : "germanCity/Cologne",
        "isCapital" : false,
        "population" : 1000000
      },
      {
        "_id" : "frenchCity/Lyon",
        "isCapital" : false
      }
    ]
  },
  {
    "germanCity/Cologne" : [
      {
        "_id" : "germanCity/Hamburg",
        "isCapital" : false,
        "population" : 1000000
      },
      {
        "_id" : "frenchCity/Lyon",
        "isCapital" : false
      }
    ]
  },
  {
    "frenchCity/Lyon" : [
      {
        "_id" : "germanCity/Hamburg",
        "isCapital" : false
      },
      {
        "_id" : "germanCity/Cologne",
        "isCapital" : false
      }
    ]
  }
]
```

A route planner example, all cities which share same properties except for population.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_COMMON_PROPERTIES("
........> +"'routeplanner', {}, {}, {ignoreProperties:
'population'}) RETURN e"
........> ).toArray();
[
  {
    "germanCity/Berlin" : [
      {
        "_id" : "frenchCity/Paris",
        "isCapital" : true
      }
    ]
  },
  {
    "frenchCity/Paris" : [
      {
```

```
            "_id" : "germanCity/Berlin",
            "isCapital" : true
          }
        ]
    },
    {
      "germanCity/Hamburg" : [
          {
            "_id" : "germanCity/Cologne",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "frenchCity/Lyon",
            "isCapital" : false
          }
        ]
    },
    {
      "germanCity/Cologne" : [
          {
            "_id" : "germanCity/Hamburg",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "frenchCity/Lyon",
            "isCapital" : false
          }
        ]
    },
    {
      "frenchCity/Lyon" : [
          {
            "_id" : "germanCity/Hamburg",
            "isCapital" : false
          },
          {
            "_id" : "germanCity/Cologne",
            "isCapital" : false
          }
        ]
    }
]
```

Shortest Paths, distances and traversals.

This section describes AQL functions, that calculate pathes from a subset of vertices in a graph to another subset of vertices.

GRAPH_PATHS

The GRAPH_PATHS function returns all paths of a graph.

```
GRAPH_PATHS (graphName, options)
```

The complexity of this method is **O(n*n*m)** with *n* being the amount of vertices in the graph and *m* the average amount of connected edges;

Parameters

- *graphName* : The name of the graph as a string.
- *options* : An object containing the following options::
  - *direction* : The direction of the edges. Possible values are *any, inbound* and *outbound* (default).
  - *followCycles* (optional) : If set to *true* the query follows cycles in the graph, default is false.
  - *minLength* (optional) : Defines the minimal length a path must have to be returned (default is 0).
  - *maxLength* (optional) : Defines the maximal length a path must have to be returned (default is 10).

Examples

Return all paths of the graph "social":

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("social");
arangosh> db._query("RETURN GRAPH_PATHS('social')").toArray();
[
  [
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "231629493",
          "_key" : "diana",
          "name" : "Diana"
        }
      ],
      "edges" : [ ],
      "source" : {
        "_id" : "female/diana",
        "_rev" : "231629493",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
        "_id" : "female/diana",
        "_rev" : "231629493",
        "_key" : "diana",
        "name" : "Diana"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "female/alice",
          "_rev" : "230908597",
          "_key" : "alice",
          "name" : "Alice"
        }
      ],
      "edges" : [ ],
      "source" : {
        "_id" : "female/alice",
        "_rev" : "230908597",
```

```
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "female/alice",
          "_rev" : "230908597",
          "_key" : "alice",
          "name" : "Alice"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "female/alice",
            "_rev" : "230908597",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/bob",
            "_rev" : "231236277",
            "_key" : "bob",
            "name" : "Bob"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndBob",
            "_rev" : "231957173",
            "_key" : "aliceAndBob",
            "_from" : "female/alice",
            "_to" : "male/bob",
            "type" : "married"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "230908597",
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "male/bob",
          "_rev" : "231236277",
          "_key" : "bob",
          "name" : "Bob"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "female/alice",
            "_rev" : "230908597",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/bob",
            "_rev" : "231236277",
            "_key" : "bob",
```

```
            "name" : "Bob"
          },
          {
            "_id" : "female/diana",
            "_rev" : "231629493",
            "_key" : "diana",
            "name" : "Diana"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndBob",
            "_rev" : "231957173",
            "_key" : "aliceAndBob",
            "_from" : "female/alice",
            "_to" : "male/bob",
            "type" : "married"
          },
          {
            "_id" : "relation/bobAndDiana",
            "_rev" : "232612533",
            "_key" : "bobAndDiana",
            "_from" : "male/bob",
            "_to" : "female/diana",
            "type" : "friend"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "230908597",
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "female/diana",
          "_rev" : "231629493",
          "_key" : "diana",
          "name" : "Diana"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "female/alice",
            "_rev" : "230908597",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/charly",
            "_rev" : "231432885",
            "_key" : "charly",
            "name" : "Charly"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndCharly",
            "_rev" : "232219317",
            "_key" : "aliceAndCharly",
```

```
            "_from" : "female/alice",
            "_to" : "male/charly",
            "type" : "friend"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "230908597",
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "male/charly",
          "_rev" : "231432885",
          "_key" : "charly",
          "name" : "Charly"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "female/alice",
            "_rev" : "230908597",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/charly",
            "_rev" : "231432885",
            "_key" : "charly",
            "name" : "Charly"
          },
          {
            "_id" : "female/diana",
            "_rev" : "231629493",
            "_key" : "diana",
            "name" : "Diana"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndCharly",
            "_rev" : "232219317",
            "_key" : "aliceAndCharly",
            "_from" : "female/alice",
            "_to" : "male/charly",
            "type" : "friend"
          },
          {
            "_id" : "relation/charlyAndDiana",
            "_rev" : "232415925",
            "_key" : "charlyAndDiana",
            "_from" : "male/charly",
            "_to" : "female/diana",
            "type" : "married"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "230908597",
```

```
        "_key" : "alice",
        "name" : "Alice"
      },
      "destination" : {
        "_id" : "female/diana",
        "_rev" : "231629493",
        "_key" : "diana",
        "name" : "Diana"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "male/bob",
          "_rev" : "231236277",
          "_key" : "bob",
          "name" : "Bob"
        }
      ],
      "edges" : [ ],
      "source" : {
        "_id" : "male/bob",
        "_rev" : "231236277",
        "_key" : "bob",
        "name" : "Bob"
      },
      "destination" : {
        "_id" : "male/bob",
        "_rev" : "231236277",
        "_key" : "bob",
        "name" : "Bob"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "male/bob",
          "_rev" : "231236277",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "female/diana",
          "_rev" : "231629493",
          "_key" : "diana",
          "name" : "Diana"
        }
      ],
      "edges" : [
        {
          "_id" : "relation/bobAndDiana",
          "_rev" : "232612533",
          "_key" : "bobAndDiana",
          "_from" : "male/bob",
          "_to" : "female/diana",
          "type" : "friend"
        }
      ],
      "source" : {
        "_id" : "male/bob",
```

```
      "_rev" : "231236277",
      "_key" : "bob",
      "name" : "Bob"
    },
    "destination" : {
      "_id" : "female/diana",
      "_rev" : "231629493",
      "_key" : "diana",
      "name" : "Diana"
    }
  },
  {
    "vertices" : [
      {
        "_id" : "male/charly",
        "_rev" : "231432885",
        "_key" : "charly",
        "name" : "Charly"
      }
    ],
    "edges" : [ ],
    "source" : {
      "_id" : "male/charly",
      "_rev" : "231432885",
      "_key" : "charly",
      "name" : "Charly"
    },
    "destination" : {
      "_id" : "male/charly",
      "_rev" : "231432885",
      "_key" : "charly",
      "name" : "Charly"
    }
  },
  {
    "vertices" : [
      {
        "_id" : "male/charly",
        "_rev" : "231432885",
        "_key" : "charly",
        "name" : "Charly"
      },
      {
        "_id" : "female/diana",
        "_rev" : "231629493",
        "_key" : "diana",
        "name" : "Diana"
      }
    ],
    "edges" : [
      {
        "_id" : "relation/charlyAndDiana",
        "_rev" : "232415925",
        "_key" : "charlyAndDiana",
        "_from" : "male/charly",
        "_to" : "female/diana",
        "type" : "married"
      }
    ],
    "source" : {
```

```
      "_id" : "male/charly",
      "_rev" : "231432885",
      "_key" : "charly",
      "name" : "Charly"
    },
    "destination" : {
      "_id" : "female/diana",
      "_rev" : "231629493",
      "_key" : "diana",
      "name" : "Diana"
    }
  }
  ]
]
```

Return all inbound paths of the graph "social" with a maximal length of 1 and a minimal length of 2:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("social");
arangosh> db._query(
........> "RETURN GRAPH_PATHS('social', {direction : 'inbound',
minLength : 1, maxLength :  2})"
........> ).toArray();
[
  [
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "557408949",
          "_key" : "diana",
          "name" : "Diana"
        },
        {
          "_id" : "male/charly",
          "_rev" : "557212341",
          "_key" : "charly",
          "name" : "Charly"
        }
      ],
      "edges" : [
        {
          "_id" : "relation/charlyAndDiana",
          "_rev" : "558195381",
          "_key" : "charlyAndDiana",
          "_from" : "male/charly",
          "_to" : "female/diana",
          "type" : "married"
        }
      ],
      "source" : {
        "_id" : "female/diana",
        "_rev" : "557408949",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
```

```
        "_id" : "male/charly",
        "_rev" : "557212341",
        "_key" : "charly",
        "name" : "Charly"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "557408949",
          "_key" : "diana",
          "name" : "Diana"
        },
        {
          "_id" : "male/charly",
          "_rev" : "557212341",
          "_key" : "charly",
          "name" : "Charly"
        },
        {
          "_id" : "female/alice",
          "_rev" : "556688053",
          "_key" : "alice",
          "name" : "Alice"
        }
      ],
      "edges" : [
        {
          "_id" : "relation/charlyAndDiana",
          "_rev" : "558195381",
          "_key" : "charlyAndDiana",
          "_from" : "male/charly",
          "_to" : "female/diana",
          "type" : "married"
        },
        {
          "_id" : "relation/aliceAndCharly",
          "_rev" : "557998773",
          "_key" : "aliceAndCharly",
          "_from" : "female/alice",
          "_to" : "male/charly",
          "type" : "friend"
        }
      ],
      "source" : {
        "_id" : "female/diana",
        "_rev" : "557408949",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
        "_id" : "female/alice",
        "_rev" : "556688053",
        "_key" : "alice",
        "name" : "Alice"
      }
    },
    {
      "vertices" : [
```

```
    {
      "_id" : "female/diana",
      "_rev" : "557408949",
      "_key" : "diana",
      "name" : "Diana"
    },
    {
      "_id" : "male/bob",
      "_rev" : "557015733",
      "_key" : "bob",
      "name" : "Bob"
    }
  ],
  "edges" : [
    {
      "_id" : "relation/bobAndDiana",
      "_rev" : "558391989",
      "_key" : "bobAndDiana",
      "_from" : "male/bob",
      "_to" : "female/diana",
      "type" : "friend"
    }
  ],
  "source" : {
    "_id" : "female/diana",
    "_rev" : "557408949",
    "_key" : "diana",
    "name" : "Diana"
  },
  "destination" : {
    "_id" : "male/bob",
    "_rev" : "557015733",
    "_key" : "bob",
    "name" : "Bob"
  }
},
{
  "vertices" : [
    {
      "_id" : "female/diana",
      "_rev" : "557408949",
      "_key" : "diana",
      "name" : "Diana"
    },
    {
      "_id" : "male/bob",
      "_rev" : "557015733",
      "_key" : "bob",
      "name" : "Bob"
    },
    {
      "_id" : "female/alice",
      "_rev" : "556688053",
      "_key" : "alice",
      "name" : "Alice"
    }
  ],
  "edges" : [
    {
      "_id" : "relation/bobAndDiana",
```

```
      "_rev" : "558391989",
      "_key" : "bobAndDiana",
      "_from" : "male/bob",
      "_to" : "female/diana",
      "type" : "friend"
    },
    {
      "_id" : "relation/aliceAndBob",
      "_rev" : "557736629",
      "_key" : "aliceAndBob",
      "_from" : "female/alice",
      "_to" : "male/bob",
      "type" : "married"
    }
  ],
  "source" : {
    "_id" : "female/diana",
    "_rev" : "557408949",
    "_key" : "diana",
    "name" : "Diana"
  },
  "destination" : {
    "_id" : "female/alice",
    "_rev" : "556688053",
    "_key" : "alice",
    "name" : "Alice"
  }
},
{
  "vertices" : [
    {
      "_id" : "male/bob",
      "_rev" : "557015733",
      "_key" : "bob",
      "name" : "Bob"
    },
    {
      "_id" : "female/alice",
      "_rev" : "556688053",
      "_key" : "alice",
      "name" : "Alice"
    }
  ],
  "edges" : [
    {
      "_id" : "relation/aliceAndBob",
      "_rev" : "557736629",
      "_key" : "aliceAndBob",
      "_from" : "female/alice",
      "_to" : "male/bob",
      "type" : "married"
    }
  ],
  "source" : {
    "_id" : "male/bob",
    "_rev" : "557015733",
    "_key" : "bob",
    "name" : "Bob"
  },
  "destination" : {
```

```
            "_id" : "female/alice",
            "_rev" : "556688053",
            "_key" : "alice",
            "name" : "Alice"
          }
        },
        {
          "vertices" : [
            {
              "_id" : "male/charly",
              "_rev" : "557212341",
              "_key" : "charly",
              "name" : "Charly"
            },
            {
              "_id" : "female/alice",
              "_rev" : "556688053",
              "_key" : "alice",
              "name" : "Alice"
            }
          ],
          "edges" : [
            {
              "_id" : "relation/aliceAndCharly",
              "_rev" : "557998773",
              "_key" : "aliceAndCharly",
              "_from" : "female/alice",
              "_to" : "male/charly",
              "type" : "friend"
            }
          ],
          "source" : {
            "_id" : "male/charly",
            "_rev" : "557212341",
            "_key" : "charly",
            "name" : "Charly"
          },
          "destination" : {
            "_id" : "female/alice",
            "_rev" : "556688053",
            "_key" : "alice",
            "name" : "Alice"
          }
        }
      ]
  ]
```

GRAPH_SHORTEST_PATH

The GRAPH_SHORTEST_PATH function returns all shortest paths of a graph.

```
GRAPH_SHORTEST_PATH (graphName, startVertexExample, endVertexExample,
options)
```

This function determines all shortest paths in a graph identified by *graphName*. If one wants to call this function to receive nearly all shortest paths for a graph the option *algorithm* should be set to Floyd-Warshall to increase performance. If no algorithm is provided in the options the function chooses the appropriate one (either Floyd-Warshall or Dijkstra) according to its parameters. The length of a path is by default the amount of edges from one start vertex to an end vertex. The option weight allows the user to

define an edge attribute representing the length.

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *startVertexExample* : An example for the desired start Vertices (see [example](#)).
- *endVertexExample* : An example for the desired end Vertices (see [example](#)).
- *options* (optional) : An object containing the following options::
  - *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or multiple edge collection names. Only edges from these collections will be considered for the path.
  - *startVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as start vertex of a path.
  - *endVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as end vertex of a path.
  - *edgeExamples* : A filter example for the edges in the shortest paths (see [example](#)).
  - *algorithm* : The algorithm to calculate the shortest paths. If both start and end vertex examples are empty [Floyd-Warshall](#) is used, otherwise the default is [Dijkstra](#).
  - *weight* : The name of the attribute of the edges containing the length as a string.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path could not be calculated.

Examples

A route planner example, shortest distance from all german to all french cities:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_SHORTEST_PATH("
........> + "'routeplanner', {}, {}, {" +
........> "weight : 'distance', endVertexCollectionRestriction :
'frenchCity', " +
........> "startVertexCollectionRestriction : 'germanCity'}) RETURN
[e.startVertex, e.vertex._id, " +
........> "e.distance, LENGTH(e.paths)]"
........> ).toArray();
[
  [
    "germanCity/Berlin",
    "frenchCity/Lyon",
    1100,
    1
  ],
  [
    "germanCity/Berlin",
    "frenchCity/Paris",
    1200,
    1
  ],
  [
    "germanCity/Cologne",
    "frenchCity/Paris",
    550,
    1
```

```
    ],
    [
      "germanCity/Cologne",
      "frenchCity/Lyon",
      700,
      1
    ],
    [
      "germanCity/Hamburg",
      "frenchCity/Lyon",
      1200,
      1
    ],
    [
      "germanCity/Hamburg",
      "frenchCity/Paris",
      900,
      1
    ]
]
```

A route planner example, shortest distance from Hamburg and Cologne to Lyon:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_SHORTEST_PATH("
........> +"'routeplanner', [{_id: 'germanCity/Cologne'},{_id:
'germanCity/Munich'}], " +
........> "'frenchCity/Lyon', " +
........> "{weight : 'distance'}) RETURN [e.startVertex,
e.vertex._id, e.distance, LENGTH(e.paths)]"
........> ).toArray();
[
  [
    "germanCity/Cologne",
    "frenchCity/Lyon",
    700,
    1
  ]
]
```

GRAPH_TRAVERSAL

The GRAPH_TRAVERSAL function traverses through the graph.

```
GRAPH_TRAVERSAL (graphName, startVertexExample, direction, options)
```

This function performs traversals on the given graph. For a more detailed documentation on the optional parameters see Traversals.

The complexity of this function strongly depends on the usage.

Parameters

- *graphName* : The name of the graph as a string.
- *startVertexExample* : An example for the desired vertices (see example).
- *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any*

(default).

- *options* (optional) : Object containing optional options, see [Traversals](#):

Examples

A route planner example, start a traversal from Hamburg :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_TRAVERSAL('routeplanner',
'germanCity/Hamburg'," +
........> " 'outbound') RETURN e"
........> ).toArray();
[
  [
    {
      "vertex" : {
        "_id" : "germanCity/Hamburg",
        "_rev" : "702636725",
        "_key" : "Hamburg",
        "isCapital" : false,
        "population" : 1000000
      }
    },
    {
      "vertex" : {
        "_id" : "germanCity/Cologne",
        "_rev" : "702440117",
        "_key" : "Cologne",
        "isCapital" : false,
        "population" : 1000000
      }
    },
    {
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "703029941",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      }
    },
    {
      "vertex" : {
        "_id" : "frenchCity/Paris",
        "_rev" : "703226549",
        "_key" : "Paris",
        "isCapital" : true,
        "population" : 4000000
      }
    },
    {
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "703029941",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
```

```
        }
      },
      {
        "vertex" : {
          "_id" : "frenchCity/Paris",
          "_rev" : "703226549",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      },
      {
        "vertex" : {
          "_id" : "frenchCity/Lyon",
          "_rev" : "703029941",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      },
      {
        "vertex" : {
          "_id" : "frenchCity/Lyon",
          "_rev" : "703029941",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      }
    ]
  ]
```

A route planner example, start a traversal from Hamburg with a max depth of 1 so only the direct neighbors of Hamburg are returned:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_TRAVERSAL('routeplanner',
'germanCity/Hamburg'," +
........> " 'outbound', {maxDepth : 1}) RETURN e"
........> ).toArray();
[
  [
    {
      "vertex" : {
        "_id" : "germanCity/Hamburg",
        "_rev" : "308306613",
        "_key" : "Hamburg",
        "isCapital" : false,
        "population" : 1000000
      }
    },
    {
      "vertex" : {
        "_id" : "germanCity/Cologne",
        "_rev" : "308110005",
        "_key" : "Cologne",
```

```
          "isCapital" : false,
          "population" : 1000000
        }
      },
      {
        "vertex" : {
          "_id" : "frenchCity/Paris",
          "_rev" : "308896437",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      },
      {
        "vertex" : {
          "_id" : "frenchCity/Lyon",
          "_rev" : "308699829",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      }
    ]
]
```

GRAPH_TRAVERSAL_TREE

The GRAPH_TRAVERSAL_TREE function traverses through the graph.

GRAPH_TRAVERSAL_TREE (graphName, startVertexExample, direction, connectName, options) This function creates a tree format from the result for a better visualization of the path. This function performs traversals on the given graph. For a more detailed documentation on the optional parameters see Traversals.

The complexity of this function strongly depends on the usage.

Parameters

- *graphName* : The name of the graph as a string.
- *startVertexExample* : An example for the desired vertices (see example).
- *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
- *connectName* : The result attribute which contains the connection.
- *options* (optional) : An object containing options, see Traversals:

Examples

A route planner example, start a traversal from Hamburg :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_TRAVERSAL_TREE('routeplanner',
'germanCity/Hamburg'," +
........> " 'outbound', 'connnection') RETURN e"
........> ).toArray();
[
  [
    [
```

```
{
  "_id" : "germanCity/Hamburg",
  "_rev" : "178807477",
  "_key" : "Hamburg",
  "isCapital" : false,
  "population" : 1000000,
  "connnection" : [
    {
      "_id" : "germanCity/Cologne",
      "_rev" : "178610869",
      "_key" : "Cologne",
      "isCapital" : false,
      "population" : 1000000,
      "connnection" : [
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "179200693",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "179397301",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000,
          "connnection" : [
            {
              "_id" : "frenchCity/Lyon",
              "_rev" : "179200693",
              "_key" : "Lyon",
              "isCapital" : false,
              "population" : 80000
            }
          ]
        }
      ]
    },
    {
      "_id" : "frenchCity/Paris",
      "_rev" : "179397301",
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000,
      "connnection" : [
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "179200693",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
    },
    {
      "_id" : "frenchCity/Lyon",
      "_rev" : "179200693",
      "_key" : "Lyon",
      "isCapital" : false,
```

```
                          "population" : 80000
                    }
                ]
            }
        ]
      ]
   ]
]
```

A route planner example, start a traversal from Hamburg with a max depth of 1 so only the direct neighbors of Hamburg are returned:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_TRAVERSAL_TREE('routeplanner',
'germanCity/Hamburg',"+
........> " 'outbound', 'connnection', {maxDepth : 1}) RETURN e"
........> ).toArray();
[
  [
    [
      {
        "_id" : "germanCity/Hamburg",
        "_rev" : "184902325",
        "_key" : "Hamburg",
        "isCapital" : false,
        "population" : 1000000,
        "connnection" : [
          {
            "_id" : "germanCity/Cologne",
            "_rev" : "184705717",
            "_key" : "Cologne",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "frenchCity/Paris",
            "_rev" : "185492149",
            "_key" : "Paris",
            "isCapital" : true,
            "population" : 4000000
          },
          {
            "_id" : "frenchCity/Lyon",
            "_rev" : "185295541",
            "_key" : "Lyon",
            "isCapital" : false,
            "population" : 80000
          }
        ]
      }
    ]
  ]
]
```

GRAPH_DISTANCE_TO

The GRAPH_DISTANCE_TO function returns all paths and there distance within a graph.

```
GRAPH_DISTANCE_TO (graphName, startVertexExample, endVertexExample,
options)
```

This function is a wrapper of GRAPH_SHORTEST_PATH. It does not return the actual path but only the distance between two vertices.

Examples

A route planner example, distance from all french to all german cities:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_DISTANCE_TO("
........> +"'routeplanner', {}, {}, { " +
........> " weight : 'distance', endVertexCollectionRestriction :
'germanCity', " +
........> "startVertexCollectionRestriction : 'frenchCity'}) RETURN
[e.startVertex, e.vertex._id, " +
........> "e.distance]"
........> ).toArray();
[
  [
    "frenchCity/Paris",
    "germanCity/Berlin",
    1200
  ],
  [
    "frenchCity/Paris",
    "germanCity/Cologne",
    550
  ],
  [
    "frenchCity/Paris",
    "germanCity/Hamburg",
    900
  ],
  [
    "frenchCity/Lyon",
    "germanCity/Hamburg",
    1200
  ],
  [
    "frenchCity/Lyon",
    "germanCity/Berlin",
    1100
  ],
  [
    "frenchCity/Lyon",
    "germanCity/Cologne",
    700
  ]
]
```

A route planner example, distance from Hamburg and Cologne to Lyon:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("FOR e IN GRAPH_DISTANCE_TO("
........> + "'routeplanner', [{_id: 'germanCity/Cologne'},{_id:
'germanCity/Hamburg'}], " +
........> "'frenchCity/Lyon', " +
........> "{weight : 'distance'}) RETURN [e.startVertex,
e.vertex._id, e.distance]"
........> ).toArray();
[
  [
    "germanCity/Cologne",
    "frenchCity/Lyon",
    700
  ],
  [
    "germanCity/Hamburg",
    "frenchCity/Lyon",
    1200
  ]
]
```

Graph measurements.

This section describes AQL functions to calculate various graph related measurements as defined in the mathematical graph theory.

GRAPH_ABSOLUTE_ECCENTRICITY

GRAPH_ABSOLUTE_ECCENTRICITY (graphName, vertexExample, options)

The GRAPH_ABSOLUTE_ECCENTRICITY function returns the [eccentricity](#) of the vertices defined by the examples.

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *vertexExample* : An example for the desired vertices (see [example](#)).
- *options* (optional) : An object containing the following options::
  - *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or multiple edge collection names. Only edges from these collections will be considered for the path.
  - *startVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as start vertex of a path.
  - *endVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as end vertex of a path.
  - *edgeExamples* : A filter example for the edges in the shortest paths (see [example](#)).
  - *algorithm* : The algorithm to calculate the shortest paths as a string. If vertex example is empty [Floyd-Warshall](#) is used as default, otherwise the default is [Dijkstra](#)
  - *weight* : The name of the attribute of the edges containing the length as a string.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the absolute eccentricity of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_ABSOLUTE_ECCENTRICITY('routeplanner', {})").toArray();
[
  {
    "germanCity/Berlin" : 1,
    "germanCity/Hamburg" : 1,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 1,
    "frenchCity/Lyon" : 1
  }
]
```

A route planner example, the absolute eccentricity of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_ECCENTRICITY("
........> +"'routeplanner', {}, {weight : 'distance'})").toArray();
[
  {
    "germanCity/Hamburg" : 1200,
    "germanCity/Cologne" : 850,
    "germanCity/Berlin" : 1200,
    "frenchCity/Paris" : 1200,
    "frenchCity/Lyon" : 1200
  }
]
```

A route planner example, the absolute eccentricity of all German cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_ECCENTRICITY("
........> + "'routeplanner', {}, {startVertexCollectionRestriction
: 'germanCity', " +
........> "direction : 'outbound', weight :
'distance'})").toArray();
[
  {
    "germanCity/Berlin" : 1200,
    "germanCity/Hamburg" : 1200,
    "germanCity/Cologne" : 700
  }
]
```

GRAPH_ECCENTRICITY

```
GRAPH_ECCENTRICITY (graphName, options)
```

The GRAPH_ECCENTRICITY function returns the normalized [eccentricity](#) of the graphs vertices

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *options* (optional) : An object containing the following options::
    - *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
    - *algorithm* : The algorithm to calculate the shortest paths as a string. Possible values are [Floyd-Warshall](#) (default) and [Dijkstra](#).
    - *weight* : The name of the attribute of the edges containing the length as a string.
    - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the eccentricity of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_ECCENTRICITY('routeplanner')").toArray();
[
  {
    "germanCity/Hamburg" : 1,
    "germanCity/Cologne" : 1,
    "germanCity/Berlin" : 1,
    "frenchCity/Paris" : 1,
    "frenchCity/Lyon" : 1
  }
]
```

A route planner example, the eccentricity of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ECCENTRICITY('routeplanner',
{weight : 'distance'})"
........> ).toArray();
[
  {
    "germanCity/Berlin" : 0.7083333333333335,
    "germanCity/Hamburg" : 0.7083333333333335,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0.7083333333333335,
    "frenchCity/Lyon" : 0.7083333333333335
  }
]
```

GRAPH_ABSOLUTE_CLOSENESS

```
GRAPH_ABSOLUTE_CLOSENESS (graphName, vertexExample, options)
```

The GRAPH_ABSOLUTE_CLOSENESS function returns the [closeness](#) of the vertices defined by the examples.

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *vertexExample* : An example for the desired vertices (see [example](#)).
- *options* : An object containing the following options::
  - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or multiple edge collection names. Only edges from these collections will be considered for the path.
  - *startVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as start vertex of a path.
  - *endVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as end vertex of a path.
  - *edgeExamples* : A filter example for the edges in the shortest paths (see [example](#)).
  - *algorithm* : The algorithm to calculate the shortest paths. Possible values are [Floyd-Warshall](#) (default) and [Dijkstra](#).
  - *weight* : The name of the attribute of the edges containing the length.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the absolute closeness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_ABSOLUTE_CLOSENESS('routeplanner', {})").toArray();
[
  {
    "germanCity/Hamburg" : 4,
    "germanCity/Cologne" : 4,
    "germanCity/Berlin" : 4,
    "frenchCity/Paris" : 4,
    "frenchCity/Lyon" : 4
  }
]
```

A route planner example, the absolute closeness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_CLOSENESS("
........> +"'routeplanner', {}, {weight : 'distance'})").toArray();
[
```

```
  {
    "germanCity/Hamburg" : 3000,
    "germanCity/Cologne" : 2600,
    "germanCity/Berlin" : 3550,
    "frenchCity/Paris" : 3200,
    "frenchCity/Lyon" : 3550
  }
]
```

A route planner example, the absolute closeness of all German cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_CLOSENESS("
........> + "'routeplanner', {}, {startVertexCollectionRestriction
: 'germanCity', " +
........> "direction : 'outbound', weight :
'distance'})").toArray();
[
  {
    "germanCity/Berlin" : 3550,
    "germanCity/Hamburg" : 2600,
    "germanCity/Cologne" : 1250
  }
]
```

GRAPH_CLOSENESS

`GRAPH_CLOSENESS (graphName, options)`

The GRAPH_CLOSENESS function returns the normalized [closeness](#) of graphs vertices.

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *options* : An object containing the following options::
  - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm* : The algorithm to calculate the shortest paths. Possible values are [Floyd-Warshall](#) (default) and [Dijkstra](#).
  - *weight* : The name of the attribute of the edges containing the length.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the closeness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
```

```
GRAPH_CLOSENESS('routeplanner')").toArray();
[
  {
    "germanCity/Berlin" : 1,
    "germanCity/Cologne" : 1,
    "germanCity/Hamburg" : 1,
    "frenchCity/Paris" : 1,
    "frenchCity/Lyon" : 1
  }
]
```

A route planner example, the closeness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_CLOSENESS("
........> +"'routeplanner', {weight : 'distance'})").toArray();
[
  {
    "germanCity/Berlin" : 0.7323943661971831,
    "germanCity/Cologne" : 1,
    "germanCity/Hamburg" : 0.8666666666666666,
    "frenchCity/Paris" : 0.8125,
    "frenchCity/Lyon" : 0.7323943661971831
  }
]
```

A route planner example, the absolute closeness of all cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_CLOSENESS("
........> + "'routeplanner',{direction : 'outbound', weight :
'distance'})"
........> ).toArray();
[
  {
    "germanCity/Hamburg" : 0.7279057017543861,
    "germanCity/Berlin" : 1,
    "frenchCity/Paris" : 0.3355263157894737,
    "germanCity/Cologne" : 0.5991541353383459,
    "frenchCity/Lyon" : 0
  }
]
```

GRAPH_ABSOLUTE_BETWEENNESS

```
GRAPH_ABSOLUTE_BETWEENNESS (graphName, vertexExample, options)
```

The GRAPH_ABSOLUTE_BETWEENNESS function returns the betweenness of all vertices in the graph.

The complexity of the function is described here.

- *graphName* : The name of the graph as a string.

- *options* : An object containing the following options::
    - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
    - *weight* : The name of the attribute of the edges containing the length.
    - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the betweenness can not be calculated.

Examples

A route planner example, the absolute betweenness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_ABSOLUTE_BETWEENNESS('routeplanner', {})").toArray();
[
  {
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 0,
    "germanCity/Berlin" : 0,
    "frenchCity/Paris" : 0,
    "frenchCity/Lyon" : 0
  }
]
```

A route planner example, the absolute betweenness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_BETWEENNESS("
........> +"'routeplanner', {weight : 'distance'})").toArray();
[
  {
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 2,
    "germanCity/Berlin" : 0,
    "frenchCity/Lyon" : 0,
    "frenchCity/Paris" : 0
  }
]
```

A route planner example, the absolute closeness regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_BETWEENNESS("
........> + "'routeplanner', {direction : 'outbound', weight :
'distance'})"
........> ).toArray();
[
  {
    "germanCity/Berlin" : 0,
```

```
      "germanCity/Cologne" : 1,
      "germanCity/Hamburg" : 0,
      "frenchCity/Lyon" : 0,
      "frenchCity/Paris" : 0
   }
]
```

GRAPH_BETWEENNESS

```
GRAPH_BETWEENNESS (graphName, options)
```

The GRAPH_BETWEENNESS function returns the [betweenness](#) of graphs vertices.

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *options* : An object containing the following options::
  - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *weight* : The name of the attribute of the edges containing the length.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the betweenness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_BETWEENNESS('routeplanner')").toArray();
[
   {
     "germanCity/Berlin" : null,
     "germanCity/Cologne" : null,
     "germanCity/Hamburg" : null,
     "frenchCity/Lyon" : null,
     "frenchCity/Paris" : null
   }
]
```

A route planner example, the betweenness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_BETWEENNESS('routeplanner',
{weight : 'distance'})").toArray();
[
   {
     "germanCity/Berlin" : 0,
     "germanCity/Hamburg" : 0,
```

```
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0,
    "frenchCity/Lyon" : 0
  }
]
```

A route planner example, the betweenness regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_BETWEENNESS("
........> + "'routeplanner', {direction : 'outbound', weight :
'distance'})").toArray();
[
  {
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0,
    "frenchCity/Lyon" : 0,
    "germanCity/Berlin" : 0
  }
]
```

GRAPH_RADIUS

```
GRAPH_RADIUS (graphName, options)
```

The GRAPH_RADIUS function returns the [radius](#) of a graph.

The complexity of the function is described [here](#).

- *graphName* : The name of the graph as a string.
- *options* : An object containing the following options::
  - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm* : The algorithm to calculate the shortest paths as a string. Possible values are [Floyd-Warshall](#) (default) and [Dijkstra](#).
  - *weight* : The name of the attribute of the edges containing the length.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the radius of the graph.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_RADIUS('routeplanner')").toArray();
[
  1
]
```

A route planner example, the radius of the graph. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_RADIUS('routeplanner', {weight :
'distance'})").toArray();
[
   850
]
```

A route planner example, the radius of the graph regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_RADIUS("
........> + "'routeplanner', {direction : 'outbound', weight :
'distance'})"
........> ).toArray();
[
   550
]
```

GRAPH_DIAMETER

GRAPH_DIAMETER (graphName, options)

The GRAPH_DIAMETER function returns the [diameter](#) of a graph.

The complexity of the function is described [here](#).

Parameters

- *graphName* : The name of the graph as a string.
- *options* : An object containing the following options::
  - *direction* : The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm* : The algorithm to calculate the shortest paths as a string. Possible values are [Floyd-Warshall](#) (default) and [Dijkstra](#).
  - *weight* : The name of the attribute of the edges containing the length.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the diameter of the graph.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN
GRAPH_DIAMETER('routeplanner')").toArray();
[
   1
```

```
    ]
```

A route planner example, the diameter of the graph. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_DIAMETER('routeplanner', {weight
: 'distance'})").toArray();
[
   1200
]
```

A route planner example, the diameter of the graph regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_DIAMETER("
........> + "'routeplanner', {direction : 'outbound', weight :
'distance'})"
........> ).toArray();
[
   1200
]
```

Advanced features

Subqueries

Wherever an expression is allowed in AQL, a subquery can be placed. A subquery is a query part that can introduce its own local variables without affecting variables and values in its outer scope(s).

It is required that subqueries be put inside parentheses *"("* and *")"* to explicitly mark their start and end points:

```
FOR u IN users
  LET recommendations = (
FOR r IN recommendations
  FILTER u.id == r.userId
  SORT u.rank DESC
  LIMIT 10
  RETURN r
  )
  RETURN { "user" : u, "recommendations" : recommendations }


FOR u IN users
  COLLECT city = u.city INTO g
  RETURN { "city" : city, "numUsers" : LENGTH(g), "maxRating": MAX(
FOR r IN g
  RETURN r.user.rating
  ) }
```

Subqueries might also include other subqueries themselves.

Variable expansion

In order to access a named attribute from all elements in a list easily, AQL offers the shortcut operator *[\]* * for variable expansion.

Using the *[\]* *operator with a variable will iterate over all elements in the variable thus allowing to access a particular attribute of each element. It is required that the expanded variable is a list. The result of the* [*]* operator is again a list.

```
FOR u IN users
  RETURN { "user" : u, "friendNames" : u.friends[*].name }
```

In the above example, the attribute *name* is accessed for each element in the list *u.friends*. The result is a flat list of friend names, made available as the attribute *friendNames*.

Extending AQL with User Functions

AQL comes with a built-in set of functions, but it is no full-feature programming language.

To add missing functionality or to simplify queries, users may add own functions to AQL. These functions can be written in Javascript, and must be registered via an API.

In order to avoid conflicts with existing or future built-in function names, all user functions must be put into separate namespaces. Invoking a user functions is then possible by referring to the fully-qualified function name, which includes the namespace, too.

Conventions

The **::** symbol is used inside AQL as the namespace separator. Using the namespace separator, users can create a multi-level hierarchy of function groups if required.

Examples:

```
RETURN myfunctions::myfunc()

RETURN myfunctions::math::random()
```

Note: As all function names in AQL, user function names are also case-insensitive.

Built-in AQL functions reside in the namespace *_aql*, which is also the default namespace to look in if an unqualified function name is found. Adding user functions to the *_aql* namespace is disallowed and will fail.

User functions can take any number of input arguments and should provide one result. They should be kept purely functional and thus free of side effects and state.

Especially it is unsupported to modify any global variables, or to change data of a collection from an AQL user function.

User function code is late-bound, and may thus not rely on any variables that existed at the time of declaration. If user function code requires access to any external data, it must take care to set up the data by itself.

User functions must only return primitive types (i.e. *null*, boolean values, numeric values, string values) or aggregate types (lists or documents) composed of these types. Returning any other Javascript object type from a user function may lead to undefined behavior and should be avoided.

Internally, user functions are stored in a system collection named *_aqlfunctions*. That means that by default they are excluded from dumps created with arangodump. To include AQL user functions in a dump, the dump should be started with the option *--include-system-collections true*.

Registering and Unregistering User Functions

AQL user functions can be registered using the *aqlfunctions* object as follows:

```
var aqlfunctions = require("org/arangodb/aql/functions");
```

To register a function, the fully qualified function name plus the function code must be specified.

Register

```
aqlfunctions.register(name, code, isDeterministic)
```

Registers an AQL user function, identified by a fully qualified function name. The function code in *code* must be specified as a Javascript function or a string representation of a Javascript function.

If a function identified by *name* already exists, the previous function definition will be updated.

The *isDeterministic* attribute can be used to specify whether the function results are fully deterministic (i.e. depend solely on the input and are the same for repeated calls with the same input values). It is not used at the moment but may be used for optimizations later.

Examples

```
require("org/arangodb/aql/functions").register("myfunctions::temperat

function (celsius) {
return celsius * 1.8 + 32;
});
```

Unregister

```
aqlfunctions.unregister(name)
```

Unregisters an existing AQL user function, identified by the fully qualified function name.

Trying to unregister a function that does not exist will result in an exception.

Examples

```
require("org/arangodb/aql/functions").unregister("myfunctions::temper
```

Unregister Group

```
aqlfunctions.unregisterGroup(prefix)
```

Unregisters a group of AQL user function, identified by a common function group prefix.

This will return the number of functions unregistered.

Examples

```
require("org/arangodb/aql/functions").unregisterGroup("myfunctions::t
```

```
require("org/arangodb/aql/functions").unregisterGroup("myfunctions");
```

To Array

`aqlfunctions.toArray()`

Returns all previously registered AQL user functions, with their fully qualified names and function code.

The result may optionally be restricted to a specified group of functions by specifying a group prefix:

`aqlfunctions.toArray(prefix)`

Examples

To list all available user functions:

```
require("org/arangodb/aql/functions").toArray();
```

To list all available user functions in the *myfunctions* namespace:

```
require("org/arangodb/aql/functions").toArray("myfunctions");
```

To list all available user functions in the *myfunctions::temperature* namespace:

```
require("org/arangodb/aql/functions").toArray("myfunctions::temperatu
```

Graphs

This chapter describes the general-graph module. It allows you to define a graph that is spread across several edge and document collections. This allows you to structure your models in line with your domain and group them logically in collections giving you the power to query them in the same graph queries. There is no need to include the referenced collections within the query, this module will handle it for you.

First Steps with Graphs

A Graph consists of *vertices* and *edges*. Edges are stored as documents in *edge collections*. In general a vertex is stored in a document collection. Which collections are used within a graph is defined via *edge definitions*. A graph can have an arbitrary number of edge definitions.

Three Steps to create a graph

- Create a graph

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> var graph = graph_module._create("myGraph");
arangosh> graph;
[ Graph myGraph EdgeDefinitions: [ ] VertexCollections: [ ] ]
```

- Add some vertex collections

```
arangosh> graph._addVertexCollection("shop");
arangosh> graph._addVertexCollection("customer");
arangosh> graph._addVertexCollection("pet");
arangosh> graph;
[ Graph myGraph EdgeDefinitions: [ ] VertexCollections: [
  "shop",
  "customer",
  "pet"
] ]
```

- Define relations on the

```
arangosh> var rel = graph_module._directedRelation("isCustomer",
["shop"], ["customer"]);
arangosh> graph._extendEdgeDefinitions(rel);
arangosh> graph;
[ Graph myGraph EdgeDefinitions: [
  "isCustomer: [shop] -> [customer]"
] VertexCollections: [ ] ]
```

Graph Management

Before we create our first graph, the philosophy of handling the graph content has to be introduced. A graph contains of a set of edge definitions each referring to one edge collection and defining constraints on the vertex collections used as start and end points of the edges. Furthermore a graph can contain an arbitrary amount of vertex collections, called orphan collections, that are not used in any edge definition but should be managed by the graph. In order to create a non empty graph the functionality to create edge definitions has to be introduced first:

Edge Definitions

An edge definition is a directed or undirected relation of a graph. Each graph can have arbitrary many relations defined within the edge definitions array.

Initialize the list

`graph_module._edgeDefinitions(relation1, relation2, ..., relationN)`
Create a list of edge definitions to construct a graph.

The list of edge definitions of a graph can be managed by the graph module itself. This function is the entry point for the management and will return the correct list.

Parameter

- *relationX*: An object representing a definition of one relation in the graph

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> directed_relation =
graph_module._directedRelation("lives_in", "user", "city");
{
  "collection" : "lives_in",
  "from" : [
    "user"
  ],
  "to" : [
    "city"
  ]
}
arangosh> undirected_relation =
graph_module._undirectedRelation("knows", "user");
{
  "collection" : "knows",
  "from" : [
    "user"
  ],
  "to" : [
    "user"
  ]
}
arangosh> edgedefinitions =
graph_module._edgeDefinitions(directed_relation,
undirected_relation);
[
  {
    "collection" : "lives_in",
    "from" : [
      "user"
    ],
    "to" : [
      "city"
    ]
  },
  {
    "collection" : "knows",
    "from" : [
      "user"
```

```
    ],
    "to" : [
      "user"
    ]
  }
]
```

Extend the list

```
graph_module._extendEdgeDefinitions(edgeDefinitions, relation1,
relation2, ..., relationN)
```
Extend the list of edge definitions to construct a graph.

In order to add more edge definitions to the graph before creating this function can be used to add more definitions to the initial list.

Parameter

- *edgeDefinitions*: A list of relation definition objects.
- *relationX*: An object representing a definition of one relation in the graph

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> directed_relation =
graph_module._directedRelation("lives_in", "user", "city");
{
  "collection" : "lives_in",
  "from" : [
    "user"
  ],
  "to" : [
    "city"
  ]
}
arangosh> undirected_relation =
graph_module._directedRelation("knows", "user");
[ArangoError 1935: Invalid number of arguments. Expected: 3]
arangosh> edgedefinitions =
graph_module._edgeDefinitions(directed_relation);
[
  {
    "collection" : "lives_in",
    "from" : [
      "user"
    ],
    "to" : [
      "city"
    ]
  }
]
arangosh> edgedefinitions =
graph_module._extendEdgeDefinitions(undirected_relation);
```

Undirected Relation

```
graph_module._undirectedRelation(relationName, vertexCollections)
```
Define an undirected relation.

Defines an undirected relation with the name *relationName* using the list of *vertexCollections*. This relation allows the user to store edges in any direction between any pair of vertices within the *vertexCollections*.

Parameter

- *relationName*: The name of the edge collection where the edges should be stored. Will be created if it does not yet exist.
- *vertexCollections*: One or a list of collection names for which connections are allowed. Will be created if they do not exist.

Examples

To define simple relation with only one vertex collection:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph_module._undirectedRelation("friend", "user");
{
  "collection" : "friend",
  "from" : [
    "user"
  ],
  "to" : [
    "user"
  ]
}
```

To define a relation between several vertex collections:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph_module._undirectedRelation("marriage", ["female",
"male"]);
{
  "collection" : "marriage",
  "from" : [
    "female",
    "male"
  ],
  "to" : [
    "female",
    "male"
  ]
}
```

Directed Relation

```
graph_module._directedRelation(relationName, fromVertexCollections,
toVertexCollections)
```
Define a directed relation.

The *relationName* defines the name of this relation and references to the underlying edge collection. The *fromVertexCollections* is an Array of document collections holding the start vertices. The *toVertexCollections* is an Array of document collections holding the target vertices. Relations are only allowed in the direction from any collection in *fromVertexCollections* to any collection in *toVertexCollections*.

Parameter

- *relationName*: The name of the edge collection where the edges should be stored. Will be created

if it does not yet exist.
- *fromVertexCollections*: One or a list of collection names. Source vertices for the edges have to be stored in these collections. Collections will be created if they do not exist.
- *toVertexCollections*: One or a list of collection names. Target vertices for the edges have to be stored in these collections. Collections will be created if they do not exist.

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph_module._directedRelation("has_bought", ["Customer",
"Company"], ["Groceries", "Electronics"]);
{
  "collection" : "has_bought",
  "from" : [
    "Customer",
    "Company"
  ],
  "to" : [
    "Groceries",
    "Electronics"
  ]
}
```

Vertex Collections

Each graph can have an arbitrary amount of vertex collections, which are not part of any edge definition of the graph. These collections are called orphan collections. If the graph is extended with an edge definition using one of the orphans, it will be removed from the set of orphan collection automatically.

Add

graph._addVertexCollection(vertexCollectionName, createCollection) *Add a vertex collection to the graph*

Adds a vertex collection to the set of orphan collections of the graph. If the collection does not exist, it will be created. If it is already used by any edge definition of the graph, an error will be thrown.

Parameter

- *vertexCollectionName*: Name of vertex collection.
- *createCollection* (optional): If true the collection will be created if it does not exist. Default: true.

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> var ed1 = graph_module._directedRelation("myEC1",
["myVC1"], ["myVC2"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._addVertexCollection("myVC3", true);
```

Get

graph._orphanCollections() *Get all orphan collections*

Returns all vertex collections of the graph that are not used in any edge definition.

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph")
arangosh> var ed1 = graph_module._directedRelation("myEC1",
["myVC1"], ["myVC2"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._addVertexCollection("myVC3", true);
arangosh> graph._orphanCollections();
[
   "myVC3"
]
```

Remove

```
graph._removeVertexCollection(vertexCollectionName, dropCollection)
```
Remove a vertex collection from the graph

Removes a vertex collection from the graph. Only collections not used in any relation definition can be removed. Optionally the collection can be deleted, if it is not used in any other graph.

Parameter

- *vertexCollectionName*: Name of vertex collection.
- *dropCollection* (optional): If true the collection will be dropped if it is not used in any other graph. Default: false.

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph")
arangosh> var ed1 = graph_module._directedRelation("myEC1",
["myVC1"], ["myVC2"]);
arangosh> var graph = graph_module._create("myGraph", [ed1]);
arangosh> graph._addVertexCollection("myVC3", true);
arangosh> graph._addVertexCollection("myVC4", true);
arangosh> graph._orphanCollections();
[
   "myVC3",
   "myVC4"
]
arangosh> graph._removeVertexCollection("myVC3");
arangosh> graph._orphanCollections();
[
   "myVC4"
]
```

Create a graph

After having introduced edge definitions and orphan collections a graph can be created.

`graph_module._create(graphName, edgeDefinitions, orphanCollections)`
Create a graph

The creation of a graph requires the name of the graph and a definition of its edges.

For every type of edge definition a convenience method exists that can be used to create a graph. Optionally a list of vertex collections can be added, which are not used in any edge definition. These collections are referred to as orphan collections within this chapter. All collections used within the creation process are created if they do not exist.

Parameter

- *graphName*: Unique identifier of the graph
- *edgeDefinitions* (optional): List of relation definition objects
- *orphanCollections* (optional): List of additional vertex collection names

Examples

Create an empty graph, edge definitions can be added at runtime:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph = graph_module._create("myGraph");
[ Graph myGraph EdgeDefinitions: [ ] VertexCollections: [ ] ]
```

Create a graph with edge definitions and orphan collections:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph = graph_module._create("myGraph",
........> [graph_module._undirectedRelation("myRelation", ["male",
"female"])], ["sessions"]);
[ Graph myGraph EdgeDefinitions: [
  "myRelation: [female, male] -> [female, male]"
] VertexCollections: [
  "sessions"
] ]
```

Complete Example to create a graph

Example Call:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> var edgeDefinitions = graph_module._edgeDefinitions();
arangosh> graph_module._extendEdgeDefinitions(edgeDefinitions,
graph_module._undirectedRelation("friend_of", ["Customer"]));
arangosh> graph_module._extendEdgeDefinitions(
........> edgeDefinitions, graph_module._directedRelation(
........> "has_bought", ["Customer", "Company"], ["Groceries",
"Electronics"]));
arangosh> graph_module._create("myStore", edgeDefinitions);
[ Graph myStore EdgeDefinitions: [
  "friend_of: [Customer] -> [Customer]",
  "has_bought: [Company, Customer] -> [Electronics, Groceries]"
```

```
] VertexCollections: [ ] ]
```

alternative call:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh>  var edgeDefinitions = graph_module._edgeDefinitions(
........>  graph_module._undirectedRelation("friend_of",
["Customer"]), graph_module._directedRelation(
........>   "has_bought", ["Customer", "Company"], ["Groceries",
"Electronics"]));
arangosh> graph_module._create("myStore", edgeDefinitions);
[ Graph myStore EdgeDefinitions: [
  "friend_of: [Customer] -> [Customer]",
  "has_bought: [Company, Customer] -> [Electronics, Groceries]"
] VertexCollections: [ ] ]
```

List available graphs

`graph_module._list()` *List all graphs.*

Lists all graph names stored in this database.

Examples

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph_module._list();
[
  "social",
  "routeplanner"
]
```

Load a graph

`graph_module._graph(graphName)` *Get a graph*

A graph can be get by its name.

Parameter

- *graphName*: Unique identifier of the graph

Examples

Get a graph:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph = graph_module._graph("social");
[ Graph social EdgeDefinitions: [
  "relation: [female, male] -> [female, male]"
] VertexCollections: [ ] ]
```

Remove a graph

`graph_module._drop(graphName, dropCollections)` *Remove a graph*

A graph can be dropped by its name. This will automatically drop all collections contained in the graph as long as they are not used within other graphs. To drop the collections, the optional parameter *drop-collections* can be set to *true*.

Parameter

- *graphName*: Unique identifier of the graph
- *dropCollections* (optional): Define if collections should be dropped (default: false)

Examples

Drop a graph and keep collections:

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph_module._drop("social");
true
arangosh> db._collection("female");
[ArangoCollection 242442933, "female" (type document, status
loaded)]
arangosh> db._collection("male");
[ArangoCollection 242574005, "male" (type document, status loaded)]
arangosh> db._collection("relation");
[ArangoCollection 242705077, "relation" (type edge, status loaded)]
```

```
arangosh> var graph_module = require("org/arangodb/general-graph");
arangosh> graph_module._drop("social", true);
true
arangosh> db._collection("female");
null
arangosh> db._collection("male");
null
arangosh> db._collection("relation");
null
```

Vertex

Save

`graph.vertexCollectionName.save(data)` *Create a new vertex in vertexCollectionName*

Parameter

- *data*: JSON data of vertex.

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.male.save({name: "Floyd", _key: "floyd"});
{
  "error" : false,
  "_id" : "male/floyd",
  "_rev" : "533815989",
  "_key" : "floyd"
}
```

Replace

`graph.vertexCollectionName.replace(vertexId, data, options)` *Replaces the data of a vertex in collection vertexCollectionName*

Parameter

- *vertexId*: *_id* attribute of the vertex
- *data*: JSON data of vertex.
- *options* (optional): See [collection documentation](#)

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.male.save({neym: "Jon", _key: "john"});
{
  "error" : false,
  "_id" : "male/john",
  "_rev" : "441803445",
  "_key" : "john"
}
arangosh> graph.male.replace("male/john", {name: "John"});
{
  "error" : false,
  "_id" : "male/john",
  "_rev" : "442000053",
  "_key" : "john"
}
```

Update

`graph.vertexCollectionName.update(vertexId, data, options)` *Updates the data of a vertex in collection vertexCollectionName*

Parameter

- *vertexId*: _id attribute of the vertex
- *data*: JSON data of vertex.
- *options* (optional): See collection documentation

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.female.save({name: "Lynda", _key: "linda"});
{
  "error" : false,
  "_id" : "female/linda",
  "_rev" : "655647413",
  "_key" : "linda"
}
arangosh> graph.female.update("female/linda", {name: "Linda", _key:
"linda"});
{
  "error" : false,
  "_id" : "female/linda",
  "_rev" : "655844021",
  "_key" : "linda"
}
```

Remove

`graph.vertexCollectionName.remove(vertexId, options)` Removes a vertex in collection *vertexCollectionName*

Additionally removes all ingoing and outgoing edges of the vertex recursively (see edge remove).

Parameter

- *vertexId*: _id attribute of the vertex
- *options* (optional): See collection documentation

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.male.save({name: "Kermit", _key: "kermit"});
{
  "error" : false,
  "_id" : "male/kermit",
  "_rev" : "659710645",
  "_key" : "kermit"
}
arangosh> db._exists("male/kermit")
true
arangosh> graph.male.remove("male/kermit")
true
```

```
arangosh> db._exists("male/kermit")
false
```

Edge

Save

`graph.edgeCollectionName.save(from, to, data, options)` *Creates an edge from vertex* from *to vertex* to *in collection edgeCollectionName*

Parameter

- *from*: *_id* attribute of the source vertex
- *to*: *_id* attribute of the target vertex
- *data*: JSON data of the edge
- *options* (optional): See collection documentation

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("male/bob", "female/alice", {type:
"married", _key: "bobAndAlice"});
{
  "error" : false,
  "_id" : "relation/bobAndAlice",
  "_rev" : "724525749",
  "_key" : "bobAndAlice"
}
```

If the collections of *from* and *to* are not defined in an edge definition of the graph, the edge will not be stored.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("relation/aliceAndBob",
"female/alice", {type: "married", _key: "bobAndAlice"});
[ArangoError 1906: invalid edge between relation/aliceAndBob and
female/alice.]
```

Replace

`graph.edgeCollectionName.replace(edgeId, data, options)` *Replaces the data of an edge in collection edgeCollectionName*

Parameter

- *edgeId*: *_id* attribute of the edge
- *data*: JSON data of the edge
- *options* (optional): See collection documentation

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("female/alice", "female/diana",
```

```
{typo: "nose", _key: "aliceAndDiana"});
{
  "error" : false,
  "_id" : "relation/aliceAndDiana",
  "_rev" : "30302901",
  "_key" : "aliceAndDiana"
}
arangosh> graph.relation.replace("relation/aliceAndDiana", {type:
"knows"});
{
  "error" : false,
  "_id" : "relation/aliceAndDiana",
  "_rev" : "30499509",
  "_key" : "aliceAndDiana"
}
```

Update

`graph.edgeCollectionName.update(edgeId, data, options)` *Updates the data of an edge in collection edgeCollectionName*

Parameter

- *edgeId*: *_id* attribute of the edge
- *data*: JSON data of the edge
- *options* (optional): See collection documentation

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("female/alice", "female/diana",
{type: "knows", _key: "aliceAndDiana"});
{
  "error" : false,
  "_id" : "relation/aliceAndDiana",
  "_rev" : "305029813",
  "_key" : "aliceAndDiana"
}
arangosh> graph.relation.update("relation/aliceAndDiana", {type:
"quarrelled", _key: "aliceAndDiana"});
{
  "error" : false,
  "_id" : "relation/aliceAndDiana",
  "_rev" : "305226421",
  "_key" : "aliceAndDiana"
}
```

Remove

`graph.edgeCollectionName.remove(edgeId, options)` *Removes an edge in collection edgeCollectionName*

Parameter

- *edgeId*: *_id* attribute of the edge

- *options* (optional): See collection documentation

If this edge is used as a vertex by another edge, the other edge will be removed (recursively).

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph.relation.save("female/alice", "female/diana",
{_key: "aliceAndDiana"});
{
  "error" : false,
  "_id" : "relation/aliceAndDiana",
  "_rev" : "421552821",
  "_key" : "aliceAndDiana"
}
arangosh> db._exists("relation/aliceAndDiana")
true
arangosh> graph.relation.remove("relation/aliceAndDiana")
true
arangosh> db._exists("relation/aliceAndDiana")
false
```

Graph Functions

This chapter describes various functions on a graph. A lot of these accept a vertex (or edge) example as parameter as defined in the next section.

Definition of examples

For many of the following functions *examples* can be passed in as a parameter. *Examples* are used to filter the result set for objects that match the conditions. These *examples* can have the following values:

- *null*, there is no matching executed all found results are valid.
- A *string*, only results are returned, which *_id* equal the value of the string
- An example *object*, defining a set of attributes. Only results having these attributes are matched.
- A *list* containing example *objects* and/or *strings*. All results matching at least one of the elements in the list are returned.

Get vertices from edges.

Get vertex *from* of an edge

`graph._fromVertex(edgeId)` *Get the source vertex of an edge*

Returns the vertex defined with the attribute *_from* of the edge with *edgeId* as its *_id*.

Parameter

- *edgeId*: *_id* attribute of the edge

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._fromVertex("relation/aliceAndBob")
{
  "name" : "Alice",
  "_id" : "female/alice",
  "_rev" : "411787957",
  "_key" : "alice"
}
```

Get vertex *to* of an edge

`graph._toVertex(edgeId)` *Get the target vertex of an edge*

Returns the vertex defined with the attribute *_to* of the edge with *edgeId* as its *_id*.

Parameter

- *edgeId*: *_id* attribute of the edge

Examples

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._toVertex("relation/aliceAndBob")
{
  "name" : "Bob",
  "_id" : "male/bob",
  "_rev" : "42689205",
  "_key" : "bob"
}
```

_neighbors

graph._neighbors(vertexExample, options) *Get all neighbors of the vertices defined by the example*

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertexExample. The complexity of this method is $O(n*m^x)$ with n being the vertices defined by the parameter vertexExamplex, m the average amount of neighbors and x the maximal depths. Hence the default call would have a complexity of $O(n*m)$;

Parameter

- *vertexExample*: See [Definition of examples](#)
- *options* (optional): An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeExamples*: Filter the edges, see [Definition of examples](#)
  - *neighborExamples*: Filter the neighbor vertices, see [Definition of examples](#)
  - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
  - *vertexCollectionRestriction* : One or a list of vertex-collection names that should be considered on the intermediate vertex steps.
  - *minDepth*: Defines the minimal number of intermediate steps to neighbors (default is 1).
  - *maxDepth*: Defines the maximal number of intermediate steps to neighbors (default is 1).

Examples

A route planner example, all neighbors of capitals.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._neighbors({isCapital : true});
[
  {
    "_id" : "germanCity/Cologne",
    "_rev" : "450978485",
    "_key" : "Cologne",
    "isCapital" : false,
    "population" : 1000000
  },
  {
    "_id" : "germanCity/Hamburg",
    "_rev" : "451175093",
    "_key" : "Hamburg",
    "isCapital" : false,
    "population" : 1000000
  },
  {
    "_id" : "frenchCity/Lyon",
    "_rev" : "451568309",
    "_key" : "Lyon",
    "isCapital" : false,
    "population" : 80000
  },
  {
    "_id" : "frenchCity/Paris",
    "_rev" : "451764917",
    "_key" : "Paris",
```

```
      "isCapital" : true,
      "population" : 4000000
    },
    {
      "_id" : "frenchCity/Lyon",
      "_rev" : "451568309",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    {
      "_id" : "germanCity/Berlin",
      "_rev" : "450781877",
      "_key" : "Berlin",
      "isCapital" : true,
      "population" : 3000000
    },
    {
      "_id" : "germanCity/Cologne",
      "_rev" : "450978485",
      "_key" : "Cologne",
      "isCapital" : false,
      "population" : 1000000
    },
    {
      "_id" : "germanCity/Hamburg",
      "_rev" : "451175093",
      "_key" : "Hamburg",
      "isCapital" : false,
      "population" : 1000000
    }
]
```

A route planner example, all outbound neighbors of Hamburg.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._neighbors('germanCity/Hamburg', {direction :
'outbound', maxDepth : 2});
[
    {
      "_id" : "germanCity/Cologne",
      "_rev" : "444883637",
      "_key" : "Cologne",
      "isCapital" : false,
      "population" : 1000000
    },
    {
      "_id" : "frenchCity/Lyon",
      "_rev" : "445473461",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    {
      "_id" : "frenchCity/Paris",
      "_rev" : "445670069",
```

```
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000
    },
    {
      "_id" : "frenchCity/Paris",
      "_rev" : "445670069",
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000
    },
    {
      "_id" : "frenchCity/Lyon",
      "_rev" : "445473461",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    {
      "_id" : "frenchCity/Lyon",
      "_rev" : "445473461",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    }
  ]
```

_commonNeighbors

graph._commonNeighbors(vertex1Example, vertex2Examples, optionsVertex1, optionsVertex2) *Get all common neighbors of the vertices defined by the examples.*

This function returns the intersection of *graph_module._neighbors(vertex1Example, optionsVertex1)* and *graph_module._neighbors(vertex2Example, optionsVertex2)*. For parameter documentation see _neighbors.

The complexity of this method is **O(n*m^x)** with *n* being the maximal amount of vertices defined by the parameters vertexExamples, *m* the average amount of neighbors and *x* the maximal depths. Hence the default call would have a complexity of **O(n*m)**;

Examples

A route planner example, all common neighbors of capitals.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonNeighbors({isCapital : true}, {isCapital :
true});
[
  {
    "frenchCity/Paris" : {
      "germanCity/Berlin" : [
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "663315125",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "germanCity/Hamburg",
          "_rev" : "663511733",
          "_key" : "Hamburg",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "663904949",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
    }
  },
  {
    "germanCity/Berlin" : {
      "frenchCity/Paris" : [
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "663315125",
          "_key" : "Cologne",
          "isCapital" : false,
```

```
          "population" : 1000000
        },
        {
          "_id" : "germanCity/Hamburg",
          "_rev" : "663511733",
          "_key" : "Hamburg",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "663904949",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
    }
  }
]
```

A route planner example, all common outbound neighbors of Hamburg with any other location which have a maximal depth of 2 :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonNeighbors(
........>     'germanCity/Hamburg',
........>     {},
........>     {direction : 'outbound', maxDepth : 2},
........>     {direction : 'outbound', maxDepth : 2});
[
  {
    "germanCity/Hamburg" : {
      "germanCity/Berlin" : [
        {
          "_id" : "germanCity/Cologne",
          "_rev" : "776823477",
          "_key" : "Cologne",
          "isCapital" : false,
          "population" : 1000000
        },
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "777413301",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "777609909",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      ],
```

```
      "germanCity/Cologne" : [
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "777413301",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        },
        {
          "_id" : "frenchCity/Paris",
          "_rev" : "777609909",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      ],
      "frenchCity/Paris" : [
        {
          "_id" : "frenchCity/Lyon",
          "_rev" : "777413301",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      ]
    }
  }
]
```

_countCommonNeighbors

```
graph._countCommonNeighbors(vertex1Example, vertex2Examples,
optionsVertex1, optionsVertex2) Get the amount of common neighbors of the vertices
defined by the examples.
```

Similar to _commonNeighbors but returns count instead of the elements.

Examples

A route planner example, all common neighbors of capitals.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._countCommonNeighbors({isCapital : true},
{isCapital : true});
[
  {
    "frenchCity/Paris" : [
      {
        "germanCity/Berlin" : 3
      }
    ]
  },
  {
    "germanCity/Berlin" : [
      {
        "frenchCity/Paris" : 3
      }
    ]
  }
]
```

A route planner example, all common outbound neighbors of Hamburg with any other location which have a maximal depth of 2 :

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._countCommonNeighbors('germanCity/Hamburg', {},
{direction : 'outbound', maxDepth : 2},
........> {direction : 'outbound', maxDepth : 2});
[
  {
    "germanCity/Hamburg" : [
      {
        "germanCity/Berlin" : 3
      },
      {
        "germanCity/Cologne" : 2
      },
      {
        "frenchCity/Paris" : 1
      }
    ]
  }
```

```
]
```

_commonProperties

`graph._commonProperties(vertex1Example, vertex2Examples, options)` *Get the vertices of the graph that share common properties.*

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertex1Example and vertex2Example.

The complexity of this method is **O(n)** with *n* being the maximal amount of vertices defined by the parameters vertexExamples.

Parameter

- *vertex1Examples*: Filter the set of source vertices, see [Definition of examples](#)
- *vertex2Examples*: Filter the set of vertices compared to, see [Definition of examples](#)
- *options* (optional): An object defining further options. Can have the following values:
    - *vertex1CollectionRestriction* : One or a list of vertex-collection names that should be searched for source vertices.
    - *vertex2CollectionRestriction* : One or a list of vertex-collection names that should be searched for compare vertices.
    - *ignoreProperties* : One or a list of attribute names of a document that should be ignored.

Examples

A route planner example, all locations with the same properties:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonProperties({}, {});
[
  {
    "frenchCity/Lyon" : [
      {
        "_id" : "germanCity/Hamburg",
        "isCapital" : false
      },
      {
        "_id" : "germanCity/Cologne",
        "isCapital" : false
      }
    ]
  },
  {
    "frenchCity/Paris" : [
      {
        "_id" : "germanCity/Berlin",
        "isCapital" : true
      }
    ]
  },
  {
    "germanCity/Berlin" : [
      {
        "_id" : "frenchCity/Paris",
        "isCapital" : true
      }
    ]
  },
```

```
  {
    "germanCity/Cologne" : [
      {
        "_id" : "germanCity/Hamburg",
        "isCapital" : false,
        "population" : 1000000
      },
      {
        "_id" : "frenchCity/Lyon",
        "isCapital" : false
      }
    ]
  },
  {
    "germanCity/Hamburg" : [
      {
        "_id" : "germanCity/Cologne",
        "isCapital" : false,
        "population" : 1000000
      },
      {
        "_id" : "frenchCity/Lyon",
        "isCapital" : false
      }
    ]
  }
]
```

A route planner example, all cities which share same properties except for population.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._commonProperties({}, {}, {ignoreProperties:
'population'});
[
  {
    "frenchCity/Lyon" : [
      {
        "_id" : "germanCity/Hamburg",
        "isCapital" : false
      },
      {
        "_id" : "germanCity/Cologne",
        "isCapital" : false
      }
    ]
  },
  {
    "frenchCity/Paris" : [
      {
        "_id" : "germanCity/Berlin",
        "isCapital" : true
      }
    ]
  },
  {
    "germanCity/Berlin" : [
```

```
      {
        "_id" : "frenchCity/Paris",
        "isCapital" : true
      }
    ]
  },
  {
    "germanCity/Cologne" : [
      {
        "_id" : "germanCity/Hamburg",
        "isCapital" : false,
        "population" : 1000000
      },
      {
        "_id" : "frenchCity/Lyon",
        "isCapital" : false
      }
    ]
  },
  {
    "germanCity/Hamburg" : [
      {
        "_id" : "germanCity/Cologne",
        "isCapital" : false,
        "population" : 1000000
      },
      {
        "_id" : "frenchCity/Lyon",
        "isCapital" : false
      }
    ]
  }
]
```

_countCommonProperties

```
graph._countCommonProperties(vertex1Example, vertex2Examples, options)
```
Get the amount of vertices of the graph that share common properties.

Similar to [_commonProperties](#) but returns count instead of the objects.

Examples

A route planner example, all locations with the same properties:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._countCommonProperties({}, {});
[
  {
    "frenchCity/Lyon" : 2
  },
  {
    "frenchCity/Paris" : 1
  },
  {
    "germanCity/Berlin" : 1
  },
  {
    "germanCity/Cologne" : 2
  },
  {
    "germanCity/Hamburg" : 2
  }
]
```

A route planner example, all German cities which share same properties except for population.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._countCommonProperties({}, {},
{vertex1CollectionRestriction : 'germanCity',
........> vertex2CollectionRestriction : 'germanCity'
,ignoreProperties: 'population'});
[
  {
    "germanCity/Cologne" : 1
  },
  {
    "germanCity/Hamburg" : 1
  }
]
```

_paths

`graph._paths(options)` *The _paths function returns all paths of a graph.*

This function determines all available paths in a graph.

The complexity of this method is **O(n*n*m)** with *n* being the amount of vertices in the graph and *m* the average amount of connected edges;

Parameters

- *options* (optional) : An object containing options, see below:
    - *direction* : The direction of the edges. Possible values are *any*, *inbound* and *outbound* (default).
    - *followCycles* (optional) : If set to *true* the query follows cycles in the graph, default is false.
    - *minLength* (optional) : Defines the minimal length a path must have to be returned (default is 0).
    - *maxLength* (optional) : Defines the maximal length a path must have to be returned (default is 10).

Examples

Return all paths of the graph "social":

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("social");
arangosh> g._paths();
[
  [
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "226452149",
          "_key" : "diana",
          "name" : "Diana"
        }
      ],
      "edges" : [ ],
      "source" : {
        "_id" : "female/diana",
        "_rev" : "226452149",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
        "_id" : "female/diana",
        "_rev" : "226452149",
        "_key" : "diana",
        "name" : "Diana"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "female/alice",
          "_rev" : "225731253",
```

```
        "_key" : "alice",
        "name" : "Alice"
      }
    ],
    "edges" : [ ],
    "source" : {
      "_id" : "female/alice",
      "_rev" : "225731253",
      "_key" : "alice",
      "name" : "Alice"
    },
    "destination" : {
      "_id" : "female/alice",
      "_rev" : "225731253",
      "_key" : "alice",
      "name" : "Alice"
    }
  },
  {
    "vertices" : [
      {
        "_id" : "female/alice",
        "_rev" : "225731253",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "male/bob",
        "_rev" : "226058933",
        "_key" : "bob",
        "name" : "Bob"
      }
    ],
    "edges" : [
      {
        "_id" : "relation/aliceAndBob",
        "_rev" : "226779829",
        "_key" : "aliceAndBob",
        "_from" : "female/alice",
        "_to" : "male/bob",
        "type" : "married"
      }
    ],
    "source" : {
      "_id" : "female/alice",
      "_rev" : "225731253",
      "_key" : "alice",
      "name" : "Alice"
    },
    "destination" : {
      "_id" : "male/bob",
      "_rev" : "226058933",
      "_key" : "bob",
      "name" : "Bob"
    }
  },
  {
    "vertices" : [
      {
        "_id" : "female/alice",
```

```
            "_rev" : "225731253",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/bob",
            "_rev" : "226058933",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "female/diana",
            "_rev" : "226452149",
            "_key" : "diana",
            "name" : "Diana"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndBob",
            "_rev" : "226779829",
            "_key" : "aliceAndBob",
            "_from" : "female/alice",
            "_to" : "male/bob",
            "type" : "married"
          },
          {
            "_id" : "relation/bobAndDiana",
            "_rev" : "227435189",
            "_key" : "bobAndDiana",
            "_from" : "male/bob",
            "_to" : "female/diana",
            "type" : "friend"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "225731253",
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "female/diana",
          "_rev" : "226452149",
          "_key" : "diana",
          "name" : "Diana"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "female/alice",
            "_rev" : "225731253",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/charly",
            "_rev" : "226255541",
            "_key" : "charly",
```

```
            "name" : "Charly"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndCharly",
            "_rev" : "227041973",
            "_key" : "aliceAndCharly",
            "_from" : "female/alice",
            "_to" : "male/charly",
            "type" : "friend"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "225731253",
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "male/charly",
          "_rev" : "226255541",
          "_key" : "charly",
          "name" : "Charly"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "female/alice",
            "_rev" : "225731253",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "male/charly",
            "_rev" : "226255541",
            "_key" : "charly",
            "name" : "Charly"
          },
          {
            "_id" : "female/diana",
            "_rev" : "226452149",
            "_key" : "diana",
            "name" : "Diana"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/aliceAndCharly",
            "_rev" : "227041973",
            "_key" : "aliceAndCharly",
            "_from" : "female/alice",
            "_to" : "male/charly",
            "type" : "friend"
          },
          {
            "_id" : "relation/charlyAndDiana",
            "_rev" : "227238581",
            "_key" : "charlyAndDiana",
```

```
            "_from" : "male/charly",
            "_to" : "female/diana",
            "type" : "married"
          }
        ],
        "source" : {
          "_id" : "female/alice",
          "_rev" : "225731253",
          "_key" : "alice",
          "name" : "Alice"
        },
        "destination" : {
          "_id" : "female/diana",
          "_rev" : "226452149",
          "_key" : "diana",
          "name" : "Diana"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "male/bob",
            "_rev" : "226058933",
            "_key" : "bob",
            "name" : "Bob"
          }
        ],
        "edges" : [ ],
        "source" : {
          "_id" : "male/bob",
          "_rev" : "226058933",
          "_key" : "bob",
          "name" : "Bob"
        },
        "destination" : {
          "_id" : "male/bob",
          "_rev" : "226058933",
          "_key" : "bob",
          "name" : "Bob"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "male/bob",
            "_rev" : "226058933",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "female/diana",
            "_rev" : "226452149",
            "_key" : "diana",
            "name" : "Diana"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/bobAndDiana",
            "_rev" : "227435189",
```

```
            "_key" : "bobAndDiana",
            "_from" : "male/bob",
            "_to" : "female/diana",
            "type" : "friend"
          }
        ],
        "source" : {
          "_id" : "male/bob",
          "_rev" : "226058933",
          "_key" : "bob",
          "name" : "Bob"
        },
        "destination" : {
          "_id" : "female/diana",
          "_rev" : "226452149",
          "_key" : "diana",
          "name" : "Diana"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "male/charly",
            "_rev" : "226255541",
            "_key" : "charly",
            "name" : "Charly"
          }
        ],
        "edges" : [ ],
        "source" : {
          "_id" : "male/charly",
          "_rev" : "226255541",
          "_key" : "charly",
          "name" : "Charly"
        },
        "destination" : {
          "_id" : "male/charly",
          "_rev" : "226255541",
          "_key" : "charly",
          "name" : "Charly"
        }
      },
      {
        "vertices" : [
          {
            "_id" : "male/charly",
            "_rev" : "226255541",
            "_key" : "charly",
            "name" : "Charly"
          },
          {
            "_id" : "female/diana",
            "_rev" : "226452149",
            "_key" : "diana",
            "name" : "Diana"
          }
        ],
        "edges" : [
          {
            "_id" : "relation/charlyAndDiana",
```

```
          "_rev" : "227238581",
          "_key" : "charlyAndDiana",
          "_from" : "male/charly",
          "_to" : "female/diana",
          "type" : "married"
        }
      ],
      "source" : {
        "_id" : "male/charly",
        "_rev" : "226255541",
        "_key" : "charly",
        "name" : "Charly"
      },
      "destination" : {
        "_id" : "female/diana",
        "_rev" : "226452149",
        "_key" : "diana",
        "name" : "Diana"
      }
    }
  ]
]
```

Return all inbound paths of the graph "social" with a maximal length of 1 and a minimal length of 2:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("social");
arangosh> g._paths({direction : 'inbound', minLength : 1, maxLength
:  2});
[
  [
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "631399093",
          "_key" : "diana",
          "name" : "Diana"
        },
        {
          "_id" : "male/charly",
          "_rev" : "631202485",
          "_key" : "charly",
          "name" : "Charly"
        }
      ],
      "edges" : [
        {
          "_id" : "relation/charlyAndDiana",
          "_rev" : "632185525",
          "_key" : "charlyAndDiana",
          "_from" : "male/charly",
          "_to" : "female/diana",
          "type" : "married"
        }
      ],
      "source" : {
```

```
        "_id" : "female/diana",
        "_rev" : "631399093",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
        "_id" : "male/charly",
        "_rev" : "631202485",
        "_key" : "charly",
        "name" : "Charly"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "631399093",
          "_key" : "diana",
          "name" : "Diana"
        },
        {
          "_id" : "male/charly",
          "_rev" : "631202485",
          "_key" : "charly",
          "name" : "Charly"
        },
        {
          "_id" : "female/alice",
          "_rev" : "630678197",
          "_key" : "alice",
          "name" : "Alice"
        }
      ],
      "edges" : [
        {
          "_id" : "relation/charlyAndDiana",
          "_rev" : "632185525",
          "_key" : "charlyAndDiana",
          "_from" : "male/charly",
          "_to" : "female/diana",
          "type" : "married"
        },
        {
          "_id" : "relation/aliceAndCharly",
          "_rev" : "631988917",
          "_key" : "aliceAndCharly",
          "_from" : "female/alice",
          "_to" : "male/charly",
          "type" : "friend"
        }
      ],
      "source" : {
        "_id" : "female/diana",
        "_rev" : "631399093",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
        "_id" : "female/alice",
        "_rev" : "630678197",
```

```
        "_key" : "alice",
        "name" : "Alice"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "631399093",
          "_key" : "diana",
          "name" : "Diana"
        },
        {
          "_id" : "male/bob",
          "_rev" : "631005877",
          "_key" : "bob",
          "name" : "Bob"
        }
      ],
      "edges" : [
        {
          "_id" : "relation/bobAndDiana",
          "_rev" : "632382133",
          "_key" : "bobAndDiana",
          "_from" : "male/bob",
          "_to" : "female/diana",
          "type" : "friend"
        }
      ],
      "source" : {
        "_id" : "female/diana",
        "_rev" : "631399093",
        "_key" : "diana",
        "name" : "Diana"
      },
      "destination" : {
        "_id" : "male/bob",
        "_rev" : "631005877",
        "_key" : "bob",
        "name" : "Bob"
      }
    },
    {
      "vertices" : [
        {
          "_id" : "female/diana",
          "_rev" : "631399093",
          "_key" : "diana",
          "name" : "Diana"
        },
        {
          "_id" : "male/bob",
          "_rev" : "631005877",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "female/alice",
          "_rev" : "630678197",
          "_key" : "alice",
```

```json
        "name" : "Alice"
      }
    ],
    "edges" : [
      {
        "_id" : "relation/bobAndDiana",
        "_rev" : "632382133",
        "_key" : "bobAndDiana",
        "_from" : "male/bob",
        "_to" : "female/diana",
        "type" : "friend"
      },
      {
        "_id" : "relation/aliceAndBob",
        "_rev" : "631726773",
        "_key" : "aliceAndBob",
        "_from" : "female/alice",
        "_to" : "male/bob",
        "type" : "married"
      }
    ],
    "source" : {
      "_id" : "female/diana",
      "_rev" : "631399093",
      "_key" : "diana",
      "name" : "Diana"
    },
    "destination" : {
      "_id" : "female/alice",
      "_rev" : "630678197",
      "_key" : "alice",
      "name" : "Alice"
    }
  },
  {
    "vertices" : [
      {
        "_id" : "male/bob",
        "_rev" : "631005877",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "female/alice",
        "_rev" : "630678197",
        "_key" : "alice",
        "name" : "Alice"
      }
    ],
    "edges" : [
      {
        "_id" : "relation/aliceAndBob",
        "_rev" : "631726773",
        "_key" : "aliceAndBob",
        "_from" : "female/alice",
        "_to" : "male/bob",
        "type" : "married"
      }
    ],
    "source" : {
```

```
      "_id" : "male/bob",
      "_rev" : "631005877",
      "_key" : "bob",
      "name" : "Bob"
    },
    "destination" : {
      "_id" : "female/alice",
      "_rev" : "630678197",
      "_key" : "alice",
      "name" : "Alice"
    }
  },
  {
    "vertices" : [
      {
        "_id" : "male/charly",
        "_rev" : "631202485",
        "_key" : "charly",
        "name" : "Charly"
      },
      {
        "_id" : "female/alice",
        "_rev" : "630678197",
        "_key" : "alice",
        "name" : "Alice"
      }
    ],
    "edges" : [
      {
        "_id" : "relation/aliceAndCharly",
        "_rev" : "631988917",
        "_key" : "aliceAndCharly",
        "_from" : "female/alice",
        "_to" : "male/charly",
        "type" : "friend"
      }
    ],
    "source" : {
      "_id" : "male/charly",
      "_rev" : "631202485",
      "_key" : "charly",
      "name" : "Charly"
    },
    "destination" : {
      "_id" : "female/alice",
      "_rev" : "630678197",
      "_key" : "alice",
      "name" : "Alice"
    }
  }
]
]
```

_shortestPath

`graph._shortestPath(startVertexExample, endVertexExample, options)` *The _shortestPath function returns all shortest paths of a graph.*

This function determines all shortest paths in a graph. The function accepts an id, an example, a list of examples or even an empty example as parameter for start and end vertex. If one wants to call this function to receive nearly all shortest paths for a graph the option *algorithm* should be set to [Floyd-Warshall](#) to increase performance. If no algorithm is provided in the options the function chooses the appropriate one (either [Floyd-Warshall](#) or [Dijkstra](#)) according to its parameters. The length of a path is by default the amount of edges from one start vertex to an end vertex. The option weight allows the user to define an edge attribute representing the length.

The complexity of the function is described [here](#).

Parameters

- *startVertexExample* (optional) : An example for the desired start Vertices (see [Definition of examples](#)).
- *endVertexExample* (optional) : An example for the desired end Vertices (see [Definition of examples](#)).
- *options* (optional) : An object containing options, see below:
  - *direction* : The direction of the edges as a string. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or multiple edge collection names. Only edges from these collections will be considered for the path.
  - *startVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as start vertex of a path.
  - *endVertexCollectionRestriction* : One or multiple vertex collection names. Only vertices from these collections will be considered as end vertex of a path.
  - *edgeExamples* : A filter example for the edges in the shortest paths (see [example](#)).
  - *algorithm* : The algorithm to calculate the shortest paths. If both start and end vertex examples are empty [Floyd-Warshall](#) is used, otherwise the default is [Dijkstra](#)
  - *weight* : The name of the attribute of the edges containing the length as a string.
  - *defaultWeight* : Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as length. If no default is supplied the default would be positive Infinity so the path could not be calculated.

Examples

A route planner example, shortest path from all german to all french cities:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._shortestPath({}, {}, {weight : 'distance',
endVertexCollectionRestriction : 'frenchCity',
........> startVertexCollectionRestriction : 'germanCity'});
[
  [
    {
      "startVertex" : "germanCity/Berlin",
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "354050741",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      },
```

```
    "paths" : [
      {
        "edges" : [
          {
            "_id" : "internationalHighway/355623605",
            "_rev" : "355623605",
            "_key" : "355623605",
            "_from" : "germanCity/Berlin",
            "_to" : "frenchCity/Lyon",
            "distance" : 1100
          }
        ],
        "vertices" : [
          "germanCity/Berlin",
          "frenchCity/Lyon"
        ]
      }
    ],
    "distance" : 1100
  },
  {
    "startVertex" : "germanCity/Berlin",
    "vertex" : {
      "_id" : "frenchCity/Paris",
      "_rev" : "354247349",
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000
    },
    "paths" : [
      {
        "edges" : [
          {
            "_id" : "internationalHighway/355820213",
            "_rev" : "355820213",
            "_key" : "355820213",
            "_from" : "germanCity/Berlin",
            "_to" : "frenchCity/Paris",
            "distance" : 1200
          }
        ],
        "vertices" : [
          "germanCity/Berlin",
          "frenchCity/Paris"
        ]
      }
    ],
    "distance" : 1200
  },
  {
    "startVertex" : "germanCity/Hamburg",
    "vertex" : {
      "_id" : "frenchCity/Lyon",
      "_rev" : "354050741",
      "_key" : "Lyon",
      "isCapital" : false,
      "population" : 80000
    },
    "paths" : [
      {
```

```
        "edges" : [
          {
            "_id" : "germanHighway/354968245",
            "_rev" : "354968245",
            "_key" : "354968245",
            "_from" : "germanCity/Hamburg",
            "_to" : "germanCity/Cologne",
            "distance" : 500
          },
          {
            "_id" : "internationalHighway/356410037",
            "_rev" : "356410037",
            "_key" : "356410037",
            "_from" : "germanCity/Cologne",
            "_to" : "frenchCity/Lyon",
            "distance" : 700
          }
        ],
        "vertices" : [
          "germanCity/Hamburg",
          "germanCity/Cologne",
          "frenchCity/Lyon"
        ]
      }
    ],
    "distance" : 1200
  },
  {
    "startVertex" : "germanCity/Hamburg",
    "vertex" : {
      "_id" : "frenchCity/Paris",
      "_rev" : "354247349",
      "_key" : "Paris",
      "isCapital" : true,
      "population" : 4000000
    },
    "paths" : [
      {
        "edges" : [
          {
            "_id" : "internationalHighway/356016821",
            "_rev" : "356016821",
            "_key" : "356016821",
            "_from" : "germanCity/Hamburg",
            "_to" : "frenchCity/Paris",
            "distance" : 900
          }
        ],
        "vertices" : [
          "germanCity/Hamburg",
          "frenchCity/Paris"
        ]
      }
    ],
    "distance" : 900
  },
  {
    "startVertex" : "germanCity/Cologne",
    "vertex" : {
      "_id" : "frenchCity/Lyon",
```

```
        "_rev" : "354050741",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      },
      "paths" : [
        {
          "edges" : [
            {
              "_id" : "internationalHighway/356410037",
              "_rev" : "356410037",
              "_key" : "356410037",
              "_from" : "germanCity/Cologne",
              "_to" : "frenchCity/Lyon",
              "distance" : 700
            }
          ],
          "vertices" : [
            "germanCity/Cologne",
            "frenchCity/Lyon"
          ]
        }
      ],
      "distance" : 700
    },
    {
      "startVertex" : "germanCity/Cologne",
      "vertex" : {
        "_id" : "frenchCity/Paris",
        "_rev" : "354247349",
        "_key" : "Paris",
        "isCapital" : true,
        "population" : 4000000
      },
      "paths" : [
        {
          "edges" : [
            {
              "_id" : "internationalHighway/356606645",
              "_rev" : "356606645",
              "_key" : "356606645",
              "_from" : "germanCity/Cologne",
              "_to" : "frenchCity/Paris",
              "distance" : 550
            }
          ],
          "vertices" : [
            "germanCity/Cologne",
            "frenchCity/Paris"
          ]
        }
      ],
      "distance" : 550
    }
  ]
]
```

A route planner example, shortest path from Hamburg and Cologne to Lyon:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._shortestPath([{_id: 'germanCity/Cologne'},{_id:
'germanCity/Munich'}], 'frenchCity/Lyon',
........> {weight : 'distance'});
[
  [
    {
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "361128629",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      },
      "distance" : 700,
      "path" : {
        "vertices" : [
          {
            "_id" : "germanCity/Cologne",
            "_rev" : "360538805",
            "_key" : "Cologne",
            "isCapital" : false,
            "population" : 1000000
          },
          {
            "_id" : "frenchCity/Lyon",
            "_rev" : "361128629",
            "_key" : "Lyon",
            "isCapital" : false,
            "population" : 80000
          }
        ],
        "edges" : [
          {
            "_id" : "internationalHighway/363487925",
            "_rev" : "363487925",
            "_key" : "363487925",
            "_from" : "germanCity/Cologne",
            "_to" : "frenchCity/Lyon",
            "distance" : 700
          }
        ]
      },
      "startVertex" : "germanCity/Cologne",
      "paths" : [
        {
          "vertices" : [
            {
              "_id" : "germanCity/Cologne",
              "_rev" : "360538805",
              "_key" : "Cologne",
              "isCapital" : false,
              "population" : 1000000
            },
            {
              "_id" : "frenchCity/Lyon",
              "_rev" : "361128629",
              "_key" : "Lyon",
```

```
            "isCapital" : false,
            "population" : 80000
          }
        ],
        "edges" : [
          {
            "_id" : "internationalHighway/363487925",
            "_rev" : "363487925",
            "_key" : "363487925",
            "_from" : "germanCity/Cologne",
            "_to" : "frenchCity/Lyon",
            "distance" : 700
          }
        ]
      }
    ]
  }
]
```

_distanceTo

graph._distanceTo(startVertexExample, endVertexExample, options) *The _distanceTo function returns all paths and there distance within a graph.*

This function is a wrapper of [graph._shortestPath](). It does not return the actual path but only the distance between two vertices.

Examples

A route planner example, shortest distance from all german to all french cities:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._distanceTo({}, {}, {weight : 'distance',
endVertexCollectionRestriction : 'frenchCity',
........> startVertexCollectionRestriction : 'germanCity'});
[
  [
    {
      "startVertex" : "germanCity/Hamburg",
      "distance" : 1200,
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "488334005",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      }
    },
    {
      "startVertex" : "germanCity/Hamburg",
      "distance" : 900,
      "vertex" : {
        "_id" : "frenchCity/Paris",
        "_rev" : "488530613",
        "_key" : "Paris",
        "isCapital" : true,
        "population" : 4000000
      }
    },
    {
      "startVertex" : "germanCity/Cologne",
      "distance" : 700,
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "488334005",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      }
    },
    {
      "startVertex" : "germanCity/Cologne",
      "distance" : 550,
      "vertex" : {
        "_id" : "frenchCity/Paris",
        "_rev" : "488530613",
```

```
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      },
      {
        "startVertex" : "germanCity/Berlin",
        "distance" : 1200,
        "vertex" : {
          "_id" : "frenchCity/Paris",
          "_rev" : "488530613",
          "_key" : "Paris",
          "isCapital" : true,
          "population" : 4000000
        }
      },
      {
        "startVertex" : "germanCity/Berlin",
        "distance" : 1100,
        "vertex" : {
          "_id" : "frenchCity/Lyon",
          "_rev" : "488334005",
          "_key" : "Lyon",
          "isCapital" : false,
          "population" : 80000
        }
      }
    ]
]
```

A route planner example, shortest distance from Hamburg and Cologne to Lyon:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> g._distanceTo([{_id: 'germanCity/Cologne'},{_id:
'germanCity/Munich'}], 'frenchCity/Lyon',
........> {weight : 'distance'});
[
  [
    {
      "startVertex" : "germanCity/Cologne",
      "distance" : 700,
      "vertex" : {
        "_id" : "frenchCity/Lyon",
        "_rev" : "482239157",
        "_key" : "Lyon",
        "isCapital" : false,
        "population" : 80000
      }
    }
  ]
]
```

_absoluteEccentricity

Get the [eccentricity](#) of the vertices defined by the examples.

```
graph._absoluteEccentricity(vertexExample, options)
```

The function accepts an id, an example, a list of examples or even an empty example as parameter for vertexExample.

The complexity of the function is described [here](#).

Parameter

- *vertexExample*: Filter the vertices, see [Definition of examples](#)
- *options* (optional): An object defining further options. Can have the following values:
    - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
    - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
    - *startVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for source vertices.
    - *endVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for target vertices.
    - *edgeExamples*: Filter the edges to be followed, see [Definition of examples](#)
    - *algorithm*: The algorithm to calculate the shortest paths, possible values are [Floyd-Warshall](#) and [Dijkstra](#).
    - *weight*: The name of the attribute of the edges containing the weight.
    - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the eccentricity can not be calculated.

Examples

A route planner example, the absolute eccentricity of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> db._query("RETURN GRAPH_ABSOLUTE_ECCENTRICITY("
........>    + "'routeplanner', {})"
........> ).toArray();
[
  {
    "germanCity/Hamburg" : 1,
    "germanCity/Cologne" : 1,
    "germanCity/Berlin" : 1,
    "frenchCity/Paris" : 1,
    "frenchCity/Lyon" : 1
  }
]
```

A route planner example, the absolute eccentricity of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteEccentricity({}, {weight : 'distance'});
```

```
[
  {
    "germanCity/Berlin" : 1200,
    "germanCity/Hamburg" : 1200,
    "germanCity/Cologne" : 850,
    "frenchCity/Paris" : 1200,
    "frenchCity/Lyon" : 1200
  }
]
```

A route planner example, the absolute eccentricity of all cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteEccentricity({},
{startVertexCollectionRestriction : 'germanCity',
........> direction : 'outbound', weight : 'distance'});
[
  {
    "germanCity/Hamburg" : 1200,
    "germanCity/Berlin" : 1200,
    "germanCity/Cologne" : 700
  }
]
```

_eccentricity

graph._eccentricity(vertexExample, options) *Get the normalized* eccentricity *of the vertices defined by the examples.*

Similar to _absoluteEccentricity but returns a normalized result.

The complexity of the function is described here.

Examples

A route planner example, the eccentricity of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._eccentricity();
[
  {
    "germanCity/Hamburg" : 1,
    "germanCity/Cologne" : 1,
    "germanCity/Berlin" : 1,
    "frenchCity/Paris" : 1,
    "frenchCity/Lyon" : 1
  }
]
```

A route planner example, the weighted eccentricity.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._eccentricity({weight : 'distance'});
[
  {
    "germanCity/Berlin" : 0.7083333333333335,
    "germanCity/Hamburg" : 0.708333333333335,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0.7083333333333335,
    "frenchCity/Lyon" : 0.708333333333335
  }
]
```

_absoluteCloseness

`graph._absoluteCloseness(vertexExample, options)` *Get the* [closeness](#) *of the vertices defined by the examples.*

The function accepts an id, an example, a list of examples or even an empty example as parameter for *vertexExample*.

The complexity of the function is described [here](#).

Parameter

- *vertexExample*: Filter the vertices, see [Definition of examples](#)
- *options* (optional): An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
  - *startVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for source vertices.
  - *endVertexCollectionRestriction* : One or a list of vertex-collection names that should be considered for target vertices.
  - *edgeExamples*: Filter the edges to be followed, see [Definition of examples](#)
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are [Floyd-Warshall](#) and [Dijkstra](#).
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the closeness can not be calculated.

Examples

A route planner example, the absolute closeness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteCloseness({});
[
  {
    "germanCity/Berlin" : 4,
    "germanCity/Cologne" : 4,
    "germanCity/Hamburg" : 4,
    "frenchCity/Paris" : 4,
    "frenchCity/Lyon" : 4
  }
]
```

A route planner example, the absolute closeness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteCloseness({}, {weight : 'distance'});
[
  {
    "germanCity/Hamburg" : 3000,
```

```
    "germanCity/Cologne" : 2600,
    "germanCity/Berlin" : 3550,
    "frenchCity/Paris" : 3200,
    "frenchCity/Lyon" : 3550
  }
]
```

A route planner example, the absolute closeness of all German Cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteCloseness({},
{startVertexCollectionRestriction : 'germanCity',
........> direction : 'outbound', weight : 'distance'});
[
  {
    "germanCity/Berlin" : 3550,
    "germanCity/Hamburg" : 2600,
    "germanCity/Cologne" : 1250
  }
]
```

_closeness

`graph._closeness(options)` *Get the normalized* closeness *of graphs vertices.*

Similar to _absoluteCloseness but returns a normalized value.

