School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Backend development using C++

Bogdan-Alexandru Ciurea

Submitted in accordance with the requirements for the degree of Computer Science MEng, BSc

2022/23

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (27/04/2023)
Backend GitLab Repository	https://gitlab.com/sc20bac/final-project	Sent to supervisor and assessor (25/04/2023)
Frontend GitLab Repository	https://gitlab.com/sc20bac/frontend-final-project	Sent to supervisor and assessor (25/04/2023)
Deployment GitLab Repository	https://gitlab.com/sc20bac/deploy-final-project	Sent to supervisor and assessor (25/04/2023)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Bogdan-Alexandru Ciurea

Summary

This project aims to demonstrate the benefits of using a language with better compile speeds for backend development. The resulting school management system will be subject to multiple non-functional requirements, including security, privacy, scalability, and reliability. The selection of a suitable database and the development of a basic frontend are critical components of this project.

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Nowadays, we are experiencing a shift in the backend development of the programming industry from programming languages that have faster computational speeds but are harder to write to languages that have slower computational speeds but are faster to create and also have features such as memory safety or the safety of not preparing for a buffer overflow attack. This project aims to prove that, although not ideal in some scenarios, it is better for a software in this field to use better compile speeds, and, to prove this, a backend software is going to be developed in the C++ language. Furthermore, this project is also aiming to prove that good coding practices and code quality are essential factors in software development.

The software that this project intends to build is a fully usable school management system/web application that is going to have multiple features such as managing users, files, courses, or announcements or assigning grades/results for different users. Furthermore, multiple non-functional requirements are applied to make the system more complex but also to better compare this software with other methods of writing such as software. The non-functional requirements are going to be the security of the system and the privacy of the data but, the most important will be the scalability and the reliability of our system.

The last two features are going to be achieved by choosing a suitable database that is going to fulfil the needs of having a database that is capable of storing large amounts of data, a big number of writes/reads per minute but also has a replication factor so that, in the case that the database is down for any reason, the data is not going to be lost.

Considering that this application is intended to be deployed and used by any person that does not necessarily have a background in computer science, the deliverable is also going to have a basic frontend to be used in the browser.

1.2 Background research

To achieve the goal of the project, it is important to conduct research in the fields of backend development, security, and software development [9]. Backend development is the part of software development that deals with the server-side logic of an application or website. It is responsible for handling tasks such as data storage, data retrieval, and authentication. Best practices in backend development ensure that the system is reliable, scalable, and secure.

Security is also an important aspect of this application and will involve protecting the system from unauthorised access, data breaches, and other cyber threats [9]. This way, the system will be secure from both internal and external threats. This includes measures such as access control, secure coding practices as well as data encryption. In the future chapters, this report is going to tackle multiple security measures taken into consideration for our application when choosing frameworks or when making design decisions.

The software development process will use the best practices in this field, these being the design of the application, the coding standards used, testing, and ease of maintenance. This will involve various methods and frameworks that will ensure that the software is of high quality and that it meets the requirement of the user [9]. Using these best practices will ensure that the software is robust, scalable, and maintainable.

API design will be another important aspect of our application that will determine how the system will be interacting with external parties. The API design best practices [6] will ensure that the API will be not only easy to use but also that the endpoints will be standardised so that the future usage will be intuitive but also that the information is secured and easy to gather.

The different tables[3], and the relations between them, is a crucial design decision that will, in the end, provide us with the best speed that we can harness from our system. In this project, the database chosen to store our information will be Apache Cassandra, which is a non-relational database [2] therefore, the relations between tables will be different from a traditional and more used, relational database like PostgreSQL or MySQL.

Moreover, parallel computation is another important technique used to improve the speed of software systems [11]. In this project, the information is going to be executed in parallel when receiving and processing the request as this process will ensure that the biggest computationally important task is executed in the fastest manner.

In conclusion, the project will utilise the best practices in backend development, security, and software development [9]. The chosen technologies and frameworks will be used to ensure the reliability, scalability, and security of the system. The API design will incorporate various security features to ensure the security of the system [10]. Finally, the database design will be optimised to ensure optimal performance [3], and parallel computation will be used to improve the speed of the system [11].

1.3 Existing Solutions

Currently, there are several existing solutions for school management systems that we can have as a source of inspiration. These solutions range from open-source software to software that is owned by a company and that requires a business-to-business collaboration to gain access to them. Some of the most popular open-source software solutions that can be found online are Open-School, Fedena, and SchoolTool, all of them giving access to features like user management, course management, attendance tracking, and grading methods. Furthermore, the proprietary solutions are giving access to more advanced features and even live support but come at a higher price tag. Some of the most popular solutions in this regard are Blackboard and PowerSchool, which offer their users features such as institution analytics and customise dashboards.

It is worth noting that the previously mentioned software solutions have gone through multiple versions of the final/intended product that the business has regarding its customers. Furthermore, they have a considerable amount of developers, quality assurance engineers, and dev-ops engineers therefore, it is expected that the end product will be of good quality.

In comparison to these existing solutions, the software developed in this project will aim to provide a lightweight, scalable, and secure solution that can also be easily deployable, thus making it the ideal solution for institutions that do not have the budget to spend on maintaining or developing their management application.

1.4 Requirements

Before commencing the development of our school management application, it is imperative to establish the functional and non-functional requirements that our system must meet. Some of these requirements have been derived from the functionalities of Minerva, the university's management website.

The objective of our application is to facilitate the management of school operations. To achieve this, our system will have three distinct types of users: administrators, teachers, and students.

The following functional requirements were identified:

- Users should be able to log in and log out of their accounts.
- Admins should be able to create new user accounts.
- Admins and teachers should be able to create, modify, or delete courses.
- Admins and teachers should be able to create, modify, or delete files within a course.
- Admins and teachers should be able to create, modify, or delete announcements.
- Users should be able to create, modify, or delete questions
- Admins and teachers should be able to create, modify, or delete grades.
- Admins and teachers should be able to create, modify, or delete tags.
- Users should be able to create, modify, or delete items in a personal to-do list.
- Student references should receive an email notification when a grade is added.

The following non-functional requirements were identified:

- The system must securely store and transmit data to avoid possible leaks of information.
- The system should be fast and maintain up-to-date data to avoid data overlapping.
- The application should be easily deployable.
- The data should be stored in multiple locations to achieve redundancy and prevent data loss.

1.4.1 Application features

Having the above functional requirements the application is going to have the next features:

1. Users will need to be able to log in and log out from the application. The users will be able to log in using their email and password as well as the school that they are assigned to.

2. In order to create an account, the admin will need to create an account for the user. The admin will not have access to the user's password (for security reasons), and the password is going to be sent to the user via email.

3. The application is going to support the usage of courses. They can only be created by admins or teachers. In order to assign other users to the course, the creator can add users individually or, for convenience, by tags.

4. All courses will contain different folders/files in them that will be able to be downloaded by other users. Note that the software will not support nested folders, therefore any folder will only contain files.

5. The application is going to support the usage of announcements. They can be created by admins or teachers and can be seen by other users if they are added to the list of users that can see the specific announcement (via tags). The announcements can also have files linked to them. Furthermore, other users that can view the announcement, can also ask questions if anything is unclear.

6. The application is going to support the usage of tags. Tags can be described as groups of people and only the teachers and the administrators will be able to assign/remove other users to/from this list.

7. The application is going to support the usage of grades. They can be assigned to a student by either an admin or a teacher. This feature will also be able to generate a final grade for the student. Note that, besides the actual grade that the user received, the grade will also have the option of "out of" but also weight. This way, some of the grades will be able to weigh more with respect to other grades.

8. In order for the software to also help the users to manage their tasks, the to-do list feature has also been added. It will function like a normal to-do list, creating tasks, moving them around (to change their status), and deleting them upon completion.

Because this application can be used by multiple schools, the decision has been made to build this application as a business-to-business service, that can be used as a subscription-based service, for different schools. Although this feature is not fully developed as of this moment, because it is not in the functional or non-functional requirements, the application will be able to support multiple schools, each with its own users, courses, etc. Because the "multiple schools" feature would mean that we would have to restructure our database, the decision was made to implement this feature now and leave space for improvement in a later stage of this project.

1.5 Chosen technologies and frameworks

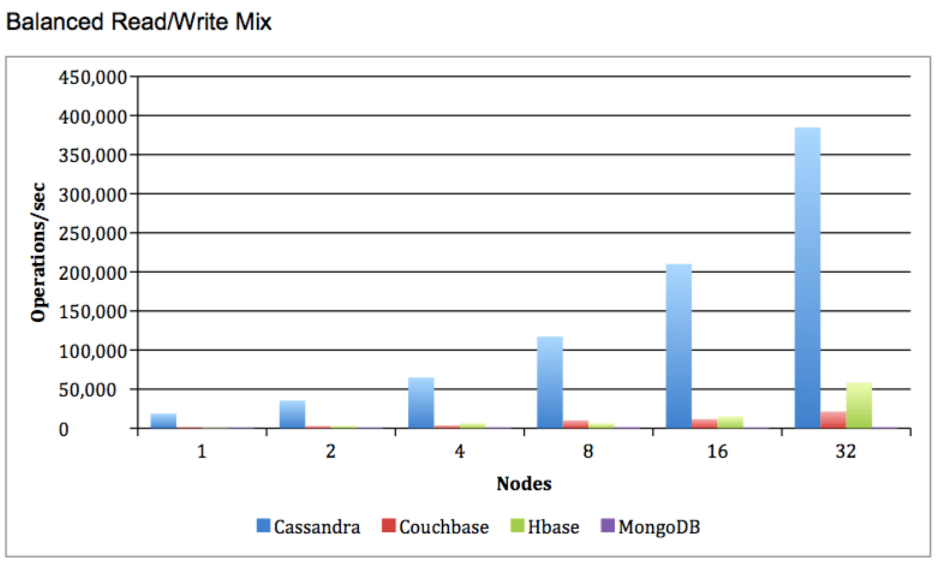
In order to achieve the requirements presented above, as well as the principles that have been identified in the "1.2 Background research" chapter, the end software will use the next technologies.

