Development of a Full Stack Mobile Application for Dynamic Quiz Generation with AI-Driven Hints using ChatGPT

**Bogdan TATU**

Computers and Information Technology   
**Faculty of Automation and Computers**

**Polytechnic University of Timisoara**

Timișoara, 06, 2023

Table of Contents

[1 The Quizzdos Project 11](#_Toc138528387)

[1.1 Introduction 11](#_Toc138528388)

[2 Specifications, requirements and design 12](#_Toc138528389)

[2.1 Similar applications 12](#_Toc138528390)

[2.1.1 MyFitnessPal 12](#_Toc138528391)

[2.1.2 Rust 12](#_Toc138528392)

[2.2 Project Requirements 12](#_Toc138528393)

[2.3 Project Specifications 13](#_Toc138528394)

[2.4 Block diagram 13](#_Toc138528395)

[2.5 Class diagrams 15](#_Toc138528396)

[2.6 Controllers 15](#_Toc138528397)

[2.7 Repositories 16](#_Toc138528398)

[3 Technical background 17](#_Toc138528399)

[3.1 Programming languages 17](#_Toc138528400)

[3.1.1 TypeScript (JavaScript) 17](#_Toc138528401)

[3.1.2 Rust 18](#_Toc138528402)

[3.2 Frontend technologies 18](#_Toc138528403)

[3.2.1 React Native 18](#_Toc138528404)

[3.2.2 Nativewind (Tailwind CSS) 18](#_Toc138528405)

[3.2.3 Axios 18](#_Toc138528406)

[3.2.4 TanStack Query 18](#_Toc138528407)

[3.2.5 React Native Reanimated 18](#_Toc138528408)

[3.2.6 Expo Libraries 18](#_Toc138528409)

[3.3 Backend technologies 18](#_Toc138528410)

[3.3.1 Diesel 18](#_Toc138528411)

[3.3.2 Rocket & Serde 18](#_Toc138528412)

[3.3.3 Reqwest 18](#_Toc138528413)

[3.3.4 Tokio 18](#_Toc138528414)

[4 Application utilization 18](#_Toc138528415)

[4.1 General 18](#_Toc138528416)

[4.2 Student Experience 19](#_Toc138528417)

[5 Gantt Chart 19](#_Toc138528418)

[5.1 Definitions 19](#_Toc138528419)

[5.2 The Project’s Gantt Chart 19](#_Toc138528420)

[6 Implementation 19](#_Toc138528421)

[6.1 Backend 20](#_Toc138528422)

[6.1.1 Models 20](#_Toc138528423)

[6.1.2 Database Diagram 24](#_Toc138528424)

[6.1.3 Migrations 24](#_Toc138528425)

[6.1.4 quizzdos-backend 26](#_Toc138528426)

[6.2 Frontend 51](#_Toc138528427)

[6.2.1 App.tsx 52](#_Toc138528428)

[6.2.2 Api 54](#_Toc138528429)

[6.2.3 Assets 55](#_Toc138528430)

[6.2.4 Components 55](#_Toc138528431)

[6.2.5 Constants 60](#_Toc138528432)

[6.2.6 Screens 60](#_Toc138528433)

[7 Testing and Validation 66](#_Toc138528434)

[7.1 Functional Testing 66](#_Toc138528435)

[7.2 User Interface Testing 66](#_Toc138528436)

[7.3 Error Handling and Validation 66](#_Toc138528437)

[7.4 Integration Testing 66](#_Toc138528438)

[7.5 Data Validation and Database Testing 66](#_Toc138528439)

[8 Conclusions 67](#_Toc138528440)

[9 References 68](#_Toc138528441)

Illustrations

1. The Quizzdos Project
   1. Introduction

In recent times, mobile phones have become an indispensable aspect of our everyday lives, providing us with convenience, flexibility, and instant access to information. As a result, mobile applications have appeared in various domains, including education. Against this backdrop, the main point of this graduation thesis revolves around the development and presentation of a mobile application designed with the intent to aid students, aiming to facilitate the management of quizzes and the understanding of classroom material.

In the modern educational system, professors are constantly exploring innovative ways to help students comprehend and retain the material covered in class. Quizzes, serving as a common tool for assessing students' knowledge of subject matter, have long been used as an effective aid. However, they have greater potential than knowledge evaluation. Quizzes can be used to allow students to review and reinforce concepts, as well as identify areas where further exploration and clarification may be required.

The aim of this thesis is to develop a mobile application that not only handles the administration of quizzes but also assumes the role of a comprehensive resource, serving to help the consolidation of information acquired within the classroom.

The proposed mobile application will provide students unrestricted access to quizzes, regardless of their geographic location or time constraints, through the medium of their smartphones. Whether students find themselves at home or through the daily commute on their way to an exam, this application promises to provide them with the access to educational content.

Additionally, the mobile application will be easy to use and understand. It will be designed in a way that anyone, regardless of their background or technical skills, can navigate through it effortlessly.

To cater to different learning preferences, the app will offer various types of quizzes, such as choosing one answer, selecting multiple answers, or determining if a statement is true or false. This wide range of quiz formats will accommodate the unique learning styles of students, helping them better understand and remember the topics they are studying.

In addition to providing expeditious feedback and facilitating progress tracking, the mobile application integrates a feature that distinguishes itself from conventional quiz platforms This feature combines an AI-powered tool called ChatGPT, which provides automated hints for quiz questions. ChatGPT uses speech recognition and a wide range of resources to offer personalized insights and guidance to students.

Imagine a situation where a student faces a challenging question. Instead of waiting for a teacher's assistance, the student can rely on the mobile app to get helpful hints that lead to the correct answer. This integration of AI technology aims to enhance the learning experience, enabling students to learn independently and develop a deep understanding of the subject matter.

This graduation thesis will explore the importance of quizzes in education and examine the numerous benefits of using mobile learning tools in teaching. Furthermore, it will provide a detailed explanation of how the mobile application was developed and designed. This includes carefully selecting the right tools and technologies, creating a user interface that is easy to use and accessible to all, and seamlessly integrating and implementing the quiz features.

The development of the QuizzDos application was motivated by the need for a specialized and user-friendly platform that complements existing learning management systems like Moodle. While Moodle is a widely adopted and comprehensive e-learning platform, QuizzDos offers unique features and advantages.

To sum up, this graduation thesis aims to make a meaningful and significant contribution to the field of education. Its goal is to provide students with a practical, versatile, and effective tool that strengthens their knowledge and enhances their academic accomplishments. By integrating quizzes, ensuring accessibility, employing an intuitive design, and introducing the innovative AI-powered hints feature, this mobile application represents a fresh approach to learning. It embraces the potential of mobile technology to help students, encouraging them to actively engage in their educational journey. Ultimately, the application seeks to develop a passion for learning and give students the essential skills and knowledge needed to become successful.

1. Specifications, requirements and design
   1. Similar applications
      1. MyFitnessPal
      2. Rust
   2. Project Requirements

A user will be able to:

* create a profile with data about their body.
* use the application on multiple mobile platforms, such as iOS and android.
* get a meal plan with a given calorie and macronutrient goal depending on their personal information and weight goal.
* adjust their calorie and macronutrient goal manually.
* search a curated list of foods with nutritional information to add to their meals.
* have a good user experience, with a user-friendly interface that is easy to navigate.
* have a responsive and smooth UI, with screen transitions and animations.
* search for food by their name or brand.
* see their recently tracked.
* favourite foods for easier tracking.
* scan foods by their bar code to bring up the nutritional information.
* log their daily water intake.
* see a daily summary with the number of eaten calories and macronutrients, in comparison to their goal and a micronutrient breakdown.
* see their calorie, macronutrient, water, weight progress for a given period of time.
* see their profile with their user data and personal details.
* adjust their water habits.
* change their dietary needs and preferences.
  1. Project Specifications

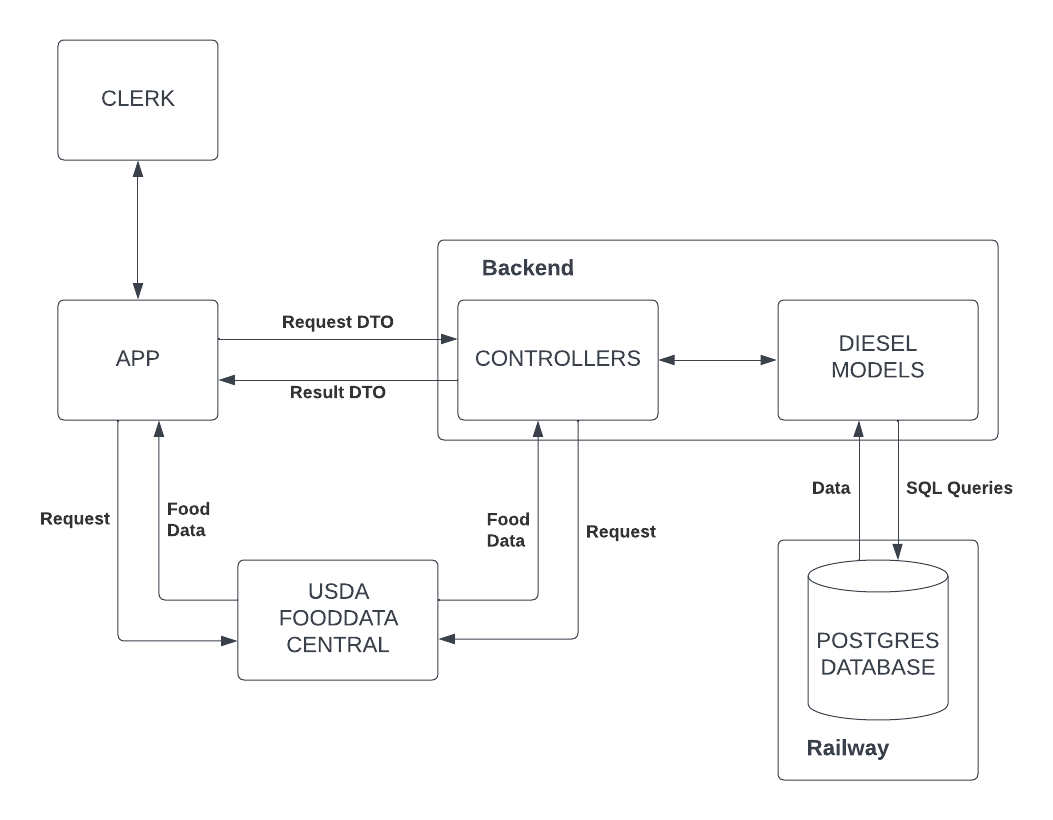
The frontend of the mobile application will be built using the Expo framework on top of React Native, with TypeScript, for multiple platform support, but prioritising iOS.

The backend will be written in Rust, utilizing Rocket, to build a type-safe, fast, and reliable API to handle requests from the application. It will also use a database-first approach with Diesel as an ORM (Object Relational Mapper) to communicate and query the database.

User authentication and management is done using Clerk, which also handles OAuth and user persistance using JSON Web Tokens.

