VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Phan Tri Hoang Duong

BASIC ONLINE LEARNING PLATFORM

School of Technology
2023

ACHKNOWLEDGEMENTS

To complete this thesis project, I would like to express my warmest appreciation to all VAMK teachers, especially to two teachers. The first one is my thesis supervisor, Dr. Ghodrat Moghadampour, for his improvement ideas and valuable advice for my project work. The second one is Mr. Timo Kankaanpää, who provided me an enormous opportunity to collaborate with Wärtsilä, a big cooperation in Finland.

Moreover, I want to thank Markku T. Mäenpää and Ilkka Ristiluoma from Wärtsilä and Benjamin Drury, my internship's mentor, for allowing me to handle two biggest challenges that I have ever faced in my life.

Moreover, I want to send my endless gratefulness to my family, my biggest and most enthusiastic mental encouragement.

Helsinki, April 27, 2023

Hoang Duong

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Phan Tri Hoang Duong
Title	Basic Online Learning Platform
Year	2023
Language	English
Pages	81 + 2 appendices
Name of Supervisor	Ghodrat Moghadampour

Nowadays, online classes have a radical development, due to their flexibility, cost-effectiveness, and accessibility. With an online platform, learners can easily access a variety of online courses in many areas of expertise and teachers have a place for expressing knowledge in a flexible, time-effective, and economical way.

With that motivation and inspiration, this thesis project aimed to develop a basic online courses platform for teachers and students. Learners can enroll easily and access the courses' documents and materials, including the lectures and quizzes, while teachers can create and modify their course contents and quizzes data.

This platform used TypeScript as the programming language. Some modern technologies were implemented to develop a server-side application, including Next.JS, a modern front-end framework, and GraphQL was implemented to communicate with MongoDB database.

The final product is an online learning platform with some of the latest technologies in full-stack development. This platform works for both teachers and students, with different interface designs for different user types. Students can search for the course, enroll in any courses based on their needs, and experience with the course lectures and quizzes. On the other hand, teachers can create a course and they have rights to maintain and update the course lectures and quizzes.

Keywords TypeScript, Next.JS, Node.JS, MongoDB, GraphQL

CONTENTS

ACHKNOWLEDGEMENTS

ABSTRACT

1 INTRODUCTION 12

2 RELEVANT TECHNOLOGIES 14

2.1 React.JS 14

2.2 Next.JS 14

2.3 TypeScript 15

2.4 CSS and SCSS 16

2.5 Node.JS 16

2.5.1 Node Package Manager (npm) and Yarn 17

2.5.2 Middleware 17

2.5.3 Express.JS 18

2.6 MongoDB 18

2.6.1 NoSQL 18

2.6.2 MongoDB – a NoSQL Database 19

2.7 GraphQL 19

2.7.1 Apollo Client 20

2.7.2 Apollo Server 20

2.8 JSON Web Token (JWT) 21

3 APPLICATION DESCRIPTION 22

3.1 List of requirements 22

3.2 Use-case Diagrams 23

3.2.1 Student Use-Case Diagram 24

3.2.2 Teacher Use-Case Diagram 25

3.3 Class Diagram 26

3.3.1 Student and Teacher 26

3.3.2 Course 27

3.3.3 LessonCompleted Data Type 28

3.3.4 StudyProgress Data Type 28

3.4 Sequence Diagram 29
3.4.1 User Authentication Diagram 29
3.4.2 User Register Diagram 29
3.4.3 Answer Quiz Diagram 30
3.4.4 Course Creation Diagram 31
3.4.5 Quiz Creation Diagram 32
3.5 Architectural Diagram 33
4 DATABASE AND GUI DESIGN 34
4.1 Design of the Database 34
4.1.1 Course Schema 34
4.1.2 Student Schema 37
4.1.3 Teacher Schema 41
4.2 GUI Design 42
5 IMPLEMENTATION 54
5.1 General Description 54
5.1.1 Front-end Client 55
5.1.2 Setting up the Back End. 58
5.1.3 Setting up Apollo Server 60
5.1.4 Setting up Apollo Client 62
5.2 Implementation of Various Parts 63
5.2.1 User Authentication 63
5.2.2 Sign up process 67
5.2.3 Processing Student's Answers for the Quiz 71
6 TESTING 76
7 CONCLUSIONS 78
7.1 Achievements 78
7.2 Challenges 78
7.3 Future Development 78
REFERENCES 80
APPENDICES

LIST OF FIGURES, CODE SNIPPETS, AND TABLES

Figure 1. Node.JS system architecture /11/ 17
Figure 2. Student use-case diagram..... 24
Figure 3. Teacher use-case diagram 25
Figure 4. Class diagram 26
Figure 5. User authentication diagram 29
Figure 6. User register diagram 30
Figure 7. Answer quiz diagram 31
Figure 8. Course creation diagram..... 32
Figure 9. Quiz creation diagram..... 32
Figure 10. Architectural diagram 33
Figure 11. MongoDB database collections 42
Figure 12. Sign-in form..... 43
Figure 13. Sign-up form 43
Figure 14. Teacher's home page 44
Figure 15. Student's home page 44
Figure 16. Teacher's profile page..... 45
Figure 17. Student's profile page 45
Figure 18. Course creation page 46
Figure 19. Category page 46
Figure 20. Result after searching successfully. 47
Figure 21. Successfully enrolled message..... 47
Figure 22. A course's main page 48
Figure 23. Course lesson's card..... 48
Figure 24. Course owner's lesson content..... 49
Figure 25. Content creation form 49
Figure 26. Create content success, which returns a message. 49
Figure 27. Quiz creation form 50
Figure 28. Successful announcement after successfully creating a quiz. 50
Figure 29. Course's enrolled student list 51
Figure 30. Course content page 51

Figure 31. Quiz page 52
Figure 32. An announcement of finishing the quiz 52
Figure 33. Quiz result page 52
Figure 34. Course completion confirmation box 53
Figure 35. Main structure of Next.JS front-end 55
Figure 36. Next.JS /src folder structure 56
Figure 37. Styling folder 57
Figure 38. File structure of the server 58
Figure 39. Folder of MongoDB models 59
Figure 40. Types folder 59
Figure 41. The utils folder 59
Figure 42. Testing the back-end folder 76

Code Snippet 1. Course type definition in GraphQL 34
Code Snippet 2. Course schema in MongoDB 35
Code Snippet 3. Lesson type in GraphQL 36
Code Snippet 4. Lesson Schema in MongoDB 36
Code Snippet 5. Quiz type in GraphQL 37
Code Snippet 6. Quiz schema in MongoDB 37
Code Snippet 7. Student type definition in GraphQL 38
Code Snippet 8. Student schema in MongoDB 38
Code Snippet 9. Plugin the schema with "mongoose-unique-validator" library . 39
Code Snippet 10. StudyProgress type definition in GraphQL 39
Code Snippet 11. StudentProgress schema in MongoDB 40
Code Snippet 12. Status - an enum type definition in GraphQL 41
Code Snippet 13. Teacher type definition in GraphQL 41
Code Snippet 14. Teacher schema in MongoDB 42
Code Snippet 15. Run tsc (TypeScript compiler) 54
Code Snippet 16. The Layout component 57
Code Snippet 17. Wrapping Layout around the Component object 57
Code Snippet 18. Head component in the root index.tsx file 58

Code Snippet 19. Install Apollo Server command. 60
Code Snippet 20. Setting up Apollo Server..... 60
Code Snippet 21. Set up the Express middleware..... 61
Code Snippet 22. Install Apollo Client 62
Code Snippet 23. Create new Apollo Client with the server URL..... 62
Code Snippet 24. Wrap the ApolloProvider with the defined client into Layout component. 63
Code Snippet 25. Login useMutation hook and the submitting action. 63
Code Snippet 26. LOG_IN variable 64
Code Snippet 27. Log-in mutation function in the back-end 65
Code Snippet 28. Function authLink in the front-end 65
Code Snippet 29. Getting user data from the homepage 65
Code Snippet 30. The GET_USER variable. 66
Code Snippet 31. Get user data in back-end server. 66
Code Snippet 32. User, a Union type definition in GraphQL..... 67
Code Snippet 33. User __resolveType() function 67
Code Snippet 34. The sign-up function inside the useMutation hook..... 68
Code Snippet 35. SIGN_UP mutation variable 68
Code Snippet 36. Submit sign-up mutation function. 70
Code Snippet 37. Password creation verification..... 70
Code Snippet 38. Save the new user into MongoDB database. 71
Code Snippet 39. Get the quiz result query. 71
Code Snippet 40. GET_QUIZ_RESULT query 72
Code Snippet 41. Check if user has completed the quiz. 72
Code Snippet 42. Quiz result processing function..... 74
Code Snippet 43. Study progress percentage calculation 74
Code Snippet 44. Quiz overall result calculation 75
Code Snippet 45. Server's tsconfig.json file 82
Code Snippet 46. Mutation processes the student's answer for the quiz. 84

Table 1. List of functional requirements and its priority 23

Table 2. Testing back-end report 76
Table 3. Testing front-end report 77

LIST OF APPENDICES

APPENDIX 1. Server's tsconfig.json configuration file

APPENDIX 2. The answerQuiz() mutation in the server.

LIST OF ABBREVIATIONS

API	Application Programming Interface
HTML	Hyper Text Markup Language
JSX	JavaScript Syntax Extension
DOM	Document Object Model
I/O	Input/Output
HTTP	Hypertext Transfer Protocol
CRUD	Create, Read, Update, and Delete
JSON	JavaScript Object Notation
CORS	Cross-origin resource sharing
UI	User interface
HMAC	Hashed/Hash-based message authentication code.
RSA	Rivest–Shamir–Adleman
ECDSA	Elliptic Curve Digital Signature Algorithm

1 INTRODUCTION

Full-stack applications have existed more commonly in our life, with different UI and different purposes. There are a lot of technologies that are suitable for developing a full-stack application, such as React and Angular for the front-end; for example, Node.js, Java, and Python for the backend; and MySQL and MongoDB for the database. According to some surveys conducted by State of JavaScript (2022, online), in the year of 2022:

- 81.8% of survey respondents use React.JS for the front-end framework. Nearly half of the number of respondents implement Next.JS for their rendering framework. /1, 2/
- In backend development, 50.7% of survey respondents use Express.JS for the backend. Furthermore, Node.JS is the most popular execution environment for developers with 70.9% of the surveyed developers. /3/

This thesis project will introduce a TypeScript full-stack application, which will implement the most popular technologies mentioned above: React.JS and Next.JS for the front-end, Node.JS and Express.JS for the back-end, and MongoDB for the database. The main purpose is to develop an online application platform, which is becoming more popular with people from different backgrounds.

Nowadays, thanks to the outstanding development of technology, online learning has made a powerful impact on education around the world. There are many well-known online learning platforms, such as Udemy, Coursera, and edX. According to a report from the National Center for Education Statistics (2022, online), there are about 11.8 million undergraduate students enrolled in at least one online course in fall 2020. This number was approximately two times higher than in the same period of 2019 (6.0 million). /4/

There are two main advantages of online teaching:

- First, online learning is more time-effective and convenient than traditional teaching, as learners positively arrange their studying timetable. Moreover, everything people need to establish and/or study an online course is an internet connection and a portable device, such as a laptop, or smartphone.
- Second, learners can easily access a variety of topics and courses from many famous institutions around the world among an online learning platform. These courses contain detailed instructions, exercises, projects, and exams align with criteria and curriculum provided by those institutions, which is not different from traditional teaching methods. After completing courses on those platforms, users will receive some certifications issued by the institution, which supports their purposes in their studies and/or their working careers.

The main objective of this thesis is to develop a basic online learning platform for students and teachers. After successfully enrolling in a course, learners can easily access the course materials and participate in every activity inside it, including reading lectures notes, doing quizzes and assignments. After finishing a course, students receive a certification for their completion of the course. On the other hand, teachers can create new courses and update their contents and quizzes details.

2 RELEVANT TECHNOLOGIES

2.1 React.JS

React.JS (known as React) is an open-source JavaScript library for front-end development, originated by Meta (formerly Facebook) engineers. Since it was officially open sourced in 2013, React has become an effective selection for many companies in their production environments. /5/

The greatest advantage of React is its flexibility and user experience. Instead of developing an HTML single page, React allows users to develop a multiple-pages application by developing React components. Those components will be returned as a JSX element then DOM will render them in the root element in the HTML file. For that reason, React allows developers to create advanced web and mobile applications. /5, 6/

2.2 Next.JS

Next.JS is an open-source JavaScript framework which supports React in building advanced web applications. Today, it is widely used by top-level companies such as Netflix, TikTok, Nike, and Uber. This framework originated to solve the problem of time optimization from React, since React runs on the web browser, thus, the browser must download the entire application bundle, execute all React components and render the result to the browser via DOM. However, Next.JS is a server-side rendering framework, which will render the HTML on the server before sending it to the browser. /7/

Moreover, Next.JS provides advanced built-in components and functions for file-based routing systems and route pre-fetching. Here are some examples:

- Every Next.JS project contains a **/pages** directory, where every file inside this directory is equivalent to a new route for the application.
- To create a dynamic route for the applications, instead of creating separate page every time or functions which saves the route as a variable, we can

just create a new folder, where its name is a route variable wrapped inside square brackets. That variable will contain the exact value received from the browser's address bar.

These features will be more time-efficient for developers, which allows both client and server sides to share the same code. /7/

2.3 TypeScript

JavaScript, a scripting language whose age is 28 in 2023, is familiar to web developers. JavaScript is flexible, dynamic, and cross-platform, i.e., it can work well on different platforms. /8/

Despite those advantages, JavaScript contains two main disadvantages. First, it is weakly typed. The freedom of dynamic typing leads to some errors in displaying data to the browser, which increases the work for developers and the application processing time by adding new lines of code. Second, JavaScript is quite complicated for developers who want to build object-oriented architecture since it is a scripting language. Hence, TypeScript was originated to solve this problem. /8/

TypeScript is an object-oriented and open-source programming language developed by Microsoft developers in 2012. TypeScript inherits all functionalities from JavaScript, together with the characteristics of an object-oriented programming language, such as classes, visibility scopes, namespaces, inheritance, unions, and interfaces. Also, it offers comments, variables, statements, expressions, modules, and functions. For that reason, TypeScript becomes more popular for developers to build large-scale web applications. /8/

Due to inheriting JavaScript, TypeScript contains most of JavaScript basic data type. However, there are some different types:

- **Any:** Type "any" which can be known as "anything that you wish". This type is served for any object whose incoming types are invalidated, unknown, or too complicated to define briefly. In many cases, especially with the

types from a third-party (e.g., MongoDB, GraphQL APIs), type “any” economizes the working time for developers. /8/

- **Unknown:** Similar to “any”; however, “unknown” requires some explicitly type-checked before handling the object containing that type. /8/
- **Never:** Never returns type that should never be happened.
- **Void:** Void is mainly used for function that returns no object.
- **Intersection Types**: Intersection can support developer to select the property from two or more different types and create a new type. /8/
- **Union Types**: Union Types allows an object to take one among the various data type. /8/

2.4 CSS and SCSS

CSS (Cascading Style Sheet) is a scripting language whose main purposes are styling and developing web pages. It was created to make a clearer presentation for the content of a website by adding color, layout, and fonts. /9/

SCSS, which is also referred to as Sass, Sassy CSS, stands for Sassy Cascading Style Sheet. This is a pre-processor language that is compiled or interrupted into the CSS. SCSS adheres to the concepts from CSS, however, it contains many advanced features, especially variables and nesting. This allows developers to build a more flexible, cleaner, and quicker CSS file in the whole program structure. SCSS also supports nested syntax and math functions to create a livelier and more colorful web application. Though SCSS is more commonly used in Ruby, it works normally with JavaScript and TypeScript applications. /9/