1.5.1 Database

The database chosen for this project is Apache Cassandra. The main reasons why this database was chosen for this project are:

1. Scalability and reliability. Cassandra clusters/nodes can be connected, and they will communicate between them so that the data is stored on every node. This will ensure that, if one node breaks down, the data will not be lost but will be spread around all the available nodes.

2. Speed. Compared to other databases, Cassandra is much faster especially when we are using it with multiple connected nodes as we can see from the article "Performance test on Cassandra NoSQL". The problem that we are going to run into is the slow read of Cassandra, but this can be overcome by structuring our data in a way that allows the database to execute the fast reads. [1]



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	18,925.59	1,554.14	973.85	1,278.81
2	35,539.69	2,985.28	3,430.59	1,441.32
4	64,911.39	3,755.28	6,451.95	1,801.06
8	117,237.91	10,138.80	6,262.63	2,195.92
16	210,237.90	11,761.31	15,268.93	1,230.96
32	384,682.44	21,375.02	58,463.15	2,335.14

Figure 1.1: Speed Comparison of different databases

3. How the data is stored. Cassandra is a non-relational database, and it is not stored in a JSON format, but as a table. Although we will not be able to use some of the features a relation database has (such as joins), Cassandra has an important feature: we can use maps, sets, and lists.

4. Compatibility with C++. The Cassandra cpp-driver made by Datastax is an easy-to-use, reliable, and tested driver used by multiple companies for their software.

1.5.2 Programming Language

To achieve this project's scope, the programming language that was chosen is C++. The main reasons are that C++ has one of the fastest compile speeds [5] but it also supports classes, compared to C.

On the other side, one of the main problems that we will have to take care of is the possibility of a heap attack, therefore making out system less secure and vulnerable to a hacker attack. This problem will be taken care of automatically, using the already implemented API Framework.

1.5.3 API Framework

To make the work process faster, the backend will use an already existing API framework. The one that was chosen for the project is the Drogon C++ library as it already has implemented features such as asynchronous programming, SSL communication, endpoint definitions that accept parameters, JSON formatting but also file download and upload. Some of the problems with this library are that the documentation does not go too deep into explaining some features, the integration with the browser CORS policy is lacking in functionality and it is also hard to deploy, as the building of the library is time-consuming but it also takes more than 1Gb of storage.

1.5.4 Other Frameworks used in the backend

In order to successfully achieve some of the functional and non-functional requirements previously mentioned, the application will need to have some additional modules added. For this the next libraries have been imported:

Name	Link	Functionality
Google tests	https://github.com/google/googletest.git	Used for testing the application.
SMTPMail	https://github.com/ihmc3jn09hk/SMTPMail-drogon.git	Used for sending emails.
Bcrypt	https://github.com/hilch/Bcrypt.cpp.git	Used for encrypting passwords
JWT-cpp	https://github.com/Thalhammer/jwt-cpp	Used for header encryption of the request

Table 1.1: Third-party libraries used in the project

1.5.5 Frontend

For the frontend part of the application, one of the most convenient libraries to build and deploy at the moment is Next.js. It is a library built on top of the React stack, therefore it will be able to use any React libraries. The reasons why this library was chosen are:

- Automatic routing. The routing of the browser pages is automatically done by the library.
- Server-side rendering. Next.js increases the security of the application (compared to base React) by automatically creating the pages in the backend rather than on the user's browser.
- Easy deployment. Because Next.js is built by the company Vercel, they have one of the best deployment experiences for any Next.js application. Even more, the deployment is also optimised for better run times.

Chapter 2

Methods

2.1 Backlog and UML diagrams

2.1.1 Overall architecture

The school management application will consist of three primary components: the database node(s), the backend server, and the frontend server. These components will be interconnected, with communication flowing from the frontend to the backend and then to the database.

Considering that we are not able to use technologies such as GraphQL or tRPC for connecting the frontend and the backend we will have to explicitly define the API endpoints and the data that they are going to require or respond with. Furthermore, for the backend to use the database, we will have to build from scratch a communication mechanism that would allow the backend data structures to be easily adapted to our database.

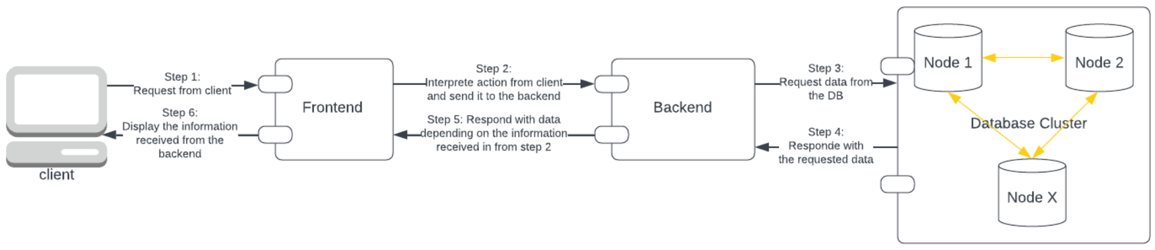


Figure 2.1: System overall architecture

The next chapters are going to explain in detail how the system is going to operate and how different classes are going to communicate in order to achieve the desired outcome.

2.2 Database

2.2.1 Database diagram

The tables will be split into two keyspaces (think of them as some kind of groups of tables). The **environment** keyspace will contain the tables schools, countries, and holidays_by_country_or_school. The rest of the tables will be stored in the **schools** keyspace. Unfortunately for us, the library that we are using will not automatically create tables that will manage the many-to-many/one-to-many/many-to-one relations (like Prisma for JavaScript or Django/Flask for Python), therefore, we will have to manually create these tables. For convenience, the naming rules will apply the next format: `<object>s_by_<object>`.

The next diagram is going to present the overall architecture of the database:

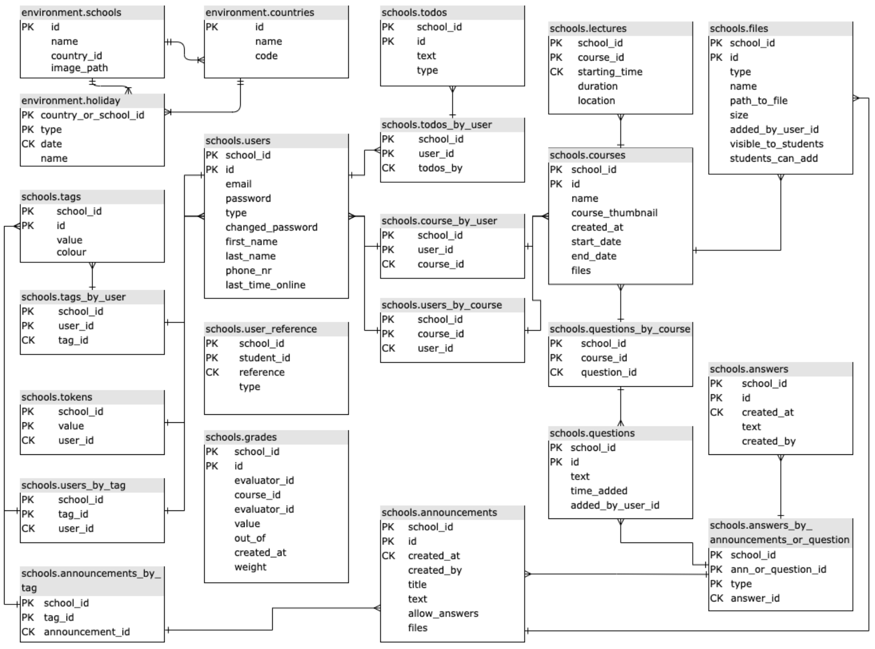


Figure 2.2: Database overall architecture and relationships

2.2.2 Database objects in C++

For most of the tables, there will be a C++ class that will store the correlated information from the database. This is done to keep the information easy to access from other classes that are used in the backend.

Database table	Correlated C++ class
environment.schools	SchoolObject
environment.countries	CountyObject
environment.holidays_by_country_or_school	HolidayObject
schools.announcements	AnnouncementObject
schools.answers	AnswerObject
schools.courses	CourseObject
schools.files	FileObject
schools.grades	GradeObject
schools.questions	QuestionObject
schools.student_references	StudentReferenceObject
schools.tags	TagObject
schools.todos	TodoObject
schools.user	UsersObject

Table 2.1: Correspondence between database tables and C++ classes

The classes will be simple and will have three main parts: the constructor and destructor, a function that will return the data in a JSON format, and all the variables that the class will have, these being all the fields that can be found in the correlated table.
The constructor will take as parameters a variable for each of the class's variables.
As mentioned, each of these classes will have a function named to_json() that will be responsible for returning the class's data in a JSON format so that it can be transferred from the backend

to the frontend.
Furthermore, all the variables will be stored as public variables of the classes to keep the code clean. Usually, in production, to get or set a variable, you would have to add two functions for each variable of a class: a getter and a setter, but, as mentioned above, we will prioritize keeping things simple and clear and not using this approach.

2.3 Database integration with C++

To explain exactly how the database integration was achieved, multiple classes have been created to make the transition from the Cassandra C++ library to our setup.