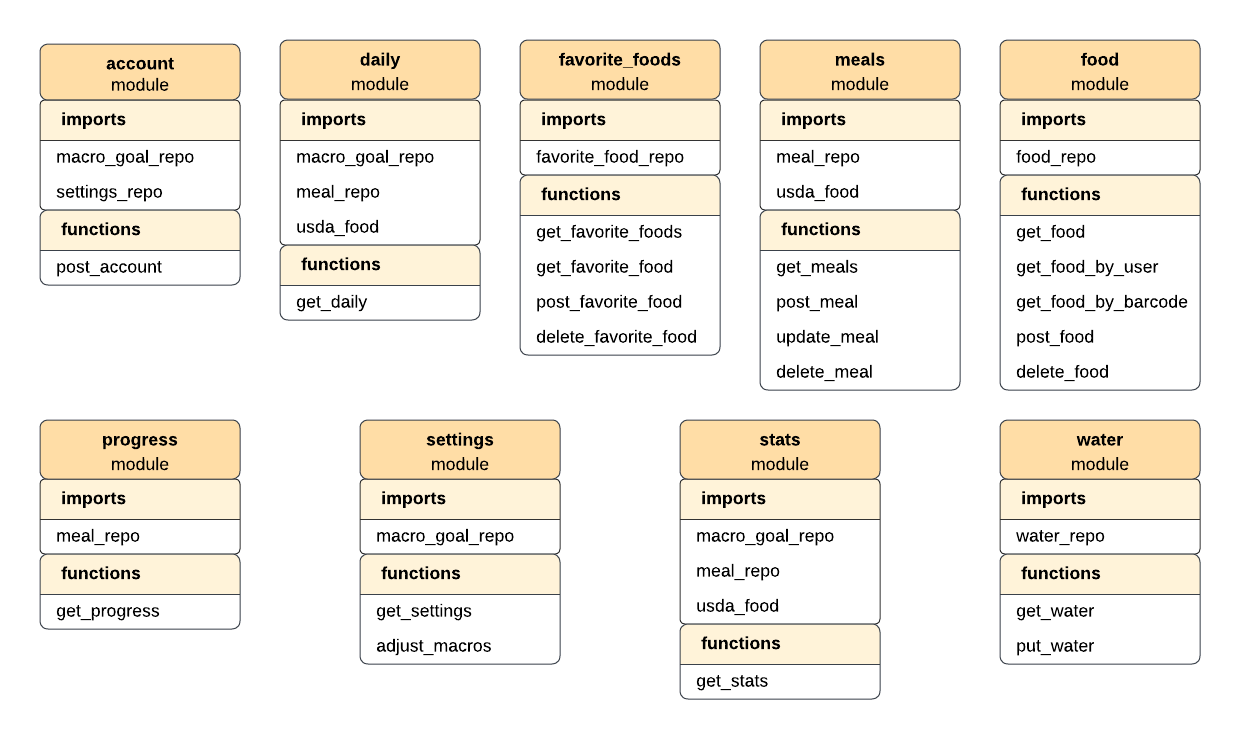
The database will be a relational database, using the PostgreSQL relational database management system. It will be hosted on and managed by Railway, for convenience during the development process.

* 1. Block diagram



The system is comprised of multiple components: the application, the Clerk platform, the USDA FoodData Central API, the backend, that has the controllers, the DTOs and Diesel Models, and the Postgres database managed by Railway.

1. App (Frontend) – The frontend is the interface of the user that communicates with clerk to get the user data, with the USDA API directly to get data on a specific food or search for food. It also communicates with the backend through a variety of routes to request data to be displayed to the user.
2. Backend – The backend is the component responsible for processing requests sent from the frontend, performing calculations and business logic, handling data retrieval to generate responses through DTOs (Data Transfer Object) to be sent back to the client-side of the application. It is made up of:
   1. DTOs – Data transfer objects that serve as containers for data that allow for information to be transported between different layers of the system. They can either be sent as a request from the client-side to the server-side and deserialized by the backend, or vice-versa, to be serialized into JSON as a response back to the client.
   2. Controllers – The controllers are a group of functions that receive and handle the HTTP requests dispatched to them by Rocket, that ties each controller to a given route. They handle GET, POST, PUT and DELETE requests, perform business logic, and send back appropriate responses to the client-side. Some controllers also communicate with the USDA API to handle more complex requests containing the food data that should be processed in the backend.
   3. Diesel Models – Diesel is an ORM that abstracts the database system and allows for querying and data manipulation of the database in an object-oriented paradigm. The code is translated by Diesel into the appropriate SQL query to perform CRUD operations over the database.
3. Postgres Database – The database hosted on Railway stores and manages part of the data used in the application.
4. USDA FoodData Central API – An external API provided by the United States Department of Agriculture that contains relevant nutritional data for a whole array of foods. Either handles requests directly from the client-side for simple tasks, such as displaying the raw data, or from the server-side for more processing-intensive ones, that need to alter the data before sending it to the client.
   1. Class diagrams
      1. Controllers

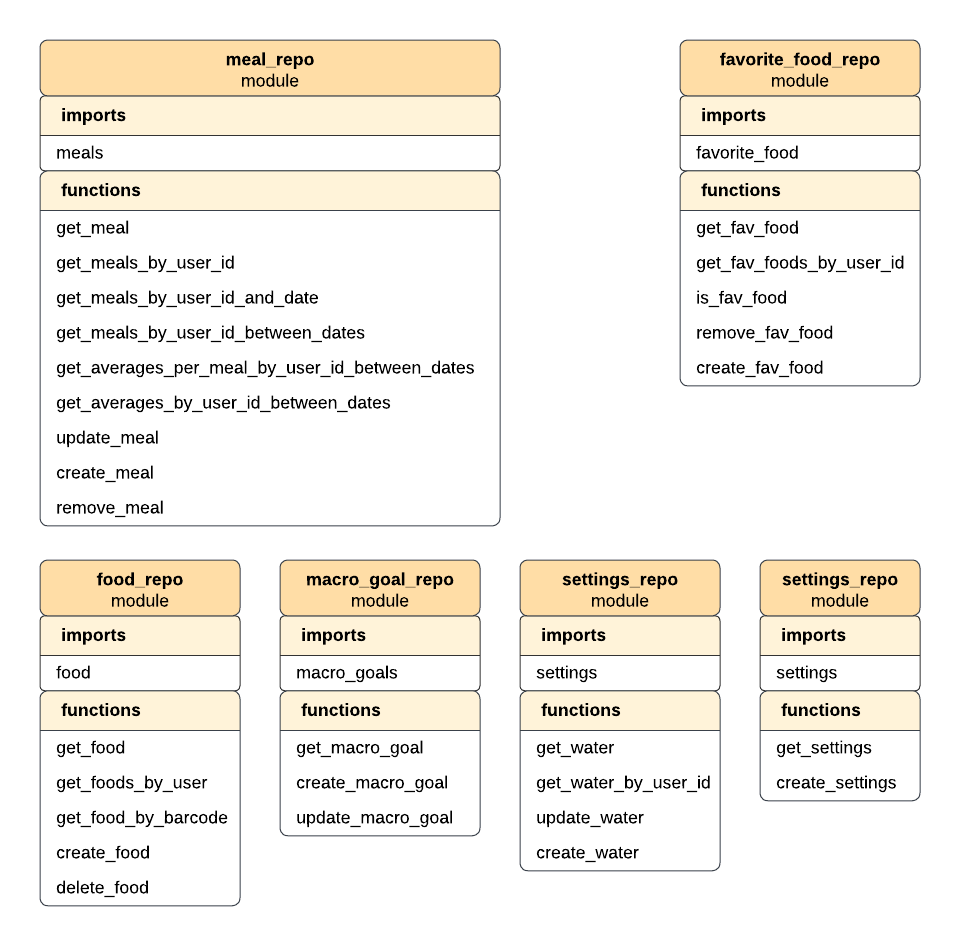


Rust, being a multi-paradigm language, doesn’t force the developer into grouping controller functionality into a class, so, naturally, each function is bundled up into a module, using repositories and/or the ‘usda\_food’ service as imports.

They handle the HTTP requests that are routed by Rocket to them and return the appropriate response.

The controllers are:

1. account – handles account creation actions, such as creating the settings of a user and calculating their macronutrient goals.
2. daily – used to congregate the information of a user’s daily meals and goal.
3. favorite\_foods – handles operations related to users’ favourite foods.
4. meals – handles the management of users’ meals, creating, reading, updating, and deleting.
5. food – handles the retrieval of the food by different criteria and the modification of food.
6. progress – handles the calculation and statistical operations necessary to provide information on a user’s progress over time.
7. settings – represents the operations tied to the reding and updating of a user’s settings.
8. stats – gives insight on a user’s daily statistics.
9. water – manages the users’ daily water intake.
   * 1. Repositories



In the backend, each repository is grouped intro their own module. They connect the controllers to the Diesel models and the database, providing them data access and storage by creating SQL queries using Diesel’s ORM.

1. Technical background
   1. Programming languages
      1. TypeScript (JavaScript)

TypeScript is a high-level programming language and developer tool what adds static typing with optional type annotations to JavaScript, allowing developers to define and enforce types for variables, function parameters, return values and more. JavaScript, TypeScript’s underlying programming language, offers a variety of programming paradigms, including but not limited to imperative programming, object-oriented programming, supporting principles such as encapsulation, inheritance, and polymorphism, functional programming, event-driven programming, with its integrations with the Document Object Model (DOM), having the ability to respond to user interactions, and asynchronous programming, through mechanisms such as callbacks, Promises and the async/await syntax.

Due in part to the language’s ease of use, but also JavaScript’s dominance on the web, since it was originally designed to be one of the core technologies powering web browsers, TypeScript has amassed a vast ecosystem of third-party packages for UI and UX design and development, such as frameworks (e.g., Angular, Svelte, Astro) and libraries (e.g., React, SolidJS). It also offers a multitude of utility libraries for tasks such as animations, handling HTTP requests, graph and chart visualizations, machine learning, and some that provide utility functions for working with arrays, objects, functions, and other data types in a functional programming style.

* + 1. Rust
  1. Frontend technologies
     1. React Native
     2. Nativewind (Tailwind CSS)
     3. Axios
     4. TanStack Query
     5. React Native Reanimated
     6. Expo Libraries
  2. Backend technologies
     1. Diesel
     2. Rocket & Serde
     3. Reqwest
     4. Tokio

1. Application utilization
   1. General
   2. Student Experience
2. Gantt Chart
   1. Definitions

A Gantt diagram [24], also known as a Gantt chart, is a type of bar chart that visually represents a project schedule.

In a Gantt diagram, the horizontal axis typically represents the project's time scale, while the vertical axis lists the tasks or activities involved in the project. Each task is represented by a horizontal bar, with the length of the bar corresponding to the duration of the task. The position of the bar on the time scale indicates the start and end dates of the task.

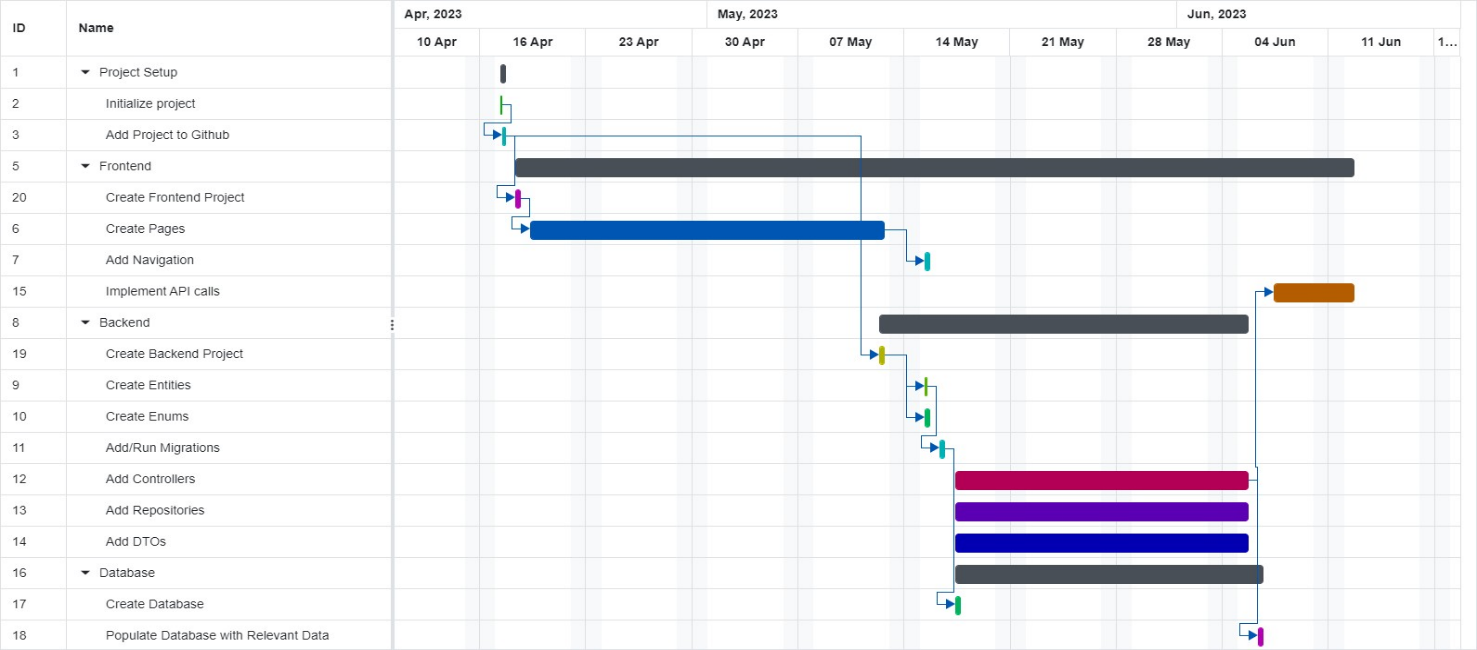
* 1. The Project’s Gantt Chart

Figure 5.2.1 Gantt Chart

1. Implementation

In this section, I will present the implementation details of the Quizzdos mobile application. The implementation phase involved developing the backend, frontend, and database components of the application, as well as integrating AI-driven hints using ChatGPT. The chosen technologies, such as .NET 6.0, React Native with Expo, and SQL Server Management Studio (SSMS), played a crucial role in achieving a robust and efficient implementation.