2.5 Node.JS

Node.JS (also known as Node) is a JavaScript runtime environment built on Chrome's V8 JavaScript engine. Figure 1 is the diagram of Node.JS system architecture. /10/

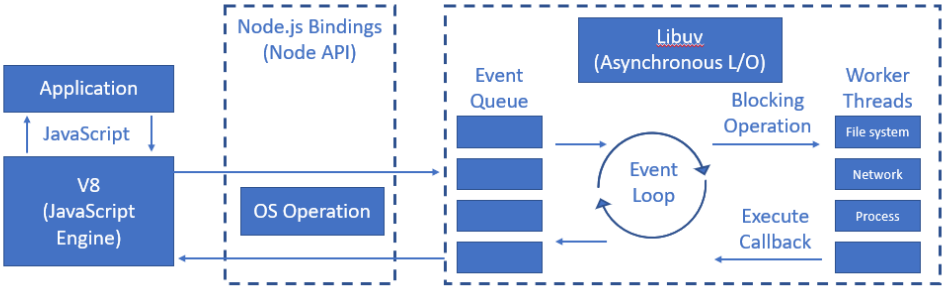

Figure 1. Node.JS system architecture /11/

Node's system contains an event-driven architecture which supports asynchronous and non-blocking I/O model. This allows JavaScript to be used on the server side, so many advanced use cases will be done within a browser. Its system decreases the waiting approach to serving requests, as a result, the server efficiently handles multiple requests. /10/

2.5.1 Node Package Manager (npm) and Yarn

Building a Node.js backend requires one most essential element: the Node Package Manager (npm). npm is a key for developers to access an ecosystem of supported Node packages and open-source libraries built by developers around the world. /10/

Another option is using yarn, a newer package manager originated by Meta. Yarn connects the developer with some packages which are not yet supported via npm. /10/

2.5.2 Middleware

In software engineering, middleware is a functionality which connects different old and new components in an application and makes them work properly. /10/

In server-side web application frameworks, middleware is a function which has a right to access the HTTP request-response pipeline. This also means middleware can access HTTP request-response objects and the further middleware functions

in the web application's request-response cycle. Creating middleware in a back-end application also makes the codes clearer and maintainable. /10/

2.5.3 Express.JS

Developing a complete backend server with only Node.JS is not enough, since Node cannot support some features, especially API routing. Hence, Express.JS (known as Express) has been developed to create an extensible server-side application. This framework is complementary to Node in providing HTTP utility methods and middleware functionality. /10/

Express is an essential part to develop a complete API routings and middleware framework, which is capable of handling almost any compatible middleware into the request handling chain. Moreover, Express can serve static files to the client, manage the access to a CRUD operation with authentication integration, error handling, and contains any middleware package that will extend the application. /10/

2.6 MongoDB

2.6.1 NoSQL

NoSQL officially stands for “non-SQL” or “not-only SQL”, which is a non-tabular database, and its database structure is opposite from MySQL, a relational-tables database. It provides flexible schemas with many different variable types for the value and handles smoothly with a big amount of data and user loads. NoSQL has four major types of databases: /12/

- Document databases: Storing data method is like JSON, which contains pairs of fields and keys.
- Key-value databases: Each item contains keys and values.
- Wide-column stores: Store data in dynamic tables, rows, and columns.
- Graph databases: Store the value of items as nodes and the relationships between the nodes by the edges. /12/

2.6.2 MongoDB – a NoSQL Database

MongoDB is a NoSQL database whose architecture is a document database. It stores data in flexible and JSON-like documents. MongoDB is also flexible, contains great indexing which allows fast lookups, and enables ad hoc queries, i.e., queries with loosely typed and relies upon some variables. /10, 13/

Applications use MongoDB as the database contains some benefits, including the built-in support for high availability, multi-data center scalability across different geographic distributions, and a real-time aggregation that allows servers to analyze data and more quickly and efficient even for an enormous volume of data. /10/

MongoDB is suitable for many programming languages and frameworks, including Node and Express. These three technologies together build a strong and fully JavaScript-based and standalone server-side application, i.e., an application that operates independently of any software on the server. Together with React or Angular for the front-end, Node, Express, and MongoDB creates a MERN or MEAN stack – a powerful full-stack technology which is the backbone of many JavaScript applications. /10/

2.7 GraphQL

GraphQL is a query language for APIs and a runtime for handling APIs communications among the client, the server, and the databases. GraphQL requires developers to provide a detailed and clear data of APIs, so it becomes easier for the server to evolve APIs and fetch the data from database /14/

One of the most advantages of GraphQL is its principle of working. It will ask the user to send the query by declaring the type of system to the API and return data that user needs without redundancies. For that reason, one endpoint can return different data structures depending on what user declares in the type of system declarations. Moreover, GraphQL APIs get all data that an application needs smoothly in one single request, saving processing time for the application. /14/

Every GraphQL project must contain a **schema** and a **resolver**.

- A **schema** defines the data structure and the resolver functions expected data types. The required type will include an exclamation mark.
- A **resolver** is a set of functions that control the final data to align the defined schema saved in the **typeDef** property. Those data can be the response of one or some CRUD operations from the back-end or a third-party API.

2.7.1 Apollo Client

Apollo Client is a JavaScript state-management library for handling GraphQL resolver. Its main function is to allow developers to manage the local and remote data. All data, which can be understood as a state, will be handled by GraphQL's query and mutations, which are the read and write operations. The data can be fetched, cached, or modified easily when there is any change in the UI. /15/

Defining middleware before handling the first request and caching the response data are unnecessary and all what a developer needs to define is the data a component needs. The **useQuery** and **useMutation** hooks will take the user's query or mutation schema, as long as the data is the variables. The server will then respond with enough information that the user needs. /16/

Moreover, when using Apollo Server for mutating data or handling query request data, the result will be updated everywhere in the application. Instead of defining a new local store for storing state, Apollo Client library provides the **InMemoryCache** class which developer must define while creating a server. The server then updates the cache memory by saving the object with the same identity property from a query's data into separate entries inside the cache. /16/

2.7.2 Apollo Server

Apollo Server is an open-source and spec-compliant server for handling GraphQL APIs request from any GraphQL client, for example, Apollo Client. Inside this

server, developers can build a production-ready and defined GraphQL APIs document that can be used in any source. /17/

Apollo Server can be a standalone back-end server by defining the s**tart-StandaloneServer** function. However, in some complex Express projects, especially any project required a customized CORS behavior, or a back-end server contains some advanced middleware before processing the GraphQL request, using **expressMiddleware** is a better choice than **startStandaloneServer**. /17, 18/

2.8 JSON Web Token (JWT)

JSON Web Token (JWT) is an open standard that allows the information inside a JSON object to be securely transmitted between different parties for some specific purposes. JWT uses the HMAC algorithm (contains hashing string functions and a secret key), an RSA public key, or a public/private key pair using or ECDSA. JWT produces a signed token. This token will prevent other parties from claiming it easily. This token is a private key and only the party holding the key can use it. /19/

A JWT key comprises three parts: the header (including the signing technologies/algorithms being used), the payload contains the claims (i.e., the statement about the user and some further data), and a signature (encoded data from the header and payload). /19/

3 APPLICATION DESCRIPTION

The main objective of this online learning platform is to provide a studying environment for learners with the most basic features that any online education platform should have. The project strictly follows the architecture of any full-stack application, with TypeScript as the main programming language. This platform consists of three parts:

- The front-end application displays the list of courses, list of different expertise, user's profile page, course page and course's contents. Students can see their studying progress on their profile page, while teachers can see their courses' students and they have a new course and contents creation form.
- The back-end server handles requests from the front-end server and returns the data by GraphQL APIs, such as authentication, get courses, course enrollments, update studying progress, add course's content and quizzes.
- The database contains course information and user data, i.e., students' and teachers' data.

3.1 List of requirements

Three are three main requirements type:

- Must-have: The most important and essential functionalities that the application must successfully handle in the program.
- Should-have: The necessary functionalities that an application should successfully handle, but it is not required.
- Nice-to-have: The unnecessary functionalities at this moment when the project is being conducted but can be considered to develop in the future.

In this basic platform, some advanced functions on current online learning platform for students will be listed as should-have and nice-to-have requirements in this project. The table of priority of requirements in this project are presented in Table 1.

Table 1. List of functional requirements and its priority

(1: Must have, 2: Should have, 3: Nice to have)

ID	Description	Priority
F1	User sign in/sign up	1
F2	Send the enrollment requests	1
F3	View user's profile	1
F4	View the course's details and materials (i.e., lessons, quizzes)	1
F5	View the course's enrolled students for course's owner	1
F6	View courses list	1
F7	Search for the courses by category and by keywords	1
F8	Teacher creates new course	1
F9	Teacher creates the course question and quiz details	1
F10	Return the certificates/equivalent documents after student's finishing the courses	1
F11	Calculate and display user's quiz points, study progress, and the final grade.	1
F12	Update user's personal information	2
F13	Delete and update course contents	2
F14	Student and teacher comments and/or discuss about the course via the Comments part	3

3.2 Use-case Diagrams

Before experiencing the application, a user must sign in or sign up if he/she is a new user. During signing up, the user must decide their roles (student or teacher). After that, they can experience the services in the application based on their selection role: Teacher or Student. Any user can view their user profiles, and search for any courses by name and category. However, users have to sign in successfully before experiencing the application.

3.2.1 Student Use-Case Diagram

Besides those functions mentioned above, students have a base functionality, which is a course learner. This means students will be the ones to experience the course.

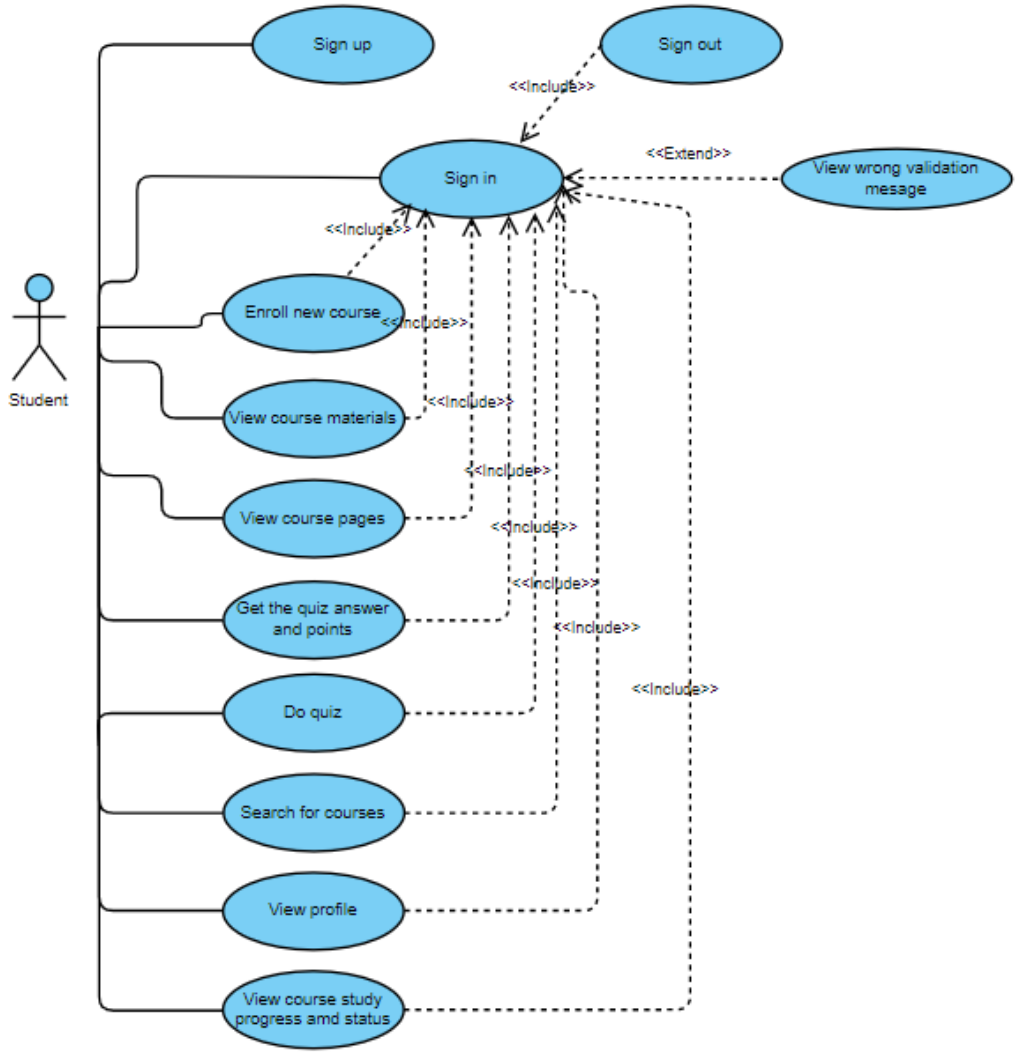

Figure 2. Student use-case diagram

A student can search for a new course, see the basic details of any course. To achieve the right to access the course materials and details, student must enroll for that course. After successfully enrolling, students can view the course materials and do the assignments and quizzes. Students can do the quiz and receive the points and comments for all the exercises that they have done after submitting

their answers. Students will receive a confirmation certificate after finishing the course.

3.2.2 Teacher Use-Case Diagram

The use-case diagram for teachers is shown in Figure 3. Currently in this project, the teacher's functionalities are quite limited. The most essential function that differentiate Teacher and Student is that the teacher can create and manage the course details.

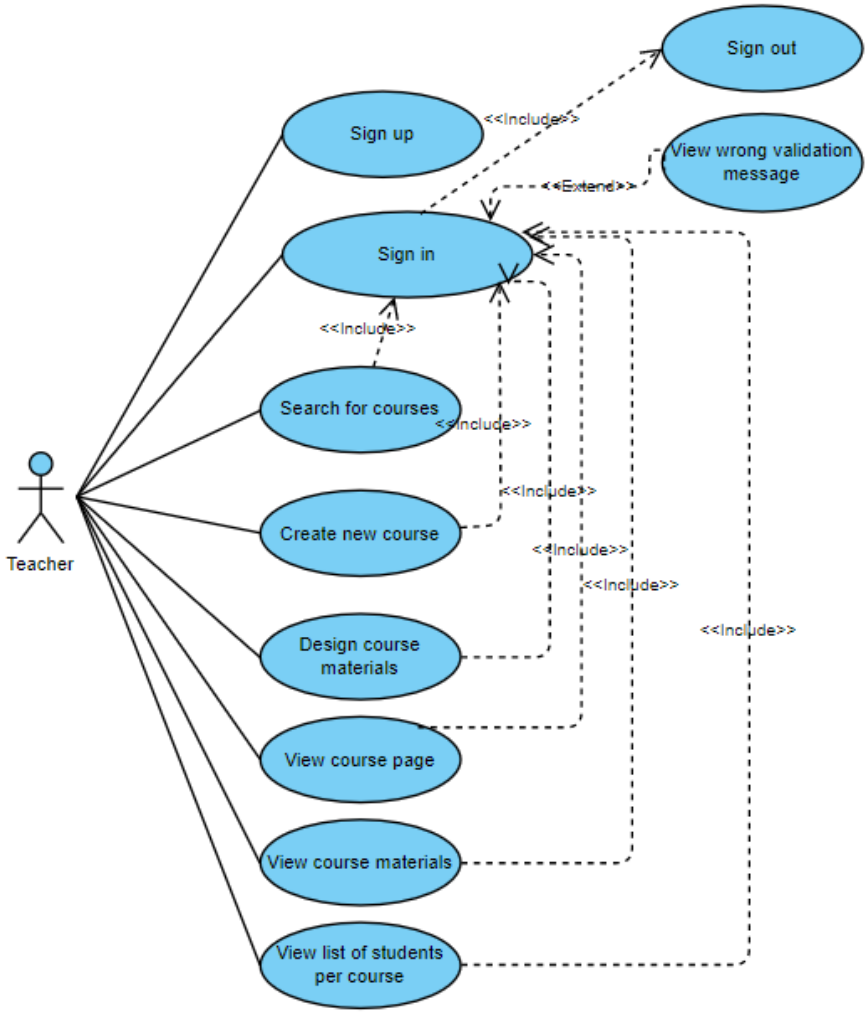

Figure 3. Teacher use-case diagram

For that reason, he/she is the owner and admin of their own courses, who gets a full right to view, create and update the materials (including quizzes and lessons'

contents) for the course. He/she can also search and see the basic details of the course.

