Diploma thesis in the higher department of informatics

Quap - a simple quiz app

Design, development and implementation of a quiz and learning app

5AHINF-2023/24-DA12

Frontend, Backend, Documentation, Database

Florian Piberger   5AHINF   **Primary supervisor**:
                                               Dipl.-Ing (FH) Dietmar Winkler

                                               Secondary supervisor:
                                               Prof. Dipl.-Ing (FH) Markus Falkensteiner

Completed in the school year 2023/2024

1. List of abbreviations

API: Application Programming Interface

CI/CD: Continuous Integration/Continuous Deployment

CLI: Code Line Interface

CRUD: Create, Read, Update, Delete

CSS: Cascading Style Sheets

DBMS: Database Management System

DOM: Document Object Model

DTO: Data Transfer Object

FOSS: Free and Open Source

FTP: File Transfer Protocol

GUI: Graphical User Interface

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

IDE: Integrated Development Environment

IP: Internet Protocol

JSON: JavaScript Object Notation

JSX: JavaScript Extensible Markup Language (XML)

MVC: Model View Controller

NPM: Node Package Manager

OOP: Object Oriented Programming

ORM: Object Relational Mapper

OS: Operating System

PDF: Portable Document Format

REST: Representational State Transfer

SQL: Structured Query Language

UI: User Interface

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

VCS: Version Control System

VsCode: Visual Studio Code

WYSIWYG: What you see is what you get

2. Theoretical Foundations

This chapter delves into various applications, programming languages, frameworks, and related technologies pertinent to this work. Each component receives a thorough examination, covering its functionalities, potential use cases, unique features, benefits, drawbacks, and any existing limitations. By illuminating these strengths and weaknesses, I aim to equip readers with a comprehensive understanding of these technologies and their significance to the overall project. This section is meant as a reference point for the rest of the documentation. An explanation of what was achieved can be found in the Implementation section.

2.1. Software used

Many programs and other software were used in order to complete the must-goals. The following section will cover them in detail. To qualify as a program, a piece of software has to be executable.

2.1.1. Development

This section outlines the software used for the development of the application. The specific products were chosen for their support of the used languages as well as their ability to be installed and run on the Windows Operating System (OS). The personal preferences of the developer also played a large influence in the decision process.

2.1.1.1. Visual Studio

Visual Studio is an Integrated Development Environment (IDE) developed by the Redmond, WA-based technology company Microsoft. [1] It allows the developer to write code in multiple languages including C# and C++. It also provides tools for building and executing the written code such as compilers and tools to improve the developer experience, the most notable of which being a Debugger and several code completion tools. A debugger allows the developer to diagnose problems with the code by setting breakpoints, where the program is paused and the values of variables are shown. It also provides functionality to inform the developer when a variable changes. Visual Studio is available in 3 Versions:

- Community: Free and Open Source (FOSS) version with minimal proprietary additions; targeted towards students
- Professional: paid version; targeted towards small teams
- Enterprise: targeted towards large cooperations

The Community version was used for this project. [2]

In this project, Visual Studio was primarily used for the Backend of the application.

2.1.1.2. Visual Studio Code

Visual Studio Code (VsCode) is a code editor developed by Microsoft with a FOSS core, that also contains proprietary components such as telemetry. Although it is not a full-fledged IDE, as its core does not contain tools typically found in an IDE, such as debuggers and code completion utilities, it is still a very powerful tool thanks to its expansive and well-supported extension eco-system. It is thanks to said extensions, that VsCode has become very popular in the field of Web Development, as many developers are dependent on VsCode extensions when working with popular frameworks and libraries. When some commonly utilized tech-

nologies are used, no extensions are needed, since Microsoft has included tooling for web technologies out of the box such as a basic debugger and code highlighting for the popular programming languages JavaScript and Python [3] as well as EMMET, an extension providing productivity-enhancing abbreviations and snippets for HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). [4]

In this project, Visual Studio was primarily used when developing the Frontend of the application.