The separation of concerns between the two projects allows for a more modular and maintainable codebase. The "QuizzDos-Backend" project handles the business logic, while the "QuizzDos-EFCore" project is responsible for the data persistence and modelling.

This architecture promotes code organization, scalability, and code reusability. It adheres to the principle of separation of concerns, enabling easier development and maintenance.

* 1. Backend

The backend part of the solution is divided in two projects, the ‘quizzdos-backend’ and the ‘quizzdos-EFCore’.

The first contains repositories and controllers and it is used to handle the data access and manipulation operations, providing an abstraction layer between the application and the underlying data storage. The controllers define the API endpoints and handle the incoming requests, facilitating the communication between the clients and the application.

The former focuses on the entity framework core (EF Core) implementation. It houses the entity classes that represent the data entities in the application and the migrations. These entities define the structure and behavior of the data within the application's domain model.

* + 1. Models

The backend entities were designed to capture the necessary data and relationships required for the Quizzdos application. The relationships between these entities were established using Entity Framework Core's fluent API, ensuring accurate and efficient data retrieval and manipulation.

In the implementation of the Quizzdos mobile application, the data model was designed to handle entities such as the User entity. This model utilized a base class called BaseEntity which served as the foundation for other entities within the application.

* + - 1. Meal



The ‘meal-model’ module defines several data structures and enumerations.

The ‘Meal’ struct represents a user-added meal with its own id field and primary key for a given user with ‘user\_id’ and the added food with ‘food\_id’. It also contains information about the meal type that it was added for, that being the meals of the day, breakfast, lunch and dinner, or a snack, specified by the ‘MealType’ enum, the date on which the meal was added, and the proportion of the meal, comprised of the number of portions and the portion size.

The ‘source’ field denotes the database from which the food was added, be it a user made food, or a food from the USDA database.

Additionally, the struct also contains duplicate fields that could’ve been acquired from the ‘Food’ specified by ‘food\_id’ and calculated given the ‘portion’ and ‘portion\_size’ fields. These fields in question are “calories”, “carbs”, “protein”, and “fat” – the duplication serving the purpose of mitigating the cost of another request to the USDA API to get the ‘Food’ data for some requests.

* + - 1. Settings



This module also defines multiple enums that pertain to the ‘Settings’ struct.

The struct represents the initial personal details provided by the user during the account creation process that are used to create an informed dietary plan for the user, with a given macronutrient intake.

The details provided represent the weight goal of the user, this being the intention to lose, maintain or gain weight, the gender and age of the user. It also contains details with meanings dependant on the system of measurement used (imperial or metric), height representing centimetres in the metric system and inches in imperial system, respectively, weight, that represents either kilograms or pounds.

The primary key of the model is the ‘user\_id’ since each user will have one and only one ‘Settings’ data tied to them.

* + - 1. Macro Goal



The struct ‘MacroGoal’ represents the goal macronutrient intake of the user, inserted into the database alongside the ‘Settings’ row.

The fields of this struct represent the goal caloric intake for a day, ‘calories’, measured in kilocalories, and the target weight of each macronutrient (in grams), ‘carbs’, ‘protein’, and ‘fat’.

The remaining three fields represent the target percentage of macronutrients eaten throughout the day, ‘percent\_carbs’, ‘percent\_protein’, and ‘percent\_fat’. Each of these fields are, in part, set by default to values of 0.5 (50%), 0.3 (30%), respectively 0.2 (20%).

* + - 1. Favourite Food



The ‘FavoriteFood’ struct represents a single instance of one of the favourite foods set by the user. Its primary key is given by both fields combined.

It is an association between a person and a food, and it forms a many-to-many relationship connectiong them.

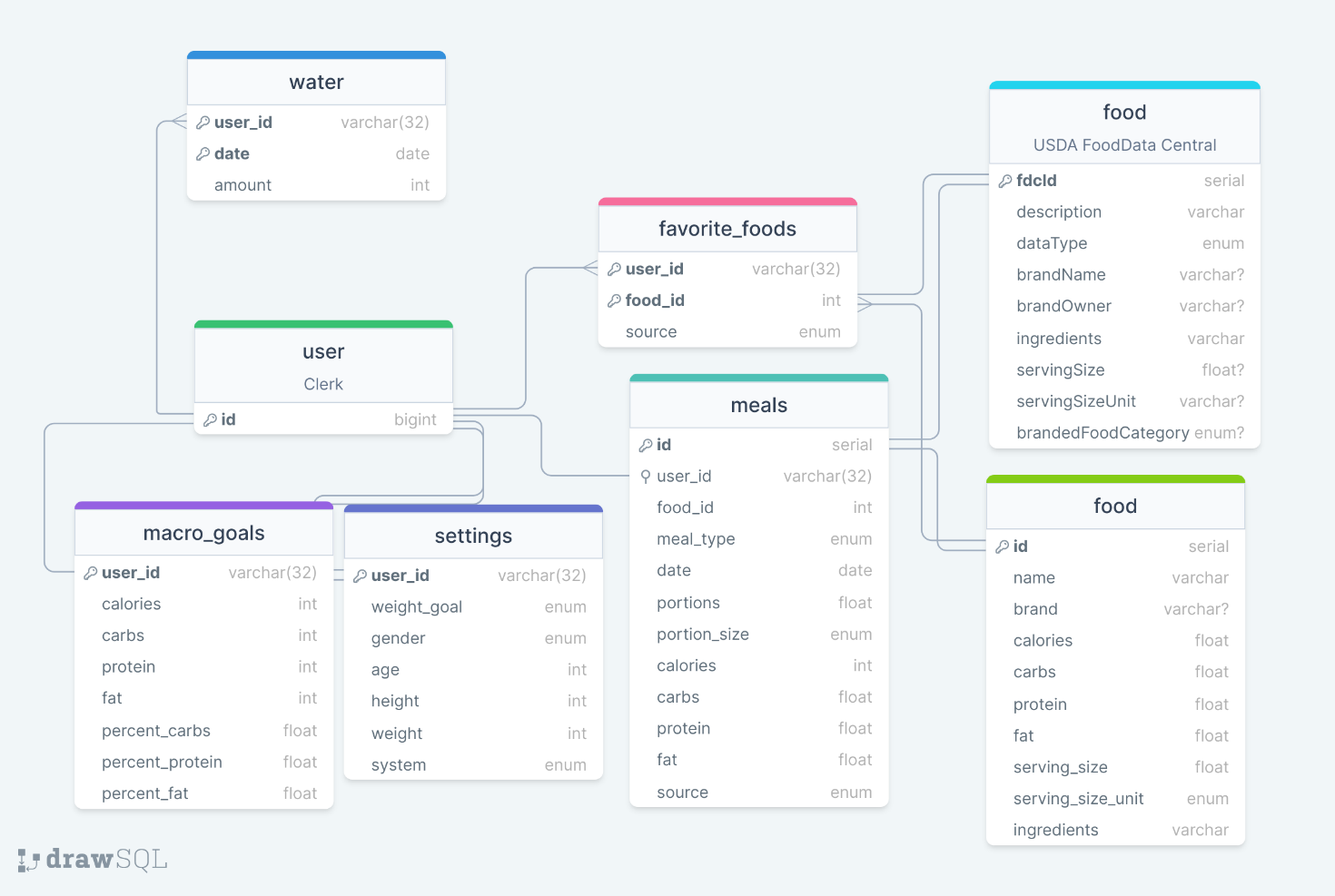
The source is the same as in the meals example.

* + - 1. Water



The ‘Water’ struct represents the water intake of a user on a given day, indexed by the compound primary key given by ‘user\_id’ and ‘date’.

* + 1. Database Diagram



As it can be seen, the meals and favorite\_foods databases are tied to either the food database food or a food from the USDA database.

* + 1. Migrations

Handling migrations was done using the diesel\_cli tool.

A migration is a set of SQL statements that describes the modifications to the database structure. Adding, removing, or altering database objects like tables, columns, or constraints enables the database schema to change over time, without losing precious data or causing disruption to the functionality of the application. Diesel's migrations are often used to maintain consistency between the database and the application's data model, by providing a version control system.

Migrations are generated using the command “diesel migration generate <migration\_name>”, creating an up.sql and a down.sql in the migrations folder.

To run migrations, you can use the “diesel migration run”, which will run the latest migration.

After running, it is advised to run the “diesel migration redo” which calls the down.sql then the up.sql again to make sure that the down.sql is correct.



In this example, we have an up.sql file, that represents the change that we want to make to the database schema, that being the creation of the macro\_goals table. The down.sql file represents the SQL query used to undo the changes made by up.sql in case of a rollback or a redo.



This is another example of a database migration, in which the intention was to add another column to the favorite\_foods database in order to allow for controlling of the source of the added favourite food, which before was only possible for usda food, thus having to set the source of old data to ‘usda’.

This is only possible by setting source to NULL, then setting the source to the desired value, and only after that setting it to NOT NULL, unless we want to set a default value for source, which we don’t want to in this case.

Next the altering of the primary key of the table takes place, since now there should be able to exist more instances of the same ‘user\_id’ and ‘food\_id’ if their source is different, but no two entries should have identical ‘user\_id’, ‘food\_id’ and ‘source’. This is done by removing the primary key constraint and adding a new one.

* + 1. quizzdos-backend

The ‘quizzdos-backend’ project is divided in: Program.cs, the controllers, the repositories, the data transfer objects (DTOs) and the responses.

* + - 1. Program.cs

Notably in Program.cs we add services configuration (controllers, endpoints api explorer, routing), we configure Swagger with version information, title, description and contact details. We also add a security definition for OAuth2 authentication using a bearer token. We add the database context configuration using EF Core. We add dependency injection by registering the repository classes with the dependency injection container using ‘AddScoped()’. We then register the HttpContextAccessor service and configure CORS to allow all headers, methods and origins.

* + - 1. Paginated Response

public class PaginatedResponse<T>

{

public int Page { get; set; }

public int PageSize { get; set; }

public int TotalCount { get; set; }

public int TotalPages => (int)Math.Ceiling((double)TotalCount /

PageSize);

public IEnumerable<T> Data { get; set; }

public PaginatedResponse(int page, int pageSize, int totalCount,

IEnumerable<T> data)

{

Page = page;

PageSize = pageSize;

TotalCount = totalCount;

Data = data;

}

}

The PaginatedResponse<T> class is designed to encapsulate the data related to paginated responses. It provides a convenient way to organize and return paginated data from APIs. The TotalCount and TotalPages properties assist in client-side pagination, allowing users to navigate through the paginated data. The Data property holds the actual collection of items for the current page.

* + - 1. Validation Response

public class ValidationResponse

{

public ExistingUserResponse? ExistingUserResponse { get; set; } = new();

public UsernameValidationResponse? UsernameValidationResponse { get; set; } = new();

public PhoneNumberValidationResponse? PhoneNumberValidationResponse { get; set; } = new();

public EmailValidationResponse? EmailValidationResponse { get; set; } = new();

public PasswordValidationResponse? PasswordValidationResponse { get; set; } = new();

}

This class represents a response object that consolidates these validation results. It’s composed of different type of validation responses.

public class ExistingUserResponse

{

public ValidationError UsernameAlreadyExists { get; set; } =

new("Username already exists");

public ValidationError EmailAlreadyExists { get; set; } =

new("Email already in use");

public ValidationError PhoneNumberAlreadyExists { get; set; } =

new("Phone number already in use");

}

This is an example of a validation response. The other validation responses are constructed similarly to this one.

public class ValidationError

{

public bool Error { get; set; } = true;

public string Message { get; }

public ValidationError(string message)

{

Message = message;

}

}

This is the ValidationError class. It has two properties:

* ‘Error’ – a boolean indicating if an error occurred. Its default value is true.