3.3 Class Diagram

Figure 4 shows the class diagram of this application. As can be seen from this diagram, the two most basic data are User and Course. Each User contains basic details: name, email, and password. The other classes are the subclasses or helper classes for the User and Course class.

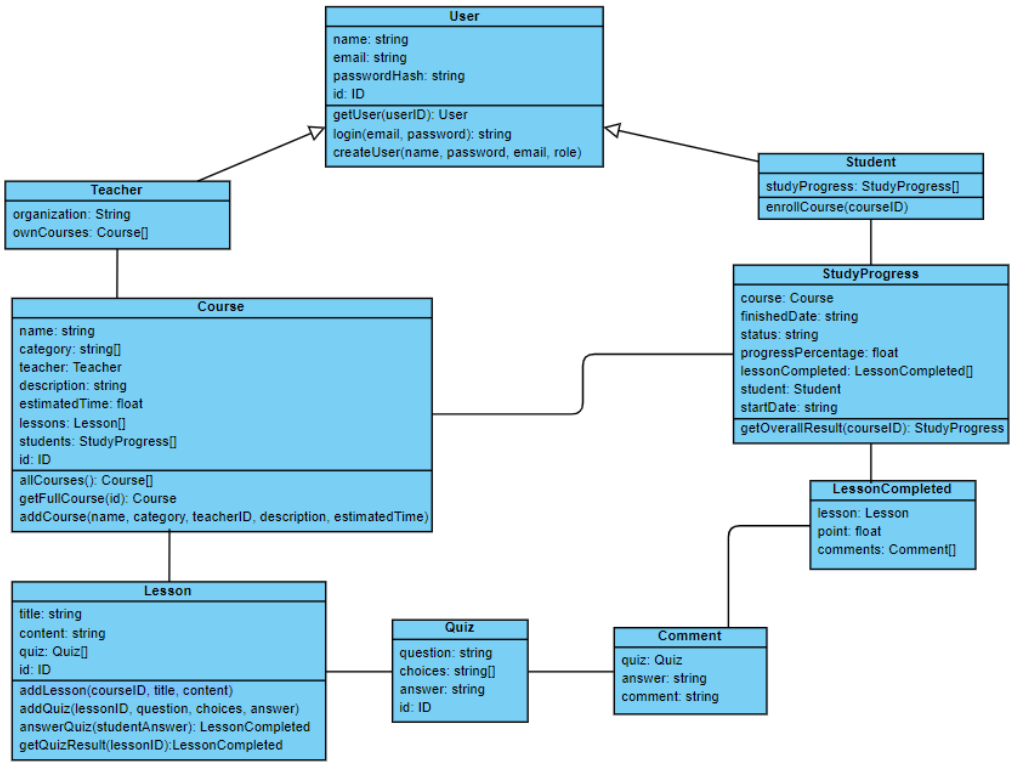

Figure 4. Class diagram

3.3.1 Student and Teacher

Teacher and Student are two sub-classes of the parent class User. The organization is required for all Teachers, while Students have a **studyProgress** array, consisting of their StudyProgress object, with the initial value being an empty array. Any User class contains three base operations:

- **getUser**: Views any user's data, including name and email. If the role is a teacher, the user can see his/her created course list. The data also includes the student's study progress list if the user's role is a student. The User ID is an optional parameter for this operation. This operation will return the current user's data if there is no user ID defined.
- **login(email, password)**: User authentication before using the application.
- **createUser**: Creates a new user for the application (i.e., sign-up operation). This operation requires the user's name, email address, password, and selected role. If the role is the teacher, the user must define his/her working organization.

The operation **enrollCourse** with course ID as a parameter is only for students to enroll in a new course.

3.3.2 Course

A teacher can be the owner of one or many courses, which is represented as a Course object.

The Course object contains name, categories list, teacher data, description, the estimatedTime (i.e., the estimated learning time) typed by the course creator, a list of lessons and a list of enrolled students. Course class functionalities include:

- **addCourse**: Creates the course with the defined name, categories list, teacher ID, description, and an estimated time.
- **getFullCourse**: Gets the completed information inside that course if the user is not an admin or not yet registered for the course, otherwise one can see just the basic information of the course (i.e., omit the lesson list).
- **allCourse**: Gets the course information by name or by category keyword included in the filter. The function can be called to view the teacher's course list. Otherwise, it will return all courses by default. Return courses contain only basic information.

A Lesson object must contain the title, content as the string. An array of Quiz is an optional property. The Quiz class contains a question, some multiple choices, and the final correct answer. Lesson class contains two attributes, which are **addQuiz** and **addContent**, which allows teachers to create and update the lessons' quizzes and contents.

3.3.3 LessonCompleted Data Type

The LessonCompleted class contains the student's completed lessons in one course. Its class has three elements: the Lesson data, overall point calculated by the server, and an array of comments. Operation **answerQuiz,** with a list of quiz IDs and their corresponding answer, is used for a student to answer the quiz, while **getQuizResult** returns the LessonCompleted object that contains the quizzes data.

Each enrolled student has one time only to do the quiz, and the server will return the Comment object after finishing the quiz (i.e., when all the quizzes have been answered).

The Comment object contains the quiz data, the answer (i.e., the student's answer for that quiz, which was received as a GraphQL Mutation argument).

3.3.4 StudyProgress Data Type

StudyProgress object records the student's study progress for a course. Any Study-Progress object must contain the student data, course data, status ("PASSED", "ONGOING", "FAILED"), progressPercentage (display the number of completed lessons), lessonCompleted list, startDate and finishedDate.

The status is "PASSED" if the overall point calculated after finishing the course is greater or equal to 5, "FAILED" if it less than 5, and "ONGOING" if the progress-Percentage is not 100. After finishing the course, this progress will be updated by adding the finished date into the data finishedDate.

The **lessonCompleted** property is an array of LessonCompleted objects, which will be added after the student finishes all quizzes contained inside that lesson.

3.4 Sequence Diagram

3.4.1 User Authentication Diagram

Figure 5 below is the user authentication diagram. Before authenticating a user, he/she must type both email and password. The result is the User information, which is saved into the cache memory of the app's Apollo Client.

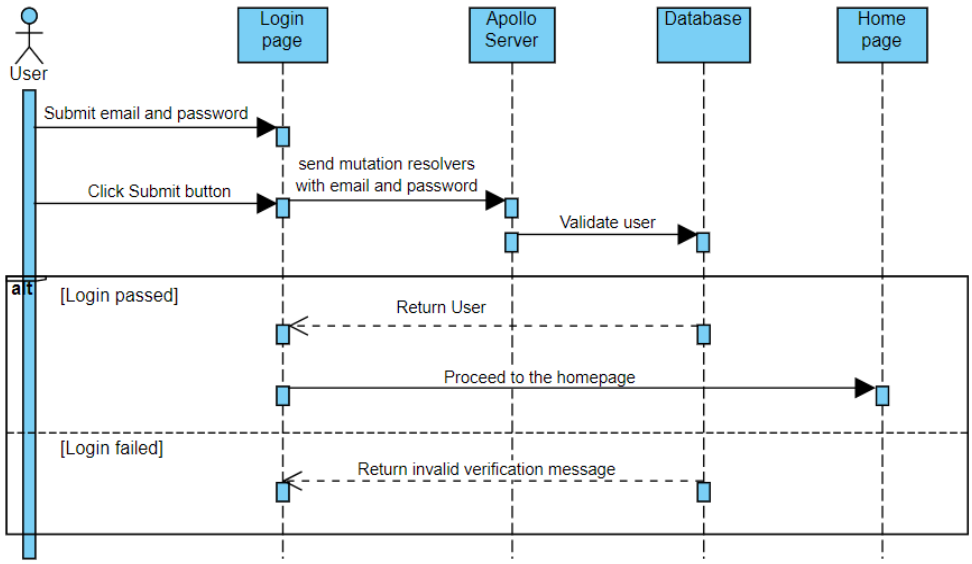

Figure 5. User authentication diagram

The authentication process starts by receiving the email and password from the client, then Apollo Client sends this request to the server as a mutation. If the request is successfully validated, the user data will be saved as a **contextValue** in both Apollo Server and Apollo Client, and the client redirects the user to the main page. If the login validation fails, the client shows an error message to the user.

3.4.2 User Register Diagram

Figure 6 shows the user register diagram. Users must log in after successfully signing up. The log-in process is not different from the one mentioned above.

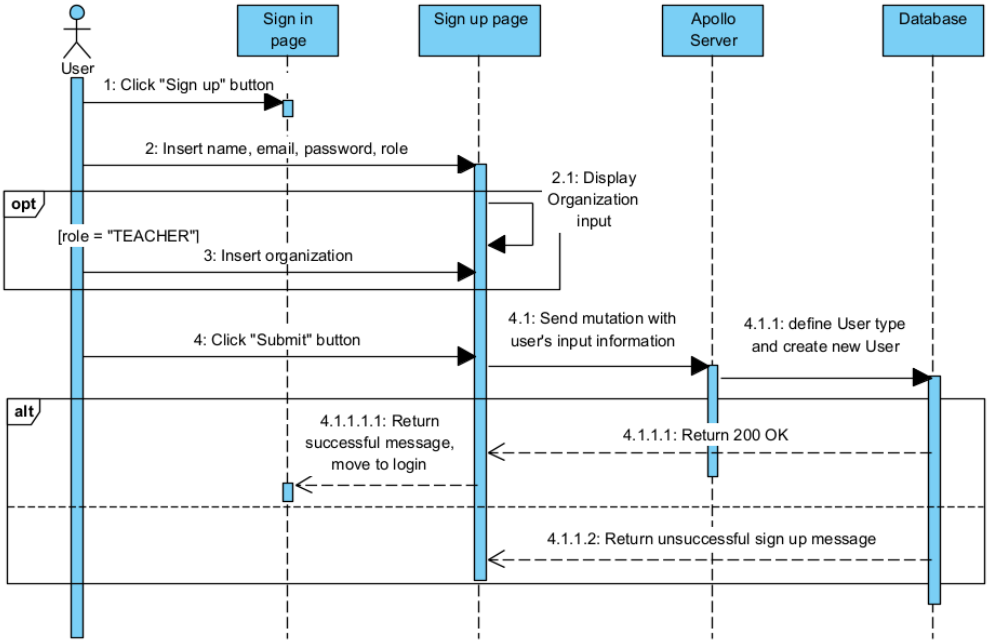

Figure 6. User register diagram

Users must click the “Sign up” button before this process starts. The client receives the basic information from the user and passes the information to the server by Apollo Client’s **useMutation** hook. The password must contain both word characters and digits. Users must also define their roles between teacher and student. If this criterion passes, the server will encrypt the user’s password and create a new user in the database.

Finally, users once again log in with the registered email and password. They can start using the platform after completing these processes successfully, otherwise an error message passes directly to the user.

3.4.3 Answer Quiz Diagram

Figure 7 shows the answer quiz diagram. This function works only with students.

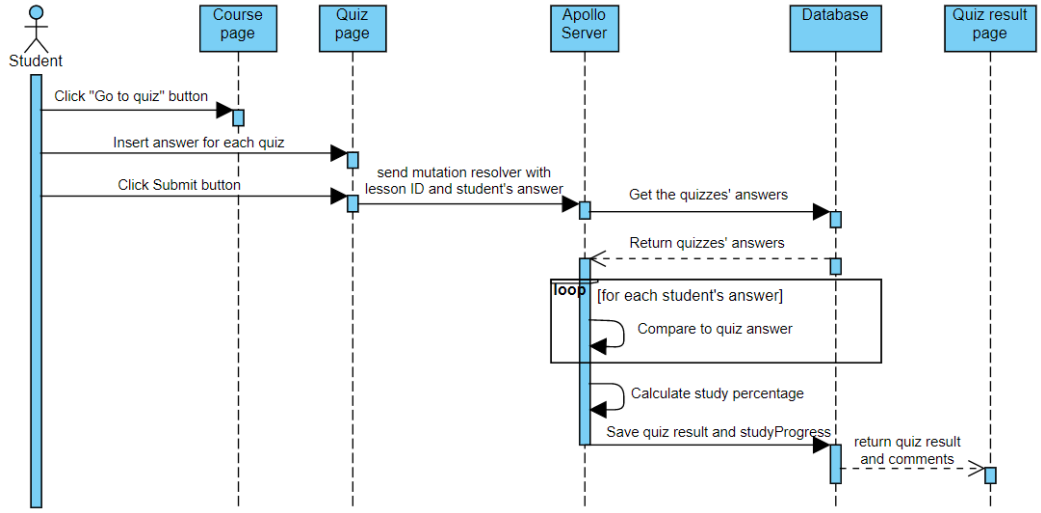

Figure 7. Answer quiz diagram

This process starts by clicking on the quiz link on the course page. The student will then answer the questions and the client sends the request (including the quiz ID and the user's answers) after the student presses the Submit button. The server then finds the answer list of the quiz from the database and operates a for-loop for comparing each of student's answers with the quiz answer. This result is saved to the database together with the student's study progress percentage that has been updated.

If the validation passes, the client disables the quiz to prevent the student from re-attempting it and displays the result. On the other hand, the student receives an error message if the validation fails.

3.4.4 Course Creation Diagram

Figure 8 shows the answer quiz diagram. This function works only with teachers.

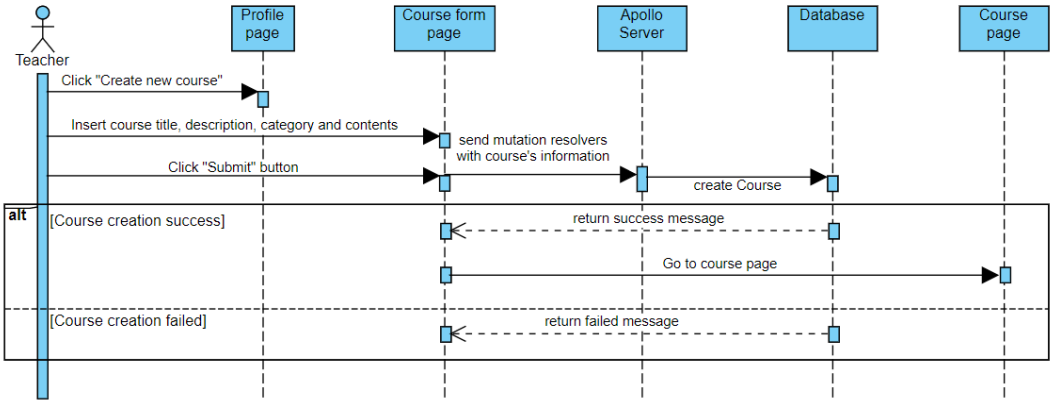

Figure 8. Course creation diagram

The server receives the course basic details from the user as a request. A new course has been created, and the response will be sent to the client. Apollo Client will then re-fetch the query which oversees displaying all courses and update the cache inside the defined client.

3.4.5 Quiz Creation Diagram

The quiz creation diagram is presented in Figure 9. This function works only with teachers.