2.1.1.3. Docker Desktop/Docker

Docker Desktop is a Graphical User Interface (GUI) application for Windows and macOs used to manage Docker Containers. It is available in both a FOSS community edition and a paid commercial version. It allows the developer to create and maintain Docker Images as well as Kubernetes Clusters.

Docker itself is a system that gives developers the ability to quickly deploy software. It might be easier to think of it as a way to package an application, including all dependencies and configuration files into a self-contained and reusable package. Said package is referred to as a Container. Docker is often used in combination with a Continuous Integration/Continuous Deployment (CI/CD) pipeline, where code is automatically tested, built, and in the case of Continuous Deployment published.

Images are the core of Docker. Images contain instructions on how to build Containers. Therefore, it is possible to think of them as "Blueprints" for creating Containers. The instructions are defined in a Dockerfile containing information about the Container OS, startup commands, and software dependencies. Containers themselves are instances of Docker images. As Containers are isolated and self-contained, it is possible to run multiple instances of them at the same time. This helps improve the stability and security of the deployed application.

Even if a developer does not containerize their software, as is the case with this project, Docker can still be a useful tool, because many common dependencies such as databases are available as docker images, which removes the need to install them on the development machine itself. [5]

2.1.2. Other Software

This section outlines software that was used for various miscellaneous tasks other than than software development.

2.1.2.1. Mozilla Firefox

Mozilla Firefox is a FOSS Web Browser developed by the Mozilla Corporation. It began development in 1998 as a fork of the then open-source Netscape Navigator Browser, which had at one point been the most popular Web Browser in the early days of the internet. It had, however, lost a great bit of popularity at the time it was made open-source. Firefox in turn became a very popular browser in the 2000s, but it has seen declining usage for the last couple of years as the Chrome Browser, developed by Google, has taken the lead. Firefox now hovers around 3 to 10% market share depending on the region.

Firefox was used in this project to test and interact with the Frontend interface. It was preferred over Chrome or Browsers based on it, because it not only offers a greater amount of privacy but also better developer tools, especially when viewing CSS properties of HTML elements. [6]

2.1.2.2. Swagger User Interface (UI)

Swagger UI is an interface built upon the OpenAPI Specification for documenting an Application Programming Interface (API) by using either a YAML or JavaScript Object Notation (JSON) document to describe the available endpoints and the models used for data transfer. It was originally known as the Swagger Specification. Swagger itself includes a set of tools, which aid the developer in the design, development, and documentation of APIs.

Swagger UI is one of these tools. It helps the developer keep an overview of the API by generating a GUI out of the YAML/JSON document displaying all endpoints. It then allows for the testing of the endpoints by providing fields for each parameter, where test data can be entered and buttons to submit the API calls. [7]

2.1.2.3. Typst

Typst is a newly developed FOSS markup system primarily targeted toward academic use. It is intended as a replacement for LaTeX. Typst is heavily inspired by Markdown as it includes built-in syntax for commonly used text elements like bold/italic text, unordered/ordered lists, and headings. The system is entirely built with Rust, a modern type-safe high-performance programming language resulting in fast compile times compared to LaTeX.

Another major difference between Typst and older solutions is that it is not separated into a distinct markup and scripting language but is instead one language containing both capabilities. Its scripting side includes implementations of conditionals, loops, and data types including Strings, Integers, Floats, Arrays, and Dictionaries, all of which are commonly found in other programming languages making Typst very intuitive for programmers, while not being too complicated for regular users. Additionally, it supports many formats for storing bibliographies, including Hayagriva YAML and BibLaTeX files. Typst documents can be exported as Portable Document Format (PDF) and HTML files.