With these classes we wanted to achieve the next objectives:

1. Memory management done automatically. We wanted to do this mainly because the library that we are using for integration has some pointer variables that have to be allocated and deallocated before and after usage and, because it is C++, we can put these variables into a class so that it is the allocation and, more importantly, deallocation is taken care automatically by C++.

2. Error management. The library also has an absurd amount of error codes that can be processed from Cassandra that we will not need for this project. To solve this, the default Cassandra error codes have been merged in 8 error codes that would provide more understanding of what whent wrong if this is the case. Furthermore, in the developed code, there are two classes: ResultCode and CqlResult. The ResultCode class will receive an enumerator value of the respective error code, while the CqlResult will be a class that is returned from any database-related function and it will contain the ResultCode of the operation as well as the error's string from the database (if the response is OK, the string is null).

3. Table to object mapping. As pointed out in Table 2.1 we will have multiple objects that we want to populate with information from the database automatically. To do this, each of the classes in the table will have a correlated class that will read the information but, because we also have a table that maps the relations in our database, we will also have classes for each of the small tables that map the relation.

To achieve these objectives, we have come up with the next classes:

1. CqlClient - The Cql client will act as a wrapper around the library that we are using. Its purpose will be to execute the commands in Cassandra and to map returned results into objects. Its function will always return a CqlResult and, if any values want to be modified, the function will get a reference to that value.

2. CqlResult - The CqlResult will be a class that contains an error code and an error string (if there is no error, the error string will be empty). Note that the error code will be of the class ResultCode.

3. ResultCode - The ResultCode class will be a wrapper around the already existing error codes provided by the Cassandra library that we are using. Considering that the number

of error codes used by the library is too big for our project and that the naming convention is not that easy to understand, this class will merge some of the library's error codes into our error codes.

4. CqlManager - The CqlManager will be the most important class in the database, as every table in the above diagram (2.2) will have one CqlManager class managing it.

The next diagram is showing how the above classes are communicating with each other:

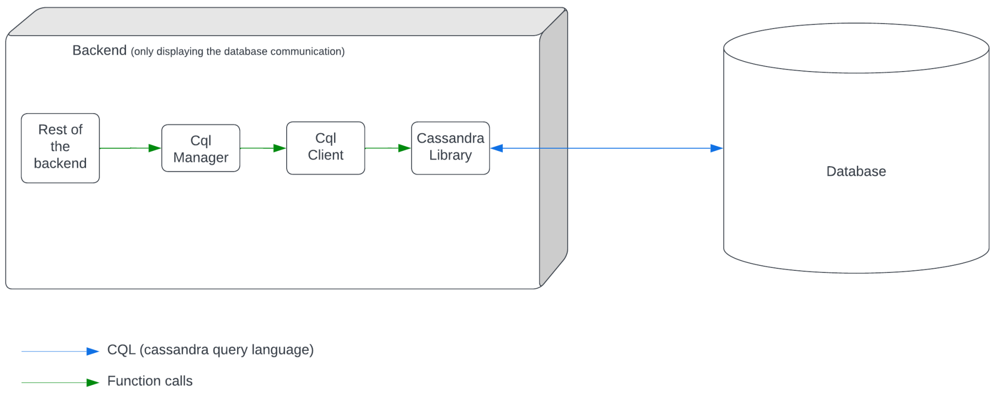

Backend (only displaying the database communication)
Rest of the backend
Cql Manager
Cql Client
Cassandra Library
Database
CQL (cassandra query language)
Function calls

Figure 2.3: Flow diagram for backend-database communication

For more details explaining the purpose of these classes and their functions but also any other functions/classes that have been created for this backend-database interaction, please read Appendix C.

2.4 Relationship managers

The purpose of the Relationship Managers section and associated classes in the backend is to validate data before sending it to the database, check user permissions, and generate the API response with HTTP code and JSON-formatted body. This is a critical and complex aspect of the backend that requires detailed information checks, as any invalid information should result in an error response. While the structure of these classes will be similar, this report will focus on providing detailed information about the `user_manager` class to illustrate the program flow and types of checks used. These classes do not use complex algorithms or programming principles but require meticulous attention to detail.

2.4.1 Integration with other features

The relationship managers will be located in the middle of the backend application. They will use the previously described CqlManagers and will be used by the correlated API manager that we are going to discuss in the next chapter. Keep in mind that every relationship manager will store pointers to multiple CqlManagers but it will be used by only one API manager. If we wanted to display this communication process between the classes we would get the next diagram:

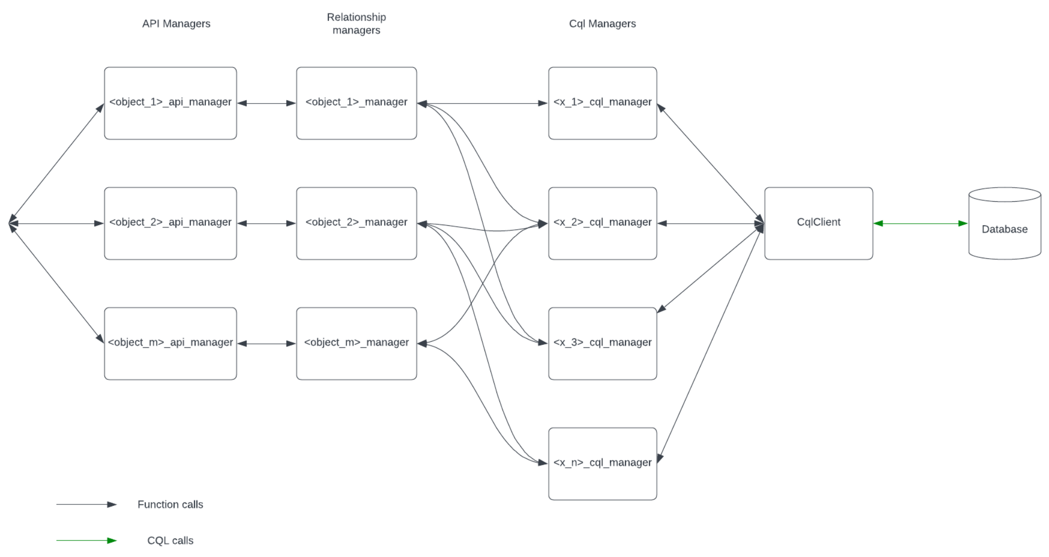


Figure 2.4: Flow diagram inside the backend

2.4.2 Authentication

Considering that we did not specify this in the previous chapters, this application, like any other stateless API, is going to provide the usage of tokens. These tokens are going to be stored in the database in the `schools.tokens` table and will be unique to the user. Furthermore, the database will also help us with security in this regard as it will automatically delete any token that was stored for more than 3 months. This information will always be important for our application as we will not get users (when we are doing authentication) based on their uuid but based on their token and their school id.

2.4.3 General structure of a relationship manager

Constructor

The constructor of each class will use multiple CqlManagers that have been presented in the precious chapter. It will take as parameters multiple pointers to such classes and will store them as private variables for further usage.

Functions

The general idea behind a function is to process the data and to check if all conditions are met, and if they are, add, remove, get, or update data. The return value of the functions is going to be a pair between a Drogon::HttpStatusCode (example k200OK) and a JSON value. Although this is not necessarily the most elegant way to do this, considering that these functions are going to be called in the API Managers to process the data but also for the final result, it would be a good idea to return the information in a way that does not require more processing. Leaving the return values as HttpStatusCode and JSON value would give us the ability for the API Managers to give the information to the caller without any more parsing.
The parameters that the functions are going to get will be the school id, a token that the user

is assigned, and other information that is needed depending on the case.
To make the process of reading the logic behind the functions easier, the algorithms applied are going to be written in a pseudo-code way. Keep in mind that the implementation in the below described functions is shortened to keep track of the most important aspects. The error checking is not described in the functions, neither is the returned value.

Possible returned values

The possible returned values from one of the classes' functions can be:

Case	HTTP_CODE	JSON
Everything went ok for GET	200 OK	The expected JSON value
Everything went ok for CREATE	201 Created	The newly created object in JSON format
Everything went ok for UPDATE and DELETE	200 OK	null
There was an error when reading the information	500 Internal Server Error	Specific description for the error in an "error" field of a JSON
Fields are incorrect or given parameters are not found (i.e., the token)	400 Bad Request	Specific description for the error in an "error" field of a JSON
Action is not allowed because of the user type	403 Forbidden	Specific description for the error in an "error" field of a JSON
Some parameters that are given could not be found in the database	404 Not Found	Specific description for the error in an "error" field of a JSON

Table 2.2: HTTP codes and JSON responses for various cases

2.4.4 User manager

The user manager, as the name suggests, is going to be the class that will manage the creation/deletion/modifications of a user but will also take care of their tokens.

Functions

The functions that the user manager is going to have as public functions that are going to help the API manager in the future are:

- create_user: It is going to create a user with the related fields given as parameters.
- get_user: It is going to return all the information about a user in JSON format.
- get_all_users: It is going to return all the users in that school.
- update_user: It is going to update a user's fields.
- delete_user: It is going to delete a user.
- log_in: Log in.
- log_out: Log out.

When calling one of these functions, the backend is going to perform multiple checks in order to prove that the user that sent the request, has access to that information or to perform that action. One example can be the `create_user` function. It will perform the next checks before adding the actual user to the database:

1. Check if the school exists.
2. Check if the creator token is valid.
3. Check if the creator user is an admin.
4. Check if the user's email that we want to add is already used by other users in that school.
5. Then we can add the user to the database and respond to the API managers with `200 OK` and the password that was generated by the backend. The password is not going to be sent back as a response but it will be used to be sent as an email to the above-specified email.