The complexity of the function is described here.

Examples

A route planner example, the normalized closeness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._closeness();
[
  {
    "germanCity/Hamburg" : 1,
    "germanCity/Cologne" : 1,
    "germanCity/Berlin" : 1,
    "frenchCity/Paris" : 1,
    "frenchCity/Lyon" : 1
  }
]
```

A route planner example, the closeness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._closeness({weight : 'distance'});
[
  {
    "germanCity/Berlin" : 0.7323943661971831,
    "germanCity/Hamburg" : 0.8666666666666666,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0.8125,
    "frenchCity/Lyon" : 0.7323943661971831
  }
]
```

A route planner example, the closeness of all cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._closeness({direction : 'outbound', weight :
'distance'});
[
  {
    "germanCity/Hamburg" : 0.7279057017543861,
    "germanCity/Berlin" : 1,
    "frenchCity/Paris" : 0.3355263157894737,
    "germanCity/Cologne" : 0.5991541353383459,
    "frenchCity/Lyon" : 0
```

```
    }
]
```

_absoluteBetweenness

`graph._absoluteBetweenness(options)` *Get the* [betweenness](#) *of all vertices in the graph.*

The complexity of the function is described [here](#).

Parameter

- *options* (optional): An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the betweeness can not be calculated.

Examples

A route planner example, the absolute betweenness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteBetweenness({});
[
  {
    "germanCity/Berlin" : 0,
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 0,
    "frenchCity/Lyon" : 0,
    "frenchCity/Paris" : 0
  }
]
```

A route planner example, the absolute betweenness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._absoluteBetweenness({weight : 'distance'});
[
  {
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 2,
    "germanCity/Berlin" : 0,
    "frenchCity/Lyon" : 0,
    "frenchCity/Paris" : 0
  }
]
```

A route planner example, the absolute betweenness of all cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
```

```
arangosh> graph._absoluteBetweenness({direction : 'outbound',
weight : 'distance'});
[
  {
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0,
    "frenchCity/Lyon" : 0,
    "germanCity/Berlin" : 0
  }
]
```

_betweenness

`graph_module._betweenness(options)` *Get the normalized* [betweenness](#) *of graphs vertices.*

Similar to [_absoluteBetweeness](#) but returns normalized values.

Examples

A route planner example, the betweenness of all locations.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._betweenness();
[
  {
    "germanCity/Berlin" : null,
    "germanCity/Hamburg" : null,
    "germanCity/Cologne" : null,
    "frenchCity/Lyon" : null,
    "frenchCity/Paris" : null
  }
]
```

A route planner example, the betweenness of all locations. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._betweenness({weight : 'distance'});
[
  {
    "germanCity/Berlin" : 0,
    "germanCity/Cologne" : 1,
    "germanCity/Hamburg" : 0,
    "frenchCity/Paris" : 0,
    "frenchCity/Lyon" : 0
  }
]
```

A route planner example, the betweenness of all cities regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._betweenness({direction : 'outbound', weight :
'distance'});
[
  {
    "germanCity/Berlin" : 0,
    "germanCity/Hamburg" : 0,
    "germanCity/Cologne" : 1,
    "frenchCity/Paris" : 0,
    "frenchCity/Lyon" : 0
  }
]
```

_radius

`graph._radius(options)` *Get the [radius](#) of a graph.*

The complexity of the function is described [here](#).

Parameter

- *options* (optional): An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are [Floyd-Warshall](#) and [Dijkstra](#).
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the radius can not be calculated.

Examples

A route planner example, the radius of the graph.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._radius();
[
   1
]
```

A route planner example, the radius of the graph. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._radius({weight : 'distance'});
[
   850
]
```

A route planner example, the radius of the graph regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._radius({direction : 'outbound', weight :
'distance'});
[
   550
]
```

_diameter

`graph._diameter(graphName, options)` *Get the [diameter](#) of a graph.*

The complexity of the function is described [here](#).

Parameter

- *options* (optional): An object defining further options. Can have the following values:
  - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
  - *algorithm*: The algorithm to calculate the shortest paths, possible values are [Floyd-Warshall](#) and [Dijkstra](#).
  - *weight*: The name of the attribute of the edges containing the weight.
  - *defaultWeight*: Only used with the option *weight*. If an edge does not have the attribute named as defined in option *weight* this default is used as weight. If no default is supplied the default would be positive infinity so the path and hence the radius can not be calculated.

Examples

A route planner example, the diameter of the graph.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._diameter();
[
  1
]
```

A route planner example, the diameter of the graph. This considers the actual distances.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._diameter({weight : 'distance'});
[
  1200
]
```

A route planner example, the diameter of the graph regarding only outbound paths.

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("routeplanner");
arangosh> graph._diameter({direction : 'outbound', weight :
'distance'});
[
  1200
]
```

Fluent AQL Interface

This chapter describes a fluent interface to query your graph. The philosophy of this interface is to select a group of starting elements (vertices or edges) at first and from there on explore the graph with your query by selecting connected elements.

As an example you can start with a set of vertices, select their direct neighbors and finally their outgoing edges.

The result of this query will be the set of outgoing edges. For each part of the query it is possible to further refine the resulting set of elements by giving examples for them.

Definition of examples

For many of the following functions *examples* can be passed in as a parameter. *Examples* are used to filter the result set for objects that match the conditions. These *examples* can have the following values:

- *null*, there is no matching executed all found results are valid.
- A *string*, only results are returned, which *_id* equal the value of the string
- An example *object*, defining a set of attributes. Only results having these attributes are matched.
- A *list* containing example *objects* and/or *strings*. All results matching at least one of the elements in the list are returned.

Starting Points

This section describes the entry points for the fluent interface. The philosophy of this module is to start with a specific subset of vertices or edges and from there on iterate over the graph.

Therefore you get exactly this two entry points:

- Select a set of edges
- Select a set of vertices

Edges

`graph._edges(examples)` *Select some edges from the graph.*

Creates an AQL statement to select a subset of the edges stored in the graph. This is one of the entry points for the fluent AQL interface. It will return a mutable AQL statement which can be further refined, using the functions described below. The resulting set of edges can be filtered by defining one or more *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

In the examples the *toArray* function is used to print the result. The description of this module can be found below.

To request unfiltered edges:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._edges().toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "523133621",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/aliceAndCharly",
    "_rev" : "523395765",
    "_key" : "aliceAndCharly",
    "_from" : "female/alice",
    "_to" : "male/charly",
    "type" : "friend"
  },
  {
    "_id" : "relation/bobAndDiana",
    "_rev" : "523788981",
    "_key" : "bobAndDiana",
    "_from" : "male/bob",
    "_to" : "female/diana",
    "type" : "friend"
  },
```

```
    {
      "_id" : "relation/charlyAndDiana",
      "_rev" : "523592373",
      "_key" : "charlyAndDiana",
      "_from" : "male/charly",
      "_to" : "female/diana",
      "type" : "married"
    }
  ]
```

To request filtered edges:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._edges({type: "married"}).toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "98132661",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/charlyAndDiana",
    "_rev" : "98591413",
    "_key" : "charlyAndDiana",
    "_from" : "male/charly",
    "_to" : "female/diana",
    "type" : "married"
  }
]
```

Vertices

`graph._vertices(examples)` *Select some vertices from the graph.*

Creates an AQL statement to select a subset of the vertices stored in the graph. This is one of the entry points for the fluent AQL interface. It will return a mutable AQL statement which can be further refined, using the functions described below. The resulting set of edges can be filtered by defining one or more *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

In the examples the *toArray* function is used to print the result. The description of this module can be found below.

To request unfiltered vertices:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
```

```
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._vertices().toArray();
[
  {
    "_id" : "female/diana",
    "_rev" : "692413109",
    "_key" : "diana",
    "name" : "Diana"
  },
  {
    "_id" : "female/alice",
    "_rev" : "691692213",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "male/bob",
    "_rev" : "692019893",
    "_key" : "bob",
    "name" : "Bob"
  },
  {
    "_id" : "male/charly",
    "_rev" : "692216501",
    "_key" : "charly",
    "name" : "Charly"
  }
]
```

To request filtered vertices:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> graph._vertices([{name: "Alice"}, {name:
"Bob"}]).toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "518349493",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "male/bob",
    "_rev" : "518677173",
    "_key" : "bob",
    "name" : "Bob"
  }
]
```

Working with the query cursor

The fluent query object handles cursor creation and maintenance for you. A cursor will be created as soon as you request the first result. If you are unhappy with the current result and want to refine it further you can execute a further step in the query which cleans up the cursor for you. In this interface you get the complete functionality available for general AQL cursors directly on your query. The cursor functionality is described in this section.

ToArray

`graph_query.toArray()` *Returns an array containing the complete result.*

This function executes the generated query and returns the entire result as one array. ToArray does not return the generated query anymore and hence can only be the endpoint of a query. However keeping a reference to the query before executing allows to chain further statements to it.

Examples

To collect the entire result of a query toArray can be used:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices();
arangosh> query.toArray();
[
  {
    "_id" : "female/diana",
    "_rev" : "370172597",
    "_key" : "diana",
    "name" : "Diana"
  },
  {
    "_id" : "female/alice",
    "_rev" : "369451701",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "male/bob",
    "_rev" : "369779381",
    "_key" : "bob",
    "name" : "Bob"
  },
  {
    "_id" : "male/charly",
    "_rev" : "369975989",
    "_key" : "charly",
    "name" : "Charly"
  }
]
```

HasNext

`graph_query.hasNext()` *Checks if the query has further results.*

The generated statement maintains a cursor for you. If this cursor is already present *hasNext()* will use this cursors position to determine if there are further results available. If the query has not yet been executed

hasNext() will execute it and create the cursor for you.

Examples

Start query execution with hasNext:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices();
arangosh> query.hasNext();
true
```

Iterate over the result as long as it has more elements:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices();
arangosh> while (query.hasNext()) {
........>   var entry = query.next();
........>   // Do something with the entry
........> }
```

Next

`graph_query.next()` *Request the next element in the result.*

The generated statement maintains a cursor for you. If this cursor is already present *next()* will use this cursors position to deliver the next result. Also the cursor position will be moved by one. If the query has not yet been executed *next()* will execute it and create the cursor for you. It will throw an error of your query has no further results.

Examples

Request some elements with next:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices();
arangosh> query.next();
{
  "_id" : "female/diana",
  "_rev" : "739402421",
  "_key" : "diana",
  "name" : "Diana"
}
arangosh> query.next();
{
  "_id" : "female/alice",
  "_rev" : "738681525",
  "_key" : "alice",
  "name" : "Alice"
}
arangosh> query.next();
```

```
{
  "_id" : "male/bob",
  "_rev" : "739009205",
  "_key" : "bob",
  "name" : "Bob"
}
arangosh> query.next();
{
  "_id" : "male/charly",
  "_rev" : "739205813",
  "_key" : "charly",
  "name" : "Charly"
}
```

The cursor is recreated if the query is changed:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices();
arangosh> query.next();
{
  "_id" : "female/diana",
  "_rev" : "493642421",
  "_key" : "diana",
  "name" : "Diana"
}
arangosh> query.edges();
[ GraphAQL social
.vertices()
.edges() ]
arangosh> query.next();
{
  "_id" : "relation/charlyAndDiana",
  "_rev" : "494428853",
  "_key" : "charlyAndDiana",
  "_from" : "male/charly",
  "_to" : "female/diana",
  "type" : "married"
}
```

Count

`graph_query.count()` *Returns the number of returned elements if the query is executed.*

This function determines the amount of elements to be expected within the result of the query. It can be used at the beginning of execution of the query before using *next()* or in between *next()* calls. The query object maintains a cursor of the query for you. *count()* does not change the cursor position.

Examples

To count the number of matched elements:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices();
```

```
arangosh> query.count();
4
```

Fluent queries

After the selection of the entry point you can now query your graph in a fluent way, meaning each of the functions on your query returns the query again. Hence it is possible to chain arbitrary many executions one after the other. In this section all available query statements are described.

Edges

`graph_query.edges(examples)` *Select all edges for the vertices selected before.*

Creates an AQL statement to select all edges for each of the vertices selected in the step before. This will include *inbound* as well as *outbound* edges. The resulting set of edges can be filtered by defining one or more *examples*.

The complexity of this method is **O(n*m^x)** with *n* being the vertices defined by the parameter vertexExamplex, *m* the average amount of edges of a vertex and *x* the maximal depths. Hence the default call would have a complexity of **O(n*m)**;

Parameter

- *examples*: See [Definition of examples](#)

Examples

To request unfiltered edges:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.edges().toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "294347445",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/aliceAndCharly",
    "_rev" : "294609589",
    "_key" : "aliceAndCharly",
    "_from" : "female/alice",
    "_to" : "male/charly",
    "type" : "friend"
  },
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "294347445",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/bobAndDiana",
```

```
    "_rev" : "295002805",
    "_key" : "bobAndDiana",
    "_from" : "male/bob",
    "_to" : "female/diana",
    "type" : "friend"
  }
]
```

To request filtered edges by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.edges({type: "married"}).toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "561668789",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "561668789",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  }
]
```

To request filtered edges by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.edges([{type: "married"}, {type:
"friend"}]).toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "715350709",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/aliceAndCharly",
    "_rev" : "715612853",
    "_key" : "aliceAndCharly",
```

```
      "_from" : "female/alice",
      "_to" : "male/charly",
      "type" : "friend"
    },
    {
      "_id" : "relation/aliceAndBob",
      "_rev" : "715350709",
      "_key" : "aliceAndBob",
      "_from" : "female/alice",
      "_to" : "male/bob",
      "type" : "married"
    },
    {
      "_id" : "relation/bobAndDiana",
      "_rev" : "716006069",
      "_key" : "bobAndDiana",
      "_from" : "male/bob",
      "_to" : "female/diana",
      "type" : "friend"
    }
]
```

OutEdges

`graph_query.outEdges(examples)` *Select all outbound edges for the vertices selected before.*

Creates an AQL statement to select all *outbound* edges for each of the vertices selected in the step before. The resulting set of edges can be filtered by defining one or more *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

To request unfiltered outbound edges:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.outEdges().toArray();
[
    {
      "_id" : "relation/aliceAndBob",
      "_rev" : "275735221",
      "_key" : "aliceAndBob",
      "_from" : "female/alice",
      "_to" : "male/bob",
      "type" : "married"
    },
    {
      "_id" : "relation/aliceAndCharly",
      "_rev" : "275997365",
      "_key" : "aliceAndCharly",
      "_from" : "female/alice",
      "_to" : "male/charly",
```

```
      "type" : "friend"
  },
  {
    "_id" : "relation/bobAndDiana",
    "_rev" : "276390581",
    "_key" : "bobAndDiana",
    "_from" : "male/bob",
    "_to" : "female/diana",
    "type" : "friend"
  }
]
```

To request filtered outbound edges by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.outEdges({type: "married"}).toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "416637621",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  }
]
```

To request filtered outbound edges by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.outEdges([{type: "married"}, {type:
"friend"}]).toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "651059893",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  },
  {
    "_id" : "relation/aliceAndCharly",
    "_rev" : "651322037",
    "_key" : "aliceAndCharly",
    "_from" : "female/alice",
    "_to" : "male/charly",
    "type" : "friend"
  },
```

```
  {
    "_id" : "relation/bobAndDiana",
    "_rev" : "651715253",
    "_key" : "bobAndDiana",
    "_from" : "male/bob",
    "_to" : "female/diana",
    "type" : "friend"
  }
]
```

InEdges

`graph_query.inEdges(examples)` *Select all inbound edges for the vertices selected before.*

Creates an AQL statement to select all *inbound* edges for each of the vertices selected in the step before. The resulting set of edges can be filtered by defining one or more *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

To request unfiltered inbound edges:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.inEdges().toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "247620277",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  }
]
```

To request filtered inbound edges by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.inEdges({type: "married"}).toArray();
[
  {
    "_id" : "relation/aliceAndBob",
    "_rev" : "69493429",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
```

```
        "type" : "married"
    }
]
```

To request filtered inbound edges by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices([{name: "Alice"}, {name:
"Bob"}]);
arangosh> query.inEdges([{type: "married"}, {type:
"friend"}]).toArray();
[
    {
        "_id" : "relation/aliceAndBob",
        "_rev" : "223044277",
        "_key" : "aliceAndBob",
        "_from" : "female/alice",
        "_to" : "male/bob",
        "type" : "married"
    }
]
```

Vertices

`graph_query.vertices(examples)` *Select all vertices connected to the edges selected before.*

Creates an AQL statement to select all vertices for each of the edges selected in the step before. This includes all vertices contained in *_from* as well as *_to* attribute of the edges. The resulting set of vertices can be filtered by defining one or more *examples*.

Parameter

* *examples*: See [Definition of examples](#)

Examples

To request unfiltered vertices:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.vertices().toArray();
[
    {
        "_id" : "female/alice",
        "_rev" : "512123573",
        "_key" : "alice",
        "name" : "Alice"
    },
    {
        "_id" : "male/bob",
        "_rev" : "512451253",
        "_key" : "bob",
        "name" : "Bob"
```

```
    },
    {
      "_id" : "female/diana",
      "_rev" : "512844469",
      "_key" : "diana",
      "name" : "Diana"
    },
    {
      "_id" : "male/charly",
      "_rev" : "512647861",
      "_key" : "charly",
      "name" : "Charly"
    }
  ]
```

To request filtered vertices by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.vertices({name: "Alice"}).toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "365716149",
    "_key" : "alice",
    "name" : "Alice"
  }
]
```

To request filtered vertices by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.vertices([{name: "Alice"}, {name:
"Charly"}]).toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "289563317",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "male/charly",
    "_rev" : "290087605",
    "_key" : "charly",
    "name" : "Charly"
  }
]
```

FromVertices

`graph_query.fromVertices(examples)` *Select all source vertices of the edges selected before.*

Creates an AQL statement to select the set of vertices where the edges selected in the step before start at. This includes all vertices contained in *_from* attribute of the edges. The resulting set of vertices can be filtered by defining one or more *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

To request unfiltered source vertices:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.fromVertices().toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "212165301",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "male/charly",
    "_rev" : "212689589",
    "_key" : "charly",
    "name" : "Charly"
  }
]
```

To request filtered source vertices by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.fromVertices({name: "Alice"}).toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "675111605",
    "_key" : "alice",
    "name" : "Alice"
  }
]
```

To request filtered source vertices by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.fromVertices([{name: "Alice"}, {name:
```

```
"Charly"}]).toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "718889653",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "male/charly",
    "_rev" : "719413941",
    "_key" : "charly",
    "name" : "Charly"
  }
]
```

ToVertices

`graph_query.toVertices(examples)` *Select all vertices targeted by the edges selected before.*

Creates an AQL statement to select the set of vertices where the edges selected in the step before end in. This includes all vertices contained in *_to* attribute of the edges. The resulting set of vertices can be filtered by defining one or more *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

To request unfiltered target vertices:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices().toArray();
[
  {
    "_id" : "male/bob",
    "_rev" : "138764981",
    "_key" : "bob",
    "name" : "Bob"
  },
  {
    "_id" : "female/diana",
    "_rev" : "139158197",
    "_key" : "diana",
    "name" : "Diana"
  }
]
```

To request filtered target vertices by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
```

```
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices({name: "Bob"}).toArray();
[
   {
     "_id" : "male/bob",
     "_rev" : "536109749",
     "_key" : "bob",
     "name" : "Bob"
   }
]
```

To request filtered target vertices by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices([{name: "Bob"}, {name:
"Diana"}]).toArray();
[
   {
     "_id" : "male/bob",
     "_rev" : "33776309",
     "_key" : "bob",
     "name" : "Bob"
   },
   {
     "_id" : "female/diana",
     "_rev" : "34169525",
     "_key" : "diana",
     "name" : "Diana"
   }
]
```

Neighbors

`graph_query.neighbors(examples, options)` *Select all neighbors of the vertices selected in the step before.*

Creates an AQL statement to select all neighbors for each of the vertices selected in the step before. The resulting set of vertices can be filtered by defining one or more *examples*.

Parameter

- *examples*: See Definition of examples
- *options* (optional): An object defining further options. Can have the following values:
    - *direction*: The direction of the edges. Possible values are *outbound*, *inbound* and *any* (default).
    - *edgeExamples*: Filter the edges to be followed, see Definition of examples
    - *edgeCollectionRestriction* : One or a list of edge-collection names that should be considered to be on the path.
    - *vertexCollectionRestriction* : One or a list of vertex-collection names that should be considered on the intermediate vertex steps.
    - *minDepth*: Defines the minimal number of intermediate steps to neighbors (default is 1).
    - *maxDepth*: Defines the maximal number of intermediate steps to neighbors (default is 1).

Examples

To request unfiltered neighbors:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.neighbors().toArray();
[
  {
    "_id" : "male/bob",
    "_rev" : "387670709",
    "_key" : "bob",
    "name" : "Bob"
  },
  {
    "_id" : "male/charly",
    "_rev" : "387867317",
    "_key" : "charly",
    "name" : "Charly"
  }
]
```

To request filtered neighbors by a single example:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.neighbors({name: "Bob"}).toArray();
[
  {
    "_id" : "male/bob",
    "_rev" : "156000949",
    "_key" : "bob",
    "name" : "Bob"
  }
]
```

To request filtered neighbors by multiple examples:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.vertices([{name: "Bob"}, {name:
"Charly"}]).toArray();
[
  {
    "_id" : "male/bob",
    "_rev" : "55206581",
    "_key" : "bob",
    "name" : "Bob"
  },
  {
    "_id" : "male/charly",
    "_rev" : "55403189",
```

```
        "_key" : "charly",
        "name" : "Charly"
    }
]
```

Restrict

`graph_query.restrict(restrictions)` *Restricts the last statement in the chain to return only elements of a specified set of collections*

By default all collections in the graph are searched for matching elements whenever vertices and edges are requested. Using *restrict* after such a statement allows to restrict the search to a specific set of collections within the graph. Restriction is only applied to this one part of the query. It does not effect earlier or later statements.

Parameter

- *restrictions*: Define either one or a list of collections in the graph. Only elements from these collections are taken into account for the result.

Examples

Request all directly connected vertices unrestricted:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.edges().vertices().toArray();
[
    {
      "_id" : "female/alice",
      "_rev" : "772170421",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "_id" : "male/bob",
      "_rev" : "772498101",
      "_key" : "bob",
      "name" : "Bob"
    },
    {
      "_id" : "female/alice",
      "_rev" : "772170421",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "_id" : "male/charly",
      "_rev" : "772694709",
      "_key" : "charly",
      "name" : "Charly"
    }
]
```

Apply a restriction to the directly connected vertices:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.edges().vertices().restrict("female").toArray();
[
  {
    "_id" : "female/alice",
    "_rev" : "373187253",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "female/alice",
    "_rev" : "373187253",
    "_key" : "alice",
    "name" : "Alice"
  }
]
```

Restriction of a query is only valid for collections known to the graph: //

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.edges().vertices().restrict(["female", "male",
"products"]).toArray();
[ArangoError 10: vertex collections: products are not known to the
graph]
```

Filter

`graph_query.filter(examples)` *Filter the result of the query*

This can be used to further specfiy the expected result of the query. The result set is reduced to the set of elements that matches the given *examples*.

Parameter

- *examples*: See [Definition of examples](#)

Examples

Request vertices unfiltered:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices().toArray();
[
  {
    "_id" : "male/bob",
    "_rev" : "709255861",
    "_key" : "bob",
    "name" : "Bob"
```

```
  },
  {
    "_id" : "female/diana",
    "_rev" : "709649077",
    "_key" : "diana",
    "name" : "Diana"
  }
]
```

Request vertices filtered:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices().filter({name: "Alice"}).toArray();
[ ]
```

Request edges unfiltered:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices().outEdges().toArray();
[
  {
    "_id" : "relation/bobAndDiana",
    "_rev" : "380199605",
    "_key" : "bobAndDiana",
    "_from" : "male/bob",
    "_to" : "female/diana",
    "type" : "friend"
  }
]
```

Request edges filtered:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._edges({type: "married"});
arangosh> query.toVertices().outEdges().filter({type:
"married"}).toArray();
[ ]
```

Path

`graph_query.path()` *The result of the query is the path to all elements.*

By defaut the result of the generated AQL query is the set of elements passing the last matches. So having a `vertices()` query as the last step the result will be set of vertices. Using `path()` as the last action before requesting the result will modify the result such that the path required to find the set vertices is returned.

Examples

Request the iteratively explored path using vertices and edges:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.outEdges().toVertices().path().toArray();
[
  [
    {
      "_id" : "female/alice",
      "_rev" : "314401461",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "_id" : "relation/aliceAndBob",
      "_rev" : "315450037",
      "_key" : "aliceAndBob",
      "_from" : "female/alice",
      "_to" : "male/bob",
      "type" : "married"
    },
    {
      "_id" : "male/bob",
      "_rev" : "314729141",
      "_key" : "bob",
      "name" : "Bob"
    }
  ],
  [
    {
      "_id" : "female/alice",
      "_rev" : "314401461",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "_id" : "relation/aliceAndCharly",
      "_rev" : "315712181",
      "_key" : "aliceAndCharly",
      "_from" : "female/alice",
      "_to" : "male/charly",
      "type" : "friend"
    },
    {
      "_id" : "male/charly",
      "_rev" : "314925749",
      "_key" : "charly",
      "name" : "Charly"
    }
  ]
]
```

When requesting neighbors the path to these neighbors is expanded:

```
arangosh> var examples = require("org/arangodb/graph-
examples/example-graph.js");
arangosh> var graph = examples.loadGraph("social");
arangosh> var query = graph._vertices({name: "Alice"});
arangosh> query.neighbors().path().toArray();
[
  [
    {
      "_id" : "female/alice",
      "_rev" : "58614453",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "edges" : [
        {
          "_id" : "relation/aliceAndBob",
          "_rev" : "59663029",
          "_key" : "aliceAndBob",
          "_from" : "female/alice",
          "_to" : "male/bob",
          "type" : "married"
        }
      ],
      "vertices" : [
        {
          "_id" : "female/alice",
          "_rev" : "58614453",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "male/bob",
          "_rev" : "58942133",
          "_key" : "bob",
          "name" : "Bob"
        }
      ]
    }
  ],
  [
    {
      "_id" : "female/alice",
      "_rev" : "58614453",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "edges" : [
        {
          "_id" : "relation/aliceAndCharly",
          "_rev" : "59925173",
          "_key" : "aliceAndCharly",
          "_from" : "female/alice",
          "_to" : "male/charly",
          "type" : "friend"
        }
      ],
      "vertices" : [
```

```
        {
          "_id" : "female/alice",
          "_rev" : "58614453",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "male/charly",
          "_rev" : "59138741",
          "_key" : "charly",
          "name" : "Charly"
        }
      ]
    }
  ]
]
```

Graphs

Warning: Deprecated

This module is deprecated and will be removed soon. Please use General Graphs instead.

A Graph consists of vertices and edges. The vertex collection contains the documents forming the vertices. The edge collection contains the documents forming the edges. Together both collections form a graph. Assume that the vertex collection is called `vertices` and the edges collection `edges`, then you can build a graph using the Graph constructor.

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> g1 = new Graph("graph", "vertices", "edges");
Graph("vertices", "edges")
```

It is possible to use different edges with the same vertices. For instance, to build a new graph with a different edge collection use

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> g2 = new Graph("graph", "vertices", "alternativeEdges");
Graph("vertices", "alternativeEdges")
```

It is, however, impossible to use different vertices with the same edges. Edges are tied to the vertices.

Graph Constructors and Methods

The graph module provides basic functions dealing with graph structures. The examples assume

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> g = new Graph("graph", "vertices", "edges");
Graph("graph")
```

`Graph(name, vertices, edges)`

Constructs a new graph object using the collection vertices for all vertices and the collection edges for all edges. Note that it is possible to construct two graphs with the same vertex set, but different edge sets.

`Graph(name)`

Returns a known graph.

Examples

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> new Graph("graph", db.vertices, db.edges);
Graph("graph")

arango> new Graph("graph", "vertices", "edges");
Graph("graph")
```

`graph.addEdge( out, in, id)`

Creates a new edge from out to in and returns the edge object. The identifier id must be a unique identifier or null. out and in can either be vertices or their IDs

`graph.addEdge( out, in, id, label)`

Creates a new edge from out to in with label and returns the edge object. out and in can either be vertices or their IDs

`graph.addEdge( out, in, id, data)`

Creates a new edge and returns the edge object. The edge contains the properties defined in data. out and in can either be vertices or their IDs

`graph.addEdge( out, in, id, label, data)`

Creates a new edge and returns the edge object. The edge has the label label and contains the properties defined in data. out and in can either be vertices or their IDs

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)
```

```
arango> e = g.addEdge(v1, v2, 3);
Edge(3)

arango> e = g.addEdge(v1, v2, 4, "1->2", { name : "Emil");
Edge(4)
```

`graph.addVertex( id)`

Creates a new vertex and returns the vertex object. The identifier id must be a unique identifier or null.

`graph.addVertex( id, data)`

Creates a new vertex and returns the vertex object. The vertex contains the properties defined in data.

Examples

Without any properties:

```
arango> v = g.addVertex("hugo");
Vertex("hugo")
```

With given properties:

```
arango> v = g.addVertex("Emil", { age : 123 });
Vertex("Emil")

arango> v.getProperty("age");
123
```

`graph.getEdges()`

Returns an iterator for all edges of the graph. The iterator supports the methods hasNext and next.

Examples

```
arango> f = g.getEdges();
[edge iterator]

arango> f.hasNext();
true

arango> e = f.next();
Edge("4636053")
```

`graph.getVertex( id)`

Returns the vertex identified by id or null.

Examples

```
arango> g.addVertex(1);
Vertex(1)
```

```
arango> g.getVertex(1)
Vertex(1)
```

graph.getVertices()

Returns an iterator for all vertices of the graph. The iterator supports the methods hasNext and next.

Examples

```
arango> f = g.getVertices();
[vertex iterator]

arango> f.hasNext();
true

arango> v = f.next();
Vertex(18364)
```

graph.removeVertex(vertex, waitForSync)

Deletes the vertex and all its edges.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> e = g.addEdge(v1, v2, 3);
Edge(3)

arango> g.removeVertex(v1);

arango> v2.edges();
[ ]
```

graph.removeEdge(vertex, waitForSync)

Deletes the edge. Note that the in and out vertices are left untouched.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> e = g.addEdge(v1, v2, 3);
Edge(3)

arango> g.removeEdge(e);
```

```
arango> v2.edges();
[ ]
```

`graph.drop( waitForSync)`

Drops the graph, the vertices, and the edges. Handle with care.

`graph.getAll()`

Returns all available graphs.

`graph.geodesics( options)`

Return all shortest paths An optional options JSON object can be specified to control the result. options can have the following sub-attributes:

grouped: if not specified or set to false, the result will be a flat list. If set to true, the result will be a list containing list of paths, grouped for each combination of source and target. threshold: if not specified, all paths will be returned. If threshold is true, only paths with a minimum length of 3 will be returned

`graph.measurement( measurement)`

Calculates the diameter or radius of a graph. measurement can either be:

- diameter: to calculate the diameter
- radius: to calculate the radius

`graph.normalizedMeasurement( measurement)`

Calculates the normalized degree, closeness, betweenness or eccentricity of all vertices in a graph measurement can either be:

- closeness: to calculate the closeness
- betweenness: to calculate the betweenness
- eccentricity: to calculate the eccentricity

Vertex Methods

`vertex.addInEdge( peer, id)`

Creates a new edge from peer to vertex and returns the edge object. The identifier id must be a unique identifier or null.

`vertex.addInEdge( peer, id, label)`

Creates a new edge from peer to vertex with given label and returns the edge object.

`vertex.addInEdge( peer, id, label, data)`

Creates a new edge from peer to vertex with given label and properties defined in data. Returns the edge object.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> v1.addInEdge(v2, "2 -> 1");
Edge("2 -> 1")

arango> v1.getInEdges();
[ Edge("2 -> 1") ]
arango> v1.addInEdge(v2, "D", "knows", { data : 1 });
Edge("D")

arango> v1.getInEdges();
[ Edge("K"), Edge("2 -> 1"), Edges("D") ]
```

`vertex.addOutEdge( peer)`

Creates a new edge from vertex to peer and returns the edge object.

`vertex.addOutEdge( peer, label)`

Creates a new edge from vertex to peer with given label and returns the edge object.

`vertex.addOutEdge( peer, label, data)`

Creates a new edge from vertex to peer with given label and properties defined in data. Returns the edge object.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> v1.addOutEdge(v2, "1->2");
```

```
Edge("1->2")

arango> v1.getOutEdges();
[ Edge(1->2") ]
arango> v1.addOutEdge(v2, 3, "knows");
Edge(3)
arango> v1.addOutEdge(v2, 4, "knows", { data : 1 });
Edge(4)
```

`vertex.edges()`

Returns a list of in- or outbound edges of the vertex.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex();
Vertex(2)

arango> e = g.addEdge(v1, v2, "1->2");
Edge("1->2")

arango> v1.edges();
[  Edge("1->2")  ]

arango> v2.edges();
[  Edge("1->2")  ]
```

`vertex.getId()`

Returns the identifier of the vertex. If the vertex was deleted, then undefined is returned.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.getId();
"1"
```

`vertex.getInEdges( label, ...)`

Returns a list of inbound edges of the vertex with given label(s).

Examples

```
arango> v1 = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v2 = g.addVertex(2, { name : "Emil" });
Vertex(2)

arango> e1 = g.addEdge(v1, v2, 3, "knows");
Edge(3)
```

```
arango> e2 = g.addEdge(v1, v2, 4, "hates");
Edge(4)

arango> v2.getInEdges();
[ Edge(3), Edge(4) ]

arango> v2.getInEdges("knows");
[ Edge(3) ]

arango> v2.getInEdges("hates");
[ Edge(4) ]

arango> v2.getInEdges("knows", "hates");
[ Edge(3), Edge(4) ]
```

`vertex.getOutEdges( label, ...)`

Returns a list of outbound edges of the vertex with given label(s).

Examples

```
arango> v1 = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v2 = g.addVertex(2, { name : "Emil" });
Vertex(2)

arango> e1 = g.addEdge(v1, v2, 3, "knows");
Edge(3)

arango> e2 = g.addEdge(v1, v2, 4, "hates");
Edge(4)

arango> v1.getOutEdges();
[ Edge(3), Edge(4) ]

arango> v1.getOutEdges("knows");
[ Edge(3) ]

arango> v1.getOutEdges("hates");
[ Edge(4) ]

arango> v1.getOutEdges("knows", "hates");
[ Edge(3), Edge(4) ]
```

`vertex.getEdges( label, ...)`

Returns a list of in- or outbound edges of the vertex with given label(s).

`vertex.getProperty( name)`

Returns the property name a vertex.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)
```

```
arango> v.getProperty("name");
Hugo

vertex.getPropertyKeys()
Returns all propety names a vertex.
```

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.getPropertyKeys();
[ "name" ]

arango> v.setProperty("email", "hugo@hugo.de");
"hugo@hugo.de"

arango> v.getPropertyKeys();
[ "name", "email" ]
```

`vertex.properties()`

Returns all properties and their values of a vertex

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.properties();
{ name : "Hugo" }
```

`vertex.setProperty( name, value)`

Changes or sets the property name a vertex to value.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.getProperty("name");
"Hugo"

arango> v.setProperty("name", "Emil");
"Emil"

arango> v.getProperty("name");
"Emil"
```

`vertex.commonNeighborsWith( target_vertex, options)`

`vertex.commonPropertiesWith( target_vertex, options)`

`vertex.pathTo( target_vertex, options)`

```
vertex.distanceTo( target_vertex, options)

vertex.determinePredecessors( source, options)

vertex.pathesForTree( tree, path_to_here)

vertex.getNeighbors( options)

vertex.measurement( measurement)
```

Calculates the eccentricity, betweenness or closeness of the vertex

Edge Methods

`edge.getId()`

Returns the identifier of the edge.

Examples

```
arango> v = g.addVertex("v");
Vertex("v")

arango> e = g.addEdge(v, v, 1, "self");
Edge(1)

arango> e.getId();
1
```

`edge.getInVertex()`

Returns the vertex at the head of the edge.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self");
Edge(2)

arango> e.getInVertex();
Vertex(1)
```

`edge.getLabel()`

Returns the label of the edge.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "knows");
Edge(2)

arango> e.getLabel();
knows
```

`edge.getOutVertex()`

Returns the vertex at the tail of the edge.

Examples

```
arango> v = g.addVertex(1);
```

```
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self");
Edge(2)

arango> e.getOutVertex();
Vertex(1)
```

`edge.getPeerVertex( vertex)`

Returns the peer vertex of the edge and the vertex.

Examples

```
arango> v1 = g.addVertex("1");
Vertex("1")
arango> v2 = g.addVertex("2");
Vertex("2")
arango> e = g.addEdge(v1, v2, "1->2", "knows");
Edge("1->2")
arango> e.getPeerVertex(v1);
Vertex(2)
```

`edge.getProperty( name)`

Returns the property name an edge.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self", { "weight" : 10 });
Edge(2)

arango> e.getProperty("weight");
10
```

`edge.getPropertyKeys()`

Returns all propety names an edge.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self", { weight: 10 })
Edge(2)

arango> e.getPropertyKeys()
[ "weight" ]

arango> e.setProperty("name", "Hugo");
Hugo
```

```
arango> e.getPropertyKeys()
[ "weight", "name" ]
```

`edge.properties()`

Returns all properties and their values of an edge

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "knows");
Edge(2)

arango> e.properties();
{ "weight" : 10 }
```

`edge.setProperty( name, value)`

Changes or sets the property name an edges to value.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self", { weight: 10 })
Edge(2)

arango> e.getPropert("weight")
10

arango> e.setProperty("weight", 20);
20

arango> e.getPropert("weight")
20
```

Traversals

ArangoDB provides several ways to query graph data. Very simple operations can be composed with the low-level edge methods *edges*, *inEdges*, and *outEdges* for edge collections. For more complex operations, ArangoDB provides predefined traversal objects.

For any of the following examples, we'll be using the example collections *v* and *e*, populated with continents, countries and capitals data listed below (see Example Data).

Starting from Scratch

ArangoDB provides the *edges*, *inEdges*, and *outEdges* methods for edge collections. These methods can be used to quickly determine if a vertex is connected to other vertices, and which. This functionality can be exploited to write very simple graph queries in JavaScript.

For example, to determine which edges are linked to the *world* vertex, we can use *inEdges*:

```
db.e.inEdges('v/world').forEach(function(edge) {
  require("internal").print(edge._from, "->", edge.type, "->",
edge._to);
});
```

inEdges will give us all ingoing edges for the specified vertex *v/world*. The result is a JavaScript list, that we can iterate over and print the results:

```
v/continent-africa -> is-in -> v/world
v/continent-south-america -> is-in -> v/world
v/continent-asia -> is-in -> v/world
v/continent-australia -> is-in -> v/world
v/continent-europe -> is-in -> v/world
v/continent-north-america -> is-in -> v/world
```

Note: *edges*, *inEdges*, and *outEdges* return a list of edges. If we want to retrieve the linked vertices, we can use each edges' *_from* and *_to* attributes as follows:

```
db.e.inEdges('v/world').forEach(function(edge) {
  require("internal").print(db._document(edge._from).name, "->",
edge.type, "->", db._document(edge._to).name);
});
```

We are using the *document* method from the *db* object to retrieve the connected vertices now.

While this may be sufficient for one-level graph operations, writing a traversal by hand may become too complex for multi-level traversals.

Getting started

To use a traversal object, we first need to require the *traversal* module:

```
var traversal = require("org/arangodb/graph/traversal");
```

We then need to setup a configuration for the traversal and determine at which vertex to start the traversal:

```
var config = {
  datasource:
traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander,
  maxDepth: 1
};

var startVertex = db._document("v/world");
```

Note: The startVertex needs to be a document, not only a document id.

We can then create a traverser and start the traversal by calling its *traverse* method. Note that *traverse* needs a *result* object, which it can modify in place:

```
var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};
var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);
```

Finally, we can print the contents of the *results* object, limited to the visited vertices. We will only print the name and type of each visited vertex for brevity:

```
require("internal").print(result.visited.vertices.map(function(vertex
{
  return vertex.name + " (" + vertex.type + ")";
}));
```

The full script, which includes all steps carried out so far is thus:

```
var traversal = require("org/arangodb/graph/traversal");

var config = {
  datasource:
traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
```

```
    expander: traversal.inboundExpander,
    maxDepth: 1
};

var startVertex = db._document("v/world");
var result = {
    visited: {
        vertices: [ ],
        paths: [ ]
    }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);

require("internal").print(result.visited.vertices.map(function(vertex
{
    return vertex.name + " (" + vertex.type + ")";
}));
```

The result is a list of vertices that were visited during the traversal, starting at the start vertex (i.e. *v/world* in our example):

```
[
    "World (root)",
    "Africa (continent)",
    "Asia (continent)",
    "Australia (continent)",
    "Europe (continent)",
    "North America (continent)",
    "South America (continent)"
]
```

Note: The result is limited to vertices directly connected to the start vertex. We achieved this by setting the *maxDepth* attribute to *1*. Not setting it would return the full list of vertices.

Traversal Direction

For the examples contained in this manual, we'll be starting the traversals at vertex *v/world*. Vertices in our graph are connected like this:

```
v/world <- is-in <- continent (Africa) <- is-in <- country
(Algeria) <- is-in <- capital (Algiers)
```

To get any meaningful results, we must traverse the graph in inbound order. This means, we'll be following all incoming edges of to a vertex. In the traversal configuration, we have specified this via the *expander* attribute:

```
var config = {
    ...
    expander: traversal.inboundExpander
};
```

For other graphs, we might want to traverse via the outgoing edges. For this, we can use the *outboundExpander*. There is also an *anyExpander*, which will follow both outgoing and incoming edges.

This should be used with care and the traversal should always be limited to a maximum number of iterations (e.g. using the *maxIterations* attribute) in order to terminate at some point.

To invoke the default outbound expander for a graph, simply use the predefined function:

```
var config = {
    ...
    expander: traversal.outboundExpander
};
```

Please note the outbound expander will not produce any output for the examples if we still start the traversal at the *v/world* vertex.

Still, we can use the outbound expander if we start somewhere else in the graph, e.g.

```
var traversal = require("org/arangodb/graph/traversal");

var config = {
  datasource:
traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.outboundExpander
};

var startVertex = db._document("v/capital-algiers");
var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);

require("internal").print(result.visited.vertices.map(function(vertex)
{
  return vertex.name + " (" + vertex.type + ")";
}));
```

The result is:

```
[
  "Algiers (capital)",
  "Algeria (country)",
  "Africa (continent)",
  "World (root)"
]
```

which confirms that now we're going outbound.

Traversal Strategy

Depth-first traversals

The visitation order of vertices is determined by the *strategy*, *order* attributes set in the configuration. We chose *depthfirst* and *preorder*, meaning the traverser will emit each vertex before handling connected edges (pre-order), and descend into any connected edges before processing other vertices on the same level (depth-first).

Let's remove the *maxDepth* attribute now. We'll now be getting all vertices (directly and indirectly connected to the start vertex):

```
var config = {
  datasource:
traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander
};

var result = {
  visited: {
    vertices: [ ],
    paths: [ ]
  }
};

var traverser = new traversal.Traverser(config);
traverser.traverse(result, startVertex);

require("internal").print(result.visited.vertices.map(function(vertex
{
  return vertex.name + " (" + vertex.type + ")";
}));
```

The result will be a longer list, assembled in depth-first, pre-order order. For each continent found, the traverser will descend into linked countries, and then into the linked capital:

```
[
  "World (root)",
  "Africa (continent)",
  "Algeria (country)",
  "Algiers (capital)",
  "Angola (country)",
  "Luanda (capital)",
  "Botswana (country)",
  "Gaborone (capital)",
  "Burkina Faso (country)",
  "Ouagadougou (capital)",
  ...
]
```

Let's switch the *order* attribute from *preorder* to *postorder*. This will make the traverser emit vertices after all connected vertices were visited (i.e. most distant vertices will be emitted first):

```
[
  "Algiers (capital)",
  "Algeria (country)",
  "Luanda (capital)",
```

```
      "Angola (country)",
      "Gaborone (capital)",
      "Botswana (country)",
      "Ouagadougou (capital)",
      "Burkina Faso (country)",
      "Bujumbura (capital)",
      "Burundi (country)",
      "Yaounde (capital)",
      "Cameroon (country)",
      "N'Djamena (capital)",
      "Chad (country)",
      "Yamoussoukro (capital)",
      "Cote d'Ivoire (country)",
      "Cairo (capital)",
      "Egypt (country)",
      "Asmara (capital)",
      "Eritrea (country)",
      "Africa (continent)",
      ...
    ]
```

Breadth-first traversals

If we go back to *preorder*, but change the strategy to *breadth-first* and re-run the traversal, we'll see that
the return order changes, and items on the same level will be returned adjacently:

```
    [
      "World (root)",
      "Africa (continent)",
      "Asia (continent)",
      "Australia (continent)",
      "Europe (continent)",
      "North America (continent)",
      "South America (continent)",
      "Burkina Faso (country)",
      "Burundi (country)",
      "Cameroon (country)",
      "Chad (country)",
      "Algeria (country)",
      "Angola (country)",
      ...
    ]
```

Note: The order of items returned for the same level is undefined. This is because there is no natural order
of edges for a vertex with multiple connected edges. To explicitly set the order for edges on the same
level, you can specify an edge comparator function with the *sort* attribute:

```
    var config = {
      ...
      sort: function (l, r) { return l._key < r._key ? 1 : -1; }
      ...
    };
```

The arguments l and r are edge documents. This will traverse edges of the same vertex in backward *_key*
order:

```
[
  "World (root)",
  "South America (continent)",
  "North America (continent)",
  "Europe (continent)",
  "Australia (continent)",
  "Asia (continent)",
  "Africa (continent)",
  "Ecuador (country)",
  "Colombia (country)",
  "Chile (country)",
  "Brazil (country)",
  "Bolivia (country)",
  "Argentina (country)",
  ...
]
```

Note that this attribute only works for the usual expanders *traversal.inboundExpander*, *traversal.outboundExpander, traversal.anyExpander* and their corresponding "WithLabels" variants. If you are using custom expanders (see @ref TraversalsCustomExpanders) you have to organize the sorting within the specified expander.

Writing Custom Visitors

So far we have used much of the traverser's default functions. The traverser is very configurable and many of the default functions can be overridden with custom functionality.

For example, we have been using the default visitor function (which is always used if the configuration does not contain the *visitor* attribute). The default visitor function is called for each vertex in a traversal, and will push it into the result. This is the reason why the *result* variable looked different after the traversal, and needed to be initialized before the traversal was started.

We can write our own visitor function if we want to. The general function signature for visitor function is as follows:

```
var config = {
  ...
  visitor: function (config, result, vertex, path) { ... }
};
```

Visitor functions are not expected to return any values. Instead, they can modify the *result* variable (e.g. by pushing the current vertex into it), or do anything else. For example, we can create a simple visitor function that only prints information about the current vertex as we traverse:

```
var config = {
  datasource:
traversal.generalGraphDatasourceFactory("world_graph"),
  strategy: "depthfirst",
  order: "preorder",
  filter: traversal.visitAllFilter,
  expander: traversal.inboundExpander,
  visitor: function (config, result, vertex, path) {
    require("internal").print("visiting vertex", vertex.name);
  }
};

var traverser = new traversal.Traverser(config);
```

```js
traverser.traverse(undefined, startVertex);
```js

<a name="filtering_vertices_and_edges"></a>
### Filtering Vertices and Edges

<a name="filtering_vertices"></a>
#### Filtering Vertices

So far we have returned all vertices that were visited during the
traversal. This is not
always required. If the result shall be restrict to just specific
vertices, we can use a
filter function for vertices. It can be defined by setting the
*filter* attribute of a
traversal configuration, e.g.:

```js
var config = {
   filter: function (config, vertex, path) {
     if (vertex.type !== 'capital') {
       return 'exclude';
     }
   }
}
```

The above filter function will exclude all vertices that do not have a *type* value of *capital*. The filter function will be called for each vertex found during the traversal. It will receive the traversal configuration, the current vertex, and the full path from the traversal start vertex to the current vertex. The path consists of a list of edges, and a list of vertices. We could also filter everything but capitals by checking the length of the path from the start vertex to the current vertex. Capitals will have a distance of 3 from the *v/world* start vertex (capital -> is-in -> country -> is-in -> continent -> is-in -> world):

```js
var config = {
   ...
   filter: function (config, vertex, path) {
     if (path.edges.length < 3) {
       return 'exclude';
     }
   }
}
```

Note: If a filter function returns nothing (or *undefined*), the current vertex will be included, and all connected edges will be followed. If a filter function returns *exclude* the current vertex will be excluded from the result, and all still all connected edges will be followed. If a filter function returns *prune*, the current vertex will be included, but no connected edges will be followed.

For example, the following filter function will not descend into connected edges of continents, limiting the depth of the traversal. Still, continent vertices will be included in the result:

```js
var config = {
   ...
   filter: function (config, vertex, path) {
     if (vertex.type === 'continent') {
       return 'prune';
     }
   }
}
```

```
}
```

It is also possible to combine *exclude* and *prune* by returning a list with both values:

```
return [ 'exclude', 'prune' ];
```

Filtering Edges

It is possible to exclude certain edges from the traversal. To filter on edges, a filter function can be defined via the *expandFilter* attribute. The *expandFilter* is a function which is called for each edge during a traversal.

It will receive the current edge (*edge* variable) and the vertex which the edge connects to (in the direction of the traversal). It also receives the current path from the start vertex up to the current vertex (excluding the current edge and the vertex the edge points to).

If the function returns *true*, the edge will be followed. If the function returns *false*, the edge will not be followed. Here is a very simple custom edge filter function implementation, which simply includes edges if the (edges) path length is less than 1, and will exclude any other edges. This will effectively terminate the traversal after the first level of edges:

```
var config = {
  ...
  expandFilter: function (config, vertex, edge, path) {
    return (path.edges.length < 1);
  }
};
```

Writing Custom Expanders

The edges connected to a vertex are determined by the expander. So far we have used a default expander (the default inbound expander to be precise). The default inbound expander simply enumerates all connected ingoing edges for a vertex, based on the edge collection specified in the traversal configuration.

There is also a default outbound expander, which will enumerate all connected outgoing edges. Finally, there is an any expander, which will follow both ingoing and outgoing edges.

If connected edges must be determined in some different fashion for whatever reason, a custom expander can be written and registered by setting the *expander* attribute of the configuration. The expander function signature is as follows:

```
var config = {
  ...
  expander: function (config, vertex, path) { ... }
}
```

It is the expander's responsibility to return all edges and vertices directly connected to the current vertex (which is passed via the *vertex* variable). The full path from the start vertex up to the current vertex is also supplied via the *path* variable. An expander is expected to return a list of objects, which need to have an *edge* and a *vertex* attribute each.

Note: If you want to rely on a particular order in which the edges are traversed, you have to sort the edges returned by your expander within the code of the expander. The functions to get outbound, inbound or any edges from a vertex do not guarantee any particular order!

A custom implementation of an inbound expander could look like this (this is a non-deterministic expander, which randomly decides whether or not to include connected edges):

```
var config = {
  ...
  expander: function (config, vertex, path) {
    var connections = [ ];
    var datasource = config.datasource;
    datasource.getInEdges(vertex._id).forEach(function (edge) {
      if (Math.random() >= 0.5) {
        connections.push({ edge: edge, vertex: (edge._from) });
      }
    });
    return connections;
  }
};
```

A custom expander can also be used as an edge filter because it has full control over which edges will be returned.

Following are two examples of custom expanders that pick edges based on attributes of the edges and the connected vertices.

Finding the connected edges / vertices based on an attribute *when* in the connected vertices. The goal is to follow the edge that leads to the vertex with the highest value in the *when* attribute:

```
var config = {
  ...
  expander: function (config, vertex, path) {
    var datasource = config.datasource;
    // determine all outgoing edges
    var outEdges = datasource.getOutEdges(vertex);

    if (outEdges.length === 0) {
      return [ ];
    }

    var data = [ ];
    outEdges.forEach(function (edge) {
      data.push({ edge: edge, vertex: datasource.getInVertex(edge)
});
    });

    // sort outgoing vertices according to "when" attribute value
    data.sort(function (l, r) {
      if (l.vertex.when === r.vertex.when) {
        return 0;
      }

      return (l.vertex.when < r.vertex.when ? 1 : -1);
    });

    // pick first vertex found (with highest "when" attribute
value)
    return [ data[0] ];
  }
  ...
};
```

Finding the connected edges / vertices based on an attribute *when* in the edge itself. The goal is to pick the one edge (out of potentially many) that has the highest *when* attribute value:

```
var config = {
  ...
  expander: function (config, vertex, path) {
    var datasource = config.datasource;
    // determine all outgoing edges
    var outEdges = datasource.getOutEdges(vertex);

    if (outEdges.length === 0) {
      return [ ]; // return an empty list
    }

    // sort all outgoing edges according to "when" attribute
    outEdges.sort(function (l, r) {
      if (l.when === r.when) {
        return 0;
      }
      return (l.when < r.when ? -1 : 1);
    });

    // return first edge (the one with highest "when" value)
    var edge = outEdges[0];
    try {
      var v = datasource.getInVertex(edge);
      return [ { edge: edge, vertex: v } ];
    }
    catch (e) { }

    return [ ];
  }
  ...
};
```

!SUBSECTION Configuration Overview

This section summarizes the configuration attributes for the traversal object. The configuration can consist of the following attributes:

- *visitor*: visitor function for vertices. The function signature is *function (config, result, vertex, path)*. This function is not expected to return a value, but may modify the *variable* as needed (e.g. by pushing vertex data into the result).
- *expander*: expander function that is responsible for returning edges and vertices directly connected to a vertex . The function signature is *function (config, vertex, path)*. The expander function is required to return a list of connection objects, consisting of an *edge* and *vertex* attribute each.
- *filter*: vertex filter function. The function signature is *function (config, vertex, path)*. It may return one of the following values:
  - *undefined*: vertex will be included in the result and connected edges will be traversed
  - *exclude*: vertex will not be included in the result and connected edges will be traversed
  - *prune*: vertex will be included in the result but connected edges will not be traversed
  - [*prune*, *exclude*]: vertex will not be included in the result and connected edges will not be returned
- *expandFilter*: filter function applied on each edge/vertex combination determined by the expander. The function signature is *function (config, vertex, edge, path)*. The function should return *true* if the edge/vertex combination should be processed, and *false* if it should be ignored.

- *sort*: a filter function to determine the order in which connected edges are processed. The function signature is *function (l, r)*. The function is required to return one of the following values:
    - *-1* if *l* should have a sort value less than *r*
    - *1* if *l* should have a higher sort value than *r*
    - *0* if *l* and *r* have the same sort value
- *strategy*: determines the visitation strategy. Possible values are *depthfirst* and *breadthfirst*.
- *order*: determines the visitation order. Possible values are *preorder* and *postorder*.
- *itemOrder*: determines the order in which connections returned by the expander will be processed. Possible values are *forward* and *backward*.
- *maxDepth*: if set to a value greater than *0*, this will limit the traversal to this maximum depth.
- *minDepth*: if set to a value greater than *0*, all vertices found on a level below the *minDepth* level will not be included in the result.
- *maxIterations*: the maximum number of iterations that the traversal is allowed to perform. It is sensible to set this number so unbounded traversals will terminate at some point.
- *uniqueness*: an object that defines how repeated visitations of vertices should be handled. The *uniqueness* object can have a sub-attribute *vertices*, and a sub-attribute *edges*. Each sub-attribute can have one of the following values:
    - *none*: no uniqueness constraints
    - *path*: element is excluded if it is already contained in the current path. This setting may be sensible for graphs that contain cycles (e.g. A -> B -> C -> A).
    - *global*: element is excluded if it was already found/visited at any point during the traversal.

Example Data

The above examples all use a vertex collection *e* and an edge collection *e*. The vertex collection *v* contains continents, countries, and capitals. The edge collection *e* contains connections between continents and countries, and between countries and capitals.

To set up the collections and populate them with initial data, the following script was used:

```
db._create("v");
db._createEdgeCollection("e");

// vertices: root node
db.v.save({ _key: "world", name: "World", type: "root" });

// vertices: continents
db.v.save({ _key: "continent-africa", name: "Africa", type:
"continent" });
db.v.save({ _key: "continent-asia", name: "Asia", type: "continent"
});
db.v.save({ _key: "continent-australia", name: "Australia", type:
"continent" });
db.v.save({ _key: "continent-europe", name: "Europe", type:
"continent" });
db.v.save({ _key: "continent-north-america", name: "North America",
type: "continent" });
db.v.save({ _key: "continent-south-america", name: "South America",
type: "continent" });

// vertices: countries
db.v.save({ _key: "country-afghanistan", name: "Afghanistan", type:
"country", code: "AFG" });
db.v.save({ _key: "country-albania", name: "Albania", type:
"country", code: "ALB" });
db.v.save({ _key: "country-algeria", name: "Algeria", type:
"country", code: "DZA" });
db.v.save({ _key: "country-andorra", name: "Andorra", type:
"country", code: "AND" });
db.v.save({ _key: "country-angola", name: "Angola", type:
"country", code: "AGO" });
db.v.save({ _key: "country-antigua-and-barbuda", name: "Antigua and
Barbuda", type: "country", code: "ATG" });
db.v.save({ _key: "country-argentina", name: "Argentina", type:
"country", code: "ARG" });
db.v.save({ _key: "country-australia", name: "Australia", type:
"country", code: "AUS" });
db.v.save({ _key: "country-austria", name: "Austria", type:
"country", code: "AUT" });
db.v.save({ _key: "country-bahamas", name: "Bahamas", type:
"country", code: "BHS" });
db.v.save({ _key: "country-bahrain", name: "Bahrain", type:
"country", code: "BHR" });
db.v.save({ _key: "country-bangladesh", name: "Bangladesh", type:
"country", code: "BGD" });
db.v.save({ _key: "country-barbados", name: "Barbados", type:
"country", code: "BRB" });
db.v.save({ _key: "country-belgium", name: "Belgium", type:
"country", code: "BEL" });
db.v.save({ _key: "country-bhutan", name: "Bhutan", type:
```

```
"country", code: "BTN" });
db.v.save({ _key: "country-bolivia", name: "Bolivia", type:
"country", code: "BOL" });
db.v.save({ _key: "country-bosnia-and-herzegovina", name: "Bosnia
and Herzegovina", type: "country", code: "BIH" });
db.v.save({ _key: "country-botswana", name: "Botswana", type:
"country", code: "BWA" });
db.v.save({ _key: "country-brazil", name: "Brazil", type:
"country", code: "BRA" });
db.v.save({ _key: "country-brunei", name: "Brunei", type:
"country", code: "BRN" });
db.v.save({ _key: "country-bulgaria", name: "Bulgaria", type:
"country", code: "BGR" });
db.v.save({ _key: "country-burkina-faso", name: "Burkina Faso",
type: "country", code: "BFA" });
db.v.save({ _key: "country-burundi", name: "Burundi", type:
"country", code: "BDI" });
db.v.save({ _key: "country-cambodia", name: "Cambodia", type:
"country", code: "KHM" });
db.v.save({ _key: "country-cameroon", name: "Cameroon", type:
"country", code: "CMR" });
db.v.save({ _key: "country-canada", name: "Canada", type:
"country", code: "CAN" });
db.v.save({ _key: "country-chad", name: "Chad", type: "country",
code: "TCD" });
db.v.save({ _key: "country-chile", name: "Chile", type: "country",
code: "CHL" });
db.v.save({ _key: "country-colombia", name: "Colombia", type:
"country", code: "COL" });
db.v.save({ _key: "country-cote-d-ivoire", name: "Cote d'Ivoire",
type: "country", code: "CIV" });
db.v.save({ _key: "country-croatia", name: "Croatia", type:
"country", code: "HRV" });
db.v.save({ _key: "country-czech-republic", name: "Czech Republic",
type: "country", code: "CZE" });
db.v.save({ _key: "country-denmark", name: "Denmark", type:
"country", code: "DNK" });
db.v.save({ _key: "country-ecuador", name: "Ecuador", type:
"country", code: "ECU" });
db.v.save({ _key: "country-egypt", name: "Egypt", type: "country",
code: "EGY" });
db.v.save({ _key: "country-eritrea", name: "Eritrea", type:
"country", code: "ERI" });
db.v.save({ _key: "country-finland", name: "Finland", type:
"country", code: "FIN" });
db.v.save({ _key: "country-france", name: "France", type:
"country", code: "FRA" });
db.v.save({ _key: "country-germany", name: "Germany", type:
"country", code: "DEU" });
db.v.save({ _key: "country-people-s-republic-of-china", name:
"People's Republic of China", type: "country", code: "CHN" });

// vertices: capitals
db.v.save({ _key: "capital-algiers", name: "Algiers", type:
"capital" });
db.v.save({ _key: "capital-andorra-la-vella", name: "Andorra la
Vella", type: "capital" });
db.v.save({ _key: "capital-asmara", name: "Asmara", type: "capital"
});
db.v.save({ _key: "capital-bandar-seri-begawan", name: "Bandar Seri
```

```
Begawan", type: "capital" });
db.v.save({ _key: "capital-beijing", name: "Beijing", type:
"capital" });
db.v.save({ _key: "capital-berlin", name: "Berlin", type: "capital"
});
db.v.save({ _key: "capital-bogota", name: "Bogota", type: "capital"
});
db.v.save({ _key: "capital-brasilia", name: "Brasilia", type:
"capital" });
db.v.save({ _key: "capital-bridgetown", name: "Bridgetown", type:
"capital" });
db.v.save({ _key: "capital-brussels", name: "Brussels", type:
"capital" });
db.v.save({ _key: "capital-buenos-aires", name: "Buenos Aires",
type: "capital" });
db.v.save({ _key: "capital-bujumbura", name: "Bujumbura", type:
"capital" });
db.v.save({ _key: "capital-cairo", name: "Cairo", type: "capital"
});
db.v.save({ _key: "capital-canberra", name: "Canberra", type:
"capital" });
db.v.save({ _key: "capital-copenhagen", name: "Copenhagen", type:
"capital" });
db.v.save({ _key: "capital-dhaka", name: "Dhaka", type: "capital"
});
db.v.save({ _key: "capital-gaborone", name: "Gaborone", type:
"capital" });
db.v.save({ _key: "capital-helsinki", name: "Helsinki", type:
"capital" });
db.v.save({ _key: "capital-kabul", name: "Kabul", type: "capital"
});
db.v.save({ _key: "capital-la-paz", name: "La Paz", type: "capital"
});
db.v.save({ _key: "capital-luanda", name: "Luanda", type: "capital"
});
db.v.save({ _key: "capital-manama", name: "Manama", type: "capital"
});
db.v.save({ _key: "capital-nassau", name: "Nassau", type: "capital"
});
db.v.save({ _key: "capital-n-djamena", name: "N'Djamena", type:
"capital" });
db.v.save({ _key: "capital-ottawa", name: "Ottawa", type: "capital"
});
db.v.save({ _key: "capital-ouagadougou", name: "Ouagadougou", type:
"capital" });
db.v.save({ _key: "capital-paris", name: "Paris", type: "capital"
});
db.v.save({ _key: "capital-phnom-penh", name: "Phnom Penh", type:
"capital" });
db.v.save({ _key: "capital-prague", name: "Prague", type: "capital"
});
db.v.save({ _key: "capital-quito", name: "Quito", type: "capital"
});
db.v.save({ _key: "capital-saint-john-s", name: "Saint John's",
type: "capital" });
db.v.save({ _key: "capital-santiago", name: "Santiago", type:
"capital" });
db.v.save({ _key: "capital-sarajevo", name: "Sarajevo", type:
"capital" });
db.v.save({ _key: "capital-sofia", name: "Sofia", type: "capital"
```

```
});
db.v.save({ _key: "capital-thimphu", name: "Thimphu", type:
"capital" });
db.v.save({ _key: "capital-tirana", name: "Tirana", type: "capital"
});
db.v.save({ _key: "capital-vienna", name: "Vienna", type: "capital"
});
db.v.save({ _key: "capital-yamoussoukro", name: "Yamoussoukro",
type: "capital" });
db.v.save({ _key: "capital-yaounde", name: "Yaounde", type:
"capital" });
db.v.save({ _key: "capital-zagreb", name: "Zagreb", type: "capital"
});

// edges: continent -> world
db.e.save("v/continent-africa", "v/world", { type: "is-in" });
db.e.save("v/continent-asia", "v/world", { type: "is-in" });
db.e.save("v/continent-australia", "v/world", { type: "is-in" });
db.e.save("v/continent-europe", "v/world", { type: "is-in" });
db.e.save("v/continent-north-america", "v/world", { type: "is-in"
});
db.e.save("v/continent-south-america", "v/world", { type: "is-in"
});

// edges: country -> continent
db.e.save("v/country-afghanistan", "v/continent-asia", { type: "is-
in" });
db.e.save("v/country-albania", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-algeria", "v/continent-africa", { type: "is-
in" });
db.e.save("v/country-andorra", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-angola", "v/continent-africa", { type: "is-in"
});
db.e.save("v/country-antigua-and-barbuda", "v/continent-north-
america", { type: "is-in" });
db.e.save("v/country-argentina", "v/continent-south-america", {
type: "is-in" });
db.e.save("v/country-australia", "v/continent-australia", { type:
"is-in" });
db.e.save("v/country-austria", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-bahamas", "v/continent-north-america", { type:
"is-in" });
db.e.save("v/country-bahrain", "v/continent-asia", { type: "is-in"
});
db.e.save("v/country-bangladesh", "v/continent-asia", { type: "is-
in" });
db.e.save("v/country-barbados", "v/continent-north-america", {
type: "is-in" });
db.e.save("v/country-belgium", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-bhutan", "v/continent-asia", { type: "is-in"
});
db.e.save("v/country-bolivia", "v/continent-south-america", { type:
"is-in" });
db.e.save("v/country-bosnia-and-herzegovina", "v/continent-europe",
{ type: "is-in" });
db.e.save("v/country-botswana", "v/continent-africa", { type: "is-
```

```
in" });
db.e.save("v/country-brazil", "v/continent-south-america", { type:
"is-in" });
db.e.save("v/country-brunei", "v/continent-asia", { type: "is-in"
});
db.e.save("v/country-bulgaria", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-burkina-faso", "v/continent-africa", { type:
"is-in" });
db.e.save("v/country-burundi", "v/continent-africa", { type: "is-
in" });
db.e.save("v/country-cambodia", "v/continent-asia", { type: "is-in"
});
db.e.save("v/country-cameroon", "v/continent-africa", { type: "is-
in" });
db.e.save("v/country-canada", "v/continent-north-america", { type:
"is-in" });
db.e.save("v/country-chad", "v/continent-africa", { type: "is-in"
});
db.e.save("v/country-chile", "v/continent-south-america", { type:
"is-in" });
db.e.save("v/country-colombia", "v/continent-south-america", {
type: "is-in" });
db.e.save("v/country-cote-d-ivoire", "v/continent-africa", { type:
"is-in" });
db.e.save("v/country-croatia", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-czech-republic", "v/continent-europe", { type:
"is-in" });
db.e.save("v/country-denmark", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-ecuador", "v/continent-south-america", { type:
"is-in" });
db.e.save("v/country-egypt", "v/continent-africa", { type: "is-in"
});
db.e.save("v/country-eritrea", "v/continent-africa", { type: "is-
in" });
db.e.save("v/country-finland", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-france", "v/continent-europe", { type: "is-in"
});
db.e.save("v/country-germany", "v/continent-europe", { type: "is-
in" });
db.e.save("v/country-people-s-republic-of-china", "v/continent-
asia", { type: "is-in" });

// edges: capital -> country
db.e.save("v/capital-algiers", "v/country-algeria", { type: "is-in"
});
db.e.save("v/capital-andorra-la-vella", "v/country-andorra", {
type: "is-in" });
db.e.save("v/capital-asmara", "v/country-eritrea", { type: "is-in"
});
db.e.save("v/capital-bandar-seri-begawan", "v/country-brunei", {
type: "is-in" });
db.e.save("v/capital-beijing", "v/country-people-s-republic-of-
china", { type: "is-in" });
db.e.save("v/capital-berlin", "v/country-germany", { type: "is-in"
});
db.e.save("v/capital-bogota", "v/country-colombia", { type: "is-in"
```

```
});
db.e.save("v/capital-brasilia", "v/country-brazil", { type: "is-in"
});
db.e.save("v/capital-bridgetown", "v/country-barbados", { type:
"is-in" });
db.e.save("v/capital-brussels", "v/country-belgium", { type: "is-
in" });
db.e.save("v/capital-buenos-aires", "v/country-argentina", { type:
"is-in" });
db.e.save("v/capital-bujumbura", "v/country-burundi", { type: "is-
in" });
db.e.save("v/capital-cairo", "v/country-egypt", { type: "is-in" });
db.e.save("v/capital-canberra", "v/country-australia", { type: "is-
in" });
db.e.save("v/capital-copenhagen", "v/country-denmark", { type: "is-
in" });
db.e.save("v/capital-dhaka", "v/country-bangladesh", { type: "is-
in" });
db.e.save("v/capital-gaborone", "v/country-botswana", { type: "is-
in" });
db.e.save("v/capital-helsinki", "v/country-finland", { type: "is-
in" });
db.e.save("v/capital-kabul", "v/country-afghanistan", { type: "is-
in" });
db.e.save("v/capital-la-paz", "v/country-bolivia", { type: "is-in"
});
db.e.save("v/capital-luanda", "v/country-angola", { type: "is-in"
});
db.e.save("v/capital-manama", "v/country-bahrain", { type: "is-in"
});
db.e.save("v/capital-nassau", "v/country-bahamas", { type: "is-in"
});
db.e.save("v/capital-n-djamena", "v/country-chad", { type: "is-in"
});
db.e.save("v/capital-ottawa", "v/country-canada", { type: "is-in"
});
db.e.save("v/capital-ouagadougou", "v/country-burkina-faso", {
type: "is-in" });
db.e.save("v/capital-paris", "v/country-france", { type: "is-in"
});
db.e.save("v/capital-phnom-penh", "v/country-cambodia", { type:
"is-in" });
db.e.save("v/capital-prague", "v/country-czech-republic", { type:
"is-in" });
db.e.save("v/capital-quito", "v/country-ecuador", { type: "is-in"
});
db.e.save("v/capital-saint-john-s", "v/country-antigua-and-
barbuda", { type: "is-in" });
db.e.save("v/capital-santiago", "v/country-chile", { type: "is-in"
});
db.e.save("v/capital-sarajevo", "v/country-bosnia-and-herzegovina",
{ type: "is-in" });
db.e.save("v/capital-sofia", "v/country-bulgaria", { type: "is-in"
});
db.e.save("v/capital-thimphu", "v/country-bhutan", { type: "is-in"
});
db.e.save("v/capital-tirana", "v/country-albania", { type: "is-in"
});
db.e.save("v/capital-vienna", "v/country-austria", { type: "is-in"
});
```

```
db.e.save("v/capital-yamoussoukro", "v/country-cote-d-ivoire", {
type: "is-in" });
db.e.save("v/capital-yaounde", "v/country-cameroon", { type: "is-
in" });
db.e.save("v/capital-zagreb", "v/country-croatia", { type: "is-in"
});
```

Foxx

Build APIs and simple web applications in ArangoDB

Foxx is an easy way to create APIs and simple web applications from within ArangoDB. It is inspired by Sinatra, the classy Ruby web framework. If Foxx is Sinatra, [ArandoDB's Actions](#) are the corresponding *Rack*. They provide all the HTTP goodness.

If you just want to install an existing application, please use the [Foxx Manager](#). If you want to create your own application, please continue reading.

Overview

An application built with Foxx is written in JavaScript and deployed to ArangoDB directly. ArangoDB serves this application, you do not need a separate application server.

Think of an Foxx app as a typical web app similar to any other web app using other technologies. A Foxx app provides one or more URLs, which can either be accessed directly from the browser or from a backend application written e.g. in Ruby or C#. Other features include:

- *Routing:* Define arbitrary routes namespaced via the *Controllers*
- *Accesses data:* Direct access to all data in ArangoDB using simple queries, AQL, traversals and more
- *Manipulates data:* Create new or manipulate existing entries
- Deliver *static files* like HTML pages, CSS or images directly

The typical request to a Foxx application will work as follows (only conceptually, a lot of the steps are cached in reality):

1. The request is routed to a Foxx application depending on the mount point
2. The according controller of this application is determined (via something called the manifest file)
3. The request is then routed to a specific handler in this controller

The handler will now parse the request. This includes determining all parameters from the body (which is typically JSON encoded) to the path parameters of the URL. It is then up to you to handle this request and generate a response. In this process you will probably access the database. This is done via the *Repository*: This is an entity that is responsible for a collection and specifically:

1. Creating new entries in this collection
2. Modify or delete existing entries in this collection
3. Search for entries in this collection

To represent an entry in this collection it will use a *Model*, which is a wrapper around the raw data from the database. Here you can implement helper functions or simple access methods.

Your first Foxx app in 5 minutes

Let's build an application that sends a plain-text response "Hello YourName!" for all requests to */dev/my_app/hello/YourName*.

First, create a directory *apps* somewhere in your filesystem. This will be the Foxx application base directory for your database instance. Let's assume from now on that the absolute path for this directory is */home/user/apps*. When you have created the directory, create a sub-directory *databases* in it.

Foxx applications are database-specific, and the *databases* sub-directory is used to separate the Foxx applications of different databases running in the same ArangoDB instance.

Let's assume for now that you are working in the default database (*_system*), that is used when no database name is specified otherwise. To use Foxx applications with the *_system* database, create a sub-directory

_system inside the *databases* subdirectory. All Foxx applications for the *_system* database will go into this directory. Note: to add a Foxx application for a different databases than *_system*, use the database's name as the directory name instead of *_system*.

Finally, we can add a sub-directory *my_app* in the *_system* directory and should end up with the following directory layout (starting at */home/user* in our example):

```
apps/
   databases/
      _system/
         my_app/
```

In the *my_app* directory, create a file named *app.js* and save the following content in it:

```javascript
(function() {
    "use strict";

    var Foxx = require("org/arangodb/foxx"),
        controller = new Foxx.Controller(applicationContext)

    controller.get("/hello/:name", function(req, res) {
        res.set("Content-Type", "text/plain");
        res.body = "Hello " + req.params("name");
    });

}());
```

Beside the *app.js* we need a manifest file. In order to achieve that, we create a file called *manifest.json* in our *my_app* directory with the following content:

```json
{
   "name": "my_app",
   "version": "0.0.1",
   "author": "me and myself",
   "controllers": {
     "/": "app.js"
   }
}
```

You **must** specify a name and a version number for your application, otherwise it won't be loaded into ArangoDB.

You should now have the following files and directories with your application (starting at */home/user* in our example):

```
apps/
   databases/
      _system/
         my_app/
            manifest.json
            app.js
```

This is your application, and you're ready to use it.

Testing the application

Start ArangoDB as follows:

```
$ arangod --javascript.dev-app-path /home/user/apps /tmp/fancy_db
```

If you chose a different directory name, you need to replace */home/user/apps* with the actual directory name of course. Replace */tmp/fancy_db* with the directory your database data is located in.

The command will start the ArangoDB server in a **development mode** using the directory */home/user/apps* as the workspace and */tmp/fancy_db* as your database directory. In development mode the server automatically reloads all application files on every request, so changes to the underlying files are visible instantly. Note: if you use the development mode for the first time or choose a different directory for *dev-app-path*, it may be necessary to start ArangoDB with the *--upgrade* option once. This will initialize the specified application directory.

Note: the development mode is convenient when developing applications but the permanent reloading has an impact on performance. Therefore permanent reloading is only performed in development mode and not in production mode. Whenever you think your application is ready for production, you can install it using the Foxx manager and avoid the overhead of reloading.

Now point your browser to *localhost:8529/dev/my_app/hello/YourName* and you should see "Hello YourName".

Note: the above and all following examples assume that you are using the default database (*_system*).If you use a different database than *_system*, URLs must be changed to include the database name, too. For example, if your database name is *mydb*, the above URL changes to *localhost:8529/_db/mydb/dev/my_app/hello/YourName*. For more information on how to access specific databases, please refer to [Address of a Database](#).

After this short overview, let's get into the details. There are several example apps available on Github. You can install them via Foxx manager (covered in the chapter on Foxx manager) or simply clone them from [github](#).

Start with ["hello-foxx"](#) as it contains several basic usage examples. "aye-aye" and "fugu" are more advanced apps showing how to use Backbone, Underscore and Jquery together with Foxx. "foxx-authentication" shows how to register users, log in and check permissions.

Handling Requests

In development mode all available applications from the application directory */home/user/apps/databases//* are visible under *http:// localhost:8529/dev/* where is the name of the current database and is the directory name of your application.

In our example, was *my_app* and as we didn't specify a database, defaulted to *_system*.

When applications are installed in production mode, you can change the */dev* prefix to whatever you like, see Foxx Manager.

If you do not redefine it, all requests that go to the root of your application (i.e. */*) will be redirected to *index.html*.

This means that if your application does not provide a file *index.html*, calling the application root URL may result in a 404 error. In our example, the application root URL is *http://localhost:8529/dev/my_app/hello/*. Call it, and you should something like this in return:

```
{
  "error": true,
  "code": 404,
  "errorNum": 404,
  "errorMessage": "unknown path 'dev/my_app/index.html'"
}
```

To fix that, you can give your app a different default document, e.g. "hello/unknown". The adjusted manifest now looks like this:

```
{
  "name": "my_app",
  "version": "0.0.1",
  "author": "me and myself",
  "controllers": {
    "/": "app.js"
  },
  "defaultDocument": "hello/unknown"
}
```

Note: Browsers tend to cache results of redirections. To see the new default document in effect, first clear your browser's cache and point your browser to *http:// localhost:8529/dev/my_app/*.

Accessing collections from Foxx

Foxx assumes by default that an application has it's own collections. Accessing collections directly by name could cause problems, for instance if you had two completely independent Foxx applications that both access their own collection *'users'*.

To prevent such issues, Foxx provides functions that return an application-specific collection name. For example, *applicationContext.collectionName('users')* will return the collection name prefixed with the application name, e.g. "myapp_users". This allows to have a *users* collection which is specific for each application.

Additionally, a Foxx controller has a function "collection" that returns a reference to a collection prefixed like above, in the same way as *db.* would do. In the example, controller.collection('users') would return the collection object for the "myapp_users" collection, and you could use it like any other collection with the db object, e.g.

```
    controller.collection('users').toArray()
    controller.collection('users').save(...)
    controller.collection('users').remove(...)
    controller.collection('users').replace(...)
```

Of course you still use any collection directly with the db object even from Foxx. To access an collection called "movies" this could be one solution:

```
app.get("/all", function(req, res) {
    var db = require("org/arangodb").db;
    res.json({ movies: db.movies.toArray() });
});
```

Of course this completely bypasses prefixing and repositories, but works well especially for quick tests or shared collections that are NOT application-specific.

Then there are Foxx repositories. These are objects that you can create to hide the internals of the database access from the application so that the application will just use the repository but not the database.

A repository is an object that wrap access to a collection (or multiple collections if you want), whereas controller.collection returns the collection itself. That's the main difference.

To return a list of users from a controller using a repository, you could use it like this:

```
    var foxx = require("org/arangodb/foxx");
    var db = require("org/arangodb").db;
    var usersRepo = new foxx.Repository(db._collection("users"));
    app.get("/all", function(req, res) {
        res.json({ users: usersRepo.collection.toArray() });
    });
```

Of course you can create your own methods in the repository to add extra functionality.

Application Context

JavaScript modules within a Foxx application can access the application using the variable *applicationContext*. The applicationContext provides the following methods:

`applicationContext.collectionName(name)`

This returns the collection name with the application.

`applicationContext.foxxFilename(filename)`

This returns the path to a file within the Foxx directory.

`require(name)`

This will first look into the Foxx directory for a module named *name*. If no such module can be found, the global module paths are consulted.

Optional Functionality

FormatMiddleware

To use this plugin, please require it first:

```
  FormatMiddleware =
require("org/arangodb/foxx/template_middleware").FormatMiddleware;
```

This Middleware gives you Rails-like format handling via the *extension* of the URL or the accept header. Say you request an URL like */people.json*:

The *FormatMiddleware* will set the format of the request to JSON and then delete the *.json* from the request. You can therefore write handlers that do not take an *extension* into consideration and instead handle the format via a simple string. To determine the format of the request it checks the URL and then the *accept* header. If one of them gives a format or both give the same, the format is set. If the formats are not the same, an error is raised.

Use it by calling:

```
FormatMiddleware = require('foxx').FormatMiddleware;
app.before(FormatMiddleware.new(['json']));
```

In both forms you can give a default format as a second parameter, if no format could be determined. If you give no *defaultFormat* this case will be handled as an error.

TemplateMiddleware

To use this plugin, please require it first:

```
TemplateMiddleware =
require("org/arangodb/foxx/template_middleware").TemplateMiddleware;
```

The *TemplateMiddleware* can be used to give a Foxx.Controller the capability of using templates. Currently you can only use Underscore Templates. It expects documents in the following form in this collection:

```
{
  path: "high/way",
  content: "hello <%= username %>",
  contentType: "text/plain",
  templateLanguage: "underscore"
}
```

The *content* is the string that will be rendered by the template processor. The *contentType* is the type of content that results from this call. And with the *templateLanguage* you can choose your template processor. There is only one choice now: *underscore*. Which would set the body of the response to *hello Controller* with the template defined above. It will also set the *contentType* to *text/plain* in this case. In addition to the attributes you provided, you also have access to all your view helpers.

Initialize

Initialize with the name of a collection or a collection and optionally a set of helper functions. Then use *before* to attach the initialized middleware to your Foxx.Controller

Examples

```
templateMiddleware = new TemplateMiddleware("templates", {
uppercase: function (x) { return x.toUpperCase(); }
});
// or without helpers:
//templateMiddleware = new TemplateMiddleware("templates");


app.before(templateMiddleware);
```

Render

```
response.render(templatePath, data)
```

When the TemplateMiddleware is included, you will have access to the *render* function on the response object. If you call render, Controller will look into the this collection and search by the path attribute. It will then render the template with the given data.

Examples

```
response.render("high/way", {username: 'Application'})
```

The Manifest File

In the *manifest.json* you define the components of your application. The content is a JSON object with the following attributes (not all attributes are required though):

- *assets*: Deliver pre-processed files
- *author*: The author name
- *contributors*: An array containing objects, each represents a contributor (with *name* and optional *email*)
- *controllers*: Map routes to FoxxControllers
- *exports*: Map names to Foxx exports
- *defaultDocument*: The default document when the applicated root (/) is called (defaults to *index.html*)
- *description*: A short description of the application (Meta information)
- *engines*: Should be an object with *arangodb* set to the ArangoDB version your Foxx app is compatible with.
- *files*: Deliver files
- *isSystem*: Mark an application as a system application
- *keywords*: An array of keywords to help people find your Foxx app
- *lib*: Base path for all required modules
- *license*: Short form of the license (MIT, GPL...)
- *name*: Name of the application (Meta information)
- *repository*: An object with information about where you can find the repository: *type* and *url*
- *setup*: Path to a setup script
- *teardown*: Path to a teardown script
- *thumbnail*: Path to a thumbnail that represents the application (Meta information)
- *version*: Current version of the application (Meta information)

If you install an application using the Foxx manager or are using the development mode, your manifest will be checked for completeness and common errors. You should have a look at the server log files after changing a manifest file to get informed about potential errors in the manifest.

A more complete example for a Manifest file:

```
{
  "name": "my_website",
  "version": "1.2.1",
  "description": "My Website with a blog and a shop",
  "thumnail": "images/website-logo.png",

  "controllers": {
    "/blog": "apps/blog.js",
    "/shop": "apps/shop.js"
  },

  "lib": "lib",

  "files": {
    "/images": "images"
  },

  "assets": {
    "application.js": {
      "files": [
        "vendor/jquery.js",
        "assets/javascripts/*"
      ]
    }
```

```
  },

  "setup": "scripts/setup.js",
  "teardown": "scripts/teardown.js"
}
```

The setup and teardown scripts

You can provide a path to a JavaScript file that prepares ArangoDB for your application (or respectively removes it entirely). These scripts have access to *appCollection* and *appCollectionName*. Use the *setup* script to create all collections your application needs and fill them with initial data if you want to. Use the *teardown* script to remove all collections you have created.

Note: the setup script is called on each request in the development mode. If your application needs to set up specific collections, you should always check in the setup script whether they are already there.

The teardown script is called when an application is uninstalled. It is good practice to drop any collections in the teardown script that the application used exclusively, but this is not enforced. Maybe there are reasons to keep application data even after removing an application. It's up to you to decide what to do.

controllers is an object that matches routes to files

- The *key* is the route you want to mount at

- The *value* is the path to the JavaScript file containing the *FoxxController* you want to mount

You can add multiple controllers in one manifest this way.

The files

Deliver all files in a certain folder without modifying them. You can deliver text files as well as binaries:

```
"files": {
  "/images": "images"
}
```

The assets

The value for the asset key is an object consisting of paths that are matched to the files they are composed of. Let's take the following example:

```
"assets": {
  "application.js": {
    "files": [
      "vendor/jquery.js",
      "assets/javascripts/*"
    ]
  }
}
```

If a request is made to */application.js* (in development mode), the file array provided will be processed one element at a time. The elements are paths to files (with the option to use wildcards). The files will be concatenated and delivered as a single file.

The content-type (or mime type) of the HTTP response when requesting *application.js* is automatically determined by looking at the filename extension in the asset name (*application.js* in the above example).

If the asset does not have a filename extension, the content-type is determined by looking at the filename extension of the first file in the *files* list. If no file extension can be determined, the asset will be delivered with a content-type of *text/plain*.

It is possible to explicitly override the content-type for an asset by setting the optional *contentType* attribute of an asset as follows:

```
"assets": {
  "myincludes": {
    "files": [
      "vendor/jquery.js",
      "assets/javascripts/*"
    ],
    "contentType": "text/javascript"
  }
}
```

Details on FoxxController

Create

```
new FoxxController(applicationContext, options)
```

This creates a new Controller. The first argument is the controller context available in the variable *applicationContext*. The second one is an options array with the following attributes:

- *urlPrefix*: All routes you define within will be prefixed with it.

Examples

```
app = new Controller(applicationContext, {
urlPrefix: "/meadow"
});
```

HTTP Methods

Get

```
FoxxController::get(path, callback)
```

This handles requests from the HTTP verb *get*.

When defining a route you can also define a so called 'parameterized' *path* like */goose/:barn*. In this case you can later get the value the user provided for *barn* via the *params* function (see the Request object).

Examples

```
app.get('/goose/barn', function (req, res) {
// Take this request and deal with it!
});
```

Head

```
FoxxController::head(path, callback)
```

This handles requests from the HTTP verb *head*. You have to give a function as *callback*. It will get a request and response object as its arguments

Post

```
FoxxController::post(path, callback)
```

This handles requests from the HTTP verb *post*. See above for the arguments you can give.

Examples

```
app.post('/goose/barn', function (req, res) {
// Take this request and deal with it!
});
```

Put

```
FoxxController::put(path, callback)
```

This handles requests from the HTTP verb *put*. See above for the arguments you can give.

Examples

```
app.put('/goose/barn', function (req, res) {
// Take this request and deal with it!
});
```

Patch

```
FoxxController::patch(path, callback)
```

This handles requests from the HTTP verb *patch*. See above for the arguments you can give.

Examples

```
app.patch('/goose/barn', function (req, res) {
  // Take this request and deal with it!
});
```

Delete

`FoxxController::delete(path, callback)`

This handles requests from the HTTP verb *delete*. See above for the arguments you can give.

Warning: Do not forget that *delete* is a reserved word in JavaScript and therefore needs to be called as app['delete']. There is also an alias *del* for this very reason.

Examples

```
app['delete']('/goose/barn', function (req, res) {
  // Take this request and deal with it!
});
```

```
app.del('/goose/barn', function (req, res) {
  // Take this request and deal with it!
});
```

Documenting and constraining a specific route

If you now want to document your route, you can use JSDoc style comments (a multiline comment block with the first line starting with /** instead of /*) above your routes to do that:

```
/** Get all foxxes
 *
 * If you want to get all foxxes, please use this
 * method to do that.
 */
app.get("/foxxes", function () {
  // ...
});
```

The first line will be treated as a summary (For optical reasons in the produced documentation, the summary is restricted to 60 characters). All following lines will be treated as additional notes shown in the detailed view of the route documentation. With the provided information, Foxx will generate a nice documentation for you. Furthermore you can describe your API by chaining the following methods onto your path definition:

Path Param

If you defined a route "/foxx/:id", you can constrain which format a path parameter (*/foxx/12*) can have by giving it a type. We currently support the following types:

- int
- string

You can also provide a description of this parameter.

Examples

```
app.get("/foxx/:id", function {
// Do something
}).pathParam("id", {
description: "Id of the Foxx",
type: "int"
});
```

Query Param

`FoxxController::queryParam(id, options)`

Describe a query parameter:

If you defined a route "/foxx", you can constrain which format a query parameter (*/foxx?a=12*) can have by giving it a type. We currently support the following types:

- int
- string

You can also provide a description of this parameter, if it is required and if you can provide the parameter multiple times.

Examples

```
app.get("/foxx", function {
// Do something
}).queryParam("id", {
description: "Id of the Foxx",
type: "int",
required: true,
allowMultiple: false
});
```

Body Param

```
FoxxController::bodyParam(paramName, description, Model)
```

Expect the body of the request to be a JSON with the attributes you annotated in your model. It will appear alongside the provided description in your Documentation. This will initialize a *Model* with the data and provide it to you via the params as *paramName*. For information about how to annotate your models, see the Model section. If you provide the Model in an array, the response will take multiple models instead of one.

If you wrap the provided model in an array, the body param is always an array and accordingly the return value of the *params* for the body call will also return an array of models.

The behavior of *bodyParam* changes depending on the *rootElement* option set in the manifest. If it is set to true, it is expected that the body is an object with a key of the same name as the *paramName* argument. The value of this object is either a single object or in the case of a multi element an array of objects.

Error Response

```
FoxxController::errorResponse(errorClass, code, description)
```

Define a reaction to a thrown error for this route: If your handler throws an error of the defined errorClass, it will be caught and the response will have the given status code and a JSON with error set to your description as the body.

If you want more control over the returned JSON, you can give an optional fourth parameter in form of a function. It gets the error as an argument, the return value will transformed into JSON and then be used as the body. The status code will be used as described above. The description will be used for the documentation.

It also adds documentation for this error response to the generated documentation.

Examples

```
/* define our own error type, FoxxyError */
var FoxxyError = function (message) {
this.message = "the following FoxxyError occurred: ' + message;
};
FoxxyError.prototype = new Error();


app.get("/foxx", function {
/* throws a FoxxyError */
throw new FoxxyError();
}).errorResponse(FoxxyError, 303, "This went completely wrong.
Sorry!");


app.get("/foxx", function {
```

```
throw new FoxxyError("oops!");
}).errorResponse(FoxxyError, 303, "This went completely wrong.
Sorry!", function (e) {
return {
code: 123,
desc: e.message
};
});
```

onlyif

```
FoxxController::onlyIf(check)
```

Provide it with a function that throws an exception if the normal processing should not be executed.
Provide an `errorResponse` to define the behavior in this case. This can be used for authentication or
authorization for example.

Examples

```
app.get("/foxx", function {
// Do something
}).onlyIf(aFunction).errorResponse(ErrorClass, 303, "This went
completely wrong. Sorry!");
```

onlyIfAuthenticated

```
FoxxController::onlyIf(code, reason)
```

Please activate authentification for this app if you want to use this function. If the user is logged in, it will
do nothing. Otherwise it will respond with the status code and the reason you provided (the route handler
won't be called). This will also add the according documentation for this route.

Examples

```
app.get("/foxx", function {
// Do something
}).onlyIfAuthenticated(401, "You need to be authenticated");
```

Documenting and constraining all routes

In addition to documenting a specific route, you can also do the same for all routes of a controller. For this purpose use the *allRoutes* object of the according controller. The following methods are available.

Buffer Error Response

```
RequestContextBuffer::errorResponse(errorClass, code, description)
```

Defines an *errorResponse* for all routes of this controller. For details on *errorResponse* see the according method on routes.

Examples

```
app.allroutes.errorResponse(FoxxyError, 303, "This went completely
wrong. Sorry!");


app.get("/foxx", function {
// Do something
});
```

Buffer onlyIf

```
RequestContextBuffer::onlyIf(code, reason)
```

Defines an *onlyIf* for all routes of this controller. For details on *onlyIf* see the according method on routes.

Examples

```
app.allroutes.onlyIf(myPersonalCheck);


app.get("/foxx", function {
// Do something
});
```

Buffer onlyIfAuthenticated

```
RequestContextBuffer::errorResponse(errorClass, code, description)
```

Defines an *onlyIfAuthenticated* for all routes of this controller. For details on *onlyIfAuthenticated* see the according method on routes.

Examples

```
app.allroutes.onlyIfAuthenticated(401, "You need to be
authenticated");


app.get("/foxx", function {
// Do something
});
```

Before and After Hooks

You can use the following two functions to do something before or respectively after the normal routing process is happening. You could use that for logging or to manipulate the request or response (translate it to a certain format for example).

Before

`FoxxController::before(path, callback)`

The before function takes a *path* on which it should watch and a function that it should execute before the routing takes place. If you do omit the path, the function will be executed before each request, no matter the path. Your function gets a Request and a Response object.

Examples

```
app.before('/high/way', function(req, res) {
//Do some crazy request logging
});
```

After

`FoxxController::after(path, callback)`

This works pretty similar to the before function. But it acts after the execution of the handlers (Big surprise, I suppose).

Examples

```
app.after('/high/way', function(req, res) {
//Do some crazy response logging
});
```

The Request and Response Objects

When you have created your FoxxController you can now define routes on it. You provide each with a function that will handle the request. It gets two arguments (four, to be honest. But the other two are not relevant for now):

- The *request* object
- The *response* object

These objects are provided by the underlying ArangoDB actions and enhanced by the *BaseMiddleware* provided by Foxx.

The Request Object

The *request* object inherits several attributes from the underlying Actions:

- *compatibility*: an integer specifying the compatibility version sent by the client (in request header *x-arango-version*). If the client does not send this header, ArangoDB will set this to the minimum compatible version number. The value is 10000 *major + 100* minor (e.g. *10400* for ArangoDB version 1.4).

- *user*: the name of the current ArangoDB user. This will be populated only if authentication is turned on, and will be *null* otherwise.

- *database*: the name of the current database (e.g. *_system*)

- *protocol*: *http* or *https*

- *server*: a JSON object with sub-attributes *address* (containing server host name or IP address) and *port* (server port).

- *path*: request URI path, with potential database name stripped off.

- *url*: request URI path + query string, with potential database name stripped off

- *headers*: a JSON object with the request headers as key/value pairs

- *cookies*: a JSON object with the request cookies as key/value pairs

- *requestType*: the request method (e.g. "GET", "POST", "PUT", ...)

- *requestBody*: the complete body of the request as a string

- *parameters*: a JSON object with all parameters set in the URL as key/value pairs

- *urlParameters*: a JSON object with all named parameters defined for the route as key/value pairs.

In addition to these attributes, a Foxx request objects provides the following convenience methods:

Body

```
request.body()
```

Get the JSON parsed body of the request. If you need the raw version, please refer to the *rawBody* function.

rawBody

```
request.rawBody()
```

The raw request body, not parsed. Just a String.

Params

```
request.params(key)
```

Get the parameters of the request. This process is two-fold:

- If you have defined an URL like */test/:id* and the user requested */test/1*, the call *params("id")* will return *1*.
- If you have defined an URL like */test* and the user gives a query component, the query parameters will also be returned. So for example if the user requested */test?a=2*, the call *params("a")* will return *2*.

- # The Response Object

Every response object has the body attribute from the underlying Actions to set the raw body by hand.

You provide your response body as a string here.

Response Status

```
response.status(code)
```

Set the status *code* of your response, for example:

Examples

```
response.status(404);
```

Response Set

```
response.set(key, value)
```

Set a header attribute, for example:

Examples

```
response.set("Content-Length", 123);
response.set("Content-Type", "text/plain");
```

or alternatively:

```
response.set({
"Content-Length": "123",
"Content-Type": "text/plain"
});
```

Response Json

```
response.json(object)
```

Set the content type to JSON and the body to the JSON encoded *object* you provided.

Examples

```
response.json({'born': 'December 12, 1915'});
```

Controlling Access to Foxx Applications

Access to Foxx applications is controlled by the regular authentication mechanisms present in ArangoDB. The server can be run with or without HTTP authentication.

If authentication is turned on, then every access to the server is authenticated via HTTP authentication. This includes Foxx applications. The global authentication can be toggled via the configuration option @ref CommandLineArangoDisableAuthentication "server.disable-authentication".

If global HTTP authentication is turned on, requests to Foxx applications will require HTTP authentication too, and only valid users present in the _users system collection are allowed to use the applications.

Since ArangoDB 1.4, there is an extra option to restrict the authentication to just system API calls, such as /_api/... and /_admin/.... This option can be turned on using the @ref CommandLineArangoAuthenticateSystemOnly "server.authenticate-system-only" configuration option. If it is turned on, then only system API requests need authentication whereas all requests to Foxx applications and routes will not require authentication.

This is recommended if you want to disable HTTP authentication for Foxx applications but still want the general database APIs to be protected with HTTP authentication.

If you need more fine grained control over the access to your Foxx application, we built an authentication system you can use. Currently we only support cookie-based authentication, but we will add the possibility to use Auth Tokens and external OAuth providers in the near future. Of course you can roll your own authentication mechanism if you want to, and you can do it in an application-specific way if required.

To use the per-application authentication, you should first turn off the global HTTP authentication (or at least restrict it to system API calls as mentioned above). Otherwise clients will need HTTP authentication and need additional authentication by your Foxx application.

To have global HTTP authentication turned on for system APIs but turned off for Foxx, your server startup parameters should look like this:

```
--server.disable-authentication false --server.authenticate-system-
only true
```

Note: During development, you may even turn off HTTP authentication completely:

```
--server.disable-authentication true --server.authenticate-system-
only true
```

Please keep in mind that turning HTTP authentication off completely will allow unauthenticated access by anyone to all API functions, so do not use this is production.

Now it's time to configure the application-specific authentication. We built a small demo application to demonstrate how this works.

To use the application-specific authentication in your own app, first activate it in your controller.

Active Authentication

FoxxController::activateAuthentication(opts)

To activate authentication for this authentication, first call this function. Provide the following arguments:

- *type*: Currently we only support *cookie*, but this will change in the future

- *cookieLifetime*: An integer. Lifetime of cookies in seconds
- *cookieName*: A string used as the name of the cookie
- *sessionLifetime*: An integer. Lifetime of sessions in seconds

Examples

```
app.activateAuthentication({
type: "cookie",
cookieLifetime: 360000,
cookieName: "my_cookie",
sessionLifetime: 400,
});
```

Login

```
FoxxController::login(path, opts)
```

Add a route for the login. You can provide further customizations via the the options:

- *usernameField* and *passwordField* can be used to adjust the expected attributes in the *post* request. They default to *username* and *password*.
- *onSuccess* is a function that you can define to do something if the login was successful. This includes sending a response to the user. This defaults to a function that returns a JSON with *user* set to the identifier of the user and
- *key* set to the session key.
- *onError* is a function that you can define to do something if the login did not work. This includes sending a response to the user. This defaults to a function that sets the response to 401 and returns a JSON with *error* set to "Username or Password was wrong".

Both *onSuccess* and *onError* should take request and result as arguments.

Examples

```
app.login('/login', {
onSuccess: function (req, res) {
res.json({"success": true});
}
});
```

Logut

```
FoxxController::logout(path, opts)
```

This works pretty similar to the logout function and adds a path to your app for the logout functionality. You can customize it with a custom *onSuccess* and *onError* function:

- *onSuccess* is a function that you can define to do something if the logout was successful. This includes sending a response to the user. This defaults to a function that returns a JSON with *message* set to "logged out".
- *onError* is a function that you can define to do something if the logout did not work. This includes sending a response to the user. This defaults to a function that sets the response to 401 and returns a JSON with *error* set to "No session was found".

Both *onSuccess* and *onError* should take request and result as arguments.

Examples

```
app.logout('/logout', {
onSuccess: function (req, res) {
res.json({"message": "Bye, Bye"});
}
});
```

Register

`FoxxController::register(path, opts)`

This works pretty similar to the logout function and adds a path to your app for the register functionality. You can customize it with a custom `*nSuccess* and *onError* function:

- *onSuccess* is a function that you can define to do something if the registration was successful. This includes sending a response to the user. This defaults to a function that returns a JSON with *user* set to the created user document.
- *onError* is a function that you can define to do something if the registration did not work. This includes sending a response to the user. This defaults to a function that sets the response to 401 and returns a JSON with *error* set to "Registration failed".

Both *onSuccess* and *onError* should take request and result as arguments.

You can also set the fields containing the username and password via *usernameField* (defaults to *username*) and *passwordField* (defaults to *password*). If you want to accept additional attributes for the user document, use the option *acceptedAttributes* and set it to an array containing strings with the names of the additional attributes you want to accept. All other attributes in the request will be ignored.

If you want default attributes for the accepted attributes or set additional fields (for example *admin*) use the option *defaultAttributes* which should be a hash mapping attribute names to default values.

Examples

```
app.register('/logout', {
acceptedAttributes: ['name'],
defaultAttributes: {
admin: false
}
});
```

Change Password

FoxxController::changePassword(route, opts)`

Add a route for the logged in user to change the password. You can provide further customizations via the the options:

- *passwordField* can be used to adjust the expected attribute in the *post* request. It defaults to *password*.
- *onSuccess* is a function that you can define to do something if the change was successful. This includes sending a response to the user. This defaults to a function that returns a JSON with *notice* set to "Changed password!".
- *onError* is a function that you can define to do something if the login did not work. This includes sending a response to the user. This defaults to a function that sets the response to 401 and returns a JSON with *error* set to "No session was found".

Both *onSuccess* and *onError* should take request and result as arguments.

Examples

```
app.changePassword('/changePassword', {
onSuccess: function (req, res) {
res.json({"success": true});
}
});
```

Restricting routes

To restrict routes, see the documentation for Documenting and Restraining the routes.

Details on FoxxModel

The model doesn't know anything about the database. It is just a representation of the data as an JavaScript object. You can add and overwrite the methods of the prototype in your model prototype via the object you give to extend. In your model file, export the model as *model*.

