School Engine

CQL Managers LLD

Document Number 3

Table of Contents

[Introduction 3](#_Toc123810627)

[ResultCode 4](#_Toc123810628)

[CqlResult 4](#_Toc123810629)

[Structs & Smart pointers 4](#_Toc123810630)

[CqlClient 5](#_Toc123810631)

[Constructor 5](#_Toc123810632)

[connect() 5](#_Toc123810633)

[execute\_statement() 5](#_Toc123810634)

[select\_rows() 5](#_Toc123810635)

[prepare\_statement() 5](#_Toc123810636)

[Global functions 5](#_Toc123810637)

[get\_cql\_result() 5](#_Toc123810638)

[get\_result\_code() 6](#_Toc123810639)

[was\_applied() 6](#_Toc123810640)

[Value getters 6](#_Toc123810641)

[select\_rows() 7](#_Toc123810642)

[CqlManagers 8](#_Toc123810643)

[Classes 8](#_Toc123810644)

[Private variables 8](#_Toc123810645)

[Constructor and destructor 8](#_Toc123810646)

[configure () 8](#_Toc123810647)

[init\_prepare\_statements () 8](#_Toc123810648)

[init\_schema () 8](#_Toc123810649)

[create\_object () 8](#_Toc123810650)

[get\_object () 8](#_Toc123810651)

[update\_object () 8](#_Toc123810652)

[delete\_object () 8](#_Toc123810653)

[map\_row\_to\_object () 8](#_Toc123810654)

# Introduction

The purpose of this file is to explain the logic behind all of the classes that will be part of the CQL manager group. These classes will take care of creating the connection between the backend and the Cassandra database and will also execute the CQL queries to the database.

# ResultCode

The ResultCode will be an enumerator class that will store 9 types of different results that an operation can return. Usually, we want to the get the ResultCode::OK result but that is not a rule. Other result codes may be INVALID\_REQUEST, NOT\_FOUND, CONNECTION\_ERROR, RESOURCE\_ERROR, UNKNOWN\_ERROR, UNAVAILABLE, TIMEOUT, NOT\_APPLIED (for when we want to see if the command was applied). All of these result codes will be transferred from a normal Cassandra result code to our list of codes. This implementation exists because of the high number of error codes that Cassandra provides; thus, I had decided to merge some of them.

# CqlResult

The CqlResult class will be responsible for storing the result of an operation. It will have two main fields that will be responsible for storing information: \_code and \_error. The \_code represents the result code of an operation and the \_error represents the string of the error; if the operation is successful, the string will be empty.

Bothe the \_code and \_error are private fields of the class, so functions code() and error() have been implemented as getter functions for the two variables respectively.

# Structs & Smart pointers

The Cassandra C++ library has some default functions that free memory (cass\_future\_free, cass\_statement\_free etc) but they are hard to keep track of, therefore I have added new types that will take the Cassandra object and free it automatically when the variable is out of scope. This way, the code will be safer and there will be no memory leaks. This implementation can be found in include/cql\_helpers/cql\_client.hpp.

# CqlClient

The CqlClient will be the most important class what will communicate with the database. It is responsible for taking the information, sending it to the database and process the response.

## Constructor

The constructor will take the hostname and the port on which Cassandra is running. For local development, this will be 127.0.0.1 and 9042. It will just store the data into the private fields of the class.

## connect()

This function will try to connect to the database. It will also check for if any errors occur while connecting but will also set the log level of Cassandra to only display the errors.

## execute\_statement()

This function will be the one which executes the statements to the database. It will do this asynchronously therefore making the code non-blocking. It will first wait for the command to be executed, then wait for the response.

This function is intended to receive a CassStatement as parameter. I have also given the liberty of receiving a string as parameter. If this is the case, the function will create a CassStatement object with the received string and then call the normal function.

## select\_rows()

The purpose of this function is to select the lines of a result. This will happen nearly always as, even if we change a field into the database, we want to see if that field was actually changed. Cassandra makes this process easy because, after we execute a query, we would get an extra field than expected, that being the was\_applied.

This function will use some modern C++ features. It will receive a CassStatement that will hold the query that we want to execute as parameter, two functions: one that allocates the memory for the new data and another one that will take the row as parameter and add the data to the previously allocated memory. Furthermore, we will also have a parameter that will indicate if we actually expect a row to be returned from the database.

## prepare\_statement()

This function is responsible for building a statement that can be used multiple times. This is done for security reasons in order to prevent breaches such as an SQL injection attack.

# Global functions

## get\_cql\_result()

Will return a CqlResult class that will contain the result of the operation executed on the Cassandra cluster. Note that if the result of the operation is not successful (aka. ResultCode::OK), the function will also read try and get the error message.