‘Message’ – a string that holds the error message provided during the creation of a ‘ValidationError’ instance.

* + - 1. DTOs

Data Transfer Objects are used for transferring data. They provide a way to encapsulate and transport data without exposing the underlying implementation details or models. For example, ‘CourseDTO’ is used as a lightweight representation of a course, it contains only the necessary data that’s being displayed or used in the frontend.

public class CourseDTO

{

public Guid CreatorId { get; set; }

public string Name { get; set; } = string.Empty;

public string ShortName { get; set; } = string.Empty;

public string Summary { get; set; } = string.Empty;

public string MaterialsUrl { get; set; } = string.Empty;

public string Icon { get; set; } = string.Empty;

}

* + - 1. Controller generalities

All controllers implemented have these two attributes:

[ApiController]

[Route("api/[controller]")]

The [ApiController] is an attribute provided by ASP.NET Core that is used to decorate a controller class. It is used to indicate that the class is an API controller, which means it is responsible for handling HTTP requests and returning HTTP responses as part of a web API.

The [Route("api/[controller]")] attribute is used to specify the route template for the API controller. In this case, it sets the route prefix for all the controller actions to "api/[controller]". The [controller] token is a placeholder that is replaced with the controller’s name based on naming conventions. For example, if the controller class is named AuthController, the route would be "api/auth" for the actions within this controller.

Controllers also have one or more private fields used for dependency injection.

private readonly IAuthRepository \_authRepository;

The implementation of the respective interfaces is provided at runtime. By declaring them as ‘private readonly’ we ensure that these cannot be reassigned after initialization, promoting immutability and encapsulation. The ‘\_’ prefix is used as a naming convention used to denote the private field.

Each controller has a constructor where its dependencies are provided.

public AuthController(IAuthRepository authRepository,

IValidationRepository validationRepository,

IUserRepository userRepository)

{

\_authRepository = authRepository;

\_validationRepository = validationRepository;

\_userRepository = userRepository;

}

Generally, all methods return Ok with status code 200 if they are successful and Bad Request 400 or Not Found 404 if the action fails or one parameter is not found.

* + - 1. Repository Generalities

All repositories have an interface they implement.

public interface IValidationRepository

{

Task<ExistingUserResponse?> CheckUniqueUser(UserDTO request);

...

PasswordValidationResponse? CheckPassword(string password);

}

The application's context, represented by the Context class, is the main tool used in handling data retrieval and manipulation from the underlying database. It serves as the connection between the repositories and the database, enabling seamless interaction between the application and the stored data.

private readonly Context \_context;

public ValidationRepository(Context context)

{

\_context = context;

}

* + - 1. AuthController

The AuthController has two action methods.

The first one is the ‘register’ action method.

[HttpPost("register")]

[ProducesResponseType(typeof(User), 200)]

[ProducesResponseType(typeof(ValidationResponse), 400)]

public async Task<ActionResult<User?>> Register(UserDTO request)

{

var userExists = await

\_validationRepository.CheckUniqueUser(request);

var isUsernameValid =

\_validationRepository.CheckUsername(request.Username);

var isPhoneNumberValid =

\_validationRepository.CheckPhoneNumber(request.PhoneNumber);

var isEmailValid =

\_validationRepository.CheckEmail(request.Email);

var isPasswordValid =

\_validationRepository.CheckPassword(request.Password);

if (userExists != null || isUsernameValid != null ||

isPhoneNumberValid != null || isEmailValid != null ||

isPasswordValid != null)

{

return BadRequest(new ValidationResponse

{

ExistingUserResponse = userExists,

UsernameValidationResponse = isUsernameValid,

PhoneNumberValidationResponse = isPhoneNumberValid,

EmailValidationResponse = isEmailValid,

PasswordValidationResponse = isPasswordValid

});

}

User newUser = await \_authRepository.Register(request);

return Ok(newUser);

}

The [HttpPost(„Register”)] attribute indicates that this method will handle HTTP POST requests, with the „register” route. This entails that the route will look like this: „https://[ip]:[port]/api/auth/register”.

The [ProducesResponseType(typeof(User), 200)] attribute specifies that a successful response will have a status code 200 (OK) and return an object of type User, while the [ProducesResponseType(typeof(ValidationResponse), 400)] indicates that if there are validation errors, the response will have a status code 400 (Bad Request) and return an object of type ValidationResponse.

The method Register(UserDTO request) accepts a parameter of type UserDTO, which represents the user registration data received from the client.

Inside the method, various validations are performed usnig the „\_validationRepository”. If any validation errors are found, a ‚BadRequest’ response is returned along with the object that contains all the details of the validation errors. If validadations pass successfully, the \_authRepository calls the Register(request) method to register the user and returns an Ok response with the newly registered user object.

[HttpPost("login")]

...

public async Task<ActionResult<string>> Login([FromBody] UserDTO

request)

Similarly to the previous attribute, the `[HttpPost(“login)]` attribute indicates that this method will handle HTTP POST requests, this time with the “login” route: "https://[ip]:[port]/api/auth/login".

The other attributes function similarly to the previous method.

The Login method accepts a parameter of type UserDTO, representing the user's login credentials received from the client.

Inside the method, the \_userRepository is used to retrieve the user from the database based on the provided login credentials. If no user is found, a "User not found" response is returned with a status code 404 (Not Found).

Next, the \_authRepository is used to verify the password hash against the user's stored password hash and salt. If the password is not verified, a "Wrong password" response is returned with a status code 400 (Bad Request).

If the password is verified successfully, an authentication token is created using the \_authRepository's CreateToken method.

Finally, an OK response with the authentication token, prefixed with "bearer", is returned to the client.

* + - 1. AuthRepository

The IAuthRepository interface defines the contract for handling user authentication and registration operations. It includes the following methods:

public interface IAuthRepository

{

Task<User> Register(UserDTO request);

void CreatePasswordHash(string password, out byte[] passwordHash, out byte[] passwordSalt);

string CreateToken(User user);

bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt);

}

The AuthRepository class implements the IAuthRepository interface and provides the actual implementation for the authentication and registration operations. It has the following dependencies:

1. IConfiguration: Used to access the application configuration, including the secret key for token signing.
2. Context: Represents the database context used to interact with the database.
3. IUserRepository and IPersonRepository: Dependencies for accessing user and person data.

In the Register method, a new User instance is created and populated with the user data from the UserDTO object. The password is hashed and salted using the CreatePasswordHash method, and the hash and salt values are stored in the User instance. The User object is then added to the database using the IUserRepository and IPersonRepository interfaces, and the changes are saved using the Context.

public async Task<User> Register(UserDTO request)

{

User newUser = new();

CreatePasswordHash(request.Password, out byte[] passwordHash,

out byte[] passwordSalt);

newUser.Username = request.Username;

newUser.Email = request.Email;

newUser.PhoneNumber = request.PhoneNumber;

newUser.Created = DateTime.Now;

newUser.PasswordHash = passwordHash;

newUser.PasswordSalt = passwordSalt;

await \_userRepository.AddUserAsync(newUser);

await \_personRepository.AddPersonAsync(newUser);

await \_context.SaveChangesAsync();

return newUser;

}

The CreateToken method generates a JWT for the user based on the provided claims, expiration date, and signing credentials. The token is returned as a string.

public string CreateToken(User user)

{

var role = (\_context.People.FirstOrDefault(p => p.UserId == user.Id)?.Role) ?? throw new Exception("User has no role");

List<Claim> claims = new()

{

new Claim(ClaimTypes.Name, user.Username),

new Claim(ClaimTypes.Email, user.Email),

new Claim(ClaimTypes.MobilePhone, user.PhoneNumber),

new Claim(ClaimTypes.Sid, user.Id.ToString()),

};

var key = new

SymmetricSecurityKey(System.Text.Encoding.UTF8.GetBytes(\_configuration

.GetSection("AppSettings:Token").Value));

var cred = new SigningCredentials(key,

SecurityAlgorithms.HmacSha512Signature);

var token = new JwtSecurityToken(claims: claims,

expires:

DateTime.Now.AddDays(1),

signingCredentials: cred);

return new JwtSecurityTokenHandler().WriteToken(token);

}

The VerifyPasswordHash method compares the computed hash of the provided password with the stored password hash using the salt. If the hashes match, the password is considered valid.

public bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt)

{

using var hmac = new HMACSHA512(passwordSalt);

var computedHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));

return computedHash.SequenceEqual(passwordHash);

}

Overall, the AuthRepository class handles the logic for user authentication and registration, including password hashing, token generation, and password verification. It interacts with the database through the Context and repository interfaces to persist and retrieve user data.

* + - 1. Users Controller

The `UsersController` class is used to handle user operations. It has a method, decorated with “[HttpGet(“current-user”)]”. This specifies that this method will handle HTTP GET requests, with the route “current-user”.

The GetUser() method is the implementation of the action. It retrieves the current user from the IUserRepository using the GetUser() method. If the user is found, it returns an Ok response with the user object. Otherwise, it returns a BadRequest response with an error message.

* + - 1. Users Repository

The UserRepository class implements the IUserRepository interface and is responsible for accessing and manipulating user data in the database.

The constructor of the UserRepository class takes two dependencies: IHttpContextAccessor and Context. The IHttpContextAccessor is used to access the current HTTP context, which is required for retrieving the current user information. The Context represents the database context, just like in the other repository presented before, used to interact with the underlying data storage.

The UserRepository class provides the following methods:

This method gets the current user using the \_contextAccessor and checks the claims.

public Task<User> AddUserAsync(User user);

This method adds a new user to the database asynchronously. It uses the Users DbSet of the Context to add the user and returns the added user.

public Task<User?> GetUserByAnyField(string? username, string? email, string? phoneNumber);

The GetUserByAnyField method retrieves a user from the database based on any of the provided fields (username, email, or phoneNumber). It uses the Users DbSet of the Context and the FirstOrDefaultAsync method to find the first user that matches the specified criteria. It returns the found user or null if no user is found.

public CurrentUserDTO? GetUser();

GetUser(): This method retrieves the current user information from the HTTP context. It uses the IHttpContextAccessor to access the current HTTP context and retrieves relevant user information from the claims. It returns a CurrentUserDTO object that contains the username, email, phoneNumber, and Id of the current user. If the HTTP context is not available, it returns null.

* + - 1. Notifications Controller

The NotificationsController is responsible for handling HTTP requests related to notifications.

The controller provides the following action methods:

public async Task<ActionResult> GetNotifications(Guid personId)

This action method handles HTTP GET requests to retrieve notifications for a specific person. It accepts a personId as a route parameter. It calls the GetNotificationDTOs(personId) method of the \_notificationRepository to retrieve the notification DTOs. If the notifications are found, it returns an HTTP 200 response with the list of NotificationDTO objects. Otherwise, it returns an HTTP 404 response with an appropriate error message.

public async Task<ActionResult> UpdateNotification(List<string> notificationIds)

This method handles HTTP PUT requests to update the status of notifications. It accepts a list of notificationIds in the request body. It converts the notificationIds to a list of Guid and calls the MarkAsRead(guidNotificationIds) method of the \_notificationRepository to mark the notifications as read. If the update is successful, it returns an HTTP 200 response with a true value. Otherwise, it returns an HTTP 400 response with an appropriate error message.

public async Task<ActionResult> GetUnreadNotifications(Guid personId)

This method is used to handle HTTP GET requests to check if there are any unread notifications for a specific person. It accepts a personId as a route parameter. It calls the HasUnreadNotifications(personId) method of the \_notificationRepository to check if there are any unread notifications. If there are unread notifications, it returns an HTTP 200 response with a true value. Otherwise, it returns an HTTP 404 response with an appropriate error message.