```
var Foxx = require("org/arangodb/foxx");

var TodoModel = Foxx.Model.extend({
});

exports.model = TodoModel;
```

A Foxx Model can be initialized with an object of attributes and their values.

There's also the possibility of annotation: The second hash you give to the extend method are properties of the prototype. You can define an attributes property there:

```
var Foxx = require("org/arangodb/foxx");

var PersonModel = Foxx.Model.extend({
  // Your instance properties
}, {
  // Your prototype properties
  attributes: {
    name: { type: "string", required: true },
    age: { type: "integer" },
    active: { type: "boolean", defaultValue: true }
});

exports.model = TodoModel;
```

This has two effects: On the one hand it provides documentation. If you annotated your model, you can use it in the *bodyParam* method for documentation. On the other hand it will influence the behavior of the constructor: If you provide an object to the constructor, it will only take those attributes that are listed in the attributes object. This is especially useful if you want to to initialize the Model from user input. On the other hand it will set the default value for all attributes that have not been set by hand. An example:

```
var person = new PersonModel({
  name: "Pete",
  admin: true
});

person.attributes // => { name: "Pete", active: true }
```

Extend

FoxxModel::extend(instanceProperties, classProperties)

Extend the Model prototype to add or overwrite methods. The first object contains the properties to be defined on the instance, the second object those to be defined on the prototype.

Initialize

```
new FoxxModel(data)
```

If you initialize a model, you can give it initial *data* as an object.

Examples

```
instance = new Model({
a: 1
});
```

Get

```
FoxxModel::get(name)
```

Get the value of an attribute

Examples

```
instance = new Model({
a: 1
});


instance.get("a");
```

Set

```
FoxxModel::set(name, value)
```

Set the value of an attribute or multiple attributes at once

Examples

```
instance = new Model({
a: 1
});


instance.set("a", 2);
instance.set({
b: 2
});
```

Has

```
FoxxModel::has(name)
```

Returns true if the attribute is set to a non-null or non-undefined value.

Examples

```
instance = new Model({
a: 1
});
```

```
instance.has("a"); //=> true
instance.has("b"); //=> false
```

Attributes

The attributes property is the internal hash containing the model's state.

forDB

```
FoxxModel::forDB()
```

Return a copy of the model which can be saved into ArangoDB

forClient

```
FoxxModel::forClient()
```

Return a copy of the model which you can send to the client.

Details on FoxxRepository

A repository is a gateway to the database. It gets data from the database, updates it or saves new data. It uses the given model when it returns a model and expects instances of the model for methods like save. In your repository file, export the repository as *repository*.

```
Foxx = require("org/arangodb/foxx");

TodosRepository = Foxx.Repository.extend({
});

exports.repository = TodosRepository;
```

Initialize

```
new FoxxRepository(collection, opts)
```

Create a new instance of Repository.

A Foxx Repository is always initialized with a collection object. You can get your collection object by asking your Foxx.Controller for it: the *collection* method takes the name of the collection (and will prepend the prefix of your application). It also takes two optional arguments:

1. Model: The prototype of a model. If you do not provide it, it will default to Foxx.Model
2. Prefix: You can provide the prefix of the application if you need it in your Repository (for some AQL queries probably)

Examples

```
instance = new Repository(appContext.collection("my_collection"));
// or:
instance = new Repository(appContext.collection("my_collection"), {
model: MyModelPrototype,
prefix: app.collectionPrefix,
});
```

Attributes of a Repository

Collection

The wrapped ArangoDB collection object.

ModelPrototype

The prototype of the according model.

Prefix

The prefix of the application. This is useful if you want to construct AQL queries for example.

Methods of a Repository

Adding entries to the repository

`save(model)`

Saves a model into the database. Expects a model. Will set the ID and Rev on the model. Returns the model.

Examples

```
repository.save(my_model);
```

Finding entries in the repository

`byId(id)`

Returns the model for the given ID.

Examples

```
var myModel = repository.byId('test/12411');
myModel.get('name');
```

`byExample(example)`

Returns an array of models for the given ID.

Examples

```
var myModel = repository.byExample({ amazing: true });
myModel[0].get('name');
```

`firstExample(example)`

Returns the first model that matches the given example.

Examples

```
var myModel = repository.firstExample({ amazing: true });
myModel.get('name');
```

`all()`

Returns an array of models that matches the given example. You need to provide both a skip and a limit value.

Warning: ArangoDB doesn't guarantee a specific order in this case, to make this really useful we have to explicitly provide something to order by.

Examples

```
var myModel = repository.all({ skip: 4, limit: 2 });
myModel[0].get('name');
```

Removing entries from the repository

`remove(model)`

Remove the model from the repository. Expects a model.

Examples

```
repository.remove(myModel);
```

`removeById(id)`

Remove the document with the given ID. Expects an ID of an existing document.

Examples

```
repository.removeById('test/12121');
```

`removeByExample(example)`

Find all documents that fit this example and remove them.

Examples

```
repository.removeByExample({ toBeDeleted: true });
```

Replacing entries in the repository

`replace(model)`

Find the model in the database by its _id_ and replace it with this version. Expects a model. Sets the Revision of the model. Returns the model.

Examples

```
myModel.set('name', 'Jan Steemann');
repository.replace(myModel);
```

`replaceById(id, model)`

Find the model in the database by the given ID and replace it with the given. model. Sets the ID and Revision of the model and also returns it.

Examples

```
repository.replaceById('test/123345', myNewModel);
```

`replaceByExample(example, model)`

Find the model in the database by the given example and replace it with the given. model. Sets the ID and Revision of the model and also returns it.

Examples

```
repository.replaceByExample({ replaceMe: true }, myNewModel);
```

Updating entries in the repository

```
updateById(id, object)
```

Find an item by ID and update it with the attributes in the provided object. Returns the updated model.

Examples

```
repository.updateById('test/12131', { newAttribute: 'awesome' });
```

```
updateByExample(example, object)
```

Find an item by example and update it with the attributes in the provided object. Returns the updated model.

Examples

```
repository.updateByExample({ findMe: true }, { newAttribute:
'awesome' });
```

Counting entries in the repository

```
count()
```

Returns the number of entries in this collection.

Examples

```
repository.count();
```

Deploying a Foxx application

When a Foxx application is ready to be used in production, it is time to leave the development mode and deploy the app in a production environment.

The first step is to copy the application's script directory to the target ArangoDB server. If your development and production environment are the same, there is nothing to do. If production runs on a different server, you should copy the development application directory to some temporary place on the production server.

When the application code is present on the production server, you can use the *fetch* and *mount* commands from the Foxx Manager to register the application in the production ArangoDB instance and make it available.

Here are the individual steps to carry out:

- development:

  - cd into the directory that application code is in. Then create a tar.gz file with the application code (replace *app* with the actual name):

    ```
    cd /path/to/development/apps/directory
    tar cvfz app.tar.gz app
    ```

  - copy the tar.gz file to the production server:

    ```
    scp app.tar.gz production:/tmp/
    ```

- production:

  - create a temporary directory, e.g. */tmp/apps* and extract the tar archive into this directory:

    ```
    mkdir /tmp/apps
    cd /tmp/apps
    tar xvfz /tmp/app.tar.gz
    ```

  - start the ArangoDB shell and run the following commands in it:

    ```
    fm.fetch("directory", "/tmp/apps/app");
    fm.mount("app", "/app");
    ```

More information on how to deploy applications from different sources can be found in the Foxx Manager.

Developing an Application

While developing a Foxx application, it is recommended to use the development mode. The development mode makes ArangoDB reload all components of all Foxx applications on every request. While this has a negative impact on performance, it allows to view and debug changes in the application instantly. It is not recommended to use the development mode in production.

During development it is often necessary to log some debug output. ArangoDB provides a few mechanisms for this:

- using *console.log*: ArangoDB provides the *console* module, which you can use from within your Foxx application like this:

```
require("console").log(value);
```

  The *console* module will log all output to ArangoDB's logfile. If you are not redirecting to log output to stdout, it is recommended that you follow ArangoDB's logfile using a *tail -f* command or something similar. Please refer to JSModuleConsole for details.

- using *internal.print*: The *print* method of the *internal* module writes data to the standard output of the ArangoDB server process. If you have start ArangoDB manually and do not run it as an (unattended) daemon, this is a convenient method:

```
require("internal").print(value);
```

- tapping requests / responses: Foxx allows to tap incoming requests and outgoing responses using the *before* and *after* hooks. To print all incoming requests to the stdout of the ArangoDB server process, you could use some code like this in your controller:

```
controller.before("/*", function (req, res) {
  require("internal").print(req);
});
```

  Of course you can also use *console.log* or any other means of logging output.

Development Mode

If you start ArangoDB with the option *--javascript.dev-app-path* followed by the path to an app directory (see below) and then the path to the database, you are starting ArangoDB in development mode with the application loaded.

This means that on every request:

1. All routes are dropped
2. All module caches are flushed
3. Your manifest file is read
4. All files in your lib folder are loaded
5. An app in DIRNAME is mounted at */dev/DIRNAME*
6. The request will be processed

This means that you do not have to restart ArangoDB if you change anything in your app. It is of course not meant for production, because the reloading makes the app relatively slow.

The app directory has to be structured as follows:

```
└── databases
    ├── _system
    │   ├── foxx_app_1
    │   ├── foxx_app_2
    │   └── foxx_app_3
    └── my_db
        └── foxx_app_4
```

In this case you would have four foxx apps booted, three in the *_system* database and one in the *my_db* database.

Production Mode

To run a Foxx app in production first copy your app code to the directory given in the config variable --*javascript.app-path*. After that use Foxx manager to mount the app. You can also use Foxx manager to find out your current app-path.

Foxx Dependency Injection

If you have runtime dependencies you want to access in your route handlers but don't want to define at load time (e.g. dependencies between multiple Foxx apps), *FoxxController* allows you to inject these dependencies into your route handlers by adding injectors to it.

Add an injector

Registers a dependency factory with the controller.

```
controller.addInjector(name, factory)
```

The injected dependency will be available as a property with the chosen *name* on the third argument passed to each route handler.

If *factory* is a function, it will be called the first time a route of that controller is handled and its result will be injected into each route handler. Otherwise the value will be injected as it is.

If you want to inject a function as a dependency, you need to wrap it in a function.

Parameter

- *name*: the name under which the dependency will be available in the route handler.
- *factory*: a function returning the dependency or an arbitrary value that will be passed as-is.

Examples

```
function myFunc() {
    return 'Hello';
}
controller.addInjector('something', function() {return 2;});
controller.addInjector('other', 'just a string');
controller.addInjector('fn', function() {return myFunc;});

controller.get('/some/route', function(request, response, injected)
{
    response.json({
        something: injected.something, // 2
        other: injected.other, // 'just a string'
        fn: injected.fn.name // 'myFunc'
    });
});
```

Add multiple injectors

Registers multiple dependency factories with the controller.

```
controller.addInjector(object)
```

Equivalent to calling *addInjector(name, value)* for each property of the object.

Parameter

- *object*: an object mapping dependency names to dependency factories.

Examples

```javascript
function myFunc() {
    return 'Hello';
}
controller.addInjector({
    something: function() {return 2;},
    other: 'just a string',
    fn: function() {return myFunc;}
});

controller.get('/some/route', function(request, response, injected)
{
    response.json({
        something: injected.something, // 2
        other: injected.other, // 'just a string'
        fn: injected.fn.name // 'myFunc'
    });
});
```

Working with Foxx exports

Instead of (or in addition to) defining controllers, Foxx apps can also define exports.

Foxx exports are not intended to replace regular npm modules. They simply allow you to make your app's collections and *applicationContext* available in other Foxx apps or bundling ArangoDB-specific modules in re-usable Foxx apps.

Define an export module

In order to export modules in a Foxx app, you need to list the files in your manifest:

```
{
    "name": "foxx_exports_example",
    "version": "1.0.0",
    "description": "Demonstrates Foxx exports.",
    "exports": {
        "doodads": "./doodads.js",
        "anotherModule": "./someOtherFilename.js"
    },
    "controllers": {
        "/etc": "./controllers.js"
    }
}
```

The file *doodads.js* in the app's base path could look like this:

```
var Foxx = require('org/arangodb/foxx');
var Doodad = Foxx.Model.extend({}, {});
var doodadRepo = new Foxx.Repository(
    applicationContext.collection('doodads'),
    {model: Doodad}
);
exports.repo = doodadRepo;
exports.model = Doodad;
```

This module would then export the name "repo" bound to the variable *doodads* as well as the name "model" bound to the *Doodad* model.

Note: that the *applicationContext* is available to your Foxx exports just like in your Foxx controllers.

Warning

Foxx exports only support CommonJS exports using the special *exports* variable. Node-style exports via *module.exports* are not supported.

Import from another app

In order to import from another app, you need to know where the app is mounted.

Let's say we have mounted the example app above at */my-doodads*. We could now access the app's exports in another app like so:

```
var Foxx = require('org/arangodb/foxx');
var doodads = Foxx.requireApp('/my-doodads').doodads;
var Doodad = doodads.model;
var doodadRepo = doodads.repo;

// use the imported model and repository
var myDoodad = new Doodad();
doodadRepo.save(myDoodad);
```

Warning

When using Foxx exports in other apps, the load order of apps determines when which app's exports will become available.

In order to use Foxx exports in another app's controllers it is recommended you use *controller.addInjector* to delay the import until all mounted apps have been loaded:

```
var Foxx = require('org/arangodb/foxx');
var controller = new Foxx.Controller(applicationContext);
controller.addInjector({
    doodads: function() {
        return Foxx.requireApp('/my-doodads').doodads;
    }
});

// use the imported model and repository
controller.post('/doodads', function(request, response, injected) {
    var myDoodad = new injected.doodads.model();
    injected.doodads.repo.save(myDoodad);
    response.json(myDoodad.forClient());
});
```

There is currently no workaround to allow using one app's Foxx exports in another app's Foxx exports.

If you want to import Foxx exports of an app in controllers of the same app, you can do so without knowing the mount path in advance by using *applicationContext.mount*:

```
var Foxx = require('org/arangodb/foxx');
var doodads = Foxx.requireApp(applicationContext.mount).doodads;
```

If you don't need direct access to ArangoDB's functionality or the *applicationContext*, it is a better idea to use a regular npm module instead.

Foxx Manager

Foxx Applications

Foxx is an easy way to create APIs and simple web applications from within ArangoDB. It is inspired by Sinatra, the classy Ruby web framework. An application built with Foxx is written in JavaScript and deployed to ArangoDB directly. ArangoDB serves this application, you do not need a separate application server.

In order to share your applications with the community, we have created a central GitHub repository

```
https://github.com/arangodb/foxx-apps
```

where you can register your applications. This repository also contains the hello world application for Foxx.

Applications are managed using the Foxx manager *foxx-manager*. It is similar to tools like *brew* or *aptitude*.

First Steps with the Foxx Manager

The Foxx manager is a shell program. It should have been installed under */usr/bin* or */usr/local/bin* when installing the ArangoDB package. An instance of the ArangoDB server must be up and running.

```
unix> foxx-manager
Expecting a command, please try:

Example usage:
foxx-manager install <foxx> <mount-point>
foxx-manager uninstall <mount-point>

Further help:
foxx-manager help
```

The most important commands are

- *install*: Fetches a Foxx application from the central *foxx-apps* repository, mounts it to a local URL and sets it up
- *uninstall*: Unmounts a mounted Foxx application and calls its teardown method
- *list*: Lists all installed Foxx applications (alias: *installed*)
- *config*: Get information about the configuration including the path to the app directory.

When dealing with a fresh install of ArangoDB, there should be no installed applications besides the system applications that are shipped with ArangoDB.

```
unix> foxx-manager installed
Name          Author                Description
AppID                   Version     Mount                  Active    System
---------   ------------------   ---------------------------------
---------------------   ----------------   --------   -----------
------   -------   -------
aardvark    Michael Hackstein    Foxx application manager for the
ArangoDB web interface    app:aardvark:1.0    1.0
/_admin/aardvark    yes        yes
---------   ------------------   ---------------------------------
---------------------   ----------------   --------   -----------
------   -------   -------
1 application(s) found
```

There is currently one application installed. It is called "aardvark" and it is a system application. You can safely ignore system applications.

We are now going to install the *hello world* application. It is called "hello-foxx" - no suprise there.

```
unix> foxx-manager install hello-foxx /example
Application app:hello-foxx:1.2.2 installed successfully at mount
point /example
```

The second parameter */example* is the mount path of the application. You should now be able to access the example application under

```
http://localhost:8529/example
```

using your favorite browser. It will now also be visible when using the *installed* command.

```
unix> foxx-manager installed
Name            Author                   Description
AppID                         Version     Mount                        Active
System
----------      -----------------        -------------------------------
----------------------        --------------------     --------     -----
-----------     -------     -------
hello-foxx      Frank Celler             A simple example application.
app:hello-foxx:1.2.2    1.2.2            /example                yes          no
aardvark        Michael Hackstein        Foxx application manager for the
ArangoDB web interface       app:aardvark:1.0         1.0
/_admin/aardvark     yes          yes
----------      -----------------        -------------------------------
----------------------        --------------------     --------     -----
-----------     -------     -------
2 application(s) found
```

You can install the application again under different mount path.

```
unix> foxx-manager install hello-foxx /hello
Application app:hello-foxx:1.2.2 installed successfully at mount
point /hello
```

You now have two separate instances of the same application. They are completely independent of each other.

```
unix> foxx-manager installed
Name            Author                   Description
AppID                         Version     Mount                        Active
System
----------      -----------------        -------------------------------
----------------------        --------------------     --------     -----
-----------     -------     -------
hello-foxx      Frank Celler             A simple example application.
app:hello-foxx:1.2.2    1.2.2            /example                yes          no
aardvark        Michael Hackstein        Foxx application manager for the
ArangoDB web interface       app:aardvark:1.0         1.0
/_admin/aardvark     yes          yes
hello-foxx      Frank Celler             A simple example application.
app:hello-foxx:1.2.2    1.2.2            /hello                  yes          no
----------      -----------------        -------------------------------
----------------------        --------------------     --------     -----
-----------     -------     -------
3 application(s) found
```

The current version of the application is *1.2.2* (check the output of *installed* for the current version). It is even possible to mount a different version of an application.

Now let's remove the instance mounted under */hello*.

```
unix> foxx-manager uninstall /hello
Application app:hello-foxx:1.2.2 unmounted successfully from mount
point /hello
```

Note that "uninstall" is a combination of "teardown" and "unmount". This allows the application to clean up its own data. Internally, this will call the application's *teardown* script as defined in the application manifest.

Behind the Foxx Manager scenes

In the previous chapter we have seen how to install and uninstall applications. We now go into more details.

There are five steps when installing or uninstalling applications.

- *fetch* the application from a source
- *mount* the application at a mount path
- *setup* the application, creating the necessary collections
- *teardown* the application, removing the application-specific collections
- *unmount* the application

When installing an application, the steps "fetch", "mount", and "setup" are executed automatically. When uninstalling an application, the steps "teardown" and "unmount" are executed automatically.

Installing an application manually

We are now going to install the hello world application manually. You can use *search* to find application in your local copy of the central repository.

So, first we update our local copy to get the newest versions from the central repository.

```
unix> foxx-manager update
Updated local repository information with 4 application(s)
```

You can now search for words with the description of an application.

```
unix> foxx-manager search hello
Name            Author          Description
----------      ------------    ------------------------------------
----
hello-foxx      Frank Celler    This is 'Hello World' for ArangoDB
Foxx.
----------      ------------    ------------------------------------
----
1 application(s) found
```

As soon as you know the name of the application, you can check its details.

```
unix> foxx-manager info hello-foxx
Name:        hello-foxx
Author:      Frank Celler
System:      false
Description: This is 'Hello World' for ArangoDB Foxx.

Versions:
1.1.0: fetch github "fceller/hello-foxx" "v1.1.0"
1.1.1: fetch github "fceller/hello-foxx" "v1.1.1"
1.2.0: fetch github "fceller/hello-foxx" "v1.2.0"
1.2.1: fetch github "fceller/hello-foxx" "v1.2.1"
1.2.2: fetch github "fceller/hello-foxx" "v1.2.2"
```

If you execute

```
unix> foxx-manager fetch github "fceller/hello-foxx" "v1.2.1"
```

then the version 1.2.1 of the application will be downloaded. The command *fetched* lists all fetched applications.

```
unix> foxx-manager fetched
Name            Author          Description
AppID                      Version    Path
----------      ------------    ----------------------------    ----
----------------    --------    -----------------
hello-foxx                      A simple example application.
app:hello-foxx:1.2.1    1.2.1       hello-foxx-1.2.1
hello-foxx      Frank Celler    A simple example application.
app:hello-foxx:1.2.2    1.2.2       hello-foxx-1.2.2
```

```
----------   ------------   ---------------------------   ----
----------------,  --------   ----------------
2 application(s) found
```

We have now two versions of the hello world application. The current version fetched when installing the application using *install* and the one fetched now.

Let's now mount the application in version 1.2.1 under */hello*.

```
unix> foxx-manager mount app:hello-foxx:1.2.1 /hello
unix> foxx-manager installed
Name          Author                Description
AppID                     Version    Mount               Active
System
----------   ------------------   --------------------------------
-----------------------   --------------------   --------   -----
------------   -------   -------
hello-foxx    Frank Celler        A simple example application.
app:hello-foxx:1.2.1    1.2.1      /hello              yes        no
hello-foxx    Frank Celler        A simple example application.
app:hello-foxx:1.2.2    1.2.2      /example            yes        no
aardvark      Michael Hackstein    Foxx application manager for the
ArangoDB web interface    app:aardvark:1.0       1.0
/_admin/aardvark     yes         yes
----------   ------------------   --------------------------------
-----------------------   --------------------   --------   -----
------------   -------   -------
3 application(s) found
```

The application is mounted but not yet initialized. If you check the available collections, you will see that there is no collection called *hello_texts*.

```
arangosh> db._collections()
[
   [ArangoCollection 2965927, "_routing" (type document, status
loaded)],
   [ArangoCollection 96682407, "example_texts" (type document,
status loaded)],
   ...
]
```

A collection *example_texts* exists. This belongs to the mounted application at */example*. If we set-up the application, then the setup script will create the missing collection.

```
unix> foxx-manager setup /hello
```

Now check the list of collections again.

```
arangosh> db._collections()
[
   [ArangoCollection 2965927, "_routing" (type document, status
loaded)],
   [ArangoCollection 96682407, "example_texts" (type document,
status unloaded)],
```

```
   [ArangoCollection 172900775, "hello_texts" (type document, status
loaded)],
   ...
]
```

You can now use the mounted and initialized application.

```
unix> foxx-manager installed
Name            Author                  Description
AppID                     Version    Mount                   Active
System
----------    ------------------    --------------------------------
----------------------    --------------------    --------    -----
------------    -------    -------
hello-foxx    Frank Celler          A simple example application.
app:hello-foxx:1.2.2    1.2.2      /example                yes        no
hello-foxx    Frank Celler          A simple example application.
app:hello-foxx:1.2.1    1.2.1      /hello                  yes        no
aardvark      Michael Hackstein     Foxx application manager for the
ArangoDB web interface    app:aardvark:1.0        1.0
/_admin/aardvark    yes        yes
----------    ------------------    --------------------------------
----------------------    --------------------    --------    -----
------------    -------    -------
3 application(s) found
```

As you can see, there are two instances of the application under two mount paths in two different versions.
As the collections are not shared between applications, they are completely independent from each other.

Uninstalling an application manually

Now let us uninstall the application again. First we have to call the teardown script, which will remove the collection *hello_texts*.

```
unix> foxx-manager teardown /hello
```

This will drop the collection *hello_exists*. The application is, however, still reachable. We still need to unmount it.

```
unix> foxx-manager unmount /hello
```

Removing all mounts of an application

The same application might be mounted multiple times under different mount paths. To get rid of all mounted instances of an application, there is the "purge" command. "purge" will unmount and tear down all mounted instances of the application, and finally will remove the application directory, too.

This will remove all data of all instances of the application and also the application directory, code and configured. Use with care!

Making changes to an existing application

There are two options for deploying local changes to an existing application:

- the easiest way is to start the server in development mode. This will make all available foxx applications be available in under the *dev/* URL prefix. All changes to the application code will become live instantly because all applications are reloaded on each request to a URL starting with *dev/*. **Note**: that the constant reloading in the development mode has a performance impact so it shouldn't be used in product.

- if the development mode is not an option, you can use the *replace* command from foxx-manager. It provides an easy mechanism to re-deploy the code for an already installed application. It can be used as follows:

  ```
  unix> foxx-manager replace hello-foxx /hello
  ```

  The above will re-deploy the code for the application *hello-foxx* which has to be already installed under the *hello* mount point. The application's setup function will be called when invoking *replace* but not *teardown*.

Installing an application from your own Github repository

So far we have installed Foxx applications from the central Github repository "arangodb/foxx-apps". It is also possible to install an application from another repository. This can achieved by using the *fetch* and *mount* commands as follows:

```
unix> foxx-manager fetch github <username>/<repository>
unix> foxx-manager mount <app-id> <mount>
```

Examples

```
unix> foxx-manager fetch github arangodb/fugu
unix> foxx-manager mount fugu /fugu
```

Installing an application from a local directory

You may also install Foxx applications which are already located in the filesystem. Again, you can use the *fetch* command, but with the *directory* type. Note that the directory location must be a directory accessible by the foxx-manager.

Examples

```
unix> foxx-manager fetch directory /home/developer/apps/myapp
unix> foxx-manager mount myapp /myapp
```

Installing an application from a zip file

It is also possible to install an application contained in a zip file. You can use the *fetch* command again, with the *zip* type. Note that the zip file must be accessible by the foxx-manager.

Let's first fetch a zip file. We'll be downloading the fugu application from Github and store it in file *fugu.zip* locally:

```
unix> wget -O fugu.zip
"https://github.com/arangodb/fugu/archive/master.zip"
```

Now we can install the application from the zip file:

```
unix> foxx-manager fetch zip ./fugu.zip
unix> foxx-manager mount fugu /fugu
```

Using Multiple Databases

Regular Foxx applications are database-specific. When using multiple databases inside the same ArangoDB instance, there can be different Foxx applications in each database.

Every operation executed via the *foxx-manager* is run in the context of a single database. By default (i.e. if not specified otherwise), the *foxx-manager* will work in the context of the *_system* database.

If you want the *foxx-manager* to work in the context of a different database, use the command-line argument *--server.database* when invoking the *foxx-manager* binary.

Foxx Applications and Replication

Foxx applications consist of a file system part (scripts in the application directory) and a database part. The current version of ArangoDB cannot replicate changes in the file system so installing, updating or removing a Foxx application using *foxx-manager* will not be included in the replication.

Foxx Manager Commands

Use **help** to see all commands

```
unix> foxx-manager help
```

The following commands are available:

```
available           lists all Foxx applications available in the
local repository
config              returns configuration information from the
server
fetch               fetches a Foxx application from the central
foxx-apps repository into the local repository
fetched             lists all fetched Foxx applications that were
fetched into the local repository
help                shows this help
info                displays information about a Foxx application
install             fetches a Foxx application from the central
foxx-apps repository, mounts it to a local URL and sets it up
installed           alias for the 'list' command
list                lists all installed Foxx applications
mount               mounts a fetched Foxx application to a local URL
purge               uninstalls a Foxx application with all its
mounts and physically removes the application directory
                    WARNING: this will remove all data and code of
the application!

remove              alias for the 'purge' command
replace             replaces an existing Foxx application with the
current local version found in the application directory
rescan              rescans the Foxx application directory on the
server side
                    note: this is only required if the server-side
apps directory was modified by other processes

search              searches the local foxx-apps repository
setup               executes the setup script (app must already be
mounted)
teardown            executes the teardown script (app must be still
be mounted)
                    WARNING: this action will remove application
data if the application implements teardown!

uninstall           unmounts a mounted Foxx application and calls
its teardown method
unmount             unmounts a mounted Foxx application without
calling its teardown method
update              updates the local foxx-apps repository with data
from the central foxx-apps repository
```

Frequently Used Options

Internally, *foxx-manager* is a wrapper around *arangosh*. That means you can use the options of *arangosh*. To retrieve a list of the options for *arangosh*, try

```
unix> foxx-manager --help
```

To most relevant *arangosh* options to pass to the *foxx-manager* will be:

```
--server.database <string>              database name to use when
connecting
--server.disable-authentication <bool>    disable the password
prompt and authentication when connecting to the server
--server.endpoint <string>              endpoint to connect to,
use 'none' to start without a server
--server.password <string>              password to use when
connecting
--server.username <string>              username to use when
connecting
```

These options allow you to use the foxx-manager with a different database or with another than the default user.

ArangoDB's Actions

Introduction to User Actions

In some ways the communication layer of the ArangoDB server behaves like a Web server. Unlike a Web server, it normally responds to HTTP requests by delivering JSON objects. Remember, documents in the database are just JSON objects. So, most of the time the HTTP response will contain a JSON document from the database as body. You can extract the documents stored in the database using HTTP *GET*. You can store documents using HTTP *POST*.

However, there is something more. You can write small snippets - so called actions - to extend the database. The idea of actions is that sometimes it is better to store parts of the business logic within ArangoDB.

The simplest example is the age of a person. Assume you store information about people in your database. It is an anti-pattern to store the age, because it changes every now and then. Therefore, you normally store the birthday and let the client decide what to do with it. However, if you have many different clients, it might be easier to enrich the person document with the age using actions once on the server side.

Or, for instance, if you want to apply some statistics to large data-sets and you cannot easily express this as query. You can define a action instead of transferring the whole data to the client and do the computation on the client.

Actions are also useful if you want to restrict and filter data according to some complex permission system.

The ArangoDB server can deliver all kinds of information, JSON being only one possible format. You can also generate HTML or images. However, a Web server is normally better suited for the task as it also implements various caching strategies, language selection, compression and so on. Having said that, there are still situations where it might be suitable to use the ArangoDB to deliver HTML pages - static or dynamic. A simple example is the built-in administration interface. You can access it using any modern browser and there is no need for a separate Apache or IIS.

In general you will use [Foxx](#) to easily extend the database with business logic. Foxx provides an simple to use interface to actions.

The following sections will explain the low-level actions within ArangoDB on which Foxx is built and show how to define them. The examples start with delivering static HTML pages - even if this is not the primary use-case for actions. The later sections will then show you how to code some pieces of your business logic and return JSON objects.

The interface is loosely modeled after the JavaScript classes for HTTP request and responses found in node.js and the middleware/routing aspects of connect.js and express.js.

Note that unlike node.js, ArangoDB is multi-threaded and there is no easy way to share state between queries inside the JavaScript engine. If such state information is required, you need to use the database itself.

A Hello World Example

The client API or browser sends a HTTP request to the ArangoDB server and the server returns a HTTP response to the client. A HTTP request consists of a method, normally *GET* or *POST* when using a browser, and a request path like */hello/world*. For a real Web server there are a zillion of other thing to consider, we will ignore this for the moment. The HTTP response contains a content type, describing how to interpret the returned data, and the data itself.

In the following example, we want to define an action in ArangoDB, so that the server returns the HTML document

```
<html>
  <body>
   Hello World
  </body>
</html>
```

if asked *GET /hello/world*.

The server needs to know what function to call or what document to deliver if it receives a request. This is called routing. All the routing information of ArangoDB is stored in a collection *_routing*. Each entry in this collections describes how to deal with a particular request path.

For the above example, add the following document to the _routing collection:

```
arangosh> db._routing.save({
  url: {
    match: "/hello/world"
  },
  content: {
    contentType: "text/html",
    body: "<html><body>Hello World</body></html>"
  }
});
```

In order to activate the new routing, you must either restart the server or call the internal reload function.

```
arangosh> require("internal").reloadRouting()
```

Now use the browser and access http:// localhost:8529/hello/world

You should see the *Hello World* in our browser.

Matching a URL

There are a lot of options for the *url* attribute. If you define different routing for the same path, then the following simple rule is applied in order to determine which match wins: If there are two matches, then the more specific wins. I. e, if there is a wildcard match and an exact match, the exact match is preferred. If there is a short and a long match, the longer match wins.

Exact Match

If the definition is

```
{
  url: {
    match: "/hello/world"
  }
}
```

then the match must be exact. Only the request for */hello/world* will match, everything else, e. g. */hello/world/my* or */hello/world2*, will not match.

The following definition is a short-cut for an exact match.

```
{
  url: "/hello/world"
}
```

Note: While the two definitions will result in the same URL matching, there is a subtle difference between them:

The former definition (defining *url* as an object with a *match* attribute) will result in the URL being accessible via all supported HTTP methods (e.g. *GET*, *POST*, *PUT*, *DELETE*, ...), whereas the latter definition (providing a string *url* attribute) will result in the URL being accessible via HTTP *GET* and HTTP *HEAD* only, with all other HTTP methods being disabled. Calling a URL with an unsupported or disabled HTTP method will result in an HTTP 501 (not implemented) error.

Prefix Match

If the definition is

```
{
  url: {
    match: "/hello/world/*"
  }
}
```

then the match can be a prefix match. The requests for */hello/world*, */hello/world/my*, and */hello/world/how/are/you* will all match. However */hello/world2* does not match. Prefix matches within a URL part, i. e. */hello/world**, are not allowed. The wildcard must occur at the end, i. e.

```
/hello/*/world
```

is also disallowed.

If you define two routes

```
{ url: { match: "/hello/world/*" } }
{ url: { match: "/hello/world/emil" } }
```

then the second route will be used for *\/hello\/world\/emil* because it is more specific.

Parameterized Match

A parameterized match is similar to a prefix match, but the parameters are also allowed inside the URL path.

If the definition is

```
{
  url: {
    match: "/hello/:name/world"
  }
}
```

then the URL must have three parts, the first part being *hello* and the third part *world*. For example, *\/hello\/emil\/world* will match, while *\/hello\/emil\/meyer\/world* will not.

Constraint Match

A constraint match is similar to a parameterized match, but the parameters can carry constraints.

If the definition is

```
{
  url: {
    match: "/hello/:name/world",
    constraint: {
      name: "/[a-z]+/"
    }
  }
}
```

then the URL must have three parts, the first part being *hello* and the third part *world*. The second part must be all lowercase.

It is possible to use more then one constraint for the same URL part.

```
{
  url: {
    match: "/hello/:name|:id/world",
constraint: {
      name: "/[a-z]+/", id: "/[0-9]+/"
    }
  }
}
```

Optional Match

An optional match is similar to a parameterized match, but the last parameter is optional.
```

If the definition is

```
{
  url: {
    match: "/hello/:name?",
    constraint: {
      name: "/[a-z]+/"
    }
  }
}
```

then the URL *hello* and */hello/emil* will match.

If the definitions are

```
{ url: { match: "/hello/world" } }
{ url: { match: "/hello/:name", constraint: { name: "/[a-z]+/" } } }
}
{ url: { match: "/hello/*" } }
```

then the URL */hello/world* will be matched by the first route, because it is the most specific. The URL */hello/you* will be matched by the second route, because it is more specific than the prefix match.

Method Restriction

You can restrict the match to specific HTTP methods.

If the definition is

```
{
  url: {
    match: "/hello/world",
    methods: [ "post", "put" ]
  }
}
```

then only HTTP *POST* and *PUT* requests will match. Calling with a different HTTP method will result in an HTTP 501 error.

Please note that if *url* is defined as a simple string, then only the HTTP methods *GET* and *HEAD* will be allowed, an all other methods will be disabled:

```
{
  url: "/hello/world"
}
```

More on Matching

Remember that the more specific match wins.

- A match without parameter or wildcard is more specific than a match with parameters or wildcard.
- A match with parameter is more specific than a match with a wildcard.
- If there is more than one parameter, specificity is applied from left to right.

Consider the following definitions

```
(1) { url: { match: "/hello/world" } }
(2) { url: { match: "/hello/:name", constraint: { name: "/[a-z]+/"
} } }
(3) { url: { match: "/:something/world" } }
(4) { url: { match: "/hello/*" } }
```

Then

- */hello/world* is match by (1)
- */hello/emil* is match by (2)
- */your/world* is match by (3)
- */hello/you* is match by (4)

You can write the following document into the *_routing* collection to test the above examples.

```
{
  routes: [
{ url: { match: "/hello/world" }, content: "route 1" },
{ url: { match: "/hello/:name|:id", constraint: { name: "/[a-z]+/",
id: "/[0-9]+/" } }, content: "route 2" },
{ url: { match: "/:something/world" }, content: "route 3" },
{ url: { match: "/hello/*" }, content: "route 4" },
  ]
}
```

A Hello World Example for JSON

If you change the example slightly, then a JSON object will be delivered.

```
arangosh> db._routing.save({
  url: "/hello/json",
  content: {
    contentType: "application/json",
    body: "{ \"hello\" : \"world\" }"
  }
});
arangosh> require("internal").reloadRouting()
```

Again check with your browser http:// localhost:8529/hello/json

Depending on your browser and installed add-ons you will either see the JSON object or a download dialog. If your browser wants to open an external application to display the JSON object, you can change the *contentType* to *"text/plain"* for the example. This makes it easier to check the example using a browser. Or use *curl* to access the server.

```
bash> curl "http://127.0.0.1:8529/hello/json" && echo
{ "hello" : "world" }
```

Delivering Content

There are a lot of different ways on how to deliver content. We have already seen the simplest one, where static content is delivered. The fun, however, starts when delivering dynamic content.

Static Content

You can specify a body and a content-type.

```
{
  content: {
    contentType: "text/html",
    body: "<html><body>Hello World</body></html>"
  }
}
```

If the content type is *text/plain* then you can use the short-cut

```
{
  content: "Hello World"
}
```

A Simple Action

The simplest dynamic action is:

```
{
  action: {
    do: "org/arangodb/actions/echoRequest"
  }
}
```

It is not advisable to store functions directly in the routing table. It is better to call functions defined in modules. In the above example the function can be accessed from JavaScript as:

```
require("org/arangodb/actions").echoRequest
```

The function *echoRequest* is pre-defined. It takes the request objects and echos it in the response.

The signature of such a function must be

```
function (req, res, options, next)
```

Examples

```
arangosh> db._routing.save({
  url: "/hello/echo",
  action: {
    do: "org/arangodb/actions/echoRequest"
  }
});
```

Reload the routing and check http:// 127.0.0.1:8529/hello/echo

You should see something like

```
{
  "request": {
    "path": "/hello/echo",
    "headers": {
      "accept-encoding": "gzip, deflate",
      "accept-language": "de-de,de;q=0.8,en-us;q=0.5,en;q=0.3",
      "connection": "keep-alive",
      "content-length": "0",
      "host": "localhost:8529",
      "user-agent": "Mozilla/5.0 (X11; Linux x86_64; rv:15.0)
Gecko/20100101 Firefox/15.0"
    },
    "requestType": "GET",
    "parameters": { }
  },
  "options": { }
}
```

The request might contain *path*, *prefix*, *suffix*, and *urlParameters* attributes. *path* is the complete path as supplied by the user and always available. If a prefix was matched, then this prefix is stored in the attribute *prefix* and the remaining URL parts are stored as an array in *suffix*. If one or more parameters were matched, then the parameter values are stored in *urlParameters*.

For example, if the url description is

```
{
  url: {
    match: "/hello/:name/:action"
  }
}
```

and you request the path */hello/emil/jump*, then the request object will contain the following attribute

```
urlParameters: {
  name: "emil",
  action: "jump"
}
```

Action Controller

As an alternative to the simple action, you can use controllers. A controller is a module, defines the function *get, put, post, delete, head, patch*. If a request of the corresponding type is matched, the function will be called.

Examples

```
arangosh> db._routing.save({
  url: "/hello/echo",
  action: {
    controller: "org/arangodb/actions/echoController"
  }
```

```
});
```

Prefix Action Controller

The controller is selected when the definition is read. There is a more flexible, but slower and maybe insecure variant, the prefix controller.

Assume that the url is a prefix match

```
{
  url: {
    match: /hello/*"
  }
}
```

You can use

```
{
  action: {
    prefixController: "org/arangodb/actions"
  }
}
```

to define a prefix controller. If the URL */hello/echoController* is given, then the module *org/arangodb/actions/echoController* is used.

If you use a prefix controller, you should make certain that no unwanted actions are available under the prefix.

The definition

```
{
  action: "org/arangodb/actions"
}
```

is a short-cut for a prefix controller definition.

Function Action

You can also store a function directly in the routing table.

Examples

```
arangosh> db._routing.save({
  url: "/hello/echo",
  action: {
    callback: "function(req,res) {res.statusCode=200;
res.body='Hello'}"
  }
});
```

Requests and Responses

The controller must define handler functions which take a request object and fill the response object.

A very simple example is the function *echoRequest* defined in the module *org/arangodb/actions*.

```
function (req, res, options, next) {
   var result;

   result = { request: req, options: options };

   res.responseCode = exports.HTTP_OK;
   res.contentType = "application/json";
   res.body = JSON.stringify(result);
}
```

Install it via:

```
arangosh> db._routing.save({
   url: "/echo",
   action: {
      do: "org/arangodb/actions/echoRequest"
   }
});
```

Reload the routing and check http:// 127.0.0.1:8529/hello/echo

You should see something like

```
{
   "request": {
      "prefix": "/hello/echo",
      "suffix": [
         "hello",
         "echo"
      ],
      "path": "/hello/echo",
      "headers": {
         "accept-encoding": "gzip, deflate",
         "accept-language": "de-de,de;q=0.8,en-us;q=0.5,en;q=0.3",
         "connection": "keep-alive",
         "content-length": "0",
         "host": "localhost:8529",
         "user-agent": "Mozilla/5.0 (X11; Linux x86_64; rv:15.0)
Gecko/20100101 Firefox/15.0"
      },
      "requestType": "GET",
      "parameters": { }
   },
   "options": { }
}
```

You may also pass options to the called function:

```
arangosh> db._routing.save({
   url: "/echo",
   action: {
```

```
      do: "org/arangodb/actions/echoRequest",
      options: {
        "Hello": "World"
      }
    }
});
```

You should now see the options in the result.

```
{
  "request": {
    ...
  },
  "options": {
    "Hello": "World"
  }
}
`
```

Modifying Request and Response

As we've seen in the previous examples, actions get called with the request and response objects (named *req* and *res* in the examples) passed as parameters to their handler functions.

The *req* object contains the incoming HTTP request, which might or might not have been modified by a previous action (if actions were chained).

A handler can modify the request object in place if desired. This might be useful when writing middleware (see below) that is used to intercept incoming requests, modify them and pass them to the actual handlers.

While modifying the request object might not be that relevant for non-middleware actions, modifying the response object definitely is. Modifying the response object is an action's only way to return data to the caller of the action.

We've already seen how to set the HTTP status code, the content type, and the result body. The *res* object has the following properties for these:

- *contentType*: MIME type of the body as defined in the HTTP standard (e.g. *text/html*, *text/plain*, *application/json*, ...)
- *responsecode*: the HTTP status code of the response as defined in the HTTP standard. Common values for actions that succeed are *200* or *201*. Please refer to the HTTP standard for more information.
- *body*: the actual response data

To set or modify arbitrary headers of the response object, the *headers* property can be used. For example, to add a user-defined header to the response, the following code will do:

```
res.headers = res.headers || { }; // headers might or might not be
present
res.headers['X-Test'] = 'someValue'; // set header X-Test to
"someValue"
```

This will set the additional HTTP header *X-Test* to value *someValue*. Other headers can be set as well. Note that ArangoDB might change the case of the header names to lower case when assembling the overall response that is sent to the caller.

It is not necessary to explicitly set a *Content-Length* header for the response as ArangoDB will calculate the content length automatically and add this header itself. ArangoDB might also add a *Connection* header itself to handle HTTP keep-alive.

ArangoDB also supports automatic transformation of the body data to another format. Currently, the only supported transformations are base64-encoding and base64-decoding. Using the transformations, an action can create a base64 encoded body and still let ArangoDB send the non-encoded version, for example:

```
res.body = 'VGhpcyBpcyBhIHRlc3Q=';
res.transformations = res.transformations || [ ]; // initialise
res.transformations.push('base64decode'); // will base64 decode the
response body
```

When ArangoDB processes the response, it will base64-decode what's in *res.body* and set the HTTP header *Content-Encoding: binary*. The opposite can be achieved with the *base64encode* transformation: ArangoDB will then automatically base64-encode the body and set a *Content-Encoding: base64* HTTP header.

Writing dynamic action handlers

To write your own dynamic action handlers, you must put them into modules.

Modules are a means of organizing action handlers and making them loadable under specific names.

To start, we'll define a simple action handler in a module */own/test*:

```
arangosh> db._modules.save({
  path: "/own/test",
  content: "exports.do = function(req, res, options, next) {
res.body = 'test'; res.responseCode = 200; res.contentType =
'text/html'; };"
});
```

This does nothing but register a do action handler in a module */own/test*. The action handler is not yet callable, but must be mapped to a route first. To map the action to the route */ourtest*, execute the following command:

```
arangosh> db._routing.save({
  url: "/ourtest",
  action: {
    controller: "/own/test"
  }
});
```

In order to see the module in action, you must either restart the server or call the internal reload function.

```
arangosh> require("internal").reloadRouting()
```

Now use the browser and access http:// localhost:8529/ourtest

You will see that the module's do function has been executed.

A Word about Caching

Sometimes it might seem that your change do not take effect. In this case the culprit could be one of the caches. With dynamic actions there are two caches involved:

The Routing Cache

The routing cache stores the routing information computed from the *_routing* collection. Whenever you change this collection manually, you need to call

```
arangosh> require("internal").reloadRouting();
```

in order to rebuild the cache.

The Modules Cache

If you use a dynamic action and this action is stored in module, then the module functions are also stored in a cache in order to avoid parsing the JavaScript code again and again.

Whenever you change the *modules* collections manually, you need to call

```
arangosh> require("internal").flushServerModules();
```

in order to rebuild the cache.

Flush Order

If you define a dynamic routing and the controller, then you need to flush the caches in a particular order. In order to build the routes, the module information must be known. Therefore, you need to flush the modules caches first.

```
arangosh> require("internal").flushServerModules();
arangosh> require("internal").reloadRouting();
```

Advanced Usages

For detailed information see the reference manual.

Redirects

Use the following for a permanent redirect:

```
arangosh> db._routing.save({
  url: "/",
  action: {
    do: "org/arangodb/actions/redirectRequest",
    options: {
      permanently: true,
      destination: "http://somewhere.else/"
    }
  }
});
```

Routing Bundles

Instead of adding all routes for package separately, you can specify a bundle.

```
{
  routes: [
    { url: "/url1", content: "..." },
    { url: "/url2", content: "..." },
    { url: "/url3", content: "..." },
    ...
  ]
}
```

The advantage is, that you can put all your routes into one document and use a common prefix.

```
{
  urlPrefix: "/test",
  routes: [
    { url: "/url1", content: "..." },
    { url: "/url2", content: "..." },
    { url: "/url3", content: "..." },
    ...
  ]
}
```

will define the URL */test/url1*, */test/url2*, and */test/url3*.

Writing Middleware

Assume, you want to log every request. In this case you can easily define an action for the whole url-space */*. This action simply logs the requests, calls the next in line, and logs the response.

```
exports.logRequest = function (req, res, options, next) {
  console.log("received request: %s", JSON.stringify(req));
  next();
```

```
    console.log("produced response: %s", JSON.stringify(res));
};
```

This function is available as *org/arangodb/actions/logRequest*. You need to tell ArangoDB that it is should use a prefix match and that the shortest match should win in this case:

```
arangosh> db._routing.save({
  middleware: [
    {
      url: {
        match: "/*"
      },
      action: {
        do: "org/arangodb/actions/logRequest"
      }
    }
  ]
});
```

If you call *next()*, the next specific routing will be used for the original URL. Even if you modify the URL in the request object *req*, this will not cause the *next()* to jump to the routing defined for this next URL. If proceeds occurring the origin URL. However, if you use *next(true)*, the routing will stop and request handling is started with the new URL. You must ensure that *next(true)* is never called without modifying the URL in the request object *req*. Otherwise an endless loop will occur.

Application Deployment

Using single routes or [bundles](#) can be become a bit messy in large applications. Kaerus has written a [deployment tool](#) in node.js.

Note that there is also [Foxx](#) for building applications with ArangoDB.

Common Pitfalls when using Actions

Caching

If you made any changes to the routing but the changes do not have any effect when calling the modified action's URL, you might have been hit by some caching issues.

After any modification to the routing or actions, it is thus recommended to make the changes "live" by calling the following functions from within arangosh:

```
arangosh> require("internal").flushServerModules();
arangosh> require("internal").reloadRouting();
```

You might also be affected by client-side caching. Browsers tend to cache content and also redirection URLs. You might need to clear or disable the browser cache in some cases to see your changes in effect.

Data types

When processing the request data in an action, please be aware that the data type of all URL parameters is *string*. This is because the whole URL is a string and when the individual parts are extracted, they will also be strings.

For example, when calling the URL http:// localhost:8529/hello/world?value=5

the parameter *value* will have a value of (string) *5*, not (number) *5*. This might be troublesome if you use JavaScript's === operator when checking request parameter values.

The same problem occurs with incoming HTTP headers. When sending the following header from a client to ArangoDB

```
X-My-Value: 5
```

then the header *X-My-Value* will have a value of (string) *5* and not (number) *5*.

501 Not Implemented

If you defined a URL in the routing and the URL is accessible fine via HTTP *GET* but returns an HTTP 501 (not implemented) for other HTTP methods such as *POST*, *PUT* or *DELETE*, then you might have been hit by some defaults.

By default, URLs defined like this (simple string *url* attribute) are accessible via HTTP *GET* and *HEAD* only. To make such URLs accessible via other HTTP methods, extend the URL definition with the *methods* attribute.

For example, this definition only allows access via *GET* and *HEAD*:

```
{
  url: "/hello/world"
}
```

whereas this definition allows HTTP *GET*, *POST*, and *PUT*:

```
{
  url: {
```

```
    match: "/hello/world",
    methods: [ "get", "post", "put" ]
  }
}
```

The former definition (defining *url* as an object with a *match* attribute) will result in the URL being accessible via all supported HTTP methods (e.g. *GET*, *POST*, *PUT*, *DELETE*, ...), whereas the latter definition (providing a string *url* attribute) will result in the URL being accessible via HTTP *GET* and HTTP *HEAD* only, with all other HTTP methods being disabled. Calling a URL with an unsupported or disabled HTTP method will result in an HTTP 501 error.

Introduction to Replication

Starting with ArangoDB 1.4, ArangoDB comes with optional asynchronous master-slave replication. Replication is configured on a per-database level, meaning that different databases in the same ArangoDB instance can have different replication settings. Replication must be turned on explicitly before it becomes active for a database.

In a typical master-slave replication setup, clients direct *all* their write operations for a specific database to the master. The master database is the only place to connect to when making any insertions/updates/deletions.

The master database will log all write operations in its so-called *event log*. The event log can be considered as an ordered stream of changes for the database.

Any number of slaves can then connect to the master database and fetch data from the master database's event log. The slaves then can apply all the events from the log in the same order locally. After that, they will have the same state of data as the master database.

In this setup, write operations are applied first in the master database, and applied in the slave database(s) afterwards.

For example, let's assume a write operation is applied and logged in the master database at point in time t0. To make a slave database apply the same operation, it must first fetch the write operation's data from master database's event log, then parse it and apply it locally. This will happen at some point in time after t0, let's say t1.

The difference between t1 and t0 is called the *replication lag*, and it is unavoidable in asynchronous replication. The amount of replication lag depends on many factors, a few of which are:

- the network capacity between the slaves and the master
- the load of the master and the slaves
- the frequency in which slaves poll the master for updates

Between t0 and t1, the state of data on the master is newer than the state of data on the slave(s). At point in time t1, the state of data on the master and slave(s) is consistent again (provided no new data modifications happened on the master in between). Thus, the replication will lead to an *eventually consistent* state of data.

Transactions are honored in replication, i.e. transactional write operations will become visible on slaves atomically.

As all write operations will be logged to a master database's event log, the replication in ArangoDB 1.4 cannot be used for write-scaling. The main purposes of the replication in ArangoDB 1.4 are to provide read-scalability and "hot backups" for specific databases.

It is possible to connect multiple slave databases to the same master database. Slave databases should be used as read-only instances, and no user-initiated write operations should be carried out on them. Otherwise data conflicts may occur that cannot be solved automatically, and that will make the replication stop. Master-master (or multi-master) replication is not supported in ArangoDB 1.4.

The replication in ArangoDB 1.4 is asynchronous, meaning that slaves will *pull* changes from the master database. Slaves need to know to which master database they should connect to, but a master database is not aware of the slaves that replicate from it. When the network connection between the master database and a slave goes down, write operations on the master can continue normally. When the network is up again, slaves can reconnect to the master database and transfer the remaining changes. This will happen automatically provided slaves are configured appropriately.

The replication is turned off by default. In order to create a master-slave setup, the replication features need to be enabled on both the master database and the slave databases.

Replication is configured on a per-database level. If multiple database are to be replicated, the replication must be set up individually per database.

Components

The replication architecture in ArangoDB 1.4 consists of two main components, which can be used together or in isolation: the *replication logger* and the *replication applier*.

Both are available in ArangoDB 1.4 and can be administered via the command line or a REST API (see [HTTP Interface for Replication](#)).

In most cases, the *replication logger* will be run inside a master database, and the *replication applier* will be executed in a slave database.

As there can be multiple database inside an ArangoDB instance, there can be multiple replication loggers and multiple replication appliers per ArangoDB instance.

Replication Logger

Purpose

The purpose of the replication logger is to log all changes that modify the state of data in a specific database. This includes document insertions, updates, and deletions. It also includes creating, dropping, renaming and changing collections and indexes.

When the replication logger is used, it will log all these write operations for the database in its *event log*, in the same order in which the operations were carried out originally.

Reading the event log sequentially provides a list of all write operations carried out in the database. Replication clients can request certain log events from the logger.

For example, ArangoDB's replication applier will permanently query the latest events from the replication logger. The applier of the slave database will then apply all changes locally to get to the same state of data as the master database. It will keep track of which parts of the event log it already synchronized, meaning that it will perform an incremental synchronization.

Technically, the event log is a system collection named *_replication* inside the database the applier runs in. The event log is persisted and will still be present after a server shutdown or crash. The event log's underlying collection should not be modified by users directly. It should only be accessed using the special API methods offered by ArangoDB.

Starting and Stopping

ArangoDB will only log changes if the replication logger is turned on for the specific database. Should there be any write operations in the database while the replication logger is turned off, these events will be lost for replication.

To turn on the replication logger for a database once, the following command can be executed:

```
require("org/arangodb/replication").logger.start();
```

Note: Starting the replication logger does not necessarily mean it will be started automatically on all following server startups. This can be configured separately (keep on reading for this).

Note: The above command turns on the replication logger for the current database only, but not for all databases. To turn on the replication logger for multiple databases, the above command must be executed separately for each database.

To turn the replication logger off for the current database, execute the *stop* command:

```
require("org/arangodb/replication").logger.stop();
```

This has turned off the logger, but still the logger may be started again automatically the next time the ArangoDB server is started. This can be configured separately (again, keep on reading).

To determine the current state of the replication logger in the current database (including whether it is currently running or not), use the *state* command:

```
require("org/arangodb/replication").logger.state();
```

The result might look like this:

```
{
  "state" : {
    "running" : false,
    "lastLogTick" : "255376126918573",
    "totalEvents" : 0,
    "time" : "2013-08-02T11:01:28Z"
  },
  "server" : {
    "version" : "1.4.devel",
    "serverId" : "53904504772335"
  },
  "clients" : [ ]
}
```

The *running* attribute indicates whether the logger is currently enabled and will log any events. The *totalEvents* attribute will indicate how many log events have been logged since the start of the ArangoDB server. The value will not be reset between multiple stops and restarts of the logger. Finally, the *lastLogTick* value will indicate the id of the last event that was logged. It can be used to determine whether new events were logged, and is also used by the replication applier for incremental fetching of data.

Note: Replication logger state can also be queried via the HTTP API.

Configuration

To determine whether the current database's replication logger is automatically started when the ArangoDB server is started, the logger has a separate configuration. The configuration is stored in a file *REPLICATION-LOGGER-CONFIG* inside the current database's directory. If it does not exist, ArangoDB will use default values.

To check and adjust the configuration of the replication logger for the current database, use the *properties* command. To view the current configuration, use *properties* without any arguments:

```
require("org/arangodb/replication").logger.properties();
```

The result might look like this:

```
{
  "autoStart" : false,
  "logRemoteChanges" : false,
  "maxEvents" : 1048576,
  "maxEventsSize" : 134217728
}
```

The *autoStart* attribute indicates whether the current database's replication logger is automatically started whenever the ArangoDB server is started. You may want to set it to *true* for a proper master setup. To do so, supply the updated attributes to the *properties* command, e.g.:

```
require("org/arangodb/replication").logger.properties({
  autoStart: true
});
```

This will ensure the replication logger for the current database is automatically started on all following ArangoDB starts unless *autoStart* is set to *false* again.

Note that only those attributes will be changed that you supplied in the argument to *properties*. All other configuration values will remain unchanged. Also note that the replication logger can be reconfigured while it is running.

It a replication logger is turned on, the event log may be allowed to grow indefinitely. It may be sensible to set a maximum size or a maximum number of events to keep. If one of these thresholds is reached during logging, the replication logger will start removing the oldest events from the event log automatically.

The thresholds are set by adjusting the *maxEvents* and *maxEventsSize* attributes via the *properties* command. The *maxEvents* attribute is the maximum number of events to keep in the event log before starting to remove oldest events. The *maxEventsSize* is the maximum cumulated size of event log data (in bytes) that is kept in the log before starting to remove oldest events. If both are set to a value of *0*, it means that the number and size of log events to keep is unrestricted. If both are set to a non-zero value, it means that there are restrictions on both the number and cumulated size of events to keep, and if either of the restrictions is hit, the deletion will be triggered.

The following command will set the threshold to 5000 events to keep, irrespective of the cumulated event data sizes:

```
require("org/arangodb/replication").logger.properties({
  maxEvents: 5000,
  maxEventsSize: 0
});
```

Replication Applier

Purpose

The purpose of the replication applier is to read data from a master database's event log, and apply them locally. The applier will check the master database for new events periodically. It will perform an incremental synchronization, i.e. only asking the master for events that occurred after the last synchronization.

The replication applier does not get notified by the master database when there are "new" events, but uses the pull principle. It might thus take some time (the *replication lag*) before an event from the master database gets shipped to and applied in a slave database.

The replication applier of a database is run in a separate thread. It may encounter problems when a log event from the master cannot be applied safely, or when the connection to the master database goes down (network outage, master database is down or unavailable etc.). In this case, the database's replication applier thread might terminate itself. It is then up to the administrator to fix the problem and restart the database's replication applier.

If the replication applier cannot connect to the master database, or the communication fails at some point

during the synchronization, the replication applier will try to reconnect to the master database. It will give up reconnecting only after a configurable amount of connection attempts.

The replication applier state is queryable at any time by using the *state* command of the applier. This will return the state of the applier of the current database:

```
require("org/arangodb/replication").applier.state();
```

The result might look like this:

```
{
  "state" : {
    "running" : false,
    "lastAppliedContinuousTick" : "231848832948633",
    "lastProcessedContinuousTick" : "231848832948633",
    "lastAvailableContinuousTick" : null,
    "progress" : {
      "time" : "2013-08-02T11:40:08Z",
      "message" : "applier created",
      "failedConnects" : 0
    },
    "totalRequests" : 0,
    "totalFailedConnects" : 0,
    "totalEvents" : 0,
    "lastError" : {
      "errorNum" : 0
    },
    "time" : "2013-08-02T11:40:22Z"
  },
  "server" : {
    "version" : "1.4.devel",
    "serverId" : "53904504772335"
  },
  "endpoint" : "tcp://master.domain.org:8529"
}
```

The *running* attribute indicates whether the replication applier of the current database is currently running and polling the server at *endpoint* for new events.

The *failedConnects* attribute shows how many failed connection attempts the replication applier currently has encountered in a row. In contrast, the *totalFailedConnects* attribute indicates how many failed connection attempts the applier has made in total. The *totalRequest* attribute shows how many requests the applier has sent to the master database in total. The *totalEvents* attribute shows how many log events the applier has read from the master.

The *message* sub-attribute of the *progress* sub-attribute gives a brief hint of what the applier currently does (if it is running). The *lastError* attribute also has an optional *errorMessage* sub-attribute, showing the latest error message. The *errorNum* sub-attribute of the *lastError* attribute can be used by clients to programmatically check for errors. It should be *0* if there is no error, and it should be non-zero when the applier terminated itself due to a problem.

Here is an example of the state after the replication applier terminated itself due to (repeated) connection problems:

```
{
  "state" : {
    "running" : false,
```

```
    "progress" : {
      "time" : "2013-08-02T12:40:25Z",
      "message" : "applier stopped",
      "failedConnects" : 6
    },
    "totalRequests" : 18,
    "totalFailedConnects" : 11,
    "totalEvents" : 0,
    "lastError" : {
      "time" : "2013-08-02T12:40:25Z",
      "errorMessage" : "could not connect to master at
 tcp://master.example.org:8529: Could not connect to 'tcp:/...",
      "errorNum" : 1400
    },
    ...
  }
}
```

Note: the state of a database's replication applier is queryable via the HTTP API, too. Please refer to HTTP Interface for Replication for more details.

Starting and Stopping

To start and stop the applier in the current database, the *start* and *stop* commands can be used:

```
require("org/arangodb/replication").applier.start(<tick>);
require("org/arangodb/replication").applier.stop();
```

Note: Starting a replication applier without setting up an initial configuration will fail. The replication applier will look for its configuration in a file named *REPLICATION-APPLIER-CONFIG* in the current database's directory. If the file is not present, ArangoDB will use some default configuration, but it cannot guess the endpoint (the address of the master database) the applier should connect to. Thus starting the applier without configuration will fail.

Note that the first time you start the applier, you should pass *1* as value for tick. Otherwise you might see the message

```
"lastError" : {
   "time" : "2014-04-16T16:59:16Z",
   "errorMessage" : "no start tick",
   "errorNum" : 1413
}
```

Note: Starting a database's replication applier via the *start* command will not necessarily start the applier on the next and following ArangoDB server restarts. Additionally, stopping a database's replication applier manually will not necessarily prevent the applier from being started again on the next server start. All of this is configurable separately (hang on reading).

Note: that when starting the replication applier of database, it will resume where it stopped. This is sensible because replication log events should be applied incrementally. If the replication applier of a database has never been started before, it needs some *tick* value from the master database's event log from which to start fetching events.

Configuration

To configure the replication applier of a specific database, use the *properties* command. Using it without

any arguments will return the current configuration:

```
require("org/arangodb/replication").applier.properties();
```

The result might look like this:

```
{
   "requestTimeout" : 300,
   "connectTimeout" : 10,
   "ignoreErrors" : 0,
   "maxConnectRetries" : 10,
   "chunkSize" : 0,
   "autoStart" : false,
   "adaptivePolling" : true
}
```

Note: There is no *endpoint* attribute configured yet. The *endpoint* attribute is required for the replication applier to be startable. You may also want to configure a username and password for the connection via the *username* and *password* attributes.

```
require("org/arangodb/replication").applier.properties({
   endpoint: "tcp://master.domain.org:8529",
   username:  "root",
   password: "secret"
});
```

This will re-configure the replication applier for the current database. The configuration will be used from the next start of the replication applier. The replication applier cannot be re-configured while it is running. It must be stopped first to be re-configured.

To make the replication applier of the current database start automatically when the ArangoDB server starts, use the *autoStart* attribute.

Setting the *adaptivePolling* attribute to *true* will make the replication applier poll the master database for changes with a variable frequency. The replication applier will then lower the frequency when the master is idle, and increase it when the master can provide new events). Otherwise the replication applier will poll the master database for changes with a constant frequency of 5 seconds if the master database's replication logger is turned off, and 0.5 seconds if it is turned on.

To set a timeout for connection and following request attempts, use the *connectTimeout* and *requestTimeout* values. The *maxConnectRetries* attribute configures after how many failed connection attempts in a row the replication applier will give up and turn itself off. You may want to set this to a high value so that temporary network outages do not lead to the replication applier stopping itself.

The *chunkSize* attribute can be used to control the approximate maximum size of a master's response (in bytes). Setting it to a low value may make the master respond faster (less data is assembled before the master sends the response), but may require more request-response roundtrips. Set it to *0* to use ArangoDB's built-in default value.

The following example will set most of the discussed properties for the current database's applier:

```
require("org/arangodb/replication").applier.properties({
   endpoint: "tcp://master.domain.org:8529",
   username:  "root",
   password: "secret",
```

```
    adaptivePolling: true,
    connectTimeout: 15,
    maxConnectRetries: 100,
    chunkSize: 262144,
    autoStart: true
});
```

After the applier is now fully configured, it could theoretically be started. However, we may first need an initial synchronization of all collections and their data from the master before we start the replication applier. The reason is that the replication logger of the master database may not contain all data we need for a full synchronization.

The only safe method for doing a full synchronization is thus to

- make sure the replication logger of the master database is running
- perform an initial full sync with the master database
- note the master database's replication logger *tick* value and
- start the continuous replication applier using this tick value.

The initial synchronization for the current database is executed with the *sync* command:

```
require("org/arangodb/replication").sync({
    endpoint: "tcp://master.domain.org:8529",
    username: "root",
    password: "secret
});
```

Warning: *sync* will do a full synchronization of the collections in the current database with collections present in the master database. Any local instances of the collections and all their data are removed! Only execute this command when you are sure you want to remove the local data!

As *sync* does a full synchronization, it may take a while to execute. When *sync* completes successfully, it show a list of collections it has synchronized in the *collections* attribute. It will also return the master database's replication logger tick value at the time the *sync* was started on the master. The tick value is contained in the *lastLogTick* attribute of the *sync* command:

```
{
    "lastLogTick" : "231848833079705",
    "collections" : [ ... ]
}
```

If the master database's replication logger is turned on, you can now start the continuous synchronization for the current database with the command

```
require("org/arangodb/replication").applier.start("231848833079705");
```

Note: The tick values should be handled as strings. Using numeric data types for tick values is unsafe because they might exceed the 32 bit value and the IEEE754 double accuracy ranges.

Example Setup

Setting up a working master-slave replication requires two ArangoDB instances:

- *master*: this is the instance we'll activate the replication logger on
- *slave*: on this instance, we'll start a replication applier, and this will fetch data from the master database's events log and apply all events locally

For the following example setup, we'll use the instance *tcp://master.domain.org:8529* as the master, and the instance *tcp://slave.domain.org:8530* as a slave.

The goal is to have all data from the database *_system* on master *tcp://master.domain.org:8529* be replicated to the database *_system* on the slave *tcp://slave.domain.org:8530*.

On the *master,* configure the replication logger for database *_system* to start automatically on ArangoDB startup. Additionally, set some restrictions for the event log size:

```
db._useDatabase("_system");

require("org/arangodb/replication").logger.properties({
  autoStart: true,
  maxEvents: 1048576,
  maxEventsSize: 0
});
```

After that, start the replication logger for the *_system* database on the master:

```
db._useDatabase("_system");
require("org/arangodb/replication").logger.start();
```

On the *slave*, make sure there currently is no replication applier running in the *_system* database:

```
db._useDatabase("_system");
require("org/arangodb/replication").applier.stop();
```

After that, do an initial sync of the slave with data from the master. Execute the following commands on the slave:

```
db._useDatabase("_system");
require("org/arangodb/replication").sync({
  endpoint: "tcp://master.example.org:8529",
  username: "myuser",
  password: "mypasswd"
});
```

Warning: This will replace data in the slave database with data from the master database! Only execute the commands if you have verified you are on the correct server, in the correct database!

Then re-configure the slave database's replication applier to point to the master database, set the username and password for the connection, and set the *autoStart* attribute:

```
db._useDatabase("_system");
require("org/arangodb/replication").applier.properties({
```

```
    endpoint: "tcp://master.example.org:8529",
    username: "myuser",
    password: "mypasswd",
    autoStart: true,
    adaptivePolling: true
});
```

After that, start the applier in the slave database:

```
db._useDatabase("_system");
require("org/arangodb/replication").applier.start();
```

After that, you should be able to monitor the state and progress of the replication applier by executing the *state* command in the slave database:

```
db._useDatabase("_system");
require("org/arangodb/replication").applier.state();
```

You may also want to check the master and slave states via the HTTP APIs (see HTTP Interface for Replication) or the web interface ("Dashboard" tab).

Replication Limitations

The replication in ArangoDB 1.4-alpha has a few limitations still. Some of these limitations may be removed in later versions of ArangoDB:

- the event log of the master database is currently written directly after a write operation is carried out in the master database. In case the master crashes between having executed the write operation and having it written into the event log, the write operation may have been executed on the master, but may be lost for replication and not be applied on any slaves.
- there is no feedback from the slaves to the master. If a slave cannot apply an event it got from the master, the master will have a different state of data. In this case, the replication applier on the slave will stop and report an error. Administrators can then either "fix" the problem or re-sync the data from the master to the slave and start the applier again.
- the replication is an asynchronous master*slave replication. There is currently no way to use it as a synchronous replication, or a multi-master replication.
- at the moment it is assumed that only the replication applier executes write operations on a slave. ArangoDB currently does not prevent users from carrying out their own write operations on slaves, though this might lead to undefined behavior and the replication applier stopping.
- the replication logger will log write operations for all user-defined collections and only some system collections. Write operations for the following system collections are excluded intentionally: _trx, _replication, _users, _aal, _fishbowl, _modules and _routing. Write operations for the following system collections will be logged: _aqlfunctions, _graphs.
- Foxx applications consist of database entries and application scripts in the file system. The file system parts of Foxx applications are not tracked anywhere and thus not replicated in current versions of ArangoDB. To replicate a Foxx application, it is required to copy the application to the remote server and install it there using the *foxx-manager* utility.
- master servers do not know which slaves are or will be connected to them. All servers in a replication setup are currently only loosely coupled. There currently is no way for a client to query which servers are present in a replication.
- failover must currently be handled by clients or client APIs.
- there currently is one replication logger and one replication applier per ArangoDB database. It is thus not possible to have a slave apply the events logs from multiple masters.
- replication is set up on a per-database level. When using ArangoDB with multiple databases, replication must be configured individually for each database.
- the replication applier is single-threaded, but write operations on the master may be executed in parallel if they affect different collections. Thus the replication applier might not be able to catch up with a very powerful and loaded master.
- replication is only supported between ArangoDB 1.4 masters and 1.4 slaves. It is currently not possible to replicate from/to other ArangoDB versions.
- a replication applier cannot apply data from itself.
- when doing the initial synchronization from a master to a slave using the *sync* method, collections are not prevented from being dropped in-between. That means that when a client fetches the data for collections C1, C2, and C3 (in this order), it will start with collection C1. Any other client may come along and drop collection C3 in the meantime. When the client reaches collection C3, it will not find this collection anymore, and the initial sync will fail. This will be fixed until the 1.4 stable release.

Replication Overhead

Running the replication logger will make all data modification operations more expensive, as the ArangoDB server will write the operations into the replication log, too.

Additionally, replication appliers that connect to an ArangoDB master will cause some extra work for the master as it needs to process incoming HTTP requests, and respond.

Overall, turning on the replication logger may reduce throughput on an ArangoDB master by some extent. If the replication feature is not required, the replication logger should be turned off.

Transactions are logged to the event log on the master as an uninterrupted sequence. While a transaction is written to the event log, the event log is blocked for other writes. Transactions should thus be as small as possible.

Replication Events

A database's replication logger in ArangoDB will log all its events into the *_replication* system collection in the same database. The *_replication* collection will therefore only contain events of a single database. Events will only be logged during the time the database's replication logger is enabled.

The following sections describe in detail the structure of the log events returned by the replication APIs.

Replication Event Types

The following replication event types will be logged by ArangoDB 1.4:

- 1000: the replication logger was stopped
- 1001: the replication logger was started
- 2000: a collection was created
- 2001: a collection was dropped
- 2002: a collection was renamed
- 2003: the properties of a collection were changed
- 2100: an index was created
- 2101: an index was dropped
- 2200: multi-document transaction started
- 2201: multi-document transaction committed
- 2300: document insert/update operation
- 2301: edge insert/update operation
- 2302: document/edge deletion

Each replication event itself is logged as a document with the following attributes. Only the *type* attribute is mandatory, and the other attributes are optional. Which attributes are used depends on the replication event type:

- *type*: mandatory. Indicates the type of the replication event (see list above)
- *cid*: optional. The id of the collection that is affected
- *tid*: optional. The id of the transaction the operation belongs to
- *key*: optional. Contains the document key for document operations
- *rev*: optional. Contains the document revision id for document operations
- *data*: optional. Contains the actual document/edge for document operations

When replication events are queried via the */_api/replication/logger-follow* API, the *tick* attribute will also be returned for each replication event. The *tick* value is a sequence number and is used by the replication applier to determine whether a replication event was already processed.

Examples

- 1000: the replication logger was stopped:

  This is an informational event. The *lastTick* attribute indicates the last tick value the replication logger had logged before it was stopped.

  ```
  {
    "tick":"247908129523087",
    "type":1000,
    "lastTick":"247908129326479"
  }
  ```

- 1001: the replication logger was started:

  This is an informational event. The *lastTick* attribute indicates the last tick value the replication

logger had logged before it was started.

The *tick* value is *0* if the logger was never started before:

```
{
   "tick":"247908129326479",
   "type":1001,
   "lastTick":"0"
}
```

If the logger was started before, the *tick* value will be non-zero:

```
{
   "tick":"247907874870862",
   "type":1001,
   "tick":"247907874674254"
}
```

- 2000: a collection was created:

  This event is logged when a collection is created. The *cid* attribute indicates the collection's id, and the *collection* attribute contains all details for the collection.

```
{
   "tick":"247908131947919",
   "type":2000,
   "cid":"247908131227023",
   "collection": {
     "version":4,
     "type":2,
     "cid":"247908131227023",
     "deleted":false,
     "doCompact":true,
     "maximalSize":33554432,
     "name":"UnitTestsReplication",
     "isVolatile":false,
     "waitForSync":false
   }
}
```

- 2001: a collection was dropped:

  This event is logged when a collection is dropped. The *cid* attribute indicates the id of the collection that is dropped:

```
{
   "tick":"247908135093647",
   "type":2001,
   "cid":"247908134045071"
}
```

- 2002: a collection was renamed:

  This event is logged when a collection is renamed. The *cid* attribute contains the id of the collection that is renamed. The *collection* attribute contains the *name* attribute, which contains the

new name for the collection:

```
{
   "tick":"247908136469903",
   "type":2002,
   "cid":"247908135421327",
   "collection": {
     "name":"UnitTestsReplication2"
   }
}
```

- 2003: the properties of a collection were changed:

  This event is logged when the properties of a collection are changed. The *cid* attribute contains the id of the affected collection. The *collection* attribute contains all the current (including the changed) properties. Note that renaming a collection will not trigger a property change, but a rename even:

```
{
   "tick":"247908139025807",
   "type":2003,
   "cid":"247908137977231",
   "collection": {
     "version":4,
     "type":2,
     "cid":"247908137977231",
     "deleted":false,
     "doCompact":true,
     "maximalSize":2097152,
     "name":"UnitTestsReplication",
     "isVolatile":false,
     "waitForSync":true
   }
}
```

- 2100: an index was created:

  This event is logged when a new index is created for a collection. The *cid* attribute contains the id of the collection the index is created for. The *index* attribute contains all index details. The *index* attribute will have at least the following sub-attributes:

  - *id*: the index id
  - *type*: the type of the index
  - *unique*: whether or not the index is unique

  All other sub-attributes of the *index* attribute vary depending on the index type.

  Note that as each collection will automatically have a primary index, the creation of the primary index will not be logged. For edge collections, the automatic creation of the edges index will also not be logged.

  Some examples for index creation events:

```
{
   "tick":"247908237985167",
   "type":2100,
   "cid":"247908236739983",
```

```
    "index": {
      "id":"247908237854095",
      "type":"geo2",
      "unique":false,
      "constraint":false,
      "fields": [
        "a",
        "b"
      ]
    }
  }

  {
    "tick":"247908234773903",
    "type":2100,
    "cid":"247908233594255",
    "index": {
      "id":"247908234642831",
      "type":"skiplist",
      "unique":true,
      "fields": [
        "a"
      ]
    }
  }

  {
    "tick":"247908229924239",
    "type":2100,
    "cid":"247908228679055",
    "index": {
      "id":"247908229793167",
      "type":"hash",
      "unique":true,
      "fields": [
        "a",
        "b"
      ]
    }
  }

  {
    "tick":"247908245915023",
    "type":2100,
    "cid":"247908244800911",
    "index": {
      "id":"247908245783951",
      "type":"cap",
      "unique":false,
      "size":100,
      "byteSize":0
    }
  }
```

- 2101: an index was dropped.

  This event is logged when an index of a collection is dropped. The *cid* attribute contains the id of the collection the index is dropped for. The *id* attribute contains the id of the dropped index:

```
{
   "tick":"247908249126287",
   "type":2101,
   "cid":"247908247815567",
   "id":"247908248667535"
}
```

- 2200: multi-document transaction started:

This event is logged at the start of a multi-document and/or multi-collection transaction. The *tid* attribute contains the transaction id. The *collections* attribute contains information about which collections are part of the transaction.

The *collections* attribute is a list of entries with the following attributes each:

  - *cid*: collection id
  - *name*: collection name
  - *operations*: number of operations following for the collection

Note that only those collections will be included that the transaction actually modifies. Collections that are not affected or that are only read from during a transaction will not be included:

```
{
   "tick":"247908333471119",
   "type":2200,
   "tid":"247908331963791",
   "collections": [
      {
         "cid":"247908330259855",
         "name":"UnitTestsReplication",
         "operations":1
      },
      {
         "cid":"247908330980751",
         "name":"UnitTestsReplication2",
         "operations":1
      }
   ]
}
```

Each transaction is logged as an uninterrupted sequence. All document operations following a transaction start event should have their *tid* attribute set to the current transaction's id. Successfully committed transactions are ended with a *transaction commit* event.

If a replication client encounters any other event type than document operations or a *transaction commit* event, the transaction can be considered as being failed and should not be applied by the client. In any case, clients should buffer transactional operations first, and apply them only if the transaction was committed successfully.

- 2201: multi-document transaction committed:

This event is logged for all successfully committed transactions. The event attributes are the same as for the *transaction start* event.

```
{
   "tick":"247908333864335",
   "type":2201,
   "tid":"247908331963791",
```

```
    "collections": [
      {
        "cid":"247908330259855",
        "name":"UnitTestsReplication",
        "operations":1
      },
      {
        "cid":"247908330980751",
        "name":"UnitTestsReplication2",
        "operations":1
      }
    ]
  }
```

- 2300: document insert/update operation

This event is logged for document insertions and updates. The *cid* attribute contains the id of the collection the document is inserted/updated in. The *key* attribute contains the document key, the *rev* attribute contains the document revision id.

If the *tid* attribute is present, the document was inserted/updated as part of a transaction. The *tid* attribute then contains the transaction's id. If no *tid* attribute is present, then the document was inserted/updated in a standalone operation.

If the event contains the *oldRev* attribute, then the operation was carried out as an update of an existing document. The *oldRev* attribute contains the revision id of the updated/replaced document. *data* will contain the full document and not just the updated/replaced parts.

If the *oldRev* attribute is not present, the operation was carried out as an insert. Note that some API methods cannot distinguish between inserts and updates, and will return any operation as an insert. Replication clients should therefore first check for document existance and then either carry out an insert or an update.

The *data* attribute contains the actual document, with the usual *_key* and *_rev* attributes plus all user-defined data.

Standalone document operation:

```
{
  "tick":"247908293100943",
  "type":2300,
  "cid":"247908289299855",
  "key":"abc",
  "rev":"247908292904335",
  "oldRev":"247908292117903",
  "data": {
    "_key":"abc",
    "_rev":"247908292904335",
    "test":4
  }
}
```

Transactional document operation:

```
{
  "tick":"247908333602191",
  "type":2300,
  "tid":"247908331963791",
```

```
    "cid":"247908330259855",
    "key":"12345",
    "rev":"247908332160399",
    "data": {
      "_key":"12345",
      "_rev":"247908332160399",
      "test":1
    }
  }
```

- 2301: edge insert/update operation:

  This event is logged for edge inserts/updates. The structure and meanings are the same as for the document insert/update event.

  Additional, the *data* attribute will contain the sub-attributes _from and _to, which indicate the vertices connected by the edge.

  _from and _to will contain the vertex collection id plus the vertex document key:

```
{
  "tick":"247908299916687",
  "type":2301,
  "cid":"247908298474895",
  "key":"abc",
  "rev":"247908299720079",
  "data": {
    "_key":"abc",
    "_rev":"247908299720079",
    "_from":"247908297753999/test1",
    "_to":"247908297753999/test2",
    "test":1
  }
}
```

- 2302: document/edge deletion:

  This event is logged for deletions of documents and edges. The *cid* attribute contains the id of the collection the document/edge is deleted in. The *key* attribute indicates the key of the deleted document/edge. The *rev* attribute contains the revision id of the deletion, and the *oldRev* attribute contains the revision id of the deleted document/edge.

  There is no *data* attribute with more details about the deleted document/edge.

  If the *tid* attribute is present, the document/edge was deleted as part of a transaction. The *tid* attribute then contains the transaction's id. If no *tid* attribute is present, then the document/edge was delete in a standalone operation.

```
{
  "tick":"247908303783311",
  "type":2302,
  "cid":"247908301882767",
  "key":"abc",
  "rev":"247908303586703",
  "oldRev":"247908302865807"
}
```

Transactions

Transactions are logged as an uninterrupted sequence, starting with a *transaction start* event, and finishing with a *transaction commit* event. Between these two events, all transactional document/edge operations will be logged.

All transactional document/edge operations will carry a *tid* attribute, which indicates the transaction id. Non-transactional operations will not carry a *tid* attribute.

Replication clients can apply non-transactional operations directly. In constrast, they should not apply transactional operations directly, but buffer them first and apply them altogether when they encounter the *transaction commit* event for the transaction.

If a replication client encounters some unexpected event during a transaction (i.e. some event that is neither a document/edge operation nor a *transaction commit* event), it should abort the ongoing transaction and discard all buffered operations. It can then consider the current transaction as failed.

Sharding

Sharding allows to use multiple machines to run a cluster of ArangoDB instances that together constitute a single database. This enables you to store much more data, since ArangoDB distributes the data automatically to the different servers. In many situations one can also reap a benefit in data throughput, again because the load can be distributed to multiple machines.

In a cluster there are essentially two types of processes: "DBservers" and "coordinators". The former actually store the data, the latter expose the database to the outside world. The clients talk to the coordinators exactly as they would talk to a single ArangoDB instance via the REST interface. The coordinators know about the configuration of the cluster and automatically forward the incoming requests to the right DBservers.

As a central highly available service to hold the cluster configuration and to synchronize reconfiguration and fail-over operations we currently use a an external program called *etcd* (see github page). It provides a hierarchical key value store with strong consistency and reliability promises. This is called the "agency" and its processes are called "agents".

All this is admittedly a relatively complicated setup and involves a lot of steps for the startup and shutdown of clusters. Therefore we have created convenience functionality to plan, setup, start and shutdown clusters.

The whole process works in two phases, first the "planning" phase and then the "running" phase. In the planning phase it is decided which processes with which roles run on which machine, which ports they use, where the central agency resides and what ports its agents use. The result of the planning phase is a "cluster plan", which is just a relatively big data structure in JSON format. You can then use this cluster plan to startup, shutdown, check and cleanup your cluster.

This latter phase uses so-called "dispatchers". A dispatcher is yet another ArangoDB instance and you have to install exactly one such instance on every machine that will take part in your cluster. No special configuration whatsoever is needed and you can organize authentication exactly as you would in a normal ArangoDB instance. You only have to activate the dispatcher functionality in the configuration file (see options *cluster.disable-dispatcher-kickstarter* and *cluster.disable-dispatcher-interface*, which are both initially set to *true* in the standard setup we ship).

However, you can use any of these dispatchers to plan and start your cluster. In the planning phase, you have to tell the planner about all dispatchers in your cluster and it will automatically distribute your agency, DBserver and coordinator processes amongst the dispatchers. The result is the cluster plan which you feed into the kickstarter. The kickstarter is a program that actually uses the dispatchers to manipulate the processes in your cluster. It runs on one of the dispatchers, which analyses the cluster plan and executes those actions, for which it is personally responsible, and forwards all other actions to the corresponding dispatchers. This is possible, because the cluster plan incorporates the information about all dispatchers.

We also offer a graphical user interface to the cluster planner and dispatcher.

How to try it out

In this text we assume that you are working with a standard installation of ArangoDB with at least a version number of 2.0. This means that everything is compiled for cluster operation, that *etcd* is compiled and the executable is installed in the location mentioned in the configuration file. The first step is to switch on the dispatcher functionality in your configuration of *arangod*. In order to do this, change the *cluster.disable-dispatcher-kickstarter* and *cluster.disable-dispatcher-interface* options in *arangod.conf* both to *false*.

Note: Once you switch *cluster.disable-dispatcher-interface* to *false*, the usual web front end is automatically replaced with the web front end for cluster planning. Therefore you can simply point your browser to *http://localhost:8529* (if you are running on the standard port) and you are guided through the planning and launching of a cluster with a graphical user interface. Alternatively, you can follow the instructions below to do the same on the command line interface.

We will first plan and launch a cluster, such that all your servers run on the local machine.

Start up a regular ArangoDB, either in console mode or connect to it with the Arango shell *arangosh*. Then you can ask it to plan a cluster for you:

```
arangodb> var Planner = require("org/arangodb/cluster").Planner;
arangodb> p = new Planner({numberOfDBservers:3,
numberOfCoordinators:2});
[object Object]
```

If you are curious you can look at the plan of your cluster:

```
arangodb> p.getPlan();
```

This will show you a huge JSON document. More interestingly, some further components tell you more about the layout of your cluster:

```
arangodb> p.DBservers;
[
  {
    "id" : "Pavel",
    "dispatcher" : "me",
    "port" : 8629
  },
  {
    "id" : "Perry",
    "dispatcher" : "me",
    "port" : 8630
  },
  {
    "id" : "Pancho",
    "dispatcher" : "me",
    "port" : 8631
  }
]

arangodb> p.coordinators;
[
  {
    "id" : "Claus",
```

```
      "dispatcher" : "me",
      "port" : 8530
    },
    {
      "id" : "Chantalle",
      "dispatcher" : "me",
      "port" : 8531
    }
  ]
```

This tells you the ports on which your ArangoDB processes will listen. We will need the 8530 (or whatever appears on your machine) for the coordinators below.

More interesting is that such a cluster plan document can be used to start up the cluster conveniently using a *Kickstarter* object. Please note that the *launch* method of the kickstarter shown below initialises all data directories and log files, so if you have previously used the same cluster plan you will lose all your data. Use the *relaunch* method described below instead in that case.

```
arangodb> var Kickstarter =
require("org/arangodb/cluster").Kickstarter;
arangodb> k = new Kickstarter(p.getPlan());
arangodb> k.launch();
```js

That is all you have to do, to fire up your first cluster. You will
see some
output, which you can safely ignore (as long as no error happens).

From that point on, you can contact one of the coordinators and use
the cluster
as if it were a single ArangoDB instance (use the port number from
above
instead of 8530, if you get a different one) (probably from another
shell window):

```js
$ arangosh --server.endpoint tcp://localhost:8530
[... some output omitted]
arangosh [_system]> db._listDatabases();
[
  "_system"
]
```js
This for example, lists the cluster wide databases.

Now, let us create a sharded collection. Note, that we only have to
specify
the number of shards to use in addition to the usual command.
The shards are automatically distributed among your DBservers:

```js
arangosh [_system]> example = db._create("example",
{numberOfShards:6});
[ArangoCollection 1000001, "example" (type document, status
loaded)]
arangosh [_system]> x = example.save({"name":"Hans", "age":44});
{
  "error" : false,
```

```
     "_id"  : "example/1000008",
     "_rev" : "13460426",
     "_key" : "1000008"
  }
  arangosh [_system]> example.document(x._key);
  {
     "age"  : 44,
     "name" : "Hans",
     "_id"  : "example/1000008",
     "_rev" : "13460426",
     "_key" : "1000008"
  }
  ```js

  You can shut down your cluster by using the following Kickstarter
  method (in the ArangoDB console):

  ```js
  arangodb> k.shutdown();
```

If you want to start your cluster again without losing data you have previously stored in it, you can use the *relaunch* method in exactly the same way as you previously used the *launch* method:

```
  arangodb> k.relaunch();
```

Note: If you have destroyed the object *k* for example because you have shutdown the ArangoDB instance in which you planned the cluster, then you can reproduce it for a *relaunch* operation, provided you have kept the cluster plan object provided by the *getPlan* method. If you had for example done:

```
  arangodb> var plan = p.getPlan();
  arangodb>
  require("fs").write("saved_plan.json",JSON.stringify(plan));
```

Then you can later do (in another session):

```
  arangodb> var plan = require("fs").read("saved_plan.json");
  arangodb> plan = JSON.parse(plan);
  arangodb> var Kickstarter =
  require("org/arangodb/cluster").Kickstarter;
  arangodb> var k = new Kickstarter(plan);
  arangodb> k.relaunch();
```

to start the existing cluster anew.