The markup and scripting components of Typst are very closely linked and they can both be embedded within each other. Declared variables are also shared between the two, making it very easy to for example create a list of names, programmatically modify it, and output the result in the space of a couple of lines in one file. The same procedure would require the use of multiple separate files in comparable systems. Typst also allows the user to programmatically change the properties of text elements anywhere in the document by using so-called Set Rules. The following two examples show how to add numberings to headings and how to justify the text of paragraphs:

```
set heading(numbering: "1.")
```

```
set par(
  justify: true
)
```

When used in text, programmatic elements have to be marked by writing a "#" in front of them. [8] Typst provides both a Code Line Interface (CLI) tool and a What you see is what you get (WYSIWYG) web-based editor for writing documents.

Typst allows for the creation of document-independent templates and packages. Even though Typst is still very new, there is already a thriving community of developers and general users creating said components and sharing them for free with the rest of the user base.

Typst itself is run as a startup and is headquartered in Berlin, Germany. The founders are Martin Haug and Laurenz Mädje. [9]

2.1.2.4. JabRef

JabRef is a bibliography management tool that allows the user to collect, categorize, organize, and keep track of sources. It supports searching many online repositories, the most well-known of which is Google Scholar. Additionally, it includes the functionality to extract metadata from PDF files. Collections of sources are stored as .bib BibLaTeX files, allowing them to be imported into the most common writing systems used in academia.

Sources can be categorized into several predefined categories including articles, books, thesis, and websites, each one of which requires the user to enter different information. The categories can also be edited by adding custom fields. [10] For example, this project required the addition of the date when a website was accessed.

2.1.2.5. Git/GitHub

Git is a so-called Version Control System (VCS). A VCS is an essential tool in software development. It is used to track changes made to a codebase, organize said codebase, and manage the collaboration between multiple developers in teams or open-source communities. Git was developed by Linus Torvalds, the creator of the popular FOSS OS Linux in 2005 to be a fully free and open version control standard. He created it due to the capabilities of the BitKeeper VCS, which had been used in the development of Linux, being reduced for non-paying customers.

There are a couple of concepts essential to the understanding of Git:

• **Repository**: Repositories represent the fundamental component of Git. They are used to store the code of a project or part of a project. Changes to files are also tracked on the repository level. On the technical level, it is a special database storing project revisions and history. Unlike in other similar solutions, a Git repos not only contains file data but also data about itself such as metadata and user configuration. That information is located in a ".git" directory, which is configured to be hidden. Git uses two different key structures to store data: the object store (efficiently copied during cloning) and the index (temporary and modifiable).

• **Branch**: To further segregate code changes and to avoid conflicts, where multiple developers, who are working on different features, end up overriding the work of their colleagues, Git provides the ability to create an unlimited number of branches on a repository. Upon a branch's creation, all changes made to code only affect the branch itself and are not reflected in the rest of the repository. A branch is created as follows:

```
git checkout -b branchName
or
git checkout branchName
or
git branch branchName
```

The default branch that is initialized when creating a repository has traditionally been called "Master". Due to negative historical denotations, the term "Main" is preferred nowadays.

- **Commit**: A commit is a snapshot of the current state of selected files in a repository. They are used by Git to record changes made to files over time. A single commit does not record the state of all files. Instead, Git looks for changes made to files by comparing the newly committed state to the previously stored one. Git then processes the changes found. All unchanged files and directories remain unaffected by the commit. A commit is the only way for a developer to change the files stored in the repository. Developers are also granted the freedom to choose which files are included in a commit. Each commit must have a descriptive message explaining the changes. Before creating a commit, files are typically staged using the `git add` command, creating a temporary list of changes to be included.

- **Merge**: If a developer is done working on a specific branch, they can integrate its changes into another branch, combining the commit history of both branches and bringing their changes together. This process is referred to as merging two (or more) branches. A branch is merged as follows:

```
git merge branchName
```

One problem that can occur when merging branches is Merge Conflicts. They occur when changes have been made to the same file in multiple branches. A developer or a team leader then has to manually inspect affected files and decide which version to keep and which to discard. One way to achieve this is to use the `git diff` command. This command is used to display changes between files. Other tools and strategies are also available to simplify conflict resolution. [11]

While it is possible to store a repository solely on a local computer, especially when working alone, it is way more common to host a repository on a server. Large organizations may have their private git servers, but there are also hosting sites, which allow users to manage repositories. The most popular option is the Microsoft-owned GitHub, which is used to host this project. In addition to simply hosting repositories, GitHub also allows users to host simple Websites and to track issues on repositories in addition to basic project management tools like a Kanban Board and CI/CD with GitHub Actions. [12]

2.2. Backend

In addition to the many programs used for completing this project, several technologies were also used as building blocks of the application. This section will cover all used in the Backend or the server of the project.