* + - 1. Notifications Repository

The NotificationRepository class is used for managing notifications within the system. It offers various functionalities to handle notifications effectively.

One of the main features is the ability to add notifications. It provides the methods to add both single notifications or multiple notifications in bulk, in order to improve efficiency.

Additionally, the repository handles the retrieval of targeted notifications to an id of a person.

Furthermore, it supports the functionality of marking notifications as “read”. By having a distinction between “read” and “unread” notifications, the frontend can utilize this to manage the notifications panel to easily identify unread notifications. Naturally, the repository provides the means to check if a user has unread notifications.

* + - 1. CoursesController

The CoursesController class is responsible for handling requests related to courses in the API. It provides the following action methods:

[HttpGet("{personId:Guid}/{courseId:Guid}")]

[ProducesResponseType(typeof(Course), 200)]

[ProducesResponseType(typeof(string), 404)]

public async Task<ActionResult> GetCourse(Guid courseId, Guid

personId)

{

var course = await

\_courseRepository.GetCourseByIdAsync(courseId, personId);

if (course == null)

return NotFound($"Course: {courseId} was not found!");

return Ok(course);

}

This method handles the retrieval of a specific course by its ID and person ID by calling “GetCourseByIdAsync” from “ICourseRepository”. It’s route looks like: “/api/courses/{the id of the person}/{the id of the course}”. The “personId” is passed in order to establish the degree of completion of the sections of that course. A section is complete if all of its quizzes have at least one grade of 10. This will be detailed later when discussing the Course Repository in **6.1.2.14**.

The following methods look very similar, the difference between them is the method called from the course repository.

[HttpPost]

...

public async Task<ActionResult> AddCourse(CourseDTO course)

This method handles the addition of a new course. It calls the “AddCourseAsync” method from the ICourseRepository and returns Ok if it’s successful or BadRequest if the addition fails.

[HttpPut]

public async Task<ActionResult> UpdateCourse(Guid courseId,

CourseDTO course)

This method handles the updating of a course by its id. It returns an Ok response with the updated course if it’s successful or a NotFound if the course couldn’t be found.

[HttpDelete]

public async Task<ActionResult> DeleteCourse(Guid courseId)

This action method handles the deletion of a course by its ID.

[HttpPost("{courseCode}/people/{personId}")]

public async Task<ActionResult> AddPersonToCourse(string courseCode,

Guid personId)

This method handles adding a person to a course. It accepts the course code and person ID and calls the AddPersonToCourse method of ICourseRepository to add the person to the course. It returns an Ok response with a success message if successful, or a BadRequest response if the addition fails.

[HttpGet("{personId}")]

public async Task<ActionResult> GetJoinedCoursesPaged(Guid personId, int pageParam = 1, int pageSize = 1)

This method handles the retrieval of joined courses by a specific person ID in a paginated manner. Its default values for paging are “pageParam = 1”, which means it will handle the first page, or the page number 1, and the pageSize = 1, which means from the first page, only one element will be fetched. This is used for the main page of the students.

On success, this method returns an Ok response with the PaginatedResponse <DiplayCourseDTO>

Similarly, we have the GetCreatedCoursesPaged, which handles the retrieval of the courses created by a specific creator ID in a paginated manner. This will be used for the main page of the professors.

[HttpGet("creators/{creatorId:Guid}")]

public async Task<ActionResult> GetCreatedCoursesPaged(Guid creatorId, int pageParam = 1, int pageSize = 1)

* + - 1. Course Repository

The CourseRepository class implements the ICourseRepository interface and is responsible for handling operations related to courses in the database.

public async Task<Course?> AddCourseAsync(CourseDTO addingCourse)

This method adds a new course to the database. It creates a new “Course” object from the CourseDTO and adds it to the “Courses” table. The “Code” property is generated by taking a normalized random Guid (without the dashes) and takes the first 8 hexadecimal characters. It associates the course with the creator and sends a notification. Frontend

The “DeleteCourseAsync” and “UpdateCourseAsync” methods are very simple, the first finds the course by id and removes it, while the second finds it and changes its properties according to its parameter.

The “GetJoinedCoursesPaginated” method is used to retrieve a paginated list of courses joined by a specific person.

public async Task<PaginatedResponse<DiplayCourseDTO>?>

GetJoinedCoursesPaginated(Guid personId, int page, int pageSize)

{

var person = await \_context.People

.Include(p => p.CourseAppartenences)

.ThenInclude(ca => ca.Course)

.ThenInclude(c => c.Sections)

.ThenInclude(s => s.Quizzes)

.ThenInclude(q => q.Grades)

.FirstOrDefaultAsync(p => p.Id == personId);

if (person == null)

return null;

var joinedCourses = person.CourseAppartenences

.Select(ca => new DiplayCourseDTO

{

Id = ca.CourseId.GetValueOrDefault(),

ShortName = ca.Course.ShortName,

SectionsNumber = ca.Course.Sections.Count,

Progress = GetCourseProgress(ca.Course, personId),

Icon = ca.Course.Icon,

Code = ca.Course.Code

})

.Skip((page - 1) \* pageSize)

.Take(pageSize)

.ToList();

var totalJoinedCourses = person.CourseAppartenences.Count;

return new PaginatedResponse<DiplayCourseDTO>(page, pageSize,

totalJoinedCourses, joinedCourses);

}

It retrieves the person from the database, including their associated CourseAppartenences (Course Membership) and related data.

It makes use of another private static method, “GetCourseProgress”.

private static double GetCourseProgress(Course course, Guid personId)

{

double totalSections = course.Sections.Count;

double completedSections = course.Sections

.Where(section => section.Quizzes.All(quiz =>

GetQuizStatus(quiz, personId) == EQuizStatus.Done))

.Count();

return totalSections == 0 ? 0 : completedSections /

totalSections;

}

This method calculates by getting the number of sections and dividing it by the completed sections. A complete section is defined as a section where all of its quizzes are evaluated with the status of EQuizStatus.Done. This happens the GetQuizStatus method returns the mentioned Enum.

private static EQuizStatus GetQuizStatus(Quiz q, Guid personId)

{

Grade? maxGrade = q.Grades.Where(g => g.PersonId ==

personId).OrderByDescending(g => g.GradeValue).FirstOrDefault();

return maxGrade switch

{

null => EQuizStatus.Unopened,

Grade g when g.GradeValue == 10 => EQuizStatus.Done,

\_ => EQuizStatus.InProgress,

};

}

This method searches for the highest grade of a person from a specific quiz. If they do not have a grade, it will return EQuizStatus.Unopened. If they have a 10 for that quiz, it will return EQuizStatus.Done. Otherwise, it will return EQuizStatus.InProgress.

The method designed for the professors’ created courses works in a similar way, with the difference that they do not need the progress property.

public async Task<PaginatedResponse<DiplayCourseDTO>>

GetCreatedCoursesPaged(Guid creatorId, int page, int pageSize)

The “AddPersonToCourse” method adds a person to a course. It retrieves the course based on the provided code and the person based on the personId. If the course or the person is not found, it returns false. If the person has already joined the course, it returns false. Otherwise, it creates a new CourseMembership object to represent the association between the course and the person, adds it to the CourseAppartenences table, and sends a notification. It returns true if the person is successfully added to the course.

public async Task<bool> AddPersonToCourse(string code, Guid personId)

There is one more method:

public async Task<AccessedCourseDTO?> GetCourseByIdAsync(Guid courseId, Guid personId)

This method retrieves a course by its ID, including its sections and quizzes. It calculates the progress for each section and quiz based on the person's completed quizzes and returns an AccessedCourseDTO object representing the accessed course.

* + - 1. Person Controller

This controller is used to handle HTTP requests related to people.

We have an HTTP Get method “GetPersonByUserId”, that calls the GetPersonByUserIdAsync method from the IPersonRepository.

public async Task<ActionResult<Person>> GetPersonByUserId(Guid userId)

It returns an Ok result if it’s successful or a Not Found, if the userId cannot be found.

Another method we use is the “UpdatePerson” method.

public async Task<ActionResult<UpdatedDetailsPersonDTO>>

UpdatePerson(Guid personId, string firstName, string lastName, EGender gender)

This method updates a person’s personal details by calling the UpdatePersonalDetailsByIdAsync method from the IPersonRepository.

The last method of this controller is:

public async Task<ActionResult<PersonSettingsDTO>> GetPersonSettings(Guid personId)

This method returns an Ok result with PersonSettingsDTO if it’s successful or a Not found otherwise.

* + - 1. Person Repository

The “Person Repository” implements the IPersonRepository interface. It has a few methods, of which some are used in the “Person Controller”. Its constructor takes a Context object and an INotificationRepository object as dependencies.

The first method that we will mention is:

public async Task<Person?> GetPersonByUserIdAsync(Guid userId)

This is a simple method that retrieves a person based on their “User Id” instead of the typical id of the entity.

We also have the GetPersonByIdAsync method that retrieves the person by their entity id and AddPersonAsync that adds a new person by making use of the “Person(User user)” constructor.

The UpdatePersonalDetailsByIdAsync method updates the personal details (firstName, lastName, and gender) of a person based on the provided personId. It retrieves the person from the database, updates the properties with the provided values, saves the changes to the database, and returns an UpdatedDetailsPersonDTO object containing the updated details. Additionally, it adds a notification to the person indicating that their personal details have been successfully updated.

Further on, we have the “GetPersonSettings” method. This method retrieves the person settings based on the provided personId. It includes the related User entity in the query to retrieve additional information. It returns a PersonSettingsDTO object containing the person's gender, email, first name, last name, phone number, role, and username.

* + - 1. Quizzes Controller

The QuizzesController is responsible for handling HTTP requests related to quizzes in your application. It provides endpoints to add, grade, delete, update, retrieve quiz information and hints.

[HttpPost]

public async Task<ActionResult> AddQuiz(QuizDTO quiz)

This method simply adds a new quiz calling the “AddQuizAsync” method of the IQuizRepository. If the quiz is successfully added, it returns an HTTP 200 OK response with the added quiz or a 400 Bad Request otherwise.

public async Task<ActionResult> GradeQuiz([FromBody] GradeQuizInput

gradeDetils)

This method handles the HTTP POST request to grade a quiz. It receives a GradeQuizInput object in the request body, which includes the quiz ID, person ID, and quiz grade. The method calls the AddQuizGrade method of the quizRepository to add the quiz grade to the database. If the grading is successful, it returns an HTTP 200 OK response with a boolean value indicating the success of the grading. If the grading fails, it returns a 400 Bad Request response.

Similarly, we have other methods like DeleteQuiz(Guid quizId), which deletes a quiz and UpdateQuiz(Guid quizId, UpdateQuizDTO quiz), which updates the name of a quiz.

The questions of a quiz are updated by another method, specifically:

public async Task<ActionResult> UpdateQuizQuestions(Guid quizId, List<QuizQuestionDTO> newQuestions)

It takes the quizId as a parameter in the URL and a list of QuizQuestionDTO objects in the request body, containing the updated quiz questions. The method calls the UpdateQuizQuestions method of the quizRepository to update the questions of the quiz in the database. If the quiz questions are updated successfully, it returns an HTTP 200 OK response with the updated quiz questions. If the quiz is not found, it returns a 404 Not Found response.

Then, we have the StartQuiz method:

public async Task<ActionResult> StartQuiz(Guid quizzId, Guid personId)

This method handles the HTTP Get request to start a quiz for a specific person. It needs the personId to check if the person already has a grade for this quiz.

The last method is the GetTip HTTP Get, which handles the request to retrieve a tip for a specific question. It takes the questionId as a parameter in the URL. The method calls a method in the quizRepository to get a hint or tip for the given question. If a hint is available, it returns an HTTP 200 OK response with the hint. If no hint is available or there's an error, it returns a 400 Bad Request response.

* + - 1. Quiz Repository

The QuizRepository class is responsible for handling database operations related to quizzes. It implements the IQuizRepository interface, which defines the contract for interacting with quizzes.

The constructor of the QuizRepository class initializes the repository by accepting the required dependencies and configurations.

public QuizRepository(Context context,

IConfiguration config,

INotificationRepository notificationRepository)

{

\_apiKey = config.GetValue<string>("ApiKey");

\_context = context;

\_notificationRepository = notificationRepository;

}

The “IConfiguration config” parameter represents the configuration settings for the application. Specifically, it is used to retrieve the API key required for accessing the OpenAI API. The other parameters are used similarly to how they are injected in the previous repositories.