## get\_result\_code()

Will change the error codes of a normal enumerator CassError into our personal enumerator ResultCode. In order to change the default Cassandra error messages into our custom ones, we will use the next rules:

|  |  |
| --- | --- |
| **CassError** | **ResultCode** |
| CASS\_OK | OK |
| CASS\_ERROR\_SERVER\_INVALID\_QUERY | INVALID\_REQUEST |
| CASS\_ERROR\_LIB\_NO\_HOSTS\_AVAILABLE | CONNECTION\_ERROR |
| CASS\_ERROR\_SERVER\_READ\_FAILURE  CASS\_ERROR\_SERVER\_FUNCTION\_FAILURE  CASS\_ERROR\_SERVER\_WRITE\_FAILURE | RESOURCE\_ERROR |
| CASS\_ERROR\_SERVER\_UNAVAILABLE | UNAVALABLE |
| CASS\_ERROR\_SERVER\_WRITE\_TIMEOUT  CASS\_ERROR\_SERVER\_READ\_TIMEOUT  CASS\_ERROR\_LIB\_REQUEST\_TIMED\_OUT | TIMEOUT |
| Anything else | UNKNOWN\_ERROR |

## was\_applied()

Will check if a command was applied. In practice the only time that we are not going to use this function is when readding some variables form the database and when updating/deleting an entry from the database and we did not provide all the primary keys and the clustering keys that the table has.

The function will try looking for the [applied] column of a row. After that, it will return the result. If the [applied] field is false, we will return a ResultCode::NOT\_APPLIED.

## Value getters

The next 7 function will have the same basic structure. Each one of them will take as parameters a CassRow pointer, the column index and a reference to the variable that we want to read. The function will try to read get the column at the specified index from the given row and then read the variable form that column.

In order to read the variable that we want, we will use the cass\_value\_get\_\* function provided by the Cassandra library.

The functions that we will use will be:

|  |  |
| --- | --- |
| Function name | Function scope |
| get\_bool\_value() | Will read a bool value from a column. |
| get\_int\_value() | Will read an int value from a column. |
| get\_long\_value() | Will read a long value from a column. Usually used for readding time\_t variables. Note that Cassandra also stores the milliseconds of a time, therefore we will need to also divide the result by 1000 after we read it. |
| get\_float\_value() | Will read a float value from a column. |
| get\_text\_value() | Will read a string value from a column. |
| get\_uuid\_value() | Will read an uuid value from a column. |
| get\_array\_uuids\_value() | Will read a vector of uuids from a column that contains a set of uuids (in the databse). |

## select\_rows()

The select row function will take as parameters a pointer to the CassStement (the statement that we want to execute), a function that will allocate the memory for the data that we want to read, and a function that will read the data from the database using the function from the previous subchapter.

The flow of the program will be to first execute the statement, then read the number of rows that the response will have, allocate the memory (using the first function) for the objects, then iterate through the rows retuned by Cassandra and handle each one of them with the second function.

# CqlManagers

Considering that we will have a class for each table (23 to be exact), it would be inappropriate to explain in detail what each class and its functions will do, therefore this document will describe the structure of a general class and only go in detail when it is the case.

Note that the project splits the tables into two categories: object tables (like the schools.users table) and the relationship tables (like the schools.users\_by\_tag table). Some of the implementations will be different from one type of table to the other but they apply the same code structure.

## Classes

As mentioned before, each table in our database will have a correlated C++ class that will take care of that table. The naming of them will be standardized and will use the name of the table concatenated with \_cql\_manager (example: answers\_cql\_manager).

## Private variables

Each class will store a pointer to the CqlClient that we have created prior to the creation of the class.

Each class will contain multiple CassPreparedPtr objects that we will use when creating when executing a statement. They will usually be 4 different prepared statement: \_prepared\_insert\_object, \_prepared\_get\_object, \_prepared\_update\_object, and \_prepared\_delete\_object. Some classes, such as the one that manages the grades table, will contain more prepared statements in order to manage more commands, for example getting a table’s data based on a field that is not a primary key or to delete an entry based on fewer primary keys (can be seen in any cql manager for a relation).

Furthermore, each class will contain multiple constant char pointers that will store the actual command that we will execute. They will usually be CREATE\_SCHOOL\_KEYSPACE, CREATE\_OBJECT\_TABLE, INSERT\_OBJECT, SELECT\_OBJECT, UPDATE\_OBJECT, DELETE\_OBJECT. Again, some classes will contain more than 6 commands.