2.2.1. Programming Languages

Following section outlines which Programming Languages were used in the Backend.

2.2.1.1. C#

C#, or as it is pronounced "C sharp" is a general-purpose programming language developed at Microsoft by a team headed by the Danish software engineer Anders Heijlsberg. He was previously responsible for creating the Turbo Pascal and Delphi languages. At Microsoft, he also works on TypeScript, which will be explained later on in this document.

The language is written object-oriented, meaning that the core of each C# application is Classes, independent entities containing both variables and business logic. It also provides type safety, which in combination with the language's automatic memory management and garbage collector makes C# very user-friendly and reliable. Its syntax is often compared to that of the very popular language Java. Microsoft originally developed the language only for their own Windows OS, but it has since been made available for macOS and Linux as well. [13] It is part of the larger .NET Framework used to create many different kinds of applications. The original .NET Framework is only supported on Windows. A cross-platform version called .NET Core is also available. The most recent .NET Core revision as of February 2024 is .NET Core 8. [14]

2.2.2. Frameworks/Libraries

In addition to C# as the main programming language of the Backend, several Frameworks, Libraries and Packages were also used in cases where additional features were required.

2.2.2.1. Asp.NET Core

Asp.NET is FOSS, cross-platform, and lightweight Framework built as an addition to the previously mentioned .NET Core Framework. It enables developers to create high-performing Web Applications with the C# language. It is based upon the older Asp.NET Framework, which opposed to its newer counterpart is only available for Windows, as it is based on the regular .NET Framework. Asp.NET still receives updates. They are, however, small in nature. Asp.NET Core is a total rewrite of the previously existing functionality.

Asp.NET Core provides the ability to create both Model View Controller (MVC) and Web API applications. MVC describes a popular architectural pattern, where the application logic is split into three distinct parts:

• **Models**: Represents the data and manages how data is stored, retrieved, and manipulated.

• **Views**: The GUI page(s) the user interacts with. They are used to display the data provided by the Models and provide interactive elements to mutate said data.

• **Controller**: A Controller acts as the middleman between the View and the Model and facilitates the transfer of data across the application.

Web APIs on the other hand merely provide endpoints, to which other applications can send and request data. They do not include graphical interfaces themselves. It is however possible to also define endpoints on MVC Controllers so that other applications also have access to the data. [15]

This project uses a Web API without any MVC Components.

2.2.2.1.1. Entity Framework Core

Asp.NET Core also provides an Object Relational Mapper (ORM) called Entity Framework Core. An ORM is an abstraction layer between the application and the database. It is used to perform Create, Read, Update, Delete (CRUD) operations, without having to write database queries by hand. They typically provide interfaces to interact with database tables as if they were regular objects. One thing of note is that the term is reserved for relational databases only. Non-relational solutions use different terms (for example Object Document Mapper in MongoDb). Entity Framework Core supports SQLServer out of the box, although there are third-party packages, which add support for further databases. [16]

2.2.2.2. AutoMapper

To convert between a Model and a Data Transfer Object (DTO), it is normally required to manually assign the values as follows:

```
#
model.name = dto.name;
model.date = dto.date;
model.price = dto.price;
```

Doing this can not only lead to a lot of filler code, but it may also be impossible with larger objects. DTOs themselves will be thoroughly explained later in this document. AutoMapper is a library that allows the developer to define maps between certain models/DTOs. It will then try to automatically match corresponding fields. Should this fail, the developer is able to manually create maps as well. Maps are not bidirectional, so each one has to be created twice. AutoMapper includes an easy way of doing this by simply defining the map a second time and adding `.ReverseMap()` at the end of one of them. [17]