The methods implemented in this repository are:

public async Task<QuizDTO?> AddQuizAsync(QuizDTO addingQuiz)

This method adds a new quiz to the database and creates a notification using the “\_notificationRepository”.

public async Task<bool?> AddQuizGrade(Guid quizId, Guid personId, double grade)

This method checks if a person with the “personId” exists in the database and, if they do, it adds the grade to the Grades table and then it sends a notification to the user.

public async Task<Quiz?> DeleteQuizAsync(Guid quizId)

This method simply deletes a quiz based on its id.

public async Task<StartQuizDTO?> GetQuizForStudent(Guid quizId, Guid

personId)

{

var quiz = await \_context.Quizzes

.Include(quiz => quiz.Section)

.Include(quiz => quiz.Questions)

.ThenInclude(quest => quest.Options)

.FirstOrDefaultAsync(q => q.Id == quizId);

if (quiz == null)

return null;

return new StartQuizDTO

{

Grade = quiz.Grades.OrderByDescending(g =>

g.GradeValue).FirstOrDefault(g => g.PersonId ==

personId)?.GradeValue ?? 0,

QuestionsNumber = (uint)quiz.Questions.Count,

SectionName = quiz.Section.Name,

Status = quiz.Status,

Title = quiz.Name,

Questions = quiz.Questions.Select(question => new

QuestionDTO

{

Id = question.Id,

Options = question.Options.Select(option => new

OptionDTO

{

Id = option.Id,

Text = option.Text,

ScorePercentage = option.ScorePercentage

}).ToList(),

Prompt = question.Prompt,

QuestionScore = question.QuestionScore,

TipAllowed = question.TipAllowed

}).ToList()

};

}

This method retrieves the details of a quiz for a specific student. It takes the quizId and personId as parameters. The method fetches the quiz from the database, including the associated section, questions, and options. It constructs a StartQuizDTO object containing relevant quiz details like grade, question number, section name, and question information. If the quiz is found, the method returns the StartQuizDTO object. Otherwise, it returns null.

public async Task<List<QuestionDTO>> GetQuizQuestions(Guid quizzId)

This method works similarly to the previous one, but it only retrieves the questions of the quiz, without the additional properties of Grade, QuestionsNumber, SectionName, Status and Title.

Furthermore, we have the GetTip method which will be discussed next:

public async Task<string?> GetTip(Guid questionId)

{

var question = await \_context.Questions.Where(q => q.Id == questionId)

.Include(q => q.Options)

.Select(q =>

new QuestionWithCorrectAndWrongAnswersDTO

{

Prompt = q.Prompt,

CorrectOptions = q.Options.Where(o =>

o.ScorePercentage > 0).Select(o => new QuestionOptionDTO { Text =

o.Text, ScorePercentage = o.ScorePercentage }).ToList(),

WrongOptions = q.Options.Where(o =>

o.ScorePercentage == 0).Select(o => new QuestionOptionDTO { Text =

o.Text, ScorePercentage = o.ScorePercentage }).ToList()

}).FirstOrDefaultAsync();

...

}

This method retrieves a hint or tip for a specific question. The method queries the database to fetch the question and its associated options. After constructing a QuestionWithCorrectAndWrongAnswersDTO, we have a null check and then we construct the prompt that will be passed to the OpenAI API.

string wrongOpts = question.WrongOptions.Aggregate("", (acc, x) => acc + x.Text + "; ");

string correctOpts = question.CorrectOptions.Aggregate("", (acc, x) => acc + x.Text + "; ");

var prompt =

"Pretend you are an automated quiz hint giver. " +

"I will write a question and one or more answers. " +

"Give a hint of the answers of the question that will be written without giving away the answer. " +

"Do not rephrase the question with the answer in it. Do not rephrase the given answer while answering the question. " +

"Make it not obvious. Do not write numbers using letters. Do not complete the answers, answer in a separate phrase. " +

"If it is a mathematical question, give the formula or how to derive it from known information." +

$"This is the question: {question.Prompt} " +

"These are the the wrong options:" + wrongOpts +

".This is/are the correct answers:" + correctOpts

;

After constructing the prompt, we create a new HTTP Client in order to communicate with the OpenAI API.

var url = $"https://api.openai.com/v1/completions";

var apiKey = \_apiKey;

using var client = new HttpClient();

var jsonBody = new JObject

{

["model"] = "text-davinci-003",

["prompt"] = prompt,

["temperature"] = 0,

["max\_tokens"] = 1000

};

The “text-davinci-003” model was chosen for this purpose, because it was based on GPT-3.5 architecture, which provides advanced natural language processing capabilities and, importantly, typically fast responses.

The “prompt” is the property that contains the text that will be used to generate the quiz tip.

The “temperature” determines the randomness of a generated tip. A temperature of 0 indicates that the output will be deterministic and focused.

The “max\_tokens” = 1000 property sets the maximum length of the generated tip in terms of the number of tokens. It limits to 1000 tokens to prevent excessively long responses.

client.DefaultRequestHeaders.Authorization = new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", apiKey);

Here, the Authorization header of the HTTP client is set to include the API key. This ensures that the request is authenticated and authorized to access the OpenAI API.

var content = new StringContent(jsonBody.ToString(),

Encoding.UTF8, "application/json");

A new StringContent object is created, representing the request body. The jsonBody object is converted to a JSON string using ToString(), encoded with UTF8 encoding, and the content type is set as application/json.

var response = await client.PostAsync(url, content);

The POST request is sent to the OpenAI API endpoint with the request body (content). The response from the API is awaited and stored in the response variable.

if (!response.IsSuccessStatusCode)

{

Console.WriteLine("Error with Status Code: " +

response.StatusCode);

return null;

}

The code checks if the response's status code indicates a successful operation. If the status code is not a success code (e.g., 200 OK), an error message is displayed, and null is returned.

var json = JsonConvert

.DeserializeObject<dynamic>

(response.Content.ReadAsStringAsync().Result);

if (json == null) return null;

string completion = json.choices[0].text;

return completion.Trim();

If the response was successful, the generated quiz tip is read as a string and deserialized using “JsonConvert.DeserializeObject”. The resulting object is assigned to the “json” variable. If the deserialization is successful, the generated quiz tip is extracted from the “choices” property using “json.choices[0].text”. The “text” property represents the generated text. The extracted tip is then returned after removing any leading or trailing whitespace using “Trim()”.

Another important method is UpdateQuizQuestions. This is used to perform an upsert operation on the quiz with the corresponding quizId.

First, we gather the quiz with its details:

var quiz = await \_context.Quizzes

.Include(q => q.Section)

.ThenInclude(s => s.Course)

.Include(q => q.Questions)

.ThenInclude(q => q.Options)

.FirstOrDefaultAsync(q => q.Id == quizId);

if (quiz == null)

return null;

Then, after identifying the questions that exist in the database but are not present in the list, we remove them from the database context:

var questionsToDelete = quiz.Questions.Where(q =>

!updatedQuestions.Any(uq => uq.Id == q.Id)).ToList();

\_context.Questions.RemoveRange(questionsToDelete);

Then, we iterate over each updated question and update the corresponding entity:

foreach (var updatedQuestion in updatedQuestions)

{

var existingQuestion = \_context.Questions.FirstOrDefault(q

=> q.Id == updatedQuestion.Id);

…

}

if (existingQuestion != null)

{

existingQuestion.Prompt = updatedQuestion.Prompt;

existingQuestion.QuestionScore =

updatedQuestion.QuestionScore;

existingQuestion.TipAllowed =

updatedQuestion.TipAllowed;

\_context.Options.RemoveRange(existingQuestion.Options);

await \_context.SaveChangesAsync();

existingQuestion.Options =

updatedQuestion.Options.Select(option =>

new Option

{

QuestionId = existingQuestion.Id,

Text = option.Text,

ScorePercentage = option.ScorePercentage

}).ToList();

\_context.Options.AddRange(existingQuestion.Options);

\_context.Entry(existingQuestion).State =

EntityState.Modified;

\_context.Questions.Update(existingQuestion);

}

If we find a question that matches, we update the question with the new prompt, question score and the boolean value that represents the permission to access a tip.

Then, we remove the existing options associated with the question and save the changes to the database. We then create new “Option” entities according to the updated question’s options. We add them to the context and mark the existing question entity as modified. We then update the question in the database. By doing these steps, we can ensure the existing question’s properties are updated with the new data provided in the “updatedQuestions” object.

else

{

var newQuestion = new Question

{

Id = Guid.NewGuid(),

QuizId = quizId,

Prompt = updatedQuestion.Prompt,

QuestionScore = updatedQuestion.QuestionScore,

TipAllowed = updatedQuestion.TipAllowed,

Options = updatedQuestion.Options.Select(option =>

new Option

{

Id = Guid.NewGuid(),

QuestionId = updatedQuestion.Id,

Text = option.Text,

ScorePercentage = option.ScorePercentage

}).ToList()

};

quiz.Questions.Add(newQuestion);

\_context.Options.AddRange(newQuestion.Options);

\_context.Entry(newQuestion).State = EntityState.Added;

}

}

If we can’t find a match for the question, we simply add a new question with the options to the database.

We then gather all the ids of the creator and the course’s participants and send them a notification:

var peopleIds = await GetCourseParticipantsAndProfessor(quiz);

await \_notificationRepository.BulkAddNotification(

title: "Quiz Updated",

text: $"The quiz '{quiz.Name}' has been updated",

personIds: peopleIds

);

* + - 1. Statistics Controller

The Statistics Controller retrieves the statistics for students and professors in two HTTP Get methods:

public async Task<ActionResult<CourseStudentStatisticsDTO?>> GetStudentCourseStatistics(Guid personId)

public async Task<ActionResult<CourseProfessorStatisticsDTO?>> GetProfessorCourseStatistics(Guid personId, int pageParam = 1, int pageSize = 1)

The professors’ statistics are paged due to the nature of the display of their data.

* + - 1. Statistics Repository

The StatisticsRepository class is responsible for retrieving and generating statistics related to courses and students/professors.

public async Task<List<CourseStudentStatisticsDTO>?>

GetStudentsCourseStatisticsAsync(Guid personId)

{

var student = await FetchStudentWithCourses(personId);

if (student == null)

return null;

var statistics = student.CourseAppartenences

.Where(ca => ca.Course.Sections.Any())

.Select(ca => GenerateCourseStatistics(ca, personId))

.OrderByDescending(cs => cs.Sections.Select(s =>

s.Average).DefaultIfEmpty(0).Average())

.ToList();

return statistics;

}

This method performs the following steps:

Fetches the student entity with their course membership and related data from the database using the FetchStudentWithCourses method.

Filters out the courses that have at least one section and selects the statistics for each course using the GenerateCourseStatistics method.

Orders the statistics by the average of section grades in descending order.

Returns the list of course statistics.