## Constructor and destructor

The constructor will take as parameter a pointer to the CqlClient and will be stored locally until we destruct the class. The destructor will be empty, considering that we don’t have to deallocate memory.

## configure ()

This function will take as parameters a bool value that will represent whether we want to also create the table and keyspace if they don’t exist or not. If they exist and we execute the command, it is no problem because we create them with the additional Cassandra statement “IF NOT EXISTS”, thus no errors will occur.

The function will also initialise the prepared statement using the init\_prepared\_statment function described below.

## init\_prepare\_statements ()

This function will initialise the prepared statements with the char pointers that we have mentioned previously. As we can see, each CassPreparedPtr will have a string as it’s command, excepting the CREATE\_SCHOOL\_KEYSPACE and CREATE\_OBJECT\_TABLE strings.

## init\_schema ()

The function will execute the CREATE\_SCHOOL\_KEYSPACE and CREATE\_OBJECT\_TABLE commands. This will result in the creation of the related keyspace and the related table.

The function will return ResultCode::OK or, if the execution fails, will return the correlated error code.

## create\_object ()

The create object function will take a reference to an object as parameter if it manages an object or the school’s id plus two uuids if it manages a relationship between two other objects.

It will, in a try – catch block, try to bind the prepared statement to a normal Cassandra statement and then, assign each of the object’s variables to a field in the statement. After that, the function will execute the statement and check if the command was applied. After that, the result will be returned.

If the function crashes during execution, an error message will be returned, alongside ResultCode::UNKNOWN\_ERROR.

If the command was not applied, the function will return ResultCode::NOT\_APPLIED.

If the execution was successful, the function will return ResultCode::OK.

## get\_object ()

The function will create a vector of objects that will try and read from the database.

In this function, two other lambda functions will be defined, one that will allocate memory and another one that will copy the read information from the Cassandra result and put it into the previous vector. Note that the function map\_row\_to\_object will be used to map the Cassandra row to the actual object; this will be used only when working with objects tables and not with relationships tables, they will just read a CassUuid value from the row.

After we use the select\_rows function from the CqlClient class, the function will check either we get a ResultCode::OK or not. If not, return the error code.

If we are working with the relationship tables, the function will return the result of the operations now.

If not, and we are working with a class’s table, we would first check if we only have one entry in the vector and then return the first object of the vector.

## update\_object ()

This function will be used only for the object’s tables. It will take the as parameter all of the object’s fields and try to execute a statement with this fields.

It will, in a try – catch block, try to bind the prepared statement to a normal Cassandra statement and then, assign each of the object’s variables to a field in the statement. After that, the function will execute the statement and check if the command was applied. After that, the result will be returned.

If the function crashes during execution, an error message will be returned, alongside ResultCode::UNKNOWN\_ERROR.

If the command was not applied, the function will return ResultCode::NOT\_APPLIED.

If the execution was successful, the function will return ResultCode::OK.

## delete\_object ()

This function will usually take all the primary and clustering keys of an object as parameters and try to map them to a statement in order to delete some fields in the database. It will, in a try – catch block, try to bind the prepared statement to a normal Cassandra statement and then, assign each of the object’s variables to a field in the statement. After that, the function will execute the statement and check if the command was applied. After that, the result will be returned.

Checking whether a command was applied is not a rule; in some cases (example: when we delete a field and not provide all the primary keys and clustering keys) we will not be able to execute commands that have the Cassandra “IF EXISTS” condition, therefore we will not get the [applied] field in the response. This is a problem considering that we will not be able to check if we deleted some fields, but these problems will be assessed in the testing part.

If the function crashes during execution, an error message will be returned, alongside ResultCode::UNKNOWN\_ERROR.

If the command was not applied, the function will return ResultCode::NOT\_APPLIED.

If the execution was successful, the function will return ResultCode::OK.

## map\_row\_to\_object ()

This function will be used when trying to map a Cassandra row to an object. It will take as parameters the Cassandra row and a reference to the related object.

In order to map the Cassandra columns of the row to classic variables we will use the [value getter](#_Value_getters) functions explained previously.

The function will return a ResultCode::OK if the operations went as planned, or the related ResultCode if we get an error.

# Testing

In order to test the functionality of our code each ObjectCqlManager will have its own suite of tests.

Each test, before its execution, will create a CqlClient and the related ObjectCqlManager. Furthermore, the database will be cleaned before each execution in order to be sure that there is no data that can affect the end result.

Each class will have test the next features: writing and entry, reading an entry, updating an entry, deleting an entry (the relationship tables will have 2 tests here), inserting an object twice, readding an non-existent object and deleting a non-existent object. Each test will check if the flow of the code is correct and if the returned results are the ones that we expect.