2.4.5 Other object managers' functions

The other functions that are going to be defined in other object managers as well as their purpose are going to be presented in the next chapters.

Announcement Manager

This class will take care of managing the school's announcements. This also includes managing the tags of that announcement, managing its files, and also its answers.

The correlated functions that this class is going to have are:

- create_announcement: This is going to create the announcement in the database.
- get_announcements: This is going to get the announcements that the user has access to.
- delete_announcement: This is going to delete the announcement with the given id.
- add_tag_to_announcement: This is going to add the tag to the announcement.
- get_announcement_tags: This is going to get the tags that are attached to the announcement.
- remove_tag_from_announcement: This is going to remove the tag from the announcement.
- create_announcement_file: This is going to add the file to the announcement (just the fact that the announcement has the file attached to it, not the file itself).
- delete_announcement_file: This is going to delete the file from the announcement. Not the actual file, just the information about it.
- has_permission_to_get_file: This is going to check if the user has access to the file.
- create_answer: This is going to create the answer in the database.

- get_answers: This is going to get the answers to the specific announcement.
- delete_answer: Is going to delete the answer with the given id.

Course manager

This class will manage a school's courses. It will also manage its users, its thumbnail, its files, its questions, and their answers.

The correlated functions that this class is going to have are:

- create_course: Create a new course.
- get_course: Get a course's information.
- get_all_user_courses: Get all the courses of a user.
- get_courses_users: Get all the users of a course.
- update_course: Update a course's information.
- delete_course: Delete a course.
- set_course_thumbnail: Set a course's thumbnail.
- get_course_thumbnail: Get a course's thumbnail.
- delete_course_thumbnail: Delete a course's thumbnail.
- create_course_file: Create a new file for a course.
- get_course_files: Get all the files of a course.
- update_course_files: Update a course's file.
- delete_course_file: Delete a course's file.
- has_permission_to_get_file: Check if a user has permission to get a file or folder.
- add_users: Add users to a course.
- remove_users: Remove users from a course.
- create_question: Create a new question.
- get_questions_by_course: Get all the questions of a course.
- delete_question: Delete a question.
- create_answer: Create a new answer.
- get_answers: Get all the answers to a question.
- delete_answer: Delete an answer.

Environment manager

This class will manage all the tables in the environment keyspace (schools, countries and holidays).

The correlated functions that this class is going to have are:

- create_school: Creates a school in the database.
- get_school: Gets a school from the database.
- get_all_schools: Gets all schools from the database.
- update_school: Updates a school in the database.
- delete_school: Deletes a school from the database.
- create_country: Creates a country in the database.
- get_country: Gets a country from the database.
- get_all_countries: Gets all countries from the database.
- update_country: Updates a country in the database.
- delete_country: Deletes a country from the database.
- create_holiday: Creates a holiday in the database.
- get_holidays: Gets holidays from the database.
- delete_holiday: Deletes a holiday from the database.
- delete_holidays: Deletes holidays from the database.

Tag Manager

This class will manage a school's tags. The correlated functions that this class is going to have are:

- create_tag: Create a new tag.
- get_tag: Get a tag by id.
- get_all_tags: Get all tags in a school.
- update_tag: Update a tag.
- delete_tag: Delete a tag.
- create_tag_user_relation: Create a relation between a tag and a user.
- get_users_by_tag: Get all users that have a relation with a tag.
- get_tags_by_user: Get all tags that have a relation with a user.
- delete_tag_user_relation: Delete a relation between a tag and a user.

Todo manager

This class will manage all of an user's todos. The correlated functions that this class is going to have are:

- create_todo: Create a new todo
- get_todo: Get a todo
- get_all_todos: Get all todos of a user
- update_todo: Update a todo
- delete_todo: Delete a todo

Todo manager

This class will manage all of the grades in our system. The correlated function that this class is going to have are:

- add_grade: Created a new grade
- get_personal_grades: Will return all grades that the user making the request has.
- get_user_grades: Get all grades that a specific user has.
- get_course_grades: Get all grades that are assigned to user in a specific course
- edit_grade: Updates a grade
- delete_grade: Will delete a specific grade

2.5 API Mapping and request information

The API managers are the part of the backend that is going to interact with the frontend and the most important part when it comes to security as it will take care of any possible attacks that come from external sources.

2.5.1 Integration with the rest of the backend

As described before, each action (adding a user, for example) will have its function defined in the Relationship Managers that will ask for specific information to be passed as parameters. All of these parameters are going to be received and formatted by the API managers so that they are valid (from a type point of view) but also all of them are present in the request. For a better understanding of how the classes communicate, review figure 2.4.

2.5.2 Security

The security of this application is going to be implemented at 3 different points in the program:

1. SSL communication. The application, because of the Drogon library, supports SSL certificates to make our application more secure when transporting the information from a client (normal client or our frontend) to our backend and back. Unfortunately, when running this application on `localhost`, the frontend will not work as intended and will result in it, not being able to call it; this problem only appears in the frontend, as, when running tests with Postman, everything works as planned.

2. Encrypting password. The password encryption process will mainly take place in the `user_manager` class because it will manage the password generation and its encryption but, because we want to send an email to the user with his password, this information is going to be returned from the `user_manager` to the `user_api_manager` for the email to be sent. The process of generating the password is going to be simple: we will create an 8-character long password containing characters from the Latin alphabet, digits, and special characters; keep in mind that here, because of the pseudo-randomness of the basic `random()` function that C and C++ provide we would run into security problems [12], we will use the `std::random_device` object provided by C++ for better randomness of our numbers. The next process would be to hash this newly created password; for this, we are going to use the library in table 1.1, Bcrypt, and the output of this encryption is going to be stored in our database. By doing these steps, the users' passwords are going to be stored securely in our system.

3. JWT standard. As mentioned before, the backend will expect a token (in most cases) that would grant access to the user. To not display the user's actual token and the school's id that the user is assigned to, the backend will encrypt it in the JWT standard [8]. For this to happen, the application is going to use the `JWT-cpp` library. This library is going to be used in two ways: first when the user wants to log in and second, for any other integration with other endpoints. When logging in, the school's id and the user token are going to be encrypted with our public key and, when logging out, we are going to first check if the request has the appropriate header (in our case `Authentication: Bearer <token>`), and then get the encrypted values from the token's fields.

2.5.3 API Design

The API design is going to respect the standards in the industry [10]. Because it would be hard for us to read the information that each endpoint requires (as we have more than 70 endpoints on our application), only the available endpoints and what they should do will be listed without explaining exactly what information should be in them. A list of all the endpoints that our application is going to have can be found in Appendix D.

2.6 Frontend

For the frontend to be developed, multiple libraries have been taken into consideration but, as already mentioned in the first chapter, the Next.js library was chosen for this scope as it has automatic routing as a default option and also has server-side rendering, therefore the security of our application will be increased. Another option that was chosen for the used stack of the frontend was the programming language; because we had two options JavaScript and Typescript, the decision was made to choose the latter in this case, as it provides better syntax and type safety of the code, therefore eliminating the possibility of any errors of this manner. The purpose of this feature is to make the inner workings of the backend visible to the end user but also to display the information more interactively so that the functional requirements are met. In practice, to achieve this, the frontend will call one or more specific API endpoint(s) of the backend and will parse the information so that it is displayed to the user but, because we also have information that can be modified on the screen, we also want to call these endpoints when we want to do any modifications to the displayed objects.

The development of this feature was, in principle, an easy process but, along the way of coding this feature, the communication between the backend and the frontend server was put to a put to hold because all of the requests that would have come from the client's browser, where stopped by the CORS policy [6] of the browser, therefore a countermeasure has been developed. To send the requests from the user's browser to the backend, the frontend server will also have the feature of receiving requests that are then forwarded to the backend. This feature, although not the most elegant way of resolving this, has boosted the security of the entire system as it adds another layer of authentication needed for the user to communicate with the backend. Therefore, the overall architecture of the communication will be represented in the next diagram:

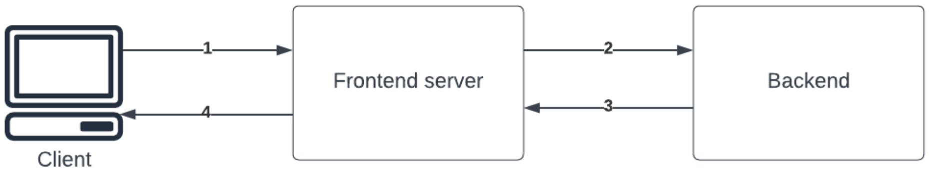


Figure 2.5: Flow diagram of client-backend communication

2.7 Deployment

Because our application is so large, the deployment of the application is going to require a lot of preparation before actually building and then deploying it.

As described in the previous chapters, the application is going to have 3 different parts: the database, the backend, and the frontend, each with its different deployment procedures.

To make the deployment of the **database** easier, 2 different methods of deployment have been created, each requiring `docker` to be installed on the local machine. The first one will be using the `run-cassandra.sh` bash script, which will deploy a single container on the local machine running the `cassandra:latest` image that will be downloaded from the official docker website. The second one, and probably the one that we use to imitate the scalable and reliable environment that we talked about in the requirements, would be to use the `docker-compose.yml` file in the backend's root folder. To run it, it would require executing `docker-compose up -d` command in the terminal; this way you would get 3 nodes that communicate between them. Please keep in mind that the setup of the database will take around 5 minutes for the nodes to be fully functional.