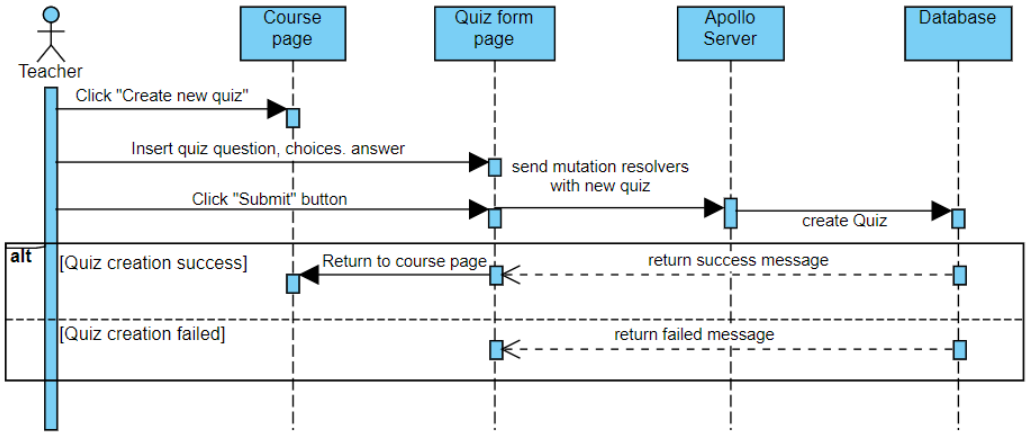

Figure 9. Quiz creation diagram

The server receives the quiz content data (question, correct answer, multiple choices) from the teacher as a request. A new quiz is created, and the response will be sent to the client. Apollo Client will then re-fetch the query which oversees displaying all courses and update the cache inside the defined client.

3.5 Architectural Diagram

The technical architecture is quite simple, with three main parts: the front-end, the back-end, and the cloud database. The front-end client and back-end server communicates by GraphQL APIs, whose request can be a mutation or a query.

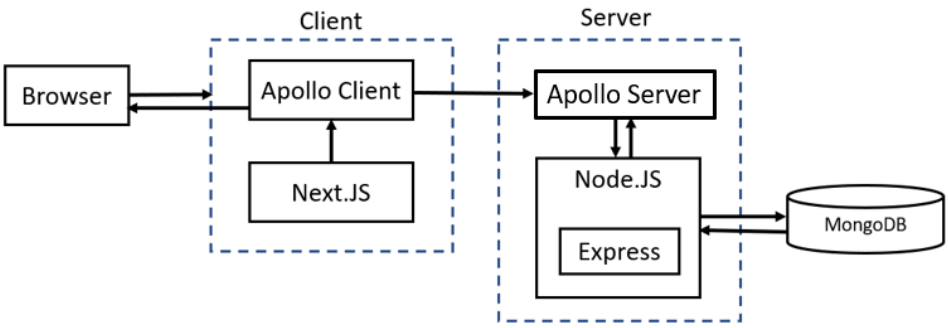

Figure 10. Architectural diagram

The front-end client includes the Next.JS application, connects with the Apollo Client. Apollo Client will handle the GraphQL APIs, contain the request and send to the server. The server is an Express app with Node.JS environment and Apollo Server. Apollo Server sends the Node.JS CRUD response back to the client. This backend is linked directly with the MongoDB cloud database.

4 DATABASE AND GUI DESIGN

4.1 Design of the Database

This database follows the NoSQL database architecture, which comprises different separate schemas, each schema represents a data type. Based on the defined Schema definition in GraphQL, we create the appropriate MongoDB schema. This will help Apollo Server to shape the data format faster to send the response to the client.

In this database, the three core data types in this database are Teacher, Student, and Courses.

4.1.1 Course Schema

The Course type in GraphQL is as follows:

```
type Course {
    id: ID!
    name: String!
    category: [String!]!
    teacher: Teacher!
    description: String!
    lessons: [Lesson]
    estimateTime: Float
    students: [StudyProgress!]
}
```

Code Snippet 1. Course type definition in GraphQL

The **students** property obtains all enrolled students in one course. This value is not saved directly in the database, but it obtains all students saved in Study Progress database. Hence, we have the Course Schema definition in MongoDB:

```
const courseSchema = new Schema({
  name: {
```

```
      type: String,
      required: true,
      minlength: 5
    },
    category: {
      type: [String],
      required: true,
    },
    teacher: {
      type: Schema.Types.ObjectId,
      ref: 'Teacher',
      required: true
    },
    description: {
      type: String,
      required: true,
    },
    estimateTime: Number,
    lessons: [{
      type: Schema.Types.ObjectId,
      ref: "Lesson"
    }]
  });
```

Code Snippet 2. Course schema in MongoDB

The **Schema.Types.ObjectId** type is the one that refers a data model to another data model. In this case, the course model refers the teacher model and the lessons model. The Lesson type's definition in GraphQL is shown in Code Snippet 3.

```
type Lesson {
    id: ID!
    title: String!
    content: String!
    quiz: [Quiz!]
  }
```

Code Snippet 3. Lesson type in GraphQL

The architecture of schema for the Lesson in MongoDB is not different with its declared type, which is:

```
 const lessonSchema = new Schema({
  title: {
    type: String,
    required: true,
  },
  content: {
    type: String,
    required: true
  },
  quiz: [
    {
      type: Schema.Types.ObjectId,
      ref: "Quiz"
    }
  ]
});
```

Code Snippet 4. Lesson Schema in MongoDB

Similarly, the lesson model refers to the quiz model. The Quiz type's definition in GraphQL is given below in Code Snippet 5.

```
type Quiz {
  id: ID!
  question: String!
  choices: [String!]
  answer: String!
}
```

Code Snippet 5. Quiz type in GraphQL

The architecture of schema for the Quiz in MongoDB is not different with its declared type, which is:

```
const quizSchema = new Schema({
  question: {
    type: String,
    required: true,
  },
  choices: [{
    type: String,
    required: true
  }],
  answer: {
    type: String,
    required: true,
  },
});
```

Code Snippet 6. Quiz schema in MongoDB

4.1.2 Student Schema

Code Snippet 7 shows the Student type's definition in GraphQL.

```
type Student {
    id: ID!
    name: String!
    email: String!
```

```
    passwordHash: String!
    studyProgress: [StudyProgress!]
}
```

Code Snippet 7. Student type definition in GraphQL

The **studyProgress** property obtains all studying status for the student's all enrolled courses recently. This value does not save directly in the database, but it obtains all data saved in Study Progress database. Hence, we have the Student Schema definition in MongoDB:

```
const studentSchema = new Schema({
  name: {
    type: String,
    minlength: [5, "Name length must be at least 5"],
    required: true
  },
  email: {
    type: String,
    minlength: [5, "Email length must be at least 5"],
    required: true,
    unique: true
  },
  passwordHash: {
    type: String,
    required: true,
  },
});
```

Code Snippet 8. Student schema in MongoDB

The email must be unique for the user identity. To allow schema in checking if one email has not been duplicated, the schema is plugged in with "mongoose-unique-validator" library, as in Code Snippet 9 below.

```
studentSchema.plugin(mongooseUniqueValidator)
```

Code Snippet 9. Plugin the schema with "mongoose-unique-validator" library

Based on Code Snippet above, the Student schema also contains a **studyProgress** property; however, it is not required. The StudyProgress type in GraphQL is shown in Code Snippet 10.

```
type StudyProgress {
    student: Student!
    course: Course!
    status: Status!
    overallPoint: Float
    startDate: String
    finishedDate: String
    progressPercentage: Float!
    lessonCompleted: [LessonCompleted!]
  }
```

Code Snippet 10. StudyProgress type definition in GraphQL

The StudyProgress schema in MongoDB is defined in Code Snippet 11.

```
const studyProgressSchema = new Schema({
  student: {
    type: Schema.Types.ObjectId,
    ref: "Student",
    required: true
  },
  course: {
    type: Schema.Types.ObjectId,
    ref: "Course",
    required: true
  },
  status: {
    type: String,
```

```
        enum: ["ONGOING", "PASSED", "FAILED"],
        required: true
      },
      overallPoint: Number,
      startDate: string
      finishedDate: String,
      progressPercentage: Number,
      lessonCompleted: [{
        lesson: {
          type: Schema.Types.ObjectId,
          ref: "Lesson",
        },
        point: Number,
        comments: [{
          quizID: String,
          answer: String,
          comment: String
        }]
      }]
    });
```

Code Snippet 11. StudentProgress schema in MongoDB

In the StudentProgress schema, the **status** property's document contains the **enum** validator. This validator accepts only Mongoose string and number type, and the returned value is one of the elements defined in that validator's array. Hence, when creating or updating a new StudyProgress, any other **status** besides those in **enum** array will not be accepted.

As seen in Code Snippet 10, there is a Status type in StudyProgress's GraphQL type definition. GraphQL also provides the **enum** type, whose purposes are similar to the **enum** property in MongoDB. Code Snippet 12 is the Status **enum** type in GraphQL.

```
enum Status {
    PASSED
    FAILED
    ONGOING
  }
```

Code Snippet 12. Status - an enum type definition in GraphQL

4.1.3 Teacher Schema

Code Snippet 13 below is Teacher type definition in GraphQL.

```
type Teacher {
  id: ID!
  name: String!
  email: String!
  passwordHash: String!
  organization: String
  ownCourses: [Course!]
}
```

Code Snippet 13. Teacher type definition in GraphQL

In Teacher type definition, the data **ownCourses** is always be fetched from the CourseModel and returns a course list. That fetched function applies only for the GraphQL type definition. It is unnecessary to save their own courses in the data model. Hence, we defined the Teacher schema in MongoDB as in Code Snippet 14 below.

```
const teacherSchema = new Schema({
  name: {
    type: String,
    minlength: [5, "Name length must be at least 5"],
    required: true
  },
  email: {
    type: String,
```

```
        minlength: [5, "Email length must be at least 5"],
        required: true,
        unique: true
      },
      passwordHash: {
        type: String,
        required: true,
      },
      organization: {
        type: String,
        minlength: 2,
      },
    });
```

Code Snippet 14. Teacher schema in MongoDB

Finally, we get the database collection obtained from MongoDB Cloud service:

test

VISUALIZE YOUR DATA

LOGICAL DATA SIZE: 6.34KB STORAGE SIZE: 180KB INDEX SIZE: 252KB TOTAL COLLECTIONS: 5

CREATE COLLECTION

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
courses	2	590B	295B	36KB	1	36KB	36KB
lessons	3	1.03KB	351B	36KB	1	36KB	36KB
quizzes	7	1.05KB	154B	36KB	1	36KB	36KB
students	3	2.45KB	836B	36KB	2	72KB	36KB
teachers	6	1.23KB	211B	36KB	2	72KB	36KB

Figure 11. MongoDB database collections

4.2 GUI Design

Every user must sign in before entering the homepage. The sign-in form contains an email box and a password box. In case any box is empty, the window will alert the user to fill in the box. A notification will be shown in case any error occurs.

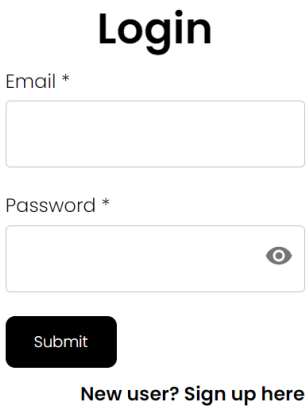

Figure 12. Sign-in form

Users must sign up if they have not been registered yet. After successfully signing up, the application returns to the login form. Users must also fill in all necessary boxes to not get an alert from the browser. The UI of the sign-up form is shown in Figure 13.

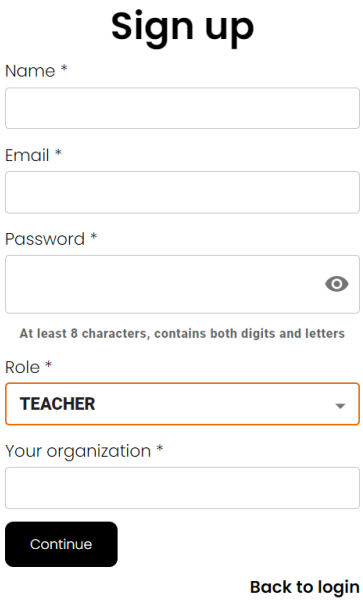

Figure 13. Sign-up form

The role field is a Selection box, with two different options: “Student” and “Teacher”. If the user selects “Teacher”, the “Organization” field will be displayed. When the sign in is done, the client will forward the user to the sign-in form to their first sign-in to the service.

After successfully signing in, the user will see the homepage. The homepage contains some suggested courses and a navigation bar. If the user is a teacher, he/she can see his/her own courses. The teacher can see the “Create the new course” part at the end of the page, where he/she will be directed to the Course Creation Form if he/she presses the “Start” button. The teacher’s home page can be seen in Figure 14.

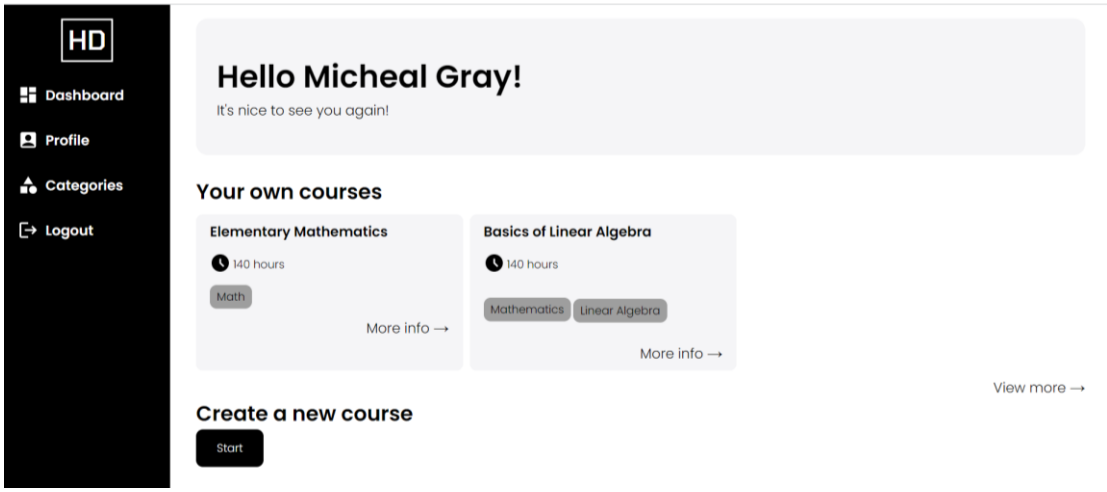

Figure 14. Teacher's home page

The student’s home page is displayed in Figure 15.

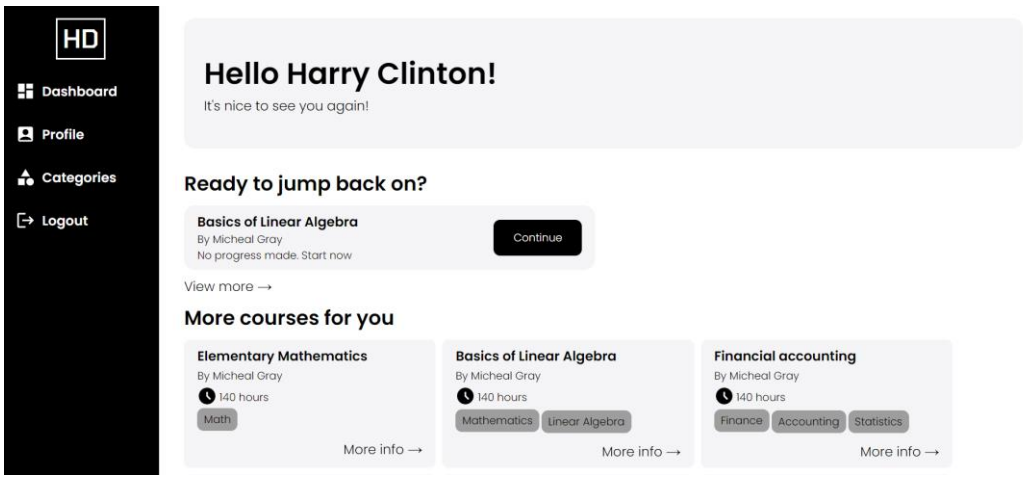

Figure 15. Student's home page

The student’s home page is different from the teacher’s home page, which contains student’s currently studying courses.