The GenerateCourseStatistics method constructs a CourseStudentStatisticsDTO object with the course’s id and short name. It then filters out the sections that have at least one quiz and selects the statistics for each using the GenerateSectionStatistics method. It then orders the statistics by the section’s average grade in descending order.

private static CourseStudentStatisticsDTO

GenerateCourseStatistics(CourseMembership courseAppartenance, Guid

personId)

{

return new CourseStudentStatisticsDTO

{

Id = courseAppartenance.CourseId.GetValueOrDefault(),

ShortName = courseAppartenance.Course.ShortName,

Sections = courseAppartenance.Course.Sections

.Where(s => s.Quizzes.Any())

.Select(s => GenerateSectionStatistics(s, personId))

.OrderByDescending(ss => ss.Average)

.ToList()

};

}

The GenerateSectionStatistics method takes in a section and a personId to follow the same pattern and create a SectionStudentStatisticsDTO:

Finally, the GenerateQuizStatistics creates a QuizStudentStatisticsDTO with the following structure:

private static QuizStudentStatisticsDTO GenerateQuizStatistics(Quiz

quiz, Guid personId)

{

return new QuizStudentStatisticsDTO

{

Id = quiz.Id,

Name = quiz.Name,

Grade = quiz.Grades

.Where(g => g.PersonId == personId)

.OrderByDescending(g => g.GradeValue)

.FirstOrDefault()?.GradeValue ?? 0

};

}

The GetProfessorsStatisticsAsync method retreives the paginated professor statsitics based on their personId.

public async Task<PaginatedResponse<CourseProfessorStatisticsDTO>?> GetProfessorsStatisticsAsync(Guid personId, int pageNumber, int pageSize)

{

if (pageNumber < 1)

pageNumber = 1;

if (pageSize < 1)

pageSize = 1;

var professorCourses = await FetchProfessorCourses(personId);

var statistics = professorCourses

.Select(c => GenerateStatistics(c))

.OrderByDescending(c => c.StudentsNumber)

.ThenByDescending(c => c.AverageGrade)

.Skip(pageNumber - 1).Take(pageSize).ToList();

return new PaginatedResponse<CourseProfessorStatisticsDTO>

(

pageNumber,

pageSize,

professorCourses.Count,

statistics

);

}

After handling the cases where the „pageNumber” and „pageSize” are less than 1, we fetch the courses that were created by the passed „personId”, generate the statistics in a similar manner and return a new „PaginatedResponse” object.

* 1. Frontend

The frontend part of the application, built with React Native and Expo, follows a folder structure that organizes the codebase into different directories.

* + 1. App.tsx

To organize the App file effectively, it utilizes an AppContainer component. The App function, exported as default, serves as the entry point for the application. It is structured as follows:

export default function App() {

return (

<AppContainer>

<Stack.Screen name="Welcome" component={Welcome} />

            <Stack.Screen name="Login" component={Login} />

            <Stack.Screen name="Settings" component={Settings} />

            <Stack.Screen name="AdminPeople" component={AdminPeople}

/>

            <Stack.Screen name="AdminHome" component={AdminHome} />

            <Stack.Screen

                name="ProfessorStatistics"

                component={ProfessorStatistics}/>

            <Stack.Screen name="Quiz" component={Quiz} />

            <Stack.Screen name="QuizStart" component={QuizStart} />

            <Stack.Screen name="Course" component={Course} />

            <Stack.Screen name="CreateCourse" component={CreateCourse}

/>

            <Stack.Screen name="CommonHome" component={CommonHome} />

            <Stack.Screen name="Home" component={Home} />

            <Stack.Screen name="SignUp" component={SignUp} />

            <Stack.Screen

                name="StudentStatistics"

                component={StudentStatistics}/>

            <Stack.Screen

name="Notifications"

component={Notifications}/>

            <Stack.Screen name="CreateQuiz" component={CreateQuiz} />

            <Stack.Screen name="QuizResult" component={QuizResult} />

            <Stack.Screen

name="FirstTimeSignIn"

component={FirstTimeSignIn}/>

</AppContainter>

);

}

Inside the AppContainer component, Stack.Screen components are defined to represent different screens or views of the application. Each Stack.Screen is associated with a unique name and component, allowing for navigation between screens.

The AppContainer looks like this:

const AppContainter = ({ children }: AppContainerProps) => {

return (

<UserProvider>

<QueryClientProvider client={querryClient}>

<NavigationContainer theme={DarkTheme}>

<StatusBar style="light" />

<Stack.Navigator

screenOptions={{

headerShown: false,

cardStyle: styles.cardStyle,

}}

>

{children}

</Stack.Navigator>

</NavigationContainer>

</QueryClientProvider>

</UserProvider>

);

};

Within the AppContainer, the UserProvider component wraps the entire application, enabling access to user-related information and functionalities throughout the app.

The QueryClientProvider component, which receives a client instance (querryClient), enables data fetching and management using a query client.

The NavigationContainer component, utilizing the DarkTheme, provides the navigation context and handles the navigation logic for the application.

The StatusBar component defines the appearance of the status bar, specifying a light style for the status bar content.

The Stack.Navigator component is responsible for managing the navigation stack and screen transitions. It defines screenOptions, such as hiding the header and specifying a cardStyle for screen transitions.

The “{children}” expression renders the child components defined within the App function, corresponding to the various Stack.Screen components.

* + 1. Api

This folder contains the necessary functions and interfaces for making API calls to the backend server. It includes modules that utilize the Axios library to handle HTTP requests and responses. These functions handle communication with the backend API, such as fetching data, sending requests, and handling responses.

* + - 1. API Constants

This is a Typescript file where we define our routes for the necessary endpoints.

Firstly, we create a function to make the route between two points and add a ‘/’ between them:

function makeRoute<B extends string, R extends string>(

    base: B,

    route: R

): `${B}/${R}` {

    return `${base}/${route}` as const

}

Then we use it to create our route to the controllers:

const baseUrl = "http://192.168.1.168:5000/api"

const controllers = {

    auth: makeRoute(baseUrl, "auth"),

...

}

This will result in “<http://192.168.1.168:5000/api/auth>”.

Finally, we create our endpoints by reusing our makeRoute function:

export const ApiEndpoints = {

    Auth: {

        login: makeRoute(controllers.auth, "login"),

...

}

We export this member which will result in us having a collection of API endpoints with the structure of: [ApiEndpoints.Controller.method]. For example:

await axios.[http method](ApiEndpoints.Auth.register,[other parameters])

* + - 1. API Calls

Let's take a closer look at how API calls are defined in the code snippets provided.

In the StudentStatistics.ts file, there is a simple HTTP Get method that retrieves the statistics of a student. The function responsible for this is defined as:

export const fetchStudentStatistics = async (personId: string) => {

    const response = await axios.get(

        `${ApiEndpoints.Statistics.studentStatistics}/${personId}`

    )

    return response.data

}

This function makes a straightforward request by appending the student's ID as a path parameter to the specified endpoint. It fetches the statistics related to that particular student.

Moving on to an HTTP POST request, we have the following code:

export const loginUser = async (userPayload: any) => {

    let { data: response } = await axios.post(ApiEndpoints.Auth.login, {

        username: userPayload.credentials,

...

        password: userPayload.password,

    })

...

In this case, the code defines a login request. It includes a body parameter that contains all the necessary information for a user login. The payload includes the user's credentials, such as the username, email or phone number and password, which are sent to the specified endpoint for authentication.

* + 1. Assets

This folder stores various assets used in the application, such as Lottie animation JSON files.

* + 1. Components

The components folder holds reusable UI components that are used across multiple screens or sections of the application. These components help to maintain a consistent look and feel throughout the app and improve code reusability.

* + - 1. BottomAppbarLayout

This functional component is responsible for rendering a bottom app bar layout with icons for navigation. Studens, professors and admins use it, but it’s customized for each role.

const BottomAppbarLayout = ({ navigation, children }: BottomAppbarProps) => {

const { role, personId } = useUser()

    const isAdmin = role === ROLES.admin

...

}

The first thing we do, is get the user’s role and personId from our useUser(). The useUser() is a context we defined in our application that holds user data. It wraps the whole application in such a way that every component has access to information like: name, personId, role etc.

const [hasUnreadNotifications, setHasUnreadNotifications] = useState(false)

    useQuery(

        ["notifications", personId],

        () => fetchHasUnreadNotifications(personId),

        {

            onSuccess: (data) => {

                setHasUnreadNotifications(data)

            },

            onError: (error) => console.log(error),

            refetchInterval: 25000,

        }

    )

Then we use React’s useState hook in order to manage the state for the unread notifications.

The “hasUnreadNotifications” is updated inside an useQuery hook, which is defined in the react-query package. The useQuery has a few parameters. The first is a query key, unique for each query which is used internally for refetching, caching, and sharing the queries throughout the application.

We then have the query function, which we imported from our API methods.

If the query is successful, we update the hasUnreadNotifications variable with the data returned from the API call. Otherwise, we log the error.

The “refetchInterval” is used to create polling, in order to continuously check every 25 seconds, in this case, if there are new notifications.

Then we have the return statement that defines the JSX (JavaScript XML) markup that will be rendered as the output of the component:

return (<Flex justify="between" style={{ height: "100%" }}>

            {children}

            <View style={{ height: 0 }} />

            <View style={{ marginTop: 64 }}>

                <Appbar style={[styles.AppBar]} elevated={true}>

                    <Appbar.Action

                        icon="home"

                        iconColor="white"

                        onPress={() => navigation.navigate("Home")}

                    />

                    {!isAdmin && (

                        <Appbar.Action

                            icon="chart-line"

                            iconColor="white"

                            onPress={() => {

                                if (role in navigationRoutes) {

                                    navigation.navigate(navigationRoutes[role])

                                }

                            }}

                        />

                    )}

                    {isAdmin && (

                        <Appbar.Action

                            icon="account-group"

                            iconColor="white"

                            onPress={() => navigation.navigate("AdminPeople")}

                        />

                    )}

                    {!isAdmin && (

                        <Appbar.Action

                            icon={

                                !hasUnreadNotifications ? "bell" : "bell-

badge"

                            }

                            iconColor="white"

                            onPress={() =>

navigation.navigate("Notifications")}

                        />

                    )}

                    <Appbar.Action

                        icon="cog"

                        iconColor="white"

                        onPress={() => navigation.navigate("Settings")}

                    />

                </Appbar>

            </View>

        </Flex>)

We can notice a few things:

1. The app bar's layout is dynamically adjusted based on the user's role. For example, if the user is an admin, certain icons or actions may be omitted from the app bar. Conversely, admins have access to specific pages like the "People" page.

It utilizes the navigationRoutes object, which maps role names to corresponding screen names. For example, navigationRoutes[role] determines the appropriate screen to navigate to, based on the user's role.

1. The notification icon's appearance is responsive and changes based on whether the user has any unread notifications. If there are no unread notifications, a regular bell icon is displayed. However, if there are unread notifications, a badge may be added to the bell icon to indicate the presence of unread notifications.
   * + 1. PaginatedHorizontalList

This component is designed to display a horizontally scrollable list with pagination functionality.

const PaginatedHorizontalList = forwardRef<any, Props>(

    ({ children, navItems }, ref) => {

        const [activeIndex, setActiveIndex] = useState(0)

        const [isScrolling, setIsScrolling] = useState(false)

        const scrollRef = useRef<ScrollView>(null)

...

}

Firstly, we define some state variables. “activeIndex” tracks the currently active item and “isScrolling” indicates if the list is currently being scrolled.

A “scrollRef” is created using useRef to access the ScrollView component.

Then we have a few functions defined:

const onPress = (targetIndex: number) => () => {

            scrollRef.current?.scrollTo({

                x: Dimensions.get("window").width \* targetIndex,

                animated: true,

            })

            setActiveIndex(targetIndex)

        }

This function is defined to handle the “onPress” event of each navigation item. It scrolls the list to the corresponding item.

        const onMomentumScrollEnd = (e: any) => {

            const { nativeEvent } = e

            const index = Math.round(

                nativeEvent.contentOffset.x / Dimensions.get("window").width

            )

            setActiveIndex(index)

            setIsScrolling(false)

        }

The onMomentumScrollEnd function is invoked when the momentum scrolling ends. It determines the current active index based on the content offset of the scroll view.

useImperativeHandle(ref, () => ({

            scrollTo: (x: number) => {

                setIsScrolling(true)

                scrollRef.current?.scrollTo({

                    x: x,

                    animated: true,

                })

                setActiveIndex(Math.round(x / Dimensions.get("window").width))

            },

        }))

The “useImperativeHandle” hook is used to expose the “scrollTo” method externally via the ref prop. This allows external components to programmatically scroll the list.

Finally, we have the return statement with the structure:

   return (

            <View style={styles.container}>

                <View style={styles.navContainer}>

                    {navItems.map((icon, index) => (

                        <MenuItem

                            children={icon}

                            active={activeIndex === index}

                            onPress={onPress(index)}

                            key={index}

                        />

                    ))}

                </View>

                <ScrollView

                    horizontal

                    pagingEnabled

                    nestedScrollEnabled

                    onMomentumScrollEnd={onMomentumScrollEnd}

                    showsHorizontalScrollIndicator={false}

                    showsVerticalScrollIndicator={false}

                    ref={scrollRef}

                >

                    {children.map((child, index) => (

                        <View style={styles.childrenContainer} key={index}>

                            {child}

                        </View>

                    ))}

                </ScrollView>

            </View>

        )

The navigation container maps the navItems array to MenuItem components. It determines the active item based on activeIndex and sets the corresponding item's active state.