You can check, whether or not, all your cluster processes are still running, by issuing:

```
  arangodb> k.isHealthy();
```

This will show you the status of all processes in the cluster. You should see "RUNNING" there, in all the relevant places.

Finally, to clean up the whole cluster (losing all the data stored in it), do:

```
arangodb> k.shutdown();
arangodb> k.cleanup();
```

We conclude this section with another example using two machines, which will act as two dispatchers. We start from scratch using two machines, running on the network addresses *tcp://192.168.173.78:8529* and *tcp://192.168.173.13:6789*. Both need to have a regular ArangoDB instance installed and running. Please make sure, that both bind to all network devices, so that they can talk to each other. Also enable the dispatcher functionality on both of them, as described above.

```
arangodb> var Planner = require("org/arangodb/cluster").Planner;
arangodb> var p = new Planner({
    dispatchers: {"me":{"endpoint":"tcp://192.168.173.78:8529"},
                 "theother":
{"endpoint":"tcp://192.168.173.13:6789"}},
    "numberOfCoordinators":2, "numberOfDBservers": 2});
```

With these commands, you create a cluster plan involving two machines. The planner will put one DBserver and one Coordinator on each machine. You can now launch this cluster exactly as explained earlier:

```
    arangodb> var Kickstarter =
require("org/arangodb/cluster").Kickstarter;
    arangodb> k = new Kickstarter(p.getPlan());
    arangodb> k.launch();
```

Likewise, the methods *shutdown*, *relaunch*, *isHealthy* and *cleanup* work exactly as in the single server case.

See the corresponding chapter of the reference manual for detailed information about the *Planner* and *Kickstarter* classes.

Status of the implementation

This version 2.0 of ArangoDB contains the first usable implementation of the sharding extensions. However, not all planned features are included in this release. In particular, automatic fail-over is fully prepared in the architecture but is not yet implemented. If you use Version 2.0 in cluster mode in a production system, you have to organize failure recovery manually. This is why, at this stage with Version 2.0 we do not yet recommend to use the cluster mode in production systems. If you really need this feature now, please contact us.

This section provides an overview over the implemented and future features.

In normal single instance mode, ArangoDB works as usual with the same performance and functionality as in previous releases.

In cluster mode, the following things are implemented in version 2.0 and work:

- All basic CRUD operations for single documents and edges work essentially with good performance.
- One can use sharded collections and can configure the number of shards for each such collection individually. In particular, one can have fully sharded collections as well as cluster-wide available collections with only a single shard. After creation, these differences are transparent to the client.
- Creating and dropping cluster-wide databases works.
- Creating, dropping and modifying cluster-wide collections all work. Since these operations occur seldom, we will only improve their performance in a future release, when we will have our own implementation of the agency as well as a cluster-wide event managing system (see road map for release 2.3).
- Sharding in a collection, can be configured to use hashing on arbitrary properties of the documents in the collection.
- Creating and dropping indices on sharded collections works. Please note that an index on a sharded collection is not a global index but only leads to a local index of the same type on each shard.
- All SimpleQueries work. Again, we will improve the performance in future releases, when we revisit the AQL query optimizer (see road map for release 2.2).
- AQL queries work, but with relatively bad performance. Also, if the result of a query on a sharded collection is large, this can lead to an out of memory situation on the coordinator handling the request. We will improve this situation when we revisit the AQL query optimizer (see road map for release 2.2).
- Authentication on the cluster works with the method known from single ArangoDB instances on the coordinators. A new cluster-internal authorization scheme has been created. See below for hints on a sensible firewall and authorization setup.
- Most standard API calls of the REST interface work on the cluster as usual, with a few exceptions, which do no longer make sense on a cluster or are harder to implement. See below for details.

The following does not yet work, but is planned for future releases (see road map):

- Transactions can be run, but do not behave like transactions. They simply execute but have no atomicity or isolation in version 2.0. See the road map for version 2.X.
- We plan to revise the AQL optimizer for version 2.2. This is necessary since for efficient queries in cluster mode we have to do as much as possible of the filtering and sorting on the individual DBservers rather than on the coordinator.
- Our software architecture is fully prepared for replication, automatic fail-over and recovery of a cluster, which will be implemented for version 2.3 (see our road map).
- This setup will at the same time, allow for hot swap and in-service maintenance and scaling of a cluster. However, in version 2.0 the cluster layout is static and no redistribution of data between the DBservers or moving of shards between servers is possible.
- For version 2.3 we envision an extension for AQL to allow writing queries. This will also allow to run modifying queries in parallel on all shards.
- At this stage the sharding of an edge collection is independent of the sharding of the corresponding vertex collection in a graph. For version 2.2 we plan to synchronize the two, to

allow for more efficient graph traversal functions in large, sharded graphs. We will also do research on distributed algorithms for graphs and implement new algorithms in ArangoDB. However, at this stage, all graph traversal algorithms are executed on the coordinator and this means relatively poor performance since every single edge step leads to a network exchange.

- In version 2.0 the import API is broken for sharded collections. It will appear to work but will in fact silently fail. Fixing this is on the road map for version 2.1.
- In version 2.0 the *arangodump* and *arangorestore* programs can not be used talking to a coordinator to directly backup sharded collections. At this stage, one has to backup the DBservers individually using *arangodump* and *arangorestore* on them. The coordinators themselves do not hold any state and therefore do not need backup. Do not forget to backup the meta data stored in the agency because this is essential to access the sharded collections. These limitations will be fixed in version 2.1.
- In version 2.0 the replication API (*/_api/replication*) does not work on coordinators. This is intentional, since the plan is to organize replication with automatic fail-over directly on the DBservers, which is planned for version 2.3.
- The *db..rotate()* method for sharded collections is not yet implemented, but will be supported from version 2.1 onwards.
- The *db..rename()* method for sharded collections is not yet implemented, but will be supported from version 2.1 onwards.
- The *db..checksum()* method for sharded collections is not yet implemented, but will be supported from version 2.1 onwards.

The following restrictions will probably stay, for cluster mode, even in future versions. This is, because they are difficult or even impossible to implement efficiently:

- Custom key generators with the *keyOptions* property in the *_create* method for collections are not supported. We plan to improve this for version 2.1 (see road map). However, due to the distributed nature of a sharded collection, not everything that is possible in the single instance situation will be possible on a cluster. For example the auto-increment feature in a cluster with multiple DBservers and coordinators would have to lock the whole collection centrally for every document creation, which essentially defeats the performance purpose of sharding.
- Unique constraints on non-sharding keys are unsupported. The reason for this is that we do not plan to have global indices for sharded collections. Therefore, there is no single authority that could efficiently decide whether or not the unique constraint is satisfied by a new document. The only possibility would be to have a central locking mechanism and use heavy communication for every document creation to ensure the unique constraint.
- The method *db..revision()* for a sharded collection returns the highest revision number from all shards. However, revision numbers are assigned per shard, so this is not guaranteed to be the revision of the latest inserted document. Again, maintaining a global revision number over all shards is very difficult to maintain efficiently.
- The methods *db..first()* and *db..last()* are unsupported for collections with more than one shard. The reason for this, is that temporal order in a highly parallelized environment like a cluster is difficult or even impossible to achieve efficiently. In a cluster it is entirely possible that two different coordinators add two different documents to two different shards *at the same time*. In such a situation it is not even well-defined which of the two documents is "later". The only way to overcome this fundamental problem would again be a central locking mechanism, which is not desirable for performance reasons.
- Contrary to the situation in a single instance, objects representing sharded collections are broken after their database is dropped. In a future version they might report that they are broken, but it is not feasible and not desirable to retain the cluster database in the background until all collection objects are garbage collected.
- In a cluster, the automatic creation of collections on a call to *db._save(ID)* is not supported. This is because one would have no way to specify the number or distribution of shards for the newly created collection. Therefore we will not offer this feature for cluster mode.

Authentication in a cluster

In this section we describe, how authentication in a cluster is done properly. For experiments it is possible to run the cluster completely unauthorized by using the option *--server.disable-authentication true* on the command line or the corresponding entry in the configuration file. However, for production use, this is not desirable.

You can turn on authentication in the cluster by switching it on in the configuration of your dispatchers. When you now use the planner and kickstarter to create and launch a cluster, the *arangod* processes in your cluster will automatically run with authentication, exactly as the dispatchers themselves. However, the cluster will have a sharded collection *_users* with one shard containing only the user *root* with an empty password. We emphasize that this sharded cluster-wide collection is different from the *_users* collections in each dispatcher!

The coordinators in your cluster will use this cluster-wide sharded collection to authenticate HTTP requests. If you add users using the usual methods via a coordinator, you will in fact change the cluster-wide collection *_users* and thus all coordinators will eventually see the new users and authenticate against them. "Eventually" means that they might need a few seconds to notice the change in user setup and update their user cache.

The DBservers will have their authentication switched on as well. However, they do not use the cluster-wide *_users* collection for authentication, because the idea is, that the outside clients do not talk to the DBservers directly, but always go via the coordinators. For the cluster-internal communication between coordinators and DBservers (in both directions), we use a simpler setup: There are two new configuration options *cluster.username* and *cluster.password*, which default to *root* and the empty password *""*. If you want to deviate from this default you have to change these two configuration options in all configuration files on all machines in the cluster. This just means that you have to set these two options to the same values in all configuration files *arangod.conf* in all dispatchers, since the coordinators and DBservers will simply inherit this configuration file from the dispatcher that has launched them.

Let us summarize what you have to do, to enable authentication in a cluster:

1. Set *server.disable-authentication* to *false* in all configuration files of all dispatchers (this is already the default).
2. Put the same values for *cluster.username* and *cluster.password* in the very same configuration files of all dispatchers.
3. Create users via the usual interface on the coordinators (initially after the cluster launch there will be a single user *root* with empty password).

Please note, that in Version 2.0 of ArangoDB you can already configure the endpoints of the coordinators to use SSL. However, this is not yet conveniently supported in the planner, kickstarter and in the graphical cluster management tools. We will fix this in the next version.

Please also consider the comments in the following section about firewall setup.

Recommended firewall setup

This section is intended for people who run a cluster in production systems.

The whole idea of the cluster setup is that the coordinators serve HTTP requests to the outside world and that all other processes (DBservers and agency) are only available from within the cluster itself. Therefore, in a production environment, one has to put the whole cluster behind a firewall and only open the ports to the coordinators to the client processes.

Note however that for the asynchronous cluster-internal communication, the DBservers perform HTTP requests to the coordinators, which means that the coordinators must also be reachable from within the cluster.

Furthermore, it is of the utmost importance to hide the agent processes of the agency behind the firewall, since, at least at this stage, requests to them are completely unauthorized. Leaving their ports exposed to the outside world, endangers all data in the cluster, because everybody on the internet could make the cluster believe that, for example, you wanted your databases dropped! This weakness will be alleviated in future versions, because we will replace *etcd* by our own specialized agency implementation, which will allow for authentication.

A further comment applies to the dispatchers. Usually you will open the HTTP endpoints of your dispatchers to the outside world and switch on authentication for them. This is necessary to contact them from the outside, in the cluster launch phase. However, actually you only need to contact one of them, who will then in turn contact the others using cluster-internal communication. You can even get away with closing access to all dispatchers to the outside world, provided the machine running your browser is within the cluster network and does not have to go through the firewall to contact the dispatchers. It is important to be aware that anybody who can reach a dispatcher and can authorize himself to it can launch arbitrary processes on the machine on which the dispatcher runs!

Therefore we recommend to use SSL endpoints with user/password authentication on the dispatchers *and* to block access to them in the firewall. You then have to launch the cluster using an *arangosh* or browser running within the cluster.

Command-line options

Configuration Files

Options can be specified on the command line or in configuration files. If a string *Variable* occurs in the value, it is replaced by the corresponding environment variable.

General Options

General help

```
--help
```

```
-h
```

Prints a list of the most common options available and then exits. In order to see all options use *--help-all*.

Upgrade

```
--upgrade
```

Specifying this option will make the server perform a database upgrade at start. A database upgrade will first compare the version number stored in the file VERSION in the database directory with the current server version.

If the two version numbers match, the server will start normally.

If the version number found in the database directory is higher than the version number the server is running, the server expects this is an unintentional downgrade and will warn about this. It will however start normally. Using the server in these conditions is however not recommended nor supported.

If the version number found in the database directory is lower than the version number the server is running, the server will check whether there are any upgrade tasks to perform. It will then execute all required upgrade tasks and print their statuses. If one of the upgrade tasks fails, the server will exit and refuse to start. Re-starting the server with the upgrade option will then again trigger the upgrade check and execution until the problem is fixed. If all tasks are finished, the server will start normally.

Whether or not this option is specified, the server will always perform a version check on startup. Running the server with a non-matching version number in the VERSION file will make the server refuse to start.

Configuration

```
--configuration filename
```

```
-c filename
```

Specifies the name of the configuration file to use.

If this command is not passed to the server, then by default, the server will attempt to first locate a file named *~/.arango/arangod.conf* in the user's home directory.

If no such file is found, the server will proceed to look for a file *arangod.conf* in the system configuration directory. The system configuration directory is platform-specific, and may be changed when compiling ArangoDB yourself. It may default to */etc/arangodb* or */usr/local/etc/arangodb*. This file is installed when using a package manager like rpm or dpkg. If you modify this file and later upgrade to a new version of ArangoDB, then the package manager normally warns you about the conflict. In order to avoid these warning for small adjustments, you can put local overrides into a file *arangod.conf.local*.

Only command line options with a value should be set within the configuration file. Command line options which act as flags should be entered on the command line when starting the server.

Whitespace in the configuration file is ignored. Each option is specified on a separate line in the form

```
key = value
```

Alternatively, a header section can be specified and options pertaining to that section can be specified in a shorter form

```
[log]
level = trace
```

rather than specifying

```
log.level = trace
```

Comments can be placed in the configuration file, only if the line begins with one or more hash symbols (#).

There may be occasions where a configuration file exists and the user wishes to override configuration settings stored in a configuration file. Any settings specified on the command line will overwrite the same setting when it appears in a configuration file. If the user wishes to completely ignore configuration files without necessarily deleting the file (or files), then add the command line option

```
-c none
```

or

```
--configuration none
```

When starting up the server. Note that, the word *none* is case-insensitive.

Daemon

```
--daemon
```

Runs the server as a daemon (as a background process). This parameter can only be set if the pid (process id) file is specified. That is, unless a value to the parameter pid-file is given, then the server will report an error and exit.

Default Language

```
--default-language default-language
```

The default language ist used for sorting and comparing strings. The language value is a two-letter language code (ISO-639) or it is composed by a two-letter language code with and a two letter country code (ISO-3166). Valid languages are "de", "en", "en_US" or "en_UK".

The default default-language is set to be the system locale on that platform.

Supervisor

```
--supervisor
```

Executes the server in supervisor mode. In the event that the server unexpectedly terminates due to an internal error, the supervisor will automatically restart the server. Setting this flag automatically implies that the server will run as a daemon. Note that, as with the daemon flag, this flag requires that the pid-file parameter will set.

```
unix> ./arangod --supervisor --pid-file /var/run/arangodb.pid
```

```
/tmp/vocbase/
2012-06-27T15:58:28Z [10133] INFO starting up in supervisor mode
```

As can be seen (e.g. by executing the ps command), this will start a supervisor process and the actual database process:

```
unix> ps fax | grep arangod
10137 ?          Ssl    0:00 ./arangod --supervisor --pid-file
/var/run/arangodb.pid /tmp/vocbase/
10142 ?          Sl     0:00  \_ ./arangod --supervisor --pid-file
/var/run/arangodb.pid /tmp/vocbase/
```

When the database process terminates unexpectedly, the supervisor process will start up a new database process:

```
> kill -SIGSEGV 10142

> ps fax | grep arangod
10137 ?          Ssl    0:00 ./arangod --supervisor --pid-file
/var/run/arangodb.pid /tmp/vocbase/
10168 ?          Sl     0:00  \_ ./arangod --supervisor --pid-file
/var/run/arangodb.pid /tmp/vocbase/
```

User identity

```
--uid uid
```

The name (identity) of the user the server will run as. If this parameter is not specified, the server will not attempt to change its UID, so that the UID used by the server will be the same as the UID of the user who started the server. If this parameter is specified, then the server will change its UID after opening ports and reading configuration files, but before accepting connections or opening other files (such as recovery files). This is useful when the server must be started with raised privileges (in certain environments) but security considerations require that these privileges be dropped once the server has started work.

Observe that this parameter cannot be used to bypass operating system security. In general, this parameter (and its corresponding relative gid) can lower privileges but not raise them.

Group identity

```
--gid gid
```

The name (identity) of the group the server will run as. If this parameter is not specified, then the server will not attempt to change its GID, so that the GID the server runs as will be the primary group of the user who started the server. If this parameter is specified, then the server will change its GID after opening ports and reading configuration files, but before accepting connections or opening other files (such as recovery files).

This parameter is related to the parameter uid.

Process identity

```
--pid-file filename
```

The name of the process ID file to use when running the server as a daemon. This parameter must be specified if either the flag *daemon* or *supervisor* is set.

Console

```
--console
```

Runs the server in an exclusive emergency console mode. When starting the server with this option, the server is started with an interactive JavaScript emergency console, with all networking and HTTP interfaces of the server disabled.

No requests can be made to the server in this mode, and the only way to work with the server in this mode is by using the emergency console. Note that the server cannot be started in this mode if it is already running in this or another mode.

Command-Line Options for Development

```
--development-mode
```

Specifying this option will start the server in development mode. The development mode forces reloading of all actions and Foxx applications on every HTTP request. This is very resource-intensive and slow, but makes developing server-side actions and Foxx applications much easier.

WARNING: Never use this option in production.

Command-Line Options for arangod

Endpoint

```
--server.endpoint endpoint
```

Specifies an *endpoint* for HTTP requests by clients. Endpoints have the following pattern:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6
- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption
- unix:path/to/socket - Unix domain socket endpoint

If a TCP/IP endpoint is specified without a port number, then the default port (8529) will be used. If multiple endpoints need to be used, the option can be repeated multiple times.

Examples

```
unix> ./arangod --server.endpoint tcp://127.0.0.1:8529
  --server.endpoint ssl://127.0.0.1:8530
  --server.keyfile server.pem /tmp/vocbase
2012-07-26T07:07:47Z [8161] INFO using SSL protocol version 'TLSv1'
2012-07-26T07:07:48Z [8161] INFO using endpoint
'ssl://127.0.0.1:8530' for http ssl requests
2012-07-26T07:07:48Z [8161] INFO using endpoint
'tcp://127.0.0.1:8529' for http tcp requests
2012-07-26T07:07:49Z [8161] INFO ArangoDB (version 1.1.alpha) is
ready for business
2012-07-26T07:07:49Z [8161] INFO Have Fun!
```

Note: If you are using SSL-encrypted endpoints, you must also supply the path to a server certificate using the --server.keyfile option.

Endpoints can also be changed at runtime. Please refer to @ref HttpEndpoint for more details.

Reuse address

```
--server.reuse-address
```

If this boolean option is set to *true* then the socket option SO_REUSEADDR is set on all server endpoints, which is the default. If this option is set to *false* it is possible that it takes up to a minute after a server has terminated until it is possible for a new server to use the same endpoint again. This is why this is activated by default.

Please note however that under some operating systems this can be a security risk because it might be possible for another process to bind to the same address and port, possibly hijacking network traffic. Under Windows, ArangoDB additionally sets the flag SO_EXCLUSIVEADDRUSE as a measure to alleviate this problem.

Disable authentication

```
--server.disable-authentication
```

Setting value to true will turn off authentication on the server side so all clients can execute any action without authorization and privilege checks.

The default value is *false*.

Disable authentication-unix-sockets

```
--server.disable-authentication-unix-sockets value
```

Setting *value* to true will turn off authentication on the server side for requests coming in via UNIX domain sockets. With this flag enabled, clients located on the same host as the ArangoDB server can use UNIX domain sockets to connect to the server without authentication. Requests coming in by other means (e.g. TCP/IP) are not affected by this option.

The default value is *false*.

Note: this option is only available on platforms that support UNIX domain sockets.

Authenticate system only

```
--server.authenticate-system-only boolean
```

Controls whether incoming requests need authentication only if they are directed to the ArangoDB's internal APIs and features, located at */_api/*, */_admin/* etc.

IF the flag is set to *true*, then HTTP authentication is only required for requests going to URLs starting with */_*, but not for other URLs. The flag can thus be used to expose a user-made API without HTTP authentication to the outside world, but to prevent the outside world from using the ArangoDB API and the admin interface without authentication. Note that checking the URL is performed after any database name prefix has been removed. That means when the actual URL called is */_db/_system/myapp/myaction*, the URL */myapp/myaction* will be used for *authenticate-system-only* check.

The default is *false*.

Note that authentication still needs to be enabled for the server regularly in order for HTTP authentication to be forced for the ArangoDB API and the web interface. Setting only this flag is not enough.

You can control ArangoDB's general authentication feature with the *--server.disable-authentication* flag.

Disable replication-logger

```
--server.disable-replication-logger flag
```

If *true* the server will start with the replication logger turned off, even if the replication logger is configured with the autoStart option. Using this option will not change the value of the autoStart option in the logger configuration, but will suppress auto-starting the replication logger just once.

If the option is not used, ArangoDB will read the logger configuration from the file REPLICATION-LOGGER-CONFIG on startup, and use the value of the autoStart attribute from this file.

The default is *false*.

Disable replication-applier

```
--server.disable-replication-applier flag
```

If *true* the server will start with the replication applier turned off, even if the replication applier is configured with the *autoStart* option. Using the command-line option will not change the value of the *autoStart* option in the applier configuration, but will suppress auto-starting the replication applier just once.

If the option is not used, ArangoDB will read the applier configuration from the file *REPLICATION-APPLIER-CONFIG* on startup, and use the value of the *autoStart* attribute from this file.

The default is *false*.

Timeout

```
--server.keep-alive-timeout
```

Allows to specify the timeout for HTTP keep-alive connections. The timeout value must be specified in seconds. Idle keep-alive connections will be closed by the server automatically when the timeout is reached. A keep-alive-timeout value 0 will disable the keep alive feature entirely.

Default Api

```
--server.default-api-compatibility
```

This option can be used to determine the API compatibility of the ArangoDB server. It expects an ArangoDB version number as an integer, calculated as follows:

10000 \ major + 100 * minor (example: *10400* for ArangoDB 1.4)*

The value of this option will have an influence on some API return values when the HTTP client used does not send any compatibility information.

In most cases it will be sufficient to not set this option explicitly but to keep the default value. However, in case an "old" ArangoDB client is used that does not send any compatibility information and that cannot handle the responses of the current version of ArangoDB, it might be reasonable to set the option to an old version number to improve compatibility with older clients.

Allow method arrive

```
--server.allow-method-override
```

When this option is set to *true*, the HTTP request method will optionally be fetched from one of the following HTTP request headers if present in the request:

- *x-http-method*
- *x-http-method-override*
- *x-method-override*

If the option is set to *true* and any of these headers is set, the request method will be overriden by the value of the header. For example, this allows issuing an HTTP DELETE request which to the outside world will look like an HTTP GET request. This allows bypassing proxies and tools that will only let certain request types pass.

Setting this option to *true* may impose a security risk so it should only be used in controlled environments.

The default value for this option is *false*.

Keyfile

```
--server.keyfile filename
```

If SSL encryption is used, this option must be used to specify the filename of the server private key. The file must be PEM formatted and contain both the certificate and the server's private key.

The file specified by *filename* should have the following structure:

```
# create private key in file "server.key"
openssl genrsa -des3 -out server.key 1024
```

```
# create certificate signing request (csr) in file "server.csr"
openssl req -new -key server.key -out server.csr


# copy away original private key to "server.key.org"
cp server.key server.key.org


# remove passphrase from the private key
openssl rsa -in server.key.org -out server.key


# sign the csr with the key, creates certificate PEM file
"server.crt"
openssl x509 -req -days 365 -in server.csr -signkey server.key -out
server.crt


# combine certificate and key into single PEM file "server.pem"
cat server.crt server.key > server.pem
```

You may use certificates issued by a Certificate Authority or self-signed certificates. Self-signed certificates can be created by a tool of your choice. When using OpenSSL for creating the self-signed certificate, the following commands should create a valid keyfile:

```
-----BEGIN CERTIFICATE-----


(base64 encoded certificate)


-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----


(base64 encoded private key)


-----END RSA PRIVATE KEY-----
```

For further information please check the manuals of the tools you use to create the certificate.

Note: the --server.keyfile option must be set if the server is started with at least one SSL endpoint.

Cafile

```
--server.cafile filename
```

This option can be used to specify a file with CA certificates that are sent to the client whenever the server requests a client certificate. If the file is specified, The server will only accept client requests with certificates issued by these CAs. Do not specify this option if you want clients to be able to connect without specific certificates.

The certificates in *filename* must be PEM formatted.

Note: this option is only relevant if at least one SSL endpoint is used.

SSL protocol

`--server.ssl-protocolvalue`

Use this option to specify the default encryption protocol to be used. The following variants are available:

- 1: SSLv2
- 2: SSLv23
- 3: SSLv3
- 4: TLSv1

The default *value* is 4 (i.e. TLSv1).

Note: this option is only relevant if at least one SSL endpoint is used.

SSL Cache

`--server.ssl-cache value`

Set to true if SSL session caching should be used.

value has a default value of *false* (i.e. no caching).

Note: this option is only relevant if at least one SSL endpoint is used, and only if the client supports sending the session id.

SSL options

`--server.ssl-options value`

This option can be used to set various SSL-related options. Individual option values must be combined using bitwise OR.

Which options are available on your platform is determined by the OpenSSL version you use. The list of options available on your platform might be retrieved by the following shell command:

```
> grep "#define SSL_OP_.*" /usr/include/openssl/ssl.h


#define SSL_OP_MICROSOFT_SESS_ID_BUG              0x00000001L
#define SSL_OP_NETSCAPE_CHALLENGE_BUG             0x00000002L
#define SSL_OP_LEGACY_SERVER_CONNECT              0x00000004L
#define SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG   0x00000008L
#define SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG        0x00000010L
#define SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER         0x00000020L
...
```

A description of the options can be found online in the [OpenSSL documentation](#)

Note: this option is only relevant if at least one SSL endpoint is used.

SSL Cipher

`--server.ssl-cipher-list cipher-list`

This option can be used to restrict the server to certain SSL ciphers only, and to define the relative usage preference of SSL ciphers.

The format of *cipher-list* is documented in the OpenSSL documentation.

To check which ciphers are available on your platform, you may use the following shell command:

```
> openssl ciphers -v


ECDHE-RSA-AES256-SHA     SSLv3 Kx=ECDH     Au=RSA   Enc=AES(256)
Mac=SHA1
ECDHE-ECDSA-AES256-SHA   SSLv3 Kx=ECDH     Au=ECDSA Enc=AES(256)
Mac=SHA1
DHE-RSA-AES256-SHA       SSLv3 Kx=DH       Au=RSA   Enc=AES(256)
Mac=SHA1
DHE-DSS-AES256-SHA       SSLv3 Kx=DH       Au=DSS   Enc=AES(256)
Mac=SHA1
DHE-RSA-CAMELLIA256-SHA SSLv3 Kx=DH        Au=RSA   Enc=Camellia(256)
Mac=SHA1
...
```

The default value for *cipher-list* is "ALL".

Note: this option is only relevant if at least one SSL endpoint is used.

Backlog

```
--server.backlog-size
```

Allows to specify the size of the backlog for the *listen* system call The default value is 10. The maximum value is platform-dependent.

Disable statics

```
--disable-statistics value
```

If this option is *value* is *true*, then ArangoDB's statistics gathering is turned off. Statistics gathering causes constant CPU activity so using this option to turn it off might relieve heavy-loaded instances. Note: this option is only available when ArangoDB has not been compiled with the option --*disable-figures*.

Directory

```
--database.directory directory
```

The directory containing the collections and datafiles. Defaults to */var/lib/arango*. When specifying the database directory, please make sure the directory is actually writable by the arangod process.

You should further not use a database directory which is provided by a network filesystem such as NFS. The reason is that networked filesystems might cause inconsistencies when there are multiple parallel readers or writers or they lack features required by arangod (e.g. flock()).

```
directory
```

When using the command line version, you can simply supply the database directory as argument.

Examples

```
> ./arangod --server.endpoint tcp://127.0.0.1:8529 --
database.directory /tmp/vocbase
```

Journal size

```
--database.maximal-journal-size size
```

Maximal size of journal in bytes. Can be overwritten when creating a new collection. Note that this also limits the maximal size of a single document.

The default is *32MB*.

Wait for sync

```
--database.wait-for-sync boolean
```

Default wait-for-sync value. Can be overwritten when creating a new collection.

The default is *false*.

Sync properties

```
--database.force-sync-properties boolean
```

Force syncing of collection properties to disk after creating a collection or updating its properties.

If turned off, syncing will still happen for collection that have a waitForSync value of *true*. If turned on, syncing of properties will always happen, regardless of the value of waitForSync.

The default is *true*.

Frequency

```
--javascript.gc-frequency frequency
```

Specifies the frequency (in seconds) for the automatic garbage collection of JavaScript objects. This setting is useful to have the garbage collection still work in periods with no or little numbers of requests.

Startup gc-interval

```
--javascript.gc-interval interval
```

Specifies the interval (approximately in number of requests) that the garbage collection for JavaScript objects will be run in each thread.

V8 options

```
--javascript.v8-options options
```

Optional arguments to pass to the V8 Javascript engine. The V8 engine will run with default settings unless explicit options are specified using this option. The options passed will be forwarded to the V8 engine which will parse them on its own. Passing invalid options may result in an error being printed on stderr and the option being ignored.

Options need to be passed in one string, with V8 option names being prefixed with double dashes. Multiple options need to be separated by whitespace. To get a list of all available V8 options, you can use the value *"--help"* as follows:

```
  --javascript.v8-options "--help"
```

Another example of specific V8 options being set at startup:

```
  --javascript.v8-options "--harmony --log"
```

Names and features or usable options depend on the version of V8 being used, and might change in the future if a different version of V8 is being used in ArangoDB. Not all options offered by V8 might be sensible to use in the context of ArangoDB. Use the specific options only if you are sure that they are not harmful for the regular database operation.

JavaScript Interface for managing Endpoints

The ArangoDB server can listen for incoming requests on multiple *endpoints*.

The endpoints are normally specified either in ArangoDB's configuration file or on the command-line, using the ["--server.endpoint"](#) option. The default endpoint for ArangoDB is *tcp://127.0.0.1:8529* or *tcp://localhost:8529*.

The number of endpoints can also be changed at runtime using the API described below. Each endpoint can optionally be restricted to a specific list of databases only, thus allowing the usage of different port numbers for different databases.

This may be useful in multi-tenant setups. A multi-endpoint setup may also be useful to turn on encrypted communication for just specific databases.

The JavaScript interface for endpoints provides operations to add new endpoints at runtime, and optionally restrict them for use with specific databases. The interface also can be used to update existing endpoints or remove them at runtime.

Please note that all endpoint management operations can only be accessed via the default database (*_system*) and none of the other databases.

When not in the default database, you must first switch to it using the *db._useDatabase* method.

Configuring and Removing Endpoints

List

```
db._listEndpoints()
```

Returns a list of all endpoints and their mapped databases.

Please note that managing endpoints can only be performed from out of the _system_ database. When not in the default database, you must first switch to it using the "db._useDatabase" method.

Configure

```
db._configureEndpoint(endpoint, databases)
```

Adds and connects or updates the *endpoint*.

The optional *databases* argument allows restricting the endpoint for use with specific databases only. The first database in the list will automatically become the default database for the endpoint. The default database will be used for incoming requests that do not specify the database name explicitly.

If *databases* is an empty list, the endpoint will allow access to all existing databases.

The adjusted list of endpoints is saved in a file *ENDPOINTS* in the database directory. The endpoints are restored from the file at server start.

Please note that managing endpoints can only be performed from out of the _system_ database. When not in the default database, you must first switch to it using the "db._useDatabase" method.

Remove

```
db._removeEndpoint(endpoint)
```

Disconnects and removes the *endpoint*. If the endpoint was not configured before, the operation will fail. If the endpoint happens to be the last bound endpoint, the operation will also fail as disconnecting would make the server unable to communicate with any clients.

The adjusted list of endpoints is saved in a file *ENDPOINTS* in the database directory. The endpoints are restored from the file at server start.

Please note that managing endpoints can only be performed from out of the _system_ database. When not in the default database, you must first switch to it using the "db._useDatabase" method.

Command-Line Options for Clusters

Agency endpoint

```
--cluster.agency-endpoint endpoint
```

An agency endpoint the server can connect to. The option can be specified multiple times so the server can use a cluster of agency servers. Endpoints have the following pattern:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6
- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption

At least one endpoint must be specified or ArangoDB will refuse to start. It is recommended to specify at least two endpoints so ArangoDB has an alternative endpoint if one of them becomes unavailable.

Examples

```
--cluster.agency-endpoint tcp://192.168.1.1:4001 --cluster.agency-
endpoint tcp://192.168.1.2:4002
```

Agency prefix

```
--cluster.agency-prefix prefix
```

The global key prefix used in all requests to the agency. The specified prefix will become part of each agency key. Specifying the key prefix allows managing multiple ArangoDB clusters with the same agency server(s).

prefix must consist of the letters *a-z*, *A-Z* and the digits *0-9* only. Specifying a prefix is mandatory.

Examples

```
--cluster.prefix mycluster
```

MyId

``--cluster.my-id id

The local server's id in the cluster. Specifying *id* is mandatory on startup. Each server of the cluster must have a unique id.

Specifying the id is very important because the server id is used for determining the server's role and tasks in the cluster.

id must be a string consisting of the letters *a-z*, *A-Z* or the digits *0-9* only.

MyAddress

```
--cluster.my-address endpoint
```

The server's endpoint for cluster-internal communication. If specified, it must have the following pattern:

- tcp://ipv4-address:port - TCP/IP endpoint, using IPv4
- tcp://[ipv6-address]:port - TCP/IP endpoint, using IPv6

- ssl://ipv4-address:port - TCP/IP endpoint, using IPv4, SSL encryption
- ssl://[ipv6-address]:port - TCP/IP endpoint, using IPv6, SSL encryption

If no *endpoint* is specified, the server will look up its internal endpoint address in the agency. If no endpoint can be found in the agency for the server's id, ArangoDB will refuse to start.

Examples

```
--cluster.my-address tcp://192.168.1.1:8530
```

Username

```
--cluster.username username
```

The username used for authorization of cluster-internal requests. This username will be used to authenticate all requests and responses in cluster-internal communication, i.e. requests exchanged between coordinators and individual database servers.

This option is used for cluster-internal requests only. Regular requests to coordinators are authenticated normally using the data in the *_users* collection.

If coordinators and database servers are run with authentication turned off, (e.g. by setting the *--server.disable-authentication* option to *true*), the cluster-internal communication will also be unauthenticated.

Password

```
--cluster.password password
```

The password used for authorization of cluster-internal requests. This password will be used to authenticate all requests and responses in cluster-internal communication, i.e. requests exchanged between coordinators and individual database servers.

This option is used for cluster-internal requests only. Regular requests to coordinators are authenticated normally using the data in the `_users` collection.

If coordinators and database servers are run with authentication turned off, (e.g. by setting the *--server.disable-authentication* option to *true*), the cluster-internal communication will also be unauthenticated.

Command-Line Options for Logging

There are two different kinds of logs. Human-readable logs and machine-readable logs. The human-readable logs are used to provide an administration with information about the server. The machine-readable logs are used to provide statistics about executed requests and timings about computation steps.

General Logging Options

Logfile

```
--log.file filename
```

This option allows the user to specify the name of a file to which information is logged. By default, if no log file is specified, the standard output is used. Note that if the file named by *filename* does not exist, it will be created. If the file cannot be created (e.g. due to missing file privileges), the server will refuse to start. If the specified file already exists, output is appended to that file.

Use + to log to standard error. Use - to log to standard output. Use ″″ to disable logging to file.

Request

```
--log.requests-file filename
```

This option allows the user to specify the name of a file to which requests are logged. By default, no log file is used and requests are not logged. Note that if the file named by *filename* does not exist, it will be created. If the file cannot be created (e.g. due to missing file privileges), the server will refuse to start. If the specified file already exists, output is appended to that file.

Use + to log to standard error. Use - to log to standard output. Use ″″ to disable request logging altogether.

Severity

```
--log.severity severity
```

This parameter provides a set of standard log severities which can be used. The currently accepted *severities* are:

* exception
* technical
* functional
* development
* human
* all (human and non-human)
* non-human (exception, technical, functional, and development)

The default is all.

Syslog

```
--log.syslog arg
```

If this option is set, then in addition to output being directed to the standard output (or to a specified file, in the case that the command line log.file option was set), log output is also sent to the system logging facility. The *arg* is the system log facility to use. See syslog for further details.

The value of *arg* depends on your syslog configuration. In general it will be *user*. Fatal messages are mapped to *crit*, so if *arg* is *user*, these messages will be logged as *user.crit*. Error messages are mapped to *err*. Warnings are mapped to *warn*. Info messages are mapped to *notice*. Debug messages are mapped to *info*. Trace messages are mapped to *debug*.

Human Readable Logging

Level

```
--log.level level
```

```
--log level
```

Allows the user to choose the level of information which is logged by the server. The argument *level* is specified as a string and can be one of the values listed below. Note that, fatal errors, that is, errors which cause the server to terminate, are always logged irrespective of the log level assigned by the user. The variant *c* log.level can be used in configuration files, the variant *c* log for command line options.

fatal: Logs errors which cause the server to terminate.

Fatal errors generally indicate some inconsistency with the manner in which the server has been coded. Fatal errors may also indicate a problem with the platform on which the server is running. Fatal errors always cause the server to terminate. For example,

```
2010-09-20T07:32:12Z [4742] FATAL a http server has already been
created
```

error: Logs errors which the server has encountered.

These errors may not necessarily result in the termination of the server. For example,

```
2010-09-17T13:10:22Z [13967] ERROR strange log level 'errors'\,
going to 'warning'
```

warning: Provides information on errors encountered by the server, which are not necessarily detrimental to it's continued operation.

For example,

```
2010-09-20T08:15:26Z [5533] WARNING got corrupted HTTP request
'POS?'
```

Note: The setting the log level to warning will also result in all errors to be logged as well.

info: Logs information about the status of the server.

For example,

```
2010-09-20T07:40:38Z [4998] INFO SimpleVOC ready for business
```

Note: The setting the log level to info will also result in all errors and warnings to be logged as well.

debug: Logs all errors, all warnings and debug information.

Debug log information is generally useful to find out the state of the server in the case of an error. For example,

```
2010-09-17T13:02:53Z [13783] DEBUG opened port 7000 for any
```

Note: The setting the log level to debug will also result in all errors, warnings and server status information to be logged as well.

trace: As the name suggests, logs information which may be useful to trace problems encountered with using the server.

For example,

```
2010-09-20T08:23:12Z [5687] TRACE trying to open port 8000
```

Note: The setting the log level to trace will also result in all errors, warnings, status information, and debug information to be logged as well.

Line number

```
--log.line-number}
```

Normally, if an human readable fatal, error, warning or info message is logged, no information about the file and line number is provided. The file and line number is only logged for debug and trace message. This option can be use to always log these pieces of information.

Prefix

```
--log.prefix prefix
```

This option is used specify an prefix to logged text.

Thread

```
--log.thread
```

Whenever log output is generated, the process ID is written as part of the log information. Setting this option appends the thread id of the calling thread to the process id. For example,

```
2010-09-20T13:04:01Z [19355] INFO ready for business
```

when no thread is logged and

```
2010-09-20T13:04:17Z [19371-18446744072487317056] ready for
business
```

when this command line option is set.

Source Filter

```
--log.source-filter arg
```

For debug and trace messages, only log those messages originated from the C source file *arg*. The argument can be used multiple times.

Content Filter

```
--log.content-filter arg
```

Only log message containing the specified string *arg*.

Machine Readable Logging

Application

```
--log.application name
```

Specifies the *name* of the application which should be logged if this item of information is to be logged.

Facility

```
--log.facility name
```

Specifies the name of the server instance which should be logged if this item of information is to be logged.

Histname

```
--log.hostname name
```

Specifies the *name* of the operating environment (the "hostname") which should be logged if this item of information is to be logged. Note that there is no default hostname.

Command-Line Options for Communication

Scheduler threads

```
--scheduler.threads arg
```

An integer argument which sets the number of threads to use in the IO scheduler. The default is 1.

Scheduler maximal queue size

```
--server.disable-authentication-unix-sockets value
```

Setting *value* to true will turn off authentication on the server side for requests coming in via UNIX domain sockets. With this flag enabled, clients located on the same host as the ArangoDB server can use UNIX domain sockets to connect to the server without authentication. Requests coming in by other means (e.g. TCP/IP) are not affected by this option.

The default value is *false*.

Note: this option is only available on platforms that support UNIX domain sockets.

Scheduler backend

```
--scheduler.backend arg
```

The I/O method used by the event handler. The default (if this option is not specified) is to try all recommended backends. This is platform specific. See libev for further details and the meaning of select, poll and epoll.

Io backends

```
--show-io-backends
```

If this option is specified, then the server will list available backends and exit. This option is useful only when used in conjunction with the option scheduler.backend. An integer is returned (which is platform dependent) which indicates available backends on your platform. See libev for further details and for the meaning of the integer returned. This describes the allowed integers for *scheduler.backend*, see here for details.

Command-Line Options for Random Numbers

```
--random.generator arg
```

The argument is an integer (1,2,3 or 4) which sets the manner in which random numbers are generated. The default method (3) is to use the a non-blocking random (or pseudorandom) number generator supplied by the operating system.

Specifying an argument of 2, uses a blocking random (or pseudorandom) number generator. Specifying an argument 1 sets a pseudorandom number generator using an implication of the Mersenne Twister MT19937 algorithm. Algorithm 4 is a combination of the blocking random number generator and the Mersenne Twister.

Authentication and Authorization

Authentication and Authorization

ArangoDB only provides a very simple authentication interface and no authorization. We plan to add authorization features in later releases, which will allow the administrator to restrict access to collections and queries to certain users, given them either read or write access.

Currently, you can only secure the access to ArangoDB in an all-or-nothing fashion. The collection _users contains all users and a salted SHA256 hash of their passwords. A user can be active or inactive. A typical document of this collection is

```
{
  "_id" : "_users/1172449",
  "_rev" : "1172449",
  "_key" : "1172449",
  "active" : true,
  "changePassword" : false,
  "user" : "root",
  "password" :
"$1$bd5458a8$8b23e2e1a762f75001ab182235b8ab1b8665bc572b0734a042a501b3

}
```

Command-Line Options for the Authentication and Authorization

`--server.disable-authentication`

Setting value to true will turn off authentication on the server side so all clients can execute any action without authorization and privilege checks.

The default value is *false*.

Introduction to User Management

ArangoDB provides basic functionality to add, modify and remove database users programmatically. The following functionality is provided by the *users* module and can be used from inside arangosh and arangod.

Note: This functionality is not available from within the web interface.

Save

`users.save(user, passwd, active, extra, changePassword)`

This will create a new ArangoDB user. The username must be specified in *user* and must not be empty.

The password must be given as a string, too, but can be left empty if required.

If the *active* attribute is not specified, it defaults to *true*. The *extra* attribute can be used to save custom data with the user.

If the *changePassword* attribute is not specified, it defaults to *false*. The *changePassword* attribute can be used to indicate that the user must change has password before logging in.

This method will fail if either the username or the passwords are not specified or given in a wrong format, or there already exists a user with the specified name.

The new user account can only be used after the server is either restarted or the server authentication cache is reloaded.

Note: this function will not work from within the web interface

Examples

```
arangosh> require("org/arangodb/users").save("my-user", "my-secret-password");
```

Document

`users.document(user)`

Fetches an existing ArangoDB user from the database.

The username must be specified in *user*.

This method will fail if the user cannot be found in the database.

Note: this function will not work from within the web interface

Replace

`users.replace(user, passwd, active, extra, changePassword)`

This will look up an existing ArangoDB user and replace its user data.

The username must be specified in *user*, and a user with the specified name must already exist in the database.

The password must be given as a string, too, but can be left empty if required.

If the *active* attribute is not specified, it defaults to *true*. The *extra* attribute can be used to save custom

data with the user.

If the *changePassword* attribute is not specified, it defaults to *false*. The *changePassword* attribute can be used to indicate that the user must change has password before logging in.

This method will fail if either the username or the passwords are not specified or given in a wrong format, or if the specified user cannot be found in the database.

Note: this function will not work from within the web interface

Examples

```
arangosh> require("org/arangodb/users").replace("my-user", "my-
changed-password");
```

Update

```
users.update(user, passwd, active, extra, changePassword)
```

This will update an existing ArangoDB user with a new password and other data.

The username must be specified in *user* and the user must already exist in the database.

The password must be given as a string, too, but can be left empty if required.

If the *active* attribute is not specified, the current value saved for the user will not be changed. The same is true for the *extra* and the *changePassword* attribute.

This method will fail if either the username or the passwords are not specified or given in a wrong format, or if the specified user cannot be found in the database.

Note: this function will not work from within the web interface

Examples

```
arangosh> require("org/arangodb/users").update("my-user", "my-
secret-password");
```

Remove

```
users.remove(user)
```

Removes an existing ArangoDB user from the database.

The username must be specified in *User* and the specified user must exist in the database.

This method will fail if the user cannot be found in the database.

Note: this function will not work from within the web interface

Examples

```
arangosh> require("org/arangodb/users").remove("my-user");
```

Reload

```
users.reload()
```

Reloads the user authentication data on the server

All user authentication data is loaded by the server once on startup only and is cached after that. When users get added or deleted, a cache flush is required, and this can be performed by called this method.

Note: this function will not work from within the web interface

isValid

```
users.isvalid(user, password)
```

Checks whether the given combination of username and password is valid. The function will return a boolean value if the combination of username and password is valid.

Each call to this function is penalized by the server sleeping a random amount of time.

Note: this function will not work from within the web interface

all()

```
users.all()
```

Fetches all existing ArangoDB users from the database.

Emergency Console

In Case Of Disaster

The following command starts a emergency console.

Note: Never start the emergency console for a database which also has a server attached to it. In general the ArangoDB shell is what you want.

```
> ./arangod --console --log error /tmp/vocbase
ArangoDB shell [V8 version 3.9.4, DB version 1.x.y]

arango> 1 + 2;
3

arango> db.geo.count();
703
```

The emergency console disables the HTTP interface of the server and opens a JavaScript console on standard output instead. This allows you to debug and examine collections and documents without interference from the outside. In most respects the emergency console behaves like the normal ArangoDB shell - but with exclusive access and no client/server communication.

However, it is very likely that you never need the emergency console unless you are an ArangoDB developer.

Arangoimp

This manual describes the ArangoDB importer *arangoimp*, which can be used for bulk imports.

The most convenient method to import a lot of data into ArangoDB is to use the *arangoimp* command-line tool. It allows you to import data records from a file into an existing database collection.

It is possible to import document keys with the documents using the *_key* attribute. When importing into an edge collection, it is mandatory that all imported documents have the *_from* and *_to* attributes, and that they contain valid references.

Let's assume for the following examples you want to import user records into an existing collection named "users" on the server.

Importing Data into an ArangoDB Database

Importing JSON-encoded Data

Let's further assume the import at hand is encoded in JSON. We'll be using these example user records to import:

```
{ "name" : { "first" : "John", "last" : "Connor" }, "active" :
true, "age" : 25, "likes" : [ "swimming"] }
{ "name" : { "first" : "Jim", "last" : "O'Brady" }, "age" : 19,
"likes" : [ "hiking", "singing" ] }
{ "name" : { "first" : "Lisa", "last" : "Jones" }, "dob" : "1981-
04-09", "likes" : [ "running" ] }
```

To import these records, all you need to do is to put them into a file (with one line for each record to import) and run the following command:

```
unix> arangoimp --file "data.json" --type json --collection "users"
```

This will transfer the data to the server, import the records, and print a status summary. To show the intermediate progress during the import process, the option --*progress* can be added. This option will show the percentage of the input file that has been sent to the server. This will only be useful for big import files.

```
unix> arangoimp --file "data.json" --type json --collection "users"
--progress true
```

By default, the endpoint *tcp://127.0.0.1:8529* will be used. If you want to specify a different endpoint, you can use the --*server.endpoint* option. You probably want to specify a database user and password as well. You can do so by using the options --*server.username* and --*server.password*. If you do not specify a password, you will be prompted for one.

```
unix> arangoimp --server.endpoint tcp://127.0.0.1:8529 --
server.username root --file "data.json" --type json --collection
"users"
```

Note that the collection (*users* in this case) must already exist or the import will fail. If you want to create a new collection with the import data, you need to specify the --*create-collection* option. Note that it is only possible to create a document collection using the --*create-collection* flag, and no edge collections.

```
unix> arangoimp --file "data.json" --type json --collection "users"
--create-collection true
```

When importing data into an existing collection it is often convenient to first remove all data from the collection and then start the import. This can be achieved by passing the --*overwrite* parameter to *arangoimp*. If it is set to *true*, any existing data in the collection will be removed prior to the import. Note that any existing index definitions for the collection will be preserved even if --*overwrite* is set to true.

```
unix> arangoimp --file "data.json" --type json --collection "users"
--overwrite true
```

As the import file already contains the data in JSON format, attribute names and data types are fully preserved. As can be seen in the example data, there is no need for all data records to have the same attribute names or types. Records can be in-homogenous.

Please note that by default, *arangoimp* will import data into the specified collection in the default database (*_system*). To specify a different database, use the *--server.database* option when invoking *arangoimp*.

An *arangoimp* import run will print out the final results on the command line. By default, it shows the number of documents created, the number of errors that occurred on the server side, and the total number of input file lines/documents that it processed. Additionally, *arangoimp* will print out details about errors that happened on the server-side (if any).

Examples

```
created:            2
errors:             0
total:              2
```

Note: *arangoimp* supports two formats when importing JSON data from a file. The first format requires the input file to contain one JSON document in each line, e.g.

```
{ "_key": "one", "value": 1 }
{ "_key": "two", "value": 2 }
{ "_key": "foo", "value": "bar" }
...
```

The above format can be imported sequentially by *arangoimp*. It will read data from the input file in chunks and send it in batches to the server. Each batch will be about as big as specified in the command-line parameter *--batch-size*.

An alternative is to put one big JSON document into the input file like this:

```
[
  { "_key": "one", "value": 1 },
  { "_key": "two", "value": 2 },
  { "_key": "foo", "value": "bar" },
  ...
]
```

This format allows line breaks within the input file as required. The downside is that the whole input file will need to be read by *arangoimp* before it can send the first batch. This might be a problem if the input file is big. By default, *arangoimp* will allow importing such files up to a size of about 16 MB.

If you want to allow your *arangoimp* instance to use more memory, you may want to increase the maximum file size by specifying the command-line option *--batch-size*. For example, to set the batch size to 32 MB, use the following command:

```
unix> arangoimp --file "data.json" --type json --collection "users"
--batch-size 33554432
```

Please also note that you may need to increase the value of *--batch-size* if a single document inside the input file is bigger than the value of *--batch-size*.

!SUBSECTION Importing CSV Data

arangoimp also offers the possibility to import data from CSV files. This comes handy when the data at hand is in CSV format already and you don't want to spend time converting them to JSON for the import.

To import data from a CSV file, make sure your file contains the attribute names in the first row. All the following lines in the file will be interpreted as data records and will be imported.

The CSV import requires the data to have a homogeneous structure. All records must have exactly the same amount of columns as there are headers.

The cell values can have different data types though. If a cell does not have any value, it can be left empty in the file. These values will not be imported so the attributes will not "be there" in document created. Values enclosed in quotes will be imported as strings, so to import numeric values, boolean values or the null value, don't enclose the value into the quotes in your file.

We'll be using the following import for the CSV import:

```
"first","name","age","active","dob"
"John","Connor",25,true,
"Jim","O'Brady",19,,
"Lisa","Jones",,,"1981-04-09"
```

The command line to execute the import then is:

```
unix> arangoimp --file "data.csv" --type csv --collection "users"
```

Note that the quote and separator characters can be adjusted via the *--quote* and *--separator* arguments when invoking *arangoimp*. The importer supports Windows (CRLF) and Unix (LF) line breaks.

!SUBSECTION Importing TSV Data

You may also import tab-separated values (TSV) from a file. This format is very simple: every line in the file represents a data record. There is no quoting or escaping. That also means that the separator character (which defaults to the tabstop symbol) must not be used anywhere in the actual data.

As with CSV, the first line in the TSV file must contain the attribute names, and all lines must have an identical number of values.

If a different separator character or string should be used, it can be specified with the *--separator* argument.

An example command line to execute the TSV import is:

```
unix> arangoimp --file "data.tsv" --type tsv --collection "users"
```

!SUBSECTION Importing into an Edge Collection

arangoimp can also be used to import data into an existing edge collection. The import data must, for each edge to import, contain at least the *_from* and *_to* attributes. These indicate which other two documents the edge should connect. It is necessary that these attributes are set for all records, and point to valid document ids in existing collections.

Examples

```
{ "_from" : "users/1234", "_to" : "users/4321", "desc" : "1234 is
connected to 4321" }
```

Note: The edge collection must already exist when the import is started. Using the *--create-collection* flag will not work because arangoimp will always try to create a regular document collection if the target collection does not exist.

!SUBSECTION Attribute Naming and Special Attributes

Attributes whose names start with an underscore are treated in a special way by ArangoDB:

- the optional *_key* attribute contains the document's key. If specified, the value must be formally valid (e.g. must be a string and conform to the naming conventions for @ref DocumentKeys). Additionally, the key value must be unique within the collection the import is run for.
- *_from*: when importing into an edge collection, this attribute contains the id of one of the documents connected by the edge. The value of *_from* must be a syntactically valid document id and the referred collection must exist.
- *_to*: when importing into an edge collection, this attribute contains the id of the other document connected by the edge. The value of *_to* must be a syntactically valid document id and the referred collection must exist.
- *_rev*: this attribute contains the revision number of a document. However, the revision numbers are managed by ArangoDB and cannot be specified on import. Thus any value in this attribute is ignored on import.
- all other attributes starting with an underscore are discarded on import without any warnings.

If you import values into *_key*, you should make sure they are valid and unique.

When importing data into an edge collection, you should make sure that all import documents can *_from* and *_to* and that their values point to existing documents.

Finally you should make sure that all other attributes in the import file do not start with an underscore - otherwise they might be discarded.

Dumping Data from an ArangoDB database

To dump data from an ArangoDB server instance, you will need to invoke *arangodump*. It can be invoked by executing the following command:

```
unix> arangodump --output-directory "dump"
```

This will connect to an ArangoDB server and dump all non-system collections from the default database (*_system*) into an output directory named *dump*. Invoking *arangodump* will fail if the output directory already exists. This is an intentional security measure to prevent you from accidentally overwriting already dumped data. If you are positive that you want to overwrite data in the output directory, you can use the parameter *--overwrite true* to confirm this:

```
unix> arangodump --output-directory "dump" --overwrite true
```

arangodump will by default connect to the *_system* database using the default endpoint. If you want to connect to a different database or a different endpoint, or use authentication, you can use the following command-line options:

- *--server.database* : name of the database to connect to
- *--server.endpoint* : endpoint to connect to
- *--server.username* : username
- *--server.password* : password to use (omit this and you'll be prompted for the password)
- *--server.disable-authentication* : whether or not to use authentication

Here's an example of dumping data from a non-standard endpoint, using a dedicated database name:

```
unix> arangodump --server.endpoint tcp://192.168.173.13:8531 --
server.username backup --server.database mydb --output-directory
"dump"
```

When finished, *arangodump* will print out a summary line with some aggregate statistics about what it did, e.g.:

```
Processed 43 collection(s), wrote 408173500 byte(s) into datafiles,
sent 88 batch(es)
```

By default, *arangodump* will dump both structural information and documents from all non-system collections. To adjust this, there are the following command-line arguments:

- *--dump-data* : set to *true* to include documents in the dump. Set to *false* to exclude documents. The default value is *true*.
- *--include-system-collections* : whether or not to include system collections in the dump. The default value is *false*.

For example, to only dump structural information of all collections (including system collections), use:

```
unix> arangodump --dump-data false --include-system-collections
true --output-directory "dump"
```

To restrict the dump to just specific collections, there is is the *--collection* option. It can be specified multiple times if required:

```
unix> arangodump --collection myusers --collection myvalues --
output-directory "dump"
```

Structural information for a collection will be saved in files with name pattern *.structure.json*. Each structure file will contains a JSON object with these attributes:

- *parameters*: contains the collection properties
- *indexes*: contains the collection indexes

Document data for a collection will be saved in files with name pattern *.data.json*. Each line in a data file is a document insertion/update or deletion marker, alongside with some meta data.

Starting with Version 2.1 of ArangoDB, the *arangodump* tool also supports sharding. Simply point it to one of the coordinators and it will behave exactly as described above, working on sharded collections in the cluster.

However, as opposed to the single instance situation, this operation does not lock the data in the cluster and can therefore not guarantee to dump a consistent snapshot if writing operations happen during the dump operation! That is, it is recommended not to perform any data modifying operations on the cluster whilst *arangodump* is running.

As above, the output will be one structure description file and one data file per sharded collection. Note that the data in the data file is sorted first by shards and within each shard by ascending timestamp. The structural information of the collection contains the number of shards and the shard keys.

Arangorestore

To reload data from a dump previously created with [arangodump](), ArangoDB provides the *arangorestore* tool.

Reloading Data into an ArangoDB database

Invoking arangorestore

arangorestore can be invoked from the command-line as follows:

```
unix> arangorestore --input-directory "dump"
```

This will connect to an ArangoDB server and reload structural information and documents found in the input directory *dump*. Please note that the input directory must have been created by running *arangodump* before.

arangorestore will by default connect to the *_system* database using the default endpoint. If you want to connect to a different database or a different endpoint, or use authentication, you can use the following command-line options:

- *--server.database* : name of the database to connect to
- *--server.endpoint* : endpoint to connect to
- *--server.username* : username
- *--server.password* : password to use (omit this and you'll be prompted for the password)
- *--server.disable-authentication* : whether or not to use authentication

Here's an example of reloading data to a non-standard endpoint, using a dedicated database name:

```
unix> arangorestore --server.endpoint tcp://192.168.173.13:8531 --
server.username backup --server.database mydb --input-directory
"dump"
```

arangorestore will print out its progress while running, and will end with a line showing some aggregate statistics:

```
Processed 2 collection(s), read 2256 byte(s) from datafiles, sent 2
batch(es)
```

By default, *arangorestore* will re-create all non-system collections found in the input directory and load data into them. If the target database already contains collections which are also present in the input directory, the existing collections in the database will be dropped and re-created with the data found in the input directory.

The following parameters are available to adjust this behavior:

- *--create-collection* : set to *true* to create collections in the target database. If the target database already contains a collection with the same name, it will be dropped first and then re-created with the properties found in the input directory. Set to *false* to keep existing collections in the target database. If set to *false* and *arangorestore* encounters a collection that is present in both the target database and the input directory, it will abort. The default value is *true*.
- *--import-data* : set to *true* to load document data into the collections in the target database. Set to *false* to not load any document data. The default value is *true*.
- *--include-system-collections* : whether or not to include system collections when re-creating collections or reloading data. The default value is *false*.

For example, to (re-)create all non-system collections and load document data into them, use:

```
unix> arangorestore --create-collection true --import-data true --
input-directory "dump"
```

This will drop potentially existing collections in the target database that are also present in the input directory.

To include system collections too, use *--include-system-collections true*:

```
unix> arangorestore --create-collection true --import-data true --
include-system-collections true --input-directory "dump"
```

To (re-)create all non-system collections without loading document data, use:

```
unix> arangorestore --create-collection true --import-data false --
input-directory "dump"
```

This will also drop existing collections in the target database that are also present in the input directory.

To just load document data into all non-system collections, use:

```
unix> arangorestore --create-collection false --import-data true --
input-directory "dump"
```

To restrict reloading to just specific collections, there is is the *--collection* option. It can be specified multiple times if required:

```
unix> arangorestore --collection myusers --collection myvalues --
input-directory "dump"
```

Collections will be processed by in alphabetical order by *arangorestore*, with all document collections being processed before all edge collections. This is to ensure that reloading data into edge collections will have the document collections linked in edges (*_from* and *_to* attributes) loaded.

Restoring Revision Ids and Collection Ids

arangorestore will reload document and edges data with the exact same *_key*, *_from* and *_to* values found in the input directory. However, when loading document data, it will assign its own values for the *_rev* attribute of the reloaded documents. Though this difference is intentional (normally, every server should create its own *_rev* values) there might be situations when it is required to re-use the exact same *_rev* values for the reloaded data. This can be achieved by setting the *--recycle-ids* parameter to *true*:

```
unix> arangorestore --collection myusers --collection myvalues --
recycle-ids true --input-directory "dump"
```

Note that setting *--recycle-ids* to *true* will also cause collections to be (re-)created in the target database with the exact same collection id as in the input directory. Any potentially existing collection in the target database with the same collection id will then be dropped.

Setting *--recycle-ids* to *false* or omitting it will only use the collection name from the input directory and allow the target database to create the collection with a different id (though with the same name) than in the input directory.

Reloading Data into a different Collection

With some creativity you can use *arangodump* and *arangorestore* to transfer data from one collection into

another (either on the same server or not). For example, to copy data from a collection *myvalues* in database *mydb* into a collection *mycopyvalues* in database *mycopy*, you can start with the following command:

```
unix> arangodump --collection myvalues --server.database mydb --
output-directory "dump"
```

This will create two files, *myvalues.structure.json* and *myvalues.data.json*, in the output directory. To load data from the datafile into an existing collection *mycopyvalues* in database *mycopy*, rename the files to *mycopyvalues.structure.json* and *mycopyvalues.data.json*. After that, run the following command:

```
unix> arangorestore --collection mycopyvalues --server.database
mycopy --input-directory "dump"
```

Using arangorestore with sharding

As of Version 2.1 the *arangorestore* tool supports sharding. Simply point it to one of the coordinators in your cluster and it will work as usual but on sharded collections in the cluster.

If *arangorestore* is asked to drop and re-create a collection, it will use the same number of shards and the same shard keys as when the collection was dumped. The distribution of the shards to the servers will also be the same as at the time of the dump. This means in particular that DBservers with the same IDs as before must be present in the cluster at time of the restore.

If a collection was dumped from a single instance, one can manually add the structural description for the shard keys and the number and distribution of the shards and then the restore into a cluster will work.

If you restore a collection that was dumped from a cluster into a single ArangoDB instance, the number of shards and the shard keys will silently be ignored.

Note that in a cluster, every newly created collection will have a new ID, it is not possible to reuse the ID from the originally dumped collection. This is for safety reasons to ensure consistency of IDs.

Http Interface

Following you have ArangoDB's Http Interface for Documents, Databases, Edges and more.

Also you have some examples for every API action.

HTTP Interface for Databases

Address of a Database

Any operation triggered via ArangoDB's HTTP REST API is executed in the context of exactly one database. To explicitly specify the database in a request, the request URI must contain the database name in front of the actual path:

```
http://localhost:8529/_db/mydb/...
```

where ... is the actual path to the accessed resource. In the example, the resource will be accessed in the context of the database *mydb*. Actual URLs in the context of *mydb* could look like this:

```
http://localhost:8529/_db/mydb/_api/version
http://localhost:8529/_db/mydb/_api/document/test/12345
http://localhost:8529/_db/mydb/myapp/get
```

Database-to-Endpoint Mapping

If a database name is present in the URI as above, ArangoDB will consult the database-to-endpoint mapping for the current endpoint, and validate if access to the database is allowed on the endpoint. If the endpoint is not restricted to a list of databases, ArangoDB will continue with the regular authentication procedure. If the endpoint is restricted to a list of specified databases, ArangoDB will check if the requested database is in the list. If not, the request will be turned down instantly. If yes, then ArangoDB will continue with the regular authentication procedure.

If the request URI was *http:// localhost:8529/_db/mydb/...*, then the request to *mydb* will be allowed (or disallowed) in the following situations:

```
Endpoint-to-database mapping              Access to *mydb* allowed?
----------------------------              -------------------------
[ ]                                       yes
[ "_system" ]                             no
[ "_system", "mydb" ]                     yes
[ "mydb" ]                                yes
[ "mydb", "_system" ]                     yes
[ "test1", "test2" ]                      no
```

In case no database name is specified in the request URI, ArangoDB will derive the database name from the endpoint-to-database mapping (see @ref NewFeatures14Endpoints} of the endpoint the connection was coming in on.

If the endpoint is not restricted to a list of databases, ArangoDB will assume the *_system* database. If the endpoint is restricted to one or multiple databases, ArangoDB will assume the first name from the list.

Following is an overview of which database name will be assumed for different endpoint-to-database mappings in case no database name is specified in the URI:

```
Endpoint-to-database mapping              Database
----------------------------              --------
[ ]                                       _system
[ "_system" ]                             _system
[ "_system", "mydb" ]                     _system
[ "mydb" ]                                mydb
[ "mydb", "_system" ]                     mydb
```

Database Management

This is an introduction to ArangoDB's Http interface for managing databases.

The HTTP interface for databases provides operations to create and drop individual databases. These are mapped to the standard HTTP methods *POST* and *DELETE*. There is also the *GET* method to retrieve a list of existing databases.

Please note that all database management operations can only be accessed via the default database (*_system*) and none of the other databases.

Managing Databases using HTTP

retrieves information about the current database

Information of the database

```
GET /_api/database/current
```

Retrieves information about the current database

The response is a JSON object with the following attributes:

- *name*: the name of the current database

- *id*: the id of the current database

- *path*: the filesystem path of the current database

- *isSystem*: whether or not the current database is the *_system* database

Return Codes

- *200*: is returned if the information was retrieved successfully.

- *400*: is returned if the request is invalid.

- *404*: is returned if the database could not be found.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/database/current
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "name" : "_system",
    "id" : "90805",
    "path" : "/tmp/vocdir.99549/databases/database-90805",
    "isSystem" : true
  },
  "error" : false,
  "code" : 200
}
```

retrieves a list of all databases the current user can access

List of accessible databases

```
GET /_api/database/user
```

Retrieves the list of all databases the current user can access without specifying a different username or password.

Return Codes

- *200*: is returned if the list of database was compiled successfully.

- *400*: is returned if the request is invalid.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/database/user
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : [
    "_system"
  ],
  "error" : false,
  "code" : 200
}
```

retrieves a list of all existing databases

List of databases

GET /_api/database

Retrieves the list of all existing databases

Note: retrieving the list of databases is only possible from within the *_system* database.

Return Codes

- *200*: is returned if the list of database was compiled successfully.

- *400*: is returned if the request is invalid.

- *403*: is returned if the request was not executed in the *_system* database.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/database
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : [
    "_system"
  ],
  "error" : false,
  "code" : 200
}
```

creates a new database

Create database

```
POST /_api/database
```

- *body*: the body with the name of the database.

Creates a new database

The request body must be a JSON object with the attribute *name*. *name* must contain a valid @ref DatabaseNames "database name".

The request body can optionally contain an attribute *users*, which then must be a list of user objects to initially create for the new database. Each user object can contain the following attributes:

- *username*: the user name as a string. This attribute is mandatory.

- *passwd*: the user password as a string. If not specified, then it defaults to the empty string.

- *active*: a boolean flag indicating whether the user accout should be actived or not. The default value is *true*.

- *extra*: an optional JSON object with extra user information. The data contained in *extra* will be stored for the user but not be interpreted further by ArangoDB.

If *users* is not specified or does not contain any users, a default user *root* will be created with an empty string password. This ensures that the new database will be accessible after it is created.

The response is a JSON object with the attribute *result* set to *true*.

Note: creating a new database is only possible from within the *_system* database.

Return Codes

- *201*: is returned if the database was created successfully.

- *400*: is returned if the request parameters are invalid or if a database with the specified name already exists.

- *403*: is returned if the request was not executed in the *_system* database.

- *409*: is returned if a database with the specified name already exists.

Examples

Creating a database named *example*.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/database
{"name":"example"}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 201
}
```

Creating a database named *mydb* with two users.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/database
{"name":"mydb","users":
[{"username":"admin","passwd":"secret","active":true},
{"username":"tester","passwd":"test001","active":false}]}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 201
}
```

drop an existing database

Drop database

```
DELETE /_api/database/database-name
```

URL Parameters

- *database-name*: The name of the database

Deletes the database along with all data stored in it.

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

Return Codes

- *200*: is returned if the database was dropped successfully.

- *400*: is returned if the request is malformed.

- *403*: is returned if the request was not executed in the *_system* database.

- *404*: is returned if the database could not be found.

Examples

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/database/example
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
```

Notes on Databases

Please keep in mind that each database contains its own system collections, which need to set up when a database is created. This will make the creation of a database take a while. Replication is configured on a per-database level, meaning that any replication logging or applying for the a new database must be configured explicitly after a new database has been created. Foxx applications are also available only in the context of the database they have been installed in. A new database will only provide access to the system applications shipped with ArangoDB (that is the web interface at the moment) and no other Foxx applications until they are explicitly installed for the particular database.

Database

ArangoDB can handle multiple databases in the same server instance. Databases can be used to logically group and separate data. An ArangoDB database consists of collections and dedicated database-specific worker processes. A database contains its own collections (which cannot be accessed from other databases), Foxx applications and replication loggers and appliers. Each ArangoDB database contains its own system collections (e.g. _users, _replication, ...).

There will always be at least one database in ArangoDB. This is the default database named _system. This database cannot be dropped and provides special operations for creating, dropping and enumerating databases. Users can create additional databases and give them unique names to access them later. Database management operations cannot be initiated from out of user-defined databases.

When ArangoDB is accessed via its HTTP REST API, the database name is read from the first part of the request URI path (e.g. /_db/_system/...). If the request URI does not contain a database name, the database name is automatically determined by the algorithm described in Database-to-Endpoint Mapping .

Database Name

A single ArangoDB instance can handle multiple databases in parallel. When multiple databases are used, each database must be given an unique name. This name is used to uniquely identify a database. The default database in ArangoDB is named *system. The database name is a string consisting of only letters, digits and the* (underscore) and - (dash) characters. User-defined database names must always start with a letter. Database names are case-sensitive.

Database Organization

A single ArangoDB instance can handle multiple databases in parallel. By default, there will be at least one database which is named _system. Databases are physically stored in separate sub-directories underneath the database directory, which itself resides in the instance's data directory.

Each database has its own sub-directory, named database-. The database directory contains sub-directories for the collections of the database, and a file named parameter.json. This file contains the database id and name.

In an example ArangoDB instance which has two databases, the filesystem layout could look like this:

```
data/                     # the instance's data directory
  databases/              # sub-directory containing all databases'
data
    database-<id>/        # sub-directory for a single database
      parameter.json      # file containing database id and name
      collection-<id>/    # directory containing data about a
collection
    database-<id>/        # sub-directory for another database
      parameter.json      # file containing database id and name
      collection-<id>/    # directory containing data about a
collection
```

```
        collection-<id>/    # directory containing data about a
collection
```

Foxx applications are also organized in database-specific directories inside the application path. The filesystem layout could look like this:

```
apps/                       # the instance's application directory
   system/                  # system applications (can be ignored)
   databases/               # sub-directory containing database-
specific applications
     <database-name>/       # sub-directory for a single database
       <app-name>           # sub-directory for a single application
       <app-name>           # sub-directory for a single application
     <database-name>/       # sub-directory for another database
       <app-name>           # sub-directory for a single application
 `
```

HTTP Interface for Transactions

Transactions

ArangoDB's transactions are executed on the server. Transactions can be initiated by clients by sending the transaction description for execution to the server.

Transactions in ArangoDB do not offer separate *BEGIN, COMMIT* and *ROLLBACK* operations as they are available in many other database products. Instead, ArangoDB transactions are described by a Javascript function, and the code inside the Javascript function will then be executed transactionally. At the end of the function, the transaction is automatically committed, and all changes done by the transaction will be persisted. If an exception is thrown during transaction execution, all operations performed in the transaction are rolled back.

For a more detailed description of how transactions work in ArangoDB please refer to [Transactions](#).

execute a server-side transaction

Execute transaction

POST /_api/transaction

- *body*: Contains the *collections* and *action*.

The transaction description must be passed in the body of the POST request.

The following attributes must be specified inside the JSON object:

- *collections*: contains the list of collections to be used in the transaction (mandatory). *collections* must be a JSON array that can have the optional sub-attributes *read* and *write*. *read* and *write* must each be either lists of collections names or strings with a single collection name.

- *action*: the actual transaction operations to be executed, in the form of stringified Javascript code. The code will be executed on server side, with late binding. It is thus critical that the code specified in *action* properly sets up all the variables it needs. If the code specified in *action* ends with a return statement, the value returned will also be returned by the REST API in the *result* attribute if the transaction committed successfully.

The following optional attributes may also be specified in the request:

- *waitForSync*: an optional boolean flag that, if set, will force the transaction to write all data to disk before returning.

- *lockTimeout*: an optional numeric value that can be used to set a timeout for waiting on collection locks. If not specified, a default value will be used. Setting *lockTimeout* to *0* will make ArangoDB not time out waiting for a lock.

- *params*: optional arguments passed to *action*.

If the transaction is fully executed and committed on the server, *HTTP 200* will be returned. Additionally, the return value of the code defined in *action* will be returned in the *result* attribute.

For successfully committed transactions, the returned JSON object has the following properties:

- *error*: boolean flag to indicate if an error occurred (*false* in this case)

- *code*: the HTTP status code

- *result*: the return value of the transaction

If the transaction specification is either missing or malformed, the server will respond with *HTTP 400*.

The body of the response will then contain a JSON object with additional error details. The object has the following attributes:

- *error*: boolean flag to indicate that an error occurred (*true* in this case)

- *code*: the HTTP status code

- *errorNum*: the server error number

- *errorMessage*: a descriptive error message

If a transaction fails to commit, either by an exception thrown in the *action* code, or by an internal error, the server will respond with an error. Any other errors will be returned with any of the return codes *HTTP 400*, *HTTP 409*, or *HTTP 500*.

Return Codes

- *200*: If the transaction is fully executed and committed on the server, *HTTP 200* will be returned.

- *400*: If the transaction specification is either missing or malformed, the server will respond with *HTTP 400*.

- *404*: If the transaction specification contains an unknown collection, the server will respond with *HTTP 404*.

- *500*: Exceptions thrown by users will make the server respond with a return code of *HTTP 500*

Examples

Executing a transaction on a single collection:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/transaction
{
  "collections" : {
    "write" : "products"
  },
  "action" : "function () { var db = require('internal').db;
db.products.save({});  return db.products.count(); }"
}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : 1,
  "error" : false,
  "code" : 200
}
```

Executing a transaction using multiple collections:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/transaction
{
  "collections" : {
    "write" : [
```

```
      "products",
      "materials"
    ]
  },
  "action" : "function () { var db = require('internal').db;
db.products.save({}); db.materials.save({}); return 'worked!'; }"
}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : "worked!",
  "error" : false,
  "code" : 200
}
```

Aborting a transaction due to an internal error:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/transaction
{
  "collections" : {
    "write" : "products"
  },
  "action" : "function () { var db = require('internal').db;
db.products.save({ _key: 'abc'}); db.products.save({ _key: 'abc'});
}"
}

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 1210,
  "errorMessage" : "unique constraint violated"
}
```

Aborting a transaction by explicitly throwing an exception:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/transaction
{
  "collections" : {
    "read" : "products"
  },
  "action" : "function () { throw 'doh!'; }"
}

HTTP/1.1 500 Internal Error
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 500,
```

```
    "errorNum" : 500,
    "errorMessage" : "doh!"
}
```

Referring to a non-existing collection:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/transaction
{
  "collections" : {
    "read" : "products"
  },
  "action" : "function () { return true; }"
}

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 404,
  "errorNum" : 1203,
  "errorMessage" : "collection not found"
}
```

General Graphs

This chapter describes the http interface for the multi-collection graph module. It allows you to define a graph that is spread across several edge and document collections. This allows you to structure your models in line with your domain and group them logically in collections and giving you the power to query them in the same graph queries. There is no need to include the referenced collections within the query, this module will handle it for you.

Manage your graphs

The graph module provides functions dealing with graph structures.

First Steps with Graphs

A Graph consists of *vertices* and *edges*. Edges are stored as documents in *edge collections*. A vertex can be a document of a *document collection* or of an edge collection (so edges can be used as vertices). Which collections are used within a graph is defined via *edge definitions*. A graph can contain more than one edge definition, at least one is needed.

Lists all graphs known to the graph module.

List all graphs

GET /_api/gharial

Lists all graph names stored in this database.ssss

Return Codes

- *200*: Returned if the module is available and the graphs could be listed.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/gharial
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "graphs" : [
    {
      "_id" : "_graphs/social",
      "_rev" : "1106076341",
      "_key" : "social",
      "edgeDefinitions" : [
        {
          "collection" : "relation",
          "from" : [
            "female",
            "male"
          ],
          "to" : [
            "female",
            "male"
          ]
        }
      ],
      "orphanCollections" : [ ]
    },
    {
      "_id" : "_graphs/routeplanner",
      "_rev" : "1110401717",
```

```
      "_key" : "routeplanner",
      "orphanCollections" : [ ],
      "edgeDefinitions" : [
        {
          "collection" : "germanHighway",
          "from" : [
            "germanCity"
          ],
          "to" : [
            "germanCity"
          ]
        },
        {
          "collection" : "frenchHighway",
          "from" : [
            "frenchCity"
          ],
          "to" : [
            "frenchCity"
          ]
        },
        {
          "collection" : "internationalHighway",
          "from" : [
            "frenchCity",
            "germanCity"
          ],
          "to" : [
            "frenchCity",
            "germanCity"
          ]
        }
      ]
    }
  ]
}
```

Create a new graph in the graph module.

Create a graph

```
POST /_api/gharial
```

The creation of a graph requires the name of the graph and a definition of its edges.

Body Parameters

- *name*: Name of the graph.

- *edgeDefinitions*: A list of definitions for the edges, see <u>edge definitions</u>.

- *orphanCollections*: A list of additional vertex collections.

Return Codes

- *201*: Returned if the graph could be listed created. The body contains the graph configuration that has been stored.

- *400*: Returned if there is a conflict storing the graph. This can occur either if a graph with this name is already stored, or if there is one edge definition with a the same edge collection but a

different signature used in any other graph.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/gharial
{
  "name" : "myGraph",
  "edgeDefinitions" : [
    {
      "collection" : "edges",
      "from" : [
        "startVertices"
      ],
      "to" : [
        "endVertices"
      ]
    }
  ]
}

HTTP/1.1 201 Created
content-type: application/json

{
  "error" : false,
  "code" : 201,
  "graph" : {
    "name" : "myGraph",
    "edgeDefinitions" : [
      {
        "collection" : "edges",
        "from" : [
          "startVertices"
        ],
        "to" : [
          "endVertices"
        ]
      }
    ],
    "orphanCollections" : [ ]
  }
}
```

Remove a graph from the graph module. Optionally drop all collections.

Drop a graph

```
DELETE /_api/gharial/graph-name
```

Removes a graph from the collection *_graphs*.

URL Parameters

- *graph-name*: The name of the graph.

Query Parameters

- *dropCollections*: Drop collections of this graph as well. Collections will only be dropped if they

are not used in other graphs.

Return Codes

- *200*: Returned if the graph could be dropped.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/gharial/social
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200
}
```

Lists all vertex collections used in this graph.

List vertex collections

```
GET /_api/gharial/graph-name/vertex
```

Lists all vertex collections within this graph.

URL Parameters

- *graph-name*: The name of the graph.

Return Codes

- *200*: Returned if the collections could be listed.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "collections" : [
    "female",
    "male"
  ]
}
```

Add an additional vertex collection to the graph.

Add vertex collection

```
POST /_api/gharial/graph-name/vertex
```

Adds a vertex collection to the set of collections of the graph. If the collection does not exist, it will be created.

URL Parameters

- *graph-name*: The name of the graph.

Return Codes

- *201*: Returned if the edge collection could be added successfully.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex
{
  "collection" : "otherVertices"
}

HTTP/1.1 201 Created
content-type: application/json

{
  "error" : false,
  "code" : 201,
  "graph" : {
    "name" : "social",
    "edgeDefinitions" : [
      {
        "collection" : "relation",
        "from" : [
          "female",
          "male"
        ],
        "to" : [
          "female",
          "male"
        ]
      }
    ],
    "orphanCollections" : [
      "otherVertices"
    ]
  }
}
```

Remove a vertex collection form the graph.

Remove vertex collection

```
DELETE /_api/gharial/graph-name/vertex/collection-name
```

Removes a vertex collection from the graph and optionally deletes the collection, if it is not used in any other graph.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the vertex collection.

Query Parameters

- *dropCollection*: Drop the collection as well. Collection will only be dropped if it is not used in other graphs.

Return Codes

- *200*: Returned if the vertex collection was removed from the graph successfully.

- *400*: Returned if the vertex collection is still used in an edge definition. In this case it cannot be removed from the graph yet, it has to be removed from the edge definition first.

- *404*: Returned if no graph with this name could be found.

Examples

You can remove vertex collections that are not used in any edge collection:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex/otherVertices
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "graph" : {
    "name" : "social",
    "edgeDefinitions" : [
      {
        "collection" : "relation",
        "from" : [
          "female",
          "male"
        ],
        "to" : [
          "female",
          "male"
        ]
      }
    ],
    "orphanCollections" : [ ]
  }
}
```

You cannot remove vertex collections that are used in edge collections:

```
shell> curl -X DELETE --data-binary @- --dump -
```

```
http://localhost:8529/_api/gharial/social/vertex/male
undefined

HTTP/1.1 400 Bad Request
content-type: application/json

{
  "error" : "The collection is not found or part of an edge
definition."
}
```

Lists all edge definitions

List edge definitions

```
GET /_api/gharial/graph-name/edge
```

Lists all edge collections within this graph.

URL Parameters

- *graph-name*: The name of the graph.

Return Codes

- *200*: Returned if the collections could be listed.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "collections" : [
    "relation"
  ]
}
```

Add a new edge definition to the graph

Add edge definition

```
POST /_api/gharial/graph-name/edge
```

Adds an additional edge definition to the graph. This edge definition has to contain a *collection* a list of each *from* and *to* vertex collections. A edge definition can only be added if this definition is either not used in any other graph, or it is used with exactly the same definition. It is not possible to store a definition "e" from "v1" to "v2" in the one graph, and "e" from "v2" to "v1" in the other graph.

URL Parameters

- *graph-name*: The name of the graph.

Body Parameters

- *collection*: The name of the edge collection to be used.

- *from*: One or many vertex collections that can contain source vertices.

- *to*: One or many edge collections that can contain target vertices.

Return Codes

- *200*: Returned if the definition could be added successfully.

- *400*: Returned if the defininition could not be added, the edge collection is used in an other graph with a different signature.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge
{
  "collection" : "lives_in",
  "from" : [
    "female",
    "male"
  ],
  "to" : [
    "city"
  ]
}

HTTP/1.1 201 Created
content-type: application/json

{
  "error" : false,
  "code" : 201,
  "graph" : {
    "name" : "social",
    "edgeDefinitions" : [
      {
        "collection" : "relation",
        "from" : [
          "female",
          "male"
        ],
        "to" : [
          "female",
          "male"
        ]
      },
      {
        "collection" : "lives_in",
        "from" : [
          "female",
          "male"
        ],
```

```
          "to" : [
            "city"
          ]
        }
      ],
      "orphanCollections" : [ ]
    }
}
```

Replace an existing edge definition

Replace an edge definition

`POST /_api/gharial/graph-name/edge/definition-name`

Change one specific edge definition. This will modify all occurrences of this definition in all graphs known to your database.

URL Parameters

- *graph-name*: The name of the graph.

- *definition-name*: The name of the edge collection used in the definition.

Body Parameters

- *collection*: The name of the edge collection to be used.

- *from*: One or many vertex collections that can contain source vertices.

- *to*: One or many edge collections that can contain target vertices.

Return Codes

- *200*: Returned if the edge definition could be replaced.

- *400*: Returned if no edge definition with this name is found in the graph.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation
{
  "collection" : "relation",
  "from" : [
    "female",
    "male",
    "animal"
  ],
  "to" : [
    "female",
    "male",
    "animal"
  ]
}

HTTP/1.1 200 OK
```

```
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "graph" : {
    "name" : "social",
    "edgeDefinitions" : [
      {
        "collection" : "relation",
        "from" : [
          "animal",
          "female",
          "male"
        ],
        "to" : [
          "animal",
          "female",
          "male"
        ]
      }
    ],
    "orphanCollections" : [ ]
  }
}
```

Remove an edge definition form the graph

Remove an edge definition from the graph

```
DELETE /_api/gharial/graph-name/edge/definition-name
```

Remove one edge definition from the graph. This will only remove the edge collection, the vertex collections remain untouched and can still be used in your queries.

URL Parameters

- *graph-name*: The name of the graph.

- *definition-name*: The name of the edge collection used in the definition.

Query Parameters

- *dropCollection*: Drop the collection as well. Collection will only be dropped if it is not used in other graphs.

Return Codes

- *200*: Returned if the edge definition could be removed from the graph.

- *400*: Returned if no edge definition with this name is found in the graph.

- *404*: Returned if no graph with this name could be found.

Examples

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation
undefined
```

```
HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "graph" : {
    "name" : "social",
    "edgeDefinitions" : [ ],
    "orphanCollections" : [
      "female",
      "male"
    ]
  }
}
```

Handling Vertices

Create a vertex

`POST /system/gharial/graph-name/vertex/collection-name`

Adds a vertex to the given collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the vertex collection the vertex belongs to.

Body Parameters

The body has to be the JSON object to be stored.

Return Codes

- *201*: Returned if the vertex could be added.

- *404*: Returned if no graph or no vertex collection with this name could be found.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex/male
{
  "name" : "Francis"
}

HTTP/1.1 201 Created
content-type: application/json
etag: 1163027125

{
  "error" : false,
  "code" : 201,
  "vertex" : {
    "_id" : "male/1163027125",
    "_rev" : "1163027125",
    "_key" : "1163027125"
  }
}
```

Get a vertex

`GET /system/gharial/graph-name/vertex/collection-name/vertex-key`

Gets a vertex from the given collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the vertex collection the vertex belongs to.

- *vertex-key*: The _key attribute of the vertex.

Return Codes

- *200*: Returned if the vertex could be found.

- *404*: Returned if no graph with this name, no vertex collection or no vertex with this id could be found.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex/female/alice
undefined

HTTP/1.1 200 OK
content-type: application/json
etag: 1164993205

{
  "error" : false,
  "code" : 200,
  "vertex" : {
    "_id" : "female/alice",
    "_rev" : "1164993205",
    "_key" : "alice",
    "name" : "Alice"
  }
}
```

Modify a vertex

`PATCH /system/gharial/graph-name/vertex/collection-name/vertex-key`

Updates the data of the specific vertex in the collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the vertex collection the vertex belongs to.

- *vertex-key*: The _key attribute of the vertex.

Body Parameters

The body has to contain a JSON object containing exactly the attributes that should be replaced.

Return Codes

- *200*: Returned if the vertex could be updated.

- *404*: Returned if no graph with this name, no vertex collection or no vertex with this id could be found.

Examples

```
shell> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex/female/alice
```

```
{
  "age" : 26
}

HTTP/1.1 200 OK
content-type: application/json
etag: 1176265397

{
  "error" : false,
  "code" : 200,
  "vertex" : {
    "_id" : "female/alice",
    "_rev" : "1176265397",
    "_oldRev" : "1173775029",
    "_key" : "alice"
  }
}
```

Replace a vertex

```
PUT /system/gharial/graph-name/vertex/collection-name/vertex-key
```

Replaces the data of a vertex in the collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the vertex collection the vertex belongs to.

- *vertex-key*: The *_key* attribute of the vertex.

Body Parameters

The body has to be the JSON object to be stored.

Return Codes

- *200*: Returned if the vertex could be replaced.

- *404*: Returned if no graph with this name, no vertex collection or no vertex with this id could be found.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex/female/alice
{
  "name" : "Alice Cooper",
  "age" : 26
}

HTTP/1.1 200 OK
content-type: application/json
etag: 1171743413

{
  "error" : false,
```

```
    "code" : 200,
    "vertex" : {
      "_id" : "female/alice",
      "_rev" : "1171743413",
      "_oldRev" : "1169253045",
      "_key" : "alice"
    }
}
```

Remove a vertex

`DELETE /system/gharial/graph-name/vertex/collection-name/vertex-key`

Removes a vertex from the collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the vertex collection the vertex belongs to.

- *vertex-key*: The *_key* attribute of the vertex.

Return Codes

- *200*: Returned if the vertex could be removed.

- *404*: Returned if no graph with this name, no vertex collection or no vertex with this id could be found.

Examples

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/vertex/female/alice
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "removed" : true
}
```

Handling Edges

Create an edge

```
POST /system/gharial/graph-name/edge/collection-name
```

Creates a new edge in the collection. Within the body the has to contain a _from_ and _to_ value referencing to valid vertices in the graph. Furthermore the edge has to be valid in the definition of this edge collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the edge collection the edge belongs to.

Body Parameters

- *_from*:
- *_to*:

The body has to be the JSON object to be stored.

Return Codes

- *201*: Returned if the edge could be created.

- *404*: Returned if no graph with this name, no edge collection or no edge with this id could be found.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation
{
  "type" : "friend",
  "_from" : "female/alice",
  "_to" : "female/diana"
}

HTTP/1.1 201 Created
content-type: application/json
etag: 1185702581

{
  "error" : false,
  "code" : 201,
  "edge" : {
    "_id" : "relation/1185702581",
    "_rev" : "1185702581",
    "_key" : "1185702581"
  }
}
```

Get an edge

```
GET /system/gharial/graph-name/edge/collection-name/edge-key
```

Gets an edge from the given collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the edge collection the edge belongs to.

- *edge-key*: The *_key* attribute of the vertex.

Return Codes

- *200*: Returned if the edge could be found.

- *404*: Returned if no graph with this name, no edge collection or no edge with this id could be found.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation/aliceAndBob
undefined

HTTP/1.1 200 OK
content-type: application/json
etag: 1188717237

{
  "error" : false,
  "code" : 200,
  "edge" : {
    "_id" : "relation/aliceAndBob",
    "_rev" : "1188717237",
    "_key" : "aliceAndBob",
    "_from" : "female/alice",
    "_to" : "male/bob",
    "type" : "married"
  }
}
```

Modify an edge

`PATCH /system/gharial/graph-name/edge/collection-name/edge-key`

Updates the data of the specific edge in the collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the edge collection the edge belongs to.

- *edge-key*: The *_key* attribute of the vertex.

Body Parameters

The body has to be a JSON object containing the attributes to be updated.

Return Codes

- *200*: Returned if the edge could be updated.

- *404*: Returned if no graph with this name, no edge collection or no edge with this id could be found.

Examples

```
shell> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation/aliceAndBob
{
  "since" : "01.01.2001"
}

HTTP/1.1 200 OK
content-type: application/json
etag: 1198678709

{
  "error" : false,
  "code" : 200,
  "edge" : {
    "_id" : "relation/aliceAndBob",
    "_rev" : "1198678709",
    "_oldRev" : "1197236917",
    "_key" : "aliceAndBob"
  }
}
```

Replace an edge

`PUT /system/gharial/graph-name/edge/collection-name/edge-key`

Replaces the data of an edge in the collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the edge collection the edge belongs to.

- *edge-key*: The *_key* attribute of the vertex.

Body Parameters

The body has to be the JSON object to be stored.

Return Codes

- *200*: Returned if the edge could be replaced.

- *404*: Returned if no graph with this name, no edge collection or no edge with this id could be found.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation/aliceAndBob
{
  "type" : "divorced"
```

```
}

HTTP/1.1 200 OK
content-type: application/json
etag: 1194222261

{
  "error" : false,
  "code" : 200,
  "edge" : {
    "_id" : "relation/aliceAndBob",
    "_rev" : "1194222261",
    "_oldRev" : "1192911541",
    "_key" : "aliceAndBob"
  }
}
```

Remove an edge

`DELETE /system/gharial/graph-name/edge/collection-name/edge-key`

Removes an edge from the collection.

URL Parameters

- *graph-name*: The name of the graph.

- *collection-name*: The name of the edge collection the edge belongs to.

- *edge-key*: The *_key* attribute of the vertex.

Return Codes

- *200*: Returned if the edge could be removed.

- *404*: Returned if no graph with this name, no edge collection or no edge with this id could be found.