In the Profile page, the user can see their basic details, including with a toggle board, which contains two tabs. Basically, the designs of these pages are not too different from each other; however, students will see their study progress with the list of current study courses and completed courses. On the other hand, teachers can see their own courses. The teacher's profile also has the Create course button, which also directs to the Course Creation Form.

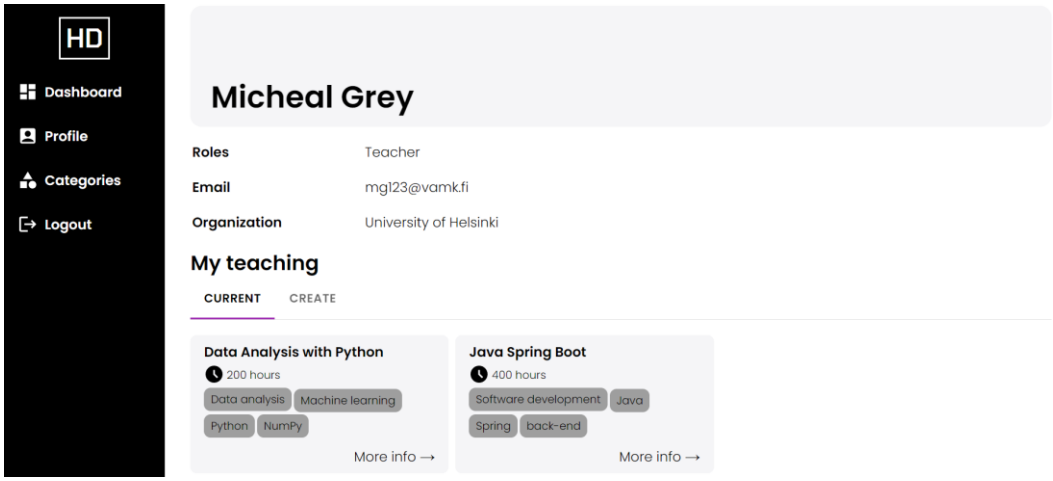

Figure 16. Teacher's profile page

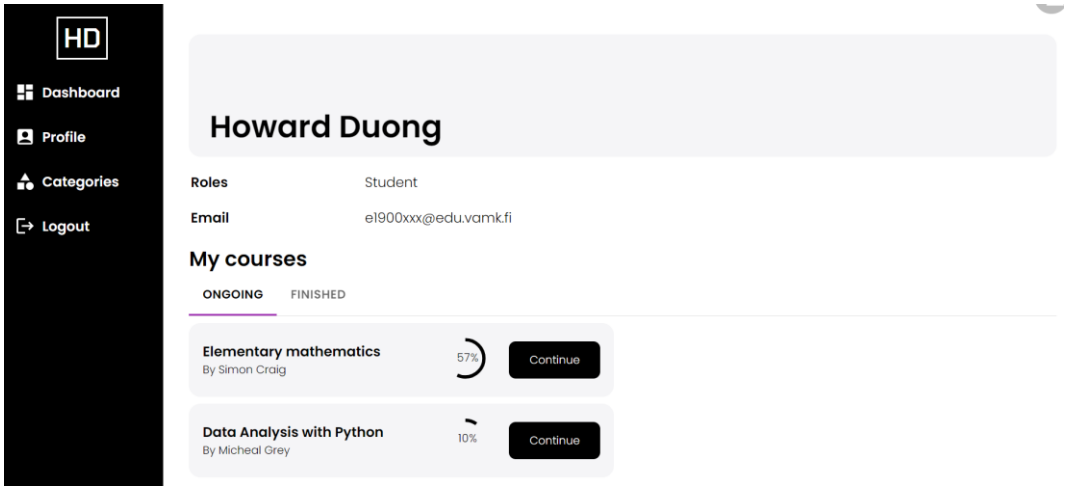

Figure 17. Student's profile page

All users can see their profile, containing the basic details. In case the user's role is Teacher, he/she can see their own courses and a button to create a new course. By clicking that button, the teacher will be forwarded to a course creation page, which is in Figure 18 below.

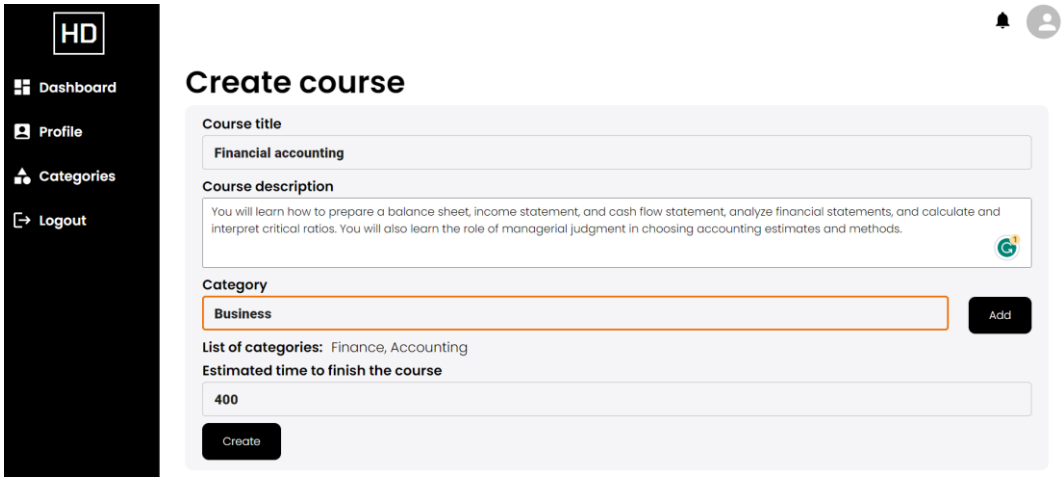

Figure 18. Course creation page

The server will update the cache with the course that successfully originated, hence, anyone can see it on pages containing courses data, such as the Dashboard, and the Category page. Figure 19 below is the Category page.

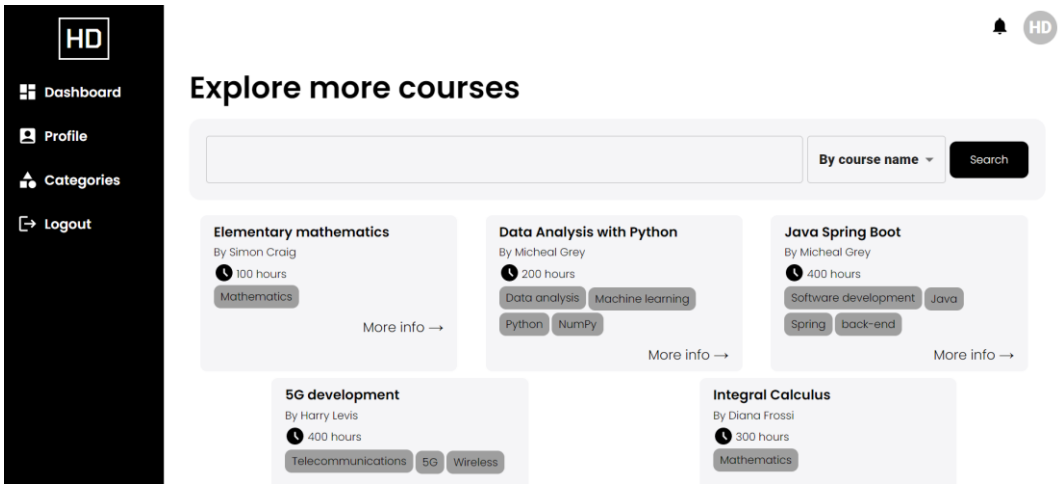

Figure 19. Category page

In the Category page, there is a filtering input and a selection box with two options: filter by name and by category. By pressing the “Search” button, the search query works and updates the courses list below the form.

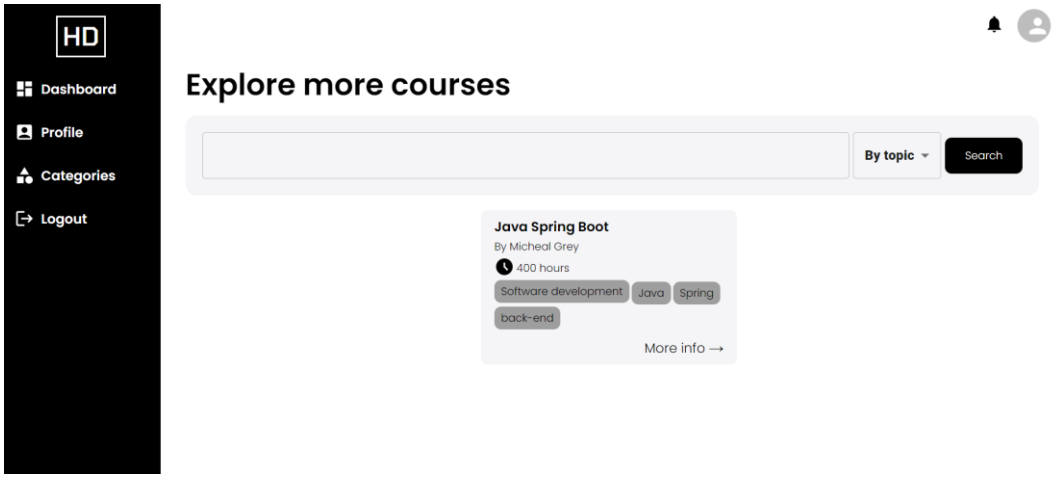

Figure 20. Result after searching successfully.

The list comprises many different course cards, whose design is same as the one in the Dashboard page for students. By clicking the card, the course main page will be displayed with basic information about the course. If the user is a Student, he/she can see the enroll button if their **studyProgress** list does not contain the course ID. The client returns a success message if the enrollment passes, or a failed alert if there is an error.

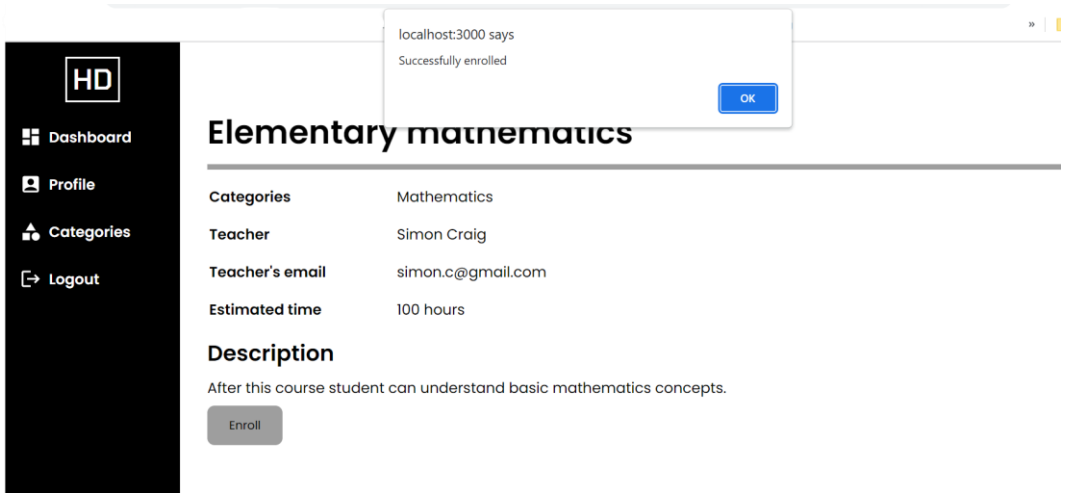

Figure 21. Successfully enrolled message

The student must enrol the course successfully to see its content. The course's owner can always view the content by default.

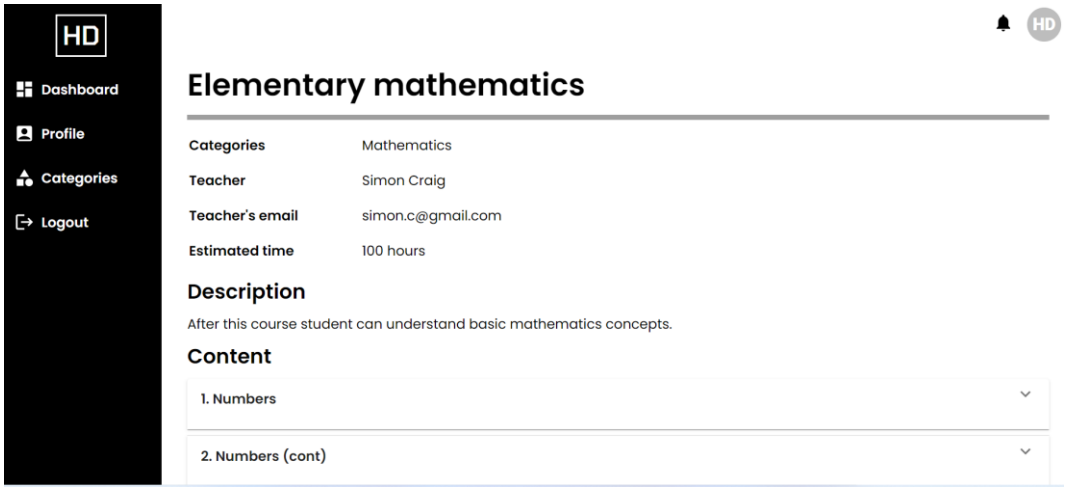

Figure 22. A course's main page

Each lesson card in the Content part is a Material UI's Accordion object (Material UI is a React library containing many production-ready components). This object contains two links at maximum: One is the content, the other is the quiz link.

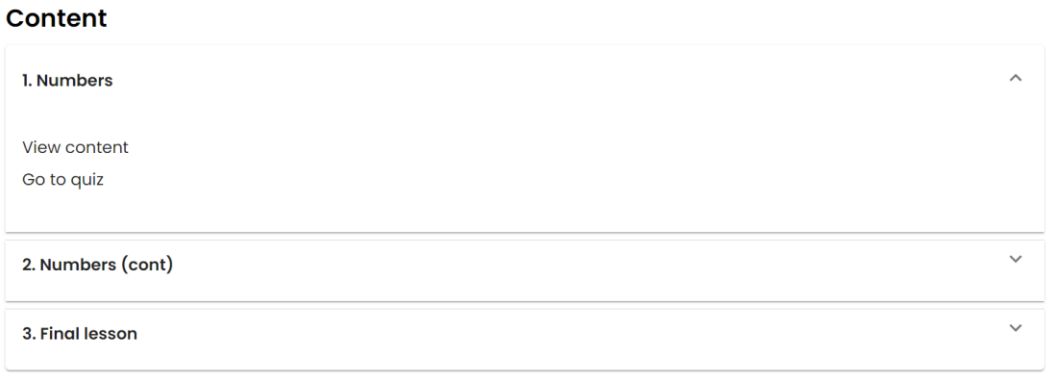

Figure 23. Course lesson's card

In the course page for Teacher, there is a "Add new lesson" button at the beginning to create a new lesson and "Add new quiz" in every Lesson card. Each button will direct to a form and the user can only create one lesson or one quiz per time. Figure 24 below shows the Content area in the course page for the course's owner.