To deploy the **backend**, you would want to first have all these dependencies installed `gcc, git, g++, cmake, make, build-essential, libuv1-dev, libjsoncpp-dev, uuid-dev, openssl, libssl-dev, zlib1g-dev` on your machine. Then, because all of the libraries are not stored locally in our repository but as git submodules, you would want to download them by executing `git submodule update -init -recursive` in the terminal. After that you can continue with the actual building of the program: **mkdir build && cd build && cmake .. && make**. This way, the application is going to be built under the name of `./api_server` and will be located in the `build` folder of the repository. By default, the application is going to listen by default in **http://localhost:8080** or **http://127.0.0.1:8080** but you can also change its address and port if so chose, but this is not recommended. Keep in mind that the application will exit if there is no Cassandra cluster running on port 9042. To make this process easier to deploy, the creation of a dockerfile would be able to build a container that will contain the application and can be downloaded from Dockerhub.

To build the **frontend**, the process is going to be much simpler. After cloning the repository, just execute the basic npm commands to run a Node application: `npm install && npm run dev`. This way, the frontend is going to be available on **http:localhost:3000** and can be accessed from your browser.

This being said, the deliverable will also contain a third project, containing a deployable docker-compose file that will make the process of setting up all of the 3 different systems available in just one executable file but, for this to be possible, the user will need to have installed git and docker-compose on his local machine. Furthermore, because Cassandra has the option to import/export data inside the database using CSV files, this executable will also initialise the database with some data.

2.8 Software development

For this project to be developed multiple software development organizational methods have been used.

2.8.1 Work distribution

The first one would be the work distribution of the project. Because, although not difficult to understand, the project was very time-consuming to develop, making only the backend have around 55.000 lines of code, the project was divided into multiple sprints. To be sure that enough time is allocated to each feature and that we will not have any surprises along the way, the project has been split into 5 sprints as the below table presents:

Sprint period	Goal that the sprint wanted to achieve
1st November 2022 – 15th November 2022	Project setup and definition of features
16th November 2022 – 31st December 2022	Database Setup and Integration
1st January 2023 – 31st January 2023	Relations Managers
1st February 2023 – 28th February 2023	API Definitions
1st March 2023 – 31st March 2023	Frontend Work
1st April 2023 – 30th April 2023	Testing, bug fixing, and finalizing the project for release

Table 2.3: Project sprint periods and goals

2.8.2 Version control

The second one would be the version control of the project. For this, the GitLab account provided by the university was used. While branches are not mandatory for this project, only one person was working on the code and committing to the repositories, their usage had come in handy for the backend repository to provide an easier understanding of the commit grouping and, for the commit naming, professional commit naming methodology has been used, each briefly explaining what that commit is adding.

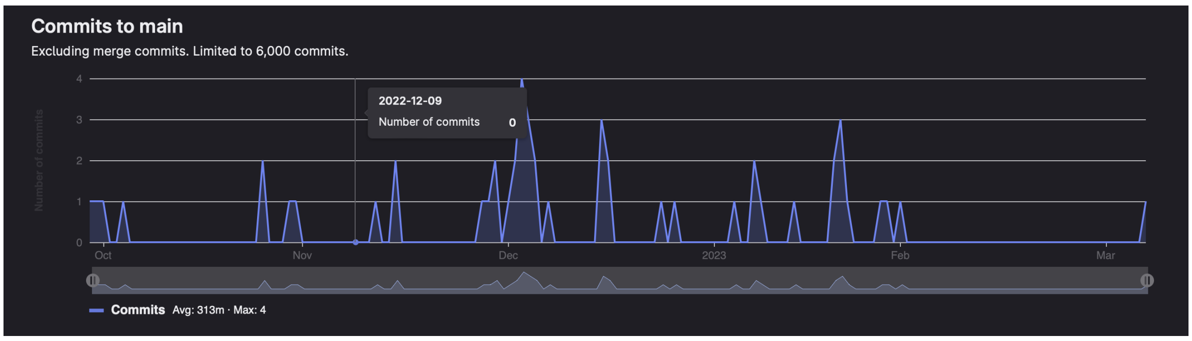


Figure 2.6: Commit history of the backend

Chapter 3

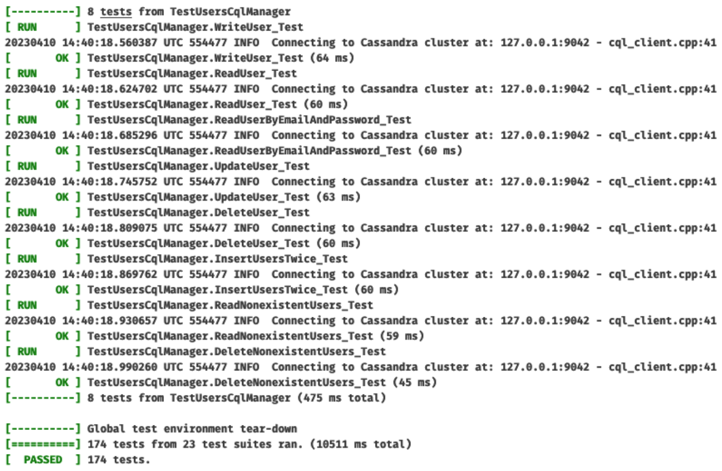
Results

The result of our application will be listed in this chapter and will tackle each of our functional and non-functional requirements. All of the results be either tested by the backend (using the Google tests library mentioned in chapter 1.5.3) or will be tested using external features.

3.1 Database Testing

To test the database we have used, as mentioned before, the Google tests library. Unfortunately, the library has a drawback, that is, before each test, the backend will have to connect to the database. This is not a problem because here, we are not testing the speeds of the application, but rather that every feature works as intended.

By doing this, 174 tests have been created for the CqlManagers (the 24 classes mentioned before) that will tests if the information is stored and gathered correctly.



```
[----------] 8 tests from TestUsersCqlManager
[ RUN      ] TestUsersCqlManager.WriteUser_Test
20230410 14:40:18.560387 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.WriteUser_Test (64 ms)
[ RUN      ] TestUsersCqlManager.ReadUser_Test
20230410 14:40:18.624702 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.ReadUser_Test (60 ms)
[ RUN      ] TestUsersCqlManager.ReadUserByEmailAndPassword_Test
20230410 14:40:18.685296 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.ReadUserByEmailAndPassword_Test (60 ms)
[ RUN      ] TestUsersCqlManager.UpdateUser_Test
20230410 14:40:18.745752 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.UpdateUser_Test (63 ms)
[ RUN      ] TestUsersCqlManager.DeleteUser_Test
20230410 14:40:18.809075 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.DeleteUser_Test (60 ms)
[ RUN      ] TestUsersCqlManager.InsertUsersTwice_Test
20230410 14:40:18.869762 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.InsertUsersTwice_Test (60 ms)
[ RUN      ] TestUsersCqlManager.ReadNonexistentUsers_Test
20230410 14:40:18.930657 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.ReadNonexistentUsers_Test (59 ms)
[ RUN      ] TestUsersCqlManager.DeleteNonexistentUsers_Test
20230410 14:40:18.990260 UTC 554477 INFO  Connecting to Cassandra cluster at: 127.0.0.1:9042 - cql_client.cpp:41
[       OK ] TestUsersCqlManager.DeleteNonexistentUsers_Test (45 ms)
[----------] 8 tests from TestUsersCqlManager (475 ms total)

[----------] Global test environment tear-down
[==========] 174 tests from 23 test suites ran. (10511 ms total)
[  PASSED  ] 174 tests.
```

Figure 3.1: Cql classes test results

Keep in mind that, when testing, the script will delete all information in a table so that we do not get any not needed information. This will be a problem if you want to run the tests on the database that you have the application information so you would have two options to keep that data:

1. Export the information before testing.

2. Stop the containers that run the application and start a new node on the same port running Cassandra. This way you would not get any errors and your information would not be lost.

Declaimers

A big problem that appeared during production was that, after deploying a new container, the testing part lasted more than expected because, for the backend to connect to the database, the connection time was around 5 seconds, when it should be around 0.5 seconds maximum.

Furthermore, even some tests ran into problems when automatically tested but ran perfectly when the tests were done manually. This problem is expected to appear because the database did not have time to finish all the background processes or because of Docker and how it works: because there was a stopped container on the same port, when created, the new container will look as if that port is used (and it is) and will create a new network on that port.

Furthermore, the database also responded with "All hosts in current policy attempted and were either unavailable or failed" when running the production software exactly because of this new database problem.

3.2 API Testing

The API testing was done externally from the backend, using the Postman application. With it, we stored each request in a group, each group representing one of the API Manager classes in our backend, and each request had the required information for each action of the backend.

By doing this, all endpoints listed in tables D.1, D.2 and D.3, therefore having 11 endpoints for the announcements, 5 endpoints for the countries, 11 endpoints for the courses, 6 endpoints for the grades, 3 endpoints for the holidays, 5 endpoints for the schools, 9 endpoints for the tags, 5 endpoints for the todos and 6 endpoints for the users.

Because the actual endpoints, what headers they want to receive, what body they want to receive, and what response they are going to give is going to be hard to format and time-consuming to do for every endpoint, considering that we have 61 endpoints, all of this information is going to display in these tests. This way, the information is going to be easier to read.

All of these tests in Postman have been exported and to import them you can build a new project into Postman and then import them from `tests/api_managers/` path of the repository.