Examples

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/gharial/social/edge/relation/aliceAndBob
undefined

HTTP/1.1 200 OK
content-type: application/json

{
  "error" : false,
  "code" : 200,
  "removed" : true
}
```

HTTP Interface for Graphs

Warning Deprecated

This api is deprecated and will be removed soon. Please use General Graphs instead.

POST /_api/graph*(create graph)*

Query parameters

waitForSync (boolean,optional)

Wait until document has been sync to disk.

Body parameters

graph (json,required)

The call expects a JSON hash array as body with the following attributes: _key: The name of the new graph. vertices: The name of the vertices collection. edges: The name of the egde collection.

Description

Creates a new graph. Returns an object with an attribute graph containing a list of all graph properties.

Return codes

HTTP 201

is returned if the graph was created successfully and waitForSync was true.

HTTP 202

is returned if the graph was created successfully and waitForSync was false.

HTTP 400

is returned if it failed. The response body contains an error document in this case.

Examples

```
unix> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/graph/
{"_key":"graph","vertices":"vertices","edges":"edges"}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8
etag: 103998433

{
  "graph" : {
    "_id" : "_graphs/graph",
    "_rev" : "103998433",
    "_key" : "graph",
    "edges" : "edges",
    "vertices" : "vertices"
  },
  "error" : false,
```

```
    "code" : 201
  }
```

GET /_api/graph/graph-name *(get the properties of a specific or all graphs)*

URL parameters

graph-name (string,optional)

The name of the graph.

HTTP header parameters

If-None-Match (string,optional)

If graph-name is specified, then this header can be used to check whether a specific graph has changed or not. If the "If-None-Match" header is given, then it must contain exactly one etag. The document is returned if it has a different revision than the given etag. Otherwise a HTTP 304 is returned.

If-Match (string,optional)

If graph-name is specified, then this header can be used to check whether a specific graph has changed or not. If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

If graph-name is specified, returns an object with an attribute graph containing a JSON hash with all properties of the specified graph.

If graph-name is not specified, returns a list of graph objects.

Return codes

HTTP 200

is returned if the graph was found (in case graph-name was specified) or the list of graphs was assembled successfully (in case graph-name was not specified).

HTTP 404

is returned if the graph was not found. This response code may only be returned if graph-name is specified in the request. The response body contains an error document in this case.

HTTP 304

"If-None-Match" header is given and the current graph has not a different version. This response code may only be returned if graph-name is specified in the request.

HTTP 412

"If-Match" header or rev is given and the current graph has a different version. This response code may only be returned if graph-name is specified in the request.

Examples

get graph by name

```
unix> curl --dump - http://localhost:8529/_api/graph/graph

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: 105440225

{
  "graph" : {
    "_id" : "_graphs/graph",
    "_rev" : "105440225",
    "_key" : "graph",
    "edges" : "edges",
    "vertices" : "vertices"
  },
  "error" : false,
  "code" : 200
}
```

get all graphs

```
unix> curl --dump - http://localhost:8529/_api/graph

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "graphs" : [
    {
      "_id" : "_graphs/graph1",
      "_rev" : "106947553",
      "_key" : "graph1",
      "edges" : "edges1",
      "vertices" : "vertices1"
    },
    {
      "_id" : "_graphs/graph2",
      "_rev" : "108192737",
      "_key" : "graph2",
      "edges" : "edges2",
      "vertices" : "vertices2"
    }
  ],
  "error" : false,
  "code" : 200
}
```

`DELETE /_api/graph/graph-name`*(delete graph)*

URL parameters

`graph-name (string,required)`

The name of the graph

HTTP header parameters

`If-Match (string,optional)`

If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Deletes graph, edges and vertices

Return codes

`HTTP 200`

is returned if the graph was deleted and waitForSync was true.

`HTTP 202`

is returned if the graph was deleted and waitForSync was false.

`HTTP 404`

is returned if the graph was not found. The response body contains an error document in this case.

`HTTP 412`

"If-Match" header or rev is given and the current graph has a different version

Examples

delete graph by name

```
unix> curl -X DELETE --dump -
http://localhost:8529/_api/graph/graph

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "deleted" : true,
  "error" : false,
  "code" : 200
}
```

!CHAPTE Vertex

`POST /_api/graph/graph-name/vertex-name`*(create vertex)*

URL parameters

`graph-name (string,required)`

The name of the graph

`vertex-name (string,required)`

The name of the vertex

Query parameters

`waitForSync (boolean,optional)`

Wait until document has been sync to disk.

Body parameters

`vertex (json,required)`

The call expects a JSON hash array as body with the vertex properties: _key: The name of the vertex (optional). further optional attributes.

Description

Creates a vertex in a graph. Returns an object with an attribute vertex containing a list of all vertex properties.

Return codes

`HTTP 201`

is returned if the graph was created successfully and waitForSync was true.

`HTTP 202`

is returned if the graph was created successfully and waitForSync was false.

Examples

```
unix> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/vertex
{"_key":"v1","optional1":"val1"}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: 112518113

{
  "vertex" : {
    "_id" : "vertices/v1",
    "_rev" : "112518113",
    "_key" : "v1",
    "optional1" : "val1"
  },
```

```
    "error" : false,
    "code" : 202
  }
```

GET /_api/graph/graph-name/vertex-name*(get vertex)*

URL parameters

`graph-name (string,required)`

The name of the graph

`vertex-name (string,required)`

The name of the vertex

Query parameters

`rev (string,optional)`

Revision of a vertex

HTTP header parameters

`If-None-Match (string,optional)`

If the "If-None-Match" header is given, then it must contain exactly one etag. The document is returned, if it has a different revision than the given etag. Otherwise a HTTP 304 is returned.

`If-Match (string,optional)`

If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Returns an object with an attribute vertex containing a list of all vertex properties.

Return codes

`HTTP 200`

is returned if the graph was found

`HTTP 304`

"If-Match" header is given and the current graph has not a different version

`HTTP 404`

is returned if the graph or vertex was not found. The response body contains an error document in this case.

`HTTP 412`

"If-None-Match" header or rev is given and the current graph has a different version

Examples

get vertex properties by name

```
unix> curl --dump -
http://localhost:8529/_api/graph/graph/vertex/v1

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: 115532769

{
  "vertex" : {
    "_id" : "vertices/v1",
    "_rev" : "115532769",
    "_key" : "v1",
    "optional1" : "val1"
  },
  "error" : false,
  "code" : 200
}
```

PUT /_api/graph/graph-name/vertex-name*(update vertex)*

URL parameters

graph-name (string,required)

The name of the graph

vertex-name (string,required)

The name of the vertex

Query parameters

waitForSync (boolean,optional)

Wait until vertex has been sync to disk.

rev (string,optional)

Revision of a vertex

Body parameters

vertex (json,required)

The call expects a JSON hash array as body with the new vertex properties.

HTTP header parameters

if-match (string,optional)

If the "If-Match" header is given, then it must contain exactly one etag. The document is updated, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Replaces the vertex properties. Returns an object with an attribute vertex containing a list of all vertex properties.

Return codes

`HTTP 201`

is returned if the vertex was updated successfully and waitForSync was true.

`HTTP 202`

is returned if the vertex was updated successfully and waitForSync was false.

`HTTP 404`

is returned if the graph or the vertex was not found. The response body contains an error document in this case.

`HTTP 412`

"If-Match" header or rev is given and the current vertex has a different version

Examples

```
unix> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/vertex/v1
{"optional1":"val2"}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: 120579041

{
  "vertex" : {
    "_id" : "vertices/v1",
    "_rev" : "120579041",
    "_key" : "v1",
    "optional1" : "val2"
  },
  "error" : false,
  "code" : 202
}
```

`PATCH /_api/graph/graph-name/vertex-name`*(update vertex)*

URL parameters

`graph-name (string,required)`

The name of the graph

`vertex-name (string,required)`

The name of the vertex

Query parameters

`waitForSync (boolean,optional)`

Wait until vertex has been sync to disk.

`rev (string,optional)`

Revision of a vertex

`keepNull (boolean,optional)`

Modify the behavior of the patch command to remove any attribute

Body parameters

`graph (json,required)`

The call expects a JSON hash array as body with the properties to patch.

HTTP header parameters

`if-match (string,optional)`

If the "If-Match" header is given, then it must contain exactly one etag. The document is updated, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Partially updates the vertex properties. Setting an attribute value to null in the patch document will cause a value of null be saved for the attribute by default. If the intention is to delete existing attributes with the patch command, the URL parameter keepNull can be used with a value of false. This will modify the behavior of the patch command to remove any attributes from the existing document that are contained in the patch document with an attribute value of null.

Returns an object with an attribute vertex containing a list of all vertex properties.

Return codes

`HTTP 201`

is returned if the vertex was updated successfully and waitForSync was true.

`HTTP 202`

is returned if the vertex was updated successfully and waitForSync was false.

`HTTP 404`

is returned if the graph or the vertex was not found. The response body contains an error document in this case.

`HTTP 412`

"If-Match" header or rev is given and the current vertex has a different version

Examples

```
unix> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/vertex/v1
{"optional1":"vertexPatch"}

HTTP/1.1 202 Accepted
```

```
 content-type: application/json; charset=utf-8
 etag: 123659233

 {
   "vertex" : {
     "_id" : "vertices/v1",
     "_rev" : "123659233",
     "_key" : "v1",
     "optional1" : "vertexPatch"
   },
   "error" : false,
   "code" : 202
 }

 unix> curl -X PATCH --data-binary @- --dump -
 http://localhost:8529/_api/graph/graph/vertex/v1
 {"optional1":null}

 HTTP/1.1 202 Accepted
 content-type: application/json; charset=utf-8
 etag: 124117985

 {
   "vertex" : {
     "_id" : "vertices/v1",
     "_rev" : "124117985",
     "_key" : "v1",
     "optional1" : null
   },
   "error" : false,
   "code" : 202
 }
```

DELETE /_api/graph/graph-name/vertex-name *(delete vertex)*

URL parameters

`graph-name (string,required)`

The name of the graph

`vertex-name (string,required)`

The name of the vertex

Query parameters

`waitForSync (boolean,optional)`

Wait until document has been sync to disk.

`rev (string,optional)`

Revision of a vertex

HTTP header parameters

`If-Match (string,optional)`

If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Deletes vertex and all in and out edges of the vertex

Return codes

`HTTP 200`

is returned if the vertex was deleted and waitForSync was true.

`HTTP 202`

is returned if the vertex was deleted and waitForSync was false.

`HTTP 404`

is returned if the graph or the vertex was not found. The response body contains an error document in this case.

`HTTP 412`

"If-Match" header or rev is given and the current vertex has a different version

Examples

```
unix> curl -X DELETE --dump -
http://localhost:8529/_api/graph/graph/vertex/v1

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8

{
  "deleted" : true,
  "error" : false,
  "code" : 202
}
```

@CLEARPAGE @anchor A_JSF_POST_graph_vertex @copydetails JSF_post_graph_vertex
@CLEARPAGE @anchor A_JSF_GET_graph_vertex @copydetails JSF_get_graph_vertex
@CLEARPAGE @anchor A_JSF_PUT_graph_vertex @copydetails JSF_put_graph_vertex
@CLEARPAGE @anchor A_JSF_PATCH_graph_vertex @copydetails JSF_patch_graph_vertex
@CLEARPAGE @anchor A_JSF_DELETE_graph_vertex @copydetails JSF_delete_graph_vertex -->

Edge

POST /_api/graph/graph-name/edge-name *(create edge)*

URL parameters

`graph-name (string,required)`

The name of the graph

`edge-name (string,required)`

The name of the edge

Query parameters

`waitForSync (boolean,optional)`

Wait until edge has been sync to disk.

Body parameters

`edge (json,required)`

The call expects a JSON hash array as body with the edge properties:

Description

Creates an edge in a graph. The call expects a JSON hash array as body with the edge properties:

- _key: The name of the edge (optional, if edge collection allows user defined keys).
- _from: The name of the from vertex.
- _to: The name of the to vertex.
- $label: A label for the edge (optional).

Returns an object with an attribute edge containing the list of all edge properties.

Return codes

`HTTP 201`

is returned if the edge was created successfully and waitForSync was true.

`HTTP 202`

is returned if the edge was created successfully and waitForSync was false.

Examples

```
unix> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/edge
{"_key":"edge1","_from":"vert2","_to":"vert1","optional1":"val1"}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: 144630753

{
```

```
    "edge" : {
      "_id" : "edges/edge1",
      "_rev" : "144630753",
      "_key" : "edge1",
      "_from" : "vertices/vert2",
      "_to" : "vertices/vert1",
      "$label" : null,
      "optional1" : "val1"
    },
    "error" : false,
    "code" : 202
}
```

GET /_api/graph/graph-name/edge*(get edge)*

URL parameters

graph-name (string,required)

The name of the graph

edge-name (string,required)

The name of the edge

Query parameters

rev (string,optional)

Revision of an edge

HTTP header parameters

if-none-match (string,optional)

If the "If-None-Match" header is given, then it must contain exactly one etag. The document is returned, if it has a different revision than the given etag. Otherwise a HTTP 304 is returned. if-match (string,optional) If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Returns an object with an attribute edge containing a list of all edge properties.

Return codes

HTTP 200

is returned if the edge was found

HTTP 304

"If-Match" header is given and the current edge has not a different version

HTTP 404

is returned if the graph or edge was not found. The response body contains an error document in this case.

```
HTTP 412
```

"If-None-Match" header or rev is given and the current edge has a different version

Examples

```
unix> curl --dump -
http://localhost:8529/_api/graph/graph/edge/edge1

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: 147579873

{
  "edge" : {
    "_id" : "edges/edge1",
    "_rev" : "147579873",
    "_key" : "edge1",
    "_from" : "vertices/vert1",
    "_to" : "vertices/vert2",
    "$label" : null,
    "optional1" : "val1"
  },
  "error" : false,
  "code" : 200
}
```

`PUT /_api/graph/graph-name/edge-name`*(update edge)*

URL parameters

`graph-name (string,required)`

The name of the graph

`edge-name (string,required)`

The name of the edge

Query parameters

`waitForSync (boolean,optional)`

Wait until edge has been sync to disk.

`rev (string,optional)`

Revision of an edge

Body parameters

`edge (json,required)`

The call expects a JSON hash array as body with the new edge properties.

HTTP header parameters

`if-match (string,optional)`

If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Replaces the optional edge properties. The call expects a JSON hash array as body with the new edge properties.

Returns an object with an attribute edge containing a list of all edge properties.

Return codes

`HTTP 201`

is returned if the edge was updated successfully and waitForSync was true.

`HTTP 202`

is returned if the edge was updated successfully and waitForSync was false.

`HTTP 404`

is returned if the graph or the edge was not found. The response body contains an error document in this case.

`HTTP 412`

"If-Match" header or rev is given and the current edge has a different version *Examples*

```
unix> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/edge/edge1
{"optional1":"val2"}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: 154526689

{
  "edge" : {
    "_id" : "edges/edge1",
    "_rev" : "154526689",
    "_key" : "edge1",
    "_from" : "vertices/vert1",
    "_to" : "vertices/vert2",
    "$label" : null,
    "optional1" : "val2"
  },
  "error" : false,
  "code" : 202
}
```

`PATCH /_api/graph/graph-name/edge-name`*(update edge)*

URL parameters

`graph-name (string,required)`

The name of the graph

```
edge-name (string,required)
```

The name of the edge

Query parameters

```
waitForSync (boolean,optional)
```

Wait until edge has been sync to disk.

```
rev (string,optional)
```

Revision of an edge

```
keepNull (boolean,optional)
```

Modify the behavior of the patch command to remove any attribute

Body parameters

```
edge-properties (json,required)
```

The call expects a JSON hash array as body with the properties to patch.

HTTP header parameters

```
if-match (string,optional)
```

If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Partially updates the edge properties. Setting an attribute value to null in the patch document will cause a value of null be saved for the attribute by default. If the intention is to delete existing attributes with the patch command, the URL parameter keepNull can be used with a value of false. This will modify the behavior of the patch command to remove any attributes from the existing document that are contained in the patch document with an attribute value of null.

Returns an object with an attribute edge containing a list of all edge properties.

Return codes

```
HTTP 201
```

is returned if the edge was updated successfully and waitForSync was true.

```
HTTP 202
```

is returned if the edge was updated successfully and waitForSync was false.

```
HTTP 404
```

is returned if the graph or the edge was not found. The response body contains an error document in this case.

```
HTTP 412
```

"If-Match" header or rev is given and the current edge has a different version

Examples

```
unix> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/edge/edge1
{"optional3":"val3"}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: 158065633

{
  "edge" : {
    "_id" : "edges/edge1",
    "_rev" : "158065633",
    "_key" : "edge1",
    "_from" : "vertices/vert1",
    "_to" : "vertices/vert2",
    "$label" : null,
    "optional1" : "val1",
    "optional3" : "val3"
  },
  "error" : false,
  "code" : 202
}
```

DELETE /_api/graph/graph-name/edge-name*(delete edge)*

URL parameters

graph-name (string,required)

The name of the graph

edge-name (string,required)

The name of the edge

Query parameters

waitForSync (boolean,optional)

Wait until edge has been sync to disk.

rev (string,optional)

Revision of an edge

HTTP header parameters

if-match (string,optional)

If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a HTTP 412 is returned. As an alternative you can supply the etag in an attribute rev in the URL.

Description

Deletes an edge of the graph

Return codes

`HTTP 200`

is returned if the edge was deleted successfully and waitForSync was true.

`HTTP 202`

is returned if the edge was deleted successfully and waitForSync was false.

`HTTP 404`

is returned if the graph or the edge was not found. The response body contains an error document in this case.

`HTTP 412`

"If-Match" header or rev is given and the current edge has a different version

Examples

```
unix> curl -X DELETE --dump -
http://localhost:8529/_api/graph/graph/edge/edge1

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8

{
  "deleted" : true,
  "error" : false,
  "code" : 202
}
```

`POST /_api/graph/graph-name/vertices/vertice-name`*(get vertices)*

URL parameters

`graph-name (string,required)`

The name of the graph

Body parameters

`graph (json,required)`

The call expects a JSON hash array as body to filter the result:

Description

Returns a cursor. The call expects a JSON hash array as body to filter the result:

- batchSize: the batch size of the returned cursor
- limit: limit the result size
- count: return the total number of results (default "false")
- filter: a optional filter

The attributes of filter

- direction: Filter for inbound (value "in") or outbound (value "out") neighbors. Default value is

"any".
- labels: filter by an array of edge labels (empty array means no restriction)
- properties: filter neighbors by an array of edge properties

The attributes of a property filter

- key: filter the result vertices by a key value pair
- value: the value of the key
- compare: a compare operator

Return codes

`HTTP 201`

is returned if the cursor was created

Examples

Select all vertices

```
unix> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/vertices/v2
{"batchSize" : 100, "filter" : {"direction" : "any", "properties":
[] }}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
   "result" : [
     {
       "_id" : "vertices/v1",
       "_rev" : "132637665",
       "_key" : "v1",
       "optional1" : "val1"
     },
     {
       "_id" : "vertices/v4",
       "_rev" : "133620705",
       "_key" : "v4",
       "optional1" : "val1"
     }
   ],
   "hasMore" : false,
   "error" : false,
   "code" : 201
}
```

Select vertices by direction and property filter

```
unix> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/vertices/v2
{"batchSize" : 100, "filter" : {"direction" : "out", "properties":[
{ "key": "optional1", "value": "val2", "compare" : "==" }, ] }}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8
```

```
{
   "result" : [
      {
         "_id" : "vertices/v4",
         "_rev" : "139125729",
         "_key" : "v4",
         "optional1" : "val2"
      },
      {
         "_id" : "vertices/v1",
         "_rev" : "138142689",
         "_key" : "v1",
         "optional1" : "val1"
      }
   ],
   "hasMore" : false,
   "error" : false,
   "code" : 201
}
```

POST /_api/graph/graph-name/edges/vertex-name*(get edges)*

URL parameters

graph-name (string,required)

The name of the graph

vertex-name (string,required)

The name of the vertex

Body parameters

edge-properties (json,required)

The call expects a JSON hash array as body to filter the result:

Description

Returns a cursor.

The call expects a JSON hash array as body to filter the result:

- batchSize: the batch size of the returned cursor
- limit: limit the result size
- count: return the total number of results (default "false")
- filter: a optional filter

The attributes of filter

- direction: Filter for inbound (value "in") or outbound (value "out") neighbors. Default value is "any".
- labels: filter by an array of edge labels
- properties: filter neighbors by an array of properties

The attributes of a property filter

- key: filter the result vertices by a key value pair

- value: the value of the key
- compare: a compare operator

Return codes

```
HTTP 201
```

is returned if the cursor was created

Examples

Select all edges

```
unix> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/graph/graph/edges/v2
{"batchSize" : 100, "filter" : { "direction" : "any" }}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "edges/edge1",
      "_rev" : "167568353",
      "_key" : "edge1",
      "_from" : "vertices/v1",
      "_to" : "vertices/v2",
      "$label" : null,
      "optional1" : "val1"
    },
    {
      "_id" : "edges/edge3",
      "_rev" : "168485857",
      "_key" : "edge3",
      "_from" : "vertices/v2",
      "_to" : "vertices/v4",
      "$label" : null,
      "optional1" : "val1"
    }
  ],
  "hasMore" : false,
  "error" : false,
  "code" : 201
}
```

HTTP Interface for Traversals

Traversals

ArangoDB's graph traversals are executed on the server. Traversals can be initiated by clients by sending the traversal description for execution to the server.

Traversals in ArangoDB are used to walk over a graph stored in one edge collection. It can easily be described which edges of the graph should be followed and which actions should be performed on each visited vertex. Furthermore the ordering of visiting the nodes can be specified, for instance depth-first or breadth-first search are offered.

Executing Traversals via HTTP

executes a traversal

POST /_api/traversal

- *body*:

Starts a traversal starting from a given vertex and following. edges contained in a given edgeCollection. The request must contain the following attributes.

- *startVertex*: id of the startVertex, e.g. *"users/foo"*.

- *edgeCollection*: **Deprecated** name of the collection that contains the edges.

- *graphName*: name of the graph that contains the edges.

- *filter* (optional, default is to include all nodes): body (JavaScript code) of custom filter function function signature: (config, vertex, path) -> mixed can return four different string values:
  - *"exclude"* -> this vertex will not be visited.
  - *"prune"* -> the edges of this vertex will not be followed.
  - *""* or *undefined* -> visit the vertex and follow it's edges.
  - *Array* -> containing any combination of the above. If there is at least one *"exclude"* or *"prune"* respectivly is contained, it's effect will occur.

- *minDepth* (optional, ANDed with any existing filters): visits only nodes in at least the given depth

- *maxDepth* (optional, ANDed with any existing filters): visits only nodes in at most the given depth

- *visitor* (optional): body (JavaScript) code of custom visitor function function signature: (config, result, vertex, path) -> void visitor function can do anything, but its return value is ignored. To populate a result, use the *result* variable by reference

- *direction* (optional): direction for traversal
  - *if set*, must be either *"outbound"*, *"inbound"*, or *"any"*
  - *if not set*, the *expander* attribute must be specified

- *init* (optional): body (JavaScript) code of custom result initialisation function function signature: (config, result) -> void initialise any values in result with what is required

- *expander* (optional): body (JavaScript) code of custom expander function *must* be set if *direction* attribute is **not** set function signature: (config, vertex, path) -> array expander must return an array of the connections for *vertex* each connection is an object with the attributes *edge* and *vertex*
- *sort* (optional): body (JavaScript) code of a custom comparison function for the edges. The signature of this function is (l, r) -> integer (where l and r are edges) and must return -1 if l is smaller than, +1 if l is greater than, and 0 if l and r are equal. The reason for this is the following: The order of edges returned for a certain vertex is undefined. This is because there is no natural order of edges for a vertex with multiple connected edges. To explicitly define the order in which edges on the vertex are followed, you can specify an edge comparator function with this attribute. Note that the value here has to be a string to conform to the JSON standard, which in turn is parsed as function body on the server side. Furthermore note that this attribute is only used for the standard expanders. If you use your custom expander you have to do the sorting yourself within the expander code.

- *strategy* (optional): traversal strategy can be *"depthfirst"* or *"breadthfirst"*

- *order* (optional): traversal order can be *"preorder"* or *"postorder"*

- *itemOrder* (optional): item iteration order can be *"forward"* or *"backward"*

- *uniqueness* (optional): specifies uniqueness for vertices and edges visited if set, must be an object like this: *"uniqueness": {"vertices": "none"|"global"|path", "edges": "none"|"global"|"path"}*

- *maxIterations* (optional): Maximum number of iterations in each traversal. This number can be set to prevent endless loops in traversal of cyclic graphs. When a traversal performs as many iterations as the *maxIterations* value, the traversal will abort with an error. If *maxIterations* is not set, a server-defined value may be used.

If the Traversal is successfully executed *HTTP 200* will be returned. Additionally the *result* object will be returned by the traversal.

For successful traversals, the returned JSON object has the following properties:

- *error*: boolean flag to indicate if an error occurred (*false* in this case)

- *code*: the HTTP status code

- *result*: the return value of the traversal

If the traversal specification is either missing or malformed, the server will respond with *HTTP 400*.

The body of the response will then contain a JSON object with additional error details. The object has the following attributes:

- *error*: boolean flag to indicate that an error occurred (*true* in this case)

- *code*: the HTTP status code

- *errorNum*: the server error number

- *errorMessage*: a descriptive error message

Return Codes

- *200*: If the traversal is fully executed *HTTP 200* will be returned.

- *400*: If the traversal specification is either missing or malformed, the server will respond with *HTTP 400*.

- *404*: The server will responded with *HTTP 404* if the specified edge collection does not exist, or the specified start vertex cannot be found.

- *500*: The server will responded with *HTTP 500* when an error occurs inside the traversal or if a traversal performs more than *maxIterations* iterations.

Examples

In the following examples the underlying graph will contain five persons *Alice*, *Bob*, *Charlie*, *Dave* and *Eve*. We will have the following directed relations:

- *Alice* knows *Bob*
- *Bob* knows *Charlie*
- *Bob* knows *Dave*
- *Eve* knows *Alice*
- *Eve* knows *Bob*

The starting vertex will always be Alice.

Follow only outbound edges:

```
shell> curl -X POST --data-binary @- --dump -
```

```
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "outbound"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1034183349",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1034379957",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1034642101",
          "_key" : "charlie",
          "name" : "Charlie"
        },
        {
          "_id" : "persons/dave",
          "_rev" : "1034838709",
          "_key" : "dave",
          "name" : "Dave"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1034183349",
              "_key" : "alice",
              "name" : "Alice"
            }
          ]
        },
        {
          "edges" : [
            {
              "_id" : "knows/1035297461",
              "_rev" : "1035297461",
              "_key" : "1035297461",
              "_from" : "persons/alice",
              "_to" : "persons/bob"
            }
          ],
          "vertices" : [
            {
```

```json
        "_id" : "persons/alice",
        "_rev" : "1034183349",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1034379957",
        "_key" : "bob",
        "name" : "Bob"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1035297461",
        "_rev" : "1035297461",
        "_key" : "1035297461",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      },
      {
        "_id" : "knows/1035494069",
        "_rev" : "1035494069",
        "_key" : "1035494069",
        "_from" : "persons/bob",
        "_to" : "persons/charlie"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1034183349",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1034379957",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/charlie",
        "_rev" : "1034642101",
        "_key" : "charlie",
        "name" : "Charlie"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1035297461",
        "_rev" : "1035297461",
        "_key" : "1035297461",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      },
```

```
                  {
                    "_id" : "knows/1035690677",
                    "_rev" : "1035690677",
                    "_key" : "1035690677",
                    "_from" : "persons/bob",
                    "_to" : "persons/dave"
                  }
                ],
                "vertices" : [
                  {
                    "_id" : "persons/alice",
                    "_rev" : "1034183349",
                    "_key" : "alice",
                    "name" : "Alice"
                  },
                  {
                    "_id" : "persons/bob",
                    "_rev" : "1034379957",
                    "_key" : "bob",
                    "name" : "Bob"
                  },
                  {
                    "_id" : "persons/dave",
                    "_rev" : "1034838709",
                    "_key" : "dave",
                    "name" : "Dave"
                  }
                ]
            }
          ]
        }
      },
    "error" : false,
    "code" : 200
}
```

Follow only inbound edges:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "inbound"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
    "result" : {
      "visited" : {
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1038639797",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
```

```
          "_rev" : "1039491765",
          "_key" : "eve",
          "name" : "Eve"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1038639797",
              "_key" : "alice",
              "name" : "Alice"
            }
          ]
        },
        {
          "edges" : [
            {
              "_id" : "knows/1040343733",
              "_rev" : "1040343733",
              "_key" : "1040343733",
              "_from" : "persons/eve",
              "_to" : "persons/alice"
            }
          ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1038639797",
              "_key" : "alice",
              "name" : "Alice"
            },
            {
              "_id" : "persons/eve",
              "_rev" : "1039491765",
              "_key" : "eve",
              "name" : "Eve"
            }
          ]
        }
      ]
    }
  },
  "error" : false,
  "code" : 200
}
```

Follow any direction of edges:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","direction":
{"vertices":"none","edges":"global"}}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
```

```json
{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1042834101",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1043686069",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1043030709",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1042834101",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1043292853",
          "_key" : "charlie",
          "name" : "Charlie"
        },
        {
          "_id" : "persons/dave",
          "_rev" : "1043489461",
          "_key" : "dave",
          "name" : "Dave"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1042834101",
              "_key" : "alice",
              "name" : "Alice"
            }
          ]
        },
        {
          "edges" : [
            {
              "_id" : "knows/1044538037",
              "_rev" : "1044538037",
              "_key" : "1044538037",
```

```json
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1042834101",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1043686069",
          "_key" : "eve",
          "name" : "Eve"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1044538037",
          "_rev" : "1044538037",
          "_key" : "1044538037",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1044734645",
          "_rev" : "1044734645",
          "_key" : "1044734645",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1042834101",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1043686069",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1043030709",
          "_key" : "bob",
          "name" : "Bob"
        }
      ]
    },
    {
      "edges" : [
        {
```

```json
        "_id" : "knows/1044538037",
        "_rev" : "1044538037",
        "_key" : "1044538037",
        "_from" : "persons/eve",
        "_to" : "persons/alice"
      },
      {
        "_id" : "knows/1044734645",
        "_rev" : "1044734645",
        "_key" : "1044734645",
        "_from" : "persons/eve",
        "_to" : "persons/bob"
      },
      {
        "_id" : "knows/1043948213",
        "_rev" : "1043948213",
        "_key" : "1043948213",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1042834101",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1043686069",
        "_key" : "eve",
        "name" : "Eve"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1043030709",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/alice",
        "_rev" : "1042834101",
        "_key" : "alice",
        "name" : "Alice"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1044538037",
        "_rev" : "1044538037",
        "_key" : "1044538037",
        "_from" : "persons/eve",
        "_to" : "persons/alice"
      },
      {
        "_id" : "knows/1044734645",
        "_rev" : "1044734645",
```

```json
          "_key" : "1044734645",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1044144821",
          "_rev" : "1044144821",
          "_key" : "1044144821",
          "_from" : "persons/bob",
          "_to" : "persons/charlie"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1042834101",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1043686069",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1043030709",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1043292853",
          "_key" : "charlie",
          "name" : "Charlie"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1044538037",
          "_rev" : "1044538037",
          "_key" : "1044538037",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1044734645",
          "_rev" : "1044734645",
          "_key" : "1044734645",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1044341429",
          "_rev" : "1044341429",
          "_key" : "1044341429",
          "_from" : "persons/bob",
```

```
                    "_to" : "persons/dave"
                 }
              ],
              "vertices" : [
                 {
                    "_id" : "persons/alice",
                    "_rev" : "1042834101",
                    "_key" : "alice",
                    "name" : "Alice"
                 },
                 {
                    "_id" : "persons/eve",
                    "_rev" : "1043686069",
                    "_key" : "eve",
                    "name" : "Eve"
                 },
                 {
                    "_id" : "persons/bob",
                    "_rev" : "1043030709",
                    "_key" : "bob",
                    "name" : "Bob"
                 },
                 {
                    "_id" : "persons/dave",
                    "_rev" : "1043489461",
                    "_key" : "dave",
                    "name" : "Dave"
                 }
              ]
           }
        ]
     }
  },
  "error" : false,
  "code" : 200
}
```

Excluding *Charlie* and *Bob*:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "outbound", "filter" : "if (vertex.name === \"Bob\"
|| vertex.name === \"Charlie\") {return \"exclude\";}return;"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
     "visited" : {
        "vertices" : [
           {
              "_id" : "persons/alice",
              "_rev" : "1048011445",
              "_key" : "alice",
              "name" : "Alice"
           },
```

```
          {
            "_id" : "persons/dave",
            "_rev" : "1048666805",
            "_key" : "dave",
            "name" : "Dave"
          }
        ],
        "paths" : [
          {
            "edges" : [ ],
            "vertices" : [
              {
                "_id" : "persons/alice",
                "_rev" : "1048011445",
                "_key" : "alice",
                "name" : "Alice"
              }
            ]
          },
          {
            "edges" : [
              {
                "_id" : "knows/1049125557",
                "_rev" : "1049125557",
                "_key" : "1049125557",
                "_from" : "persons/alice",
                "_to" : "persons/bob"
              },
              {
                "_id" : "knows/1049518773",
                "_rev" : "1049518773",
                "_key" : "1049518773",
                "_from" : "persons/bob",
                "_to" : "persons/dave"
              }
            ],
            "vertices" : [
              {
                "_id" : "persons/alice",
                "_rev" : "1048011445",
                "_key" : "alice",
                "name" : "Alice"
              },
              {
                "_id" : "persons/bob",
                "_rev" : "1048208053",
                "_key" : "bob",
                "name" : "Bob"
              },
              {
                "_id" : "persons/dave",
                "_rev" : "1048666805",
                "_key" : "dave",
                "name" : "Dave"
              }
            ]
          }
        ]
      }
    },
```

```
    "error" : false,
    "code" : 200
}
```

Do not follow edges from *Bob*:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "outbound", "filter" : "if (vertex.name === \"Bob\")
{return \"prune\";}return;"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1052467893",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1052664501",
          "_key" : "bob",
          "name" : "Bob"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1052467893",
              "_key" : "alice",
              "name" : "Alice"
            }
          ]
        },
        {
          "edges" : [
            {
              "_id" : "knows/1053582005",
              "_rev" : "1053582005",
              "_key" : "1053582005",
              "_from" : "persons/alice",
              "_to" : "persons/bob"
            }
          ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1052467893",
```

```
                  "_key" : "alice",
                  "name" : "Alice"
                },
                {
                  "_id" : "persons/bob",
                  "_rev" : "1052664501",
                  "_key" : "bob",
                  "name" : "Bob"
                }
              ]
            }
          ]
        }
      },
    "error" : false,
    "code" : 200
}
```

Visit only nodes in a depth of at least 2:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "outbound", "minDepth" : 2}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/charlie",
          "_rev" : "1057055413",
          "_key" : "charlie",
          "name" : "Charlie"
        },
        {
          "_id" : "persons/dave",
          "_rev" : "1057252021",
          "_key" : "dave",
          "name" : "Dave"
        }
      ],
      "paths" : [
        {
          "edges" : [
            {
              "_id" : "knows/1057710773",
              "_rev" : "1057710773",
              "_key" : "1057710773",
              "_from" : "persons/alice",
              "_to" : "persons/bob"
            },
            {
              "_id" : "knows/1057907381",
              "_rev" : "1057907381",
```

```json
          "_key" : "1057907381",
          "_from" : "persons/bob",
          "_to" : "persons/charlie"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1056596661",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1056793269",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1057055413",
          "_key" : "charlie",
          "name" : "Charlie"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1057710773",
          "_rev" : "1057710773",
          "_key" : "1057710773",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1058103989",
          "_rev" : "1058103989",
          "_key" : "1058103989",
          "_from" : "persons/bob",
          "_to" : "persons/dave"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1056596661",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1056793269",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/dave",
          "_rev" : "1057252021",
          "_key" : "dave",
```

```
                "name" : "Dave"
              }
            ]
          }
        ]
      }
    },
    "error" : false,
    "code" : 200
}
```

Visit only nodes in a depth of at most 1:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "outbound", "maxDepth" : 1}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1062429365",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1062625973",
          "_key" : "bob",
          "name" : "Bob"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1062429365",
              "_key" : "alice",
              "name" : "Alice"
            }
          ]
        },
        {
          "edges" : [
            {
              "_id" : "knows/1063543477",
              "_rev" : "1063543477",
              "_key" : "1063543477",
              "_from" : "persons/alice",
              "_to" : "persons/bob"
```

```
            }
          ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1062429365",
              "_key" : "alice",
              "name" : "Alice"
            },
            {
              "_id" : "persons/bob",
              "_rev" : "1062625973",
              "_key" : "bob",
              "name" : "Bob"
            }
          ]
        }
      ]
    }
  },
  "error" : false,
  "code" : 200
}
```

Count all visited nodes and return a list of nodes only:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{ "startVertex": "persons/alice", "graphName" : "knows_graph",
"direction" : "outbound", "init" : "result.visited = 0;
result.myVertices = [ ];", "visitor" : "result.visited++;
result.myVertices.push(vertex);"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : 4,
    "myVertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1066558133",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1066754741",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/charlie",
        "_rev" : "1067016885",
        "_key" : "charlie",
        "name" : "Charlie"
      },
```

```
      {
        "_id" : "persons/dave",
        "_rev" : "1067213493",
        "_key" : "dave",
        "name" : "Dave"
      }
    ]
  },
  "error" : false,
  "code" : 200
}
```

Expand only inbound edges of *Alice* and outbound edges of *Eve*:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","expander":"
connections = [ ];if (vertex.name === \"Alice\")
{config.datasource.getInEdges(vertex).forEach(function (e)
{connections.push({ vertex:
require(\"internal\").db._document(e._from), edge: e});});}if
(vertex.name === \"Eve\")
{config.datasource.getOutEdges(vertex).forEach(function (e)
{connections.push({vertex:
require(\"internal\").db._document(e._to), edge: e});});}return
connections;"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1071014581",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1071866549",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1071211189",
          "_key" : "bob",
          "name" : "Bob"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
```

```
            "_id" : "persons/alice",
            "_rev" : "1071014581",
            "_key" : "alice",
            "name" : "Alice"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1072718517",
            "_rev" : "1072718517",
            "_key" : "1072718517",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1071014581",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1071866549",
            "_key" : "eve",
            "name" : "Eve"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1072718517",
            "_rev" : "1072718517",
            "_key" : "1072718517",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          },
          {
            "_id" : "knows/1072915125",
            "_rev" : "1072915125",
            "_key" : "1072915125",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1071014581",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1071866549",
            "_key" : "eve",
```

```
                "name" : "Eve"
              },
              {
                "_id" : "persons/bob",
                "_rev" : "1071211189",
                "_key" : "bob",
                "name" : "Bob"
              }
            ]
          }
        ]
      }
    },
    "error" : false,
    "code" : 200
}
```

Follow the *depthfirst* strategy:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","direction":

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
   "result" : {
     "visited" : {
       "vertices" : [
         {
           "_id" : "persons/alice",
           "_rev" : "1075339957",
           "_key" : "alice",
           "name" : "Alice"
         },
         {
           "_id" : "persons/eve",
           "_rev" : "1076191925",
           "_key" : "eve",
           "name" : "Eve"
         },
         {
           "_id" : "persons/bob",
           "_rev" : "1075536565",
           "_key" : "bob",
           "name" : "Bob"
         },
         {
           "_id" : "persons/alice",
           "_rev" : "1075339957",
           "_key" : "alice",
           "name" : "Alice"
         },
         {
           "_id" : "persons/charlie",
           "_rev" : "1075798709",
```

```
        "_key" : "charlie",
        "name" : "Charlie"
      },
      {
        "_id" : "persons/dave",
        "_rev" : "1075995317",
        "_key" : "dave",
        "name" : "Dave"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1075536565",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1076191925",
        "_key" : "eve",
        "name" : "Eve"
      },
      {
        "_id" : "persons/alice",
        "_rev" : "1075339957",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/charlie",
        "_rev" : "1075798709",
        "_key" : "charlie",
        "name" : "Charlie"
      },
      {
        "_id" : "persons/dave",
        "_rev" : "1075995317",
        "_key" : "dave",
        "name" : "Dave"
      }
    ],
    "paths" : [
      {
        "edges" : [ ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1075339957",
            "_key" : "alice",
            "name" : "Alice"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1077043893",
            "_rev" : "1077043893",
            "_key" : "1077043893",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
```

```
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1075339957",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1076191925",
        "_key" : "eve",
        "name" : "Eve"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1077043893",
        "_rev" : "1077043893",
        "_key" : "1077043893",
        "_from" : "persons/eve",
        "_to" : "persons/alice"
      },
      {
        "_id" : "knows/1077240501",
        "_rev" : "1077240501",
        "_key" : "1077240501",
        "_from" : "persons/eve",
        "_to" : "persons/bob"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1075339957",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1076191925",
        "_key" : "eve",
        "name" : "Eve"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1075536565",
        "_key" : "bob",
        "name" : "Bob"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1077043893",
        "_rev" : "1077043893",
```

```
          "_key" : "1077043893",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1077240501",
          "_rev" : "1077240501",
          "_key" : "1077240501",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1076454069",
          "_rev" : "1076454069",
          "_key" : "1076454069",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1075339957",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1076191925",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1075536565",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1075339957",
          "_key" : "alice",
          "name" : "Alice"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1077043893",
          "_rev" : "1077043893",
          "_key" : "1077043893",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1077240501",
          "_rev" : "1077240501",
          "_key" : "1077240501",
          "_from" : "persons/eve",
```

```
            "_to" : "persons/bob"
          },
          {
            "_id" : "knows/1076650677",
            "_rev" : "1076650677",
            "_key" : "1076650677",
            "_from" : "persons/bob",
            "_to" : "persons/charlie"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1075339957",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1076191925",
            "_key" : "eve",
            "name" : "Eve"
          },
          {
            "_id" : "persons/bob",
            "_rev" : "1075536565",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "persons/charlie",
            "_rev" : "1075798709",
            "_key" : "charlie",
            "name" : "Charlie"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1077043893",
            "_rev" : "1077043893",
            "_key" : "1077043893",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          },
          {
            "_id" : "knows/1077240501",
            "_rev" : "1077240501",
            "_key" : "1077240501",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          },
          {
            "_id" : "knows/1076847285",
            "_rev" : "1076847285",
            "_key" : "1076847285",
            "_from" : "persons/bob",
            "_to" : "persons/dave"
          }
```

```
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1075339957",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1076191925",
        "_key" : "eve",
        "name" : "Eve"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1075536565",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/dave",
        "_rev" : "1075995317",
        "_key" : "dave",
        "name" : "Dave"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1076454069",
        "_rev" : "1076454069",
        "_key" : "1076454069",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1075339957",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1075536565",
        "_key" : "bob",
        "name" : "Bob"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1076454069",
        "_rev" : "1076454069",
        "_key" : "1076454069",
        "_from" : "persons/alice",
```

```
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1077240501",
          "_rev" : "1077240501",
          "_key" : "1077240501",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1075339957",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1075536565",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1076191925",
          "_key" : "eve",
          "name" : "Eve"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1076454069",
          "_rev" : "1076454069",
          "_key" : "1076454069",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1077240501",
          "_rev" : "1077240501",
          "_key" : "1077240501",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1077043893",
          "_rev" : "1077043893",
          "_key" : "1077043893",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1075339957",
          "_key" : "alice",
```

```json
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1075536565",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1076191925",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1075339957",
          "_key" : "alice",
          "name" : "Alice"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1076454069",
          "_rev" : "1076454069",
          "_key" : "1076454069",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1076650677",
          "_rev" : "1076650677",
          "_key" : "1076650677",
          "_from" : "persons/bob",
          "_to" : "persons/charlie"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1075339957",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1075536565",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1075798709",
          "_key" : "charlie",
          "name" : "Charlie"
        }
      ]
    },
```

```
            {
              "edges" : [
                {
                  "_id" : "knows/1076454069",
                  "_rev" : "1076454069",
                  "_key" : "1076454069",
                  "_from" : "persons/alice",
                  "_to" : "persons/bob"
                },
                {
                  "_id" : "knows/1076847285",
                  "_rev" : "1076847285",
                  "_key" : "1076847285",
                  "_from" : "persons/bob",
                  "_to" : "persons/dave"
                }
              ],
              "vertices" : [
                {
                  "_id" : "persons/alice",
                  "_rev" : "1075339957",
                  "_key" : "alice",
                  "name" : "Alice"
                },
                {
                  "_id" : "persons/bob",
                  "_rev" : "1075536565",
                  "_key" : "bob",
                  "name" : "Bob"
                },
                {
                  "_id" : "persons/dave",
                  "_rev" : "1075995317",
                  "_key" : "dave",
                  "name" : "Dave"
                }
              ]
            }
          }
        ]
      }
    },
    "error" : false,
    "code" : 200
}
```

Using *postorder* ordering:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","direction":

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
```

```
"vertices" : [
  {
    "_id" : "persons/alice",
    "_rev" : "1081500341",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "persons/charlie",
    "_rev" : "1081959093",
    "_key" : "charlie",
    "name" : "Charlie"
  },
  {
    "_id" : "persons/dave",
    "_rev" : "1082155701",
    "_key" : "dave",
    "name" : "Dave"
  },
  {
    "_id" : "persons/bob",
    "_rev" : "1081696949",
    "_key" : "bob",
    "name" : "Bob"
  },
  {
    "_id" : "persons/eve",
    "_rev" : "1082352309",
    "_key" : "eve",
    "name" : "Eve"
  },
  {
    "_id" : "persons/alice",
    "_rev" : "1081500341",
    "_key" : "alice",
    "name" : "Alice"
  },
  {
    "_id" : "persons/eve",
    "_rev" : "1082352309",
    "_key" : "eve",
    "name" : "Eve"
  },
  {
    "_id" : "persons/charlie",
    "_rev" : "1081959093",
    "_key" : "charlie",
    "name" : "Charlie"
  },
  {
    "_id" : "persons/dave",
    "_rev" : "1082155701",
    "_key" : "dave",
    "name" : "Dave"
  },
  {
    "_id" : "persons/bob",
    "_rev" : "1081696949",
    "_key" : "bob",
    "name" : "Bob"
```

```
      },
      {
        "_id" : "persons/alice",
        "_rev" : "1081500341",
        "_key" : "alice",
        "name" : "Alice"
      }
    ],
    "paths" : [
      {
        "edges" : [
          {
            "_id" : "knows/1083204277",
            "_rev" : "1083204277",
            "_key" : "1083204277",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          },
          {
            "_id" : "knows/1083400885",
            "_rev" : "1083400885",
            "_key" : "1083400885",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          },
          {
            "_id" : "knows/1082614453",
            "_rev" : "1082614453",
            "_key" : "1082614453",
            "_from" : "persons/alice",
            "_to" : "persons/bob"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1081500341",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1082352309",
            "_key" : "eve",
            "name" : "Eve"
          },
          {
            "_id" : "persons/bob",
            "_rev" : "1081696949",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "persons/alice",
            "_rev" : "1081500341",
            "_key" : "alice",
            "name" : "Alice"
          }
        ]
      },
```

```
{
  "edges" : [
    {
      "_id" : "knows/1083204277",
      "_rev" : "1083204277",
      "_key" : "1083204277",
      "_from" : "persons/eve",
      "_to" : "persons/alice"
    },
    {
      "_id" : "knows/1083400885",
      "_rev" : "1083400885",
      "_key" : "1083400885",
      "_from" : "persons/eve",
      "_to" : "persons/bob"
    },
    {
      "_id" : "knows/1082811061",
      "_rev" : "1082811061",
      "_key" : "1082811061",
      "_from" : "persons/bob",
      "_to" : "persons/charlie"
    }
  ],
  "vertices" : [
    {
      "_id" : "persons/alice",
      "_rev" : "1081500341",
      "_key" : "alice",
      "name" : "Alice"
    },
    {
      "_id" : "persons/eve",
      "_rev" : "1082352309",
      "_key" : "eve",
      "name" : "Eve"
    },
    {
      "_id" : "persons/bob",
      "_rev" : "1081696949",
      "_key" : "bob",
      "name" : "Bob"
    },
    {
      "_id" : "persons/charlie",
      "_rev" : "1081959093",
      "_key" : "charlie",
      "name" : "Charlie"
    }
  ]
},
{
  "edges" : [
    {
      "_id" : "knows/1083204277",
      "_rev" : "1083204277",
      "_key" : "1083204277",
      "_from" : "persons/eve",
      "_to" : "persons/alice"
    },
```

```
          {
            "_id" : "knows/1083400885",
            "_rev" : "1083400885",
            "_key" : "1083400885",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          },
          {
            "_id" : "knows/1083007669",
            "_rev" : "1083007669",
            "_key" : "1083007669",
            "_from" : "persons/bob",
            "_to" : "persons/dave"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1081500341",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1082352309",
            "_key" : "eve",
            "name" : "Eve"
          },
          {
            "_id" : "persons/bob",
            "_rev" : "1081696949",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "persons/dave",
            "_rev" : "1082155701",
            "_key" : "dave",
            "name" : "Dave"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1083204277",
            "_rev" : "1083204277",
            "_key" : "1083204277",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          },
          {
            "_id" : "knows/1083400885",
            "_rev" : "1083400885",
            "_key" : "1083400885",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          }
        ],
        "vertices" : [
```

```json
        {
          "_id" : "persons/alice",
          "_rev" : "1081500341",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1082352309",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1081696949",
          "_key" : "bob",
          "name" : "Bob"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1083204277",
          "_rev" : "1083204277",
          "_key" : "1083204277",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1081500341",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1082352309",
          "_key" : "eve",
          "name" : "Eve"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1082614453",
          "_rev" : "1082614453",
          "_key" : "1082614453",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1083400885",
          "_rev" : "1083400885",
          "_key" : "1083400885",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
```

```
        },
        {
          "_id" : "knows/1083204277",
          "_rev" : "1083204277",
          "_key" : "1083204277",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1081500341",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1081696949",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1082352309",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1081500341",
          "_key" : "alice",
          "name" : "Alice"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1082614453",
          "_rev" : "1082614453",
          "_key" : "1082614453",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1083400885",
          "_rev" : "1083400885",
          "_key" : "1083400885",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1081500341",
          "_key" : "alice",
          "name" : "Alice"
        },
```

```
      {
        "_id" : "persons/bob",
        "_rev" : "1081696949",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1082352309",
        "_key" : "eve",
        "name" : "Eve"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1082614453",
        "_rev" : "1082614453",
        "_key" : "1082614453",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      },
      {
        "_id" : "knows/1082811061",
        "_rev" : "1082811061",
        "_key" : "1082811061",
        "_from" : "persons/bob",
        "_to" : "persons/charlie"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1081500341",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1081696949",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/charlie",
        "_rev" : "1081959093",
        "_key" : "charlie",
        "name" : "Charlie"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1082614453",
        "_rev" : "1082614453",
        "_key" : "1082614453",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
```

```
      },
      {
        "_id" : "knows/1083007669",
        "_rev" : "1083007669",
        "_key" : "1083007669",
        "_from" : "persons/bob",
        "_to" : "persons/dave"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1081500341",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1081696949",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/dave",
        "_rev" : "1082155701",
        "_key" : "dave",
        "name" : "Dave"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1082614453",
        "_rev" : "1082614453",
        "_key" : "1082614453",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1081500341",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1081696949",
        "_key" : "bob",
        "name" : "Bob"
      }
    ]
  },
  {
    "edges" : [ ],
    "vertices" : [
      {
        "_id" : "persons/alice",
```

```
                        "_rev" : "1081500341",
                        "_key" : "alice",
                        "name" : "Alice"
                }
            ]
        }
    ]
    }
  },
  "error" : false,
  "code" : 200
}
```

Using *backward* item-ordering:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","direction":

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
            "_id" : "persons/alice",
            "_rev" : "1087660725",
            "_key" : "alice",
            "name" : "Alice"
        },
        {
            "_id" : "persons/bob",
            "_rev" : "1087857333",
            "_key" : "bob",
            "name" : "Bob"
        },
        {
            "_id" : "persons/dave",
            "_rev" : "1088316085",
            "_key" : "dave",
            "name" : "Dave"
        },
        {
            "_id" : "persons/charlie",
            "_rev" : "1088119477",
            "_key" : "charlie",
            "name" : "Charlie"
        },
        {
            "_id" : "persons/eve",
            "_rev" : "1088512693",
            "_key" : "eve",
            "name" : "Eve"
        },
        {
```

```
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1088512693",
        "_key" : "eve",
        "name" : "Eve"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1087857333",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/dave",
        "_rev" : "1088316085",
        "_key" : "dave",
        "name" : "Dave"
      },
      {
        "_id" : "persons/charlie",
        "_rev" : "1088119477",
        "_key" : "charlie",
        "name" : "Charlie"
      },
      {
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      }
    ],
    "paths" : [
      {
        "edges" : [ ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1087660725",
            "_key" : "alice",
            "name" : "Alice"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1088774837",
            "_rev" : "1088774837",
            "_key" : "1088774837",
            "_from" : "persons/alice",
            "_to" : "persons/bob"
          }
        ],
        "vertices" : [
          {
```

```
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1087857333",
        "_key" : "bob",
        "name" : "Bob"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1088774837",
        "_rev" : "1088774837",
        "_key" : "1088774837",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      },
      {
        "_id" : "knows/1089168053",
        "_rev" : "1089168053",
        "_key" : "1089168053",
        "_from" : "persons/bob",
        "_to" : "persons/dave"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1087857333",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/dave",
        "_rev" : "1088316085",
        "_key" : "dave",
        "name" : "Dave"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1088774837",
        "_rev" : "1088774837",
        "_key" : "1088774837",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      },
```

```
      {
        "_id" : "knows/1088971445",
        "_rev" : "1088971445",
        "_key" : "1088971445",
        "_from" : "persons/bob",
        "_to" : "persons/charlie"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1087857333",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
        "_id" : "persons/charlie",
        "_rev" : "1088119477",
        "_key" : "charlie",
        "name" : "Charlie"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1088774837",
        "_rev" : "1088774837",
        "_key" : "1088774837",
        "_from" : "persons/alice",
        "_to" : "persons/bob"
      },
      {
        "_id" : "knows/1089561269",
        "_rev" : "1089561269",
        "_key" : "1089561269",
        "_from" : "persons/eve",
        "_to" : "persons/bob"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1087857333",
        "_key" : "bob",
        "name" : "Bob"
      },
      {
```

```json
          "_id" : "persons/eve",
          "_rev" : "1088512693",
          "_key" : "eve",
          "name" : "Eve"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1088774837",
          "_rev" : "1088774837",
          "_key" : "1088774837",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1089561269",
          "_rev" : "1089561269",
          "_key" : "1089561269",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1089364661",
          "_rev" : "1089364661",
          "_key" : "1089364661",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1087660725",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1087857333",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1088512693",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1087660725",
          "_key" : "alice",
          "name" : "Alice"
        }
      ]
    },
    {
      "edges" : [
```

```
      {
        "_id" : "knows/1089364661",
        "_rev" : "1089364661",
        "_key" : "1089364661",
        "_from" : "persons/eve",
        "_to" : "persons/alice"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1088512693",
        "_key" : "eve",
        "name" : "Eve"
      }
    ]
  },
  {
    "edges" : [
      {
        "_id" : "knows/1089364661",
        "_rev" : "1089364661",
        "_key" : "1089364661",
        "_from" : "persons/eve",
        "_to" : "persons/alice"
      },
      {
        "_id" : "knows/1089561269",
        "_rev" : "1089561269",
        "_key" : "1089561269",
        "_from" : "persons/eve",
        "_to" : "persons/bob"
      }
    ],
    "vertices" : [
      {
        "_id" : "persons/alice",
        "_rev" : "1087660725",
        "_key" : "alice",
        "name" : "Alice"
      },
      {
        "_id" : "persons/eve",
        "_rev" : "1088512693",
        "_key" : "eve",
        "name" : "Eve"
      },
      {
        "_id" : "persons/bob",
        "_rev" : "1087857333",
        "_key" : "bob",
        "name" : "Bob"
      }
    ]
```

```
      },
      {
        "edges" : [
          {
            "_id" : "knows/1089364661",
            "_rev" : "1089364661",
            "_key" : "1089364661",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          },
          {
            "_id" : "knows/1089561269",
            "_rev" : "1089561269",
            "_key" : "1089561269",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          },
          {
            "_id" : "knows/1089168053",
            "_rev" : "1089168053",
            "_key" : "1089168053",
            "_from" : "persons/bob",
            "_to" : "persons/dave"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1087660725",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1088512693",
            "_key" : "eve",
            "name" : "Eve"
          },
          {
            "_id" : "persons/bob",
            "_rev" : "1087857333",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "persons/dave",
            "_rev" : "1088316085",
            "_key" : "dave",
            "name" : "Dave"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1089364661",
            "_rev" : "1089364661",
            "_key" : "1089364661",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
```

```
        },
        {
          "_id" : "knows/1089561269",
          "_rev" : "1089561269",
          "_key" : "1089561269",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1088971445",
          "_rev" : "1088971445",
          "_key" : "1088971445",
          "_from" : "persons/bob",
          "_to" : "persons/charlie"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1087660725",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1088512693",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1087857333",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1088119477",
          "_key" : "charlie",
          "name" : "Charlie"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1089364661",
          "_rev" : "1089364661",
          "_key" : "1089364661",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1089561269",
          "_rev" : "1089561269",
          "_key" : "1089561269",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
```

```
                      "_id" : "knows/1088774837",
                      "_rev" : "1088774837",
                      "_key" : "1088774837",
                      "_from" : "persons/alice",
                      "_to" : "persons/bob"
                    }
                ],
                "vertices" : [
                    {
                      "_id" : "persons/alice",
                      "_rev" : "1087660725",
                      "_key" : "alice",
                      "name" : "Alice"
                    },
                    {
                      "_id" : "persons/eve",
                      "_rev" : "1088512693",
                      "_key" : "eve",
                      "name" : "Eve"
                    },
                    {
                      "_id" : "persons/bob",
                      "_rev" : "1087857333",
                      "_key" : "bob",
                      "name" : "Bob"
                    },
                    {
                      "_id" : "persons/alice",
                      "_rev" : "1087660725",
                      "_key" : "alice",
                      "name" : "Alice"
                    }
                ]
            }
        ]
      }
  },
  "error" : false,
  "code" : 200
}
```

Edges should only be included once globally, but nodes are included every time they are visited:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","direction":
{"vertices":"none","edges":"global"}}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "visited" : {
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1093821109",
```

```
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1094673077",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1094017717",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1093821109",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/charlie",
          "_rev" : "1094279861",
          "_key" : "charlie",
          "name" : "Charlie"
        },
        {
          "_id" : "persons/dave",
          "_rev" : "1094476469",
          "_key" : "dave",
          "name" : "Dave"
        }
      ],
      "paths" : [
        {
          "edges" : [ ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1093821109",
              "_key" : "alice",
              "name" : "Alice"
            }
          ]
        },
        {
          "edges" : [
            {
              "_id" : "knows/1095525045",
              "_rev" : "1095525045",
              "_key" : "1095525045",
              "_from" : "persons/eve",
              "_to" : "persons/alice"
            }
          ],
          "vertices" : [
            {
              "_id" : "persons/alice",
              "_rev" : "1093821109",
```

```
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1094673077",
          "_key" : "eve",
          "name" : "Eve"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1095525045",
          "_rev" : "1095525045",
          "_key" : "1095525045",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1095721653",
          "_rev" : "1095721653",
          "_key" : "1095721653",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1093821109",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1094673077",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1094017717",
          "_key" : "bob",
          "name" : "Bob"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1095525045",
          "_rev" : "1095525045",
          "_key" : "1095525045",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1095721653",
```

```
          "_rev" : "1095721653",
          "_key" : "1095721653",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1094935221",
          "_rev" : "1094935221",
          "_key" : "1094935221",
          "_from" : "persons/alice",
          "_to" : "persons/bob"
        }
      ],
      "vertices" : [
        {
          "_id" : "persons/alice",
          "_rev" : "1093821109",
          "_key" : "alice",
          "name" : "Alice"
        },
        {
          "_id" : "persons/eve",
          "_rev" : "1094673077",
          "_key" : "eve",
          "name" : "Eve"
        },
        {
          "_id" : "persons/bob",
          "_rev" : "1094017717",
          "_key" : "bob",
          "name" : "Bob"
        },
        {
          "_id" : "persons/alice",
          "_rev" : "1093821109",
          "_key" : "alice",
          "name" : "Alice"
        }
      ]
    },
    {
      "edges" : [
        {
          "_id" : "knows/1095525045",
          "_rev" : "1095525045",
          "_key" : "1095525045",
          "_from" : "persons/eve",
          "_to" : "persons/alice"
        },
        {
          "_id" : "knows/1095721653",
          "_rev" : "1095721653",
          "_key" : "1095721653",
          "_from" : "persons/eve",
          "_to" : "persons/bob"
        },
        {
          "_id" : "knows/1095131829",
          "_rev" : "1095131829",
          "_key" : "1095131829",
```

```
            "_from" : "persons/bob",
            "_to" : "persons/charlie"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1093821109",
            "_key" : "alice",
            "name" : "Alice"
          },
          {
            "_id" : "persons/eve",
            "_rev" : "1094673077",
            "_key" : "eve",
            "name" : "Eve"
          },
          {
            "_id" : "persons/bob",
            "_rev" : "1094017717",
            "_key" : "bob",
            "name" : "Bob"
          },
          {
            "_id" : "persons/charlie",
            "_rev" : "1094279861",
            "_key" : "charlie",
            "name" : "Charlie"
          }
        ]
      },
      {
        "edges" : [
          {
            "_id" : "knows/1095525045",
            "_rev" : "1095525045",
            "_key" : "1095525045",
            "_from" : "persons/eve",
            "_to" : "persons/alice"
          },
          {
            "_id" : "knows/1095721653",
            "_rev" : "1095721653",
            "_key" : "1095721653",
            "_from" : "persons/eve",
            "_to" : "persons/bob"
          },
          {
            "_id" : "knows/1095328437",
            "_rev" : "1095328437",
            "_key" : "1095328437",
            "_from" : "persons/bob",
            "_to" : "persons/dave"
          }
        ],
        "vertices" : [
          {
            "_id" : "persons/alice",
            "_rev" : "1093821109",
            "_key" : "alice",
```

```
                "name" : "Alice"
              },
              {
                "_id" : "persons/eve",
                "_rev" : "1094673077",
                "_key" : "eve",
                "name" : "Eve"
              },
              {
                "_id" : "persons/bob",
                "_rev" : "1094017717",
                "_key" : "bob",
                "name" : "Bob"
              },
              {
                "_id" : "persons/dave",
                "_rev" : "1094476469",
                "_key" : "dave",
                "name" : "Dave"
              }
            ]
          }
        ]
      }
    },
    "error" : false,
    "code" : 200
}
```

If the underlying graph is cyclic, *maxIterations* should be set:

The underlying graph has two vertices *Alice* and *Bob*. With the directed edges:

- *Alice* knows *Bob* _ *Bob* knows *Alice*

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/traversal
{"startVertex":"persons/alice","graphName":"knows_graph","direction":
{"vertices":"none","edges":"none"},"maxIterations":5}

HTTP/1.1 500 Internal Error
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 500,
  "errorNum" : 1909,
  "errorMessage" : "too many iterations"
}
```

HTTP Interface for Replication

Replication

This is an introduction to ArangoDB's HTTP replication interface. The replication architecture and components are described in more details in Replication.

The HTTP replication interface serves four main purposes:

- fetch initial data from a server (e.g. for a backup, or for the initial synchronization of data before starting the continuous replication applier)
- administer the replication logger (starting, stopping, querying state)
- fetch the changelog from a server (used for incremental synchronization of changes)
- administer the replication applier (starting, stopping, configuring, querying state)

Please note that all replication operations work on a database level. If an ArangoDB server contains more than one database, the replication system must be configured individually per database, and replicating the data of multiple databases will require multiple operations.

Replication Dump Commands

The *inventory* method can be used to query an ArangoDB database's current set of collections plus their indexes. Clients can use this method to get an overview of which collections are present in the database. They can use this information to either start a full or a partial synchronization of data, e.g. to initiate a backup or the incremental data synchronization.

returns the server inventory

Return inventory of collections and indexes

GET /_api/replication/inventory

Query Parameters

- *includeSystem*: Include system collections in the result. The default value is *false*.

Returns the list of collections and indexes available on the server. This list can be used by replication clients to initiate an initial sync with the server.

The response will contain a JSON hash array with the *collection* and *state* attributes.

collections is a list of collections with the following sub-attributes:

- *parameters*: the collection properties

- *indexes*: a list of the indexes of a the collection. Primary indexes and edges indexes are not included in this list.

tick: the system-wide tick value at the start of the dump

The *state* attribute contains the current state of the replication logger. It contains the following sub-attributes:

- *running*: whether or not the replication logger is currently active

- *lastLogTick*: the value of the last tick the replication logger has written

- *time*: the current time on the server

Replication clients should note the *lastLogTick* value returned. They can then fetch collections' data using the dump method up to the value of lastLogTick, and query the continuous replication log for log events after this tick value.

To create a full copy of the collections on the logger server, a replication client can execute these steps:

- call the */inventory* API method. This returns the *lastLogTick* value and the list of collections and indexes from the logger server.

- for each collection returned by */inventory*, create the collection locally and call */dump* to stream the collection data to the client, up to the value of *lastLogTick*. After that, the client can create the indexes on the collections as they were reported by */inventory*.

If the clients wants to continuously stream replication log events from the logger server, the following additional steps need to be carried out:

- the client should call */logger-follow* initially to fetch the first batch of replication events that were logged after the client's call to */inventory*.

The call to */logger-follow* should use a *from* parameter with the value of the *lastLogTick* as reported by

/inventory. The call to */logger-follow* will return the *x-arango-replication-lastincluded* which will contain the last tick value included in the response.

- the client can then continuously call */logger-follow* to incrementally fetch new replication events that occurred after the last transfer.

Calls should use a *from* parameter with the value of the *x-arango-replication-lastincluded* header of the previous response. If there are no more replication events, the response will be empty and clients can go to sleep for a while and try again later.

Note: on a coordinator, this request must have the URL parameter *DBserver* which must be an ID of a DBserver. The very same request is forwarded synchronously to that DBserver. It is an error if this attribute is not bound in the coordinator case.

Return Codes

- *200*: is returned if the request was executed successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/inventory
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "collections" : [
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "803758773",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "animals",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "616456885",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "better-example",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
```

```
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "711287477",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "Company",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "711025333",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "Customer",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "802841269",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "demo",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "711549621",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "Electronics",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
```

```
      "type" : 2,
      "cid" : "1199858357",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "female",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1109942965",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "frenchCity",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1109680821",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "germanCity",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "711418549",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "Groceries",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1199989429",
      "deleted" : false,
      "doCompact" : true,
```

```json
          "maximalSize" : 1048576,
          "name" : "male",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "1135305397",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "otherVertices",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "912679605",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "vertices1",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 3,
          "cid" : "914121397",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "edges2",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 3,
          "cid" : "1110074037",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "frenchHighway",
          "isVolatile" : false,
          "waitForSync" : false
```

```
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 3,
        "cid" : "711156405",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "friend_of",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 3,
        "cid" : "1109811893",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "germanHighway",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 3,
        "cid" : "711680693",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "has_bought",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 3,
        "cid" : "1110205109",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "internationalHighway",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
```

```
        "parameters" : {
          "version" : 5,
          "type" : 3,
          "cid" : "1200120501",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "relation",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      }
    ],
    "state" : {
      "running" : true,
      "lastLogTick" : "1238655669",
      "totalEvents" : 7636,
      "time" : "2014-07-02T15:27:01Z"
    },
    "tick" : "1238655669"
}
```

With some additional indexes:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/inventory
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "collections" : [
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "803758773",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "animals",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 2,
        "cid" : "616456885",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "better-example",
        "isVolatile" : false,
```

```
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "711287477",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "Company",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "711025333",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "Customer",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "802841269",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "demo",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "711549621",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "Electronics",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
```

```json
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1199858357",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "female",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1109942965",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "frenchCity",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1109680821",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "germanCity",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "711418549",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "Groceries",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
```

```
      "cid" : "1238721205",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "IndexedCollection1",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [
      {
        "id" : "1238983349",
        "type" : "hash",
        "unique" : false,
        "fields" : [
          "name"
        ]
      },
      {
        "id" : "1239311029",
        "type" : "skiplist",
        "unique" : true,
        "fields" : [
          "a",
          "b"
        ]
      },
      {
        "id" : "1239507637",
        "type" : "cap",
        "unique" : false,
        "size" : 500,
        "byteSize" : 0
      }
    ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 2,
      "cid" : "1239704245",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "IndexedCollection2",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [
      {
        "id" : "1239966389",
        "type" : "fulltext",
        "unique" : false,
        "minLength" : 10,
        "fields" : [
          "text"
        ]
      },
      {
        "id" : "1240228533",
        "type" : "skiplist",
```

```
            "unique" : false,
            "fields" : [
              "a"
            ]
          },
          {
            "id" : "1240425141",
            "type" : "cap",
            "unique" : false,
            "size" : 0,
            "byteSize" : 1048576
          }
        ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "1199989429",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "male",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "1135305397",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "otherVertices",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 2,
          "cid" : "912679605",
          "deleted" : false,
          "doCompact" : true,
          "maximalSize" : 1048576,
          "name" : "vertices1",
          "isVolatile" : false,
          "waitForSync" : false
        },
        "indexes" : [ ]
      },
      {
        "parameters" : {
          "version" : 5,
          "type" : 3,
```

```json
      "cid" : "914121397",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "edges2",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 3,
      "cid" : "1110074037",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "frenchHighway",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 3,
      "cid" : "711156405",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "friend_of",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 3,
      "cid" : "1109811893",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
      "name" : "germanHighway",
      "isVolatile" : false,
      "waitForSync" : false
    },
    "indexes" : [ ]
  },
  {
    "parameters" : {
      "version" : 5,
      "type" : 3,
      "cid" : "711680693",
      "deleted" : false,
      "doCompact" : true,
      "maximalSize" : 1048576,
```

```
        "name" : "has_bought",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 3,
        "cid" : "1110205109",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "internationalHighway",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    },
    {
      "parameters" : {
        "version" : 5,
        "type" : 3,
        "cid" : "1200120501",
        "deleted" : false,
        "doCompact" : true,
        "maximalSize" : 1048576,
        "name" : "relation",
        "isVolatile" : false,
        "waitForSync" : false
      },
      "indexes" : [ ]
    }
  ],
  "state" : {
    "running" : true,
    "lastLogTick" : "1238786741",
    "totalEvents" : 7649,
    "time" : "2014-07-02T15:27:01Z"
  },
  "tick" : "1240556213"
}
```

The *dump* method can be used to fetch data from a specific collection. As the results of the dump command can be huge, *dump* may not return all data from a collection at once. Instead, the dump command may be called repeatedly by replication clients until there is no more data to fetch. The dump command will not only return the current documents in the collection, but also document updates and deletions.

To get to an identical state of data, replication clients should apply the individual parts of the dump results in the same order as they are served to them.

dumps the data of a collection

Return data of a collection

```
GET /_api/replication/dump
```

Query Parameters

- *collection*: The name or id of the collection to dump.

- *from*: Lower bound tick value for results.

- *to*: Upper bound tick value for results.

- *chunkSize*: Approximate maximum size of the returned result.

- *ticks*: Whether or not to include tick values in the dump. Default value is *true*.

Returns the data from the collection for the requested range.

When the *from* URL parameter is not used, collection events are returned from the beginning. When the *from* parameter is used, the result will only contain collection entries which have higher tick values than the specified *from* value (note: the log entry with a tick value equal to *from* will be excluded).

The *to* URL parameter can be used to optionally restrict the upper bound of the result to a certain tick value. If used, the result will only contain collection entries with tick values up to (including) *to*.

The *chunkSize* URL parameter can be used to control the size of the result. It must be specified in bytes. The *chunkSize* value will only be honored approximately. Otherwise a too low *chunkSize* value could cause the server to not be able to put just one entry into the result and return it. Therefore, the *chunkSize* value will only be consulted after an entry has been written into the result. If the result size is then bigger than *chunkSize*, the server will respond with as many entries as there are in the response already. If the result size is still smaller than *chunkSize*, the server will try to return more data if there's more data left to return.

If *chunkSize* is not specified, some server-side default value will be used.

The *Content-Type* of the result is *application/x-arango-dump*. This is an easy-to-process format, with all entries going onto separate lines in the response body.

Each line itself is a JSON hash, with at least the following attributes:

- *type*: the type of entry. Possible values for *type* are:

- 2300: document insertion/update

- 2301: edge insertion/update

- 2302: document/edge deletion

- *key*: the key of the document/edge or the key used in the deletion operation

- *rev*: the revision id of the document/edge or the deletion operation

- *data*: the actual document/edge data for types 2300 and 2301. The full document/edge data will be returned even for updates.

A more detailed description of the different entry types and their data structures can be found in @ref RefManualReplicationEventTypes.

Note: there will be no distinction between inserts and updates when calling this method.

Return Codes

- *200*: is returned if the request was executed successfully.

- *400*: is returned if either the *from* or *to* values are invalid.

- *404*: is returned when the collection could not be found.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

Empty collection:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/dump?
collection=testCollection
undefined

HTTP/1.1 204 No Content
content-type: application/x-arango-dump; charset=utf-8
x-arango-replication-checkmore: false
x-arango-replication-lastincluded: 0
```

Non-empty collection:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/dump?
collection=testCollection
undefined

HTTP/1.1 200 OK
content-type: application/x-arango-dump; charset=utf-8
x-arango-replication-checkmore: false
x-arango-replication-lastincluded: 1242981045

"
{\"tick\":\"1242325685\",\"type\":2300,\"key\":\"123456\",\"rev\":\"1
{\"_key\":\"123456\",\"_rev\":\"1242260149\",\"c\":false,\"b\":1,\"d\
value\"}}
{\"tick\":\"1242784437\",\"type\":2302,\"key\":\"foobar\",\"rev\":\"1

{\"tick\":\"1242981045\",\"type\":2302,\"key\":\"abcdef\",\"rev\":\"1

"
```

The *sync* method can be used by replication clients to connect an ArangoDB database to a remote endpoint, fetch the remote list of collections and indexes, and collection data. It will thus create a local backup of the state of data at the remote ArangoDB database. *sync* works on a database level.

sync will first fetch the list of collections and indexes from the remote endpoint. It does so by calling the *inventory* API of the remote database. It will then purge data in the local ArangoDB database, and after start will transfer collection data from the remote database to the local ArangoDB database. It will extract data from the remote database by calling the remote database's *dump* API until all data are fetched.

As mentioned, *sync* will remove data from the local instance, and thus must be handled with caution.

synchronizes data from a remote endpoint

Synchronize data from a remote endpoint

```
PUT /_api/replication/sync
```

- *configuration*: A JSON representation of the configuration.

Starts a full data synchronization from a remote endpoint into the local ArangoDB database.

The body of the request must be JSON hash with the configuration. The following attributes are allowed for the configuration:

- *endpoint*: the endpoint to connect to (e.g. "tcp://192.168.173.13:8529").

- *database*: the database name on the master (if not specified, defaults to the name of the local current database).

- *username*: an optional ArangoDB username to use when connecting to the endpoint.

- *password*: the password to use when connecting to the endpoint.

- *restrictType*: an optional string value for collection filtering. When specified, the allowed values are *include* or *exclude*.

- *restrictCollections*: an optional list of collections for use with *restrictType*. If *restrictType* is *include*, only the specified collections will be sychronised. If *restrictType* is *exclude*, all but the specified collections will be synchronized.

In case of success, the body of the response is a JSON hash with the following attributes:

- *collections*: a list of collections that were transferred from the endpoint

- *lastLogTick*: the last log tick on the endpoint at the time the transfer was started. Use this value as the *from* value when starting the continuous synchronization later.

WARNING: calling this method will sychronise data from the collections found on the remote endpoint to the local ArangoDB database. All data in the local collections will be purged and replaced with data from the endpoint.

Use with caution!

Note: this method is not supported on a coordinator in a cluster.

Return Codes

- *200*: is returned if the request was executed successfully.

- *400*: is returned if the configuration is incomplete or malformed.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred during sychronisation.

- *501*: is returned when this operation is called on a coordinator in a cluster.

Replication Logger Commands

The logger commands allow starting, starting, and fetching the current state of a database's replication logger.

get the configuration of the replication logger

Configuration of the replication logger

GET /_api/replication/logger-config

Returns the configuration of the replication logger.

The body of the response is a JSON hash with the configuration. The following attributes may be present in the configuration:

- *autoStart*: whether or not to automatically start the replication logger on server startup

- *logRemoteChanges*: whether or not externally created changes should be logged by the local logger

- *maxEvents*: the maximum number of log events kept by the replication logger before deleting oldest events. A value of *0* means that the number of events is not restricted.

- *maxEventsSize*: the maximum cumulated size of log event data kept by the replication logger before deleting oldest events. A value of *0* means that the cumulated size of events is not restricted.

Return Codes

- *200*: is returned if the request was executed successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-config
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "autoStart" : true,
  "logRemoteChanges" : true,
  "maxEvents" : 0,
  "maxEventsSize" : 0
}
```

set the configuration of the replication logger

Configuration of the replication logger

PUT /_api/replication/logger-config

- *configuration*: A JSON representation of the configuration.

Synchronizationn logger.

The body of the request must be JSON hash with the configuration. The following attributes are allowed for the configuration:

- *autoStart*: whether or not to automatically start the replication logger on server startup

- *logRemoteChanges*: whether or not externally created changes should be logged by the local logger

- *maxEvents*: the maximum number of log events kept by the replication logger before deleting oldest events. Use a value of *0* to not restrict the number of events.

- *maxEventsSize*: the maximum cumulated size of log event data kept by the replication logger before deleting oldest events. Use a value of *0* to not restrict the size.

Note: When setting both *maxEvents* and *maxEventsSize*, reaching either limitation will trigger a deletion of the "oldest" log events from the replication log.

In case of success, the body of the response is a JSON hash with the updated configuration.

Return Codes

- *200*: is returned if the request was executed successfully.

- *400*: is returned if the configuration is incomplete or malformed.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-config
{"logRemoteChanges":true,"maxEvents":1048576}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "autoStart" : true,
  "logRemoteChanges" : true,
  "maxEvents" : 0,
  "maxEventsSize" : 0
}
```

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-config
{"logRemoteChanges":false,"maxEvents":16384,"maxEventsSize":16777216}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "autoStart" : true,
```

```
    "logRemoteChanges" : true,
    "maxEvents" : 0,
    "maxEventsSize" : 0
}
```

starts the replication logger

Start replication logger

`PUT /_api/replication/logger-start`

Starts the server's replication logger. Will do nothing if the replication logger is already running.

The body of the response contains a JSON object with the following attributes:

- *running*: will contain *true*

Note: this method is not supported on a coordinator in a cluster.

Return Codes

- *200*: is returned if the logger was started successfully, or was already running.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if the logger could not be started.

- *501*: is returned when this operation is called on a coordinator in a cluster.

Examples

Starts the replication logger.

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/replication/logger-start

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
    "running" : true
}
```

stops the replication logger

Stop replication logger

`PUT /_api/replication/logger-stop`

Stops the server's replication logger. Will do nothing if the replication logger is not running.

The body of the response contains a JSON object with the following attributes:

- *running*: will contain *false*

Note: this method is not supported on a coordinator in a cluster.

Return Codes

- *200*: is returned if the logger was stopped successfully, or was not running before.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if the logger could not be stopped.

- *501*: is returned when this operation is called on a coordinator in a cluster.

Examples

Starts the replication logger.

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/replication/logger-stop

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "running" : true
}
```

returns the state of the replication logger

Return replication logger state

GET /_api/replication/logger-state

Returns the current state of the server's replication logger. The state will include information about whether the logger is running and about the last logged tick value. This tick value is important for incremental fetching of data.

The state API can be called regardless of whether the logger is currently running or not.

The body of the response contains a JSON object with the following attributes:

- *state*: the current logger state as a JSON hash array with the following sub-attributes:

- *running*: whether or not the logger is running

- *lastLogTick*: the tick value of the latest tick the logger has logged. This value can be used for incremental fetching of log data.

- *totalEvents*: total number of events logged since the server was started. The value is not reset between multiple stops and re-starts of the logger.

- *time*: the current date and time on the logger server

- *server*: a JSON hash with the following sub-attributes:

- *version*: the logger server's version

- *serverId*: the logger server's id

- *clients*: a list of all replication clients that ever connected to the logger since it was started. This list can be used to determine approximately how much data the individual clients have already fetched from the logger server. Each entry in the list contains a *time* value indicating the server time the client last fetched data from the replication logger. The *lastServedTick* value of each client indicates the latest tick value sent to the client upon a client request to the replication logger.

Return Codes

- *200*: is returned if the logger state could be determined successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if the logger state could not be determined.

Examples

Returns the state of an inactive replication logger.

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-state
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "state" : {
    "running" : true,
    "lastLogTick" : "1235313333",
    "totalEvents" : 7611,
    "time" : "2014-07-02T15:26:59Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "141206089767669"
  },
  "clients" : [ ]
}
```

Returns the state of an active replication logger.

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-state
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "state" : {
    "running" : true,
    "lastLogTick" : "1235313333",
    "totalEvents" : 7611,
    "time" : "2014-07-02T15:26:59Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "141206089767669"
  },
  "clients" : [ ]
}
```

To query the latest changes logged by the replication logger, the HTTP interface also provides the

```
logger-follow.
```

This method should be used by replication clients to incrementally fetch updates from an ArangoDB database.

returns ranged data from the replication log

Returns log entries

```
GET /_api/replication/logger-follow
```

Query Parameters

- *from*: Lower bound tick value for results.

- *to*: Upper bound tick value for results.

- *chunkSize*: Approximate maximum size of the returned result.

Returns data from the server's replication log. This method can be called by replication clients after an initial synchronization of data. The method will return all "recent" log entries from the logger server, and the clients can replay and apply these entries locally so they get to the same data state as the logger server.

Clients can call this method repeatedly to incrementally fetch all changes from the logger server. In this case, they should provide the *from* value so they will only get returned the log events since their last fetch.

When the *from* URL parameter is not used, the logger server will return log entries starting at the beginning of its replication log. When the *from* parameter is used, the logger server will only return log entries which have higher tick values than the specified *from* value (note: the log entry with a tick value equal to *from* will be excluded). Use the *from* value when incrementally fetching log data.

The *to* URL parameter can be used to optionally restrict the upper bound of the result to a certain tick value. If used, the result will contain only log events with tick values up to (including) *to*. In incremental fetching, there is no need to use the *to* parameter. It only makes sense in special situations, when only parts of the change log are required.

The *chunkSize* URL parameter can be used to control the size of the result. It must be specified in bytes. The *chunkSize* value will only be honored approximately. Otherwise a too low *chunkSize* value could cause the server to not be able to put just one log entry into the result and return it. Therefore, the *chunkSize* value will only be consulted after a log entry has been written into the result. If the result size is then bigger than *chunkSize*, the server will respond with as many log entries as there are in the response already. If the result size is still smaller than *chunkSize*, the server will try to return more data if there's more data left to return.

If *chunkSize* is not specified, some server-side default value will be used.

The *Content-Type* of the result is *application/x-arango-dump*. This is an easy-to-process format, with all log events going onto separate lines in the response body. Each log event itself is a JSON hash, with at least the following attributes:

- *tick*: the log event tick value

- *type*: the log event type

Individual log events will also have additional attributes, depending on the event type. A few common attributes which are used for multiple events types are:

- *cid*: id of the collection the event was for

- *tid*: id of the transaction the event was contained in

- *key*: document key

- *rev*: document revision id

- *data*: the original document data

A more detailed description of the individual replication event types and their data structures can be found in @ref RefManualReplicationEventTypes.

The response will also contain the following HTTP headers:

- *x-arango-replication-active*: whether or not the logger is active. Clients can use this flag as an indication for their polling frequency. If the logger is not active and there are no more replication events available, it might be sensible for a client to abort, or to go to sleep for a long time and try again later to check whether the logger has been activated.

- *x-arango-replication-lastincluded*: the tick value of the last included value in the result. In incremental log fetching, this value can be used as the *from* value for the following request. **Note** that if the result is empty, the value will be *0*. This value should not be used as *from* value by clients in the next request (otherwise the server would return the log events from the start of the log again).

- *x-arango-replication-lasttick*: the last tick value the logger server has logged (not necessarily included in the result). By comparing the the last tick and last included tick values, clients have an approximate indication of how many events there are still left to fetch.

- *x-arango-replication-checkmore*: whether or not there already exists more log data which the client could fetch immediately. If there is more log data available, the client could call *logger-follow* again with an adjusted *from* value to fetch remaining log entries until there are no more.

If there isn't any more log data to fetch, the client might decide to go to sleep for a while before calling the logger again.

Note: this method is not supported on a coordinator in a cluster.

Return Codes

- *200*: is returned if the request was executed successfully, and there are log events available for the requested range. The response body will not be empty in this case.

- *204*: is returned if the request was executed successfully, but there are no log events available for the requested range. The response body will be empty in this case.

- *400*: is returned if either the *from* or *to* values are invalid.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

- *501*: is returned when this operation is called on a coordinator in a cluster.

Examples

No log events available:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-follow?
from=1235313333
undefined

HTTP/1.1 204 No Content
```

```
content-type: application/x-arango-dump; charset=utf-8
x-arango-replication-active: true
x-arango-replication-checkmore: false
x-arango-replication-lastincluded: 0
x-arango-replication-lasttick: 1235313333
```

A few log events:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-follow?
from=1235313333
undefined

HTTP/1.1 200 OK
content-type: application/x-arango-dump; charset=utf-8
x-arango-replication-active: true
x-arango-replication-checkmore: false
x-arango-replication-lastincluded: 1236558517
x-arango-replication-lasttick: 1236558517

"
{\"tick\":\"1235444405\",\"type\":2000,\"database\":\"90805\",\"cid\"
{\"version\":5,\"type\":2,\"cid\":\"1235378869\",\"deleted\":false,\"

{\"tick\":\"1235772085\",\"type\":2300,\"database\":\"90805\",\"cid\"
{\"_key\":\"p1\",\"_rev\":\"1235706549\",\"name\":\"flux
compensator\"}}
{\"tick\":\"1236099765\",\"type\":2300,\"database\":\"90805\",\"cid\"
{\"_key\":\"p2\",\"_rev\":\"1236034229\",\"hp\":5100,\"name\":\"hybri
hovercraft\"}}
{\"tick\":\"1236296373\",\"type\":2302,\"database\":\"90805\",\"cid\"

{\"tick\":\"1236492981\",\"type\":2300,\"database\":\"90805\",\"cid\"
{\"_key\":\"p2\",\"_rev\":\"1236427445\",\"hp\":5100,\"name\":\"broke
hovercraft\"}}
{\"tick\":\"1236558517\",\"type\":2001,\"database\":\"90805\",\"cid\"

"
```

More events than would fit into the response:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/logger-follow?
from=1236558517&chunkSize=400
undefined

HTTP/1.1 200 OK
content-type: application/x-arango-dump; charset=utf-8
x-arango-replication-active: true
x-arango-replication-checkmore: true
x-arango-replication-lastincluded: 1237017269
x-arango-replication-lasttick: 1238655669

"
{\"tick\":\"1236689589\",\"type\":2000,\"database\":\"90805\",\"cid\"
{\"version\":5,\"type\":2,\"cid\":\"1236624053\",\"deleted\":false,\"
```

{\"tick\":\"1237017269\",\"type\":2300,\"database\":\"90805\",\"cid\"
{\"_key\":\"p1\",\"_rev\":\"1236951733\",\"name\":\"flux
compensator\"}}
"

Replication Applier Commands

The applier commands allow to remotely start, stop, and query the state and configuration of an ArangoDB database's replication applier.

get the configuration of the replication applier

Return configuration of replication applier

`GET /_api/replication/applier-config`

Returns the configuration of the replication applier.

The body of the response is a JSON hash with the configuration. The following attributes may be present in the configuration:

- *endpoint*: the logger server to connect to (e.g. "tcp://192.168.173.13:8529").

- *database*: the name of the database to connect to (e.g. "_system").

- *username*: an optional ArangoDB username to use when connecting to the endpoint.

- *password*: the password to use when connecting to the endpoint.

- *maxConnectRetries*: the maximum number of connection attempts the applier will make in a row. If the applier cannot establish a connection to the endpoint in this number of attempts, it will stop itself.

- *connectTimeout*: the timeout (in seconds) when attempting to connect to the endpoint. This value is used for each connection attempt.

- *requestTimeout*: the timeout (in seconds) for individual requests to the endpoint.

- *chunkSize*: the requested maximum size for log transfer packets that is used when the endpoint is contacted.

- *autoStart*: whether or not to auto-start the replication applier on (next and following) server starts

- *adaptivePolling*: whether or not the replication applier will use adaptive polling.

Return Codes

- *200*: is returned if the request was executed successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/applier-config
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "requestTimeout" : 300,
```

```
    "connectTimeout" : 10,
    "maxConnectRetries" : 100,
    "sslProtocol" : 0,
    "chunkSize" : 0,
    "autoStart" : false,
    "adaptivePolling" : true
}
```

set the configuration of the replication applier

Adjust configuration of replication applier

`PUT /_api/replication/applier-config`

- *configuration*: A JSON representation of the configuration.

Sets the configuration of the replication applier. The configuration can only be changed while the applier is not running. The updated configuration will be saved immediately but only become active with the next start of the applier.

The body of the request must be JSON hash with the configuration. The following attributes are allowed for the configuration:

- *endpoint*: the logger server to connect to (e.g. "tcp://192.168.173.13:8529"). The endpoint must be specified.

- *database*: the name of the database on the endpoint. If not specified, defaults to the current local database name.

- *username*: an optional ArangoDB username to use when connecting to the endpoint.

- *password*: the password to use when connecting to the endpoint.

- *maxConnectRetries*: the maximum number of connection attempts the applier will make in a row. If the applier cannot establish a connection to the endpoint in this number of attempts, it will stop itself.

- *connectTimeout*: the timeout (in seconds) when attempting to connect to the endpoint. This value is used for each connection attempt.

- *requestTimeout*: the timeout (in seconds) for individual requests to the endpoint.

- *chunkSize*: the requested maximum size for log transfer packets that is used when the endpoint is contacted.

- *autoStart*: whether or not to auto-start the replication applier on (next and following) server starts

- *adaptivePolling*: if set to *true*, the replication applier will fall to sleep for an increasingly long period in case the logger server at the endpoint does not have any more replication events to apply. Using adaptive polling is thus useful to reduce the amount of work for both the applier and the logger server for cases when there are only infrequent changes. The downside is that when using adaptive polling, it might take longer for the replication applier to detect that there are new replication events on the logger server.

Setting *adaptivePolling* to false will make the replication applier contact the logger server in a constant interval, regardless of whether the logger server provides updates frequently or seldomly.

In case of success, the body of the response is a JSON hash with the updated configuration.

Return Codes

- *200*: is returned if the request was executed successfully.

- *400*: is returned if the configuration is incomplete or malformed, or if the replication applier is currently running.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/replication/applier-config
{"endpoint":"tcp://127.0.0.1:8529","username":"replicationApplier","p


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "endpoint" : "tcp://127.0.0.1:8529",
  "database" : "_system",
  "username" : "replicationApplier",
  "requestTimeout" : 300,
  "connectTimeout" : 10,
  "maxConnectRetries" : 100,
  "sslProtocol" : 0,
  "chunkSize" : 4194304,
  "autoStart" : false,
  "adaptivePolling" : true
}
```

start the replication applier

Start replication applier

```
PUT /_api/replication/applier-start
```

Query Parameters

- *from*: The remote *lastLogTick* value from which to start applying. If not specified, the last saved tick from the previous applier run is used. If there is no previous applier state saved, the applier will start at the beginning of the logger server's log.

Starts the replication applier. This will return immediately if the replication applier is already running.

If the replication applier is not already running, the applier configuration will be checked, and if it is complete, the applier will be started in a background thread. This means that even if the applier will encounter any errors while running, they will not be reported in the response to this method.

To detect replication applier errors after the applier was started, use the */_api/replication/applier-state* API instead.

Return Codes

- *200*: is returned if the request was executed successfully.

- *400*: is returned if the replication applier is not fully configured or the configuration is invalid.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/replication/applier-start

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "state" : {
    "running" : true,
    "lastAppliedContinuousTick" : null,
    "lastProcessedContinuousTick" : null,
    "lastAvailableContinuousTick" : null,
    "progress" : {
      "time" : "2014-07-02T15:26:31Z",
      "message" : "applier created",
      "failedConnects" : 0
    },
    "totalRequests" : 0,
    "totalFailedConnects" : 0,
    "totalEvents" : 0,
    "lastError" : {
      "errorNum" : 0
    },
    "time" : "2014-07-02T15:27:03Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "141206089767669"
  },
  "endpoint" : "tcp://127.0.0.1:8529",
  "database" : "_system"
}
```

stops the replication applier

Stop replication applier

```
PUT /_api/replication/applier-stop
```

Stops the replication applier. This will return immediately if the replication applier is not running.

Return Codes

- *200*: is returned if the request was executed successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl -X PUT --dump -
```

```
http://localhost:8529/_api/replication/applier-stop

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "state" : {
    "running" : false,
    "lastAppliedContinuousTick" : null,
    "lastProcessedContinuousTick" : null,
    "lastAvailableContinuousTick" : null,
    "progress" : {
      "time" : "2014-07-02T15:27:03Z",
      "message" : "fetching master state information",
      "failedConnects" : 0
    },
    "totalRequests" : 0,
    "totalFailedConnects" : 0,
    "totalEvents" : 0,
    "lastError" : {
      "time" : "2014-07-02T15:27:03Z",
      "errorMessage" : "got incompatible master version from
endpoint 'tcp://127.0.0.1:8529': '2.0.2'",
      "errorNum" : 1403
    },
    "time" : "2014-07-02T15:27:03Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "141206089767669"
  },
  "endpoint" : "tcp://127.0.0.1:8529",
  "database" : "_system"
}
```

returns the state of the replication applier

State of the replication applier

GET /_api/replication/applier-state

Returns the state of the replication applier, regardless of whether the applier is currently running or not.

The response is a JSON hash with the following attributes:

- *state*: a JSON hash with the following sub-attributes:

- *running*: whether or not the applier is active and running

- *lastAppliedContinuousTick*: the last tick value from the continuous replication log the applier has applied.

- *lastProcessedContinuousTick*: the last tick value from the continuous replication log the applier has processed.

Regularly, the last applied and last processed tick values should be identical. For transactional operations, the replication applier will first process incoming log events before applying them, so the processed tick value might be higher than the applied tick value. This will be the case until the applier encounters the *transaction commit* log event for the transaction.

- *lastAvailableContinuousTick*: the last tick value the logger server can provide.

- *time*: the time on the applier server.

- *totalRequests*: the total number of requests the applier has made to the endpoint.

- *totalFailedConnects*: the total number of failed connection attempts the applier has made.

- *totalEvents*: the total number of log events the applier has processed.

- *progress*: a JSON hash with details about the replication applier progress. It contains the following sub-attributes if there is progress to report:

- *message*: a textual description of the progress

- *time*: the date and time the progress was logged

- *failedConnects*: the current number of failed connection attempts

- *lastError*: a JSON hash with details about the last error that happened on the applier. It contains the following sub-attributes if there was an error:

- *errorNum*: a numerical error code

- *errorMessage*: a textual error description

- *time*: the date and time the error occurred

In case no error has occurred, *lastError* will be empty.

- *server*: a JSON hash with the following sub-attributes:

- *version*: the applier server's version

- *serverId*: the applier server's id

- *endpoint*: the endpoint the applier is connected to (if applier is active) or will connect to (if applier is currently inactive)

- *database*: the name of the database the applier is connected to (if applier is active) or will connect to (if applier is currently inactive)

Return Codes

- *200*: is returned if the request was executed successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

Fetching the state of an inactive applier:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/applier-state
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
```

```
{
  "state" : {
    "running" : false,
    "lastAppliedContinuousTick" : null,
    "lastProcessedContinuousTick" : null,
    "lastAvailableContinuousTick" : null,
    "progress" : {
      "time" : "2014-07-02T15:27:03Z",
      "message" : "fetching master state information",
      "failedConnects" : 0
    },
    "totalRequests" : 0,
    "totalFailedConnects" : 0,
    "totalEvents" : 0,
    "lastError" : {
      "time" : "2014-07-02T15:27:03Z",
      "errorMessage" : "got incompatible master version from
endpoint 'tcp://127.0.0.1:8529': '2.0.2'",
      "errorNum" : 1403
    },
    "time" : "2014-07-02T15:27:04Z"
  },
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "141206089767669"
  },
  "endpoint" : "tcp://127.0.0.1:8529",
  "database" : "_system"
}
```

Fetching the state of an active applier:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/applier-state
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "state" : {
    "running" : true,
    "lastAppliedContinuousTick" : null,
    "lastProcessedContinuousTick" : null,
    "lastAvailableContinuousTick" : null,
    "progress" : {
      "time" : "2014-07-02T15:27:03Z",
      "message" : "fetching master state information",
      "failedConnects" : 0
    },
    "totalRequests" : 0,
    "totalFailedConnects" : 0,
    "totalEvents" : 0,
    "lastError" : {
      "errorNum" : 0
    },
    "time" : "2014-07-02T15:27:04Z"
  },
```

```
  "server" : {
    "version" : "2.2.0-devel",
    "serverId" : "141206089767669"
  },
  "endpoint" : "tcp://127.0.0.1:8529",
  "database" : "_system"
}
```

Other Replication Commands

get the server's id

Return server id

`GET /_api/replication/server-id`

Returns the servers id. The id is also returned by other replication API methods, and this method is an easy means of determining a server's id.

The body of the response is a JSON hash with the attribute *serverId*. The server id is returned as a string.

Return Codes

- *200*: is returned if the request was executed successfully.

- *405*: is returned when an invalid HTTP method is used.