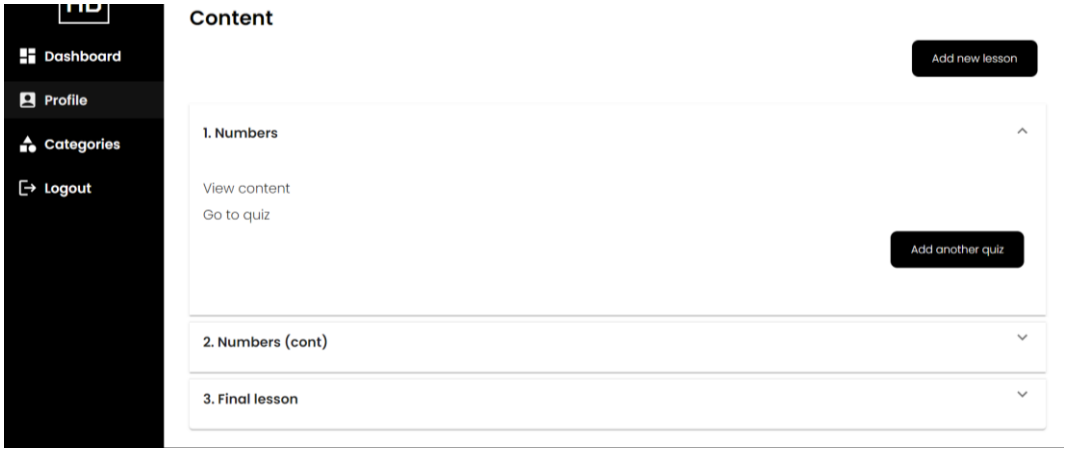

Figure 24. Course owner's lesson content

The course's content creation form is shown in Figure 25.

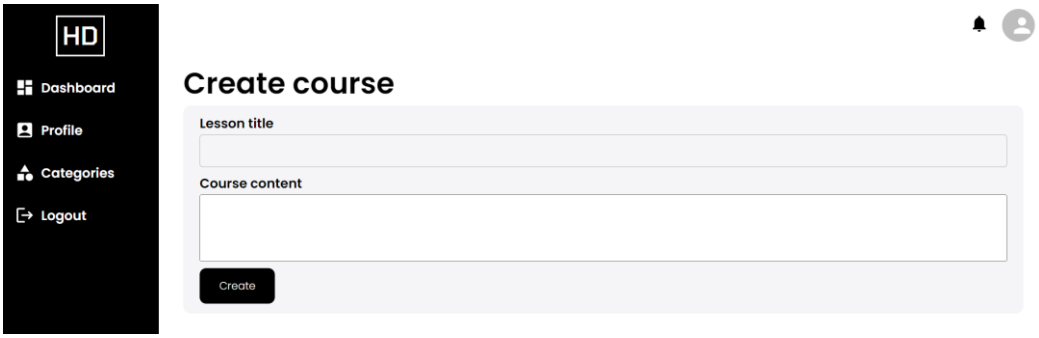

Figure 25. Content creation form

Figure 26 is a confirmation message after successfully creating a new course.

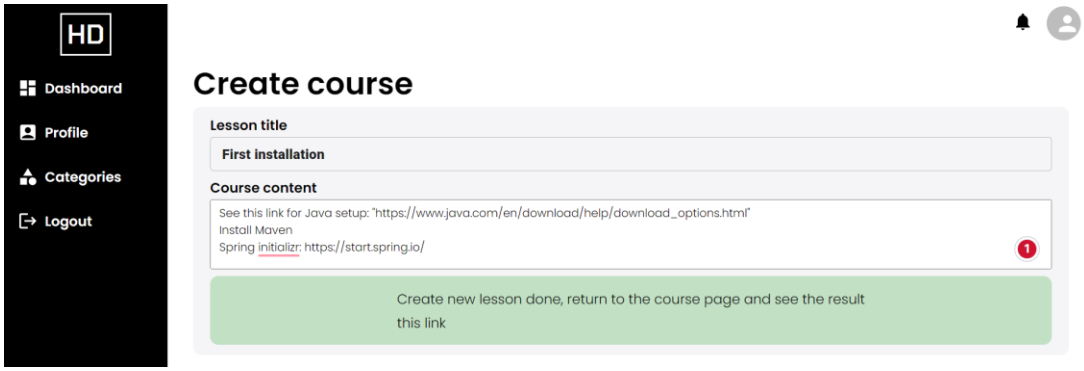

Figure 26. Create content success, which returns a message.

When the creation succeeds, the user will receive a confirmation message, which contains a link return to the course page.

The quiz creation form is shown in Figure 27, comprises the fields of quiz's question, correct answer, and the choices. The choices can be left empty, which means that students must fill in the correct answer by typing.

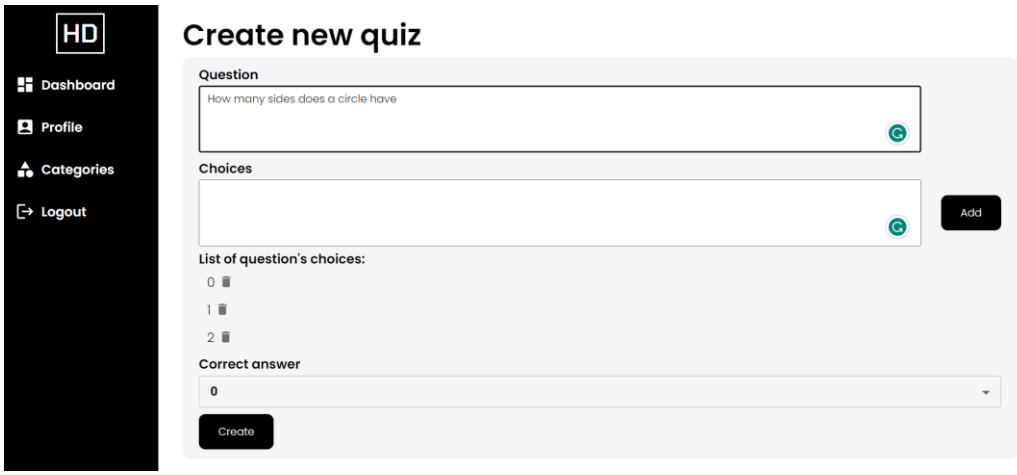

Figure 27. Quiz creation form

When the creation is accepted, the user will receive a confirmation message, which contains a link return to the course page.

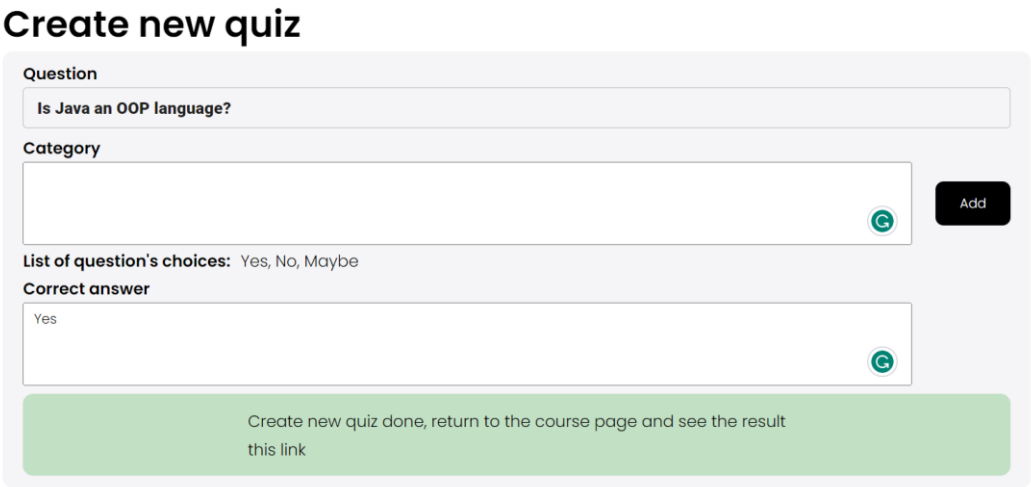

Figure 28. Successful announcement after successfully creating a quiz.

By following the link attached in "this link" **** tag, the user gets back to the course page with the updated data. The new quiz can be seen when clicking to the "Go to quiz" link.

For each lesson card, the teacher can see the list of students enrolled for the course.

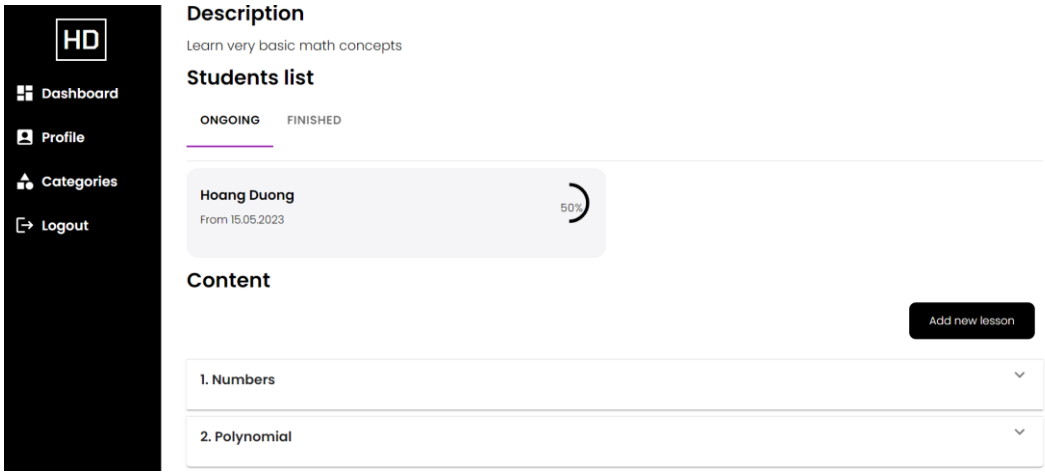

Figure 29. Course's enrolled student list

Any enrolled student or course owner can see the content and/or this quiz based on their selection between two links: “View content” and “Go to quiz”. Figure 30 below shows a sample of the content page.

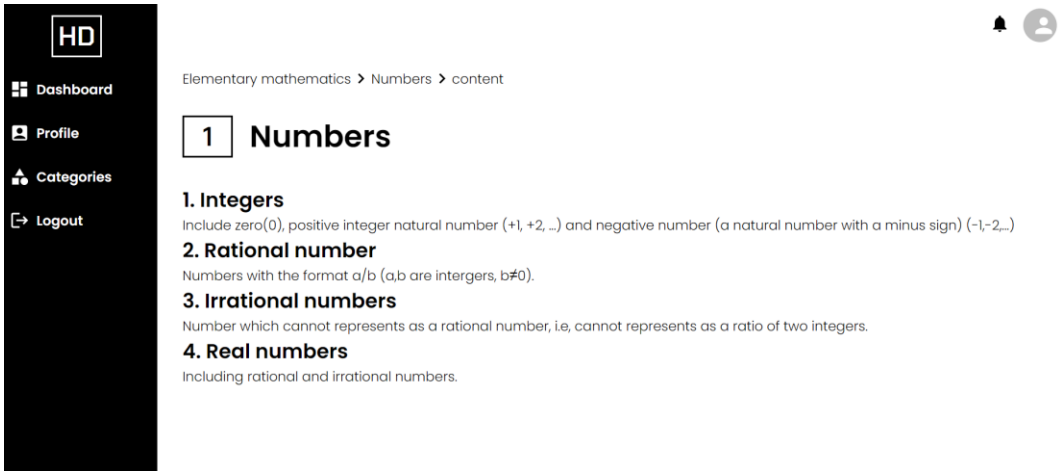

Figure 30. Course content page

On the quiz page, the selection will be disabled if the user is Teacher, or if student has done that quiz before. After finishing the quiz, the student receives the score and comment for each question in the quiz, also he/she can see the overall score as well. Figure 31 below is an example of a quiz page.

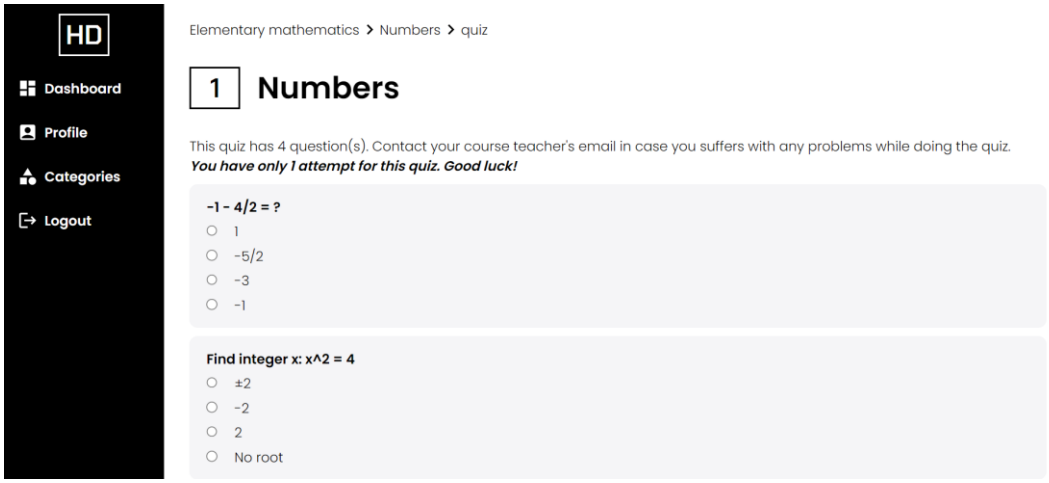

Figure 31. Quiz page

After submitting the answer, the quiz will be closed, and a confirmation message appears. Figure 32 shows how it works.

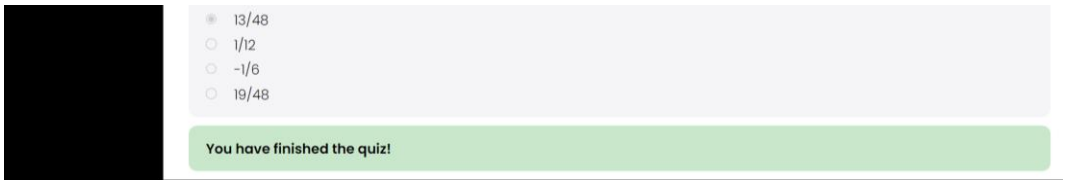

Figure 32. An announcement of finishing the quiz

The student's overall result is automatically re-fetch for each two seconds, so the quiz result will pop up soon after the submission, for example in Figure 33 below. The result contains the final score and the comment for each question.

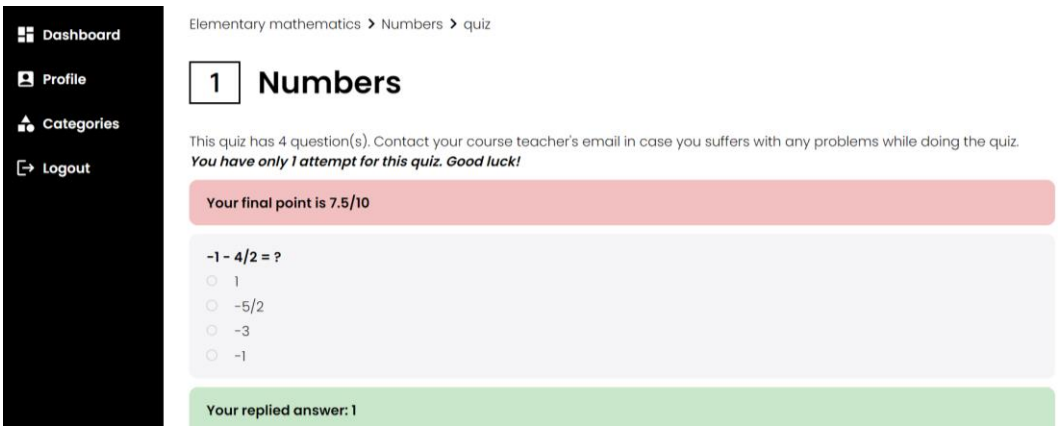

Figure 33. Quiz result page

After finishing the course (i.e., the percentage of study progress is 100%), the student will see an announcement board with the overall point (on the scale of 10), alongside with the finished date and the teacher's name.