3.2.1 API speed

Considering that this project's most important non-functional requirement is the speed of the API, the endpoints test is the one responsible for testing this feature. The endpoints, as pointed out above, have been tested with Postman, therefore, because of the features provided by this software, we are also able to see the response time of the backend. The results that have been achieved when testing on localhost, Postman got the next responses:

- For the **POST** request, that would create a new entry, the backend responded in 35 to 60 milliseconds.
- For the **GET** request, that would return information to the requester, the backend responded in 30 to 80 milliseconds. This number can vary depending on the amount of information. For example, when having around 100 entries of one type, the response was around 300 milliseconds but that is a special case when larger than normal amounts of information are being processed; but, in some cases, the first response was returned in 33 milliseconds and the next ones have been returned in 7 milliseconds. To test all of this, the automated "Run collection" function provided by Postman has been used and, because of this, the requests that last less than 100 milliseconds, will keep the connection open for faster communication with the backend.

- For the **PUT** request, that would update an entry's data, the backend responds in 40 to 80 milliseconds.
- For the **DELETE** request, that would delete an entry in the database, the backend responses in 30 to 70 milliseconds.

Having these results, the backend can now be compared with other services of this kind. According to Google [7], the server can be considered good if the response time is under 200 milliseconds, if the response time is bigger, you would want to achieve the 200 milliseconds speed. Furthermore, another article [4] would add to this by categorizing the response times into 4 categories:

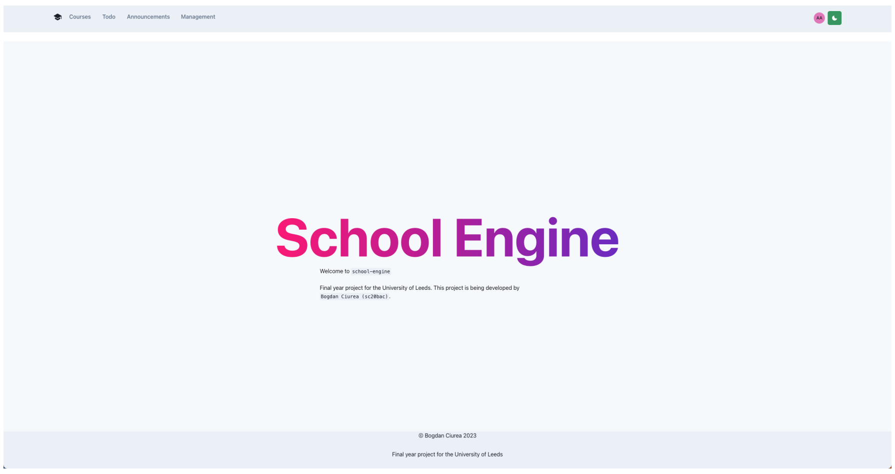
- Excellent, when the response time is under 100ms,
- Good, if the response time is between 100ms and 200ms,
- Should be improved, if the response time is between 200ms and 1 second, and
- Too slow, if the response time is above 1 second.

Having the above-mentioned result, the objective of the application, that being the creation of a fast and reliable backend, has been achieved successfully.

3.3 Frontend

The design of this feature is the most important part of the aspect of the frontend therefore, the decision was made to apply a simple but easy to understand user interface that would make the end user understand where all the information is and how to access it.

The bellow screen shot displays the principles enumerated above, having a simple header (that also changes depending on the type of user), main body (that will display the actual information), and a simple footer.



Welcome to `school-engine`

Final year project for the University of Leeds. This project is being developed by `Bogdan Ciurea (sc20bac)`.

© Bogdan Ciurea 2023

Final year project for the University of Leeds

Figure 3.2: Index page of the frontend

Chapter 4

Discussion

4.1 Conclusions

Although not mentioned before, the most important aspect of this project was mainly the good management of the code. Considering that the result had more than 55.000 lines of code, the project became, over the development process, hard to go throw but, because of the way everything was organised and because of the steps taken before the coding period, the result was easy to comprehend and, more importantly, manageable in the end. By using this report as a "High-level design" of the application, this project was successfully described in an easy-to-understand manner, describing only the bigger picture, thus further proving the point of the usage of good coding practices and project management.

In addition to good coding practices, the coding process also emphasised testing and code review. Code reviews have been applied constantly to ensure that the process achieved its intended goal and also to ensure that the code adhered to the coding standards applied in the industry. This, along the way, provided essential to the result as it ensured that the quality of the product was met and that future maintenance is minimal.

In conclusion, the project was successful in achieving the objective, which is, the implementation of a robust backend system using C++ and good software design principles. Therefore, by putting into action the best programming practices and design decisions, the project gave a maintainable, scalable, and efficient backend system that met the requirements of the project.

The success of this project can be attributed to several factors such as the initial planning and design of the general application(explained in the 2nd chapter), the usage of a robust programming language that, although not providing memory safety, was the best option for this project's aims, and the focus on using practices such as encapsulation and modality, therefore splitting the application into as many parts as possible to ensure that testing could be done individually and the code to be as readable and easy to understand as possible.

Overall, this project is an excellent example of good software design and how C++ can be used to create a robust and scalable system.

4.2 Disclaimer

One of the problems that this entire application (database, backend, and frontend) has, and that was often run into during deployment, is the communication with the database. As mentioned before, the Cassandra database has some apparent problems when running in a Docker container, thus making the process either run slowly or making the mutation of the tables impossible, these modifications or addition of data were unusable. To solve this problem, the solution that was found was to first deploy the docker container running Cassandra, then run the backend, and after that upload the dummy data that was provided. This problem was not replicable from the tests that have been made when deploying so this problem may or may not occur when testing on the local environment.

4.3 Ideas for future work

From the project's start, this school management system was implemented as a business-to-business software that would help multiple schools to manage their data, which is also another reason why the software had such strict requirements. Furthermore, the project has some additional information in its database, to provide access to additional features to be added in the future. This can be seen very clearly in the database as, in the `environment` keyspace, the table `holidays` is not used; this was done on purpose to leave space for further development of a personal calendar of the school that would after it's implementation, be able to provide each student with its lectures on a specific date.

Furthermore, the development of an automated calendar generator would also be a viable option that would be capable of generating lectures for a specific course without overlapping lectures. This being said, it would be a difficult task as it is a requirement for this system to be customise therefore imposing two requirements: formatting the conditions imposed by the admins but also making the admins understand how they work and the backend to be able to parse and process this information.

Adding to this, the application can also support the creation of more complex formulas for calculating the final grade because different schools might have different requests regarding grading, thus this feature would be needed.

Another feature that would be added in the future is the ability for a global/company administrator to manage the environment and will be able to manage the system's data. This way, the software can be displayable on an online platform and be used as an actual business-to-business product that would be capable of generating revenue.

Bibliography

[1] Performance test on cassandra nosql, Sep 2018.

[2] A. Cassandra. Apache cassandra documentation, Accessed: April 4, 2023.

[3] Database.Guide. The 3 types of relationships in database design, Accessed: April 4, 2023.

[4] Datadome. How to reduce server response time. Datadome Learning Center, 2022. [Online; accessed 6 April 2023].

[5] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *BMC Bioinformatics*, 9(1):82, 2008.

[6] Google. Apigee cors policy.

[7] Google Developers. Server response time. Google Developers website, 2023. [Online; accessed 6 April 2023].

[8] IETF. JSON web token (JWT). RFC 7519, 2015. [Online; accessed 6 April 2023].

[9] R. C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall, 1st edition, 2017.

[10] G. C. Platform. Design patterns - api design guide, Accessed: April 4, 2023.

[11] H. T. Takes. Parallel computing and its modern uses, Accessed: April 4, 2023.

[12] J. Viega and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More.* O'Reilly Media, Inc., 1st edition, 2003.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

The project, as of this moment, is just half of what it can be. Although it has some very intuitive features, it lacks other ones like timetable management and some administrative features for the whole application (a management platform for the business administrator). Furthermore, the frontend part of the application, although not the main point of this application, is lacking functionality and does not use all of the endpoints opened in the backend.

The positive part of the application is that it implements multiple principles found in the industry such as data redundancy, multiple layers of security, scalability, and how easy it is to deploy. Furthermore, the approach that was used for this project, was able to foresee any possible code-related issues.

A.2 Personal reflection and lessons learned

When discussing account the project in terms of code development, the project can be considered a success as the goal of using C++ for backend development and the problems that come with this has been achieved. The extensive amount of code, which exceeded $50,000$ lines, was managed effectively due to thorough planning and organisation before the coding phase. The step-by-step process of developing each feature made the coding period intuitive, with each piece falling into place perfectly to achieve the desired outcome.

However, the challenges that appeared have taken place in two parts. The first one was the Database. As mentioned before, the Cassandra docker container proved to be unusable sometimes and the results were different when deploying it in a fully-fledged environment. The other challenge that appeared came as a result of the large amounts of code generated in the end; because of this, the explanation process proved to be extremely difficult if the report wanted to go more in-depth on some features.

Aside from technical skills, the project also provided valuable insights into project management and software architecture. The importance of effective project organisation was emphasised, as well as the value of spending time in the planning phase to design the software architecture. Overall, the project provided a wealth of learning experiences beyond just technical development.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

The legal issues that this application will be concerning the right to intellectual property, privacy, and data-safety laws. Therefore, this application must not violate any patents therefore all of the code and designs must be legally licensed but will also have to comply with the regulations in place regarding the data protection and privacy of our users.

Furthermore, the developer will be held responsible for any loss of data or the damage that it suffers, thus making the developer liable for such incidents.