2.2.2.3. SignalR

In a traditional Web API, clients can only send requests to the server, to which the server may then respond. This system works well for certain use cases, but for others, for example, chats or games, it is too limited. Instead, these applications may benefit from the use of another technology, namely WebSockets. They differ from traditional HyperText Transfer Protocol (HTTP) endpoints by establishing a persistent connection between clients and servers with two-way communication channels. The technical term for communication of that sort is "Full-Duplex Communication". Messages transmitted over WebSockets are also delivered or received in real time. Although their usefulness is without doubt, WebSockets also possess certain limitations, which a developer has to keep in mind when using them:

• **Poor Security:** WebSocket messages are by default not transported securely. This may lead to unauthorized access to systems and data breaches.

• **Browser Compatibility:** Some older Browsers may not support WebSockets. Some Browsers also require other metadata or security certifications when using them.

• **Complexity:** Compared to HTTP endpoints, WebSockets are more difficult to use and implement for the developer. The developer also has to maintain the connection and manage clients, which adds even more difficulty.

As a solution to said problems, Microsoft released the SignalR library for .NET and .NET Core. It enables developers to work with WebSockets easily. At the core of SignalR are so-called Hubs. There, functions can be created, which correspond to a certain task performed by the application. Within them, business logic is defined and Model data is interacted with. Should the changes to the application state performed in a function require updates in the clients, messages can be sent to either all clients, groups of clients, or specific clients. SignalR is also very easily combined with Microsoft Identity to associate specific clients with logged-in users.

In the clients, for example, a website, SignalR is also available as a package. Once installed, event listeners can be defined, which waits for a specific message from the server. Each message requires its own listener. Additionally, clients can also call the previously discussed functions on the server by invoking them with their name and, if needed, adding data needed in the Backend.

SignalR also allows for the creation of a custom function that is executed on a client establishing a connection by overriding the `Hub.OnConnected()` or `Hub.OnConnectedAsync()` methods. It is also possible to invoke functions on the Hub on the same ASP.NET application by using a HubContext. Lastly, it has to be mentioned that SignalR also supports other real-time communication technologies other than WebSockets. They are, however, excluded in this documentation, since they are not used. An example of one of them would be Server Side Events. [18]

2.2.2.4. Bcrypt.NET

Bcrypt.NET is a .NET library implementing the popular Bcrypt password-hashing function. [19]

Bcrypt was designed by Niels Provos and David Mazières in 1999 and is based on the 1993 developed Blowfish Cipher, itself created by American cryptographer Bruce Schneider. Although there are newer ciphers and algorithms out there, Blowfish even has a successor called Twofish, Bcrypt is still considered secure and widely used thanks to its availability on many platforms.

It does not only just hash passwords, it also salts them. Salting adds additional security compared to just hashing, where passwords are merely encrypted using a predefined algorithm. One downside of doing things this way is that hackers retain the ability to use so-called Rainbow Tables, which are created by first gathering a list of commonly used passwords, most of them regular words without major alterations. Frequently used tricks such as replacing the letters "O" and "E" with the numbers 0 and 3 respectively are also taken into consideration. This list is then run through an algorithm to get a list of hashed passwords. Salting mitigates this issue like so: When passwords are hashed, in addition to simply applying the algorithm a random string of numbers and characters is used as a second argument to the hashing function. This string is referred to as the Salt. It is used inside of the hashing function to modify the resulting hashed password so that encrypting the same password with multiple Salts results in different outputs rendering Rainbow Tables useless.