- *500*: is returned if an error occurred while assembling the response.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/replication/server-id
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "serverId" : "141206089767669"
}
```

HTTP Interface for Bulk Imports

ArangoDB provides an HTTP interface to import multiple documents at once into a collection. This is known as a bulk import.

The data uploaded must be provided in JSON format. There are two mechanisms to import the data:

- self-contained JSON documents: in this case, each document contains all attribute names and values. Attribute names may be completely different among the documents uploaded
- attribute names plus document data: in this case, the first document must be a JSON list containing the attribute names of the documents that follow. The following documents must be lists containing only the document data. Data will be mapped to the attribute names by attribute positions.

The endpoint address is */_api/import* for both input mechanisms. Data must be sent to this URL using an HTTP POST request. The data to import must be contained in the body of the POST request.

The *collection* URL parameter must be used to specify the target collection for the import. The optional URL parameter *createCollection* can be used to create a non-existing collection during the import. If not used, importing data into a non-existing collection will produce an error. Please note that the *createCollection* flag can only be used to create document collections, not edge collections.

The *waitForSync* URL parameter can be set to *true* to make the import only return if all documents have been synced to disk.

The *complete* URL parameter can be set to *true* to make the entire import fail if any of the uploaded documents is invalid and cannot be imported. In this case, no documents will be imported by the import run, even if a failure happens at the end of the import.

If *complete* has a value other than *true*, valid documents will be imported while invalid documents will be rejected, meaning only some of the uploaded documents might have been imported.

The *details* URL parameter can be set to *true* to make the import API return details about documents that could not be imported. If *details* is *true*, then the result will also contain a *details* attribute which is a list of detailed error messages. If the *details* is set to *false* or omitted, no details will be returned.

Importing Self-Contained JSON Documents

This import method allows uploading self-contained JSON documents. The documents must be uploaded in the body of the HTTP POST request. Each line of the body will be interpreted as one stand-alone document. Empty lines in the body are allowed but will be skipped. Using this format, the documents are imported line-wise.

Example input data: { "_key": "key1", ... } { "_key": "key2", ... } ...

To use this method, the *type* URL parameter should be set to *documents*.

It is also possible to upload self-contained JSON documents that are embedded into a JSON list. Each element from the list will be treated as a document and be imported.

Example input data for this case:

```
[
  { "_key": "key1", ... },
  { "_key": "key2", ... },
  ...
]
```

This format does not require each document to be on a separate line, and any whitespace in the JSON data is allowed. It can be used to import a JSON-formatted result list (e.g. from arangosh) back into ArangoDB. Using this format requires ArangoDB to parse the complete list and keep it in memory for the duration of the import. This might be more resource-intensive than the line-wise processing.

To use this method, the *type* URL parameter should be set to *array*.

Setting the *type* URL parameter to *auto* will make the server auto-detect whether the data are line-wise JSON documents (type = documents) or a JSON list (type = array).

Examples

```
curl --data-binary @- -X POST --dump -
"http://localhost:8529/_api/import?
type=documents&collection=test&createCollection=true"
{ "name" : "test", "gender" : "male", "age" : 39 }
{ "type" : "bird", "name" : "robin" }

HTTP/1.1 201 Created
server: triagens GmbH High-Performance HTTP Server
connection: Keep-Alive
content-type: application/json; charset=utf-8

{"error":false,"created":2,"empty":0,"errors":0}
```

The server will respond with an HTTP 201 if everything went well. The number of documents imported will be returned in the *created* attribute of the response. If any documents were skipped or incorrectly formatted, this will be returned in the *errors* attribute. There will also be an attribute *empty* in the response, which will contain a value of *0*.

If the *details* parameter was set to *true* in the request, the response will also contain an attribute *details* which is a list of details about errors that occurred on the server side during the import. This list might be empty if no errors occurred.

Importing Headers and Values

When using this type of import, the attribute names of the documents to be imported are specified separate from the actual document value data. The first line of the HTTP POST request body must be a JSON list containing the attribute names for the documents that follow. The following lines are interpreted as the document data. Each document must be a JSON list of values. No attribute names are needed or allowed in this data section.

Examples

```
curl --data-binary @- -X POST --dump -
"http://localhost:8529/_api/import?
collection=test&createCollection=true"
[ "firstName", "lastName", "age", "gender" ]
[ "Joe", "Public", 42, "male" ]
[ "Jane", "Doe", 31, "female" ]

HTTP/1.1 201 Created
server: triagens GmbH High-Performance HTTP Server
connection: Keep-Alive
content-type: application/json; charset=utf-8

{"error":false,"created":2,"empty":0,"errors":0}
```

The server will again respond with an HTTP 201 if everything went well. The number of documents imported will be returned in the *created* attribute of the response. If any documents were skipped or incorrectly formatted, this will be returned in the *errors* attribute. The number of empty lines in the input file will be returned in the *empty* attribute.

If the *details* parameter was set to *true* in the request, the response will also contain an attribute *details* which is a list of details about errors that occurred on the server side during the import. This list might be empty if no errors occurred.

Importing into Edge Collections

Please note that when importing documents into an edge collection, it is mandatory that all imported documents contain the *_from* and *_to* attributes, and that these contain references to existing collections.

Please also note that it is not possible to create a new edge collection on the fly using the *createCollection* parameter.

HTTP Interface for Batch Requests

Clients normally send individual operations to ArangoDB in individual HTTP requests. This is straightforward and simple, but has the disadvantage that the network overhead can be significant if many small requests are issued in a row.

To mitigate this problem, ArangoDB offers a batch request API that clients can use to send multiple operations in one batch to ArangoDB. This method is especially useful when the client has to send many HTTP requests with a small body/payload and the individual request results do not depend on each other.

Clients can use ArangoDB's batch API by issuing a multipart HTTP POST request to the URL */_api/batch* handler. The handler will accept the request if the Content-Type is *multipart/form-data* and a boundary string is specified. ArangoDB will then decompose the batch request into its individual parts using this boundary. This also means that the boundary string itself must not be contained in any of the parts. When ArangoDB has split the multipart request into its individual parts, it will process all parts sequentially as if it were a standalone request. When all parts are processed, ArangoDB will generate a multipart HTTP response that contains one part for each part operation result. For example, if you send a multipart request with 5 parts, ArangoDB will send back a multipart response with 5 parts as well.

The server expects each part message to start with exactly the following "header":

```
Content-Type: application/x-arango-batchpart
```

You can optionally specify a *Content-Id* "header" to uniquely identify each part message. The server will return the *Content-Id* in its response if it is specified. Otherwise, the server will not send a Content-Id "header" back. The server will not validate the uniqueness of the Content-Id. After the mandatory *Content-Type* and the optional *Content-Id* header, two Windows line breaks (i.e. *\r\n\r\n*) must follow. Any deviation of this structure might lead to the part being rejected or incorrectly interpreted. The part request payload, formatted as a regular HTTP request, must follow the two Windows line breaks literal directly.

Note that the literal *Content-Type: application/x-arango-batchpart* technically is the header of the MIME part, and the HTTP request (including its headers) is the body part of the MIME part.

An actual part request should start with the HTTP method, the called URL, and the HTTP protocol version as usual, followed by arbitrary HTTP headers. Its body should follow after the usual *\r\n\r\n* literal. Part requests are therefore regular HTTP requests, only embedded inside a multipart message.

The following example will send a batch with 3 individual document creation operations. The boundary used in this example is *XXXsubpartXXX*.

Examples

```
> curl -X POST --data-binary @- --header "Content-Type:
multipart/form-data; boundary=XXXsubpartXXX"
http://localhost:8529/_api/batch
--XXXsubpartXXX
Content-Type: application/x-arango-batchpart
Content-Id: 1

POST /_api/document?collection=xyz&createCollection=true HTTP/1.1

{"a":1,"b":2,"c":3}
--XXXsubpartXXX
Content-Type: application/x-arango-batchpart
Content-Id: 2
```

```
POST /_api/document?collection=xyz HTTP/1.1

{"a":1,"b":2,"c":3,"d":4}
--XXXsubpartXXX
Content-Type: application/x-arango-batchpart
Content-Id: 3

POST /_api/document?collection=xyz HTTP/1.1

{"a":1,"b":2,"c":3,"d":4,"e":5}
--XXXsubpartXXX--
```

The server will then respond with one multipart message, containing the overall status and the individual results for the part operations. The overall status should be 200 except there was an error while inspecting and processing the multipart message. The overall status therefore does not indicate the success of each part operation, but only indicates whether the multipart message could be handled successfully.

Each part operation will return its own status value. As the part operation results are regular HTTP responses (just included in one multipart response), the part operation status is returned as a HTTP status code. The status codes of the part operations are exactly the same as if you called the individual operations standalone. Each part operation might also return arbitrary HTTP headers and a body/payload:

Examples

```
HTTP/1.1 200 OK
connection: Keep-Alive
content-type: multipart/form-data; boundary=XXXsubpartXXX
content-length: 1055

--XXXsubpartXXX
Content-Type: application/x-arango-batchpart
Content-Id: 1

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "9514299"
content-length: 53

{"error":false,"_id":"xyz/9514299","_key":"9514299","_rev":"9514299"}

--XXXsubpartXXX
Content-Type: application/x-arango-batchpart
Content-Id: 2

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "9579835"
content-length: 53

{"error":false,"_id":"xyz/9579835","_key":"9579835","_rev":"9579835"}

--XXXsubpartXXX
Content-Type: application/x-arango-batchpart
Content-Id: 3

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "9645371"
content-length: 53
```

```
{"error":false,"_id":"xyz/9645371","_key":"9645371","_rev":"9645371"}

--XXXsubpartXXX--
```

In the above example, the server returned an overall status code of 200, and each part response contains its own status value (202 in the example):

When constructing the multipart HTTP response, the server will use the same boundary that the client supplied. If any of the part responses has a status code of 400 or greater, the server will also return an HTTP header *x-arango-errors* containing the overall number of part requests that produced errors:

Examples

```
> curl -X POST --data-binary @- --header "Content-Type:
multipart/form-data; boundary=XXXsubpartXXX"
http://localhost:8529/_api/batch
--XXXsubpartXXX
Content-Type: application/x-arango-batchpart

POST /_api/document?collection=nonexisting

{"a":1,"b":2,"c":3}
--XXXsubpartXXX
Content-Type: application/x-arango-batchpart

POST /_api/document?collection=xyz

{"a":1,"b":2,"c":3,"d":4}
--XXXsubpartXXX--
```

In this example, the overall response code is 200, but as some of the part request failed (with status code 404), the *x-arango-errors* header of the overall response is *1*:

Examples

```
HTTP/1.1 200 OK
x-arango-errors: 1
content-type: multipart/form-data; boundary=XXXsubpartXXX
content-length: 711

--XXXsubpartXXX
Content-Type: application/x-arango-batchpart

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8
content-length: 111

{"error":true,"code":404,"errorNum":1203,"errorMessage":"collection
\/_api\/collection\/nonexisting not found"}
--XXXsubpartXXX
Content-Type: application/x-arango-batchpart

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "9841979"
content-length: 53
```

```
{"error":false,"_id":"xyz/9841979","_key":"9841979","_rev":"9841979"}

--XXXsubpartXXX--
```

Please note that the database used for all part operations of a batch request is determined by scanning the original URL (the URL that contains */_api/batch*). It is not possible to override the database name in part operations of a batch. When doing so, any other database name used in a batch part will be ignored.

HTTP Interface for Administration and Monitoring

This is an introduction to ArangoDB's Http interface for administration and monitoring of the server.

returns the log files

Read global log from the server

GET /_admin/log

Query Parameters

- *upto*: Returns all log entries up to log level *upto*. Note that *upto* must be:
- *fatal* or *0*
- *error* or *1*
- *warning* or *2*
- *info* or *3*
- *debug* or *4* The default value is *info*.

- *level*: Returns all log entries of log level *level*. Note that the URL parameters *upto* and *level* are mutually exclusive.

- *start*: Returns all log entries such that their log entry identifier (*lid* value) is greater or equal to *start*.

- *size*: Restricts the result to at most *size* log entries.

- *offset*: Starts to return log entries skipping the first *offset* log entries. *offset* and *size* can be used for pagination.

- *search*: Only return the log entries containing the text specified in *search*.

- *sort*: Sort the log entries either ascending (if *sort* is *asc*) or descending (if *sort* is *desc*) according to their *lid* values. Note that the *lid* imposes a chronological order. The default value is *asc*.

Returns fatal, error, warning or info log messages from the server's global log. The result is a JSON object with the following attributes:

- *lid*: a list of log entry identifiers. Each log message is uniquely identified by its @LIT{lid} and the identifiers are in ascending order.

- *level*: a list of the log-levels for all log entries.

- *timestamp*: a list of the timestamps as seconds since 1970-01-01 for all log entries.

- *text* a list of the texts of all log entries

- *totalAmount*: the total amount of log entries before pagination.

Return Codes

- *400*: is returned if invalid values are specified for *upto* or *level*.

- *403*: is returned if the log is requested for any database other than *_system*.

- *500*: is returned if the server cannot generate the result due to an out-of-memory error.

  reloads the routing information

Reload routing collection

```
POST /_admin/routing/reload
```

Reloads the routing information from the collection *routing*.

Return Codes

- *200*: Routing information was reloaded successfully.

  returns system status information for the server

Read the statistics

```
GET /_admin/statistics
```

Returns the statistics information. The returned object contains the statistics figures grouped together according to the description returned by *_admin/statistics-description*. For instance, to access a figure *userTime* from the group *system*, you first select the sub-object describing the group stored in *system* and in that sub-object the value for *userTime* is stored in the attribute of the same name.

In case of a distribution, the returned object contains the total count in *count* and the distribution list in *counts*. The sum (or total) of the individual values is returned in *sum*.

Return Codes

- *200*: Statistics were returned successfully.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_admin/statistics
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "time" : 1404314816.163653,
  "system" : {
    "minorPageFaults" : 98598,
    "majorPageFaults" : 2015,
    "userTime" : 6.863472,
    "systemTime" : 5.818084,
    "numberOfThreads" : 18,
    "residentSize" : 158740480,
    "residentSizePercent" : 0.01847982406616211,
    "virtualSize" : 5136932864
  },
  "client" : {
    "httpConnections" : 1,
    "connectionTime" : {
      "sum" : 0.0006968975067138672,
      "count" : 1,
      "counts" : [
        1,
        0,
        0,
        0
      ]
    },
    "totalTime" : {
```

      "sum" : 16.838430881500244,
      "count" : 8676,
      "counts" : [
        8410,
        232,
        20,
        7,
        6,
        1,
        0
      ]
    },
    "requestTime" : {
      "sum" : 15.587429523468018,
      "count" : 8676,
      "counts" : [
        8418,
        224,
        20,
        7,
        6,
        1,
        0
      ]
    },
    "queueTime" : {
      "sum" : 0.20531082153320312,
      "count" : 8676,
      "counts" : [
        8676,
        0,
        0,
        0,
        0,
        0,
        0
      ]
    },
    "ioTime" : {
      "sum" : 1.0456905364990234,
      "count" : 8676,
      "counts" : [
        8676,
        0,
        0,
        0,
        0,
        0,
        0
      ]
    },
    "bytesSent" : {
      "sum" : 3155311,
      "count" : 8676,
      "counts" : [
        2639,
        5903,
        9,
        62,
        63,

```
            0
          ]
        },
        "bytesReceived" : {
          "sum" : 2055784,
          "count" : 8676,
          "counts" : [
            6464,
            2212,
            0,
            0,
            0,
            0
          ]
        }
      },
      "http" : {
        "requestsTotal" : 8676,
        "requestsAsync" : 0,
        "requestsGet" : 3591,
        "requestsHead" : 293,
        "requestsPost" : 3143,
        "requestsPut" : 676,
        "requestsPatch" : 28,
        "requestsDelete" : 945,
        "requestsOptions" : 0,
        "requestsOther" : 0
      },
      "server" : {
        "uptime" : 24.27107810974121,
        "physicalMemory" : 8589934592
      },
      "error" : false,
      "code" : 200
 }
```

returns statistics description

Statistics description

```
GET /_admin/statistics-description
```

Returns a description of the statistics returned by *_admin/statistics*. The returned objects contains a list of statistics groups in the attribute *groups* and a list of statistics figures in the attribute *figures*.

A statistics group is described by

- *group*: The identifier of the group.
- *name*: The name of the group.
- *description*: A description of the group.

A statistics figure is described by

- *group*: The identifier of the group to which this figure belongs.
- *identifier*: The identifier of the figure. It is unique within the group.
- *name*: The name of the figure.
- *description*: A description of the figure.
- *type*: Either *current*, *accumulated*, or *distribution*.
- *cuts*: The distribution vector.

- *units*: Units in which the figure is measured.

Return Codes

- *200*: Description was returned successfully.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_admin/statistics-description
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "groups" : [
    {
      "group" : "system",
      "name" : "Process Statistics",
      "description" : "Statistics about the ArangoDB process"
    },
    {
      "group" : "client",
      "name" : "Client Connection Statistics",
      "description" : "Statistics about the connections."
    },
    {
      "group" : "http",
      "name" : "HTTP Request Statistics",
      "description" : "Statistics about the HTTP requests."
    },
    {
      "group" : "server",
      "name" : "Server Statistics",
      "description" : "Statistics about the ArangoDB server"
    }
  ],
  "figures" : [
    {
      "group" : "system",
      "identifier" : "userTime",
      "name" : "User Time",
      "description" : "Amount of time that this process has been
scheduled in user mode, measured in seconds.",
      "type" : "accumulated",
      "units" : "seconds"
    },
    {
      "group" : "system",
      "identifier" : "systemTime",
      "name" : "System Time",
      "description" : "Amount of time that this process has been
scheduled in kernel mode, measured in seconds.",
      "type" : "accumulated",
      "units" : "seconds"
    },
    {
      "group" : "system",
```

```
      "identifier" : "numberOfThreads",
      "name" : "Number of Threads",
      "description" : "Number of threads in the arangod process.",
      "type" : "current",
      "units" : "number"
    },
    {
      "group" : "system",
      "identifier" : "residentSize",
      "name" : "Resident Set Size",
      "description" : "The total size of the number of pages the
process has in real memory. This is just the pages which count
toward text, data, or stack space. This does not include pages
which have not been demand-loaded in, or which are swapped out. The
resident set size is reported in bytes.",
      "type" : "current",
      "units" : "bytes"
    },
    {
      "group" : "system",
      "identifier" : "residentSizePercent",
      "name" : "Resident Set Size",
      "description" : "The percentage of physical memory used by
the process as resident set size.",
      "type" : "current",
      "units" : "percent"
    },
    {
      "group" : "system",
      "identifier" : "virtualSize",
      "name" : "Virtual Memory Size",
      "description" : "On Windows, this figure contains the total
amount of memory that the memory manager has committed for the
arangod process. On other systems, this figure contains The size of
the virtual memory the process is using.",
      "type" : "current",
      "units" : "bytes"
    },
    {
      "group" : "system",
      "identifier" : "minorPageFaults",
      "name" : "Minor Page Faults",
      "description" : "The number of minor faults the process has
made which have not required loading a memory page from disk. This
figure is not reported on Windows.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "system",
      "identifier" : "majorPageFaults",
      "name" : "Major Page Faults",
      "description" : "On Windows, this figure contains the total
number of page faults. On other system, this figure contains the
number of major faults the process has made which have required
loading a memory page from disk.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
```

```json
        "group" : "client",
        "identifier" : "httpConnections",
        "name" : "Client Connections",
        "description" : "The number of connections that are currently
open.",
        "type" : "current",
        "units" : "number"
    },
    {
        "group" : "client",
        "identifier" : "totalTime",
        "name" : "Total Time",
        "description" : "Total time needed to answer a request.",
        "type" : "distribution",
        "cuts" : [
          0.01,
          0.05,
          0.1,
          0.2,
          0.5,
          1
        ],
        "units" : "seconds"
    },
    {
        "group" : "client",
        "identifier" : "requestTime",
        "name" : "Request Time",
        "description" : "Request time needed to answer a request.",
        "type" : "distribution",
        "cuts" : [
          0.01,
          0.05,
          0.1,
          0.2,
          0.5,
          1
        ],
        "units" : "seconds"
    },
    {
        "group" : "client",
        "identifier" : "queueTime",
        "name" : "Queue Time",
        "description" : "Queue time needed to answer a request.",
        "type" : "distribution",
        "cuts" : [
          0.01,
          0.05,
          0.1,
          0.2,
          0.5,
          1
        ],
        "units" : "seconds"
    },
    {
        "group" : "client",
        "identifier" : "bytesSent",
        "name" : "Bytes Sent",
```

```json
        "description" : "Bytes sents for a request.",
        "type" : "distribution",
        "cuts" : [
          250,
          1000,
          2000,
          5000,
          10000
        ],
        "units" : "bytes"
      },
      {
        "group" : "client",
        "identifier" : "bytesReceived",
        "name" : "Bytes Received",
        "description" : "Bytes receiveds for a request.",
        "type" : "distribution",
        "cuts" : [
          250,
          1000,
          2000,
          5000,
          10000
        ],
        "units" : "bytes"
      },
      {
        "group" : "client",
        "identifier" : "connectionTime",
        "name" : "Connection Time",
        "description" : "Total connection time of a client.",
        "type" : "distribution",
        "cuts" : [
          0.1,
          1,
          60
        ],
        "units" : "seconds"
      },
      {
        "group" : "http",
        "identifier" : "requestsTotal",
        "name" : "Total requests",
        "description" : "Total number of HTTP requests.",
        "type" : "accumulated",
        "units" : "number"
      },
      {
        "group" : "http",
        "identifier" : "requestsAsync",
        "name" : "Async requests",
        "description" : "Number of asynchronously executed HTTP
requests.",
        "type" : "accumulated",
        "units" : "number"
      },
      {
        "group" : "http",
        "identifier" : "requestsGet",
        "name" : "HTTP GET requests",
```

```
      "description" : "Number of HTTP GET requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsHead",
      "name" : "HTTP HEAD requests",
      "description" : "Number of HTTP HEAD requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsPost",
      "name" : "HTTP POST requests",
      "description" : "Number of HTTP POST requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsPut",
      "name" : "HTTP PUT requests",
      "description" : "Number of HTTP PUT requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsPatch",
      "name" : "HTTP PATCH requests",
      "description" : "Number of HTTP PATCH requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsDelete",
      "name" : "HTTP DELETE requests",
      "description" : "Number of HTTP DELETE requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsOptions",
      "name" : "HTTP OPTIONS requests",
      "description" : "Number of HTTP OPTIONS requests.",
      "type" : "accumulated",
      "units" : "number"
    },
    {
      "group" : "http",
      "identifier" : "requestsOther",
      "name" : "other HTTP requests",
      "description" : "Number of other HTTP requests.",
      "type" : "accumulated",
      "units" : "number"
    },
```

```
    {
      "group" : "server",
      "identifier" : "uptime",
      "name" : "Server Uptime",
      "description" : "Number of seconds elapsed since server
start.",
      "type" : "current",
      "units" : "seconds"
    },
    {
      "group" : "server",
      "identifier" : "physicalMemory",
      "name" : "Physical Memory",
      "description" : "Physical memory in bytes.",
      "type" : "current",
      "units" : "bytes"
    }
  ],
  "error" : false,
  "code" : 200
}
```

returns the role of a server in a cluster

Return role of a server in a cluster

```
GET /_admin/server/role
```

Returns the role of a server in a cluster. The role is returned in the *role* attribute of the result. Possible return values for *role* are:

- *COORDINATOR*: the server is a coordinator in a cluster
- *PRIMARY*: the server is a primary database server in a cluster
- *SECONDARY*: the server is a secondary database server in a cluster
- *UNDEFINED*: in a cluster, *UNDEFINED* is returned if the server role cannot be determined. On a single server, *UNDEFINED* is the only possible return value.

Return Codes

- *200*: Is returned in all cases.

HTTP Interface for User Management

This is an introduction to ArangoDB's Http interface for managing users.

The interface provides a simple means to add, update, and remove users. All users managed through this interface will be stored in the system collection _users_.

This specialized interface intentionally does not provide all functionality that is available in the regular document REST API.

Operations on users may become more restricted than regular document operations, and extra privileges and security security checks may be introduced in the future for this interface.

Please note that user operations are not included in ArangoDB's replication.

Create User

```
POST /_api/user
```

The following data need to be passed in a JSON representation in the body of the POST request:

- *user*: The name of the user as a string. This is mandatory
- *passwd*: The user password as a string. If no password is specified, the empty string will be used
- *active*: An optional flag that specifies whether the user is active. If not specified, this will default to true
- *extra*: An optional JSON object with arbitrary extra data about the user
- *changePassword*: An optional flag that specifies whethers the user must change the password or not. If not specified, this will default to false

If set to true, the only operations allowed are PUT /_api/user or PATCH /_api/user. All other operations will result in a HTTP 403. If the user can be added by the server, the server will respond with HTTP 201. In case of success, the returned JSON object has the following properties:

- *error*: Boolean flag to indicate that an error occurred (false in this case)
- *code*: The HTTP status code

If the JSON representation is malformed or mandatory data is missing from the request, the server will respond with HTTP 400.

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: Boolean flag to indicate that an error occurred (true in this case)
- *code*: The HTTP status code
- *errorNum*: The server error number
- *errorMessage*: A descriptive error message

Return Codes

- *201*: Returned if the user can be added by the server

- *400*: If the JSON representation is malformed or mandatory data is missing from the request.

Replace User

```
PUT /_api/user/{user}
```

Replaces the data of an existing user. The name of an existing user must be specified in user.

The following data can to be passed in a JSON representation in the body of the POST request:

- *passwd*: The user password as a string. Specifying a password is mandatory, but the empty string is allowed for passwords
- *active*: An optional flag that specifies whether the user is active. If not specified, this will default to true
- *extra*: An optional JSON object with arbitrary extra data about the user
- *changePassword*: An optional flag that specifies whether the user must change the password or not. If not specified, this will default to false

If the user can be replaced by the server, the server will respond with HTTP 200.

In case of success, the returned JSON object has the following properties:

- *error*: Boolean flag to indicate that an error occurred (false in this case)
- *code*: The HTTP status code

If the JSON representation is malformed or mandatory data is missing from the request, the server will respond with HTTP 400. If the specified user does not exist, the server will respond with HTTP 404.

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: Boolean flag to indicate that an error occurred (true in this case)
- *code*: The HTTP status code
- *errorNum*: The server error number
- *errorMessage*: A descriptive error message

Return Codes

- *200*: Is returned if the user data can be replaced by the server

- *400*: The JSON representation is malformed or mandatory data is missing from the request

- *404*: The specified user does not exist

Update User

```
PATCH /_api/user/{user}
```

Partially updates the data of an existing user. The name of an existing user must be specified in user.

The following data can be passed in a JSON representation in the body of the POST request:

- *passwd*: The user password as a string. Specifying a password is optional. If not specified, the previously existing value will not be modified.
- *active*: An optional flag that specifies whether the user is active. If not specified, the previously existing value will not be modified.
- *extra*: An optional JSON object with arbitrary extra data about the user. If not specified, the previously existing value will not be modified.
- *changePassword*: An optional flag that specifies whether the user must change the password or not. If not specified, the previously existing value will not be modified.

If the user can be updated by the server, the server will respond with HTTP 200.

In case of success, the returned JSON object has the following properties:

- *error*: Boolean flag to indicate that an error occurred (false in this case)
- *code*: The HTTP status code

If the JSON representation is malformed or mandatory data is missing from the request, the server will

respond with HTTP 400. If the specified user does not exist, the server will respond with HTTP 404.

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: Boolean flag to indicate that an error occurred (true in this case)
- *code*: The HTTP status code
- *errorNum*: The server error number
- *errorMessage*: A descriptive error message

Return Codes

- *200*: Is returned if the user data can be replaced by the server

- *400*: The JSON representation is malformed or mandatory data is missing from the request

- *404*: The specified user does not exist

Remove User

```
DELETE /_api/user/{user}
```

Removes an existing user, identified by user.

If the user can be removed, the server will respond with HTTP 202. In case of success, the returned JSON object has the following properties:

- *error*: Boolean flag to indicate that an error occurred (false in this case)
- *code*: The HTTP status code

If the specified user does not exist, the server will respond with HTTP 404.

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: Boolean flag to indicate that an error occurred (true in this case)
- *code*: The HTTP status code
- *errorNum*: The server error number
- *errorMessage*: A descriptive error message

Return Codes

- *202*: Is returned if the user was removed by the server

- *404*: The specified user does not exist

Fetch User

```
GET /_api/user/{user}
```

Fetches data about the specified user.

The call will return a JSON document with at least the following attributes on success:

- *user*: The name of the user as a string.
- *active*: An optional flag that specifies whether the user is active.
- *extra*: An optional JSON object with arbitrary extra data about the user.
- *changePassword*: An optional flag that specifies whether the user must change the password or not.

Return Codes

- *200*: The user was found

- *404*: The user with user does not exist

HTTP Interface for Async Results Management

Request Execution

ArangoDB provides various methods of executing client requests. Clients can choose the appropriate method on a per-request level based on their throughput, control flow, and durability requirements.

Blocking execution

ArangoDB is a multi-threaded server, allowing the processing of multiple client requests at the same time. Communication handling and the actual work can be performed by multiple worker threads in parallel.

Though multiple clients can connect and send their requests in parallel to ArangoDB, clients may need to wait for their requests to be processed.

By default, the server will fully process an incoming request and then return the result to the client. The client must wait for the server's response before it can send additional requests over the connection. For clients that are single-threaded or not event-driven, waiting for the full server response may be non-optimal.

Furthermore, please note that even if the client closes the HTTP connection, the request running on the server will still continue until it is complete and only then notice that the client no longer listens. Thus closing the connection does not help to abort a long running query! See below under Async Execution and later Result Retrieval and HttpJobPutCancel for details.

Fire and Forget

To mitigate client blocking issues, ArangoDB since version 1.4. offers a generic mechanism for non-blocking requests: if clients add the HTTP header *x-arango-async: true* to their requests, ArangoDB will put the request into an in-memory task queue and return an HTTP 202 (accepted) response to the client instantly. The server will execute the tasks from the queue asynchronously, decoupling the client requests and the actual work.

This allows for much higher throughput than if clients would wait for the server's response. The downside is that the response that is sent to the client is always the same (a generic HTTP 202) and clients cannot make a decision based on the actual operation's result. In fact, the operation might have not even been executed at the time the generic response has reached the client. Clients can thus not rely on their requests having been processed successfully.

The asynchronous task queue on the server is not persisted, meaning not-yet processed tasks from the queue might be lost in case of a crash.

Clients should thus not send the extra header when they have strict durability requirements or if they rely on result of the sent operation for further actions.

The maximum number of queued tasks is determined by the startup option *-scheduler.maximal-queue-size*. If more than this number of tasks are already queued, the server will reject the request with an HTTP 500 error.

Finally, please note that it is not possible to cancel such a non-blocking request after the fact. If you need to cancel requests, use Async Execution and later Result Retrieval and HttpJobPutCancel below.

Async Execution and later Result Retrieval

By adding the HTTP header *x-arango-async: store* to a request, clients can instruct the ArangoDB server to execute the operation asynchronously as above "above", but also store the operation result in memory for a later retrieval. The server will return a job id in the HTTP response header *x-arango-async-id*. The client can use this id in conjunction with the HTTP API at */_api/job*, which is described in detail in this

manual.

Clients can ask the ArangoDB server via the async jobs API which results are ready for retrieval, and which are not. Clients can also use the async jobs API to retrieve the original results of an already executed async job by passing it the originally returned job id. The server will then return the job result as if the job was executed normally. Furthermore, clients can cancel running async jobs by their job id, see HttpJobPutCancel.

ArangoDB will keep all results of jobs initiated with the *x-arango-async: store* header. Results are removed from the server only if a client explicitly asks the server for a specific result.

The async jobs API also provides methods for garbage collection that clients can use to get rid of "old" not fetched results. Clients should call this method periodically because ArangoDB does not artificially limit the number of not-yet-fetched results.

It is thus a client responsibility to store only as many results as needed and to fetch available results as soon as possible, or at least to clean up not fetched results from time to time.

The job queue and the results are kept in memory only on the server, so they might be lost in case of a crash.

Canceling asynchronous jobs

As mentioned above it is possible to cancel an asynchronously running job using its job ID. This is done with a PUT request as described in HttpJobPutCancel.

However, a few words of explanation about what happens behind the scenes are in order. Firstly, a running async query can internally be executed by C++ code or by JavaScript code. For example CRUD operations are executed directly in C++, whereas AQL queries and transactions are executed by JavaScript code. The job cancellation only works for JavaScript code, since the mechanism used is simply to trigger an uncatchable exception in the JavaScript thread, which will be caught on the C++ level, which in turn leads to the cancellation of the job. No result can be retrieved later, since all data about the request is discarded.

If you cancel a job running on a coordinator of a cluster (Sharding), then only the code running on the coordinator is stopped, there may remain tasks within the cluster which have already been distributed to the DBservers and it is currently not possible to cancel them as well.

Async Execution and Authentication

If a request requires authentication, the authentication procedure is run before queueing. The request will only be queued if it valid credentials and the authentication succeeds. If the request does not contain valid credentials, it will not be queued but rejected instantly in the same way as a "regular", non-queued request.

Managing Async Results via HTTP

Returns the result of an async job

Return result of an async job

`PUT /_api/job/job-id`

URL Parameters

- *job-id*: The async job id.

Returns the result of an async job identified by job-id. If the async job result is present on the server, the result will be removed from the list of result. That means this method can be called for each job-id once. The method will return the original job result's headers and body, plus the additional HTTP header x-arango-async-job-id. If this header is present, then the job was found and the response contains the original job's result. If the header is not present, the job was not found and the response contains status information from the job manager.

Return Codes

- *204*: is returned if the job requested via job-id is still in the queue of pending (or not yet finished) jobs. In this case, no x-arango-async-id HTTP header will be returned.

- *400*: is returned if no job-id was specified in the request. In this case, no x-arango-async-id HTTP header will be returned.

- *404*: is returned if the job was not found or already deleted or fetched from the job result list. In this case, no x-arango-async-id HTTP header will be returned.

Examples Not providing a job-id:

```
unix> curl -X PUT --dump - http://localhost:8529/_api/job/


HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8


{"error":true,"errorMessage":"bad
parameter","code":400,"errorNum":400}
```

Providing a job-id for a non-existing job:

```
unix> curl -X PUT --dump - http://localhost:8529/_api/job/foobar


HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8


{"error":true,"errorMessage":"not found","code":404,"errorNum":404}
```

Fetching the result of an HTTP GET job:

```
unix> curl --header 'x-arango-async: store' --dump -
```

```
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 265413601


unix> curl -X PUT --dump - http://localhost:8529/_api/job/265413601


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
x-arango-async-id: 265413601


{"server":"arango","version":"2.1.0"}
```

Fetching the result of an HTTP POST job that failed:

```
unix> curl -X POST --header 'x-arango-async: store' --data-binary
@- --dump - http://localhost:8529/_api/collection
{"name":" this name is invalid "}


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 265479137


unix> curl -X PUT --dump - http://localhost:8529/_api/job/265479137


HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8
x-arango-async-id: 265479137


{"error":true,"code":400,"errorNum":1208,"errorMessage":"cannot
create collection: illegal name"}
```

Cancels the result an async job

Cancel async job

PUT /_api/job/job-id/cancel

URL Parameters

- *job-id*: The async job id.

Cancels the currently running job identified by job-id. Note that it still might take some time to actually cancel the running async job.

Return Codes

- *200*: cancel has been initiated.

- *400*: is returned if no job-id was specified in the request. In this case, no x-arango-async-id HTTP header will be returned.

- *404*: is returned if the job was not found or already deleted or fetched from the job result list. In this case, no x-arango-async-id HTTP header will be returned.

Examples

```
unix> curl -X POST --header 'x-arango-async: store' --data-binary
@- --dump - http://localhost:8529/_api/cursor
{"query": "FOR i IN 1..10 FOR j IN 1..10 LET x = sleep(1.0) FILTER
i == 5 && j == 5 RETURN 42"}


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 268952545


unix> curl --dump - http://localhost:8529/_api/job/pending


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


["268952545"]


unix> curl -X PUT --dump -
http://localhost:8529/_api/job/268952545/cancel


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


{"result":true}


unix> curl --dump - http://localhost:8529/_api/job/pending


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


["268952545"]
```

Deletes the result of an async job

Deletes async job

```
DELETE /_api/job/type
```

URL Parameters

- *type*: The type of jobs to delete. type can be:
- *all*: Deletes all jobs results. Currently executing or queued async jobs will not be stopped by this

call.

- *expired*: Deletes expired results. To determine the expiration status of a result, pass the stamp URL parameter. stamp needs to be a UNIX timestamp, and all async job results created at a lower timestamp will be deleted.
- *an actual job-id*: In this case, the call will remove the result of the specified async job. If the job is currently executing or queued, it will not be aborted.

Query Parameters

- *stamp*:

A UNIX timestamp specifying the expiration threshold when type is expired.

Deletes either all job results, expired job results, or the result of a specific job. Clients can use this method to perform an eventual garbage collection of job results.

Return Codes

- *200*: is returned if the deletion operation was carried out successfully. This code will also be returned if no results were deleted.

- *400*: is returned if type is not specified or has an invalid value.

- *404*: is returned if type is a job-id but no async job with the specified id was found.

Examples

Deleting all jobs:

```
unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 270132193


unix> curl -X DELETE --dump - http://localhost:8529/_api/job/all


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


{
"result" : true
}
```

Deleting expired jobs:

```
unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 270197729
```

```
unix> curl -X DELETE --dump -
http://localhost:8529/_api/job/expired?stamp=1401376184


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


{
"result" : true
}
```

Deleting the result of a specific job:

```
unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 270263265


unix> curl -X DELETE --dump -
http://localhost:8529/_api/job/270263265


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


{
"result" : true
}
```

Deleting the result of a non-existing job:

```
unix> curl -X DELETE --dump - http://localhost:8529/_api/job/foobar


HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8


{
"error" : true,
"errorMessage" : "not found",
"code" : 404,
"errorNum" : 404
}
```

Returns the status of an async job

Returns async job

```
GET /_api/job/job-id
```

URL Parameters

- *job-id*: The async job id.

Returns the processing status of the specified job. The processing status can be determined by peeking into the HTTP response code of the response.

Return Codes

- *200*: is returned if the job requested via job-id has been executed successfully and its result is ready to fetch.

- *204*: is returned if the job requested via job-id is still in the queue of pending (or not yet finished) jobs.

- *404*: is returned if the job was not found or already deleted or fetched from the job result list.

Examples

Querying the status of a done job:

```
unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 270328801


unix> curl --dump - http://localhost:8529/_api/job/270328801


HTTP/1.1 200 OK
content-type: text/plain; charset=utf-8


Querying the status of a pending job:


unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_admin/sleep?duration=3


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 270394337


unix> curl --dump - http://localhost:8529/_api/job/270394337


HTTP/1.1 204 No Content
content-type: text/plain; charset=utf-8
```

Returns the list of job result id with a specific status

Returns list of async job

GET /_api/job/type

URL Parameters

- *type*: The type of jobs to return. The type can be either done or pending. Setting the type to done will make the method return the ids of already completed async jobs for which results can be fetched. Setting the type to pending will return the ids of not yet finished async jobs.

Query Parameters

- *count*:

The maximum number of ids to return per call. If not specified, a server-defined maximum value will be used.

Returns the list of ids of async jobs with a specific status (either done or pending). The list can be used by the client to get an overview of the job system status and to retrieve completed job results later.

Return Codes

- *200*: is returned if the list can be compiled successfully. Note: the list might be empty.

- *400*: is returned if type is not specified or has an invalid value.

Examples

Fetching the list of done jobs:

```
unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
x-arango-async-id: 270459873


unix> curl --dump - http://localhost:8529/_api/job/done


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


[
"270459873"
]
```

Fetching the list of pending jobs:

```
unix> curl --header 'x-arango-async: store' --dump -
http://localhost:8529/_api/version


HTTP/1.1 202 Accepted
content-type: text/plain; charset=utf-8
```

```
x-arango-async-id: 270525409


unix> curl --dump - http://localhost:8529/_api/job/pending


HTTP/1.1 200 OK
content-type: application/json; charset=utf-8


[ ]
```

HTTP Interface for Documents

Documents, Identifiers, Handles

This is an introduction to ArangoDB's REST interface for documents.

Documents in ArangoDB are JSON objects. These objects can be nested (to any depth) and may contain lists. Each document is uniquely identified by its document handle.

An example document:

```
{
  "_id" : "myusers/2345678",
  "_key" : "3456789",
  "_rev" : "3456789",
  "firstName" : "Hugo",
  "lastName" : "Schlonz",
  "address" : {
    "street" : "Street of Happiness",
    "city" : "Heretown"
  },
  "hobbies" : [
    "swimming",
    "biking",
    "programming"
  ]
}
```

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

Document Handle

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

Document Key

A document key uniquely identifies a document in a given collection. It can and should be used by clients when specific documents are searched. Document keys are stored in the _key attribute of documents. The key values are automatically indexed by ArangoDB in a collection's primary index. Thus looking up a document by its key is regularly a fast operation. The _key value of a document is immutable once the document has been created. By default, ArangoDB will auto-generate a document key if no _key attribute is specified, and use the user-specified _key otherwise.

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

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

Document Revision

As ArangoDB supports MVCC, documents can exist in more than one revision. The document revision is the MVCC token used to identify a particular revision of a document. It is a string value currently

containing an integer number and is unique within the list of document revisions for a single document. Document revisions can be used to conditionally update, replace or delete documents in the database. In order to find a particular revision of a document, you need the document handle and the document revision. ArangoDB currently uses 64bit unsigned integer values to maintain document revisions internally. When returning document revisions to clients, ArangoDB will put them into a string to ensure the revision id is not clipped by clients that do not support big integers. Clients should treat the revision id returned by ArangoDB as an opaque string when they store or use it locally. This will allow ArangoDB to change the format of revision ids later if this should be required. Clients can use revisions ids to perform simple equality/non-equality comparisons (e.g. to check whether a document has changed or not), but they should not use revision ids to perform greater/less than comparisons with them to check if a document revision is older than one another, even if this might work for some cases.

Note: Revision ids have been returned as integers up to including ArangoDB 1.1

Document Etag

The document revision enclosed in double quotes. The revision is returned by several HTTP API methods in the Etag HTTP header.

The basic operations (create, read, update, delete) for documents are mapped to the standard HTTP methods (*POST*, *GET*, *PUT*, *DELETE*). There is also a partial update method, which is mapped to the HTTP *PATCH* method.

An identifier for the document revision is returned in the *ETag* HTTP header. If you modify a document, you can use the *If-Match* field to detect conflicts. The revision of a document can be checking using the HTTP method *HEAD*.

Address and ETag of a Document

All documents in ArangoDB have a document handle. This handle uniquely identifies a document. Any document can be retrieved using its unique URI:

```
http://server:port/_api/document/<document-handle>
```

For example, assumed that the document handle, which is stored in the _id attribute of the document, is *demo/362549736*, then the URL of that document is:

```
http://localhost:8529/_api/document/demo/362549736
```

The above URL scheme does not specify a database name explicitly, so the default database will be used. To explicitly specify the database context, use the following URL schema:

```
http://server:port/_db/<database-name>/_api/document/<document-
handle>
```

Example:

```
http://localhost:8529/_db/mydb/_api/document/demo/362549736
```

Note: The following examples use the short URL format for brevity.

Each document also has a document revision or etag with is returned in the "ETag" HTTP header when requesting a document.

If you obtain a document using *GET* and you want to check if a newer revision is available, then you can use the *If-None-Match* header. If the document is unchanged, a *HTTP 412* (precondition failed) error is returned.

If you want to update or delete a document, then you can use the *If-Match* header. If the document has changed, then the operation is aborted and a *HTTP 412* error is returned.

Working with Documents using REST

reads a single document

Read document

```
GET /_api/document/document-handle
```

URL Parameters

- *document-handle*: The handle of the document.

Header Parameters

- *If-None-Match*: If the "If-None-Match" header is given, then it must contain exactly one etag. The document is returned, if it has a different revision than the given etag. Otherwise an *HTTP 304* is returned.

- *If-Match*: If the "If-Match" header is given, then it must contain exactly one etag. The document is returned, if it has the same revision ad the given etag. Otherwise a *HTTP 412* is returned. As an alternative you can supply the etag in an attribute *rev* in the URL.

Returns the document identified by *document-handle*. The returned document contains two special attributes: *_id* containing the document handle and *_rev* containing the revision.

Return Codes

- *200*: is returned if the document was found

- *304*: is returned if the "If-None-Match" header is given and the document has the same version

- *404*: is returned if the document or collection was not found

- *412*: is returned if a "If-Match" header or *rev* is given and the found document has a different version. The response will also contain the found document's current revision in the *_rev* attribute. Additionally, the attributes *_id* and *_key* will be returned.

Examples

Use a document handle:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/document/products/1206084277
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: "1206084277"

{
  "hello" : "world",
  "_id" : "products/1206084277",
  "_rev" : "1206084277",
  "_key" : "1206084277"
}
```

Use a document handle and an etag:

```
shell> curl --header 'If-None-Match: "1206674101"' --dump -
http://localhost:8529/_api/document/products/1206674101
```

Unknown document handle:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/document/products/unknownhandle
undefined

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "errorMessage" : "document /_api/document/products/unknownhandle
not found",
  "code" : 404,
  "errorNum" : 1202
}
```

creates a document

Create document

POST /_api/document

- *document*: A JSON representation of the document.

Query Parameters

- *collection*: The collection name.

- *createCollection*: If this parameter has a value of *true* or *yes*, then the collection is created if it does not yet exist. Other values will be ignored so the collection must be present for the operation to succeed.

Note: this flag is not supported in a cluster. Using it will result in an error.

- *waitForSync*: Wait until document has been synced to disk.

Creates a new document in the collection named *collection*. A JSON representation of the document must be passed as the body of the POST request.

If the document was created successfully, then the "Location" header contains the path to the newly created document. The "ETag" header field contains the revision of the document.

The body of the response contains a JSON object with the following attributes:

- *_id* contains the document handle of the newly created document
- *_key* contains the document key
- *_rev* contains the document revision

If the collection parameter *waitForSync* is *false*, then the call returns as soon as the document has been accepted. It will not wait until the document has been synced to disk.

Optionally, the URL parameter *waitForSync* can be used to force synchronisation of the document creation operation to disk even in case that the *waitForSync* flag had been disabled for the entire collection. Thus, the *waitForSync* URL parameter can be used to force synchronisation of just this specific

operations. To use this, set the *waitForSync* parameter to *true*. If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* URL parameter cannot be used to disable synchronisation for collections that have a default *waitForSync* value of *true*.

Return Codes

- *201*: is returned if the document was created successfully and *waitForSync* was *true*.

- *202*: is returned if the document was created successfully and *waitForSync* was *false*.

- *400*: is returned if the body does not contain a valid JSON representation of a document. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Create a document given a collection named *products*. Note that the revision identifier might or might not by equal to the auto-generated key.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/document?collection=products
{ "Hello": "World" }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8
etag: "1203987125"
location: /_db/_system/_api/document/products/1203987125

{
  "error" : false,
  "_id" : "products/1203987125",
  "_rev" : "1203987125",
  "_key" : "1203987125"
}
```

Create a document in a collection named *products* with a collection-level *waitForSync* value of *false*.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/document?collection=products
{ "Hello": "World" }

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1204511413"
location: /_db/_system/_api/document/products/1204511413

{
  "error" : false,
  "_id" : "products/1204511413",
  "_rev" : "1204511413",
  "_key" : "1204511413"
}
```

Create a document in a collection with a collection-level *waitForSync* value of *false*, but using the *waitForSync* URL parameter.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/document?
collection=products&waitForSync=true
{ "Hello": "World" }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8
etag: "1205035701"
location: /_db/_system/_api/document/products/1205035701

{
  "error" : false,
  "_id" : "products/1205035701",
  "_rev" : "1205035701",
  "_key" : "1205035701"
}
```

Create a document in a new, named collection

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/document?
collection=products&createCollection=true
{ "Hello": "World" }

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1205559989"
location: /_db/_system/_api/document/products/1205559989

{
  "error" : false,
  "_id" : "products/1205559989",
  "_rev" : "1205559989",
  "_key" : "1205559989"
}
```

Unknown collection name:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/document?collection=products
{ "Hello": "World" }

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "errorMessage" : "collection 'products' not found",
  "code" : 404,
  "errorNum" : 1203
}
```

Illegal document:

```
shell> curl -X POST --data-binary @- --dump -
```

```
http://localhost:8529/_api/document?collection=products
{ 1: "World" }

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "errorMessage" : "expecting attribute name",
  "code" : 400,
  "errorNum" : 600
}
```

replaces a document

Replace document

PUT /_api/document/document-handle

- *document*: A JSON representation of the new document.

URL Parameters

- *document-handle*: The handle of the document.

Query Parameters

- *waitForSync*: Wait until document has been synced to disk.

- *rev*: You can conditionally replace a document based on a target revision id by using the *rev* URL parameter.

- *policy*: To control the update behavior in case there is a revision mismatch, you can use the *policy* parameter (see below).

Header Parameters

- *If-Match*: You can conditionally replace a document based on a target revision id by using the *if-match* HTTP header.

Completely updates (i.e. replaces) the document identified by *document-handle*. If the document exists and can be updated, then a *HTTP 201* is returned and the "ETag" header field contains the new revision of the document.

If the new document passed in the body of the request contains the *document-handle* in the attribute *_id* and the revision in *_rev*, these attributes will be ignored. Only the URI and the "ETag" header are relevant in order to avoid confusion when using proxies.

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

The body of the response contains a JSON object with the information about the handle and the revision. The attribute *_id* contains the known *document-handle* of the updated document, the attribute *_rev* contains the new document revision.

If the document does not exist, then a *HTTP 404* is returned and the body of the response contains an error document.

There are two ways for specifying the targeted document revision id for conditional replacements (i.e. replacements that will only be executed if the revision id found in the database matches the document revision id specified in the request):

- specifying the target revision in the *rev* URL query parameter
- specifying the target revision in the *if-match* HTTP header

Specifying a target revision is optional, however, if done, only one of the described mechanisms must be used (either the *rev* URL parameter or the *if-match* HTTP header). Regardless which mechanism is used, the parameter needs to contain the target document revision id as returned in the *_rev* attribute of a document or by an HTTP *etag* header.

For example, to conditionally replace a document based on a specific revision id, you can use the following request:

```
PUT /_api/document/document-handle?rev=etag
```

If a target revision id is provided in the request (e.g. via the *etag* value in the *rev* URL query parameter above), ArangoDB will check that the revision id of the document found in the database is equal to the target revision id provided in the request. If there is a mismatch between the revision id, then by default a *HTTP 412* conflict is returned and no replacement is performed.

The conditional update behavior can be overriden with the *policy* URL query parameter:

```
PUT /_api/document/document-handle?policy=policy
```

If *policy* is set to *error*, then the behavior is as before: replacements will fail if the revision id found in the database does not match the target revision id specified in the request.

If *policy* is set to *last*, then the replacement will succeed, even if the revision id found in the database does not match the target revision id specified in the request. You can use the *last policy* to force replacements.

Return Codes

- *201*: is returned if the document was replaced successfully and *waitForSync* was *true*.

- *202*: is returned if the document was replaced successfully and *waitForSync* was *false*.

- *400*: is returned if the body does not contain a valid JSON representation of a document. The response body contains an error document in this case.

- *404*: is returned if the collection or the document was not found

- *412*: is returned if a "If-Match" header or *rev* is given and the found document has a different version. The response will also contain the found document's current revision in the *_rev* attribute. Additionally, the attributes *_id* and *_key* will be returned.

Examples

Using document handle:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/document/products/1209164469
{"Hello": "you"}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1209492149"
```

```
location: /_db/_system/_api/document/products-8/1209164469

{
  "error" : false,
  "_id" : "products/1209164469",
  "_rev" : "1209492149",
  "_key" : "1209164469"
}
```

Unknown document handle:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/document/products/1210016437
{}

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "errorMessage" : "document /_api/document/products/1210016437 not
found",
  "code" : 404,
  "errorNum" : 1202
}
```

Produce a revision conflict:

```
shell> curl -X PUT --header 'If-Match: "1211261621"' --data-binary
@- --dump - http://localhost:8529/_api/document/products/1210933941
{"other":"content"}

HTTP/1.1 412 Precondition Failed
content-type: application/json; charset=utf-8
etag: "1210933941"

{
  "error" : true,
  "code" : 412,
  "errorNum" : 1200,
  "errorMessage" : "precondition failed",
  "_id" : "products/1210933941",
  "_rev" : "1210933941",
  "_key" : "1210933941"
}
```

Last write wins:

```
shell> curl -X PUT --header 'If-Match: "1212375733"' --data-binary
@- --dump -
http://localhost:8529/_api/document/products/1212048053?policy=last
{}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1212637877"
```

```
location: /_db/_system/_api/document/products/1212048053

{
  "error" : false,
  "_id" : "products/1212048053",
  "_rev" : "1212637877",
  "_key" : "1212048053"
}
```

Alternative to header field:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/document/products/1213162165?
rev=1213489845
{"other":"content"}

HTTP/1.1 412 Precondition Failed
content-type: application/json; charset=utf-8
etag: "1213162165"

{
  "error" : true,
  "code" : 412,
  "errorNum" : 1200,
  "errorMessage" : "precondition failed",
  "_id" : "products/1213162165",
  "_rev" : "1213162165",
  "_key" : "1213162165"
}
```

updates a document

Patch document

```
PATCH /_api/document/document-handle
```

- *document*: A JSON representation of the document update.

URL Parameters

- *document-handle*: The handle of the document.

Query Parameters

- *keepNull*: If the intention is to delete existing attributes with the patch command, the URL query parameter *keepNull* can be used with a value of *false*. This will modify the behavior of the patch command to remove any attributes from the existing document that are contained in the patch document with an attribute value of *null*.

- *waitForSync*: Wait until document has been synced to disk.

- *rev*: You can conditionally patch a document based on a target revision id by using the *rev* URL parameter.

- *policy*: To control the update behavior in case there is a revision mismatch, you can use the *policy* parameter.

Header Parameters

- *If-Match*: You can conditionally patch a document based on a target revision id by using the *if-match* HTTP header.

Partially updates the document identified by *document-handle*. The body of the request must contain a JSON document with the attributes to patch (the patch document). All attributes from the patch document will be added to the existing document if they do not yet exist, and overwritten in the existing document if they do exist there.

Setting an attribute value to *null* in the patch document will cause a value of *null* be saved for the attribute by default.

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

The body of the response contains a JSON object with the information about the handle and the revision. The attribute *_id* contains the known *document-handle* of the updated document, the attribute *_rev* contains the new document revision.

If the document does not exist, then a *HTTP 404* is returned and the body of the response contains an error document.

You can conditionally update a document based on a target revision id by using either the *rev* URL parameter or the *if-match* HTTP header. To control the update behavior in case there is a revision mismatch, you can use the *policy* parameter. This is the same as when replacing documents (see replacing documents for details).

Return Codes

- *201*: is returned if the document was created successfully and *waitForSync* was *true*.

- *202*: is returned if the document was created successfully and *waitForSync* was *false*.

- *400*: is returned if the body does not contain a valid JSON representation of a document. The response body contains an error document in this case.

- *404*: is returned if the collection or the document was not found

- *412*: is returned if a "If-Match" header or *rev* is given and the found document has a different version. The response will also contain the found document's current revision in the *_rev* attribute. Additionally, the attributes *_id* and *_key* will be returned.

Examples

patches an existing document with new content.

```
shell> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/document/products/1214276277
{
  "hello" : "world"
}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1214603957"
location: /_db/_system/_api/document/products/1214276277
```

```
{
  "error" : false,
  "_id" : "products/1214276277",
  "_rev" : "1214603957",
  "_key" : "1214276277"
}
shell> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/document/products/1214276277
{
  "numbers" : {
    "one" : 1,
    "two" : 2,
    "three" : 3,
    "empty" : null
  }
}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1215193781"
location: /_db/_system/_api/document/products/1214276277

{
  "error" : false,
  "_id" : "products/1214276277",
  "_rev" : "1215193781",
  "_key" : "1214276277"
}
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/document/products/1214276277
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: "1215193781"

{
  "one" : "world",
  "hello" : "world",
  "numbers" : {
    "empty" : null,
    "one" : 1,
    "two" : 2,
    "three" : 3
  },
  "_id" : "products/1214276277",
  "_rev" : "1215193781",
  "_key" : "1214276277"
}
shell> curl -X PATCH --data-binary @- --dump -
http://localhost:8529/_api/document/products/1214276277?
keepNull=false
{
  "hello" : null,
  "numbers" : {
    "four" : 4
  }
}

HTTP/1.1 202 Accepted
```

```
content-type: application/json; charset=utf-8
etag: "1215652533"
location: /_db/_system/_api/document/products/1214276277

{
  "error" : false,
  "_id" : "products/1214276277",
  "_rev" : "1215652533",
  "_key" : "1214276277"
}
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/document/products/1214276277
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: "1215652533"

{
  "one" : "world",
  "numbers" : {
    "empty" : null,
    "one" : 1,
    "two" : 2,
    "three" : 3,
    "four" : 4
  },
  "_id" : "products/1214276277",
  "_rev" : "1215652533",
  "_key" : "1214276277"
}
```

deletes a document

Deletes document

```
DELETE /_api/document/document-handle
```

URL Parameters

- *document-handle*: Deletes the document identified by *document-handle*.

Query Parameters

- *rev*: You can conditionally delete a document based on a target revision id by using the *rev* URL parameter.

- *policy*: To control the update behavior in case there is a revision mismatch, you can use the *policy* parameter. This is the same as when replacing documents (see replacing documents for more details).

- *waitForSync*: Wait until document has been synced to disk.

Header Parameters

- *If-Match*: You can conditionally delete a document based on a target revision id by using the *if-match* HTTP header.

The body of the response contains a JSON object with the information about the handle and the revision. The attribute *_id* contains the known *document-handle* of the deleted document, the attribute *_rev*

contains the document revision.

If the *waitForSync* parameter is not specified or set to *false*, then the collection's default *waitForSync* behavior is applied. The *waitForSync* URL parameter cannot be used to disable synchronisation for collections that have a default *waitForSync* value of *true*.

Return Codes

- *200*: is returned if the document was deleted successfully and *waitForSync* was *true*.

- *202*: is returned if the document was deleted successfully and *waitForSync* was *false*.

- *404*: is returned if the collection or the document was not found. The response body contains an error document in this case.

- *412*: is returned if a "If-Match" header or *rev* is given and the found document has a different version. The response will also contain the found document's current revision in the *_rev* attribute. Additionally, the attributes *_id* and *_key* will be returned.

Examples

Using document handle:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/document/products/1216242357
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "error" : false,
  "_id" : "products/1216242357",
  "_rev" : "1216242357",
  "_key" : "1216242357"
}
```

Unknown document handle:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/document/products/1216963253
undefined

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "errorMessage" : "document /_api/document/products/1216963253 not
found",
  "code" : 404,
  "errorNum" : 1202
}
```

Revision conflict:

```
shell> curl -X DELETE --header 'If-Match: "1218142901"' --dump -
```

```
http://localhost:8529/_api/document/products/1217815221

HTTP/1.1 412 Precondition Failed
content-type: application/json; charset=utf-8
etag: "1217815221"

{
  "error" : true,
  "code" : 412,
  "errorNum" : 1200,
  "errorMessage" : "precondition failed",
  "_id" : "products/1217815221",
  "_rev" : "1217815221",
  "_key" : "1217815221"
}
```

reads a single document head

Read document header

```
HEAD /_api/document/document-handle
```

URL Parameters

- *document-handle*: The handle of the document.

Query Parameters

- *rev*: You can conditionally fetch a document based on a target revision id by using the *rev* URL parameter.

Header Parameters

- *If-None-Match*: If the "If-None-Match" header is given, then it must contain exactly one etag. If the current document revision is different to the specified etag, an *HTTP 200* response is returned. If the current document revision is identical to the specified etag, then an *HTTP 304* is returned.

- *If-Match*: You can conditionally fetch a document based on a target revision id by using the *if-match* HTTP header.

Like *GET*, but only returns the header fields and not the body. You can use this call to get the current revision of a document or check if the document was deleted.

Return Codes

- *200*: is returned if the document was found

- *304*: is returned if the "If-None-Match" header is given and the document has same version *
- *404*: is returned if the document or collection was not found

- *412*: is returned if a "If-Match" header or *rev* is given and the found document has a different version. The response will also contain the found document's current revision in the *etag* header.

Examples

```
shell> curl -X HEAD --data-binary @- --dump -
http://localhost:8529/_api/document/products/1208574645
undefined
```

reads all documents from collection

Read all documents

```
GET /_api/document
```

Query Parameters

- *collection*: The name of the collection.

Returns a list of all URI for all documents from the collection identified by *collection*.

Return Codes

- *200*: All went good.

- *404*: The collection does not exist.

Examples

Returns a all ids.

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/document/?collection=products
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "documents" : [
    "/_api/document/products/1207984821",
    "/_api/document/products/1207329461",
    "/_api/document/products/1207657141"
  ]
}
```

Collection does not exist.

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/document/?collection=doesnotexist
undefined

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "errorMessage" : "collection 'doesnotexist' not found",
  "code" : 404,
  "errorNum" : 1203
}
```

HTTP Interface for Endpoints

The ArangoDB server can listen for incoming requests on multiple *endpoints*.

The endpoints are normally specified either in ArangoDB's configuration file or on the command-line, using the "--server.endpoint" option. The default endpoint for ArangoDB is *tcp://127.0.0.1:8529* or *tcp://localhost:8529*.

The number of endpoints can also be changed at runtime using the API described below. Each endpoint can optionally be restricted to a specific list of databases only, thus allowing the usage of different port numbers for different databases.

This may be useful in multi-tenant setups. A multi-endpoint setup may also be useful to turn on encrypted communication for just specific databases.

The HTTP interface provides operations to add new endpoints at runtime, and optionally restrict them for use with specific databases. The interface also can be used to update existing endpoints or remove them at runtime.

Please note that all endpoint management operations can only be accessed via the default database (*_system*) and none of the other databases.

Managing Endpoints via HTTP

connects a new endpoint or reconfigures an existing endpoint

Add new endpoint or reconfigures an existing endpoint

```
POST /_api/endpoint
```

- *description*: A JSON object describing the endpoint.

The request body must be JSON hash with the following attributes:

- *endpoint*: the endpoint specification, e.g. *tcp://127.0.0.1:8530*

- *databases*: a list of database names the endpoint is responsible for.

If *databases* is an empty list, all databases present in the server will become accessible via the endpoint, with the *_system* database being the default database.

If *databases* is non-empty, only the specified databases will become available via the endpoint. The first database name in the *databases* list will also become the default database for the endpoint. The default database will always be used if a request coming in on the endpoint does not specify the database name explicitly.

Note: adding or reconfiguring endpoints is allowed in the system database only. Calling this action in any other database will make the server return an error.

Adding SSL endpoints at runtime is only supported if the server was started with SSL properly configured (e.g. *--server.keyfile* must have been set).

Return Codes

- *200*: is returned when the endpoint was added or changed successfully.

- *400*: is returned if the request is malformed or if the action is not carried out in the system database.

- *405*: The server will respond with *HTTP 405* if an unsupported HTTP method is used.

Examples Adding an endpoint *tcp://127.0.0.1:8532* with two mapped databases (*mydb1* and *mydb2*). *mydb1* will become the default database for the endpoint.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/endpoint
{"endpoint":"tcp://127.0.0.1:8532","databases":["mydb1","mydb2"]}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
```

Adding an endpoint *tcp://127.0.0.1:8533* with no database names specified. This will allow access to all databases on this endpoint. The *_system* database will become the default database for requests that come in on this endpoint and do not specify the database name explicitly.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/endpoint
{"endpoint":"tcp://127.0.0.1:8533","databases":[]}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
```

Adding an endpoint *tcp://127.0.0.1:8533* without any databases first, and then updating the databases for the endpoint to *testdb1*, *testdb2*, and *testdb3*.

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/endpoint
{"endpoint":"tcp://127.0.0.1:8533","databases":[]}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/endpoint
{"endpoint":"tcp://127.0.0.1:8533","databases":[],"database":
["testdb1","testdb2","testdb3"]}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
```

disconnects an existing endpoint

Delete and disconnects an existing endpoint

```
DELETE /_api/endpoint/{endpoint}
```

URL Parameters

- *endpoint*: The endpoint to delete, e.g. *tcp://127.0.0.1:8529*.

This operation deletes an existing endpoint from the list of all endpoints, and makes the server stop listening on the endpoint.

Note: deleting and disconnecting an endpoint is allowed in the system database only. Calling this action in any other database will make the server return an error.

Futhermore, the last remaining endpoint cannot be deleted as this would make the server kaputt.

Return Codes

- *200*: is returned when the endpoint was deleted and disconnected successfully.

- *400*: is returned if the request is malformed or if the action is not carried out in the system database.

- *404*: is returned if the endpoint is not found.

- *405*: The server will respond with *HTTP 405* if an unsupported HTTP method is used.

Examples

Deleting an existing endpoint

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/endpoint/tcp%3A%2F%2F127.0.0.1%3A8532
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
```

Deleting a non-existing endpoint

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/endpoint/tcp%3A%2F%2F127.0.0.1%3A8532
undefined

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 404,
  "errorNum" : 1231,
  "errorMessage" : "endpoint not found"
}
```

returns a list of all endpoints

Return list of all endpoints

```
GET /_api/endpoint
```

Returns a list of all configured endpoints the server is listening on. For each endpoint, the list of allowed databases is returned too if set.

The result is a JSON hash which has the endpoints as keys, and the list of mapped database names as values for each endpoint.

If a list of mapped databases is empty, it means that all databases can be accessed via the endpoint. If a list of mapped databases contains more than one database name, this means that any of the databases might be accessed via the endpoint, and the first database in the list will be treated as the default database for the endpoint. The default database will be used when an incoming request does not specify a database name in the request explicitly.

Note: retrieving the list of all endpoints is allowed in the system database only. Calling this action in any other database will make the server return an error.

Return Codes

- *200*: is returned when the list of endpoints can be determined successfully.

- *400*: is returned if the action is not carried out in the system database.

- *405*: The server will respond with *HTTP 405* if an unsupported HTTP method is used.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/endpoint
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

[
  {
    "endpoint" : "tcp://127.0.0.1:39549",
    "databases" : [ ]
  },
  {
    "endpoint" : "tcp://127.0.0.1:8532",
    "databases" : [
      "mydb1",
      "mydb2"
    ]
  }
]
```

HTTP Interface for Sharding

executes a cluster roundtrip for sharding

Execute cluster roundtrip

`GET /_admin/cluster-test`

Executes a cluster roundtrip from a coordinator to a DB server and back. This call only works in a coordinator node in a cluster. One can and should append an arbitrary path to the URL and the part after */_admin/cluster-test* is used as the path of the HTTP request which is sent from the coordinator to a DB node. Likewise, any form data appended to the URL is forwarded in the request to the DB node. This handler takes care of all request types (see below) and uses the same request type in its request to the DB node.

The following HTTP headers are interpreted in a special way:

- *X-Shard-ID*: This specifies the ID of the shard to which the cluster request is sent and thus tells the system to which DB server to send the cluster request. Note that the mapping from the shard ID to the responsible server has to be defined in the agency under *Current/ShardLocation/*. One has to give this header, otherwise the system does not know where to send the request.
- *X-Client-Transaction-ID*: the value of this header is taken as the client transaction ID for the request
- *X-Timeout*: specifies a timeout in seconds for the cluster operation. If the answer does not arrive within the specified timeout, an corresponding error is returned and any subsequent real answer is ignored. The default if not given is 24 hours.
- *X-Synchronous-Mode*: If set to *true* the test function uses synchronous mode, otherwise the default asynchronous operation mode is used. This is mainly for debugging purposes.
- *Host*: This header is ignored and not forwarded to the DB server.
- *User-Agent*: This header is ignored and not forwarded to the DB server.

All other HTTP headers and the body of the request (if present, see other HTTP methods below) are forwarded as given in the original request.

In asynchronous mode the DB server answers with an HTTP request of its own, in synchronous mode it sends a HTTP response. In both cases the headers and the body are used to produce the HTTP response of this API call.

Return Codes

The return code can be anything the cluster request returns, as well as:

- *200*: is returned when everything went well, or if a timeout occurred. In the latter case a body of type application/json indicating the timeout is returned.

- *403*: is returned if ArangoDB is not running in cluster mode.

- *404*: is returned if ArangoDB was not compiled for cluster operation.

  executes a cluster roundtrip for sharding

Execute cluster roundtrip

`POST /_admin/cluster-test`

- *body*:

See GET method. The body can be any type and is simply forwarded.

executes a cluster roundtrip for sharding

Execute cluster roundtrip

`PUT /_admin/cluster-test`

- *body*:

See GET method. The body can be any type and is simply forwarded. //

executes a cluster roundtrip for sharding

Delete cluster roundtrip

`DELETE /_admin/cluster-test`

See GET method. The body can be any type and is simply forwarded.

executes a cluster roundtrip for sharding

Update cluster roundtrip

`PATCH /_admin/cluster-test`

- *body*:

See GET method. The body can be any type and is simply forwarded.

executes a cluster roundtrip for sharding

Execute cluster roundtrip

`HEAD /_admin/cluster-test`

See GET method. The body can be any type and is simply forwarded.

exposes the cluster planning functionality

Produce cluster startup plan

`POST /_admin/clusterPlanner`

- *body*:

Given a description of a cluster, this plans the details of a cluster and returns a JSON description of a plan to start up this cluster. See @ref JSF_Cluster_Planner_Constructor for details.

Return Codes

- *200*: is returned when everything went well.

- *400*: the posted body was not valid JSON.

  exposes the dispatcher functionality to start up, shutdown, relaunch, upgrade or cleanup a cluster according to a cluster plan as for example provided by the kickstarter.

execute startup commands

`POST /_admin/clusterDispatch`

Query Parameters

- *body*:

The body must be an object with the following properties:

- *clusterPlan*: is a cluster plan (see JSF_cluster_planner_POST),
- *myname*: is the ID of this dispatcher, this is used to decide which commands are executed locally and which are forwarded to other dispatchers
- *action*: can be one of the following:

- "launch": the cluster is launched for the first time, all data directories and log files are cleaned and created
- "shutdown": the cluster is shut down, the additional property *runInfo* (see below) must be bound as well
- "relaunch": the cluster is launched again, all data directories and log files are untouched and need to be there already
- "cleanup": use this after a shutdown to remove all data in the data directories and all log files, use with caution
- "isHealthy": checks whether or not the processes involved in the cluster are running or not. The additional property *runInfo* (see above) must be bound as well
- "upgrade": performs an upgrade of a cluster, to this end, the agency is started, and then every server is once started with the "--upgrade" option, and then normally. Finally, the script "verion-check.js" is run on one of the coordinators for the cluster.

- *runInfo*: this is needed for the "shutdown" and "isHealthy" actions only and should be the structure that "launch", "relaunch" or "upgrade" returned. It contains runtime information like process IDs.

This call executes the plan by either doing the work personally or by delegating to other dispatchers.

Return Codes

- *200*: is returned when everything went well.

- *400*: the posted body was not valid JSON, or something went wrong with the startup.

  allows to check whether a given port is usable

Check port

```
GET /_admin/clusterCheckPort
```

Query Parameters

- *port*:

Checks whether the requested port is usable.

Return Codes

- *200*: is returned when everything went well.

- *400*: the parameter port was not given or is no integer.

HTTP Interface for Miscellaneous functions

This is an overview of ArangoDB's HTTP interface for miscellaneous functions.

returns the server version number

Return server version

```
GET /_api/version
```

Query Parameters

- *details*: If set to *true*, the response will contain a *details* attribute with additional information about included components and their versions. The attribute names and internals of the *details* object may vary depending on platform and ArangoDB version.

Returns the server name and version number. The response is a JSON object with the following attributes:

- *server*: will always contain *arango*

- *version*: the server version string. The string has the format "*major.minor.sub*". *major* and *minor* will be numeric, and *sub* may contain a number or a textual version.

- *details*: an optional JSON object with additional details. This is returned only if the *details* URL parameter is set to *true* in the request.

Return Codes

- *200*: is returned in all cases.

Examples

Returns the version information.

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/version
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "server" : "arango",
  "version" : "2.2.0-devel"
}
```

Returns the version information with details.

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/version?details=true
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "server" : "arango",
```

```
    "version" : "2.2.0-devel",
    "details" : {
      "build-date" : "Jun 30 2014 11:22:29",
      "configure" : "'./configure' '--enable-all-in-one-icu' '--
enable-maintainer-mode' '--with-
bison=/usr/local/Cellar/bison/3.0.2/bin/bison'",
      "icu-version" : "52.1",
      "libev-version" : "4.11",
      "openssl-version" : "OpenSSL 0.9.8y 5 Feb 2013",
      "repository-version" : "heads/devel-0-
g08f829f801d459bc764c0481bae29dab084a79e2-dirty",
      "server-version" : "2.2.0-devel",
      "sizeof int" : "4",
      "sizeof void*" : "8",
      "v8-version" : "3.16.14"
  }
}
```

returns the system time

Return system time

`GET /_admin/time`

The call returns an object with the attribute *time*. This contains the current system time as a Unix timestamp with microsecond precision.

Return Codes

- *200*: Time was returned successfully.

  returns the request

Return current request

`GET /_admin/echo`

The call returns an object with the following attributes:

- *headers*: a list of HTTP headers received

- *requestType*: the HTTP request method (e.g. GET)

- *parameters*: list of URL parameters received

Return Codes

- *200*: Echo was returned successfully.

  initiates the shutdown sequence

Initiate shutdown sequence

`GET /_admin/shutdown`

This call initiates a clean shutdown sequence.

Return Codes

- *200*: is returned in all cases.

General HTTP Request Handling in ArangoDB

Protocol

ArangoDB exposes its API via HTTP, making the server accessible easily with a variety of clients and tools (e.g. browsers, curl, telnet). The communication can optionally be SSL-encrypted.

ArangoDB uses the standard HTTP methods (e.g. *GET*, *POST*, *PUT*, *DELETE*) plus the *PATCH* method described in [RFC 5789](#).

Most server APIs expect clients to send any payload data in [JSON](#) format. Details on the expected format and JSON attributes can be found in the documentation of the individual server methods.

Clients sending requests to ArangoDB must use either HTTP 1.0 or HTTP 1.1. Other HTTP versions are not supported by ArangoDB and any attempt to send a different HTTP version signature will result in the server responding with an HTTP 505 (HTTP version not supported) error.

ArangoDB will always respond to client requests with HTTP 1.1. Clients should therefore support HTTP version 1.1.

Blocking vs. Non-blocking Requests

ArangoDB supports both blocking and non-blocking requests.

ArangoDB is a multi-threaded server, allowing the processing of multiple client requests at the same time. Request/response handling and the actual work are performed on the server in parallel by multiple worker threads.

Still, clients need to wait for their requests to be processed by the server. By default, the server will fully process an incoming request and then return the result to the client when the operation is finished. The client must wait for the server's response before it can send additional requests over the same connection. For clients that are single-threaded and/or are blocking on I/O themselves, waiting idle for the server response may be non-optimal.

To reduce blocking on the client side, ArangoDB since version 1.4 offers a generic mechanism for non-blocking, asynchronous execution: clients can add the HTTP header *x-arango-async: true* to any of their requests, marking them as to be executed asynchronously on the server. ArangoDB will put such requests into an in-memory task queue and return an *HTTP 202* (accepted) response to the client instantly. The server will execute the tasks from the queue asynchronously as fast as possible, while clients can continue to work. If the server queue is full (i.e. contains as many tasks as specified by the option ["--scheduler.maximal-queue-size"](#), then the request will be rejected instantly with an *HTTP 500* (internal server error) response.

Asynchronous execution decouples the request/response handling from the actual work to be performed, allowing fast server responses and greatly reducing wait time for clients. Overall this allows for much higher throughput than if clients would always wait for the server's response.

Keep in mind that the asynchronous execution is just "fire and forget". Clients will get any of their asynchronous requests answered with a generic HTTP 202 response. At the time the server sends this response, it does not know whether the requested operation can be carried out successfully (the actual operation execution will happen at some later point). Clients therefore cannot make a decision based on the server response and must rely on their requests being valid and processable by the server.

Additionally, the server's asynchronous task queue is an in-memory data structure, meaning not-yet processed tasks from the queue might be lost in case of a crash. Clients should therefore not use the asynchronous feature when they have strict durability requirements or if they rely on the immediate result of the request they send.

HTTP Keep-Alive

ArangoDB supports HTTP keep-alive. If the client does not send a *Connection* header in its request, and the client uses HTTP version 1.1, ArangoDB will assume the client wants to keep alive the connection. If clients do not wish to use the keep-alive feature, they should explicitly indicate that by sending a *Connection: Close* HTTP header in the request.

ArangoDB will close connections automatically for clients that send requests using HTTP 1.0, except if they send an *Connection: Keep-Alive* header.

The default Keep-Alive timeout can be specified at server start using the *--server.keep-alive-timeout* parameter.

Authentication

Client authentication can be achieved by using the *Authorization* HTTP header in client requests. ArangoDB supports HTTP Basic authentication.

Authentication is optional. To enforce authentication for incoming requested, the server must be started with the option [--server.disable-authentication](). Please note that requests using the HTTP OPTIONS method will be answered by ArangoDB in any case, even if no authentication data is sent by the client or if the authentication data is wrong. This is required for handling CORS preflight requests (see [Cross Origin Resource Sharing requests]() requests)). The response to an HTTP OPTIONS request will be generic and not expose any private data.

Please note that when authentication is turned on in ArangoDB, it will by default affect all incoming requests.

Since ArangoDB 1.4, there is an additional option [-server.authenticate-system-only]() "--server.authenticate-system-only" to restrict authentication to requests to the ArangoDB internal APIs and the admin interface. This option can be used to expose a public API built with ArangoDB to the outside world without the need for HTTP authentication, but to still protect the usage of the ArangoDB API (i.e. */_api/**) and the admin interface (i.e.* /_admin/**) with HTTP authentication.

If the server is started with the *--server.authenticate-system-only* parameter set to *false* (which is the default), all incoming requests need HTTP authentication if the server is configured to require HTTP authentication. Setting the option to *false* will make the server require authentication only for requests to the internal functionality at */_api/* or */_admin* and will allow unauthenticated requests to all other URLs.

Whenever authentication is required and the client has not yet authenticated, ArangoDB will return *HTTP 401* (Unauthorized). It will also send the *WWW-Authenticate* response header, indicating that the client should prompt the user for username and password if supported. If the client is a browser, then sending back this header will normally trigger the display of the browser-side HTTP authentication dialog. As showing the browser HTTP authentication dialog is undesired in AJAX requests, ArangoDB can be told to not send the *WWW-Authenticate* header back to the client. Whenever a client sends the *X-Omit-WWW-Authenticate* HTTP header (with an arbitrary value) to ArangoDB, ArangoDB will only send status code 401, but no *WWW-Authenticate* header. This allows clients to implement credentials handling and bypassing the browser's built-in dialog.

Error Handling

The following should be noted about how ArangoDB handles client errors in its HTTP layer:

- client requests using an HTTP version signature different than *HTTP/1.0* or *HTTP/1.1* will get an *HTTP 505* (HTTP version not supported) error in return.
- ArangoDB will reject client requests with a negative value in the *Content-Length* request header with *HTTP 411* (Length Required).
- the maximum URL length accepted by ArangoDB is 16K. Incoming requests with longer URLs will be rejected with an *HTTP 414* (Request-URI too long) error.
- if the client sends a *Content-Length* header with a value bigger than 0 for an HTTP GET, HEAD, or DELETE request, ArangoDB will process the request, but will write a warning to its log file.
- when the client sends a *Content-Length* header that has a value that is lower than the actual size of the body sent, ArangoDB will respond with *HTTP 400* (Bad Request).
- if clients send a *Content-Length* value bigger than the actual size of the body of the request, ArangoDB will wait for about 90 seconds for the client to complete its request. If the client does not send the remaining body data within this time, ArangoDB will close the connection. Clients should avoid sending such malformed requests as they will make ArangoDB block waiting for more data to arrive.
- when clients send a body or a *Content-Length* value bigger than the maximum allowed value (512 MB), ArangoDB will respond with *HTTP 413* (Request Entity Too Large).
- if the overall length of the HTTP headers a client sends for one request exceeds the maximum allowed size (1 MB), the server will fail with *HTTP 431* (Request Header Fields Too Large).

- if clients request a HTTP method that is not supported by the server, ArangoDB will return with *HTTP 405* (Method Not Allowed). ArangoDB offers general support for the following HTTP methods:

  - GET
  - POST
  - PUT
  - DELETE
  - HEAD
  - PATCH
  - OPTIONS

  Please note that not all server actions allow using all of these HTTP methods. You should look up up the supported methods for each method you intend to use in the manual.

  Requests using any other HTTP method (such as for example CONNECT, TRACE etc.) will be rejected by ArangoDB.

Cross Origin Resource Sharing (CORS) requests

ArangoDB will automatically handle CORS requests as follows:

- when the client sends an *Origin* HTTP header, ArangoDB will return a header *access-control-allow-origin* containing the value the client sent in the *Origin* header.
- for non-trivial CORS requests, clients may issue a preflight request via an additional HTTP OPTIONS request. ArangoDB will automatically answer such preflight HTTP OPTIONS requests with an HTTP 200 response with an empty body. ArangoDB will return the following headers in the response:
  - *access-control-allow-origin*: will contain the value that the client provided in the *Origin* header of the request
  - *access-control-allow-methods*: will contain a list of all HTTP methods generally supported by ArangoDB. This list does not depend on the URL the client requested and is the same for all CORS requests.
  - *access-control-allow-headers*: will contain exactly the value that the client has provided in the *Access-Control-Request-Header* header of the request. This header will only be returned if the client has specified the header in the request. ArangoDB will send back the original value without further validation.
  - *access-control-max-age*: will return a cache lifetime for the preflight response as determined by ArangoDB.
- any *access-control-allow-credentials* header sent by the client is ignored by ArangoDB its value is not *true*. If a client sends a header value of *true*, ArangoDB will return the header *access-control-allow-credentials: true*, too.

Note that CORS preflight requests will probably not send any authentication data with them. One of the purposes of the preflight request is to check whether the server accepts authentication or not.

A consequence of this is that ArangoDB will allow requests using the HTTP OPTIONS method without credentials, even when the server is run with authentication enabled.

The response to the HTTP OPTIONS request will however be a generic response that will not expose any private data and thus can be considered "safe" even without credentials.

HTTP method overriding

Since version 1.4, ArangoDB provides a startup option *--server.allow-method-override*. This option can be set to allow overriding the HTTP request method (e.g. GET, POST, PUT, DELETE, PATCH) of a request using one of the following custom HTTP headers:

- *x-http-method-override*
- *x-http-method*
- *x-method-override*

This allows using HTTP clients that do not support all "common" HTTP methods such as PUT, PATCH and DELETE. It also allows bypassing proxies and tools that would otherwise just let certain types of requests (e.g. GET and POST) pass through.

Enabling this option may impose a security risk, so it should only be used in very controlled environments. Thus the default value for this option is *false* (no method overriding allowed). You need to enable it explicitly if you want to use this feature.

HTTP Interface for Edges

This is an introduction to ArangoDB's REST interface for edges.

ArangoDB offers also some graph functionality. A graph consists of nodes, edges and properties. ArangoDB stores the information how the nodes relate to each other aside from the properties.

A graph data model always consists of two collections: the relations between the nodes in the graphs are stored in an "edges collection", the nodes in the graph are stored in documents in regular collections.

Example:

- the "edge" collection stores the information that a company's reception is sub-unit to the services unit and the services unit is sub-unit to the CEO. You would express this relationship with the *to* and _*to* attributes
- the "normal" collection stores all the properties about the reception, e.g. that 20 people are working there and the room number etc
- _*from* is the document handle of the linked vertex (incoming relation)
- _*to* is the document handle of the linked vertex (outgoing relation)

Documents, Identifiers, Handles

Edges in ArangoDB are special documents. In addition to the internal attributes _key, _id and _rev, they have two attributes _from and _to, which contain document handles, namely the start-point and the end-point of the edge.

The values of _from and _to are immutable once saved.

Address and ETag of an Edge

All documents in ArangoDB have a document handle. This handle uniquely identifies a document. Any document can be retrieved using its unique URI:

```
http://server:port/_api/document/<document-handle>
```

Edges are a special variation of documents, and to work with edges, the above URL format changes to:

```
http://server:port/_api/edge/<document-handle>
```

For example, assumed that the document handle, which is stored in the *_id* attribute of the edge, is *demo/362549736*, then the URL of that edge is:

```
http://localhost:8529/_api/edge/demo/362549736
```

The above URL scheme does not specify a database name explicitly, so the default database will be used. To explicitly specify the database context, use the following URL schema:

```
http://server:port/_db/<database-name>/_api/edge/<document-handle>
```

Example:

```
http://localhost:8529/_db/mydb/_api/edge/demo/362549736
```

Note: that the following examples use the short URL format for brevity.

Working with Edges using REST

Read edge

```
GET /_api/edge
```

See [HTTP Interface for Documents](#) for details.

creates an edge

Create edge

```
POST /_api/edge
```

- *edge-document*: A JSON representation of the edge document must be passed as the body of the POST request. This JSON object may contain the edge's document key in the _key attribute if needed.

Query Parameters

- *collection*: Creates a new edge in the collection identified by *collection* name.

- *createCollection*: If this parameter has a value of *true* or *yes*, then the collection is created if it does not yet exist. Other values will be ignored so the collection must be present for the operation to succeed.

Note: This flag is not supported in a cluster. Using it will result in an error.

- *waitForSync*: Wait until the edge document has been synced to disk.

- *from*: The document handle of the start point must be passed in *from* handle.

- *to*: The document handle of the end point must be passed in *to* handle.

Creates a new edge document in the collection named *collection*. A JSON representation of the document must be passed as the body of the POST request.

The *from* and *to* handles are immutable once the edge has been created.

In all other respects the method works like *POST /document*.

Return Codes

- *201*: is returned if the edge was created successfully and *waitForSync* was *true*.

- *202*: is returned if the edge was created successfully.

- *400*: is returned if the body does not contain a valid JSON representation of an edge, or if the collection specified is not an edge collection. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Create an edge and read it back:

```
shell> curl -X POST --data-binary @- --dump -
```

```
http://localhost:8529/_api/edge/?
collection=edges&from=vertices/1&to=vertices/2
{
  "name" : "Emil"
}

HTTP/1.1 202 Accepted
content-type: application/json; charset=utf-8
etag: "1220502197"
location: /_db/_system/_api/document/edges/1220502197

{
  "error" : false,
  "_id" : "edges/1220502197",
  "_rev" : "1220502197",
  "_key" : "1220502197"
}
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/edge/edges/1220502197
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: "1220502197"

{
  "name" : "Emil",
  "_id" : "edges/1220502197",
  "_rev" : "1220502197",
  "_key" : "1220502197",
  "_from" : "vertices/1",
  "_to" : "vertices/2"
}
```

Update edge

PUT /_api/edge

See [HTTP Interface for Documents](#) for details.

Partially Update Edge

PATCH /_api/edge

See [HTTP Interface for Documents](#) for details.

Delete edge

DELETE /_api/edge

See [HTTP Interface for Documents](#) for details.

Read edge header

HEAD /_api/edge

See [HTTP Interface for Documents](#) for details.

get edges

Read in- or outbound edges

```
GET /_api/edges/collection-id
```

URL Parameters

- *collection-id*: The id of the collection.

Query Parameters

- *vertex*: The id of the start vertex.

- *direction*: Selects *in* or *out* direction for edges. If not set, any edges are returned.

Returns the list of edges starting or ending in the vertex identified by *vertex-handle*.

Examples

Any direction

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/edges/edges?vertex=vertices/1
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "edges" : [
    {
      "_id" : "edges/6",
      "_rev" : "897868469",
      "_key" : "6",
      "_from" : "vertices/2",
      "_to" : "vertices/1",
      "$label" : "v2 -> v1"
    },
    {
      "_id" : "edges/7",
      "_rev" : "898392757",
      "_key" : "7",
      "_from" : "vertices/4",
      "_to" : "vertices/1",
      "$label" : "v4 -> v1"
    },
    {
      "_id" : "edges/5",
      "_rev" : "897344181",
      "_key" : "5",
      "_from" : "vertices/1",
      "_to" : "vertices/3",
      "$label" : "v1 -> v3"
    }
  ],
  "error" : false,
  "code" : 200
}
```

In edges

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/edges/edges?
vertex=vertices/1&direction=in
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "edges" : [
    {
      "_id" : "edges/6",
      "_rev" : "902652597",
      "_key" : "6",
      "_from" : "vertices/2",
      "_to" : "vertices/1",
      "$label" : "v2 -> v1"
    },
    {
      "_id" : "edges/7",
      "_rev" : "903176885",
      "_key" : "7",
      "_from" : "vertices/4",
      "_to" : "vertices/1",
      "$label" : "v4 -> v1"
    }
  ],
  "error" : false,
  "code" : 200
}
```

Out edges

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/edges/edges?
vertex=vertices/1&direction=out
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "edges" : [
    {
      "_id" : "edges/5",
      "_rev" : "906912437",
      "_key" : "5",
      "_from" : "vertices/1",
      "_to" : "vertices/3",
      "$label" : "v1 -> v3"
    }
  ],
  "error" : false,
  "code" : 200
}
```

HTTP Interface for AQL Query Cursors

Database Cursors

This is an introduction to ArangoDB's HTTP Interface for Queries. Results of AQL and simple queries are returned as cursors in order to batch the communication between server and client. Each call returns a number of documents in a batch and an indication, if the current batch has been the final batch. Depending on the query, the total number of documents in the result set might or might not be known in advance. In order to free server resources the client should delete the cursor as soon as it is no longer needed.

To execute a query, the query details need to be shipped from the client to the server via an HTTP POST request.

Retrieving query results

Select queries are executed on-the-fly on the server and the result set will be returned back to the client.

There are two ways the client can get the result set from the server:

- In a single roundtrip
- Using a cursor

Single roundtrip

The server will only transfer a certain number of result documents back to the client in one roundtrip. This number is controllable by the client by setting the *batchSize* attribute when issuing the query.

If the complete result can be transferred to the client in one go, the client does not need to issue any further request. The client can check whether it has retrieved the complete result set by checking the *hasMore* attribute of the result set. If it is set to *false*, then the client has fetched the complete result set from the server. In this case no server side cursor will be created.

```
> curl --data @- -X POST --dump - http://localhost:8529/_api/cursor
{ "query" : "FOR u IN users LIMIT 2 RETURN u", "count" : true,
"batchSize" : 2 }

HTTP/1.1 201 Created
content-type: application/json

{
  "hasMore" : false,
  "error" : false,
  "result" : [
    {
      "name" : "user1",
      "_rev" : "210304551",
      "_key" : "210304551",
      "_id" : "users/210304551"
    },
    {
      "name" : "user2",
      "_rev" : "210304552",
      "_key" : "210304552",
      "_id" : "users/210304552"
    }
  ],
  "code" : 201,
  "count" : 2
}
```

Using a Cursor

If the result set contains more documents than should be transferred in a single roundtrip (i.e. as set via the *batchSize* attribute), the server will return the first few documents and create a temporary cursor. The cursor identifier will also be returned to the client. The server will put the cursor identifier in the *id* attribute of the response object. Furthermore, the *hasMore* attribute of the response object will be set to *true*. This is an indication for the client that there are additional results to fetch from the server.

Examples:

Create and extract first batch:

```
> curl --data @- -X POST --dump - http://localhost:8529/_api/cursor
{ "query" : "FOR u IN users LIMIT 5 RETURN u", "count" : true,
"batchSize" : 2 }

HTTP/1.1 201 Created
content-type: application/json

{
  "hasMore" : true,
  "error" : false,
  "id" : "26011191",
  "result" : [
    {
      "name" : "user1",
      "_rev" : "258801191",
      "_key" : "258801191",
      "_id" : "users/258801191"
    },
    {
      "name" : "user2",
      "_rev" : "258801192",
      "_key" : "258801192",
      "_id" : "users/258801192"
    }
  ],
  "code" : 201,
  "count" : 5
}
```

Extract next batch, still have more:

```
> curl -X PUT --dump - http://localhost:8529/_api/cursor/26011191

HTTP/1.1 200 OK
content-type: application/json

{
  "hasMore" : true,
  "error" : false,
  "id" : "26011191",
  "result": [
    {
      "name" : "user3",
      "_rev" : "258801193",
      "_key" : "258801193",
      "_id" : "users/258801193"
    },
    {
      "name" : "user4",
      "_rev" : "258801194",
      "_key" : "258801194",
      "_id" : "users/258801194"
    }
  ],
  "code" : 200,
  "count" : 5
```

```
    }
```

Extract next batch, done:

```
> curl -X PUT --dump - http://localhost:8529/_api/cursor/26011191

HTTP/1.1 200 OK
content-type: application/json

{
  "hasMore" : false,
  "error" : false,
  "result" : [
    {
      "name" : "user5",
      "_rev" : "258801195",
      "_key" : "258801195",
      "_id" : "users/258801195"
    }
  ],
  "code" : 200,
  "count" : 5
}
```

Do not do this:

```
> curl -X PUT --dump - http://localhost:8529/_api/cursor/26011191

HTTP/1.1 404 Not Found
content-type: application/json

{
  "errorNum": 1600,
  "errorMessage": "cursor not found: disposed or unknown cursor",
  "error": true,
  "code": 404
}
```

Accessing Cursors via HTTP

create a cursor and return the first results

Create cursor

```
POST /_api/cursor
```

- *query*: A JSON object describing the query and query parameters.

The query details include the query string plus optional query options and bind parameters. These values need to be passed in a JSON representation in the body of the POST request.

The following attributes can be used inside the JSON object:

- *query*: contains the query string to be executed (mandatory)

- *count*: boolean flag that indicates whether the number of documents in the result set should be returned in the "count" attribute of the result (optional). Calculating the "count" attribute might in the future have a performance impact for some queries so this option is turned off by default, and "count" is only returned when requested.

- *batchSize*: maximum number of result documents to be transferred from the server to the client in one roundtrip (optional). If this attribute is not set, a server-controlled default value will be used.

- *bindVars*: key/value list of bind parameters (optional).

- *options*: key/value list of extra options for the query (optional).

The following options are supported at the moment:

- *fullCount*: if set to *true* and the query contains a *LIMIT* clause, then the result will contain an extra attribute *extra* with a sub-attribute *fullCount*. This sub-attribute will contain the number of documents in the result before the last LIMIT in the query was applied. It can be used to count the number of documents that match certain filter criteria, but only return a subset of them, in one go. It is thus similar to MySQL's *SQL_CALC_FOUND_ROWS* hint. Note that setting the option will disable a few LIMIT optimizations and may lead to more documents being processed, and thus make queries run longer. Note that the *fullCount* sub-attribute will only be present in the result if the query has a LIMIT clause and the LIMIT clause is actually used in the query.

If the result set can be created by the server, the server will respond with *HTTP 201*. The body of the response will contain a JSON object with the result set.

The returned JSON object has the following properties:

- *error*: boolean flag to indicate that an error occurred (*false* in this case)

- *code*: the HTTP status code

- *result*: an array of result documents (might be empty if query has no results)

- *hasMore*: a boolean indicator whether there are more results available for the cursor on the server

- *count*: the total number of result documents available (only available if the query was executed with the *count* attribute set)

- *id*: id of temporary cursor created on the server (optional, see above)

- *extra*: an optional JSON object with extra information about the query result

If the JSON representation is malformed or the query specification is missing from the request, the server will respond with *HTTP 400*.