A.3.2 Social issues

The social issues that this application will have are concerning accessibility, inclusiveness, and user safety. Because of this, the application will have to be designed in such a way that it allows users with different special needs or disabilities to have access and to successfully use this application. Also, the application will not discriminate against any user regardless of gender, race, religion, or any other biased characteristic of our users therefore, any act of discrimination will be reported and taken care of with grates priority.

Furthermore, the user's information must be the top priority, especially when dealing with sensitive information (i.e. passwords or emails).

A.3.3 Ethical issues

The ethical issues that this application will have will be privacy and the non-discrimination policy. Therefore, the application will have to respect the privacy of the users and ensure that it is protected from unauthorised access but the school's information (as an institution) will also fall under this privacy and data protection policy. Furthermore, as pointed out before, the application will not tolerate any discriminatory action against any user regardless of gender, race, religion, or any other biased characteristic of our users, thus severe actions will be taken to ensure that this policy is respected.

A.3.4 Professional issues

Because this application is only developed by one person, the professional issues are going to be similar to the best coding practices pointed out in earlier chapters, mainly the code quality (the usage of comments, standardised formatting, etc.) and code testing.

Appendix B

External Material

For this project, multiple libraries have been used in order to achieve the end result. Bellow, is a list of all dependencies used in this project.

Name	Link	Functionality
Drogon	https://github.com/ drogon-framework/drogon	Used as API framework
Google tests	https://github.com/ google/googletest.git	Used for testing the application.
SMTPMail	https://github.com/ ihmc3jn09hk/SMTPMail-drogon.git	Used for sending emails.
Bcrypt	https://github.com/ hilch/Bcrypt.cpp.git	Used for encrypting passwords
JWT-cpp	https://github.com/ Thalhammer/jwt-cpp	Used for header encryption of the request
Apache Cassandra	https://cassandra.apache.org /_/index.html	Used Database
Cassandra C++ driver	https://github.com/datastax /cpp-driver	Used for C++ to Database communication
Next.js	https://nextjs.org/	Used as frontend framework
Chakra UI	https://chakra-ui.com/	Used as component library for the frontend
react-dnd	https://react-dnd.github.io/ react-dnd/about	Used for frontend utilities
Docker	https://www.docker.com/	Used for deployment and containerisation

Table B.1: Third-party libraries used in the project

Appendix C

Database classes

C.0.1 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 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 decided to merge some of them.

C.0.2 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.
Both 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.

C.0.3 Structs and Smart pointers

As pointed out in the requirements, the database library, when we call some specific functions, will automatically allocate memory for us to use. To keep the program safe, and not leak memory, we are given some default functions that will free this 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.

C.0.4 CqlClient

The CqlClient will be the most important class that will communicate with the database. It is responsible for taking the information, sending it to the database, and processing 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 in the private fields of the class.

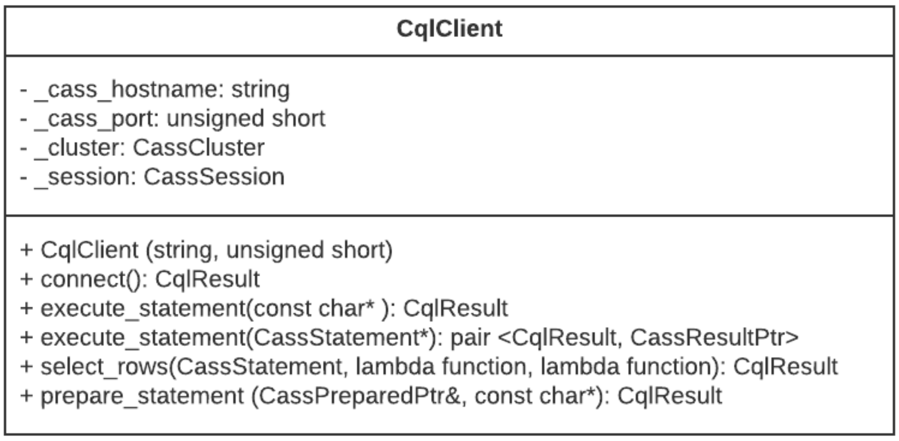

CqlClient
- _cass_hostname: string
- _cass_port: unsigned short
- _cluster: CassCluster
- _session: CassSession
+ CqlClient (string, unsigned short)
+ connect(): CqlResult
+ execute_statement(const char*): CqlResult
+ execute_statement(CassStatement*): pair <CqlResult, CassResultPtr>
+ select_rows(CassStatement, lambda function, lambda function): CqlResult
+ prepare_statement (CassPreparedPtr&, const char*): CqlResult

Figure C.1: Cql Client UML diagram 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 that 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 a parameter. I have also given the liberty of receiving a string as a 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 in the database, we want to see if that field was 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 a parameter, two functions: one that allocates the memory for the new data and another one that will take the row as a parameter and add the data to the previously allocated memory. Furthermore, we will also have a parameter that will indicate if we 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 to prevent breaches such as an SQL injection attack.

C.0.5 Global database related 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 enumerator ResultCode. 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	RESOURCE_ERROR
CASS_ERROR_SERVER_FUNCTION_FAILURE	RESOURCE_ERROR
CASS_ERROR_SERVER_WRITE_FAILURE	RESOURCE_ERROR
CASS_ERROR_SERVER_UNAVAILABLE	UNAVAILABLE
CASS_ERROR_SERVER_WRITE_TIMEOUT	TIMEOUT
CASS_ERROR_SERVER_READ_TIMEOUT	TIMEOUT
CASS_ERROR_LIB_REQUEST_TIMED_OUT	TIMEOUT
Anything else	UNKNOWN_ERROR

Table C.1: Correspondence between database tables and C++ classes

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 reading some variables from 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

When reading from Cassandra, the data is going to come separated into multiple columns, each with a specific data type (like integer, string, or array). To read them, 7 different functions have been created to read the types of data that we are using in our data types. Each one of them will take as parameters a CassRow pointer, the column index (in which the data is stored), 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 from that column.
To read the variable that we want, we will use the `cass_value_get_<field>` function provided by the Cassandra library.
The functions that we will use will be:

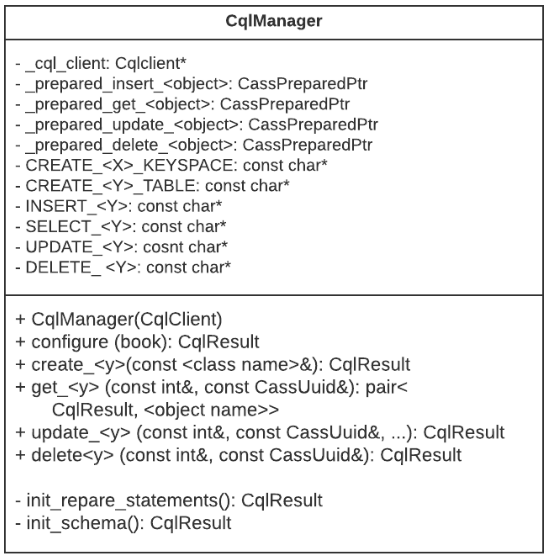
Function name	Function scope
get_bool_value()	Will read a bool value from a column.
get_int_value()	Will read an int value from a column.
get_long_value()	Will read a long value from a column. Usually used for reading time_t variables. Note that Cassandra also stores the milliseconds of a time, therefore 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 database).

Table C.2: Function scope for reading values from a column

C.0.6 Cql Manager classes

The main interaction with the database is going to be done through these classes. Each table in the database is going to have a correlated Cql Manager class. 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 into 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 will all have the same code structure and apply the same principles.

To get a first impression of the class that we are going to describe in the next chapters, the below-displayed diagram represents it.


```
CqlManager

- _cql_client: Cqlclient*
- _prepared_insert_<object>: CassPreparedPtr
- _prepared_get_<object>: CassPreparedPtr
- _prepared_update_<object>: CassPreparedPtr
- _prepared_delete_<object>: CassPreparedPtr
- CREATE_<X>_KEYSPACE: const char*
- CREATE_<Y>_TABLE: const char*
- INSERT_<Y>: const char*
- SELECT_<Y>: const char*
- UPDATE_<Y>: cosnt char*
- DELETE_ <Y>: const char*

+ CqlManager(CqlClient)
+ configure (book): CqlResult
+ create_<y>(const <class name>&): CqlResult
+ get_<y> (const int&, const CassUuid&): pair<
      CqlResult, <object name>>
+ update_<y> (const int&, const CassUuid&, ...): CqlResult
+ delete<y> (const int&, const CassUuid&): CqlResult

- init_repare_statements(): CqlResult
- init_schema(): CqlResult
```

Figure C.2: Cql Manager UML diagram of the class template

Naming conventions

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 before the creation of the class.
Each class will contain multiple CassPreparedPtr objects that we will use when creating and 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 to manage more commands, for example getting a table's data based on a field that is not a primary key or deleting 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 a 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 initialize the prepared statement using the `init_prepared_statment` function described below.

init_prepare_statements ()

This function will initialize the prepared statements with the char pointers that we have mentioned previously. As we can see, each CassPreparedPtr will have a string as its 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 a 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 whether 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 as a parameter all of the object's fields and try to execute a statement with these 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 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 (for example: when we delete a field and do 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 object which values we want to modify. To map the Cassandra columns of the row to classic variables we will use the value-getter functions explained previously.
The function will return a `ResultCode::OK` if the operations went as planned, or the related ResultCode of the error if we get an error.

C.0.7 Gluing up the program

To better explain how the above code will function, three diagrams have been made to display the process of getting data from the DB, updating/deleting data from the DB, and also the process of initializing one of the Cql Manager's statements.