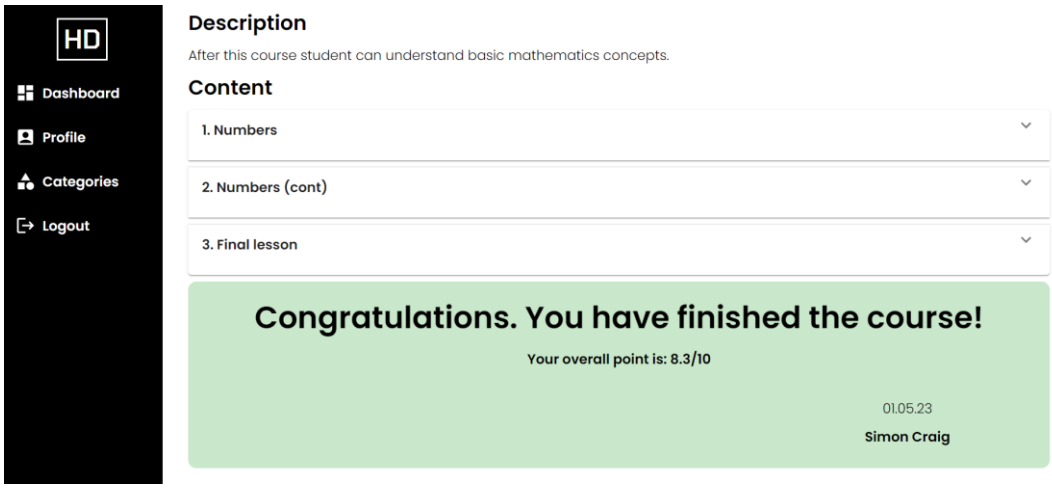

Figure 34. Course completion confirmation box

5 IMPLEMENTATION

5.1 General Description

This project uses Visual Studio Code (VSC) as the code editor. Before setting up the client and the server, Node.JS must be included with the version from 10.13 or later, so that Next.JS can work normally. Some VSC extensions must be implemented as well, especially:

- ES7+ React/Redux/React-Native/JS snippets.
- ES7 JavaScript/Node/Mongoose/MongoDB-Mysql snippets.
- Gitignore extensions: prevent GitHub from committing undesired files, such as ".env" (environment variables) file.

Since the programming language is TypeScript, the project needs to be emitted to JavaScript project before being deployed. Hence, this command must always be run before working with the project:

```
npx run tsc -- --init
```

Code Snippet 15. Run tsc (TypeScript compiler)

This command will automatically create a tsconfig.json file, which contains an object with a set of requirements that the root directory and its children folder inside it must follow. In this part, the report introduces some of the most important aspects in every configuration file. Every configuration file must have a Compiler Options property, which defines the compiling rules for a TypeScript project. Inside this property, some important aspects must be noticed are:

- **"target":** to define the JavaScript standard version that the developer wants to emit the JavaScript code and support implement libraries. Most browsers nowadays work with ES6+. This project selects ES2022, one of the latest versions.

- **“modules”:** to declare how the modules import/export in the project. This project uses “NodeNext”, which is compatible with the Next.JS project and the declared target value.
- **“outDir”:** to define the folder for all emitted files located in.
- **“allowSyntheticDefaultImports”:** Enables the format “import x from y” (with x, y is the modules).
- **“strict”**: this will ensure all strict type-checking before compiling in the project.
- **“baseUrl”:** to define the base directory of all TypeScript code, which decreases the time importing the whole URL of the modules.

The property **“include”** helps the tsconfig identify the files that applies the rules declared in the compilerOptions. On the other hand, **“exclude”** property prevents tsconfig from checking the files inside that property’s array, especially node_modules folder.

5.1.1 Front-end Client

Figure 35 below is the main structure of the Next.JS front-end.

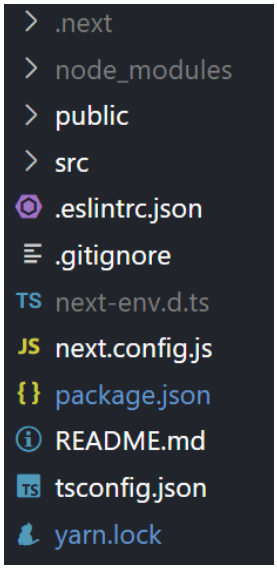

Figure 35. Main structure of Next.JS front-end

We will focus on the **/src** folder, whose structure is mentioned in Figure 36 below.

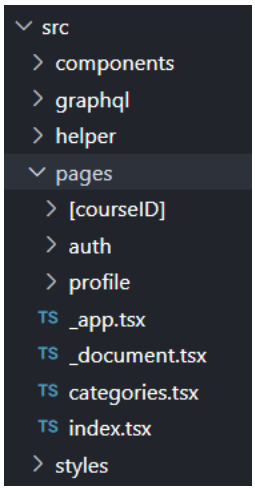

Figure 36. Next.JS /src folder structure

Inside the **/page** folder, there is an **_app.tsx**, which contains a Component object. This object, in fact, is an active object. As mentioned above, folder **/pages** is designed for dynamic routing. Hence, Next.JS will forward to the correct file in the **/pages** folder by following the routing and Component will become the component inside that file. The **_document.tsx** returns the root HTML file that will pop-up in the browser.

The **/src** folder includes **components** folder, which includes a set of small components for building each individual page, while the **graphql** folder includes a set of query and mutation requests and the client settings, and the **helper** folder contains helper functions for displaying date and converting Unicode symbols.

Moreover, as can be seen in section 4.2, all pages in the application have the same layout. This layout is taken from the Layout component located inside the Component folder.

```
<main className={styles.main}>
  <Sidebar logout={logout} />
  <div className={styles.mainPart}>
    <NavBar />
    <div className={styles.content}>
```

```
            {children}
          </div>
        </div>
      </main>
```

Code Snippet 16. The Layout component

This layout gets children components as a parameter. Those are taken from the **_app.tsx** file by wrapping the Layout around the Component object.

```
<Layout>
  <Component {...pageProps} />
</Layout>
```

Code Snippet 17. Wrapping Layout around the Component object

Styling files are in the same folder, which is the default setting for any Next.JS project. The default CSS file name formula for any Components is **<Component-Name>.module.css**. Any other CSS or SCSS file whose name format is different and/or outside the **styles** folder must be declared in the **_app.tsx** files. Finally, we got the styling folder, shown in Figure 37 below.

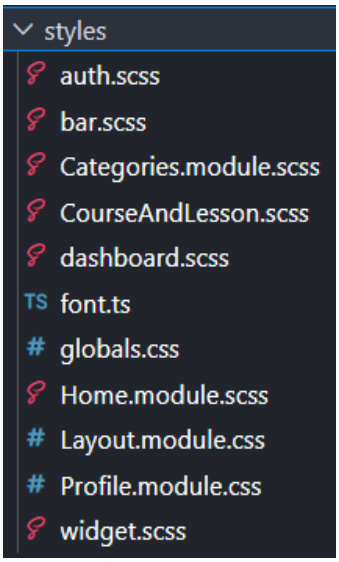

Figure 37. Styling folder

Any pages inside the **/pages** folder must contain a Head component at the very beginning so that it can run normally when user routes that file on the browser's

address bar. Otherwise, it will return the default **<Head>** from the **_document.tsx** file.

```
<Head>
    <title>Home | HD learning platform</title>
    <meta name="description" content="Generated by create next app" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="icon" href="/favicon.ico" />
</Head>
```

Code Snippet 18. Head component in the root index.tsx file

5.1.2 Setting up the Back End.

First, we discuss the main structure of the back-end.

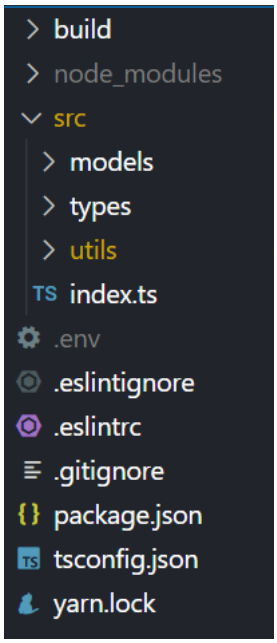

Figure 38. File structure of the server

The model folder contains a list of MongoDB data models. Here is the structure.

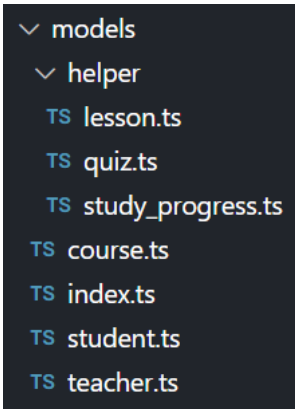

Figure 39. Folder of MongoDB models

The **type** folder contains TypeScript type declaration.

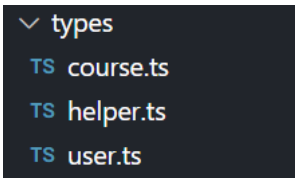

Figure 40. Types folder

The **user.ts** defines the type of User and two inheritance classes: Student and Teacher. The **course.ts** defines the Course type and the **helper.ts** contains helper types, including StudyProgress, Lesson, Quiz, and Comment.

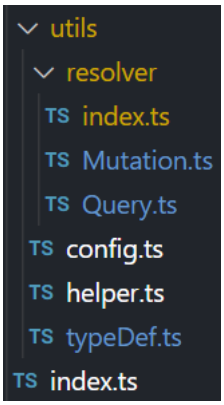

Figure 41. The utils folder

The **utils** folder contains **helper.ts** file, a GraphQL type definition and a resolver. The **helper.ts** file supports the server in some calculation problems. The **index.ts** file in resolver folder returns an object containing Query, Mutation, and some

other resolvers with special settings, for example, **__resolveType** for Type classification.

5.1.3 Setting up Apollo Server

First, the Apollo Server is installed by the command in Code Snippet 19 below.

```
yarn run @apollo/server graphql
```

Code Snippet 19. Install Apollo Server command.

The Apollo Server will be implemented as a part of the Express app. Creating a new server must include two required parameters: **typeDef** and **resolvers**. The server must be ready before processing the other Express middleware. The **plugins** variable is used to attach the HTTP server into the Apollo Server, to handle the further Express functions in this project.

```
const app = express();
const httpServer = http.createServer(app);
const server = new ApolloServer({
    typeDefs,
    resolvers,
   plugins: [ApolloServerPluginDrainHttpServer({
      httpServer
      })],
  });
await server.start()
```

Code Snippet 20. Setting up Apollo Server

The **ApolloServerPluginDrainHttpServer** is used to make the server shut down gracefully (i.e., server after shutting down cannot be approached by any new connections. It also quickly sets all remaining connections to be idle and close them). /20/.

In this project, we will use **expressMiddleware** function in Apollo Client, which attached defined server with the Express app. The code snippet below is the Express middleware, which also handles CORS, JSON-parser, and contains the **expressMiddleware** function.

```
app.use("/",
    cors<cors.CorsRequest>(),
    express.json(),
     expressMiddleware(server, {context: async ({req}) => {
        const auth = req
              ? req.headers.authorization
              : null;
        if (auth && auth.startsWith("Bearer ")) {
          const decodedToken = jwt.verify(
                    auth.substring(7), SECRET);
          const currentUser = await StudentModel
              .findById(decodedToken.id)
              .populate({path: "studyProgress",
                         populate: ["course", {
                          path: "lessonCompleted",
                          populate: ["comments", "lesson"]
                         }],
                    });
          if (!currentUser) {
            const teacher = await TeacherModel
                    .findById(decodedToken.id);
            return { currentUser: teacher };
          } return { currentUser };
        } else return { currentUser: null };
      },
    }));
```

Code Snippet 21. Set up the Express middleware.

The **expressMiddleware** contains two parameters, first is the server, the second one is the **context** object. The **context** value is an asynchronous function that processes the header received from the GraphQL APIs. In this project, contextValue's function returns the current sign-in user, which is used for all query and mutation operations in the server.

5.1.4 Setting up Apollo Client

First, we must install the library:

```
yarn run @apollo/client graphql
```

Code Snippet 22. Install Apollo Client

The code snippet below is a new ApolloClient object. It must be created and connect with the server, so that they can communicate easily. The **InMemoryCache** function will cache the memory during running all queries and mutations in this app.

```
const HttpLink = createHttpLink({
  uri: "http://localhost:4000/",
})
const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: authLink.concat(HttpLink),
})
```

Code Snippet 23. Create new Apollo Client with the server URL.

The **authLink** is a function which returns the header of the GraphQL APIs request. This header is collected after successfully sign in. The client will then become a parameter in the **ApolloProvider** object, which is wrapped into the main component, and it is ready to connect with the server.

```
<ApolloProvider client={client}>
  <Layout>
    <Component {...pageProps} />
  </Layout>
</ApolloProvider>
```

Code Snippet 24. Wrap the ApolloProvider with the defined client into Layout component.

5.2 Implementation of Various Parts

5.2.1 User Authentication

The authentication process starts when user presses "Submit" button in log-in form. The function **onSubmit**, which is a parameter in the form, starts working. The hook **useMutation** will get the email and password as the parameters and send the request to the server.

```
const [loginMutation] = useMutation(LOGIN, {
    onError: (error) => setNotification(error.message),
    onCompleted: (data) => {
      setToken(data.login)
     localStorage.setItem("new-user-token",
           data.login)
    },
  })
const submit = (event: any) => {
    event.preventDefault()
    loginMutation({ variables: { email, password } })
}
```

Code Snippet 25. Login useMutation hook and the submitting action.

The **LOGIN** variable contains a mutation log-in request in GraphQL, wrapping in a **gql** template literal from **"@apollo/client"** library with two backticks. The request

can be created in Apollo Server Sandbox. It has a structure as shown in Code Snippet 26 below.

```
const LOGIN = gql`
     mutation Login($email: String!, $pasword: String!) {
       login(email: $email, password: $password)
     }`
```

Code Snippet 26. LOG_IN variable

This request will be sent to the server for handling data. Here is the login function:

```
login: async (_root: any, args:
   {email: string; password: string}) => {
   const findTeacher = await TeacherModel
              .findOne({ email: args.email });
       const user = findTeacher
         ? findTeacher
         : await StudentModel.findOne({email: args.email});
       const password = !user
         ? false
         : await bcrypt.compare(
           args.password, user.passwordHash);
       if (!(user && password)) {
    throw new GraphQLError("Invalid username or password", {
           extensions: {
             code: "BAD_USER_INPUT",
           },
         });
       }
       const token = jwt.sign({
           id: user._id,
           email: user.email }, config.SECRET);
```

```
    return `Bearer ${token}`;
  },
```

Code Snippet 27. Log-in mutation function in the back-end

If the **loginMutation** returns data, the **onCompleted** property will be enabled and sends the token to the browser's local storage. The function **authLink** defined below will parse into the authorization header before sending it to the client.

```
const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem("new-user-token")
  return {
    headers: {
      ...headers,
      authorization: token ? token : null,
    },
  }
})
```

Code Snippet 28. Function authLink in the front-end

The server will now receive the request with a header. By using the function inside the context value, which was defined in Code Snippet 21, the user data will be returned and saved in the **currentUser** property, which belongs to the **contextValue**. With a new token, user will be forwarded into the homepage, where the **GET_USER** query is fetched to obtain the user's data. The loading, error, data are three attributes obtained from any **useQuery** hook.

```
const { loading, error, data } = useQuery(GET_USER)
```

Code Snippet 29. Getting user data from the homepage

Here is the GET_USER query request taken from the Apollo Server Sandbox:

```
const GET_USER = gql
`query Query {
  getUser {
    ... on Student {
      Email
      Id
      name
    }
    ... on Teacher {
      email
      name
      organization
      passwordHash
      ownCourses {
        id
      }
    }
  }
}`
```

Code Snippet 30. The GET_USER variable.

Here is the back-end code for getting user information:

```
getUser: (
   _root: any,
   _args: any,
   contextValue: { currentUser?: any }) => {
       if (contextValue.currentUser) {
            return contextValue.currentUser;
       }
   },
```

Code Snippet 31. Get user data in back-end server.