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: boolean flag to indicate that an error occurred (*true* in this case)

- *code*: the HTTP status code

- *errorNum*: the server error number

- *errorMessage*: a descriptive error message

If the query specification is complete, the server will process the query. If an error occurs during query processing, the server will respond with *HTTP 400*. Again, the body of the response will contain details about the error.

A list of query errors can be found @ref ArangoErrors here.

Return Codes

- *201*: is returned if the result set can be created by the server.

- *400*: is returned if the JSON representation is malformed or the query specification is missing from the request.

- *404*: The server will respond with *HTTP 404* in case a non-existing collection is accessed in the query.

- *405*: The server will respond with *HTTP 405* if an unsupported HTTP method is used.

Examples

Executes a query and extract the result in a single go:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/cursor
{"query":"FOR p IN products LIMIT 2 RETURN
p","count":true,"batchSize":2}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/831414965",
      "_rev" : "831414965",
      "_key" : "831414965",
      "hello1" : "world1"
    },
    {
      "_id" : "products/831742645",
      "_rev" : "831742645",
      "_key" : "831742645",
      "hello2" : "world1"
    }
  ],
  "hasMore" : false,
```

```
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

Executes a query and extracts part of the result:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/cursor
{"query":"FOR p IN products LIMIT 5 RETURN
p","count":true,"batchSize":2}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/833643189",
      "_rev" : "833643189",
      "_key" : "833643189",
      "hello5" : "world1"
    },
    {
      "_id" : "products/832660149",
      "_rev" : "832660149",
      "_key" : "832660149",
      "hello2" : "world1"
    }
  ],
  "hasMore" : true,
  "id" : "833839797",
  "count" : 5,
  "error" : false,
  "code" : 201
}
```

Using a query option:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/cursor
{"query":"FOR i IN 1..1000 FILTER i > 500 LIMIT 10 RETURN
i","count":true,"options":{"fullCount":true}}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    501,
    502,
    503,
    504,
    505,
    506,
    507,
    508,
```

```
    509,
    510
  ],
  "hasMore" : false,
  "count" : 10,
  "extra" : {
    "fullCount" : 500
  },
  "error" : false,
  "code" : 201
}
```

Bad queries:

Missing body:

```
shell> curl -X POST --dump - http://localhost:8529/_api/cursor

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 1502,
  "errorMessage" : "query is empty"
}
```

Unknown collection:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/cursor
{"query":"FOR u IN unknowncoll LIMIT 2 RETURN
u","count":true,"batchSize":2}

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 404,
  "errorNum" : 1203,
  "errorMessage" : "cannot execute query: collection not found:
'unknowncoll'"
}
```

parse a query and return information about it

Parse query

POST /_api/query

- *query*: To validate a query string without executing it, the query string can be passed to the server via an HTTP POST request.

These query string needs to be passed in the attribute *query* of a JSON object as the body of the POST

request.

Return Codes

- *200*: If the query is valid, the server will respond with *HTTP 200* and return the names of the bind parameters it found in the query (if any) in the *"bindVars"* attribute of the response.

- *400*: The server will respond with *HTTP 400* in case of a malformed request, or if the query contains a parse error. The body of the response will contain the error details embedded in a JSON object.

Examples

Valid query:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/query
{ "query" : "FOR p IN products FILTER p.name == @name LIMIT 2
RETURN p.n" }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "bindVars" : [
    "name"
  ],
  "collections" : [
    "products"
  ],
  "error" : false,
  "code" : 200
}
```

Invalid query:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/query
{ "query" : "FOR p IN products FILTER p.name = @name LIMIT 2 RETURN
p.n" }

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 1501,
  "errorMessage" : "syntax error, unexpected assignment near '=
@name LIMIT 2 RETURN p.n' at position 1:33"
}
```

return the next results from an existing cursor

Read next batch from cursor

```
PUT /_api/cursor/cursor-identifier
```

URL Parameters

- *cursor-identifier*: The name of the cursor

If the cursor is still alive, returns an object with the following attributes.

- *id*: the *cursor-identifier*
- *result*: a list of documents for the current batch
- *hasMore*: *false* if this was the last batch
- *count*: if present the total number of elements

Note that even if *hasMore* returns *true*, the next call might still return no documents. If, however, *hasMore* is *false*, then the cursor is exhausted. Once the *hasMore* attribute has a value of *false*, the client can stop.

Return Codes

- *200*: The server will respond with *HTTP 200* in case of success.

- *400*: If the cursor identifier is omitted, the server will respond with *HTTP 404*.

- *404*: If no cursor with the specified identifier can be found, the server will respond with *HTTP 404*.

Examples

Valid request for next batch:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/cursor
{"query":"FOR p IN products LIMIT 5 RETURN
p","count":true,"batchSize":2}

shell> curl -X PUT --dump -
http://localhost:8529/_api/cursor/835871413

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/835019445",
      "_rev" : "835019445",
      "_key" : "835019445",
      "hello3" : "world1"
    },
    {
      "_id" : "products/834691765",
      "_rev" : "834691765",
      "_key" : "834691765",
      "hello2" : "world1"
    }
  ],
  "hasMore" : true,
  "id" : "835871413",
  "count" : 5,
  "error" : false,
  "code" : 200
}
```

Missing identifier

```
shell> curl -X PUT --dump - http://localhost:8529/_api/cursor

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 400,
  "errorMessage" : "bad parameter"
}
```

Unknown identifier

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/cursor/123123

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 404,
  "errorNum" : 1600,
  "errorMessage" : "cursor not found"
}
```

dispose an existing cursor

Delete cursor

```
DELETE /_api/cursor/cursor-identifier
```

URL Parameters

- *cursor-identifier*: The name of the cursor

Deletes the cursor and frees the resources associated with it.

The cursor will automatically be destroyed on the server when the client has retrieved all documents from it. The client can also explicitly destroy the cursor at any earlier time using an HTTP DELETE request. The cursor id must be included as part of the URL.

Note: the server will also destroy abandoned cursors automatically after a certain server-controlled timeout to avoid resource leakage.

Return Codes

- *202*: is returned if the server is aware of the cursor.

- *404*: is returned if the server is not aware of the cursor. It is also returned if a cursor is used after it has been destroyed.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/cursor
{"query":"FOR p IN products LIMIT 5 RETURN
p","count":true,"batchSize":2}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/836330165",
      "_rev" : "836330165",
      "_key" : "836330165",
      "hello1" : "world1"
    },
    {
      "_id" : "products/836657845",
      "_rev" : "836657845",
      "_key" : "836657845",
      "hello2" : "world1"
    }
  ],
  "hasMore" : true,
  "id" : "837837493",
  "count" : 5,
  "error" : false,
  "code" : 201
}
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/cursor/837837493
undefined
```

HTTP Interface for AQL Queries

ArangoDB has an Http interface to syntactically validate AQL queries. Furthermore, it offers an Http interface to retrieve the execution plan for any valid AQL query.

Both functionalities do not actually execute the supplied AQL query, but only inspect it and return meta information about it.

explain a query and return information about it

Explain query

POST /_api/explain

- *body*: The query string needs to be passed in the attribute *query* of a JSON object as the body of the POST request. If the query references any bind variables, these must also be passed in the attribute *bindVars*.

To explain how an AQL query would be executed on the server, the query string can be sent to the server via an HTTP POST request. The server will then validate the query and create an execution plan for it, but will not execute it.

The execution plan that is returned by the server can be used to estimate the probable performance of an AQL query. Though the actual performance will depend on many different factors, the execution plan normally can give some good hint on the amount of work the server needs to do in order to actually run the query.

The top-level statements will appear in the result in the same order in which they have been used in the original query. Each result element has at most the following attributes:

- *id*: the row number of the top-level statement, starting at 1
- *type*: the type of the top-level statement (e.g. *for*, *return* ...)
- *loopLevel*: the nesting level of the top-level statement, starting at 1 Depending on the type of top-level statement, there might be other attributes providing additional information, for example, if and which indexed will be used. Many top-level statements will provide an *expression* attribute that contains data about the expression they operate on. This is true for *FOR*, *FILTER*, *SORT*, *COLLECT*, and *RETURN* statements. The *expression* attribute has the following sub-attributes:
- *type*: the type of the expression. Some possible values are:
  - *collection*: an iteration over documents from a collection. The *value* attribute will then contain the collection name. The *extra* attribute will contain information about if and which index is used when accessing the documents from the collection. If no index is used, the *accessType* sub-attribute of the *extra* attribute will have the value *all*, otherwise it will be *index*.
  - *list*: a list of dynamic values. The *value* attribute will contain the list elements.
  - *const list*: a list of constant values. The *value* attribute will contain the list elements.
  - *reference*: a reference to another variable. The *value* attribute will contain the name of the variable that is referenced.

Please note that the structure of the explain result data might change in future versions of ArangoDB without further notice and without maintaining backwards compatibility.

Return Codes

- *200*: If the query is valid, the server will respond with *HTTP 200* and return a list of the individual query execution steps in the *"plan"* attribute of the response.

- *400*: The server will respond with *HTTP 400* in case of a malformed request, or if the query contains a parse error. The body of the response will contain the error details embedded in a JSON object. Omitting bind variables if the query references any will result also result in an *HTTP*

400 error.

- *404*: The server will respond with *HTTP 404* in case a non-existing collection is accessed in the query.

Examples

Valid query:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/explain
{ "query" : "FOR p IN products FILTER p.id == @id LIMIT 2 RETURN
p.name", "bindVars": { "id" : 3 } }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "plan" : [
    {
      "id" : 1,
      "loopLevel" : 1,
      "type" : "for",
      "resultVariable" : "p",
      "limit" : true,
      "expression" : {
        "type" : "collection",
        "value" : "products",
        "extra" : {
          "accessType" : "all"
        }
      }
    },
    {
      "id" : 2,
      "loopLevel" : 1,
      "type" : "filter",
      "expression" : {
        "type" : "expression",
        "value" : "p.id == 3"
      }
    },
    {
      "id" : 3,
      "loopLevel" : 1,
      "type" : "return",
      "expression" : {
        "type" : "expression",
        "value" : "p.name"
      }
    }
  ],
  "error" : false,
  "code" : 200
}
```

Invalid query:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/explain
{ "query" : "FOR p IN products FILTER p.id == @id LIMIT 2 RETURN
p.n" }

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 1551,
  "errorMessage" : "no value specified for declared bind parameter
'id'"
}
```

The data returned in the *plan* attribute of the result contains one element per AQL top-level statement (i.e. *FOR*, *RETURN*, *FILTER* etc.). If the query optimiser removed some unnecessary statements, the result might also contain less elements than there were top-level statements in the AQL query. The following example shows a query with a non-sensible filter condition that the optimiser has removed so that there are less top-level statements:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/explain
{ "query" : "FOR i IN [ 1, 2, 3 ] FILTER 1 == 2 RETURN i" }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "plan" : [
    {
      "id" : 1,
      "loopLevel" : 0,
      "type" : "return (empty)",
      "expression" : {
        "type" : "const list",
        "value" : "[  ]"
      }
    }
  ],
  "error" : false,
  "code" : 200
}
```

parse a query and return information about it

Parse query

POST /_api/query

- *query*: To validate a query string without executing it, the query string can be passed to the server via an HTTP POST request.

These query string needs to be passed in the attribute *query* of a JSON object as the body of the POST request.

Return Codes

- *200*: If the query is valid, the server will respond with *HTTP 200* and return the names of the bind parameters it found in the query (if any) in the *"bindVars"* attribute of the response.

- *400*: The server will respond with *HTTP 400* in case of a malformed request, or if the query contains a parse error. The body of the response will contain the error details embedded in a JSON object.

Examples

Valid query:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/query
{ "query" : "FOR p IN products FILTER p.name == @name LIMIT 2
RETURN p.n" }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "bindVars" : [
    "name"
  ],
  "collections" : [
    "products"
  ],
  "error" : false,
  "code" : 200
}
```

Invalid query:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/query
{ "query" : "FOR p IN products FILTER p.name = @name LIMIT 2 RETURN
p.n" }

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 1501,
  "errorMessage" : "syntax error, unexpected assignment near '=
@name LIMIT 2 RETURN p.n' at position 1:33"
}
```

HTTP Interface for AQL User Functions Management

AQL User Functions Management

This is an introduction to ArangoDB's Http interface for managing AQL user functions. AQL user functions are a means to extend the functionality of ArangoDB's query language (AQL) with user-defined Javascript code.

For an overview of how AQL user functions work, please refer to [Extending Aql](#).

The Http interface provides an API for adding, deleting, and listing previously registered AQL user functions.

All user functions managed through this interface will be stored in the system collection _aqlfunctions. Documents in this collection should not be accessed directly, but only via the dedicated interfaces.

create a new AQL user function

Create AQL user function

```
POST /_api/aqlfunction
```

- *body*: the body with name and code of the aql user function.

The following data need to be passed in a JSON representation in the body of the POST request:

- *name*: the fully qualified name of the user functions.

- *code*: a string representation of the function body.

- *isDeterministic*: an optional boolean value to indicate that the function results are fully deterministic (function return value solely depends on the input value and return value is the same for repeated calls with same input). The *isDeterministic* attribute is currently not used but may be used later for optimisations.

In case of success, the returned JSON object has the following properties:

- *error*: boolean flag to indicate that an error occurred (*false* in this case)

- *code*: the HTTP status code

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: boolean flag to indicate that an error occurred (*true* in this case)

- *code*: the HTTP status code

- *errorNum*: the server error number

- *errorMessage*: a descriptive error message

Return Codes

- *200*: If the function already existed and was replaced by the call, the server will respond with *HTTP 200*.

- *201*: If the function can be registered by the server, the server will respond with *HTTP 201*.

- *400*: If the JSON representation is malformed or mandatory data is missing from the request, the server will respond with *HTTP 400*.

Examples

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/aqlfunction
{ "name" : "myfunctions::temperature::celsiustofahrenheit", "code"
: "function (celsius) { return celsius * 1.8 + 32; }" }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "error" : false,
  "code" : 201
}
```

remove an existing AQL user function

Remove existing AQL user function

```
DELETE /_api/aqlfunction/name
```

URL Parameters

- *name*: the name of the AQL user function.

Query Parameters

- *group*: If set to *true*, then the function name provided in *name* is treated as a namespace prefix, and all functions in the specified namespace will be deleted. If set to *false*, the function name provided in *name* must be fully qualified, including any namespaces.

Removes an existing AQL user function, identified by *name*.

In case of success, the returned JSON object has the following properties:

- *error*: boolean flag to indicate that an error occurred (*false* in this case)

- *code*: the HTTP status code

The body of the response will contain a JSON object with additional error details. The object has the following attributes:

- *error*: boolean flag to indicate that an error occurred (*true* in this case)

- *code*: the HTTP status code

- *errorNum*: the server error number

- *errorMessage*: a descriptive error message

Return Codes

- *200*: If the function can be removed by the server, the server will respond with *HTTP 200*.

- *400*: If the user function name is malformed, the server will respond with *HTTP 400*.

- *404*: If the specified user user function does not exist, the server will respond with *HTTP 404*.

Examples

deletes a function:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/aqlfunction/square::x::y
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "error" : false,
  "code" : 200
}
```

function not found:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/aqlfunction/myfunction::x::y
undefined

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 404,
  "errorNum" : 1582,
  "errorMessage" : "user function '%s()' not found"
}
```

gets all reqistered AQL user functions

Return registered AQL user functions

GET /_api/aqlfunction

Query Parameters

- *namespace*: Returns all registered AQL user functions from namespace *namespace*.

Returns all registered AQL user functions.

The call will return a JSON list with all user functions found. Each user function will at least have the following attributes:

- *name*: The fully qualified name of the user function

- *code*: A string representation of the function body

Return Codes

- *200*: if success *HTTP 200* is returned.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/aqlfunction
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

[ ]
```

HTTP Interface for Simple Queries

Simple Queries

This is an introduction to ArangoDB's Http interface for simple queries.

Simple queries can be used if the query condition is straight forward simple, i.e., a document reference, all documents, a query-by-example, or a simple geo query. In a simple query you can specify exactly one collection and one condition. The result can then be sorted and can be split into pages.

Working with Simples Queries using HTTP

To limit the amount of results to be transferred in one batch, simple queries support a *batchSize* parameter that can optionally be used to tell the server to limit the number of results to be transferred in one batch to a certain value. If the query has more results than were transferred in one go, more results are waiting on the server so they can be fetched subsequently. If no value for the *batchSize* parameter is specified, the server will use a reasonable default value.

If the server has more documents than should be returned in a single batch, the server will set the *hasMore* attribute in the result. It will also return the id of the server-side cursor in the *id* attribute in the result. This id can be used with the cursor API to fetch any outstanding results from the server and dispose the server-side cursor afterwards.

returns all documents of a collection

Return all

PUT /_api/simple/all

- *query*: Contains the query.

Returns all documents of a collections. The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *skip*: The number of documents to skip in the query (optional).

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Limit the amount of documents using *limit*

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/all
{ "collection": "products", "skip": 2, "limit" : 2 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/979591861",
      "_rev" : "979591861",
```

```
      "_key" : "979591861",
      "Hello3" : "World3"
    },
    {
      "_id" : "products/980247221",
      "_rev" : "980247221",
      "_key" : "980247221",
      "Hello5" : "World5"
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

Using a *batchSize* value

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/all
{ "collection": "products", "batchSize" : 3 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/981557941",
      "_rev" : "981557941",
      "_key" : "981557941",
      "Hello3" : "World3"
    },
    {
      "_id" : "products/980902581",
      "_rev" : "980902581",
      "_key" : "980902581",
      "Hello1" : "World1"
    },
    {
      "_id" : "products/982213301",
      "_rev" : "982213301",
      "_key" : "982213301",
      "Hello5" : "World5"
    }
  ],
  "hasMore" : true,
  "id" : "982409909",
  "count" : 5,
  "error" : false,
  "code" : 201
}
```

returns all documents of a collection matching a given example

Simple query by-example

```
PUT /_api/simple/by-example
```

- *query*: Contains the query.

This will find all documents matching a given example.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *example*: The example document.

- *skip*: The number of documents to skip in the query (optional).

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Matching an attribute:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/by-example
{ "collection": "products", "example" :   { "i" : 1 }   }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/998728373",
      "_rev" : "998728373",
      "_key" : "998728373",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    },
    {
      "_id" : "products/998400693",
      "_rev" : "998400693",
      "_key" : "998400693",
      "i" : 1,
      "a" : {
        "k" : 1,
        "j" : 1
      }
    },
    {
```

```
      "_id" : "products/999187125",
      "_rev" : "999187125",
      "_key" : "999187125",
      "i" : 1,
      "a" : {
        "k" : 2,
        "j" : 2
      }
    },
    {
      "_id" : "products/998990517",
      "_rev" : "998990517",
      "_key" : "998990517",
      "i" : 1
    }
  ],
  "hasMore" : false,
  "count" : 4,
  "error" : false,
  "code" : 201
}
```

Matching an attribute which is a sub-document:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/by-example
{ "collection": "products", "example" : { "a.j" : 1 } }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/1000759989",
      "_rev" : "1000759989",
      "_key" : "1000759989",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    },
    {
      "_id" : "products/1000432309",
      "_rev" : "1000432309",
      "_key" : "1000432309",
      "i" : 1,
      "a" : {
        "k" : 1,
        "j" : 1
      }
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

Matching an attribute within a sub-document:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } } }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/1002791605",
      "_rev" : "1002791605",
      "_key" : "1002791605",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    }
  ],
  "hasMore" : false,
  "count" : 1,
  "error" : false,
  "code" : 201
}
```

returns one document of a collection matching a given example

Document matching an example

PUT /_api/simple/first-example

- *query*: Contains the query.

This will return the first document matching a given example.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *example*: The example document.

Returns a result containing the document or *HTTP 404* if no document matched the example.

If more than one document in the collection matches the specified example, only one of these documents will be returned, and it is undefined which of the matching documents is returned.

Return Codes

- *200*: is returned when the query was successfully executed.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

If a matching document was found:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/first-example
{ "collection": "products", "example" :   { "i" : 1 }   }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
   "document" : {
     "_id" : "products/1004495541",
     "_rev" : "1004495541",
     "_key" : "1004495541",
     "i" : 1,
     "a" : {
        "k" : 1,
        "j" : 1
     }
   },
   "error" : false,
   "code" : 200
}
```

If no document was found:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/first-example
{ "collection": "products", "example" :   { "l" : 1 }   }

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
   "error" : true,
   "code" : 404,
   "errorNum" : 404,
   "errorMessage" : "no match"
}
```

returns all documents of a collection matching a given example, using a specific hash index

Hash index

PUT /_api/simple/by-example-hash

- *query*: Contains the query specification.

This will find all documents matching a given example, using the specified hash index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *index*: The id of the index to be used for the query. The index must exist and must be of type *hash*.

- *example*: an example document. The example must contain a value for each attribute in the index.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal number of documents to return. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case. The same error code is also returned if an invalid index id or type is used. returns all documents of a collection matching a given example, using a specific skiplist index

Skiplist index

```
PUT /_api/simple/by-example-skiplist
```

- *query*: Contains the query specification.

This will find all documents matching a given example, using the specified skiplist index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *index*: The id of the index to be used for the query. The index must exist and must be of type *skiplist*.

- *example*: an example document. The example must contain a value for each attribute in the index.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal number of documents to return. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case. The same error code is also returned if an invalid index id or type is used. returns all documents of a collection matching a given example, using a specific bitarray index

Bitarray index

```
PUT /_api/simple/by-example-bitarray
```

- *query*: Contains the query specification.

This will find all documents matching a given example, using the specified skiplist index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *index*: The id of the index to be used for the query. The index must exist and must be of type *bitarray*.

- *example*: an example document. The example must contain a value for each attribute in the index.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal number of documents to return. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case. The same error code is also returned if an invalid index id or type is used. returns all documents of a collection matching a given condition, using a specific skiplist index

Query by-condition using Skiplist index

```
PUT /_api/simple/by-condition-skiplist
```

- *query*: Contains the query specification.

This will find all documents matching a given condition, using the specified skiplist index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *index*: The id of the index to be used for the query. The index must exist and must be of type *skiplist*.

- *condition*: the condition which all returned documents shall satisfy. Conditions must be specified for all indexed attributes.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal number of documents to return. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case. The same error code is also returned if an invalid index id or type is used. returns all documents of a collection matching a given condition, using a specific bitarray index

Query by-condition using bitarray index

`PUT /_api/simple/by-condition-bitarray`

- *query*: Contains the query specification.

This will find all documents matching a given condition, using the specified skiplist index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *index*: The id of the index to be used for the query. The index must exist and must be of type *bitarray*.

- *condition*: the condition which all returned documents shall satisfy. Conditions must be specified for all indexed attributes.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal number of documents to return. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case. The same error code is also returned if an invalid index id or type is used.

returns a random document from a collection

Random document

`PUT /_api/simple/any`

- *query*: Contains the query.

Returns a random document from a collection. The call expects a JSON object as body with the following attributes:

- *collection*: The identifier or name of the collection to query.

Returns a JSON object with the document stored in the attribute *document* if the collection contains at least one document. If the collection is empty, the *document* attrbute contains null.

Return Codes

- *200*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The

response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/any
{ "collection": "products" }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
   "document" : {
     "_id" : "products/983851701",
     "_rev" : "983851701",
     "_key" : "983851701",
     "Hello4" : "World4"
   },
   "error" : false,
   "code" : 200
}
```

returns all documents of a collection within a range

Simple range query

PUT /_api/simple/range

- *query*: Contains the query.

This will find all documents within a given range. In order to execute a range query, a skip-list index on the queried attribute must be present.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *attribute*: The attribute path to check.

- *left*: The lower bound.

- *right*: The upper bound.

- *closed*: If *true*, use interval including *left* and *right*, otherwise exclude *right*, but include *left*.

- *skip*: The number of documents to skip in the query (optional).

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The

response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/range
{ "collection": "products", "attribute" : "i", "left" : 2, "right"
: 4 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/1015112373",
      "_rev" : "1015112373",
      "_key" : "1015112373",
      "i" : 2
    },
    {
      "_id" : "products/1015308981",
      "_rev" : "1015308981",
      "_key" : "1015308981",
      "i" : 3
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

returns all documents of a collection near a given location

Near query

`PUT /_api/simple/near`

- *query*: Contains the query.

The default will find at most 100 documents near the given coordinate. The returned list is sorted according to the distance, with the nearest document being first in the list. If there are near documents of equal distance, documents are chosen randomly from this set until the limit is reached.

In order to use the *near* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* field to select a particular index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *latitude*: The latitude of the coordinate.

- *longitude*: The longitude of the coordinate.

- *distance*: If given, the attribute key used to return the distance to the given coordinate. (optional). If specified, distances are returned in meters.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. The default is 100. (optional)

- *geo*: If given, the identifier of the geo-index to use. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Without distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 2 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/986276533",
      "_rev" : "986276533",
      "_key" : "986276533",
      "name" : "Name/0.002/",
      "loc" : [
        0.002,
        0
      ]
    },
    {
      "_id" : "products/985883317",
      "_rev" : "985883317",
      "_key" : "985883317",
      "name" : "Name/-0.002/",
      "loc" : [
        -0.002,
        0
      ]
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
```

```
    "code" : 201
}
```

With distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 3, "distance" : "distance" }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/988897973",
      "_rev" : "988897973",
      "_key" : "988897973",
      "name" : "Name/-0.002/",
      "loc" : [
        -0.002,
        0
      ],
      "distance" : 222.38985328911744
    },
    {
      "_id" : "products/989291189",
      "_rev" : "989291189",
      "_key" : "989291189",
      "name" : "Name/0.002/",
      "loc" : [
        0.002,
        0
      ],
      "distance" : 222.38985328911744
    },
    {
      "_id" : "products/988701365",
      "_rev" : "988701365",
      "_key" : "988701365",
      "name" : "Name/-0.004/",
      "loc" : [
        -0.004,
        0
      ],
      "distance" : 444.779706578235
    }
  ],
  "hasMore" : false,
  "count" : 3,
  "error" : false,
  "code" : 201
}
```

returns all documents of a collection within a given radius

Within query

```
PUT /_api/simple/within
```

- *query*: Contains the query.

This will find all documents within a given radius around the coordinate (*latitude*, *longitude*). The returned list is sorted by distance.

In order to use the *within* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* field to select a particular index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *latitude*: The latitude of the coordinate.

- *longitude*: The longitude of the coordinate.

- *radius*: The maximal radius (in meters).

- *distance*: If given, the attribute key used to return the distance to the given coordinate. (optional). If specified, distances are returned in meters.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. The default is 100. (optional)

- *geo*: If given, the identifier of the geo-index to use. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Without distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 2, "radius" : 500 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/992305845",
      "_rev" : "992305845",
      "_key" : "992305845",
```

```
        "name" : "Name/0.002/",
        "loc" : [
          0.002,
          0
        ]
      },
      {
        "_id" : "products/991912629",
        "_rev" : "991912629",
        "_key" : "991912629",
        "name" : "Name/-0.002/",
        "loc" : [
          -0.002,
          0
        ]
      }
    ],
    "hasMore" : false,
    "count" : 2,
    "error" : false,
    "code" : 201
}
```

With distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 3, "distance" : "distance", "radius" : 300 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
    "result" : [
      {
        "_id" : "products/994927285",
        "_rev" : "994927285",
        "_key" : "994927285",
        "name" : "Name/-0.002/",
        "loc" : [
          -0.002,
          0
        ],
        "distance" : 222.38985328911744
      },
      {
        "_id" : "products/995320501",
        "_rev" : "995320501",
        "_key" : "995320501",
        "name" : "Name/0.002/",
        "loc" : [
          0.002,
          0
        ],
        "distance" : 222.38985328911744
      },
      {
```

```
        "_id" : "products/994730677",
        "_rev" : "994730677",
        "_key" : "994730677",
        "name" : "Name/-0.004/",
        "loc" : [
          -0.004,
          0
        ],
        "distance" : 444.779706578235
      }
   ],
   "hasMore" : false,
   "count" : 3,
   "error" : false,
   "code" : 201
}
```

returns documents of a collection as a result of a fulltext query

Fulltext index query

```
PUT /_api/simple/fulltext
```

- *query*: Contains the query.

This will find all documents from the collection that match the fulltext query specified in *query*.

In order to use the *fulltext* operator, a fulltext index must be defined for the collection and the specified attribute.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *attribute*: The attribute that contains the texts.

- *query*: The fulltext query.

- *skip*: The number of documents to skip in the query (optional).

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. (optional)

- *index*: The identifier of the fulltext-index to use.

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
```

```
http://localhost:8529/_api/simple/fulltext
{ "collection": "products", "attribute" : "text", "query" : "word"
}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/997024437",
      "_rev" : "997024437",
      "_key" : "997024437",
      "text" : "this text also has a word"
    },
    {
      "_id" : "products/996827829",
      "_rev" : "996827829",
      "_key" : "996827829",
      "text" : "this text contains word"
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

removes all documents of a collection that match an example

Remove documents by example

PUT /_api/simple/remove-by-example

- *query*: Contains the query.

This will find all documents in the collection that match the specified example object.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to remove from.

- *example*: An example document that all collection documents are compared against.

- options: an json object which can contains following attributes:

- *waitForSync*: if set to true, then all removal operations will instantly be synchronised to disk. If this is not specified, then the collection's default sync behavior will be applied.

- *limit*: an optional value that determines how many documents to delete at most. If *limit* is specified but is less than the number of documents in the collection, it is undefined which of the documents will be deleted.

Note: the *limit* attribute is not supported on sharded collections. Using it will result in an error. The options attributes waitForSync and limit can given yet without an ecapsulation into a json object. but this may be deprecated in future versions of arango

Returns the number of documents that were deleted.

Return Codes

- *200*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/remove-by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } } }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
   "deleted" : 1,
   "error" : false,
   "code" : 200
}
```

Using Parameter: waitForSync and limit

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/remove-by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 }
},"waitForSync": true, "limit": 2 }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
   "deleted" : 1,
   "error" : false,
   "code" : 200
}
```

Using Parameter: waitForSync and limit with new signature

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/remove-by-example
{"collection": "products","example" : { "a" : { "j" : 1 }
},"options": {"waitForSync": true, "limit": 2} }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
   "deleted" : 1,
   "error" : false,
   "code" : 200
}
```

replaces the body of all documents of a collection that match an example

Replace documents by example

```
PUT /_api/simple/replace-by-example
```

- *query*: Contains the query.

This will find all documents in the collection that match the specified example object, and replace the entire document body with the new value specified. Note that document meta-attributes such as *_id*, *_key*, *_from*, *_to* etc. cannot be replaced.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to replace within.

- *example*: An example document that all collection documents are compared against.

- *newValue*: The replacement document that will get inserted in place of the "old" documents.

- *options*: an json object which can contain following attributes

- *waitForSync*: if set to true, then all removal operations will instantly be synchronised to disk. If this is not specified, then the collection's default sync behavior will be applied.

- *limit*: an optional value that determines how many documents to replace at most. If *limit* is specified but is less than the number of documents in the collection, it is undefined which of the documents will be replaced.

Note: the *limit* attribute is not supported on sharded collections. Using it will result in an error. The options attributes waitForSync and limit can given yet without an ecapsulation into a json object. but this may be deprecated in future versions of arango

Returns the number of documents that were replaced.

Return Codes

- *200*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/replace-by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } },
"newValue" : {"foo" : "bar"}, "limit" : 3 }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "replaced" : 1,
  "error" : false,
  "code" : 200
}
```

Using new Signature for attributes WaitForSync and limit

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/replace-by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } },
"newValue" : {"foo" : "bar"}, "options": {"limit" : 3,
"waitForSync": true  }}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "replaced" : 1,
  "error" : false,
  "code" : 200
}
```

partially updates the body of all documents of a collection that match an example

Update documents by example

PUT /_api/simple/update-by-example

- *query*: Contains the query.

This will find all documents in the collection that match the specified example object, and partially update the document body with the new value specified. Note that document meta-attributes such as *_id*, *_key*, *_from*, *_to* etc. cannot be replaced.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to update within.

- *example*: An example document that all collection documents are compared against.

- *newValue*: A document containing all the attributes to update in the found documents.

- *options*: a json object wich can contains following attributes:

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

- *waitForSync*: if set to true, then all removal operations will instantly be synchronised to disk. If this is not specified, then the collection's default sync behavior will be applied.

- *limit*: an optional value that determines how many documents to update at most. If *limit* is specified but is less than the number of documents in the collection, it is undefined which of the documents will be updated.

Note: the *limit* attribute is not supported on sharded collections. Using it will result in an error.

Returns the number of documents that were updated.

Return Codes

- *200*: is returned if the collection was updated successfully and *waitForSync* was *true*.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples using old syntax for options

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/update-by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } },
"newValue" : { "a" : { "j" : 22 } }, "limit" : 3 }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "updated" : 1,
  "error" : false,
  "code" : 200
}
```

using new signature for options

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/update-by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } },
"newValue" : { "a" : { "j" : 22 } }, "options" :  { "limit" : 3,
"waitForSync": true }  }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "updated" : 1,
  "error" : false,
  "code" : 200
}
```

returns the first document(s) of a collection

First document of a collection

```
PUT /_api/simple/first
```

- *query*: Contains the query.

This will return the first document(s) from the collection, in the order of insertion/update time. When the *count* argument is supplied, the result will be a list of documents, with the "oldest" document being first in the result list. If the *count* argument is not supplied, the result is the "oldest" document of the collection, or *null* if the collection is empty.

The request body must be a JSON object with the following attributes:

- *collection*: the name of the collection

- *count*: the number of documents to return at most. Specifiying count is optional. If it is not specified, it defaults to 1.

Note: this method is not supported for sharded collections with more than one shard.

Return Codes

- *200*: is returned when the query was successfully executed.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Retrieving the first n documents:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/first
{ "collection": "products", "count" : 2 }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/1008427701",
      "_rev" : "1008427701",
      "_key" : "1008427701",
      "i" : 1,
      "a" : {
        "k" : 1,
        "j" : 1
      }
    },
    {
      "_id" : "products/1008755381",
      "_rev" : "1008755381",
      "_key" : "1008755381",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    }
  ],
  "error" : false,
  "code" : 200
}
```

Retrieving the first document:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/first
{ "collection": "products" }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
```

```
      "_id" : "products/1010066101",
      "_rev" : "1010066101",
      "_key" : "1010066101",
      "i" : 1,
      "a" : {
        "k" : 1,
        "j" : 1
      }
    },
    "error" : false,
    "code" : 200
}
```

returns the last document(s) of a collection

Last document of a collection

```
PUT /_api/simple/last
```

- *query*: Contains the query.

This will return the last documents from the collection, in the order of insertion/update time. When the *count* argument is supplied, the result will be a list of documents, with the "latest" document being first in the result list.

The request body must be a JSON object with the following attributes:

- *collection*: the name of the collection

- *count*: the number of documents to return at most. Specifiying count is optional. If it is not specified, it defaults to 1.

If the *count* argument is not supplied, the result is the "latest" document of the collection, or *null* if the collection is empty.

Note: this method is not supported for sharded collections with more than one shard.

Return Codes

- *200*: is returned when the query was successfully executed.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Retrieving the last n documents:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/last
{ "collection": "products", "count" : 2 }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
```

```
  "result" : [
    {
      "_id" : "products/1012490933",
      "_rev" : "1012490933",
      "_key" : "1012490933",
      "i" : 1,
      "a" : {
        "k" : 2,
        "j" : 2
      }
    },
    {
      "_id" : "products/1012294325",
      "_rev" : "1012294325",
      "_key" : "1012294325",
      "i" : 1
    }
  ],
  "error" : false,
  "code" : 200
}
```

Retrieving the first document:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/last
{ "collection": "products" }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : {
    "_id" : "products/1014129333",
    "_rev" : "1014129333",
    "_key" : "1014129333",
    "i" : 1,
    "a" : {
      "k" : 2,
      "j" : 2
    }
  },
  "error" : false,
  "code" : 200
}
```

HTTP Interface for Collections

Collections

This is an introduction to ArangoDB's Http interface for collections.

Collection

A collection consists of documents. It is uniquely identified by its collection identifier. It also has a unique name that clients should use to identify and access it. Collections can be renamed. This will change the collection name, but not the collection identifier. Collections have a type that is specified by the user when the collection is created. There are currently two types: document and edge. The default type is document.

Collection Identifier

A collection identifier lets you refer to a collection in a database. It is a string value and is unique within the database. Up to including ArangoDB 1.1, the collection identifier has been a client's primary means to access collections. Starting with ArangoDB 1.2, clients should instead use a collection's unique name to access a collection instead of its identifier. ArangoDB currently uses 64bit unsigned integer values to maintain collection ids internally. When returning collection ids to clients, ArangoDB will put them into a string to ensure the collection id is not clipped by clients that do not support big integers. Clients should treat the collection ids returned by ArangoDB as opaque strings when they store or use it locally.

Note: collection ids have been returned as integers up to including ArangoDB 1.1

Collection Name

A collection name identifies a collection in a database. It is a string and is unique within the database. Unlike the collection identifier it is supplied by the creator of the collection. The collection name must consist of letters, digits, and the _ (underscore) and - (dash) characters only. Please refer to Naming Conventions in ArangoDB for more information on valid collection names.

Key Generator

ArangoDB allows using key generators for each collection. Key generators have the purpose of auto-generating values for the _key attribute of a document if none was specified by the user. By default, ArangoDB will use the traditional key generator. The traditional key generator will auto-generate key values that are strings with ever-increasing numbers. The increment values it uses are non-deterministic.

Contrary, the auto increment key generator will auto-generate deterministic key values. Both the start value and the increment value can be defined when the collection is created. The default start value is 0 and the default increment is 1, meaning the key values it will create by default are:

1, 2, 3, 4, 5, ...

When creating a collection with the auto increment key generator and an increment of 5, the generated keys would be:

1, 6, 11, 16, 21, ...

The basic operations (create, read, update, delete) for documents are mapped to the standard HTTP methods (*POST*, *GET*, *PUT*, *DELETE*).

Address of a Collection

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

```
http://server:port/_api/collection/collection-name
```

For example: Assume that the collection identifier is *7254820* and the collection name is *demo*, then the URL of that collection is:

```
http://localhost:8529/_api/collection/demo
```

Creating and Deleting Collections

creates a collection

Create collection

```
POST /_api/collection
```

- *body*: the body with name and options for a collection.

Creates an new collection with a given name. The request must contain an object with the following attributes.

- *name*: The name of the collection.

- *waitForSync* (optional, default: false): If *true* then the data is synchronised to disk before returning from a create or update of a document.

- *doCompact* (optional, default is *true*): whether or not the collection will be compacted.

- *journalSize* (optional, default is a @ref CommandLineArangod "configuration parameter"): The maximal size of a journal or datafile.
  Note: This also limits the maximal size of a single object. Must be at least 1MB.

- *isSystem* (optional, default is *false*): If *true*, create a system collection. In this case *collection-name* should start with an underscore. End users should normally create non-system collections only. API implementors may be required to create system collections in very special occasions, but normally a regular collection will do.

- *isVolatile* (optional, default is *false*): If *true* then the collection data is kept in-memory only and not made persistent. Unloading the collection will cause the collection data to be discarded. Stopping or re-starting the server will also cause full loss of data in the collection. Setting this option will make the resulting collection be slightly faster than regular collections because ArangoDB does not enforce any synchronisation to disk and does not calculate any CRC checksums for datafiles (as there are no datafiles).

This option should threrefore be used for cache-type collections only, and not for data that cannot be re-created otherwise.

- *keyOptions* (optional) additional options for key generation. If specified, then *keyOptions* should be a JSON array containing the following attributes (note: some of them are optional):
  - *type*: specifies the type of the key generator. The currently available generators are *traditional* and *autoincrement*.
  - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the *_key* attribute of a document. If set to *false*, then the key generator will solely be responsible for generating keys and supplying own key values in the *_key* attribute of documents is considered an error.
  - *increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
  - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.

- *type* (optional, default is *2*): the type of the collection to create. The following values for *type* are valid:
  - *2*: document collection
  - *3*: edges collection

- *numberOfShards* (optional, default is *1*): in a cluster, this value determines the number of shards to create for the collection. In a single server setup, this option is meaningless.

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

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/collection
{"name":"testCollectionBasics"}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/testCollectionBasics

{
  "id" : "805593781",
  "name" : "testCollectionBasics",
  "waitForSync" : false,
  "isVolatile" : false,
  "isSystem" : false,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/collection
{"name":"testCollectionEdges","type":3}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/testCollectionEdges

{
  "id" : "805724853",
  "name" : "testCollectionEdges",
  "waitForSync" : false,
  "isVolatile" : false,
  "isSystem" : false,
  "status" : 3,
  "type" : 3,
  "error" : false,
  "code" : 200
}
```

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/collection
{"name":"testCollectionUsers","keyOptions":
{"type":"autoincrement","increment":5,"allowUserKeys":true}}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/testCollectionUsers

{
  "id" : "805986997",
  "name" : "testCollectionUsers",
```

```
    "waitForSync" : false,
    "isVolatile" : false,
    "isSystem" : false,
    "status" : 3,
    "type" : 2,
    "error" : false,
    "code" : 200
}
```

deletes a collection

Delete collection

```
DELETE /_api/collection/{collection-name}
```

URL Parameters

- *collection-name*: The name of the collection to delete.

Deletes a collection identified by *collection-name*.

If the collection was successfully deleted then, an object is returned with the following attributes:

- *error*: *false*

- *id*: The identifier of the deleted collection.

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Using an identifier:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/collection/830694069
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "830694069",
  "error" : false,
  "code" : 200
}
```

Using a name:

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/collection/products1
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
```

```
{
  "id" : "830890677",
  "error" : false,
  "code" : 200
}
```

truncates a collection

Truncate collection

```
PUT /_api/collection/{collection-name}/truncate
```

URL Parameters

- *collection-name*: The name of the collection.

Removes all documents from the collection, but leaves the indexes intact.

Examples

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/collection/828859061/truncate

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "828859061",
  "name" : "products",
  "isSystem" : false,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

Getting Information about a Collection

returns a collection

Return information about a collection

`GET /_api/collection/{collection-name}`

URL Parameters

- *collection-name*: The name of the collection.

The result is an object describing the collection with the following attributes:

- *id*: The identifier of the collection.

- *name*: The name of the collection.

- *status*: The status of the collection as number.
  - 1: new born collection
  - 2: unloaded
  - 3: loaded
  - 4: in the process of being unloaded
  - 5: deleted

Every other status indicates a corrupted collection.

- *type*: The type of the collection as number.
  - 2: document collection (normal case)
  - 3: edges collection

Return Codes

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Read properties of a collection

`GET /_api/collection/{collection-name}/properties`

URL Parameters

- *collection-name*: The name of the collection.

In addition to the above, the result will always contain the *waitForSync*, *doCompact*, *journalSize*, and *isVolatile* attributes. This is achieved by forcing a load of the underlying collection.

- *waitForSync*: If *true* then creating or changing a document will wait until the data has been synchronised to disk.

- *doCompact*: Whether or not the collection will be compacted.

- *journalSize*: The maximal size setting for journals / datafiles.

- *isVolatile*: If *true* then the collection data will be kept in memory only and ArangoDB will not write or sync the data to disk.

In a cluster setup, the result will also contain the following attributes:

- *numberOfShards*: the number of shards of the collection.

- *shardKeys*: contains the names of document attributes that are used to determine the target shard for documents. **Return Codes**

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Using an identifier:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/806183605/properties
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/products/properties

{
  "id" : "806183605",
  "name" : "products",
  "isSystem" : false,
  "doCompact" : true,
  "isVolatile" : false,
  "journalSize" : 1048576,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  },
  "waitForSync" : true,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

Using a name:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/products/properties
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/products/properties

{
  "id" : "806380213",
  "name" : "products",
  "isSystem" : false,
  "doCompact" : true,
  "isVolatile" : false,
  "journalSize" : 1048576,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  },
```

```
    "waitForSync" : true,
    "status" : 3,
    "type" : 2,
    "error" : false,
    "code" : 200
}
```

Return number of documents in a collection

```
GET /_api/collection/{collection-name}/count
```

URL Parameters

- *collection-name*: The name of the collection.

In addition to the above, the result also contains the number of documents. **Note** that this will always load the collection into memory.

- *count*: The number of documents inside the collection.

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Using an identifier and requesting the number of documents:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/806576821/count
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/products/count

{
  "id" : "806576821",
  "name" : "products",
  "isSystem" : false,
  "doCompact" : true,
  "isVolatile" : false,
  "journalSize" : 1048576,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  },
  "waitForSync" : true,
  "count" : 100,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

Return stats for a collection

```
GET /_api/collection/{collection-name}/figures
```

URL Parameters

- *collection-name*: The name of the collection.

In addition to the above, the result also contains the number of documents and additional statistical information about the collection.
Note : This will always load the collection into memory.

- *count*: The number of documents currently present in the collection.

- *figures.alive.count*: The number of currently active documents.

- *figures.alive.size*: The total size in bytes used by all active documents.

- *figures.dead.count*: The number of dead documents. This includes document versions that have been deleted or replaced by a newer version.

- *figures.dead.size*: The total size in bytes used by all dead documents.

- *figures.dead.deletion*: The total number of deletion markers in the collection.

- *figures.datafiles.count*: The number of current datafiles.
- *figures.datafiles.fileSize*: The total filesize of all datafiles.

- *figures.journals.count*: The number of journal files.
- *figures.journals.fileSize*: The total filesize of all journal files.

- *figures.compactors.count*: The number of compactor files.
- *figures.compactors.fileSize*: The total filesize of all compactor files.

- *figures.shapefiles.count*: The number of shape files.
- *figures.shapefiles.fileSize*: The total filesize of all shape files.

- *figures.shapes.count*: The total number of shapes in the collection (this includes shapes that are not in use anymore)

- *figures.attributes.count*: The total number of attributes in the collection (this includes attributes that are not in use anymore)

- *figures.attributes.size*: The total size used by all attribute names in the collection (this includes attributes that are not in use anymore)

- *figures.indexes.count*: The total number of indexes defined for the collection (this includes internal indexes such as the primary index)

- *figures.indexes.size*: The total memory allocated by the indexes in bytes

- *journalSize*: The maximal size of the journal in bytes.

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

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

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Using an identifier and requesting the figures of the collection:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/826630837/figures
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
location: /_db/_system/_api/collection/products/figures

{
  "id" : "826630837",
  "name" : "products",
  "isSystem" : false,
  "doCompact" : true,
  "isVolatile" : false,
  "journalSize" : 1048576,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  },
  "waitForSync" : true,
  "count" : 0,
  "figures" : {
    "alive" : {
      "count" : 0,
      "size" : 0
    },
    "dead" : {
      "count" : 0,
      "size" : 0,
      "deletion" : 0
    },
    "datafiles" : {
      "count" : 0,
      "fileSize" : 0
    },
    "journals" : {
      "count" : 0,
      "fileSize" : 0
    },
    "compactors" : {
      "count" : 0,
      "fileSize" : 0
    },
    "shapefiles" : {
      "count" : 0,
      "fileSize" : 0
    },
    "shapes" : {
      "count" : 0,
```

```
      "size" : 0
    },
    "attributes" : {
      "count" : 0,
      "size" : 0
    },
    "indexes" : {
      "count" : 1,
      "size" : 2008
    },
    "lastTick" : "0",
    "uncollectedLogfileEntries" : 0
  },
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

Return collection revision id

```
GET /_api/collection/{collection-name}/revision
```

URL Parameters

- *collection-name*: The name of the collection.

In addition to the above, the result will also contain the collection's revision id. The revision id is a server-generated string that clients can use to check whether data in a collection has changed since the last revision check.

- *revision*: The collection revision id as a string.

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Retrieving the revision of a collection

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/826958517/revision
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "826958517",
  "name" : "products",
  "isSystem" : false,
  "status" : 3,
  "type" : 2,
  "revision" : "0",
  "error" : false,
  "code" : 200
```

```
    }
```

Return checksum for the collection

```
GET /_api/collection/{collection-name}/checksum
```

URL Parameters

- *collection-name*: The name of the collection.

Query Parameters

- *withRevisions*: Whether or not to include document revision ids in the checksum calculation.

- *withData*: Whether or not to include document body data in the checksum calculation.

Will calculate a checksum of the meta-data (keys and optionally revision ids) and optionally the document data in the collection.

The checksum can be used to compare if two collections on different ArangoDB instances contain the same contents. The current revision of the collection is returned too so one can make sure the checksums are calculated for the same state of data.

By default, the checksum will only be calculated on the _key system attribute of the documents contained in the collection. For edge collections, the system attributes _from and _to will also be included in the calculation.

By setting the optional URL parameter *withRevisions* to *true*, then revision ids (_rev system attributes) are included in the checksumming.

By providing the optional URL parameter *withData* with a value of *true*, the user-defined document attributes will be included in the calculation too. **Note**: Including user-defined attributes will make the checksumming slower.

The response is a JSON object with the following attributes:

- *checksum*: The calculated checksum as a number.

- *revision*: The collection revision id as a string.

Note: this method is not available in a cluster.

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Retrieving the checksum of a collection:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/827220661/checksum
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
```

```
    "id" : "827220661",
    "name" : "products",
    "isSystem" : false,
    "status" : 3,
    "type" : 2,
    "checksum" : 547113386,
    "revision" : "827548341",
    "error" : false,
    "code" : 200
}
```

Retrieving the checksum of a collection including the collection data, but not the revisions:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection/827810485/checksum?
withRevisions=false&withData=true
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
    "id" : "827810485",
    "name" : "products",
    "isSystem" : false,
    "status" : 3,
    "type" : 2,
    "checksum" : 2127139178,
    "revision" : "828138165",
    "error" : false,
    "code" : 200
}
```

returns all collections

reads all collections

GET /_api/collection

Query Parameters

- *excludeSystem*: Whether or not system collections should be excluded from the result.

Returns an object with an attribute *collections* containing a list of all collection descriptions. The same information is also available in the *names* as hash map with the collection names as keys.

By providing the optional URL parameter *excludeSystem* with a value of *true*, all system collections will be excluded from the response.

Examples

Return information about all collections:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/collection
undefined
```

```
HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "collections" : [
    {
      "id" : "4022965",
      "name" : "_aqlfunctions",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "1794741",
      "name" : "_modules",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "711549621",
      "name" : "Electronics",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "3564213",
      "name" : "_aal",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "798253749",
      "name" : "germanHighway",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    {
      "id" : "798646965",
      "name" : "internationalHighway",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    {
      "id" : "771515061",
      "name" : "male",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "711025333",
      "name" : "Customer",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
```

```
    },
    {
      "id" : "1663669",
      "name" : "_graphs",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "711418549",
      "name" : "Groceries",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "798122677",
      "name" : "germanCity",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "2056885",
      "name" : "_cluster_kickstarter_plans",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "798515893",
      "name" : "frenchHighway",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    {
      "id" : "771383989",
      "name" : "female",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "284385973",
      "name" : "_statisticsRaw",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "711287477",
      "name" : "Company",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "803758773",
      "name" : "animals",
```

```
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "284779189",
      "name" : "_statistics",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "156341",
      "name" : "_users",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "4154037",
      "name" : "_configuration",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "1925813",
      "name" : "_routing",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "711680693",
      "name" : "has_bought",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    {
      "id" : "802841269",
      "name" : "demo",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "798384821",
      "name" : "frenchCity",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    {
      "id" : "285172405",
      "name" : "_statistics15",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
```

```
      {
        "id" : "616456885",
        "name" : "better-example",
        "isSystem" : false,
        "status" : 3,
        "type" : 2
      },
      {
        "id" : "771646133",
        "name" : "relation",
        "isSystem" : false,
        "status" : 3,
        "type" : 3
      },
      {
        "id" : "711156405",
        "name" : "friend_of",
        "isSystem" : false,
        "status" : 3,
        "type" : 3
      }
    ],
    "names" : {
      "_aqlfunctions" : {
        "id" : "4022965",
        "name" : "_aqlfunctions",
        "isSystem" : true,
        "status" : 3,
        "type" : 2
      },
      "_modules" : {
        "id" : "1794741",
        "name" : "_modules",
        "isSystem" : true,
        "status" : 3,
        "type" : 2
      },
      "Electronics" : {
        "id" : "711549621",
        "name" : "Electronics",
        "isSystem" : false,
        "status" : 3,
        "type" : 2
      },
      "_aal" : {
        "id" : "3564213",
        "name" : "_aal",
        "isSystem" : true,
        "status" : 3,
        "type" : 2
      },
      "germanHighway" : {
        "id" : "798253749",
        "name" : "germanHighway",
        "isSystem" : false,
        "status" : 3,
        "type" : 3
      },
      "internationalHighway" : {
        "id" : "798646965",
```

```
      "name" : "internationalHighway",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    "male" : {
      "id" : "771515061",
      "name" : "male",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "Customer" : {
      "id" : "711025333",
      "name" : "Customer",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "_graphs" : {
      "id" : "1663669",
      "name" : "_graphs",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "Groceries" : {
      "id" : "711418549",
      "name" : "Groceries",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "germanCity" : {
      "id" : "798122677",
      "name" : "germanCity",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "_cluster_kickstarter_plans" : {
      "id" : "2056885",
      "name" : "_cluster_kickstarter_plans",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "frenchHighway" : {
      "id" : "798515893",
      "name" : "frenchHighway",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    "female" : {
      "id" : "771383989",
      "name" : "female",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
```

```
    },
    "_statisticsRaw" : {
      "id" : "284385973",
      "name" : "_statisticsRaw",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "Company" : {
      "id" : "711287477",
      "name" : "Company",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "animals" : {
      "id" : "803758773",
      "name" : "animals",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "_statistics" : {
      "id" : "284779189",
      "name" : "_statistics",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "_users" : {
      "id" : "156341",
      "name" : "_users",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "_configuration" : {
      "id" : "4154037",
      "name" : "_configuration",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "_routing" : {
      "id" : "1925813",
      "name" : "_routing",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "has_bought" : {
      "id" : "711680693",
      "name" : "has_bought",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    "demo" : {
      "id" : "802841269",
      "name" : "demo",
```

```
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "frenchCity" : {
      "id" : "798384821",
      "name" : "frenchCity",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "_statistics15" : {
      "id" : "285172405",
      "name" : "_statistics15",
      "isSystem" : true,
      "status" : 3,
      "type" : 2
    },
    "better-example" : {
      "id" : "616456885",
      "name" : "better-example",
      "isSystem" : false,
      "status" : 3,
      "type" : 2
    },
    "relation" : {
      "id" : "771646133",
      "name" : "relation",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    },
    "friend_of" : {
      "id" : "711156405",
      "name" : "friend_of",
      "isSystem" : false,
      "status" : 3,
      "type" : 3
    }
  },
  "error" : false,
  "code" : 200
}
```

Modifying a Collection

loads a collection

Load collection

```
PUT /_api/collection/{collection-name}/load
```

URL Parameters

- *collection-name*: The name of the collection.

Loads a collection into memory. Returns the collection on success.

The request might optionally contain the following attribute:

- *count*: If set, this controls whether the return value should include the number of documents in the collection. Setting *count* to *false* may speed up loading a collection. The default value for *count* is *true*.

On success an object with the following attributes is returned:

- *id*: The identifier of the collection.

- *name*: The name of the collection.

- *count*: The number of documents inside the collection. This is only returned if the *count* input parameters is set to *true* or has not been specified.

- *status*: The status of the collection as number.

- *type*: The collection type. Valid types are:
  - 2: document collection
  - 3: edges collection

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/collection/828400309/load

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "828400309",
  "name" : "products",
  "isSystem" : false,
  "count" : 0,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

unloads a collection

Unload collection

```
PUT /_api/collection/{collection-name}/unload
```

URL Parameters

- *collection-name*:

Removes a collection from memory. This call does not delete any documents. You can use the collection afterwards; in which case it will be loaded into memory, again. On success an object with the following attributes is returned:

- *id*: The identifier of the collection.

- *name*: The name of the collection.

- *status*: The status of the collection as number.

- *type*: The collection type. Valid types are:
  - 2: document collection
  - 3: edges collection

Return Codes

- *400*: If the *collection-name* is missing, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

```
shell> curl -X PUT --dump -
http://localhost:8529/_api/collection/828662453/unload

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "828662453",
  "name" : "products",
  "isSystem" : false,
  "status" : 2,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

changes a collection

Change properties of a collection

```
PUT /_api/collection/{collection-name}/properties
```

URL Parameters

- *collection-name*: The name of the collection.

Changes the properties of a collection. Expects an object with the attribute(s)

- *waitForSync*: If *true* then creating or changing a document will wait until the data has been synchronised to disk.

- *journalSize*: Size (in bytes) for new journal files that are created for the collection.

If returns an object with the attributes

- *id*: The identifier of the collection.

- *name*: The name of the collection.

- *waitForSync*: The new value.

- *journalSize*: The new value.

- *status*: The status of the collection as number.

- *type*: The collection type. Valid types are:
    - 2: document collection
    - 3: edges collection

Note: some other collection properties, such as *type*, *isVolatile*, *numberOfShards* or *shardKeys* cannot be changed once a collection is created.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/collection/829252277/properties
{
  "waitForSync" : true
}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "829252277",
  "name" : "products",
  "isSystem" : false,
  "doCompact" : true,
  "isVolatile" : false,
  "journalSize" : 1048576,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  },
  "waitForSync" : true,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

renames a collection

Rename collection

```
PUT /_api/collection/{collection-name}/rename
```

URL Parameters

- *collection-name*: The name of the collection to rename.

Renames a collection. Expects an object with the attribute(s)

- *name*: The new name.

If returns an object with the attributes

- *id*: The identifier of the collection.

- *name*: The new name of the collection.

- *status*: The status of the collection as number.

- *type*: The collection type. Valid types are:
    - 2: document collection
    - 3: edges collection

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/collection/829514421/rename
{
  "name" : "newname"
}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "829514421",
  "name" : "newname",
  "isSystem" : false,
  "status" : 3,
  "type" : 2,
  "error" : false,
  "code" : 200
}
```

rotates the journal of a collection

Rotate journal of a collection

```
PUT /_api/collection/{collection-name}/rotate
```

URL Parameters

- *collection-name*: The name of the collection.

Rotates the journal of a collection. The current journal of the collection will be closed and made a read-only datafile. The purpose of the rotate method is to make the data in the file available for compaction (compaction is only performed for read-only datafiles, and not for journals).

Saving new data in the collection subsequently will create a new journal file automatically if there is no current journal.

If returns an object with the attributes

- *result*: will be *true* if rotation succeeded

Note: This method is not available in a cluster.

Return Codes

- *400*: If the collection currently has no journal, *HTTP 500* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Rotating a journal:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/collection/829776565/rotate
{
}

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "result" : true,
  "error" : false,
  "code" : 200
}
```

Rotating without a journal:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/collection/830497461/rotate
{
}

HTTP/1.1 400 Bad Request
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 400,
  "errorNum" : 1105,
  "errorMessage" : "could not rotate journal: no journal"
}
```

HTTP Interface for Indexes

Indexes

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

Index

Indexes are used to allow fast access to documents. For each collection there is always the primary index which is a hash index for the document key (_key attribute). This index cannot be dropped or changed. Edge collections will also have an automatically created edges index, which cannot be modified. This index provides quick access to documents via the _from and _to attributes.

Most user-land indexes can be created by defining the names of the attributes which should be indexed. Some index types allow indexing just one attribute (e.g. fulltext index) whereas other index types allow indexing multiple attributes.

Indexing system attributes such as _id, _key, _from, and _to in user-defined indexes is not supported by any index type. Manually creating an index that relies on any of these attributes is unsupported.

Index Handle

An index handle uniquely identifies an index in the database. It is a string and consists of a collection name and an index identifier separated by /. Geo Index: A geo index is used to find places on the surface of the earth fast. Hash Index: A hash index is used to find documents based on examples. A hash index can be created for one or multiple document attributes. A hash index will only be used by queries if all indexed attributes are present in the example or search query, and if all attributes are compared using the equality (== operator). That means the hash index does not support range queries.

If the index is declared unique, then access to the indexed attributes should be fast. The performance degrades if the indexed attribute(s) contain(s) only very few distinct values.

Edges Index

An edges index is automatically created for edge collections. It contains connections between vertex documents and is invoked when the connecting edges of a vertex are queried. There is no way to explicitly create or delete edge indexes.

Skiplist Index:

A skiplist is used to find ranges of documents.

Fulltext Index:

A fulltext index can be used to find words, or prefixes of words inside documents. A fulltext index can be set on one attribute only, and will index all words contained in documents that have a textual value in this attribute. Only words with a (specifiable) minimum length are indexed. Word tokenization is done using the word boundary analysis provided by libicu, which is taking into account the selected language provided at server start. Words are indexed in their lower-cased form. The index supports complete match queries (full words) and prefix queries.

The basic operations (create, read, update, delete) for documents are mapped to the standard HTTP methods (*POST, GET, PUT, DELETE*).

Address of an Index

All indexes in ArangoDB have an unique handle. This index handle identifies an index and is managed by ArangoDB. All indexes are found under the URI

```
http://server:port/_api/index/index-handle
```

For example: Assume that the index handle is *demo/63563528* then the URL of that index is:

```
http://localhost:8529/_api/index/demo/63563528
```

Working with Indexes using HTTP

returns an index

Read index

```
GET /_api/index/{index-handle}
```

URL Parameters

- *index-handle*: The index-handle.

The result is an objects describing the index. It has at least the following attributes:

- *id*: The identifier of the index.

All other attributes are type-dependent.

Return Codes

- *200*: If the index exists, then a *HTTP 200* is returned.

- *404*: If the index does not exist, then a *HTTP 404* is returned.

Examples

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/index/products/0
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "products/0",
  "type" : "primary",
  "unique" : true,
  "fields" : [
    "_key"
  ],
  "error" : false,
  "code" : 200
}
```

creates an index

Create index

```
POST /_api/index
```

Query Parameters

- *collection*: The collection name.

- *index-details*:

Creates a new index in the collection *collection*. Expects an object containing the index details.

The type of the index to be created must specified in the *type* attribute of the index details. Depending on the index type, additional other attributes may need to specified in the request in order to create the index.

Most indexes (a notable exception being the cap constraint) require the list of attributes to be indexed in the *fields* attribute of the index details. Depending on the index type, a single attribute or multiple attributes may be indexed.

Indexing system attributes such as *_id*, *_key*, *_from*, and *_to* is not supported by any index type. Manually creating an index that relies on any of these attributes is unsupported.

Some indexes can be created as unique or non-unique variants. Uniqueness can be controlled for most indexes by specifying the *unique* in the index details. Setting it to *true* will create a unique index. Setting it to *false* or omitting the *unique* attribute will create a non-unique index.

Note: The following index types do not support uniqueness, and using the *unique* attribute with these types may lead to an error:

- cap constraints
- fulltext indexes
- bitarray indexes

Note: Unique indexes on non-shard keys are not supported in a cluster.

Return Codes

- *200*: If the index already exists, then an *HTTP 200* is returned.

- *201*: If the index does not already exist and could be created, then an *HTTP 201* is returned.

- *400*: If an invalid index description is posted or attributes are used that the target index will not support, then an *HTTP 400* is returned.

- *404*: If *collection* is unknown, then an *HTTP 404* is returned. deletes an index

Delete index

DELETE /_api/index/{index-handle}

Query Parameters

- *index-handle*: The index handle.

Deletes an index with *index-handle*.

Return Codes

- *200*: If the index could be deleted, then an *HTTP 200* is returned.

- *404*: If the *index-handle* is unknown, then an *HTTP 404* is returned. **Examples**

```
shell> curl -X DELETE --data-binary @- --dump -
http://localhost:8529/_api/index/products/978281141
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "id" : "products/978281141",
  "error" : false,
```

```
    "code" : 200
}
```

returns all indexes of a collection

Read all indexes of a collection

```
GET /_api/index
```

Query Parameters

- *collection*: The collection name.

Returns an object with an attribute *indexes* containing a list of all index descriptions for the given collection. The same information is also available in the *identifiers* as hash map with the index handle as keys.

Examples

Return information about all indexes:

```
shell> curl --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
undefined

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "indexes" : [
    {
      "id" : "products/0",
      "type" : "primary",
      "unique" : true,
      "fields" : [
        "_key"
      ]
    }
  ],
  "identifiers" : {
    "products/0" : {
      "id" : "products/0",
      "type" : "primary",
      "unique" : true,
      "fields" : [
        "_key"
      ]
    }
  },
  "error" : false,
  "code" : 200
}
```

Working with Cap Constraints

creates a cap constraint

Create cap constraint

`POST /_api/index`

Query Parameters

- *collection*: The collection name.

- *cap-constraint*:

Creates a cap constraint for the collection *collection-name*, if it does not already exist. Expects an object containing the index details.

- *type*: must be equal to *"cap"*.

- *size*: The maximal number of documents for the collection. If specified, the value must be greater than zero.

- *byteSize*: The maximal size of the active document data in the collection (in bytes). If specified, the value must be at least 16384.

Note: The cap constraint does not index particular attributes of the documents in a collection, but limits the number of documents in the collection to a maximum value. The cap constraint thus does not support attribute names specified in the *fields* attribute nor uniqueness of any kind via the *unique* attribute.

It is allowed to specify either *size* or *byteSize*, or both at the same time. If both are specified, then the automatic document removal will be triggered by the first non-met constraint.

Return Codes

- *200*: If the index already exists, then an *HTTP 200* is returned.

- *201*: If the index does not already exist and could be created, then an *HTTP 201* is returned.

- *400*: If either *size* or *byteSize* contain invalid values, then an *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Creating a cap constraint

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{"type":"cap","size":10}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "id" : "products/974152373",
  "type" : "cap",
  "unique" : false,
  "size" : 10,
  "byteSize" : 0,
```

```
    "isNewlyCreated" : true,
    "error" : false,
    "code" : 201
}
```

Working with Hash Indexes

If a suitable hash index exists, then */_api/simple/by-example* will use this index to execute a query-by-example.

creates a hash index

Create hash index

POST /_api/index

Query Parameters

- *collection-name*: The collection name.

- *index-details*:

Creates a hash index for the collection *collection-name*, if it does not already exist. The call expects an object containing the index details.

- *type*: must be equal to *"hash"*.

- *fields*: A list of attribute paths.

- *unique*: If *true*, then create a unique index.

Note: unique indexes on non-shard keys are not supported in a cluster.

Return Codes

- *200*: If the index already exists, then a *HTTP 200* is returned.

- *201*: If the index does not already exist and could be created, then a *HTTP 201* is returned.

- *400*: If the collection already contains documents and you try to create a unique hash index in such a way that there are documents violating the uniqueness, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Creating an unique constraint:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{ "type": "hash", "unique" : true, "fields" : [ "a", "b" ] }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "id" : "products/975725237",
  "type" : "hash",
  "unique" : true,
  "fields" : [
    "a",
    "b"
  ],
  "isNewlyCreated" : true,
```

```
    "error" : false,
    "code" : 201
}
```

Creating a hash index:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{ "type": "hash", "unique" : false, "fields" : [ "a", "b" ] }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
    "id" : "products/976249525",
    "type" : "hash",
    "unique" : false,
    "fields" : [
      "a",
      "b"
    ],
    "isNewlyCreated" : true,
    "error" : false,
    "code" : 201
}
```

returns all documents of a collection matching a given example

Simple query by-example

PUT /_api/simple/by-example

- *query*: Contains the query.

This will find all documents matching a given example.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *example*: The example document.

- *skip*: The number of documents to skip in the query (optional).

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Matching an attribute:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/by-example
{ "collection": "products", "example" :  { "i" : 1 }  }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/998728373",
      "_rev" : "998728373",
      "_key" : "998728373",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    },
    {
      "_id" : "products/998400693",
      "_rev" : "998400693",
      "_key" : "998400693",
      "i" : 1,
      "a" : {
        "k" : 1,
        "j" : 1
      }
    },
    {
      "_id" : "products/999187125",
      "_rev" : "999187125",
      "_key" : "999187125",
      "i" : 1,
      "a" : {
        "k" : 2,
        "j" : 2
      }
    },
    {
      "_id" : "products/998990517",
      "_rev" : "998990517",
      "_key" : "998990517",
      "i" : 1
    }
  ],
  "hasMore" : false,
  "count" : 4,
  "error" : false,
  "code" : 201
}
```

Matching an attribute which is a sub-document:

```
shell> curl -X PUT --data-binary @- --dump -
```

```
http://localhost:8529/_api/simple/by-example
{ "collection": "products", "example" : { "a.j" : 1 } }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/1000759989",
      "_rev" : "1000759989",
      "_key" : "1000759989",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    },
    {
      "_id" : "products/1000432309",
      "_rev" : "1000432309",
      "_key" : "1000432309",
      "i" : 1,
      "a" : {
        "k" : 1,
        "j" : 1
      }
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

Matching an attribute within a sub-document:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/by-example
{ "collection": "products", "example" : { "a" : { "j" : 1 } } }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/1002791605",
      "_rev" : "1002791605",
      "_key" : "1002791605",
      "i" : 1,
      "a" : {
        "j" : 1
      }
    }
  ],
  "hasMore" : false,
  "count" : 1,
  "error" : false,
```

```
    "code" : 201
}
```

returns one document of a collection matching a given example

Document matching an example

```
PUT /_api/simple/first-example
```

- *query*: Contains the query.

This will return the first document matching a given example.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *example*: The example document.

Returns a result containing the document or *HTTP 404* if no document matched the example.

If more than one document in the collection matches the specified example, only one of these documents will be returned, and it is undefined which of the matching documents is returned.

Return Codes

- *200*: is returned when the query was successfully executed.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

If a matching document was found:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/first-example
{ "collection": "products", "example" :   { "i" : 1 }   }

HTTP/1.1 200 OK
content-type: application/json; charset=utf-8

{
  "document" : {
    "_id" : "products/1004495541",
    "_rev" : "1004495541",
    "_key" : "1004495541",
    "i" : 1,
    "a" : {
      "k" : 1,
      "j" : 1
    }
  },
  "error" : false,
  "code" : 200
}
```

If no document was found:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/first-example
{ "collection": "products", "example" :  { "l" : 1 }  }

HTTP/1.1 404 Not Found
content-type: application/json; charset=utf-8

{
  "error" : true,
  "code" : 404,
  "errorNum" : 404,
  "errorMessage" : "no match"
}
```

Working with Skiplist Indexes

If a suitable skip-list index exists, then /_api/simple/range will use this index to execute a range query.

creates a skip-list

Create skip list

```
POST /_api/index
```

Query Parameters

- *collection-name*: The collection name.

- *index-details*:

Creates a skip-list index for the collection *collection-name,* if it does not already exist. The call expects an object containing the index details.

- *type*: must be equal to *"skiplist"*.

- *fields*: A list of attribute paths.

- *unique*: If *true*, then create a unique index.

Note: unique indexes on non-shard keys are not supported in a cluster.

Return Codes

- *200*: If the index already exists, then a *HTTP 200* is returned.

- *201*: If the index does not already exist and could be created, then a *HTTP 201* is returned.

- *400*: If the collection already contains documents and you try to create a unique skip-list index in such a way that there are documents violating the uniqueness, then a *HTTP 400* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Creating a skiplist:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{ "type": "skiplist", "unique" : false, "fields" : [ "a", "b" ] }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "id" : "products/976773813",
  "type" : "skiplist",
  "unique" : false,
  "fields" : [
    "a",
    "b"
  ],
  "isNewlyCreated" : true,
  "error" : false,
```

```
    "code" : 201
}
```

Working with Geo Indexes

creates a geo index

Create geo-spatial index

`POST /_api/index`

Query Parameters

- *collection*: The collection name.

- *index-details*:

Creates a geo-spatial index in the collection *collection-name*, if it does not already exist. Expects an object containing the index details.

- *type*: must be equal to *"geo"*.

- *fields*: A list with one or two attribute paths.

If it is a list with one attribute path *location*, then a geo-spatial index on all documents is created using *location* as path to the coordinates. The value of the attribute must be a list with at least two double values. The list must contain the latitude (first value) and the longitude (second value). All documents, which do not have the attribute path or with value that are not suitable, are ignored.

If it is a list with two attribute paths *latitude* and *longitude*, then a geo-spatial index on all documents is created using *latitude* and *longitude* as paths the latitude and the longitude. The value of the attribute *latitude* and of the attribute *longitude* must a double. All documents, which do not have the attribute paths or which values are not suitable, are ignored.

- *geoJson*: If a geo-spatial index on a *location* is constructed and *geoJson* is *true*, then the order within the list is longitude followed by latitude. This corresponds to the format described in http://geojson.org/geojson-spec.html#positions

- *constraint*: If *constraint* is *true*, then a geo-spatial constraint is created. The constraint is a non-unique variant of the index. **Note**: It is also possible to set the *unique* attribute instead of the *constraint* attribute.

- *ignoreNull*: If a geo-spatial constraint is created and *ignoreNull* is true, then documents with a null in *location* or at least one null in *latitude* or *longitude* are ignored.

Note: Unique indexes on non-shard keys are not supported in a cluster.

Return Codes

- *200*: If the index already exists, then a *HTTP 200* is returned.

- *201*: If the index does not already exist and could be created, then a *HTTP 201* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Creating a geo index with a location attribute:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{ "type": "geo", "fields" : [ "b" ] }
```

```
HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "id" : "products/974676661",
  "type" : "geo1",
  "unique" : false,
  "geoJson" : false,
  "constraint" : false,
  "ignoreNull" : false,
  "fields" : [
    "b"
  ],
  "isNewlyCreated" : true,
  "error" : false,
  "code" : 201
}
```

Creating a geo index with latitude and longitude attributes:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{ "type": "geo", "fields" : [ "e", "f" ] }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "id" : "products/975200949",
  "type" : "geo2",
  "unique" : false,
  "constraint" : false,
  "ignoreNull" : false,
  "fields" : [
    "e",
    "f"
  ],
  "isNewlyCreated" : true,
  "error" : false,
  "code" : 201
}
```

returns all documents of a collection near a given location

Near query

```
PUT /_api/simple/near
```

- *query*: Contains the query.

The default will find at most 100 documents near the given coordinate. The returned list is sorted according to the distance, with the nearest document being first in the list. If there are near documents of equal distance, documents are chosen randomly from this set until the limit is reached.

In order to use the *near* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* field to select a particular index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *latitude*: The latitude of the coordinate.

- *longitude*: The longitude of the coordinate.

- *distance*: If given, the attribute key used to return the distance to the given coordinate. (optional). If specified, distances are returned in meters.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. The default is 100. (optional)

- *geo*: If given, the identifier of the geo-index to use. (optional)

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Without distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 2 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/986276533",
      "_rev" : "986276533",
      "_key" : "986276533",
      "name" : "Name/0.002/",
      "loc" : [
        0.002,
        0
      ]
    },
    {
      "_id" : "products/985883317",
      "_rev" : "985883317",
      "_key" : "985883317",
      "name" : "Name/-0.002/",
      "loc" : [
```

```
            -0.002,
            0
          ]
        }
    ],
    "hasMore" : false,
    "count" : 2,
    "error" : false,
    "code" : 201
}
```

With distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 3, "distance" : "distance" }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
    "result" : [
        {
            "_id" : "products/988897973",
            "_rev" : "988897973",
            "_key" : "988897973",
            "name" : "Name/-0.002/",
            "loc" : [
                -0.002,
                0
            ],
            "distance" : 222.38985328911744
        },
        {
            "_id" : "products/989291189",
            "_rev" : "989291189",
            "_key" : "989291189",
            "name" : "Name/0.002/",
            "loc" : [
                0.002,
                0
            ],
            "distance" : 222.38985328911744
        },
        {
            "_id" : "products/988701365",
            "_rev" : "988701365",
            "_key" : "988701365",
            "name" : "Name/-0.004/",
            "loc" : [
                -0.004,
                0
            ],
            "distance" : 444.779706578235
        }
    ],
    "hasMore" : false,
```

```
  "count" : 3,
  "error" : false,
  "code" : 201
}
```

returns all documents of a collection within a given radius

Within query

```
PUT /_api/simple/within
```

- *query*: Contains the query.

This will find all documents within a given radius around the coordinate (*latitude*, *longitude*). The returned list is sorted by distance.

In order to use the *within* operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the *geo* field to select a particular index.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *latitude*: The latitude of the coordinate.

- *longitude*: The longitude of the coordinate.

- *radius*: The maximal radius (in meters).

- *distance*: If given, the attribute key used to return the distance to the given coordinate. (optional). If specified, distances are returned in meters.

- *skip*: The number of documents to skip in the query. (optional)

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. The default is 100. (optional)

- *geo*: If given, the identifier of the geo-index to use. (optional)

Returns a cursor containing the result, see Http Cursor for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

Without distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 2, "radius" : 500 }
```

```
HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/992305845",
      "_rev" : "992305845",
      "_key" : "992305845",
      "name" : "Name/0.002/",
      "loc" : [
        0.002,
        0
      ]
    },
    {
      "_id" : "products/991912629",
      "_rev" : "991912629",
      "_key" : "991912629",
      "name" : "Name/-0.002/",
      "loc" : [
        -0.002,
        0
      ]
    }
  ],
  "hasMore" : false,
  "count" : 2,
  "error" : false,
  "code" : 201
}
```

With distance:

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/near
{ "collection": "products", "latitude" : 0, "longitude" : 0, "skip"
: 1, "limit" : 3, "distance" : "distance", "radius" : 300 }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/994927285",
      "_rev" : "994927285",
      "_key" : "994927285",
      "name" : "Name/-0.002/",
      "loc" : [
        -0.002,
        0
      ],
      "distance" : 222.38985328911744
    },
    {
      "_id" : "products/995320501",
```

```
      "_rev" : "995320501",
      "_key" : "995320501",
      "name" : "Name/0.002/",
      "loc" : [
        0.002,
        0
      ],
      "distance" : 222.38985328911744
    },
    {
      "_id" : "products/994730677",
      "_rev" : "994730677",
      "_key" : "994730677",
      "name" : "Name/-0.004/",
      "loc" : [
        -0.004,
        0
      ],
      "distance" : 444.779706578235
    }
  ],
  "hasMore" : false,
  "count" : 3,
  "error" : false,
  "code" : 201
}
```

Fulltext

If a fulltext index exists, then /_api/simple/fulltext will use this index to execute the specified fulltext query.

creates a fulltext index

Create fulltext index

```
POST /_api/index
```

Query Parameters

- *collection-name*: The collection name.

- *index-details*:

Creates a fulltext index for the collection *collection-name*, if it does not already exist. The call expects an object containing the index details.

- *type*: must be equal to *"fulltext"*.

- *fields*: A list of attribute names. Currently, the list is limited to exactly one attribute, so the value of *fields* should look like this for example: *["text"]*.

- *minLength*: Minimum character length of words to index. Will default to a server-defined value if unspecified. It is thus recommended to set this value explicitly when creating the index.

Return Codes

- *200*: If the index already exists, then a *HTTP 200* is returned.

- *201*: If the index does not already exist and could be created, then a *HTTP 201* is returned.

- *404*: If the *collection-name* is unknown, then a *HTTP 404* is returned.

Examples

Creating a fulltext index:

```
shell> curl -X POST --data-binary @- --dump -
http://localhost:8529/_api/index?collection=products
{ "type" : "fulltext", "fields" : [ "text" ] }

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "id" : "products/977232565",
  "type" : "fulltext",
  "unique" : false,
  "minLength" : 2,
  "fields" : [
    "text"
  ],
  "isNewlyCreated" : true,
  "error" : false,
  "code" : 201
}
```

returns documents of a collection as a result of a fulltext query

Fulltext index query

```
PUT /_api/simple/fulltext
```

- *query*: Contains the query.

This will find all documents from the collection that match the fulltext query specified in *query*.

In order to use the *fulltext* operator, a fulltext index must be defined for the collection and the specified attribute.

The call expects a JSON object as body with the following attributes:

- *collection*: The name of the collection to query.

- *attribute*: The attribute that contains the texts.

- *query*: The fulltext query.

- *skip*: The number of documents to skip in the query (optional).

- *limit*: The maximal amount of documents to return. The *skip* is applied before the *limit* restriction. (optional)

- *index*: The identifier of the fulltext-index to use.

Returns a cursor containing the result, see [Http Cursor](#) for details.

Return Codes

- *201*: is returned if the query was executed successfully.

- *400*: is returned if the body does not contain a valid JSON representation of a query. The response body contains an error document in this case.

- *404*: is returned if the collection specified by *collection* is unknown. The response body contains an error document in this case.

Examples

```
shell> curl -X PUT --data-binary @- --dump -
http://localhost:8529/_api/simple/fulltext
{ "collection": "products", "attribute" : "text", "query" : "word"
}

HTTP/1.1 201 Created
content-type: application/json; charset=utf-8

{
  "result" : [
    {
      "_id" : "products/997024437",
      "_rev" : "997024437",
      "_key" : "997024437",
      "text" : "this text also has a word"
    },
    {
```

```
        "_id" : "products/996827829",
        "_rev" : "996827829",
        "_key" : "996827829",
        "text" : "this text contains word"
      }
    ],
    "hasMore" : false,
    "count" : 2,
    "error" : false,
    "code" : 201
}
```

JavaScript Modules

Introduction to Javascript Modules

The ArangoDB uses a [CommonJS](#) compatible module and package concept. You can use the function *require* in order to load a module or package. It returns the exported variables and functions of the module or package.

There are some extensions to the CommonJS concept to allow ArangoDB to load Node.js modules as well.

CommonJS Modules

Unfortunately, the JavaScript libraries are just in the process of being standardized. CommonJS has defined some important modules. ArangoDB implements the following

- "console" is a well known logging facility to all the JavaScript developers. ArangoDB implements all of the functions described <u>here</u>, with the exceptions of *profile* and *count*.

- "fs" provides a file system API for the manipulation of paths, directories, files, links, and the construction of file streams. ArangoDB implements most of Filesystem/A functions described <u>here</u>.

- Modules are implemented according to <u>Modules/1.1.1</u>

- Packages are implemented according to <u>Packages/1.0</u>

ArangoDB Specific Modules

A lot of the modules, however, are ArangoDB specific. These modules are described in the following chapters.

Node Modules

ArangoDB also support some <u>node</u>modules.

- <u>"assert"</u> implements assertion and testing functions.

- <u>"buffer"</u> implements a binary data type for JavaScript.

- <u>"path"</u> implements functions dealing with filenames and paths.

- <u>"punycode"</u> implements conversion functions for <u>punycode</u> encoding.

- <u>"querystring"</u> provides utilities for dealing with query strings.

- <u>"stream"</u> provides a streaming interface.

- <u>"url"</u> has utilities for URL resolution and parsing.

Node Packages

The following <u>node packages</u> are preinstalled.

- <u>"buster-format"</u>

- <u>"Cheerio.JS"</u>

- <u>"coffee-script"</u> implements a coffee-script to JavaScript compiler. ArangoDB supports the *compile* function of the package, but not the *eval* functions.

- <u>"htmlparser2"</u>

- <u>"Sinon.JS"</u>

- <u>"underscore"</u> is a utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Prototype.js (or Ruby), but without extending any of the built-in JavaScript objects.

require

```
require(path)
```

require checks if the module or package specified by *path* has already been loaded. If not, the content of the file is executed in a new context. Within the context you can use the global variable *exports* in order to export variables and functions. This variable is returned by *require*.

Assume that your module file is *test1.js* and contains

```
exports.func1 = function() {
  print("1");
};

exports.const1 = 1;
```

Then you can use *require* to load the file and access the exports.

```
unix> ./arangosh
arangosh> var test1 = require("test1");

arangosh> test1.const1;
1

arangosh> test1.func1();
1
```

require follows the specification Modules/1.1.1.

Modules Path versus Modules Collection

ArangoDB comes with predefined modules defined in the file-system under the path specified by *startup.startup-directory*. In a standard installation this point to the system share directory. Even if you are an administrator of ArangoDB you might not have write permissions to this location. On the other hand, in order to deploy some extension for ArangoDB, you might need to install additional JavaScript modules. This would require you to become root and copy the files into the share directory. In order to ease the deployment of extensions, ArangoDB uses a second mechanism to look up JavaScript modules.

JavaScript modules can either be stored in the filesystem as regular file or in the database collection *_modules*.

If you execute

```
require("com/example/extension")
```

then ArangoDB will try to locate the corresponding JavaScript as file as follows

- There is a cache for the results of previous *require* calls. First of all ArangoDB checks if *com/example/extension* is already in the modules cache. If it is, the export object for this module is returned. No further JavaScript is executed.

- ArangoDB will then check, if there is a file called **com/example/extension.js** in the system search path. If such a file exists, it is executed in a new module context and the value of *exports* object is returned. This value is also stored in the module cache.

- If no file can be found, ArangoDB will check if the collection *_modules* contains a document of the form

```
{
  path: "/com/example/extension",
  content: "...."
}
```

Note: The leading */* is important - even if you call *require* without a leading */*. If such a document exists, then the value of the *content* attribute must contain the JavaScript code of the module. This string is executed in a new module context and the value of *exports* object is returned. This value is also stored in the module cache.

!SUBSECTION Modules Cache

As *require* uses a module cache to store the exports objects of the required modules, changing the design documents for the modules in the *_modules* collection might have no effect at all.

You need to clear the cache, when manually changing documents in the *_modules* collection.

```
arangosh> require("internal").flushServerModules()
```

This initiate a flush of the modules in the ArangoDB *server* process.

Please note, that the ArangoDB JavaScript shell uses the same mechanism as the server to locate JavaScript modules. But the do not share the same module cache. If you flush the server cache, this will not flush the shell cache - and vice versa.

In order to flush the modules cache of the JavaScript shell, you should use

```
arangosh> require("internal").flushModuleCache()
```

Console Module

The implementation follows the CommonJS specification [Console](#).

console.assert

```
console.assert(expression, format, argument1, ...)
```

Tests that an expression is *true*. If not, logs a message and throws an exception.

Examples

```
console.assert(value === "abc", "expected: value === abc, actual:",
value);
```

console.debug

```
console.debug(format, argument1, ...)
```

Formats the arguments according to *format* and logs the result as debug message. Note that debug messages will only be logged if the server is started with log levels *debug* or *trace*. Please refer to @ref CommandLineLoggingHuman for details.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Examples

```
console.debug("%s", "this is a test");
```

console.dir

```
console.dir(object)
```

Logs a listing of all properties of the object.

Example usage:

```
console.dir(myObject);
```

console.error

```
console.error(format, argument1, ...)
```

Formats the arguments according to @FA{format} and logs the result as error message.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer

- *%%f* floating point number
- *%%o* object hyperlink

Example usage:

```
console.error("error '%s': %s", type, message);
```

console.getline

```
console.getline()
```

Reads in a line from the console and returns it as string.

console.group

```
console.group(format, argument1, ...)
```

Formats the arguments according to *format* and logs the result as log message. Opens a nested block to indent all future messages sent. Call *groupEnd* to close the block. Representation of block is up to the platform, it can be an interactive block or just a set of indented sub messages.

Example usage:

```
console.group("user attributes");
console.log("name", user.name);
console.log("id", user.id);
console.groupEnd();
```

console.groupCollapsed

```
console.groupCollapsed(format, argument1, ...)
```

Same as *console.group*, but with the group initially collapsed.

console.groupEnd

```
console.groupEnd()
```

Closes the most recently opened block created by a call to *group*.

console.info

```
console.info(format, argument1, ...)
```

Formats the arguments according to *format* and logs the result as info message.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Example usage:

```
console.info("The %s jumped over %d fences", animal, count);
```

console.log

```
console.log(format, argument1, ...)
```

Formats the arguments according to *format* and logs the result as log message. This is an alias for *console.info*.

console.time

```
console.time(name)
```

Creates a new timer under the given name. Call *timeEnd* with the same name to stop the timer and log the time elapsed.

Example usage:

```
console.time("mytimer");
...
console.timeEnd("mytimer"); // this will print the elapsed time
```

console.timeEnd

```
console.timeEnd(name)
```

Stops a timer created by a call to *time* and logs the time elapsed.

console.timeEnd

```
console.trace()
```

Logs a stack trace of JavaScript execution at the point where it is called.

console.warn

```
console.warn(format, argument1, ...)
```

Formats the arguments according to *format* and logs the result as warn message.

String substitution patterns, which can be used in *format*.

- *%%s* string
- *%%d*, *%%i* integer
- *%%f* floating point number
- *%%o* object hyperlink

Module "fs"

File System Module

The implementation follows the CommonJS specification [Filesystem/A/0](#).

```
fs.exists(path)
```

Returns true if a file (of any type) or a directory exists at a given path. If the file is a broken symbolic link, returns false.

```
fs.isDirectory(path)
```

Returns true if the *path* points to a directory.

```
fs.isFile(path)
```

Returns true if the *path* points to a file.

```
fs.list(path
```

The functions returns the names of all the files in a directory, in lexically sorted order. Throws an exception if the directory cannot be traversed (or path is not a directory).

Note: this means that list("x") of a directory containing "a" and "b" would return ["a", "b"], not ["x/a", "x/b"].

```
fs.listTree(path)
```

The function returns an array that starts with the given path, and all of the paths relative to the given path, discovered by a depth first traversal of every directory in any visited directory, reporting but not traversing symbolic links to directories. The first path is always *""*, the path relative to itself.

```
fs.move(source, destination)
```

Moves *source to destination. Failure to move the file, or specifying a directory for target when source is a file will throw an exception.

```
fs.read(filename)
```

Reads in a file and returns the content as string. Please note that the file content must be encoded in UTF-8.

```
fs.read64(filename)
```

Reads in a file and returns the content as string. The file content is Base64 encoded.

```
fs.remove(filename)
```

Removes the file *filename* at the given path. Throws an exception if the path corresponds to anything that is not a file or a symbolic link. If "path" refers to a symbolic link, removes the symbolic link.

Module "graph"

Warning: Deprecated

First Steps with Graphs

A Graph consists of vertices and edges. The vertex collection contains the documents forming the vertices. The edge collection contains the documents forming the edges. Together both collections form a graph. Assume that the vertex collection is called *vertices* and the edges collection *edges*, then you can build a graph using the *Graph* constructor.

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> g1 = new Graph("graph", "vertices", "edges");
Graph("vertices", "edges")
```

It is possible to use different edges with the same vertices. For instance, to build a new graph with a different edge collection use

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> g2 = new Graph("graph", "vertices", "alternativeEdges");
Graph("vertices", "alternativeEdges")
```

It is, however, impossible to use different vertices with the same edges. Edges are tied to the vertices.

Graph Constructors and Methods

The graph module provides basic functions dealing with graph structures. The examples assume

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> g = new Graph("graph", "vertices", "edges");
Graph("graph")
```

`Graph(name, vertices, edges)`

Constructs a new graph object using the collection vertices for all vertices and the collection edges for all edges. Note that it is possible to construct two graphs with the same vertex set, but different edge sets.

`Graph(name)`

Returns a known graph.

Examples

```
arango> var Graph = require("org/arangodb/graph").Graph;

arango> new Graph("graph", db.vertices, db.edges);
Graph("graph")

arango> new Graph("graph", "vertices", "edges");
Graph("graph")
```

`graph.addEdge( out, in, id)`

Creates a new edge from out to in and returns the edge object. The identifier id must be a unique identifier or null. out and in can either be vertices or their IDs

`graph.addEdge( out, in, id, label)`

Creates a new edge from out to in with label and returns the edge object. out and in can either be vertices or their IDs

`graph.addEdge( out, in, id, data)`

Creates a new edge and returns the edge object. The edge contains the properties defined in data. out and in can either be vertices or their IDs

`graph.addEdge( out, in, id, label, data)`

Creates a new edge and returns the edge object. The edge has the label label and contains the properties defined in data. out and in can either be vertices or their IDs

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)
```

```
arango> e = g.addEdge(v1, v2, 3);
Edge(3)
arango> e = g.addEdge(v1, v2, 4, "1->2", { name : "Emil");
Edge(4)
```

`graph.addVertex( id)`

Creates a new vertex and returns the vertex object. The identifier id must be a unique identifier or null.

`graph.addVertex( id, data)`

Creates a new vertex and returns the vertex object. The vertex contains the properties defined in data.

Examples

Without any properties:

```
arango> v = g.addVertex("hugo");
Vertex("hugo")
With given properties:

arango> v = g.addVertex("Emil", { age : 123 });
Vertex("Emil")

arango> v.getProperty("age");
123
```

`graph.getEdges()`

Returns an iterator for all edges of the graph. The iterator supports the methods hasNext and next.

Examples

```
arango> f = g.getEdges();
[edge iterator]

arango> f.hasNext();
true

arango> e = f.next();
Edge("4636053")

graph.getVertex( id)
Returns the vertex identified by id or null.
```

Examples

```
arango> g.addVertex(1);
Vertex(1)

arango> g.getVertex(1)
Vertex(1)
```

`graph.getVertices()`

Returns an iterator for all vertices of the graph. The iterator supports the methods hasNext and next.

Examples

```
arango> f = g.getVertices();
[vertex iterator]

arango> f.hasNext();
true

arango> v = f.next();
Vertex(18364)
```

graph.removeVertex(vertex, waitForSync)

Deletes the vertex and all its edges.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> e = g.addEdge(v1, v2, 3);
Edge(3)

arango> g.removeVertex(v1);

arango> v2.edges();
[ ]
```

graph.removeEdge(vertex, waitForSync)

Deletes the edge. Note that the in and out vertices are left untouched.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> e = g.addEdge(v1, v2, 3);
Edge(3)

arango> g.removeEdge(e);

arango> v2.edges();
[ ]
```

graph.drop(waitForSync)

Drops the graph, the vertices, and the edges. Handle with care.

graph.getAll()

Returns all available graphs.

`graph.geodesics( options)`

Return all shortest paths An optional options JSON object can be specified to control the result. options can have the following sub-attributes:

- grouped: if not specified or set to false, the result will be a flat list. If set to true, the result will be a list containing list of paths, grouped for each combination of source and target.
- threshold: if not specified, all paths will be returned. If threshold is true, only paths with a minimum length of 3 will be returned

`graph.measurement( measurement)`

Calculates the diameter or radius of a graph. measurement can either be:

- diameter: to calculate the diameter
- radius: to calculate the radius

`graph.normalizedMeasurement( measurement)`

Calculates the normalized degree, closeness, betweenness or eccentricity of all vertices in a graph measurement can either be:

- closeness: to calculate the closeness
- betweenness: to calculate the betweenness
- eccentricity: to calculate the eccentricity

Vertex Methods

`vertex.addInEdge( peer, id)`

Creates a new edge from peer to vertex and returns the edge object. The identifier id must be a unique identifier or null.

`vertex.addInEdge( peer, id, label)`

Creates a new edge from peer to vertex with given label and returns the edge object.

`vertex.addInEdge( peer, id, label, data)`

Creates a new edge from peer to vertex with given label and properties defined in data. Returns the edge object.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> v1.addInEdge(v2, "2 -> 1");
Edge("2 -> 1")

arango> v1.getInEdges();
[ Edge("2 -> 1") ]
arango> v1.addInEdge(v2, "D", "knows", { data : 1 });
Edge("D")

arango> v1.getInEdges();
[ Edge("K"), Edge("2 -> 1"), Edges("D") ]
```

`vertex.addOutEdge( peer)`

Creates a new edge from vertex to peer and returns the edge object.

`vertex.addOutEdge( peer, label)`

Creates a new edge from vertex to peer with given label and returns the edge object.