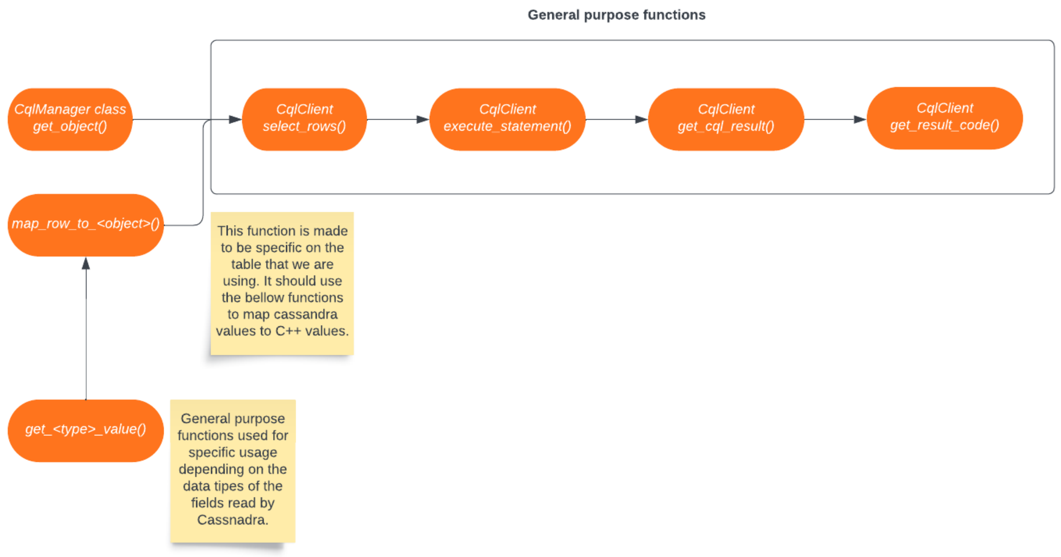

General purpose functions
CqlManager class get_object()
CqlClient select_rows()
CqlClient execute_statement()
CqlClient get_cql_result()
CqlClient get_result_code()
map_row_to_<object>()
This function is made to be specific on the table that we are using. It should use the bellow functions to map cassandra values to C++ values.
get_<type>_value()
General purpose functions used for specific usage depending on the data tipes of the fields read by Cassnadra.

Figure C.3: Flow diagram for reading data

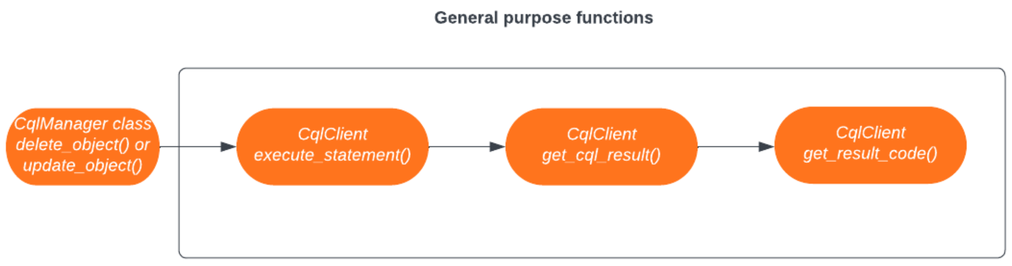

General purpose functions
CqlManager class delete_object() or update_object()
CqlClient execute_statement()
CqlClient get_cql_result()
CqlClient get_result_code()

Figure C.4: Flow diagram for deleting/updating data

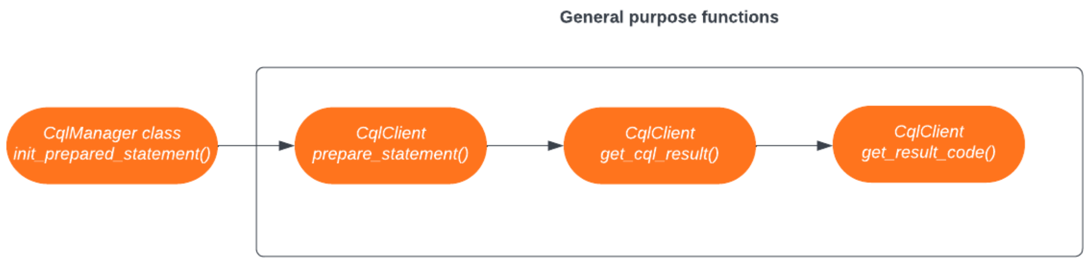

General purpose functions
CqlManager class init_prepared_statement()
CqlClient prepare_statement()
CqlClient get_cql_result()
CqlClient get_result_code()

Figure C.5: Flow diagram for initializing statements

Appendix D

API Endpoints

Method	Endpoint	Description
POST	/api/announcements	Creates a new announcement
GET	/api/announcements/announcement-id	Gets the announcement with the given id
DELETE	/api/announcements/announcement-id	Deletes the announcement with the given id
POST	/api/announcements/announcement-id/files	Creates a new file for the announcement
GET	/api/announcements/announcement-id/files?file_id=file-id	Gets the file with the given id assigned to the announcement
DELETE	/api/announcements/announcement-id/files?file_id=file-id	Deletes the file with the given id assigned to the announcement
POST	/api/announcements/announcement-id/tags	Adds the given tags to the announcement
GET	/api/announcements/announcement-id/tags	Gets the tags assigned to the announcement
DELETE	/api/announcements/announcement-id/tags	Removes the given tags from the announcement
POST	/api/announcements/announcement-id/answers	Creates a new answer for the announcement
DELETE	/api/announcements/announcement-id/answers?answer_id=answer-id	Deletes the answer with the given id assigned to the announcement
POST	/api/grades	Creates a new grade for a specific user
GET	/api/grades	Will return a list of grades that are assigned TO the user that sent the request
GET	/api/user/user_id/grades	Will return the grades assigned TO the user with the specified user id
GET	/api/course/course_id/grades	Will return a list of grades that are assigned to this course
PUT	/api/grades/grade_id	Will change a grades data
DELETE	/api/grades/grade_id	Will delete a grade

Table D.1: API endpoints part 1

Method	Endpoint	Description
POST	/api/courses	Creates a new course
GET	/api/course/course-id	Gets the course with the given id
GET	/api/course/course-id/users	Gets the users enrolled in the course with the given id
GET	/api/user_courses	Gets the courses enrolled by the current user
PUT	/api/course/course-id	Updates the course with the given id
DELETE	/api/course/course-id	Deletes the course with the given id
POST	/api/course/course-id/thumbnail	Creates a new thumbnail for the course with the given id
GET	/api/course/course-id/thumbnail?user_token=user-token	Gets the thumbnail for the course with the given id and user token
DELETE	/api/course/course-id/thumbnail	Deletes the thumbnail for the course with the given id
POST	/api/course/course-id/files	Creates a new file for the course with the given id
GET	/api/course/course-id/files	Gets all the files for the course with the given id
GET	/api/course/course-id/files?file_id=file-id	Gets the file with the given id assigned to the course with the given id
PUT	/api/course/course-id/files?file_id=file-id	Updates the file with the given id assigned to the course with the given id
DELETE	/api/course/course-id/files?file_id=file-id	Deletes the file with the given id assigned to the course with the given id
POST	/api/course/course-id/users	Adds users to the course with the given id
DELETE	/api/course/course-id/users	Removes users from the course with the given id
POST	/api/course/course-id/questions	Creates a new question for the course with the given id
GET	/api/course/course-id/questions	Gets all the questions for the course with the given id
DELETE	/api/course/course-id/questions	Deletes all the questions for the course with the given id
POST	/api/course/course-id/questions/question-id/answers	Creates a new answer for the question with the given id assigned to the course with the given id
DELETE	/api/course/course-id/questions/question-id/answers	Deletes the answer with the given id assigned to the question with the given id assigned to the course with the given id

Table D.2: API endpoints part 2

Method	Endpoint	Description
POST	/api/environment/school	Create a new school
GET	/api/environment/school?id=school-id	Get a school
PUT	/api/environment/school?id=school-id	Update a school
DELETE	/api/environment/school?id=school-id	Delete a school
POST	/api/environment/country	Create a new country
GET	/api/environment/country ?id=country-id	Get a country
PUT	/api/environment/country ?id=country-id	Update a country
DELETE	/api/environment/country ?id=country-id	Delete a country
POST	/api/environment/holidays ?school_id=school-id	Create a new holiday
GET	/api/environment/holidays ?school_id=school-id	Get all holidays
DELETE	/environment/holidays ?school_id=school-id&date=date	Delete a holiday
POST	/api/tags	Create a new tag
GET	/api/tags	Get all tags
GET	/api/tags/tag-id	Get a tag by id
PUT	/api/tags/tag-id	Update a tag by id
DELETE	/api/tags/tag-id	Delete a tag by id
POST	/api/tags/tag-id/add_user?user_id=user-id	Add a user to a tag
GET	/api/tags/tag-id/users	Get all users by tag
GET	/api/tags/personal_tags	Get all tags by user
DELETE	/api/tags/tag-id/remove_user?user_id=user-id	Remove a user from a tag
POST	/api/todos	Create a new todo
GET	/api/todos	Get all todos
GET	/api/todos/todo-id	Get a todo
PUT	/api/todos/todo-id	Update a todo
DELETE	/api/todos/todo-id	Delete a todo
POST	/api/users	Create a new user
GET	/api/users	Get all users
GET	/api/users/user-id	Get a user
PUT	/api/users/user-id	Update a user
DELETE	/api/users/user-id	Delete a user
POST	/api/login	Log in
POST	/api/logout	Log out

Table D.3: API endpoints part 3