In addition to Salts and to distinguish itself from competing solutions, Bcrypt is also significantly slower than other algorithms, making regular encryption and decryption still possible, but rendering Rainbow Tables time-consuming to produce, even without the use of Salts. [20]

2.2.3. PostgreSQL

The database used for this project is PostgreSQL, a FOSS relational database. Similar to other relational databases like MySQL and SQLServer, it stores data in tables divided into rows and columns. Another shared characteristic is that most relational databases use Structured Query Language (SQL) to write queries performing CRUD operations on stored data. What differentiates PostgreSQL from other solutions is that it is a so-called Object Relational Database, meaning that it supports concepts like custom data types and inheritance typically found in Object Oriented Programming (OOP). It also allows multiple transactions to be run at the same time, by assigning a snapshot of the database to each one and for reusable queries. [21]

This project makes use of some of these features since it uses Enums to store certain data, which are not available on other relational Database Management System (DBMS). Further details about said implementation will be covered in the Backend part of the implementation section later in this documentation. One downside of using PostgreSQL with Entity Framework Core is that it requires the installation of a third-party package called `Npgsql` to be supported.

2.2.4. Concepts

The following concepts are vital to understanding the implementation of the Backend.

2.2.4.1. REST

Representational State Transfer (REST) APIs are APIs, that use HTTP technology. The term REST itself merely describes a set of guidelines, which are recommended when designing HTTP-based APIs. They were laid out by Roy Fielding, who co-founded the Apache HTTP Server Project, in his Ph.D. thesis written in the year 2000. APIs that utilize these guidelines are called "RESTful".

Having well-designed APIs is essential to the modern web, where developers not only work with their own APIs but also with several others hosted on other servers in the web. Therefore, it is important, that all APIs are designed similarly. The core of REST is defined by its Uniform Resource Identifier (URI) format:

```
URI = scheme://authority/path[?query][#fragment]
```

- **Scheme** usually denotes the protocol used (for example HTTP or File Transfer Protocol (FTP))

- **Authority** is the place, where the server is hosted (Internet Protocol (IP) Address or Domain name)

- **Path** is used to separate and organize data/pages on the server. Subsequent paths separated by slashes always indicate a hierarchical relationship

- **Query** is optional and is for example used for data entered by a user (Search parameters, …)

- **Fragment** usually links to a specific part of data/a page (for example a heading)

Although there are several other smaller rules, six concepts build the core of REST together with the URI format. Some of these have already been alluded to previously:

1. **Client-Server:** There should always be a separation of concerns between clients (the applications processing the data) and the server (the application providing the data). Their interactions must be defined through standardized requests and responses, ensuring independence and scalability.

2. **Uniform Interface:** Resources are identified using URIs and accessed through a standard set of HTTP methods (GET, POST, PUT, DELETE). This consistency makes API usage intuitive and predictable for developers.

3. **Layered System:** This rule specifies, that it is possible for there to be middlemen, such as proxies between the client and the server as long as they are not visible to the client. They may, however, still block certain data from reaching the client. Examples would be Website-blockers on a school network.

4. **Cache:** Data should always be cachable anywhere along the network path. The responsibility to declare data as cachable is a server's job. Caching is essential to the modern web, as it massively decreases the time and cost of data transfers.

5. **Stateless:** Each separate request must include all information the server needs to be processed independently. A server should not be relied upon to have any knowledge about each client and data associated with them and it should also not store said knowledge. This improves scalability and prevents issues with server-side session management.

6. **Code-On-Demand:** Servers may transfer executable code to the client temporarily It also has to be ensured, that the server has to make sure, that the client can execute said code. An example would be JavaScript files. [22]

2.2.4.2. DTOs

In software development, a DTO is like a specialized messenger carrying only the data needed between different parts of an application. In their implementation, DTOs look very similar to regular models. The major difference is that they are not meant to represent data stored on the database, but instead data that is sent and received. A DTO is usually associated with a model and includes all fields from said model needed for a certain data transfer. They are a good way to obfuscate data and are categorized as a Design Pattern. [23]

An example for a use-case of a DTO from this project is that when information about a specific question of a quiz is transferred to a client, any data containing information about wether or not an option is correct or not is omitted to prevent students from cheating.