The query has two cases, since in the GraphQL type definition in the server, there is a User type, which is the union between Teacher and Student type.

```
union User = Student | Teacher
```

Code Snippet 32. User, a Union type definition in GraphQL

In this case, to clarify the exact role (i.e., the data types), we must define it in the resolvers the **__resolveType** function:

```
User: {
    __resolveType(
        user: any,
        _contextValue: any,
        _info: any) {
      if (user.organization) {
        return "Teacher";
      }
      else if(user) {
        return "Student";
      }
      else return null;
    }
  },
```

Code Snippet 33. User __resolveType() function

The final data will be returned with the correct type of log-in user. This data will save in **{data}** variables, which has been defined in Code Snippet 29. If the server is loading, the **{loading}** is defined, a the **{error}** is available when any error occurs.

5.2.2 Sign up process.

The hook **useMutation** will get all necessary data and process it as a request to the server. If the response status is 200, the property **onCompleted** will be enabled, otherwise the property **onError** will be enabled.

```
const [signUpMutation] = useMutation(SIGN_UP, {
    onError: (error) => setNotification(error.message),
    onCompleted: (data) => {
      alert(
        `Welcome ${data.createUser.name} as a
              ${data.createUser.__typename}`
        )
      setNewUser(false)
    },
  })
```

Code Snippet 34. The sign-up function inside the useMutation hook

Here is the SIGN_UP mutation request taken from the Apollo Server Sandbox.

```
mutation Mutation($name: String!, $email: String!,
              $password: String!, $organization: String) {
    createUser(name: $name, email: $email, password:
              $password, organization: $organization) {
      ... on Student {
        email
        id
        name
      }
      ... on Teacher {
        email
        id
        name
        organization
      }
   }}
```

Code Snippet 35. SIGN_UP mutation variable

It returns the User object, which has been mentioned before. The response data for both cases (Student and Teacher) is not too different, except that teacher's data will include also his/her working organization.

In the sign-up form, the user's input data will be obtained by clicking the submit function, which is the value of parameter **onClick** in the **<form>** tag. In the submit function, data will be classified based on the defined role between Student and Teacher, to ensure the server adds the correct data to the appropriate data model. The submit function is given below in Code Snippet 36.

```
const submit = (event: any) => {
   event.preventDefault()
   if (user.role === "TEACHER") {
     signUpMutation({
       variables: {
         name: user.name,
         email: user.email,
         password,
         organization,
       },
     })
   } else {
     signUpMutation({
       variables: {
         name: user.name,
         email: user.email,
         password,
       },
     })}
   setPassword("")
   setUser({
     name: "",
```

```
        email: "",
        role: "STUDENT",
      })
    }
```

Code Snippet 36. Submit sign-up mutation function.

The code snippet below is the back-end function for creating a user.

```
    if (
      args.password.length < 8 ||
      !/\d/.test(args.password) ||
      !/[a-zA-Z]/.test(args.password)
    ) {
      throw new GraphQLError(
        "Wrong password format (>=8 and contains at least 1
letter and 1 digit)",
        {
          extensions: {
            code: "GRAPHQL_VALIDATION_FAILED",
          },
        }
      )}
```

Code Snippet 37. Password creation verification

The function will check if the user has typed a password the minimum length of which is 8 characters, containing both digits and letters. After passing this round, the password will be encrypted by the hash function in **bcrypt** library with the salt value is 10. It returns a hashed password, and this password will be saved into the database.

```
    const saltRounds = 10;
    const passwordHash = await bcrypt.hash(args.password,
saltRounds);
    const { name, email } = args;
```

```
const newUser = !args.organization
  ? new StudentModel({
    name,
    email,
    passwordHash,
    studyProgress: [],
  })
  : new TeacherModel({
    name,
    email,
    passwordHash,
    organization: args.organization,
  });
  await newUser.save();
```

Code Snippet 38. Save the new user into MongoDB database.

The result will be sent to the client. Now the user must sign-in to get the token for accessing the page.

5.2.3 Processing Student's Answers for the Quiz

Before handling the mutation process, the application must check if student has already done the quiz by sending this query to the server:

```
const overallResult = useQuery(GET_QUIZ_RESULT, {
    variables: { courseId: courseID, lessonId: lessonID },
      pollInterval: 2000
  })
```

Code Snippet 39. Get the quiz result query.

The poll interval is used to set the time in milliseconds that the query will re-fetch after the latest call. The GET_QUIZ_RESULT will be defined in the Code Snippet below.

```
export const GET_QUIZ_RESULT = gql`
  query GetQuizResult($courseId: ID!, $lessonId: ID!) {
```

```
    getQuizResult(courseID: $courseId, lessonID: $les-
sonId) {
      comments {
        quizID
        answer
        comment
      }
      point
    }
  }`
```

Code Snippet 40. GET_QUIZ_RESULT query

The server will check and if student has already completed the quiz by searching the lesson, server returns the result automatically to the client.

```
const progress = await StudyProgressModel.findOne(
    { course: args.courseID }).populate({
      path: "lessonCompleted", populate: "lesson"
    });
    return progress
      ? progress.lessonCompleted.find( (obj) =>
         obj.lesson._id.toString() === args.lessonID)
      : null;
```

Code Snippet 41. Check if user has completed the quiz.

After receiving the student's answer for the quizzes, the server will compare the student's answer with the quiz answer and return the point on the scale of 10.

```
const quizPoints = (params: {
      data: Array<StudentAnswer>, lesson: any
}) => {
  let finalPoint = 0;
```

```
  let comments: { quizID: string, answer: string,
      comment: string }[] = [];
  for (const obj of params.lesson.quiz) {
    const quiz = params.data.find(ans =>
      ans.quizID === obj._id.toString());
    if (quiz) {
      if (obj.answer === quiz.answer.trim()) {
        finalPoint += 1;
        comments = comments.concat({
          quizID: obj._id.toString(),
          answer: quiz.answer,
          comment: `Your answer is correct!`
        });
      }
      else {
        comments = comments.concat({
          quizID: obj._id.toString(),
          answer: quiz.answer,
          comment: `Your answer is wrong, the correct
one is ${obj.answer}`
        });
      }
    } else {
      comments = comments.concat({
        quizID: obj._id.toString(),
        answer: "",
        comment: `You have not answer this question!,
the correct one is ${obj.answer}`
      });
    }
  }
  return {
    point: finalPoint*10/params.lesson.quiz.length,
```

```
    commentArray: comments
  };
};
```

Code Snippet 42. Quiz result processing function

The server will also calculate the student's study progress percentage and update that student's study progress object. The percentage will be the ratio between the number of quiz students have passed and the total of quiz in the course.

```
const progressCalculation = async
      (finished: any[], courseID: string) => {
const course = await CourseModel.findById(courseID)
                 .populate("lessons");
  if (course) {
    let quizNumber = 0;
    let completedQuiz = 0;
    for (const obj of course.lessons) {
      const lesson = await LessonModel
                       .findById(obj._id);
      if (lesson && lesson.quiz.length > 0) {
        quizNumber += lesson.quiz.length;
      }
    }
    for (const obj of finished) {
      if (obj.comments) {
        completedQuiz += obj.comments.length;
      }
    }
    return completedQuiz * 100 / quizNumber}
  else return 0
};
```

Code Snippet 43. Study progress percentage calculation

Finally, after exceeding 100% as the process percentage, the server will calculate the student's overall score for that course and evaluate if he/she passes the course. A score which at least 5 on the scale of 10 allows the student to pass the course.

```
const overallPointCalculation = (data: Array<{
  lesson: Types.ObjectId;
  point: number;
  comments: Comment[]
}>) => {
  const sum = data.reduce((prev, curr) => prev +
curr.point, 0);
  return sum / (data.length);
};
```

Code Snippet 44. Quiz overall result calculation

The data will be saved into the student's study progress object in the studyProgress array. If the student finishes the course, the study progress object will update the student's finished date.

6 TESTING

In the testing part in this application, two main testing were conducted: functional testing for the back end using Jest, and Cypress was used for the end-to-end testing in the front-end.

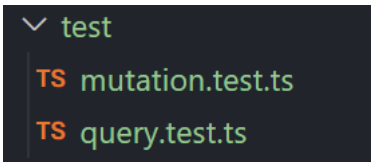

Figure 42. Testing the back-end folder

The test folder contains two files, which test the query and the mutation set of functions. Here is the testing report.

Table 2. Testing back-end report

Description	Test details	Number of tests	Number of fixed errors	Passed percentage (%)
Get all course basic data		1	0	100
Get the course by its id		1	0	100
Enroll course	Enroll by a teacher account. Student has enrolled before. Enroll by the regular way.	3	0	100
Answer quiz	Answers was missing one question. Answer full of questions.	2	0	100
Add course	Add course with a student identification. Add course by the regular way.	2	2	100
Add quiz	Add quiz whose lesson's ID was not correct. Add quiz whose answers different from the choice's selection. Add quiz by the regular way.	3	2	66.67

To test the front-end application, Cypress was applied, and the report of the test is given in Table 3 below.

Table 3. Testing front-end report

Description	Test details	Number of tests	Number of fixed errors	Passed percent-age (%)
Get all course basic data for each two seconds		1	0	100
View the course page by differ-ent cases	An owner A user with no permission An enrolled student	4	0	100
View the profile	The user is a teacher. The user is a student. The profile from another teacher	3	0	100
Search in the category page	Search by course name Search by key word	2	2	100
Log in and sign up		2	0	100
Test the layout		1	0	100
View quiz by different pur-poses	The course creator Students have already passed the quiz. Students have not yet done the quiz	3	0	100
Sending quiz answers	Answer all questions. Missing one question.	2	0	100
Create quiz and contents	Log in as a student. Log in as a teacher.	2	0	100

7 CONCLUSIONS

7.1 Achievements

In conclusion, this project successfully created an online learning platform that follows the MERN stack architecture. The Apollo Server and Apollo Client were implemented properly, so that the client and server can communicate easily by transferring the GraphQL APIs.

This platform works properly for both teachers and students, with a separate UI for each user case. Students can search for the course, enroll in any course, study its lectures, do quizzes, and get the course completion certificate. On the other hand, teachers can create a course, see the list of enrolled students and only they are allowed to update the course lectures and quizzes.

7.2 Challenges

The first challenge in building this platform was to find the most effective data architecture and define the most necessary queries and mutations in this project.

Another challenge was client creation and connecting the client with the server, since Next.JS had a bit different file organization than React.

The final challenge is also a considerable disadvantage about GraphQL, as it uses hook for managing data. The hook is a concept in React, which allows updating object's state and its properties value, but the data does not been updated right after the first render or click. This negatively affected the application by increasing the loading time for both client and server.

7.3 Future Development

This project has many potential points to develop in the future. First, every user must verify their email before signing up. Currently, the sign-in process is simple, since there is no email verification, so the user can even create an unreal email.

Secondly, the teacher functionalities must be advanced, so that they have more rights in managing the course data, controlling the student enrollment request. The quiz will be developed by adding the time, and there will be more different question types than the multiple-choice question.

The final improvement will be some features on the course page, which is a comment area for teachers and students to discuss about the course. There will also be more different lesson content types than just a reading text, for example PDF files and videos.

REFERENCES

1. State of JavaScript (2022a). Front-end frameworks. Accessed 07.05.2023. https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/
2. State of JavaScript (2022b). Rendering frameworks. Accessed 07.05.2023. https://2022.stateofjs.com/en-US/libraries/rendering-frameworks/
3. State of JavaScript (2022c). Other tools. Accessed 07.05.2023. https://2022.stateofjs.com/en-US/other-tools/
4. The National Center for Education Statistics (2022). Undergraduate Enrollment. Accessed 05.02.2023. https://nces.ed.gov/programs/coe/indicator/cha.
5. Wikipedia. React (JavaScript library). Accessed 05.02.2023. https://en.wikipedia.org/wiki/React_(JavaScript_library).
6. Surve, Suraj. (2021). Why You Should Use React.js For Web Development. Accessed 05.02.2023. https://www.freecodecamp.org/news/why-use-react-for-web-development/.
7. Riva, Michele. (2022). Real-World Next.js. Accessed 05.02.2023. https://learning.oreilly.com/library/view/real-world-next-js/9781801073493/.
8. Nihar, Raval. (2022). TypeScript vs JavaScript: The Difference You Should Know. Accessed 30.04.2023. https://radixweb.com/blog/typescript-vs-javascript.
9. javaTpoint. Difference between CSS and SCSS. Accessed 05.02.2023. https://www.javatpoint.com/css-vs-scss.
10. Hoque, Shama. (2020). Full-Stack React Projects - Second Edition. Accessed 06.02.2023. https://learning.oreilly.com/library/view/full-stack-react-projects/9781839215414/.
11. Kaneriya, Tejas. (2020). A brief introduction to how Node.js works. Accessed 06.02.2023. https://dev.to/tejaskaneriya/what-is-node-js-a-brief-introduction-to-how-node-js-works-26c8.
12. MongoDB. What is NoSQL? Accessed 07.05.2023. https://www.mongodb.com/nosql-explained.
13. Techopedia. Ad Hoc Query. Accessed 07.05.2023. https://www.techopedia.com/definition/30581/ad-hoc-query-sql-programming.

14. GraphQL Homepage. Accessed 03.06.2023. https://graphql.org/.
15. Apollo Docs a. Introduction to Apollo Client. Accessed 03.06.2023. https://www.apollographql.com/docs/react/
16. Apollo Docs b. Why Apollo Client? Accessed 28.04.2023. https://www.apollographql.com/docs/react/why-apollo
17. Apollo Docs c. Introduction to Apollo Server. Accessed 28.04.2023. https://www.apollographql.com/docs/apollo-server/
18. Apollo Docs d. API Reference: startStandaloneServer. Accessed 28.04.2023. https://www.apollographql.com/docs/apollo-server/api/standalone/
19. JWT. Introduction to JSON Web Tokens. Accessed 03.06.2023. https://jwt.io/introduction
20. Apollo Docs e. API Reference: Drain HTTP server plugin. Accessed 03.05.2023. https://www.apollographql.com/docs/apollo-server/api/plugin/drain-http-server/

APPENDIX 1

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "node",
    "baseUrl": "./src",
    "outDir": "./build/",
    "inlineSourceMap": true,
    "isolatedModules": true,
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "noFallthroughCasesInSwitch": true,
    "skipLibCheck": true
  },
  "ts-node": {
    "esm": true,
    "files": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Code Snippet 45. Server's tsconfig.json file

APPENDIX 2

```
answerQuiz: async (
      _root: any,
      args: {
            courseID: string
             lessonID: string
             answers: Array<{
               quizID: string
               answer: string
             }>
           },
    contextValue: { currentUser?: any }
  ) => {
    const lesson = await LessonModel.findById(args.lessonID);
    const progress = await StudyProgressModel.findOne({
      course: args.courseID,
    });
    if (!(contextValue.currentUser && lesson && progress)) {
      throw new GraphQLError("No result found", {
        extensions: { code: "BAD_USER_INPUT" },
      });
    }
    const result = await helper.quizPoints({ data: args.answers });
    const returnedQuizResult = {
      lesson: lesson._id,
      point: result.point,
      comments: result.commentArray,
    };
    try {
      progress.lessonCompleted =
        progress.lessonCompleted.concat(returnedQuizResult);
```

```
      progress.progressPercentage = await helper.progressCalculation(
        progress.lessonCompleted,
        args.courseID
      );
      if (progress.progressPercentage === 100) {
        const overallPoint = helper.overallPointCalculation(
          progress.lessonCompleted
        );
        progress.overallPoint = overallPoint;
        progress.status = overallPoint >= 5 ? "PASSED" : "FAILED";

        progress.finishedDate = new Date();
      }
      await progress.save();
    } catch (error) {
      throw new GraphQLError("Error in updating study progress", {
        extensions: {
          code: "GRAPHQL_VALIDATION_FAILED",
          error,
        },
      });
    }
    return returnedQuizResult;
  },
```

Code Snippet 46. Mutation processes the student's answer for the quiz.