`vertex.addOutEdge( peer, label, data)`

Creates a new edge from vertex to peer with given label and properties defined in data. Returns the edge object.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex(2);
Vertex(2)

arango> v1.addOutEdge(v2, "1->2");
```

```
Edge("1->2")

arango> v1.getOutEdges();
[ Edge(1->2") ]
arango> v1.addOutEdge(v2, 3, "knows");
Edge(3)
arango> v1.addOutEdge(v2, 4, "knows", { data : 1 });
Edge(4)
```

`vertex.edges()`

Returns a list of in- or outbound edges of the vertex.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> v2 = g.addVertex();
Vertex(2)

arango> e = g.addEdge(v1, v2, "1->2");
Edge("1->2")

arango> v1.edges();
[  Edge("1->2") ]

arango> v2.edges();
[  Edge("1->2") ]
```

`vertex.getId()`

Returns the identifier of the vertex. If the vertex was deleted, then undefined is returned.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.getId();
"1"
```

`vertex.getInEdges( label, ...)`

Returns a list of inbound edges of the vertex with given label(s).

Examples

```
arango> v1 = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v2 = g.addVertex(2, { name : "Emil" });
Vertex(2)

arango> e1 = g.addEdge(v1, v2, 3, "knows");
Edge(3)
```

```
arango> e2 = g.addEdge(v1, v2, 4, "hates");
Edge(4)

arango> v2.getInEdges();
[ Edge(3), Edge(4) ]

arango> v2.getInEdges("knows");
[ Edge(3) ]

arango> v2.getInEdges("hates");
[ Edge(4) ]

arango> v2.getInEdges("knows", "hates");
[ Edge(3), Edge(4) ]
```

vertex.getOutEdges(label, ...)

Returns a list of outbound edges of the vertex with given label(s).

Examples

```
arango> v1 = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v2 = g.addVertex(2, { name : "Emil" });
Vertex(2)

arango> e1 = g.addEdge(v1, v2, 3, "knows");
Edge(3)

arango> e2 = g.addEdge(v1, v2, 4, "hates");
Edge(4)

arango> v1.getOutEdges();
[ Edge(3), Edge(4) ]

arango> v1.getOutEdges("knows");
[ Edge(3) ]

arango> v1.getOutEdges("hates");
[ Edge(4) ]

arango> v1.getOutEdges("knows", "hates");
[ Edge(3), Edge(4) ]
```

vertex.getEdges(label, ...)

Returns a list of in- or outbound edges of the vertex with given label(s).

vertex.getProperty(name)

Returns the property name a vertex.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)
```

```
arango> v.getProperty("name");
Hugo
```

`vertex.getPropertyKeys()`

Returns all propety names a vertex.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.getPropertyKeys();
[ "name" ]

arango> v.setProperty("email", "hugo@hugo.de");
"hugo@hugo.de"

arango> v.getPropertyKeys();
[ "name", "email" ]
```

`vertex.properties()` Returns all properties and their values of a vertex

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.properties();
{ name : "Hugo" }
```

`vertex.setProperty( name, value)`

Changes or sets the property name a vertex to value.

Examples

```
arango> v = g.addVertex(1, { name : "Hugo" });
Vertex(1)

arango> v.getProperty("name");
"Hugo"

arango> v.setProperty("name", "Emil");
"Emil"

arango> v.getProperty("name");
"Emil"
```

`vertex.commonNeighborsWith( target_vertex, options)`

`vertex.commonPropertiesWith( target_vertex, options)`

`vertex.pathTo( target_vertex, options)`

```
vertex.distanceTo( target_vertex, options)
```

```
vertex.determinePredecessors( source, options)
```

```
vertex.pathesForTree( tree, path_to_here)
```

```
vertex.getNeighbors( options)
```

```
vertex.measurement( measurement)
```

Calculates the eccentricity, betweenness or closeness of the vertex

Edge Methods

`edge.getId()`

Returns the identifier of the edge.

Examples

```
arango> v = g.addVertex("v");
Vertex("v")

arango> e = g.addEdge(v, v, 1, "self");
Edge(1)

arango> e.getId();
1
```

`edge.getInVertex()`

Returns the vertex at the head of the edge.

Examples

```
arango> v1 = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self");
Edge(2)

arango> e.getInVertex();
Vertex(1)
```

`edge.getLabel()`

Returns the label of the edge.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "knows");
Edge(2)

arango> e.getLabel();
knows
```

`edge.getOutVertex()`

Returns the vertex at the tail of the edge.

Examples

```
arango> v = g.addVertex(1);
```

```
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self");
Edge(2)

arango> e.getOutVertex();
Vertex(1)
```

`edge.getPeerVertex( vertex)`

Returns the peer vertex of the edge and the vertex.

Examples

```
arango> v1 = g.addVertex("1");
Vertex("1")
arango> v2 = g.addVertex("2");
Vertex("2")
arango> e = g.addEdge(v1, v2, "1->2", "knows");
Edge("1->2")
arango> e.getPeerVertex(v1);
Vertex(2)
```

`edge.getProperty( name)`

Returns the property name an edge.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self", { "weight" : 10 });
Edge(2)

arango> e.getProperty("weight");
10
```

`edge.getPropertyKeys()`

Returns all propety names an edge.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self", { weight: 10 })
Edge(2)

arango> e.getPropertyKeys()
[ "weight" ]

arango> e.setProperty("name", "Hugo");
Hugo
```

```
arango> e.getPropertyKeys()
[ "weight", "name" ]
```

`edge.properties()`

Returns all properties and their values of an edge

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "knows");
Edge(2)

arango> e.properties();
{ "weight" : 10 }
```

`edge.setProperty( name, value)`

Changes or sets the property name an edges to value.

Examples

```
arango> v = g.addVertex(1);
Vertex(1)

arango> e = g.addEdge(v, v, 2, "self", { weight: 10 })
Edge(2)

arango> e.getPropert("weight")
10

arango> e.setProperty("weight", 20);
20

arango> e.getPropert("weight")
20
```

Module "actions"

The action module provides the infrastructure for defining HTTP actions.

Basics

Error Message

`actions.getErrorMessage(code)`

Returns the error message for an error code.

Standard HTTP Result Generators

Result Ok

```
actions.resultOk(req, res, code, result, headers)
```

The function defines a response. *code* is the status code to return. *result* is the result object, which will be returned as JSON object in the body. *headers* is an array of headers to returned. The function adds the attribute *error* with value *false* and *code* with value *code* to the *result*.

Result Bad

```
actions.resultBad(req, res, error-code, msg, headers)
```

The function generates an error response. `actions.resultNotFound(req, res, code, msg, headers)`

Result Unsupported

```
actions.resultUnsupported(req, res, headers)
```

The function generates an error response.

Result Error

*actions.resultError(*req*, res*, code*, errorNum, errorMessage, headers, keyvals)**

The function generates an error response. The response body is an array with an attribute *errorMessage* containing the error message *errorMessage*, *error* containing *true*, *code* containing *code*, *errorNum* containing *errorNum*, and *errorMessage* containing the error message *errorMessage*. *keyvals* are mixed into the result.

Module "planner"

Cluster Module

This module contains functions to plan, launch and shutdown clusters of ArangoDB instances. We distinguish between the planning phase of a cluster and the startup phase. The planning involves determining how many processes in which role to run on which server, what ports and endpoints to use, as well as the sequence of events to startup the whole machinery. The result of such a planning phase is a "cluster plan". This in turn can be given to a "Kickstarter", which uses the plan to actually fire up the necessary processes. This is done via "dispatchers". A dispatcher is nothing but a regular ArangoDB instance, compiled with the cluster extensions, but not running in cluster mode. It exposes a REST API to help in the planning and running of a cluster, mainly by starting up further processes on the local machine. The planner needs a complete description of all dispatchers in the system, which basically describes the set of machines used for the cluster. These dispatchers are also used during the planning to find free network ports.

Here are the details of the functionality:

Require

new require("org/arangodb/cluster").Planner(userConfig)

This constructor builds a cluster planner object. The one and only argument is an object that can have the properties described below. The planner can plan clusters on a single machine (basically for testing purposes) and on multiple machines. The resulting "cluster plans" can be used by the kickstarter to start up the processes comprising the cluster, including the agency. To this end, there has to be one dispatcher on every machine participating in the cluster. A dispatcher is a simple instance of ArangoDB, compiled with the cluster extensions, but not running in cluster mode. This is why the configuration option *dispatchers* below is of central importance.

- *dispatchers*: an object with a property for each dispatcher, the property name is the ID of the dispatcher and the value should be an object with at least the property *endpoint* containing the endpoint of the corresponding dispatcher. Further optional properties are:

- *avoidPorts* which is an object in which all port numbers that should not be used are bound to *true*, default is empty, that is, all ports can be used
- *arangodExtraArgs*, which is a list of additional command line arguments that will be given to DBservers and coordinators started by this dispatcher, the default is an empty list. These arguments will be appended to those produced automatically, such that one can overwrite things with this.
- *allowCoordinators*, which is a boolean value indicating whether or not coordinators should be started on this dispatcher, the default is *true*
- *allowDBservers*, which is a boolean value indicating whether or not DBservers should be started on this dispatcher, the default is *true*
- *allowAgents*, which is a boolean value indicating whether or not agents should be started on this dispatcher, the default is *true*
- *username*, which is a string that contains the user name for authentication with this dispatcher
- *passwd*, which is a string that contains the password for authentication with this dispatcher, if not both *username* and *passwd* are set, then no authentication is used between dispatchers. Note that this will not work if the dispatchers are configured with authentication.

If *.dispatchers* is empty (no property), then an entry for the local arangod itself is automatically added. Note that if the only configured dispatcher has endpoint *tcp://localhost:*, all processes are started in a special "local" mode and are configured to bind their endpoints only to the localhost device. In all other cases both agents and *arangod* instances bind their endpoints to all available network devices.

- *numberOfAgents*: the number of agents in the agency, usually there is no reason to deviate from the default of 3. The planner distributes them amongst the dispatchers, if possible.
- *agencyPrefix*: a string that is used as prefix for all keys of configuration data stored in the agency.

- *numberOfDBservers*: the number of DBservers in the cluster. The planner distributes them evenly amongst the dispatchers.
- *startSecondaries*: a boolean flag indicating whether or not secondary servers are started. In this version, this flag is silently ignored, since we do not yet have secondary servers.
- *numberOfCoordinators*: the number of coordinators in the cluster, the planner distributes them evenly amongst the dispatchers.
- *DBserverIDs*: a list of DBserver IDs (strings). If the planner runs out of IDs it creates its own ones using *DBserver* concatenated with a unique number.
- *coordinatorIDs*: a list of coordinator IDs (strings). If the planner runs out of IDs it creates its own ones using *Coordinator* concatenated with a unique number.
- *dataPath*: this is a string and describes the path under which the agents, the DBservers and the coordinators store their data directories. This can either be an absolute path (in which case all machines in the clusters must use the same path), or it can be a relative path. In the latter case it is relative to the directory that is configured in the dispatcher with the *cluster.data-path* option (command line or configuration file). The directories created will be called *data-PREFIX-ID* where *PREFIX* is replaced with the agency prefix (see above) and *ID* is the ID of the DBserver or coordinator.
- *logPath*: this is a string and describes the path under which the DBservers and the coordinators store their log file. This can either be an absolute path (in which case all machines in the cluster must use the same path), or it can be a relative path. In the latter case it is relative to the directory that is configured in the dispatcher with the *cluster.log-path* option.
- *arangodPath*: this is a string and describes the path to the actual executable *arangod* that will be started for the DBservers and coordinators. If this is an absolute path, it obviously has to be the same on all machines in the cluster as described for *dataPath*. If it is an empty string, the dispatcher uses the executable that is configured with the *cluster.arangod-path* option, which is by default the same executable as the dispatcher uses.
- *agentPath*: this is a string and describes the path to the actual executable that will be started for the agents in the agency. If this is an absolute path, it obviously has to be the same on all machines in the cluster, as described for *arangodPath*. If it is an empty string, the dispatcher uses its *cluster.agent-path* option.
- *agentExtPorts*: a list of port numbers to use for the external ports of the agents. When running out of numbers in this list, the planner increments the last one used by one for every port needed. Note that the planner checks availability of the ports during the planning phase by contacting the dispatchers on the different machines, and uses only ports that are free during the planning phase. Obviously, if those ports are connected before the actual startup, things can go wrong.
- *agentIntPorts*: a list of port numbers to use for the internal ports of the agents. The same comments as for *agentExtPorts* apply.
- *DBserverPorts*: a list of port numbers to use for the DBservers. The same comments as for *agentExtPorts* apply.
- *coordinatorPorts*: a list of port numbers to use for the coordinators. The same comments as for *agentExtPorts* apply.
- *useSSLonDBservers*: a boolean flag indicating whether or not we use SSL on all DBservers in the cluster
- *useSSLonCoordinators*: a boolean flag indicating whether or not we use SSL on all coordinators in the cluster

All these values have default values. Here is the current set of default values:

```
{
"agencyPrefix"              : "arango",
"numberOfAgents"            : 1,
"numberOfDBservers"         : 2,
"startSecondaries"          : false,
"numberOfCoordinators"      : 1,
"DBserverIDs"               : ["Pavel", "Perry", "Pancho", "Paul",
"Pierre",
"Pit", "Pia", "Pablo" ],
"coordinatorIDs"            : ["Claus", "Chantalle", "Claire",
"Claudia",
```

```
  "Claas", "Clemens", "Chris" ],
  "dataPath"                     : "",    // means configured in dispatcher
  "logPath"                      : "",    // means configured in dispatcher
  "arangodPath"                  : "",    // means configured as dispatcher
  "agentPath"                    : "",    // means configured in dispatcher
  "agentExtPorts"                : [4001],
  "agentIntPorts"                : [7001],
  "DBserverPorts"                : [8629],
  "coordinatorPorts"             : [8530],
  "dispatchers"                  : {"me": {"endpoint":
  "tcp://localhost:"}},
  // this means only we as a local instance
  "useSSLonDBservers"       : false,
  "useSSLonCoordinators"    : false
  };
```

Get Plan

```
Planner.getPlan()
```

returns the cluster plan as a JavaScript object. The result of this method can be given to the constructor of a Kickstarter.

Require

```
new require("org/arangodb/cluster").Kickstarter(plan)
```

This constructor constructs a kickstarter object. Its first argument is a cluster plan as for example provided by the planner (see Cluster Planner Constructor and the general explanations before this reference). The second argument is optional and is taken to be "me" if omitted, it is the ID of the dispatcher this object should consider itself to be. If the plan contains startup commands for the dispatcher with this ID, these commands are executed immediately. Otherwise they are handed over to another responsible dispatcher via a REST call.

The resulting object of this constructors allows to launch, shutdown, relaunch the cluster described in the plan.

Launch

```
Kickstarter.launch()
```

This starts up a cluster as described in the plan which was given to the constructor. To this end, other dispatchers are contacted as necessary. All startup commands for the local dispatcher are executed immediately.

The result is an object that contains information about the started processes, this object is also stored in the Kickstarter object itself. We do not go into details here about the data structure, but the most important information are the process IDs of the started processes. The corresponding shutdown method needs this information to shut down all processes.

Note that all data in the DBservers and all log files and all agency information in the cluster is deleted by this call. This is because it is intended to set up a cluster for the first time. See the relaunch method for restarting a cluster without data loss.

Shutdown

```
Kickstarter.shutdown()
```

This shuts down a cluster as described in the plan which was given to the constructor. To this end, other

dispatchers are contacted as necessary. All processes in the cluster are gracefully shut down in the right order.

Relaunch

```
Kickstarter.relaunch()
```

This starts up a cluster as described in the plan which was given to the constructor. To this end, other dispatchers are contacted as necessary. All startup commands for the local dispatcher are executed immediately.

The result is an object that contains information about the started processes, this object is also stored in the Kickstarter object itself. We do not go into details here about the data structure, but the most important information are the process IDs of the started processes. The corresponding [shutdown method](#) needs this information to shut down all processes.

Note that this methods needs that all data in the DBservers and the agency information in the cluster are already set up properly. See the [launch method](#) for starting a cluster for the first time.

Upgrade

`Kickstarter.upgrade()

This performs an upgrade procedure on a cluster as described in the plan which was given to the constructor. To this end, other dispatchers are contacted as necessary. All commands for the local dispatcher are executed immediately. The basic approach for the upgrade is as follows: The agency is started first (exactly as in relaunch), no configuration is sent there (exactly as in the relaunch action), all servers are first started with the option "--upgrade" and then normally. In the end, the version-check.js script is run on one of the coordinators, as in the launch action.

The result is an object that contains information about the started processes, this object is also stored in the Kickstarter object itself. We do not go into details here about the data structure, but the most important information are the process IDs of the started processes. The corresponding [shutdown method](#) needs this information to shut down all processes.

Note that this methods needs that all data in the DBservers and the agency information in the cluster are already set up properly. See the [launch method](#) for starting a cluster for the first time.

Cleanup

```
Kickstarter.cleanup()
```

This cleans up all the data and logs of a previously shut down cluster. To this end, other dispatchers are contacted as necessary. [Use shutdown](#) first and use with caution, since potentially a lot of data is being erased with this call!

Task Management

Introduction to Task Management in ArangoDB

ArangoDB can execute user-defined JavaScript functions as one-shot or periodic tasks. This functionality can be used to implement timed or recurring jobs in the database.

Tasks in ArangoDB consist of a JavaScript snippet or function that is executed when the task is scheduled. A task can be a one-shot task (meaning it is run once and not repeated) or a periodic task (meaning that it is re-scheduled after each execution). Tasks can have optional parameters, which are defined at task setup time. The parameters specified at task setup time will be passed as arguments to the task whenever it gets executed. Periodic Tasks have an execution frequency that needs to be specified when the task is set up. One-shot tasks have a configurable delay after which they'll get executed.

Tasks will be executed on the server they have been set up on. Tasks will not be shipped around in a cluster. A task will be executed in the context of the database it was created in. However, when dropping a database, any tasks that were created in the context of this database will remain active. It is therefore sensible to first unregister all active tasks for a database before dropping the database.

Tasks registered in ArangoDB will be executed until the server gets shut down or restarted. After a restart of the server, any user-defined one-shot or periodic tasks will be lost.

Commands for Working with Tasks

ArangoDB provides the following commands for working with tasks. All commands can be accessed via the *tasks* module, which can be loaded like this:

```
require("org/arangodb/tasks");
```

Please note that the *tasks* module is available inside the ArangoDB server only. It cannot be used from the ArangoShell or ArangoDB's web interface.

Register a task

To register a task, the JavaScript snippet or function needs to be specified in addition to the execution frequency. Optionally, a task can have an id and a name. If no id is specified, it will be auto-assigned for a new task. The task id is also the means to access or unregister a task later. Task names are informational only. They can be used to make a task distinguishable from other tasks also running on the server.

The following server-side commands register a task. The command to be executed is a JavaScript string snippet which prints a message to the server's logfile:

```
var tasks = require("org/arangodb/tasks");
tasks.register({
  id: "mytask-1",
  name: "this is a snippet task",
  period: 15,
  command: "require('console').log('hello from snippet task');"
});
```

The above has register a task with id *mytask-1*, which will be executed every 15 seconds on the server. The task will write a log message whenever it is invoked.

Tasks can also be set up using a JavaScript callback function like this:

```
var tasks = require("org/arangodb/tasks");
tasks.register({
  id: "mytask-2",
  name: "this is a function task",
  period: 15,
  command: function () {
    require('console').log('hello from function task');
  }
});
```

It is important to note that the callback function is late bound and will be executed in a different context than in the creation context. The callback function must therefore not access any variables defined outside of its own scope. The callback function can still define and use its own variables.

To pass parameters to a task, the *params* attribute can be set when registering a task. Note that the parameters are limited to datatypes usable in JSON (meaning no callback functions can be passed as parameters into a task):

```
var tasks = require("org/arangodb/tasks");
tasks.register({
  id: "mytask-3",
  name: "this is a parameter task",
  period: 15,
  command: function (params) {
    var greeting = params.greeting;
    var data = JSON.stringify(params.data);
    require('console').log('%s from parameter task: %s', greeting,
data);
  },
  params: { greeting: "hi", data: "how are you?" }
});
```

Registering a one-shot task works the same way, except that the *period* attribute must be omitted. If *period* is omitted, then the task will be executed just once. The task invocation delay can optionally be specified with the *offset* attribute:

```
var tasks = require("org/arangodb/tasks");
tasks.register({
  id: "mytask-once",
  name: "this is a one-shot task",
  offset: 10,
  command: function (params) {
    require('console').log('you will see me just once!');
  }
});
```

Unregister a task

After a task has been registered, it can be unregistered using its id:

```
var tasks = require("org/arangodb/tasks");
tasks.unregister("mytask-1");
```

Note that unregistering a non-existing task will throw an exception.

List all tasks

To get an overview of which tasks are registered, there is the *get* method. If the *get* method is called without any arguments, it will return a list of all tasks:

```
var tasks = require("org/arangodb/tasks");
tasks.get();
```

If *get* is called with a task id argument, it will return information about this particular task:

```
var tasks = require("org/arangodb/tasks");
tasks.get("mytask-3");
```

The *created* attribute of a task reveals when a task was created. It is returned as a Unix timestamp.

Using jsUnity and node-jscoverage

jsUnity

The ArangoDB contains a wrapper for [jsUnity](#), a lightweight universal JavaScript unit testing framework.

Running jsUnity Tests

Assume that you have a test file containing

```
function aqlTestSuite () {
  return {
testSizeOfTestCollection : function () {
  assertEqual(5, 5);
  };
}

jsUnity.run(aqlTestSuite);

return jsunity.done();
```

Then you can run the test suite using *jsunity.runTest*

```
unix> ju.runTest("test.js");
2012-01-28T19:10:23Z [10671] INFO Running aqlTestSuite
2012-01-28T19:10:23Z [10671] INFO 1 test found
2012-01-28T19:10:23Z [10671] INFO [PASSED] testSizeOfTestCollection
2012-01-28T19:10:23Z [10671] INFO 1 test passed
2012-01-28T19:10:23Z [10671] INFO 0 tests failed
2012-01-28T19:10:23Z [10671] INFO 1 millisecond elapsed
```

Administrating ArangoDB

Mostly Memory/Durability

Database documents are stored in memory-mapped files. Per default, these memory-mapped files are synced regularly but not instantly. This is often a good tradeoff between storage performance and durability. If this level of durability is too low for an application, the server can also sync all modifications to disk instantly. This will give full durability but will come with a performance penalty as each data modification will trigger a sync I/O operation.

AppendOnly/MVCC

Instead of overwriting existing documents, a completely new version of the document is generated. The two benefits are:

- Objects can be stored coherently and compactly in the main memory.
- Objects are preserved-isolated writing and reading transactions allow accessing these objects for parallel operations.

The system collects obsolete versions as garbage, recognizing them as forsaken. Garbage collection is asynchronous and runs parallel to other processes.

Configuration

Global Configuration

There are certain default values, which you can store in the configuration file or supply on the command line.

Maximal Journal Size

```
--database.maximal-journal-size size
```

Maximal size of journal in bytes. Can be overwritten when creating a new collection. Note that this also limits the maximal size of a single document.

The default is *32MB*.

Per Collection Configuration

You can configure the durability behavior on a per collection basis. Use the ArangoDB shell to change these properties.

Properties

`collection.properties()`

Returns an object containing all collection properties.

- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.

- *journalSize* : The size of the journal in bytes.

- *isVolatile*: If *true* then the collection data will be kept in memory only and ArangoDB will not write or sync the data to disk.

- *keyOptions* (optional) additional options for key generation. This is a JSON array containing the following attributes (note: some of the attributes are optional):
    - *type*: the type of the key generator used for the collection.
    - *allowUserKeys*: if set to *true*, then it is allowed to supply own key values in the _key attribute of a document. If set to *false*, then the key generator will solely be responsible for generating keys and supplying own key values in the _key attribute of documents is considered an error.
    - *increment*: increment value for *autoincrement* key generator. Not used for other key generator types.
    - *offset*: initial offset value for *autoincrement* key generator. Not used for other key generator types.

In a cluster setup, the result will also contain the following attributes:

- *numberOfShards*: the number of shards of the collection.

- *shardKeys*: contains the names of document attributes that are used to determine the target shard for documents.

`collection.properties(properties)`

Changes the collection properties. *properties* must be a object with one or more of the following attribute(s):

- *waitForSync*: If *true* creating a document will only return after the data was synced to disk.

- *journalSize* : The size of the journal in bytes.

Note: it is not possible to change the journal size after the journal or datafile has been created. Changing this parameter will only effect newly created journals. Also note that you cannot lower the journal size to less then size of the largest document already stored in the collection.

Note: some other collection properties, such as *type*, *isVolatile*, or *keyOptions* cannot be changed once the collection is created.

Examples

Read all properties

```
arangosh> db.example.properties();
{
  "doCompact" : true,
```

```
    "journalSize" : 1048576,
    "isSystem" : false,
    "isVolatile" : false,
    "waitForSync" : false,
    "keyOptions" : {
      "type" : "traditional",
      "allowUserKeys" : true
    }
  }
}
```

Change a property

```
arangosh> db.example.properties({ waitForSync : true });
{
  "doCompact" : true,
  "journalSize" : 1048576,
  "isSystem" : false,
  "isVolatile" : false,
  "waitForSync" : true,
  "keyOptions" : {
    "type" : "traditional",
    "allowUserKeys" : true
  }
}
```

Handling Indexes

Indexes, Identifiers, Handles

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

- cap constraints
- geo-spatial indexes
- hash indexes
- skip-lists

Index

Indexes are used to allow fast access to documents. For each collection there is always the primary index which is a hash index for the document key (_key attribute). This index cannot be dropped or changed. Edge collections will also have an automatically created edges index, which cannot be modified. This index provides quick access to documents via the _from and _to attributes.

Most user-land indexes can be created by defining the names of the attributes which should be indexed. Some index types allow indexing just one attribute (e.g. fulltext index) whereas other index types allow indexing multiple attributes.

Indexing system attributes such as _id, _key, _from, and _to in user-defined indexes is not supported by any index type. Manually creating an index that relies on any of these attributes is unsupported.

Index Handle

An index handle uniquely identifies an index in the database. It is a string and consists of a collection name and an index identifier separated by /.

Geo Index

A geo index is used to find places on the surface of the earth fast.

Hash Index

A hash index is used to find documents based on examples. A hash index can be created for one or multiple document attributes. A hash index will only be used by queries if all indexed attributes are present in the example or search query, and if all attributes are compared using the equality (== operator). That means the hash index does not support range queries.

If the index is declared unique, then access to the indexed attributes should be fast. The performance degrades if the indexed attribute(s) contain(s) only very few distinct values.

Edges Index

An edges index is automatically created for edge collections. It contains connections between vertex documents and is invoked when the connecting edges of a vertex are queried. There is no way to explicitly create or delete edge indexes.

Skiplist Index

A skiplist is used to find ranges of documents.

Fulltext Index

A fulltext index can be used to find words, or prefixes of words inside documents. A fulltext index can be set on one attribute only, and will index all words contained in documents that have a textual value in this attribute. Only words with a (specifyable) minimum length are indexed. Word tokenisation is done using the word boundary analysis provided by libicu, which is taking into account the selected language provided at server start. Words are indexed in their lower-cased form. The index supports complete match queries (full words) and prefix queries.

Address and ETag of an Index

All indexes in ArangoDB have an index handle. This handle uniquely defines an index and is managed by ArangoDB. The interface allows you to access the indexes of a collection as:

```
db.collection.index(index-handle)
```

For example: Assume that the index handle, which is stored in the *_id* attribute of the index, is *demo/362549736* and the index lives in a collection named *demo*, then that index can be accessed as:

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

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

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

Which Index type to use when

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

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

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

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

  The hash index is a good candidate if all or most queries on the indexed attribute(s) are equality comparisons, and if the attribute selectivity is high. That means the number of distinct attribute values in relation to the total number of documents should be high. This is the case for indexes declared *unique*.

  The hash index should not be used if duplicate index values are allowed (i.e. if the hash index is not declared *unique*) and it cannot be avoided that there will be many duplicate index values. For example, it should be avoided to use a hash index on an attribute with just 10 distinct values in a collection with a million documents.

- skip list index: skip lists keep the indexed values in an order, so they can be used for equality and range queries. Skip list indexes will have a slightly higher overhead than hash indexes in case but they are more general and allow more use cases (e.g. range queries). Additionally, they can be used for lower selectivity attributes, when non-unique hash indexes are not a good fit.

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

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

  or

  { "coords": [50.9406645, 6.9599115] }

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

  The index supports complete match queries (full words) and prefix queries.

- cap constraint: the cap constraint provided by ArangoDB indexes documents not to speed up search queries, but to limit (cap) the number or size of documents in a collection.

Currently it is not possible to index system attributes in user-defined indexes.

Working with Indexes

Collection Methods

`collection.index( index-handle)`

Returns the index with index-handle or null if no such index exists.

Examples

```
arango> db.example.getIndexes().map(function(x) { return x.id; });
["example/0"]
arango> db.example.index("93013/0");
{ "id" : "example/0", "type" : "primary", "fields" : ["_id"] }
```

returns information about the indexes

`getIndexes()`

Returns a list of all indexes defined for the collection.

Examples

```
arango> db.demo.getIndexes()
[
  {
    "id" : "demo/0",
    "type" : "primary",
    "fields" : [ "_id" ]
  },
  {
    "id" : "demo/2290971",
    "unique" : true,
    "type" : "hash",
    "fields" : [ "a" ]
  },
  {
    "id" : "demo/2946331",
    "unique" : false,
    "type" : "hash",
    "fields" : [ "b" ]
  },
  {
    "id" : "demo/3077403",
    "unique" : false,
    "type" : "skiplist",
    "fields" : [ "c" ]
  }
]
```

drops an index

`collection.dropIndex( index)`

Drops the index. If the index does not exist, then false is returned. If the index existed and was dropped, then true is returned. Note that you cannot drop some special indexes (e.g. the primary index of a collection or the edge index of an edge collection).

```
collection.dropIndex( index-handle)
```

Same as above. Instead of an index an index handle can be given.

Examples

```
arango> db.example.ensureSkiplist("a", "b");
{ "id" : "example/991154", "unique" : false, "type" : "skiplist",
"fields" : ["a", "b"], "isNewlyCreated" : true }

arango> i = db.example.getIndexes();
[
  { "id" : "example/0", "type" : "primary", "fields" : ["_id"] },
  { "id" : "example/991154", "unique" : false, "type" : "skiplist",
"fields" : ["a", "b"] }
  ]

arango> db.example.dropIndex(i[0])
false

arango> db.example.dropIndex(i[1].id)
true

arango> i = db.example.getIndexes();
[{ "id" : "example/0", "type" : "primary", "fields" : ["_id"] }]
```

```
collection.ensureIndex( index-description)
```

Ensures that an index according to the index-description exists. A new index will be created if none exists with the given description.

The index-description must contain at least a type attribute. type can be one of the following values:

- hash: hash index
- skiplist: skiplist index
- fulltext: fulltext index
- bitarray: bitarray index
- geo1: geo index, with one attribute
- geo2: geo index, with two attributes
- cap: cap constraint

Other attributes may be necessary, depending on the index type.

Calling this method returns an index object. Whether or not the index object existed before the call is indicated in the return attribute isNewlyCreated.

Examples

```
arango> db.example.ensureIndex({ type: "hash", fields: [ "name" ],
unique: true });
{
"id" : "example/30242599562",
"type" : "hash",
"unique" : true,
"fields" : [
"name"
],
"isNewlyCreated" : true
```

```
    }
```

Database Methods

`db._index(index-handle)`

Returns the index with index-handle or null if no such index exists.

Examples

```
arango> db.example.getIndexes().map(function(x) { return x.id; });
["example/0"]
arango> db._index("example/0");
{ "id" : "example/0", "type" : "primary", "fields" : ["_id"] }
```

`db._dropIndex(index)`

Drops the index. If the index does not exists, then false is returned. If the index existed and was dropped, then true is returned. Note that you cannot drop the primary index.

`db._dropIndex(index-handle)`

Drops the index with index-handle.

Examples

```
arango> db.example.ensureSkiplist("a", "b");
{ "id" : "example/1577138", "unique" : false, "type" : "skiplist",
"fields" : ["a", "b"], "isNewlyCreated" : true }

arango> i = db.example.getIndexes();
[{ "id" : "example/0", "type" : "primary", "fields" : ["_id"] },
 { "id" : "example/1577138", "unique" : false, "type" : "skiplist",
"fields" : ["a", "b"] }]

 arango> db._dropIndex(i[0]);
 false

 arango> db._dropIndex(i[1].id);
true

arango> i = db.example.getIndexes();
[{ "id" : "example/0", "type" : "primary", "fields" : ["_id"] }]
```

Cap Constraint

Introduction to Cap Constraints

This is an introduction to ArangoDB's size restrictions aka cap constraints for collections.

It is possible to restrict the size of collections. If you add a document and the size exceeds the limit, then the least recently created or updated document(s) will be dropped. The size of a collection is measured in the number of active documents a collection contains, and optionally in the total size of the active documents' data in bytes.

It is possible to only restrict the number of documents in a collection, or to only restrict the total active data size, or both at the same time. If there are restrictions on both document count and total size, then the first violated constraint will trigger the auto-deletion of "too" old documents until all constraints are satisfied.

Using a cap constraint, a collection can be used as a FIFO container, with just the newest documents remaining in the collection.

For example, a cap constraint can be used to keep a list of just the most recent log entries, and at the same time ensure that the collection does not grow indefinitely. Cap constraints can be used to automate the process of getting rid of "old" documents, and so save the user from implementing own jobs to purge "old" collection data.

Accessing Cap Constraints from the Shell

```
collection.ensureCapConstraint( size, {byteSize})
```

Creates a size restriction aka cap for the collection of size documents and/or byteSize data size. If the restriction is in place and the (size plus one) document is added to the collection, or the total active data size in the collection exceeds byteSize, then the least recently created or updated documents are removed until all constraints are satisfied.

It is allowed to specify either size or byteSize, or both at the same time. If both are specified, then the automatic document removal will be triggered by the first non-met constraint.

Note that at most one cap constraint is allowed per collection. Trying to create additional cap constraints will result in an error. Creating cap constraints is also not supported in sharded collections with more than one shard.

Note that this does not imply any restriction of the number of revisions of documents.

Examples

Restrict the number of document to at most 10 documents:

```
arango> db.examples.ensureCapConstraint(10);
{ "id" : "examples/934311", "type" : "cap", "size" : 10, "byteSize"
: 0, "isNewlyCreated" : true }

arango> for (var i = 0;  i < 20;  ++i) { var d = db.examples.save(
{ n : i } ); }

arango> db.examples.count();
10
```

Geo Indexes

Introduction to Geo Indexes

This is an introduction to ArangoDB's geo indexes.

ArangoDB uses Hilbert curves to implement geo-spatial indexes. See this [blog](#) for details.

A geo-spatial index assumes that the latitude is between -90 and 90 degree and the longitude is between -180 and 180 degree. A geo index will ignore all documents which do not fulfill these requirements.

A geo-spatial constraint makes the same assumptions, but documents not fulfilling these requirements are rejected.

Accessing Geo Indexes from the Shell

```
collection.ensureGeoIndex( location)
```

Creates a geo-spatial index on all documents using location as path to the coordinates. The value of the attribute must be a list with at least two double values. The list must contain the latitude (first value) and the longitude (second value). All documents, which do not have the attribute path or with value that are not suitable, are ignored.

In case that the index was successfully created, the index identifier is returned.

```
collection.ensureGeoIndex( location, true)
```

As above which the exception, that the order within the list is longitude followed by latitude. This corresponds to the format described in

http://geojson.org/geojson-spec.html

```
collection.ensureGeoIndex( latitude, longitude)
```

Creates a geo-spatial index on all documents using latitude and longitude as paths the latitude and the longitude. The value of the attribute latitude and of the attribute longitude must a double. All documents, which do not have the attribute paths or which values are not suitable, are ignored.

In case that the index was successfully created, the index identifier is returned.

Examples

Create an geo index for a list attribute:

```
arango> db.geo.ensureGeoIndex("loc");
{ "id" : "geo/47772301", "type" : "geo1", "geoJson" : false,
"fields" : ["loc"], "isNewlyCreated" : true }

arango> for (i = -90;  i <= 90;  i += 10) {
.......>    for (j = -180; j <= 180; j += 10) {
.......>      db.geo.save({ name : "Name/" + i + "/" + j,
.......>                  loc: [ i, j ] });
.......>    }
.......> }

arango> db.geo.count();
703

arango> db.geo.near(0,0).limit(3).toArray();
[ { "_id" : "geo/24861164", "_key" : "24861164", "_rev" :
"24861164", "name" : "Name/0/0", "loc" : [ 0, 0 ]},
  { "_id" : "geo/24926700", "_key" : "24926700", "_rev" :
"24926700", "name" : "Name/0/10", "loc" : [ 0, 10 ]},
  { "_id" : "geo/22436332", "_key" : "22436332", "_rev" :
"22436332", "name" : "Name/-10/0", "loc" : [ -10, 0 ]}]

arango> db.geo.near(0,0).count();
100
```

Create an geo index for a hash array attribute:

```
arango> db.geo2.ensureGeoIndex("location.latitude",
```

```
 "location.longitude");
 { "id" : "geo2/1070652", "type" : "geo2", "fields" :
 ["location.latitude", "location.longitude"], "isNewlyCreated" :
 true }

arango> for (i = -90;  i <= 90;  i += 10) {
.......>    for (j = -180; j <= 180; j += 10) {
.......>       db.geo2.save({ name : "Name/" + i + "/" + j,
.......>                    location: { latitude : i,
.......>                                longitude : j } });
.......>    }
.......> }

arango> db.geo2.near(0,0).limit(3).toArray();
[
   {
     "_id" : "geo2/72964588",
     "_key" : "72964588",
     "_rev" : "72964588",
     "location" : { "latitude" : 0, "longitude" : 0 },
     "name" : "Name/0/0"
   },
   {
     "_id" : "geo2/73030124",
     "_key" : "73030124",
     "_rev" : "73030124",
     "location" : { "latitude" : 0, "longitude" : 10 },
     "name" : "Name/0/10"
   },
   {
     "_id" : "geo2/70539756",
     "_key" : "70539756",
     "_rev" : "70539756",
     "location" : { "latitude" : -10, "longitude" : 0 },
     "name" : "Name/-10/0"
   }
]
```

```
collection.ensureGeoConstraint( location, ignore-null)
```

```
collection.ensureGeoConstraint( location, true, ignore-null)
```

```
collection.ensureGeoConstraint( latitude, longitude, ignore-null)
```

Works like ensureGeoIndex but requires that the documents contain a valid geo definition. If ignore-null is true, then documents with a null in location or at least one null in latitude or longitude are ignored.

```
collection.geo( location-attribute)
```

Looks up a geo index defined on attribute location-attribute.

Returns a geo index object if an index was found. The near or within operators can then be used to execute a geo-spatial query on this particular index.

This is useful for collections with multiple defined geo indexes.

```
collection.geo( location-attribute, true)
```

Looks up a geo index on a compound attribute location-attribute.

Returns a geo index object if an index was found. The near or within operators can then be used to execute a geo-spatial query on this particular index.

```
collection.geo( latitude-attribute, longitude-attribute)
```

Looks up a geo index defined on the two attributes latitude-attribute and longitude-attribute.

Returns a geo index object if an index was found. The near or within operators can then be used to execute a geo-spatial query on this particular index.

Examples

Assume you have a location stored as list in the attribute home and a destination stored in the attribute work. Then you can use the geo operator to select which geo-spatial attributes (and thus which index) to use in a near query.

```
arango> for (i = -90;  i <= 90;  i += 10) {
.......>    for (j = -180;  j <= 180;  j += 10) {
.......>      db.complex.save({ name : "Name/" + i + "/" + j,
.......>                        home : [ i, j ],
.......>                        work : [ -i, -j ] });
.......>    }
.......> }

arango> db.complex.near(0, 170).limit(5);
exception in file '/simple-query' at 1018,5: a geo-index must be
known

arango> db.complex.ensureGeoIndex(""home"");
arango> db.complex.near(0, 170).limit(5).toArray();
[ { "_id" : "complex/74655276", "_key" : "74655276", "_rev" :
"74655276", "name" : "Name/0/170", "home" : [ 0, 170 ], "work" : [
0, -170 ] },
  { "_id" : "complex/74720812", "_key" : "74720812", "_rev" :
"74720812", "name" : "Name/0/180", "home" : [ 0, 180 ], "work" : [
0, -180 ] },
  { "_id" : "complex/77080108", "_key" : "77080108", "_rev" :
"77080108", "name" : "Name/10/170", "home" : [ 10, 170 ], "work" :
[ -10, -170 ] },
  { "_id" : "complex/72230444", "_key" : "72230444", "_rev" :
"72230444", "name" : "Name/-10/170", "home" : [ -10, 170 ], "work"
: [ 10, -170 ] },
  { "_id" : "complex/72361516", "_key" : "72361516", "_rev" :
"72361516", "name" : "Name/0/-180", "home" : [ 0, -180 ], "work" :
[ 0, 180 ] } ]

arango> db.complex.geo("work").near(0, 170).limit(5);
exception in file '/simple-query' at 1018,5: a geo-index must be
known

arango> db.complex.ensureGeoIndex("work");
arango> db.complex.geo("work").near(0, 170).limit(5).toArray();
[ { "_id" : "complex/72427052", "_key" : "72427052", "_rev" :
"72427052", "name" : "Name/0/-170", "home" : [ 0, -170 ], "work" :
[ 0, 170 ] },
  { "_id" : "complex/72361516", "_key" : "72361516", "_rev" :
"72361516", "name" : "Name/0/-180", "home" : [ 0, -180 ], "work" :
[ 0, 180 ] },
  { "_id" : "complex/70002220", "_key" : "70002220", "_rev" :
```

```
  "70002220", "name" : "Name/-10/-170", "home" : [ -10, -170 ],
  "work" : [ 10, 170 ] },
    { "_id" : "complex/74851884", "_key" : "74851884", "_rev" :
  "74851884", "name" : "Name/10/-170", "home" : [ 10, -170 ], "work"
  : [ -10, 170 ] },
    { "_id" : "complex/74720812", "_key" : "74720812", "_rev" :
  "74720812", "name" : "Name/0/180", "home" : [ 0, 180 ], "work" : [
  0, -180 ] } ]
```

`collection.near( latitude, longitude)`

The returned list is sorted according to the distance, with the nearest document to the coordinate (latitude, longitude) coming first. If there are near documents of equal distance, documents are chosen randomly from this set until the limit is reached. It is possible to change the limit using the limit operator.

In order to use the near operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the geo operator to select a particular index.

Note near does not support negative skips. However, you can still use limit followed to skip.

`collection.near( latitude, longitude).limit( limit)`

Limits the result to limit documents instead of the default 100.

`Note`

Unlike with multiple explicit limits, limit will raise the implicit default limit imposed by within. collection.near(latitude, longitude).distance() This will add an attribute distance to all documents returned, which contains the distance between the given point and the document in meter.

`collection.near( latitude, longitude).distance( name)`

This will add an attribute name to all documents returned, which contains the distance between the given point and the document in meter.

Examples

To get the nearst two locations:

```
arango> db.geo.near(0,0).limit(2).toArray();
[ { "_id" : "geo/24773376", "_key" : "24773376", "_rev" :
"24773376", "name" : "Name/0/0", "loc" : [ 0, 0 ] },
  { "_id" : "geo/22348544", "_key" : "22348544", "_rev" :
"22348544", "name" : "Name/-10/0", "loc" : [ -10, 0 ] } ]
```

If you need the distance as well, then you can use the distance operator:

```
arango> db.geo.near(0,0).distance().limit(2).toArray();
[
  { "_id" : geo/24773376", "_key" : "24773376", "_rev" :
"24773376", "distance" : 0, "name" : "Name/0/0", "loc" : [ 0, 0 ]
},
  { "_id" : geo/22348544", "_key" : "22348544", "_rev" :
"22348544", "distance" : 1111949.3, "name" : "Name/-10/0", "loc" :
[ -10, 0 ] }
  ]
```

```
collection.within( latitude, longitude, radius)
```

This will find all documents within a given radius around the coordinate (latitude, longitude). The returned list is sorted by distance, beginning with the nearest document.

In order to use the within operator, a geo index must be defined for the collection. This index also defines which attribute holds the coordinates for the document. If you have more then one geo-spatial index, you can use the geo operator to select a particular index.

```
collection.within( latitude, longitude, radius).distance()
```

This will add an attribute _distance to all documents returned, which contains the distance between the given point and the document in meter.

```
collection.within( latitude, longitude, radius).distance( name)
```

This will add an attribute name to all documents returned, which contains the distance between the given point and the document in meter.

Examples

To find all documents within a radius of 2000 km use:

```
arango> db.geo.within(0, 0, 2000 * 1000).distance().toArray();
[ { "_id" : "geo/24773376", "_key" : "24773376", "_rev" :
"24773376", "distance" : 0, "name" : "Name/0/0", "loc" : [ 0, 0 ]
},
  { "_id" : "geo/24707840", "_key" : "24707840", "_rev" :
"24707840", "distance" : 1111949.3, "name" : "Name/0/-10", "loc" :
[ 0, -10 ] },
  { "_id" : "geo/24838912", "_key" : "24838912", "_rev" :
"24838912", "distance" : 1111949.3, "name" : "Name/0/10", "loc" : [
0, 10 ] },
  { "_id" : "geo/22348544", "_key" : "22348544", "_rev" :
"22348544", "distance" : 1111949.3, "name" : "Name/-10/0", "loc" :
[ -10, 0 ] },
  { "_id" : "geo/27198208", "_key" : "27198208", "_rev" :
"27198208", "distance" : 1111949.3, "name" : "Name/10/0", "loc" : [
10, 0 ] },
  { "_id" : "geo/22414080", "_key" : "22414080", "_rev" :
"22414080", "distance" : 1568520.6, "name" : "Name/-10/10", "loc" :
[ -10, 10 ] },
  { "_id" : "geo/27263744", "_key" : "27263744", "_rev" :
"27263744", "distance" : 1568520.6, "name" : "Name/10/10", "loc" :
[ 10, 10 ] },
  { "_id" : "geo/22283008", "_key" : "22283008", "_rev" :
"22283008", "distance" : 1568520.6, "name" : "Name/-10/-10", "loc"
: [ -10, -10 ] },
  { "_id" : "geo/27132672", "_key" : "27132672", "_rev" :
"27132672", "distance" : 1568520.6, "name" : "Name/10/-10", "loc" :
[ 10, -10 ] } ]
```

Fulltext indexes

Introduction to Fulltext Indexes

This is an introduction to ArangoDB's fulltext indexes.

It is possible to define a fulltext index on one textual attribute of a collection of documents. The fulltext index can then be used to efficiently find exact words or prefixes of words contained in these documents.

Accessing Fulltext Indexes from the Shell

`ensureFulltextIndex(field, minWordLength)`

Creates a fulltext index on all documents on attribute field. All documents, which do not have the attribute field or that have a non-textual value inside their field attribute are ignored.

The minimum length of words that are indexed can be specified with the minWordLength parameter. Words shorter than minWordLength characters will not be indexed. minWordLength has a default value of 2, but this value might be changed in future versions of ArangoDB. It is thus recommended to explicitly specify this value

In case that the index was successfully created, the index identifier is returned.

```
arangod> db.emails.ensureFulltextIndex("body");
{ "id" : "emails/42725508", "unique" : false, "type" : "fulltext",
"fields" : ["body"], "isNewlyCreated" : true }
```

Hash Indexes

Introduction to Hash Indexes

This is an introduction to ArangoDB's hash indexes.

It is possible to define a hash index on one or more attributes (or paths) of a document. This hash index is then used in queries to locate documents in O(1) operations. If the hash is unique, then no two documents are allowed to have the same set of attribute values.

Accessing Hash Indexes from the Shell

`ensureUniqueConstraint(field1, field2, ..., fieldn)`

Creates a unique hash index on all documents using field1, field2, ... as attribute paths. At least one attribute path must be given.

When a unique constraint is in effect for a collection, then all documents which contain the given attributes must differ in the attribute values. Creating a new document or updating a document will fail, if the uniqueness is violated. If any attribute value is null for a document, this document is ignored by the index.

Note that non-existing attribute paths in a document are treated as if the value were null.

In case that the index was successfully created, the index identifier is returned.

Examples

```
arango> db.four.ensureUniqueConstraint("a", "b.c");
{ "id" : "four/1147445", "unique" : true, "type" : "hash", "fields"
: ["a", "b.c"], "isNewlyCreated" : true }

arango> db.four.save({ a : 1, b : { c : 1 } });
{ "_id" : "four/1868341", "_key" : "1868341", "_rev" : "1868341" }

arango> db.four.save({ a : 1, b : { c : 1 } });
JavaScript exception in file '(arango)' at 1,9: [ArangoError 1210:
cannot save document]
!db.four.save({ a : 1, b : { c : 1 } });
!       ^
stacktrace: [ArangoError 1210: cannot save document]
at (arango):1:9

arango> db.four.save({ a : 1, b : { c : null } });
{ "_id" : "four/2196021", "_key" : "2196021", "_rev" : "2196021" }

arango> db.four.save({ a : 1 });
{ "_id" : "four/2196023", "_key" : "2196023", "_rev" : "2196023" }
```

`ensureHashIndex(field1, field2, ..., fieldn)`

Creates a non-unique hash index on all documents using field1, field2, ... as attribute paths. At least one attribute path must be given.

Note that non-existing attribute paths in a document are treated as if the value were null.

In case that the index was successfully created, the index identifier is returned.

Examples

```
arango> db.test.ensureHashIndex("a");
{ "id" : "test/5922391", "unique" : false, "type" : "hash",
"fields" : ["a"], "isNewlyCreated" : true }

arango> db.test.save({ a : 1 });
{ "_id" : "test/6381143", "_key" : "6381143", "_rev" : "6381143" }

arango> db.test.save({ a : 1 });
```

```
{ "_id" : "test/6446679", "_key" : "6446679", "_rev" : "6446679" }

arango> db.test.save({ a : null });
{ "_id" : "test/6708823", "_key" : "6708823", "_rev" : "6708823" }
```

Skip-Lists

Introduction to Skiplist Indexes

This is an introduction to ArangoDB's skip-lists.

It is possible to define a skip-list index on one or more attributes (or paths) of a documents. This skip-list is then used in queries to locate documents within a given range. If the skip-list is unique, then no two documents are allowed to have the same set of attribute values.

Accessing Skip-List Indexes from the Shell

`ensureUniqueSkiplist(field1, field2, ..., fieldn)`

Creates a skiplist index on all documents using attributes as paths to the fields. At least one attribute must be given. All documents, which do not have the attribute path or with one or more values that are not suitable, are ignored.

In case that the index was successfully created, the index identifier is returned.

```
arangod> db.ids.ensureUniqueSkiplist("myId");
{ "id" : "ids/42612360", "unique" : true, "type" : "skiplist",
"fields" : ["myId"], "isNewlyCreated" : true }

arangod> db.ids.save({ "myId": 123 });
{ "_id" : "ids/42743432", "_key" : "42743432", "_rev" : "42743432"
}
arangod> db.ids.save({ "myId": 456 });
{ "_id" : "ids/42808968", "_key" : "42808968", "_rev" : "42808968"
}
arangod> db.ids.save({ "myId": 789 });
{ "_id" : "ids/42874504", "_key" : "42874504", "_rev" : "42874504"
}

arangod> db.ids.save({ "myId": 123 });
JavaScript exception in file '(arango)' at 1,8: [ArangoError 1210:
cannot save document: unique constraint violated]
!db.ids.save({ "myId": 123 });
!       ^
stacktrace: [ArangoError 1210: cannot save document: unique
constraint violated]
at (arango):1:8


arangod> db.ids.ensureUniqueSkiplist("name.first", "name.last");
{ "id" : "ids/43362549", "unique" : true, "type" : "skiplist",
"fields" : ["name.first", "name.last"], "isNewlyCreated" : true }

arangod> db.ids.save({ "name" : { "first" : "hans", "last":
"hansen" }});
{ "_id" : "ids/43755765", "_rev" : "43755765", "_key" : "43755765"
}
arangod> db.ids.save({ "name" : { "first" : "jens", "last":
"jensen" }});
{ "_id" : "ids/43821301", "_rev" : "43821301", "_key" : "43821301"
}
arangod> db.ids.save({ "name" : { "first" : "hans", "last":
"jensen" }});
{ "_id" : "ids/43886837", "_rev" : "43886837", "_key" : "43886837"
}

arangod> db.ids.save({ "name" : { "first" : "hans", "last":
"hansen" }});
JavaScript exception in file '(arango)' at 1,8: [ArangoError 1210:
cannot save document: unique constraint violated]
!db.ids.save({"name" : {"first" : "hans", "last": "hansen" }});
!       ^
stacktrace: [ArangoError 1210: cannot save document: unique
constraint violated]
```

```
at (arango):1:8
```

ensureSkiplist(field1, field2, ..., fieldn)

Creates a multi skiplist index on all documents using attributes as paths to the fields. At least one attribute must be given. All documents, which do not have the attribute path or with one or more values that are not suitable, are ignored.

In case that the index was successfully created, the index identifier is returned.

```
arangod> db.names.ensureSkiplist("first");
{ "id" : "names/42725508", "unique" : false, "type" : "skiplist",
"fields" : ["first"], "isNewlyCreated" : true }

arangod> db.names.save({ "first" : "Tim" });
{ "_id" : "names/42856580", "_key" : "42856580", "_rev" :
"42856580" }
arangod> db.names.save({ "first" : "Tom" });
{ "_id" : "names/42922116", "_key" : "42922116", "_rev" :
"42922116" }
arangod> db.names.save({ "first" : "John" });
{ "_id" : "names/42987652", "_key" : "42987652", "_rev" :
"42987652" }
arangod> db.names.save({ "first" : "Tim" });
{ "_id" : "names/43053188", "_key" : "43053188", "_rev" :
"43053188" }
arangod> db.names.save({ "first" : "Tom" });
{ "_id" : "names/43118724", "_key" : "43118724", "_rev" :
"43118724" }
```

BitArray Indexes

Introduction to Bit-Array Indexes

It is possible to define a bit-array index on one or more attributes (or paths) of a documents.

Accessing BitArray Indexes from the Shell

```
collection.ensureBitarray( field1, value1, ..., fieldn, valuen)
```

Creates a bitarray index on documents using attributes as paths to the fields (field1,..., fieldn). A value (value1,..., valuen) consists of an array of possible values that the field can take. At least one field and one set of possible values must be given.

All documents, which do not have all of the attribute paths are ignored (that is, are not part of the bitarray index, they are however stored within the collection). A document which contains all of the attribute paths yet has one or more values which are not part of the defined range of values will be rejected and the document will not inserted within the collection. Note that, if a bitarray index is created subsequent to any documents inserted in the given collection, then the creation of the index will fail if one or more documents are rejected (due to attribute values being outside the designated range).

In case that the index was successfully created, the index identifier is returned.

In the example below we create a bitarray index with one field and that field can have the values of either 0 or 1. Any document which has the attribute x defined and does not have a value of 0 or 1 will be rejected and therefore not inserted within the collection. Documents without the attribute x defined will not take part in the index.

```
arango> arangod> db.example.ensureBitarray("x", [0,1]);
{
"id" : "2755894/3607862",
"unique" : false,
"type" : "bitarray",
"fields" : [["x", [0, 1]]],
"undefined" : false,
"isNewlyCreated" : true
}
```

In the example below we create a bitarray index with one field and that field can have the values of either 0, 1 or other (indicated by []). Any document which has the attribute x defined will take part in the index. Documents without the attribute x defined will not take part in the index.

```
arangod> db.example.ensureBitarray("x", [0,1,[]]);
{
"id" : "2755894/4263222",
"unique" : false,
"type" : "bitarray",
"fields" : [["x", [0, 1, [ ]]]],
"undefined" : false,
"isNewlyCreated" : true
}
```

In the example below we create a bitarray index with two fields. Field x can have the values of either 0 or 1; while field y can have the values of 2 or "a". A document which does not have both attributes x and y will not take part within the index. A document which does have both attributes x and y defined must have the values 0 or 1 for attribute x and 2 or a for attribute y, otherwise the document will not be

inserted within the collection.

```
arangod> db.example.ensureBitarray("x", [0,1], "y", [2,"a"]);
{
"id" : "2755894/5246262",
"unique" : false,
"type" : "bitarray",
"fields" : [["x", [0, 1]], ["y", [0, 1]]],
"undefined" : false,
"isNewlyCreated" : false
}
```

In the example below we create a bitarray index with two fields. Field x can have the values of either 0 or 1; while field y can have the values of 2, "a" or other . A document which does not have both attributes x and y will not take part within the index. A document which does have both attributes x and y defined must have the values 0 or 1 for attribute x and any value for attribute y will be acceptable, otherwise the document will not be inserted within the collection.

```
arangod> db.example.ensureBitarray("x", [0,1], "y", [2,"a",[]]);
{
"id" : "2755894/5770550",
"unique" : false,
"type" : "bitarray",
"fields" : [["x", [0, 1]], ["y", [2, "a", [ ]]]],
"undefined" : false,
"isNewlyCreated" : true
}
```

Datafile Debugger

In Case Of Disaster

AranagoDB uses append-only journals. Data corruption should only occur when the database server is killed. In this case, the corruption should only occur in the last object(s) that have being written to the journal.

If a corruption occurs within a normal datafile, then this can only happen if a hardware fault occurred.

If a journal or datafile is corrupt, shut down the database server and start the program

```
unix> arango-dfdb
```

in order to check the consistency of the datafiles and journals. This brings up

```
 ___/    \__ _| |_ __ _ / _(_) |  __  ___/ ___\/ ___\/ _ \
   / /\ /___`| __/ _` | |_| | |/ _ \   / /\ /__\//  / /_\/
  / /_// (_| | || (_| | | _| | |  __/  / /_// \/   \/ /_\\
 /___,'  \__,_|\__\__,_|_| |_|_|\___| /___,'\_____/\____/

Available collections:
  0: _structures
  1: _users
  2: _routing
  3: _modules
  4: _graphs
  5: products
  6: prices
  *: all

Collection to check:
```

You can now select, which collection you want to check. After you selected one or all collections, a consistency check is performed.

```
Checking collection #1: _users

Database
  path: /usr/local/var/lib/arangodb

Collection
  name: _users
  identifier: 82343

Datafiles
  # of journals: 1
  # of compactors: 1
  # of datafiles: 0

Datafile
  path: /usr/local/var/lib/arangodb/collection-82343/journal-
1065383.db
  type: journal
  current size: 33554432
  maximal size: 33554432
  total used: 256
```

```
   # of entries: 3
   status: OK
```

If there is a problem with one of the datafile, then the database debugger tries to fixed that problem.

```
WARNING: The journal was not closed properly, the last entries is
corrupted.
        This might happen ArangoDB was killed and the last entries
were not
        fully written to disk.

Wipe the last entries (Y/N)?
```

If you answer **Y**, the corrupted entry will be removed.

If you see a corruption in a datafile (and not a journal), then something is terrible wrong. These files are immutable and never changed by ArangoDB. A corruption in such a file is an indication of a hard-disk failure.

Naming Conventions in ArangoDB

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

Database Names

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

- Database names must only consist of the letters _a_ to _z_ (both lower and upper case allowed), the numbers _0_ to _9_, and the underscore (_) or dash (-) symbols This also means that any non-ASCII database names are not allowed
- Database names must always start with a letter. Database names starting with an underscore are considered to be system databases, and users should not create or delete those
- The maximum allowed length of a database name is 64 bytes
- Database names are case-sensitive

Collection Names

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

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

Document Keys

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

- The key must be at least 1 byte and at most 254 bytes long. Empty keys are disallowed when specified (though it may be valid to completely omit the _key attribute from a document)
- It must consist of the letters a-z (lower or upper case), the digits 0-9, the underscore (_), dash (-), or colon (:) characters only
  - Any other characters, especially multi-byte sequences, whitespace or punctuation characters cannot be used inside key values
- The key must be unique within the collection it is used

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

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

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

Attribute Names

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

- Attribute names starting with an underscore are considered to be system attributes for ArangoDB's internal use. Such attribute names are already used by ArangoDB for special purposes, e.g. *_id* is used to contain a document's handle, *_key* is used to contain a document's user-defined key, and *_rev* is used to contain the document's revision number. In edge collections, the *_from* and *_to* attributes are used to reference other documents.

  More system attributes may be added in the future without further notice so end users should not use attribute names starting with an underscore for their own attributes.

- Attribute names should not start with the at-mark (\@). The at-mark at the start of attribute names is reserved in ArangoDB for future use cases.

- Theoretically, attribute names can include punctuation and special characters as desired, provided the name is a valid UTF-8 string. For maximum portability, special characters should be avoided though. For example, attribute names may contain the dot symbol, but the dot has a special meaning in Javascript and also in AQL, so when using such attribute names in one of these languages, the attribute name would need to be quoted by the end user. This will work but requires more work so it might be better to use attribute names which don't require any quoting/escaping in all languages used. This includes languages used by the client (e.g. Ruby, PHP) if the attributes are mapped to object members there.
- ArangoDB does not enforce a length limit for attribute names. However, long attribute names may use more memory in result sets etc. Therefore the use of long attribute names is discouraged.
- As ArangoDB saves document attribute names separate from the actual document attribute value data, the combined length of all attribute names for a document must fit into an ArangoDB shape structure. The maximum combined names length is variable and depends on the number and data types of attributes used.
- Attribute names are case-sensitive.

- Attributes with empty names (the empty string) and attributes with names that start with an underscore and don't have a special meaning (system attributes) are removed from the document when saving it.

  When the document is later requested, it will be returned without these attributes. For example, if this document is saved

  ```
  { "a" : 1, "" : 2, "_test" : 3, "b": 4 }
  ```

  and later requested, it will be returned like this:

  ```
  { "a" : 1, "b": 4 }
  ```

Error codes and meanings

The following errors might be raised when running ArangoDB:

- 0: *no error*0: no error No error has occurred. No error has occurred.1: failed Will be raised when a general error occurred.
- 1: *failed*2: system error Will be raised when operating system error occurred. Will be raised when a general error occurred.3: out of memory Will be raised when there is a memory shortage.
- 2: *system error*4: internal error Will be raised when an internal error occurred. Will be raised when operating system error occurred.5: illegal number Will be raised when an illegal representation of a number was given.
- 3: *out of memory*6: numeric overflow Will be raised when a numeric overflow occurred. Will be raised when there is a memory shortage.7: illegal option Will be raised when an unknown option was supplied by the user.
- 4: *internal error*8: dead process identifier Will be raised when a PID without a living process was found. Will be raised when an internal error occurred.9: not implemented Will be raised when hitting an unimplemented feature.
- 5: *illegal number*10: bad parameter Will be raised when the parameter does not fulfill the requirements. Will be raised when an illegal representation of a number was given.11: forbidden Will be raised when you are missing permission for the operation.
- 6: *numeric overflow*12: out of memory in mmap Will be raised when there is a memory shortage. Will be raised when a numeric overflow occurred.13: csv is corrupt Will be raised when encountering a corrupt csv line.
- 7: *illegal option*14: file not found Will be raised when a file is not found. Will be raised when an unknown option was supplied by the user.15: cannot write file Will be raised when a file cannot be written.
- 8: *dead process identifier*16: cannot overwrite file Will be raised when an attempt is made to overwrite an existing file. Will be raised when a PID without a living process was found.17: type error Will be raised when a type error is unencountered.
- 9: *not implemented*18: lock timeout Will be raised when there's a timeout waiting for a lock. Will be raised when hitting an unimplemented feature.19: cannot create directory Will be raised when an attempt to create a directory fails.
- 10: *bad parameter*20: cannot create temporary file Will be raised when an attempt to create a temporary file fails. Will be raised when the parameter does not fulfill the requirements.21: canceled request Will be raised when a request is canceled by the user.
- 11: *forbidden*22: intentional debug error Will be raised intentionally during debugging. Will be raised when you are missing permission for the operation.400: bad parameter Will be raised when the HTTP request does not fulfill the requirements.
- 12: *out of memory in mmap*401: unauthorized Will be raised when authorisation is required but the user is not authorised. Will be raised when there is a memory shortage.403: forbidden Will be raised when the operation is forbidden.
- 13: *csv is corrupt*404: not found Will be raised when an URI is unknown. Will be raised when encountering a corrupt csv line.405: method not supported Will be raised when an unsupported HTTP method is used for an operation.
- 14: *file not found*412: precondition failed Will be raised when a precondition for an HTTP request is not met. Will be raised when a file is not found.500: internal server error Will be raised when an internal server is encountered.
- 15: *cannot write file*600: invalid JSON object Will be raised when a string representation of a JSON object is corrupt. Will be raised when a file cannot be written.601: superfluous URL suffices Will be raised when the URL contains superfluous suffices.
- 16: *cannot overwrite file*1000: illegal state Internal error that will be raised when the datafile is not in the required state. Will be raised when an attempt is made to overwrite an existing file.1001: could not shape document Internal error that will be raised when the shaper encountered a problem.
- 17: *type error*1002: datafile sealed Internal error that will be raised when trying to write to a datafile. Will be raised when a type error is unencountered.1003: unknown type Internal error that will be raised when an unknown collection type is encountered.
- 18: *lock timeout*1004: read only Internal error that will be raised when trying to write to a read-

- only datafile or collection. Will be raised when there's a timeout waiting for a lock.1005: duplicate identifier Internal error that will be raised when a identifier duplicate is detected.
- 19: *cannot create directory*1006: datafile unreadable Internal error that will be raised when a datafile is unreadable. Will be raised when an attempt to create a directory fails.1007: datafile empty Internal error that will be raised when a datafile is empty.
- 20: *cannot create temporary file*1100: corrupted datafile Will be raised when a corruption is detected in a datafile. Will be raised when an attempt to create a temporary file fails.1101: illegal parameter file Will be raised if a parameter file is corrupted.
- 21: *canceled request*1102: corrupted collection Will be raised when a collection contains one or more corrupted datafiles. Will be raised when a request is canceled by the user.1103: mmap failed Will be raised when the system call mmap failed.
- 22: *intentional debug error*1104: filesystem full Will be raised when the filesystem is full. Will be raised intentionally during debugging.1105: no journal Will be raised when a journal cannot be created.
- 23: *internal error with attribute ID in shaper*1106: cannot create/rename datafile because it already exists Will be raised when the datafile cannot be created or renamed because a file of the same name already exists. Will be raised if an attribute ID is not found in the shaper but should1107: database directory is locked Will be raised when the database directory is locked by a different process. have been.1108: cannot create/rename collection because directory already exists Will be raised when the collection cannot be created because a directory of the same name already exists.
- 24: *internal error if a legend could not be created*1109: msync failed Will be raised when the system call msync failed. Will be raised if the legend generator was only given access to the shape1110: cannot lock database directory Will be raised when the server cannot lock the database directory on startup. and some sids are in the data object (inhomogeneous lists).1200: conflict Will be raised when updating or deleting a document and a conflict has been detected.
- 400: *bad parameter*1201: invalid database directory Will be raised when a non-existing database directory was specified when starting the database. Will be raised when the HTTP request does not fulfill the requirements.1202: document not found Will be raised when a document with a given identifier or handle is unknown.
- 401: *unauthorized*1203: collection not found Will be raised when a collection with a given identifier or name is unknown. Will be raised when authorization is required but the user is not1204: parameter 'collection' not found Will be raised when the collection parameter is missing. authorised. 1205: illegal document handle Will be raised when a document handle is corrupt.
- 403: *forbidden*1206: maximal size of journal too small Will be raised when the maximal size of the journal is too small. Will be raised when the operation is forbidden.1207: duplicate name Will be raised when a name duplicate is detected.
- 404: *not found*1208: illegal name Will be raised when an illegal name is detected. Will be raised when an URI is unknown.1209: no suitable index known Will be raised when no suitable index for the query is known.
- 405: *method not supported*1210: unique constraint violated Will be raised when there is a unique constraint violation. Will be raised when an unsupported HTTP method is used for an operation.1211: geo index violated Will be raised when an illegal coordinate is used.
- 412: *precondition failed*1212: index not found Will be raised when an index with a given identifier is unknown. Will be raised when a precondition for an HTTP request is not met.1213: cross collection request not allowed Will be raised when a cross-collection is requested.
- 500: *internal server error*1214: illegal index handle Will be raised when a index handle is corrupt. Will be raised when an internal server is encountered.1215: cap constraint already defined Will be raised when a cap constraint was already defined.
- 600: *invalid JSON object*1216: document too large Will be raised when the document cannot fit into any datafile because of it is too large. Will be raised when a string representation of a JSON object is corrupt.1217: collection must be unloaded Will be raised when a collection should be unloaded, but has a different status.
- 601: *superfluous URL suffices*1218: collection type invalid Will be raised when an invalid collection type is used in a request. Will be raised when the URL contains superfluous suffices.1219: validator failed Will be raised when the validation of an attribute of a structure failed.
- 1000: *illegal state*1220: parser failed Will be raised when the parsing of an attribute of a structure failed. Internal error that will be raised when the datafile is not in the1221: illegal document key Will be raised when a document key is corrupt. required state.1222: unexpected document key

Will be raised when a user-defined document key is supplied for collections with auto key generation.

- 1001: *could not shape document*1224: server database directory not writable Will be raised when the server's database directory is not writable for the current user. Internal error that will be raised when the shaper encountered a problem.1225: out of keys Will be raised when a key generator runs out of keys.

- 1002: *datafile sealed*1226: missing document key Will be raised when a document key is missing. Internal error that will be raised when trying to write to a datafile.1227: invalid document type Will be raised when there is an attempt to create a document with an invalid type.

- 1003: *unknown type*1228: database not found Will be raised when a non-existing database is accessed. Internal error that will be raised when an unknown collection type is1229: database name invalid Will be raised when an invalid database name is used. encountered.1230: operation only allowed in system database Will be raised when an operation is requested in a database other than the system database.

- 1004: *read only*1231: endpoint not found Will be raised when there is an attempt to delete a non-existing endpoint. Internal error that will be raised when trying to write to a read-only1232: invalid key generator Will be raised when an invalid key generator description is used. datafile or collection.1233: edge attribute missing will be raised when the _from or _to values of an edge are undefined or contain an invalid value.

- 1005: *duplicate identifier*1234: index insertion warning - attribute missing in document Will be raised when an attempt to insert a document into an index is caused by in the document not having one or more attributes which the index is built on. Internal error that will be raised when a identifier duplicate is1235: index creation failed Will be raised when an attempt to create an index has failed. detected.1300: datafile full Will be raised when the datafile reaches its limit.

- 1006: *datafile unreadable*1400: no response Will be raised when the replication applier does not receive any or an incomplete response from the master. Internal error that will be raised when a datafile is unreadable.1401: invalid response Will be raised when the replication applier receives an invalid response from the master.

- 1007: *datafile empty*1402: master error Will be raised when the replication applier receives a server error from the master. Internal error that will be raised when a datafile is empty.1403: master incompatible Will be raised when the replication applier connects to a master that has an incompatible version.

- 1100: *corrupted datafile*1404: master change Will be raised when the replication applier connects to a different master than before. Will be raised when a corruption is detected in a datafile.1405: loop detected Will be raised when the replication applier is asked to connect to itself for replication.

- 1101: *illegal parameter file*1406: unexpected marker Will be raised when an unexpected marker is found in the replication log stream. Will be raised if a parameter file is corrupted.1407: invalid applier state Will be raised when an invalid replication applier state file is found.

- 1102: *corrupted collection*1408: invalid transaction Will be raised when an unexpected transaction id is found. Will be raised when a collection contains one or more corrupted datafiles.1409: invalid replication logger configuration Will be raised when the configuration for the replication logger is invalid.

- 1103: *mmap failed*1410: invalid replication applier configuration Will be raised when the configuration for the replication applier is invalid. Will be raised when the system call mmap failed.1411: cannot change applier configuration while running Will be raised when there is an attempt to change the configuration for the replication applier while it is running.

- 1104: *filesystem full*1412: replication stopped Special error code used to indicate the replication applier was stopped by a user. Will be raised when the filesystem is full.1413: no start tick Will be raised when the replication error is started without a known start tick value.

- 1105: *no journal*1450: could not connect to agency Will be raised when none of the agency servers can be connected to. Will be raised when a journal cannot be created.1451: missing coordinator header Will be raised when a DB server in a cluster receives a HTTP request without a coordinator header.

- 1106: *cannot create/rename datafile because it already exists*1452: could not lock plan in agency Will be raised when a coordinator in a cluster cannot lock the Plan hierarchy in the agency. Will be raised when the datafile cannot be created or renamed because a1453: collection ID already exists Will be raised when a coordinator in a cluster tries to create a collection and the collection ID already exists. file of the same name already exists.1454: could not create collection in plan Will be raised when a coordinator in a cluster cannot create an entry for a new collection in the

Plan hierarchy in the agency.

- 1107: *database directory is locked*1455: could not read version in current in agency Will be raised when a coordinator in a cluster cannot read the Version entry in the Current hierarchy in the agency. Will be raised when the database directory is locked by a different1456: could not create collection Will be raised when a coordinator in a cluster notices that some DBServers report problems when creating shards for a new collection. process.1457: timeout in cluster operation Will be raised when a coordinator in a cluster runs into a timeout for some cluster wide operation.
- 1108: *cannot create/rename collection because directory already exists1458: could not remove collection from plan Will be raised when a coordinator in a cluster cannot remove an entry for a collection in the Plan hierarchy in the agency. Will be raised when the collection cannot be created because a directory1459: could not remove collection from current Will be raised when a coordinator in a cluster cannot remove an entry for a collection in the Current hierarchy in the agency. of the same name already exists.1460: could not create database in plan Will be raised when a coordinator in a cluster cannot create an entry for a new database in the Plan hierarchy in the agency.
- 1109: *msync failed*1461: could not create database Will be raised when a coordinator in a cluster notices that some DBServers report problems when creating databases for a new cluster wide database. Will be raised when the system call msync failed.1462: could not remove database from plan Will be raised when a coordinator in a cluster cannot remove an entry for a database in the Plan hierarchy in the agency.
- 1110: *cannot lock database directory*1463: could not remove database from current Will be raised when a coordinator in a cluster cannot remove an entry for a database in the Current hierarchy in the agency. Will be raised when the server cannot lock the database directory on1464: no responsible shard found Will be raised when a coordinator in a cluster cannot determine the shard that is responsible for a given document. startup.1465: cluster internal HTTP connection broken Will be raised when a coordinator in a cluster loses an HTTP connection to a DBserver in the cluster whilst transferring data.
- 1200: *conflict*1466: must not specify _key for this collection Will be raised when a coordinator in a cluster finds that the _key attribute was specified in a sharded collection the uses not only _key as sharding attribute. Will be raised when updating or deleting a document and a conflict has1467: got contradicting answers from different shards Will be raised if a coordinator in a cluster gets conflicting results from different shards, which should never happen. been detected.1468: not all sharding attributes given Will be raised if a coordinator tries to find out which shard is responsible for a partial document, but cannot do this because not all sharding attributes are specified.
- 1201: *invalid database directory*1469: must not change the value of a shardkey attribute Will be raised if there is an attempt to update the value of a shard attribute. Will be raised when a non-existing database directory was specified when1470: unsupported operation or parameter Will be raised when there is an attempt to carry out an operation that is not supported in the context of a sharded collection. starting the database.1471: this operation is only valid on a coordinator in a cluster Will be raised if there is an attempt to run a coordinator-only operation on a different type of node.
- 1202: *document not found*1472: error reading Plan in agency Will be raised if a coordinator or DBserver cannot read the Plan in the agency. Will be raised when a document with a given identifier or handle is1473: could not truncate collection Will be raised if a coordinator cannot truncate all shards of a cluster collection. unknown.1500: query killed Will be raised when a running query is killed by an explicit admin command.
- 1203: *collection not found*1501: %s Will be raised when query is parsed and is found to be syntactically invalid. Will be raised when a collection with a given identifier or name is1502: query is empty Will be raised when an empty query is specified. unknown.1503: runtime error '%s' Will be raised when a runtime error is caused by the query.
- 1204: *parameter 'collection' not found*1504: number out of range Will be raised when a number is outside the expected range. Will be raised when the collection parameter is missing.1510: variable name '%s' has an invalid format Will be raised when an invalid variable name is used.
- 1205: *illegal document handle*1511: variable '%s' is assigned multiple times Will be raised when a variable gets re-assigned in a query. Will be raised when a document handle is corrupt.1512: unknown variable '%s' Will be raised when an unknown variable is used or the variable is undefined the context it is used.
- 1206: *maixmal size of journal too small*1521: unable to read-lock collection %s Will be raised when a read lock on the collection cannot be acquired. Will be raised when the maximal size of

the journal is too small.1522: too many collections Will be raised when the number of collections in a query is beyond the allowed value.

- 1207: *duplicate name*1530: document attribute '%s' is assigned multiple times Will be raised when a document attribute is re-assigned. Will be raised when a name duplicate is detected.1540: usage of unknown function '%s()' Will be raised when an undefined function is called.
- 1208: *illegal name*1541: invalid number of arguments for function '%s()' Will be raised when the number of arguments used in a function call does not match the expected number of arguments for the function. Will be raised when an illegal name is detected.1542: invalid argument type used in call to function '%s()' Will be raised when the type of an argument used in a function call does not match the expected argument type.
- 1209: *no suitable index known*1543: invalid regex argument value used in call to function '%s()' Will be raised when an invalid regex argument value is used in a call to a function that expects a regex. Will be raised when no suitable index for the query is known.1550: invalid structure of bind parameters Will be raised when the structure of bind parameters passed has an unexpected format.
- 1210: *unique constraint violated*1551: no value specified for declared bind parameter '%s' Will be raised when a bind parameter was declared in the query but the query is being executed with no value for that parameter. Will be raised when there is a unique constraint violation.1552: bind parameter '%s' was not declared in the query Will be raised when a value gets specified for an undeclared bind parameter.
- 1211: *geo index violated*1553: bind parameter '%s' has an invalid value or type Will be raised when a bind parameter has an invalid value or type. Will be raised when an illegal coordinate is used.1560: invalid logical value Will be raised when a non-boolean value is used in a logical operation.
- 1212: *index not found*1561: invalid arithmetic value Will be raised when a non-numeric value is used in an arithmetic operation. Will be raised when an index with a given identifier is unknown.1562: division by zero Will be raised when there is an attempt to divide by zero.
- 1213: *cross collection request not allowed*1563: list expected Will be raised when a non-list operand is used for an operation that expects a list argument operand. Will be raised when a cross-collection is requested.1569: FAIL(%s) called Will be raised when the function FAIL() is called from inside a query.
- 1214: *illegal index handle*1570: no suitable geo index found for geo restriction on '%s' Will be raised when a geo restriction was specified but no suitable geo index is found to resolve it. Will be raised when a index handle is corrupt.1571: no suitable fulltext index found for fulltext query on '%s' Will be raised when a fulltext query is performed on a collection without a suitable fulltext index.
- 1215: *cap constraint already defined*1572: invalid date value Will be raised when a value cannot be converted to a date. Will be raised when a cap constraint was already defined.1580: invalid user function name Will be raised when a user function with an invalid name is registered.
- 1216: *document too large*1581: invalid user function code Will be raised when a user function is registered with invalid code. Will be raised when the document cannot fit into any datafile because of1582: user function '%s()' not found Will be raised when a user function is accessed but not found. it is too large.1600: cursor not found Will be raised when a cursor is requested via its id but a cursor with that id cannot be found.
- 1217: *collection must be unloaded*1650: internal transaction error Will be raised when a wrong usage of transactions is detected. this is an internal error and indicates a bug in ArangoDB. Will be raised when a collection should be unloaded, but has a different1651: nested transactions detected Will be raised when transactions are nested. status.1652: unregistered collection used in transaction Will be raised when a collection is used in the middle of a transaction but was not registered at transaction start.
- 1218: *collection type invalid*1653: disallowed operation inside transaction Will be raised when a disallowed operation is carried out in a transaction. Will be raised when an invalid collection type is used in a request.1700: invalid user name Will be raised when an invalid user name is used.
- 1219: *validator failed*1701: invalid password Will be raised when an invalid password is used. Will be raised when the validation of an attribute of a structure failed.1702: duplicate user Will be raised when a user name already exists.
- 1220: *parser failed*1703: user not found Will be raised when a user name is updated that does not exist. Will be raised when the parsing of an attribute of a structure failed.1704: user must change his password Will be raised when the user must change his password.
- 1221: *illegal document key*1750: invalid application name Will be raised when an invalid

application name is specified. Will be raised when a document key is corrupt.1751: invalid mount Will be raised when an invalid mount is specified.

- 1222: *unexpected document key*1752: application download failed Will be raised when an application download from the central repository failed. Will be raised when a user-defined document key is supplied for1753: application upload failed Will be raised when an application upload from the client to the ArangoDB server failed. collections with auto key generation.1800: invalid key declaration Will be raised when an invalid key specification is passed to the server

- 1224: *server database directory not writable*1801: key already exists Will be raised when a key is to be created that already exists Will be raised when the server's database directory is not writable for1802: key not found Will be raised when the specified key is not found the current user.1803: key is not unique Will be raised when the specified key is not unique

- 1225: *out of keys*1804: key value not changed Will be raised when updating the value for a key does not work Will be raised when a key generator runs out of keys.1805: key value not removed Will be raised when deleting a key/value pair does not work

- 1226: *missing document key*1806: missing value Will be raised when the value is missing Will be raised when a document key is missing.1850: invalid task id Will be raised when a task is created with an invalid id.

- 1227: *invalid document type*1851: duplicate task id Will be raised when a task id is created with a duplicate id. Will be raised when there is an attempt to create a document with an1852: task not found Will be raised when a task with the specified id could not be found. invalid type.1901: invalid graph Will be raised when an invalid name is passed to the server.

- 1228: *database not found*1902: could not create graph Will be raised when an invalid name, vertices or edges is passed to the server. Will be raised when a non-existing database is accessed.1903: invalid vertex Will be raised when an invalid vertex id is passed to the server.

- 1229: *database name invalid*1904: could not create vertex Will be raised when the vertex could not be created. Will be raised when an invalid database name is used.1905: could not change vertex Will be raised when the vertex could not be changed.

- 1230: *operation only allowed in system database*1906: invalid edge Will be raised when an invalid edge id is passed to the server. Will be raised when an operation is requested in a database other than1907: could not create edge Will be raised when the edge could not be created. the system database.1908: could not change edge Will be raised when the edge could not be changed.

- 1231: *endpoint not found*1909: too many iterations Will be raised when too many iterations are done in a graph traversal. Will be raised when there is an attempt to delete a non-existing endpoint.1910: invalid filter result Will be raised when an invalid filter result is returned in a graph traversal.

- 1232: *invalid key generator*1950: unknown session Will be raised when an invalid/unknown session id is passed to the server. Will be raised when an invalid key generator description is used.1951: session expired Will be raised when a session is expired.

- 1233: *edge attribute missing*2000: unknown client error This error should not happen. will be raised when the _from or _to values of an edge are undefined or2001: could not connect to server Will be raised when the client could not connect to the server. contain an invalid value.2002: could not write to server Will be raised when the client could not write data.

- 1234: *index insertion warning - attribute missing in document*2003: could not read from server Will be raised when the client could not read data. Will be raised when an attempt to insert a document into an index is3402: bitarray index update warning - attribute missing in revised document Will be raised when an attempt to update a document results in the revised document not having one or more attributes which are required by the bitarray index. caused by in the document not having one or more attributes which the3411: bitarray index remove failure - item missing in index Will be raised when an attempt to remove a document from a bitarray index fails when document can not be located within that index. index is built on.3413: bitarray index insert failure - document attribute value unsupported in index Will be raised when an attempt to insert a document into a bitarray index fails due to the fact that one or more values for an index attribute is not supported within that index.

- 1235: *index creation failed*3415: bitarray index creation failure - one or more index attributes are duplicated. Will be raised when an attempt to create an index with two or more index attributes repeated. Will be raised when an attempt to create an index has failed.3417: bitarray index creation failure - one or more index attribute values are duplicated. Will be raised when an attempt to create an index with two or more index attribute values repeated.

- 1300: *datafile full*10000: element not inserted into structure, because key already exists Will be

returned if the element was not insert because the key already exists. Will be raised when the datafile reaches its limit.10001: element not inserted into structure, because it already exists Will be returned if the element was not insert because it already exists.

- 1400: *no response*10002: key not found in structure Will be returned if the key was not found in the structure. Will be raised when the replication applier does not receive any or an10003: element not found in structure Will be returned if the element was not found in the structure. incomplete response from the master.
- 1401: *invalid response* Will be raised when the replication applier receives an invalid response<!-- from the master.@NAVIGATE_ImplementorManualArangoErrors
- 1402: *master error*@EMBEDTOC{ImplementorManualArangoErrorsTOC* Will be raised when the replication applier receives a server error from the master.@copydetails ArangoErrors
- 1403: *master incompatible* Will be raised when the replication applier connects to a master that has@BNAVIGATE_ImplementorManualArangoErrors an incompatible version.-->
- 1404: *master change* Will be raised when the replication applier connects to a different master than before.
- 1405: *loop detected* Will be raised when the replication applier is asked to connect to itself for replication.
- 1406: *unexpected marker* Will be raised when an unexpected marker is found in the replication log stream.
- 1407: *invalid applier state* Will be raised when an invalid replication applier state file is found.
- 1408: *invalid transaction* Will be raised when an unexpected transaction id is found.
- 1409: *invalid replication logger configuration* Will be raised when the configuration for the replication logger is invalid.
- 1410: *invalid replication applier configuration* Will be raised when the configuration for the replication applier is invalid.
- 1411: *cannot change applier configuration while running* Will be raised when there is an attempt to change the configuration for the replication applier while it is running.
- 1412: *replication stopped* Special error code used to indicate the replication applier was stopped by a user.
- 1413: *no start tick* Will be raised when the replication error is started without a known start tick value.
- 1450: *could not connect to agency* Will be raised when none of the agency servers can be connected to.
- 1451: *missing coordinator header* Will be raised when a DB server in a cluster receives a HTTP request without a coordinator header.
- 1452: *could not lock plan in agency* Will be raised when a coordinator in a cluster cannot lock the Plan hierarchy in the agency.
- 1453: *collection ID already exists* Will be raised when a coordinator in a cluster tries to create a collection and the collection ID already exists.
- 1454: *could not create collection in plan* Will be raised when a coordinator in a cluster cannot create an entry for a new collection in the Plan hierarchy in the agency.
- 1455: *could not read version in current in agency* Will be raised when a coordinator in a cluster cannot read the Version entry in the Current hierarchy in the agency.
- 1456: *could not create collection* Will be raised when a coordinator in a cluster notices that some DBServers report problems when creating shards for a new collection.
- 1457: *timeout in cluster operation* Will be raised when a coordinator in a cluster runs into a timeout for some cluster wide operation.
- 1458: *could not remove collection from plan* Will be raised when a coordinator in a cluster cannot remove an entry for a collection in the Plan hierarchy in the agency.
- 1459: *could not remove collection from current* Will be raised when a coordinator in a cluster cannot remove an entry for a collection in the Current hierarchy in the agency.
- 1460: *could not create database in plan* Will be raised when a coordinator in a cluster cannot create an entry for a new database in the Plan hierarchy in the agency.
- 1461: *could not create database* Will be raised when a coordinator in a cluster notices that some DBServers report problems when creating databases for a new cluster wide database.
- 1462: *could not remove database from plan* Will be raised when a coordinator in a cluster cannot remove an entry for a database in the Plan hierarchy in the agency.
- 1463: *could not remove database from current* Will be raised when a coordinator in a cluster cannot remove an entry for a database in the Current hierarchy in the agency.
- 1464: *no responsible shard found* Will be raised when a coordinator in a cluster cannot determine

the shard that is responsible for a given document.

- 1465: *cluster internal HTTP connection broken* Will be raised when a coordinator in a cluster loses an HTTP connection to a DBserver in the cluster whilst transferring data.
- 1466: *must not specify _key for this collection* Will be raised when a coordinator in a cluster finds that the _key attribute was specified in a sharded collection the uses not only _key as sharding attribute.
- 1467: *got contradicting answers from different shards* Will be raised if a coordinator in a cluster gets conflicting results from different shards, which should never happen.
- 1468: *not all sharding attributes given* Will be raised if a coordinator tries to find out which shard is responsible for a partial document, but cannot do this because not all sharding attributes are specified.
- 1469: *must not change the value of a shardkey attribute* Will be raised if there is an attempt to update the value of a shard attribute.
- 1470: *unsupported operation or parameter* Will be raised when there is an attempt to carry out an operation that is not supported in the context of a sharded collection.
- 1471: *this operation is only valid on a coordinator in a cluster* Will be raised if there is an attempt to run a coordinator-only operation on a different type of node.
- 1472: *error reading Plan in agency* Will be raised if a coordinator or DBserver cannot read the Plan in the agency.
- 1473: *could not truncate collection* Will be raised if a coordinator cannot truncate all shards of a cluster collection.
- 1500: *query killed* Will be raised when a running query is killed by an explicit admin command.
- 1501: *\%s* Will be raised when query is parsed and is found to be syntactially invalid.
- 1502: *query is empty* Will be raised when an empty query is specified.
- 1503: *runtime error '\%s'* Will be raised when a runtime error is caused by the query.
- 1504: *number out of range* Will be raised when a number is outside the expected range.
- 1510: *variable name '\%s' has an invalid format* Will be raised when an invalid variable name is used.
- 1511: *variable '\%s' is assigned multiple times* Will be raised when a variable gets re-assigned in a query.
- 1512: *unknown variable '\%s'* Will be raised when an unknown variable is used or the variable is undefined the context it is used.
- 1521: *unable to read-lock collection \%s* Will be raised when a read lock on the collection cannot be acquired.
- 1522: *too many collections* Will be raised when the number of collections in a query is beyond the allowed value.
- 1530: *document attribute '\%s' is assigned multiple times* Will be raised when a document attribute is re-assigned.
- 1540: *usage of unknown function '\%s()'* Will be raised when an undefined function is called.
- 1541: *invalid number of arguments for function '\%s()'* Will be raised when the number of arguments used in a function call does not match the expected number of arguments for the function.
- 1542: *invalid argument type used in call to function '\%s()'* Will be raised when the type of an argument used in a function call does not match the expected argument type.
- 1543: *invalid regex argument value used in call to function '\%s()'* Will be raised when an invalid regex argument value is used in a call to a function that expects a regex.
- 1550: *invalid structure of bind parameters* Will be raised when the structure of bind parameters passed has an unexpected format.
- 1551: *no value specified for declared bind parameter '\%s'* Will be raised when a bind parameter was declared in the query but the query is being executed with no value for that parameter.
- 1552: *bind parameter '\%s' was not declared in the query* Will be raised when a value gets specified for an undeclared bind parameter.
- 1553: *bind parameter '\%s' has an invalid value or type* Will be raised when a bind parameter has an invalid value or type.
- 1560: *invalid logical value* Will be raised when a non-boolean value is used in a logical operation.
- 1561: *invalid arithmetic value* Will be raised when a non-numeric value is used in an arithmetic operation.
- 1562: *division by zero* Will be raised when there is an attempt to divide by zero.
- 1563: *list expected* Will be raised when a non-list operand is used for an operation that expects a

list argument operand.

- 1569: *FAIL(\%s) called* Will be raised when the function FAIL() is called from inside a query.
- 1570: *no suitable geo index found for geo restriction on '\%s'* Will be raised when a geo restriction was specified but no suitable geo index is found to resolve it.
- 1571: *no suitable fulltext index found for fulltext query on '\%s'* Will be raised when a fulltext query is performed on a collection without a suitable fulltext index.
- 1572: *invalid date value* Will be raised when a value cannot be converted to a date.
- 1580: *invalid user function name* Will be raised when a user function with an invalid name is registered.
- 1581: *invalid user function code* Will be raised when a user function is registered with invalid code.
- 1582: *user function '\%s()' not found* Will be raised when a user function is accessed but not found.
- 1600: *cursor not found* Will be raised when a cursor is requested via its id but a cursor with that id cannot be found.
- 1650: *internal transaction error* Will be raised when a wrong usage of transactions is detected. this is an internal error and indicates a bug in ArangoDB.
- 1651: *nested transactions detected* Will be raised when transactions are nested.
- 1652: *unregistered collection used in transaction* Will be raised when a collection is used in the middle of a transaction but was not registered at transaction start.
- 1653: *disallowed operation inside transaction* Will be raised when a disallowed operation is carried out in a transaction.
- 1700: *invalid user name* Will be raised when an invalid user name is used.
- 1701: *invalid password* Will be raised when an invalid password is used.
- 1702: *duplicate user* Will be raised when a user name already exists.
- 1703: *user not found* Will be raised when a user name is updated that does not exist.
- 1704: *user must change his password* Will be raised when the user must change his password.
- 1750: *invalid application name* Will be raised when an invalid application name is specified.
- 1751: *invalid mount* Will be raised when an invalid mount is specified.
- 1752: *application download failed* Will be raised when an application download from the central repository failed.
- 1753: *application upload failed* Will be raised when an application upload from the client to the ArangoDB server failed.
- 1800: *invalid key declaration* Will be raised when an invalid key specification is passed to the server
- 1801: *key already exists* Will be raised when a key is to be created that already exists
- 1802: *key not found* Will be raised when the specified key is not found
- 1803: *key is not unique* Will be raised when the specified key is not unique
- 1804: *key value not changed* Will be raised when updating the value for a key does not work
- 1805: *key value not removed* Will be raised when deleting a key/value pair does not work
- 1806: *missing value* Will be raised when the value is missing
- 1850: *invalid task id* Will be raised when a task is created with an invalid id.
- 1851: *duplicate task id* Will be raised when a task id is created with a duplicate id.
- 1852: *task not found* Will be raised when a task with the specified id could not be found.
- 1901: *invalid graph* Will be raised when an invalid name is passed to the server.
- 1902: *could not create graph* Will be raised when an invalid name, vertices or edges is passed to the server.
- 1903: *invalid vertex* Will be raised when an invalid vertex id is passed to the server.
- 1904: *could not create vertex* Will be raised when the vertex could not be created.
- 1905: *could not change vertex* Will be raised when the vertex could not be changed.
- 1906: *invalid edge* Will be raised when an invalid edge id is passed to the server.
- 1907: *could not create edge* Will be raised when the edge could not be created.
- 1908: *could not change edge* Will be raised when the edge could not be changed.
- 1909: *too many iterations* Will be raised when too many iterations are done in a graph traversal.
- 1910: *invalid filter result* Will be raised when an invalid filter result is returned in a graph traversal.
- 1950: *unknown session* Will be raised when an invalid/unknown session id is passed to the server.
- 1951: *session expired* Will be raised when a session is expired.
- 2000: *unknown client error* This error should not happen.
- 2001: *could not connect to server* Will be raised when the client could not connect to the server.

- 2002: *could not write to server* Will be raised when the client could not write data.
- 2003: *could not read from server* Will be raised when the client could not read data.
- 3402: *bitarray index update warning - attribute missing in revised doc Will be raised when an attempt to update a document results in the revised document not having one or more attributes which are required by the bitarray index.
- 3411: *bitarray index remove failure - item missing in index* Will be raised when an attempt to remove a document from a bitarray index fails when document can not be located within that index.
- 3413: *bitarray index insert failure - document attribute value unsuppo Will be raised when an attempt to insert a document into a bitarray index fails due to the fact that one or more values for an index attribute is not supported within that index.
- 3415: *bitarray index creation failure - one or more index attributes a Will be raised when an attempt to create an index with two or more index attributes repeated.
- 3417: *bitarray index creation failure - one or more index attribute va Will be raised when an attempt to create an index with two or more index attribute values repeated.
- 10000: *element not inserted into structure, because key already exists Will be returned if the element was not insert because the key already exists.
- 10001: *element not inserted into structure, because it already exists* Will be returned if the element was not insert because it already exists.
- 10002: *key not found in structure* Will be returned if the key was not found in the structure.
- 10003: *element not found in structure* Will be returned if the element was not found in the structure.

Inhaltsverzeichnis