2.2.4.3. Repository Pattern

The Repository pattern in ASP.NET is a design pattern, where so-called Repositories are created, that act as a middleman between Entity Framework Core and the API Controllers. All functions performing CRUD operations are defined in the Repositories, so that a Con-

troller never has to work with the database directly. The Repository pattern brings several advantages:

- **Separation of Concerns:** The repository keeps business logic decoupled from the intricate details of data access, leading to cleaner and more maintainable code. This is especially important for larger teams, where many developers work on a single codebase.

- **Reusability:** A single repository can serve multiple parts of the application, reducing code duplication and promoting consistency.

- **Flexibility:** If you need to switch data sources, you only need to modify the repository implementation, keeping the rest of the application intact. [24]

2.3. Frontend

This section will cover all technologies used in the implementation of the Frontend part of the application. The Frontend encompasses all parts of the program that a user interacts with. For example, all GUIs are part of the Frontend.

2.3.1. Programming Languages

Following Programming Languages were used in the Frontend.

2.3.1.1. JavaScript

When working on websites, developers have a limited number of technologies they can use, as most Web Browsers only support a small number fraction of what is theoretically available. Generally, only three languages are used to create websites: HTML for the markup of the general layout of the page, CSS for styling elements of a page, and JavaScript to add interactivity to a page.

Of the three, JavaScript is the only one that can be understood to be a Programming Language, although it is often referred to as a scripting language as well. In addition to common features such as Loops, Variables, and OOP, it also supports a wide variety of features specifically designed to enhance its usefulness to Web Development, as it was designed for that task. For example, it is possible to programmatically interact with elements of a HTML page by selecting them with `document.querySelector()`. It is then treated as a variable, which can be used to dynamically adjust the CSS properties of the element to modify its content. The built-in API, that provides those features is referred to as the Document Object Model (DOM) API. The name comes from the fact that a HTML document is represented as a logical tree, with each node of said tree representing an element of a page and each underlying node representing its children. [25]

2.3.1.2. TypeScript

As useful as JavaScript is to Web Development, it is also widely considered to be quite flawed. One major aspect, that is often criticized is its lack of a proper typing system. JavaScript uses so-called dynamic typing, which means that a developer does not have to specify a data type when creating a variable. This alone is not the main reason why this aspect of the language is often panned, as it is also found in other, very popular and widely praised options such as Python. The real reason is that JavaScript is very inconstant in its type-checking behavior. For example the expression `"5"` `==` `5` checking if the right value is

the same as the left one returns true, even though the values compared are a string and a number. This is the case because in JavaScript the common "==" operator does not take into consideration the type of the values. Instead, the developer is forced to use the "===" operator. JavaScript also has so-called truthy and falsy values, where non-boolean expressions such as an empty string are interpreted as false and a non-empty string as true.

All these particularities make it very easy to make mistakes when writing JavaScript, which is often quite difficult to debug. This also makes JavaScript very unsuitable for larger projects, which is why TypeScript was developed.

TypeScript is a superset of JavaScript, which means that all valid JavaScript code is also valid TypeScript code. TypeScript main addition is static typing, meaning that it makes it possible to define the data type of a variable, although it is not strictly necessary, because TypeScript automatically tries to figure out the type of a variable on creation if initialized with a value. It also offers advanced type-checking capabilities and an improved class system. TypeScript also allows for the creation of advanced types for situations such as when an API request returns data consisting of multiple values, which should be stored together. This is achieved by using the `type` keyword. It is very similar to an interface, although it cannot directly inherit other types and it also supports many advanced typing features not found on interfaces. [26]

This project uses TypeScript as opposed to regular JavaScript.

2.3.2. Frameworks/Libraries

Common web technologies are very useful for smaller applications. As the complexity of an application grows, it becomes increasingly difficult to manage the codebase, which is why several Frameworks and Libraries were used to simplify the development process.