The ScrollView component is set to scroll horizontally (horizontal prop) and enables paging (pagingEnabled prop) for smooth scrolling. It uses the onMomentumScrollEnd function to handle the end of scrolling.

* + 1. Constants

In the constants folder, there are files that define values used throughout the application. For example, we have the same constants presented in the backend part, along with some others, like the list of icons used in the icon picker.

* + 1. Screens

This folder holds the different screens or pages of the application. Each screen represents a separate user interface component that users interact with. These screens are typically composed of various UI components, and they define the structure and behavior of the user interface for specific features or functionalities.

* + - 1. Common Home

The “CommonHome” component renders the students and professors’ home screen, which includes a list of courses and various actions related to those courses.

From the component we can observe:

The component defines several state variables using the useState hook to manage the component's state.

“courses”: stores the list of courses.

“courseFullWidth” determines whether the course cards should occupy the full width of the screen.

“joinCourseModalVisible”, “joinCourseModalError”, and “joinCourseModalCourseCode” manage the visibility, error state, and entered course code for the join course modal.

“createCourseModalVisible” manages the visibility of the create course modal.

“personId”, “role”, and “firstName” are obtained from the “useUser” context hook, which provides user-related data.

const [courses, setCourses] = useState<IDisplayCourses[]>([])

    const { personId, role, firstName } = useUser()

    const [courseFullWidth, setCourseFullWidth] = useState(false)

    const [joinCourseModalVisible, setJoinCourseModalVisible] =

useState(false)

    const [joinCourseModalError, setJoinCourseModalError] = useState(false)

    const [joinCourseModalCourseCode, setJoinCourseModalCourseCode] =

        useState("")

This component uses the useInfiniteQuery hook from react-query to fetch and query data related to the courses. It makes use of the fetchFunction based on the user's role (student or professor) to fetch the appropriate data. The fetched data is stored in the “pages” variable.

const fetchFunction = isStudent ? fetchJoinedCourses : fetchCreatedCourses

    const {

        refetch,

        data: pages,

        fetchNextPage,

        hasNextPage,

    } = useInfiniteQuery(

        ["courses", personId, pageSize],

        ({ pageParam = 1 }) =>

            fetchFunction({

                personId,

                creatorId: personId,

                pageParam,

                pageSize,

            }),

        {

            getNextPageParam: (lastPage) => {

                if (lastPage.page != lastPage.totalPages) {

                    return lastPage.page + 1

                }

                return false

            },

            enabled: true,

        }

    )

The useEffect hook is used to update the courses state variable whenever the pages data changes. It flattens the pages data using flatMap and sets the courses state with the new data.

    useEffect(() => {

        if (pages) {

            const newCourses = pages.pages.flatMap((page) => page.data)

            setCourses(newCourses)

        }

    }, [pages])

The component uses the useMutation hook from react-query to handle the mutation for joining a course. It defines the mutation function and provides callbacks for successful and error cases.

    const joinCourseMutation = useMutation({

        mutationFn: (data: any) => joinCourse(data),

        onSuccess: () => {

            setJoinCourseModalVisible(false)

            refetch()

        },

        onError: (data) => {

            setJoinCourseModalError(true)

        },

    })

    const handleJoinCourse = async () => {

        joinCourseMutation.mutate({

            courseCode: joinCourseModalCourseCode,

            personId,

        })

    }

The rendering includes a “BottomAppbarLayout” with a “Scroll View” for scrollable content. The “fetchNextPage” and “hasNextPage” are defined into the useInfiniteQuery and, by using them, we can achieve a paginated scroll view.

<BottomAppbarLayout navigation={navigation}>

                <ScrollView

                    onScroll={({ nativeEvent }) => {

                        if (

                            nativeEvent.layoutMeasurement.height +

                                nativeEvent.contentOffset.y >=

                                nativeEvent.contentSize.height &&

                            hasNextPage

                        ) {

                            fetchNextPage()

                        }

                    }}

                    scrollEventThrottle={0}

                >

...

/>

Inside the Scroll View we can categorize the components as such:

The user’s welcome message;

Join/Create course buttons.

Course cards;

Modals

The user’s welcome message is composed by some simple “Text” components inside a vertical stack:

    <VStack style={{ marginHorizontal: 25 }} spacing={10}>

                        <Text

                            style={{

                                color: COLORS.white,

                                fontSize: 30,

                                fontWeight: "bold",

                            }}

                        >

                            Hello, {firstName}

                        </Text>

                        <Text style={{ color: COLORS.white, fontSize: 16 }}>

                            Welcome to your{" "}

                            <Text style={{ color: COLORS.blue }}>courses</Text>.

                        </Text>

                    </VStack>

The “Join” and “Create” course buttons allow users to join or create a course and their visibility depends on the user’s role.

<Button

                                icon="plus-circle-outline"

                                contentStyle={{

                                    backgroundColor: COLORS.blue,

                                    flexDirection: "row-reverse",

                                }}

                                mode="contained"

                                onPress={() => {

                                    isStudent

                                        ? setJoinCourseModalVisible(true)

                                        : setCreateCourseModalVisible(true)

                                }}

                            >

                                {isStudent ? "Join" : "Create"}

                            </Button>

The courses state variable is mapped to render individual course cards using a TouchableRipple component. The course details such as name, icon, section count, and progress (if the user is a student) are displayed. These cards are wrapped in a “HStack” (horizontal stack) with justified spacing and wrapped to fit the screen.

<HStack justify="between" wrap="wrap">

                            {courses.map((course) => (

                                <TouchableRipple

                                    key={course.id}

                                    style={{

...

                                    }}

                                    onPress={() =>

                                        navigation.navigate("Course", {

                                            courseId: course.id,

                                        })

                                    }

                                    onLongPress={() => {

                                        Clipboard.setStringAsync(

                                            course.code

                                        ).then((content) => {

                                            ToastAndroid.show(

                                                "Course code copied to

clipboard",

                                                ToastAndroid.SHORT

                                            )

                                        })

                                    }}

                                >

...

1. Testing and Validation

In the development of the application, manual testing and validation were conducted to ensure the functionality and reliability of the software.

* 1. Functional Testing

Functional testing involved verifying that each feature of the application worked as intended. Test cases were created based on the specified requirements and user stories. The application was tested by manually executing various actions and inputs to ensure the expected behavior was observed. For example, different user roles were tested to validate their respective access rights.

* 1. User Interface Testing

User interface (UI) testing focused on evaluating the usability and responsiveness of the application's interface. The application was tested across multiple devices to ensure a consistent and user-friendly experience. Elements such as buttons, forms, navigation, and data display were reviewed to ensure proper functionality and adherence to design guidelines.

* 1. Error Handling and Validation

The application's error handling and validation mechanisms were manually tested. Various error scenarios were simulated, such as entering invalid input, submitting incomplete forms, or triggering error conditions. The application's response to these scenarios, including error messages and appropriate validation feedback, was evaluated to ensure accurate and user-friendly error handling.

* 1. Integration Testing

Integration testing involved testing the interactions and data flow between different components of the application. For example, the integration between the frontend and backend systems was manually validated to ensure seamless communication and data consistency. API endpoints were tested to verify proper data retrieval, updates, and error handling.

* 1. Data Validation and Database Testing

Data validation and database testing were performed to ensure the integrity and accuracy of the data stored in the application's database. Test data was carefully created and inserted into the database to cover different scenarios. Queries and database operations were manually executed to verify data integrity, proper filtering, and sorting functionality.

1. Conclusions

In conclusion, this thesis project focused on the development of a web application called "QuizzDOS" for managing quizzes, courses, and user statistics. Throughout the project, several key aspects were addressed, including application architecture, user interface design, backend implementation, frontend implementation, database management, and data analysis.

The backend was built using the Microsoft .NET framework, specifically utilizing technologies such as ASP.NET Core, Entity Framework Core, and Microsoft SQL Server. These technologies provided a robust and scalable foundation for the application's backend functionality, ensuring efficient data management and seamless integration with the frontend.

The frontend of the application was designed with a user-centric approach, emphasizing intuitive user interfaces, responsive layouts, and effective data visualization. The use of modern web development frameworks and libraries, such as React Native and React Native Paper, contributed to an enhanced user experience and improved usability across different devices and screen sizes.

A major focus of the application was on managing quizzes, courses, and user statistics. Various features were implemented to allow users to create quizzes, enrol in courses, track their quiz performance, and view comprehensive statistics.

Throughout the development process, manual testing and validation were performed to ensure the application's functionality, usability, and reliability. This involved functional testing, user interface testing, error handling and validation testing, integration testing, and data validation and database testing.

Overall, the QuizzDOS web application successfully achieved its objectives by providing an intuitive and efficient platform for managing quizzes, courses, and user statistics. The project demonstrated the successful implementation of industry-standard technologies and practices in web application development.

As a result of this thesis, valuable insights were gained into the development lifecycle of a web application, including requirements analysis, system design, implementation, testing, and deployment. The project showcased the significance of user-centred design, database management, data analysis, and the importance of continuous testing and validation in ensuring a high-quality software product.

Moving forward, potential areas for improvement and future work include expanding the application's features, incorporating more advanced data analysis techniques, implementing additional automated tests, and further optimizing the application's performance and scalability.

1. References
2. “What Is .NET? An Open-Source Developer Platform.” Microsoft, [https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet. Accessed 14 June 2023](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet.%20Accessed%2014%20June%202023).
3. zuckerthoben. Get Started with Swashbuckle and ASP.NET Core. 10 Mar. 2023, https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle.
4. Json.NET - Newtonsoft. [https://www.newtonsoft.com/json. Accessed 14 June 2023](https://www.newtonsoft.com/json.%20Accessed%2014%20June%202023).
5. auth0.com. *JWT.IO*. http://jwt.io/. Accessed 14 June 2023.
6. EF Core Power Tools - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=ErikEJ.EFCorePowerTools. Accessed 14 June 2023.
7. React Native · Learn Once, Write Anywhere. https://reactnative.dev/. Accessed 14 June 2023.
8. “Expo.” Expo, https://expo.dev/. Accessed 14 June 2023.
9. Getting Started | React Native Material. https://react-native-material.com//docs/getting-started. Accessed 14 June 2023.
10. React Native Paper. <https://reactnativepaper.com/>. Accessed 14 June 2023.
11. React Navigation | React Navigation. https://reactnavigation.org/. Accessed 14 June 2023.
12. TanStack Query | React Query, Solid Query, Svelte Query, Vue Query. https://tanstack.com/query/v3/. Accessed 14 June 2023.
13. Getting Started | Axios Docs. https://axios-http.com/docs/intro. Accessed 14 June 2023.
14. “Lottie-React-Native.” Npm, 14 May 2023, <https://www.npmjs.com/package/lottie-react-native>.
15. “React-Native-Chart-Kit.” Npm, 8 Feb. 2022, <https://www.npmjs.com/package/react-native-chart-kit>.
16. React-Native-Vector-Icons Directory. https://oblador.github.io/react-native-vector-icons/. Accessed 14 June 2023.
17. JavaScript With Syntax For Types. https://www.typescriptlang.org/. Accessed 14 June 2023.
18. erinstellato-ms. Download SQL Server Management Studio (SSMS) - SQL Server Management Studio (SSMS). 24 May 2023, https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms.
19. ajcvickers. Overview of Entity Framework Core - EF Core. 25 May 2021, https://learn.microsoft.com/en-us/ef/core/.
20. “GitHub: Let’s Build from Here.” GitHub, https://github.com/. Accessed 14 June 2023.
21. “GitHub Desktop.” GitHub Desktop, https://desktop.github.com/. Accessed 14 June 2023.
22. Atlassian. “Jira | Issue & Project Tracking Software.” Atlassian, https://www.atlassian.com/software/jira. Accessed 14 June 2023.
23. Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed 14 June 2023.
24. OpenAI API. https://platform.openai.com. Accessed 14 June 2023.
25. “Gantt.Com.” *Gantt.Com*, https://www.gantt.com. Accessed 14 June 2023.