2.3.2.1. React

When creating larger web applications, it is very common to use the same elements more than once. For example, a website could have the feature to enable a dark mode with a toggle on every page. if developed traditionally, this toggle would have to be manually implemented on each page, which for a small element such as this might not seem like much work. However, as the the application scale and the complexity increases, this approach becomes unsustainable. Imagine being tasked to implement the same complex navigation bar on multiple pages. This redundancy not only wastes development time but also creates inconsistencies and makes maintenance of the codebase harder.

There are, however, many Libraries and Frameworks that aim to solve this problem by allowing the developer to create reusable blocks called Components. The most popular solutions are Angular and React, the latter of which is used in this project. Components are self-contained, customizable, and readily interchangeable.

In the previous example, the dark mode toggle becomes a Component that, after it has been implemented once, can be used anywhere. In the case of the navigation bar, where the specific links may change, it is also possible to define the content as Props, values that can be passed to a component similarly to arguments of functions. React utilizes a virtual DOM, which is an in-memory representation of the actual DOM mentioned in the section about

JavaScript. It allows React to efficiently identify and update only the necessary parts of the UI when data changes, leading to faster and smoother user experiences. Components also make it very easy to maintain code down the line. Imagine updating the toggle behavior – you do it in one place, and the change automatically reflects across all instances.

React supports both JavaScript and TypeScript. It also allows the developer to write HTML markup directly in the JavaScript/TypeScript file using JavaScript Extensible Markup Language (XML) (JSX). [27]

2.3.2.2. Vite

One problem that occurs when working with Libraries such as React is that Web Browsers do not natively support them as they only know how to work with HTML, CSS, and JavaScript, which means that code written in React first has to compiled to technologies understood by Browsers. This step also requires including or bundling in all dependencies required. The traditional tool of choice for this task was WebPack, but in recent years, another option, namely Vite has increased in popularity.

The main advantage of Vite is that it is much faster than WebPack. Vite achieves this by supporting ES Modules, the official module system for JavaScript. This increases efficiency because code no longer has to be bundled into a single file, which greatly reduces complexity. This makes it also possible for Vite to only update the modules for certain parts of the application since code is bundled into multiple different files. This is particularly useful for development, as it decreases reload times when making small changes to code. Vite is also able to determine if certain parts of code are really necessary to the application and then remove said code from the bundle, reducing its file size. [28]

2.3.2.3. Chakra-UI

When working with React or another Component-based Framework/Library, it soon becomes evident, that many different components occur on almost every page. A perfect example of such an element would be a Button or even something more complicated such as a Card or a Dialog/Modal. Additionally, as every Frontend developer knows, it usually takes quite a long time to decide on how the page should look and even longer to subsequently define all the necessary CSS rules.

Luckily, several packages and extensions aim to reduce the time it takes to implement standard Components and to style them. Their way of doing so varies greatly, but Chakra-UI, the solution used in this project, approaches this problem by providing the developer with several predefined and styled Components. These range from commonly used elements like the ones mentioned above to more abstract ones like one Component that is a container with the CSS flexbox display property. Chakra-UI also allows for basic styling of its Components by passing Props and it provides the functionality to easily make the website responsive (usable on smaller screens) and accessible. [29] The following shows a Button that is inside of a Container with flexbox:

```
<Flex justify="center" gap="2px" direction="row">
  <Button colorScheme="green" p="1px">Save</Button>
</Flex>
```

As is shown in the example given, it is very easy to work with Chakra-UI. Specific Props will not be explained here or in the following documentation of the implementation with a few exceptions.

Chakra-UI is installed using the Node Package Manager (NPM). It is then required to wrap the main Component in a ChakraProvider Component. Afterwords, Chakra-UI can be used anywhere in the application. [29]

2.3.2.4. React Router

When working on more complex projects, React alone is often not enough, since it lacks any features to declare multiple routes with different pages. There is, however, a library that adds such support called React Router. It offers several different kinds of routers. This project uses a BrowserRouter, where the current location is stored in the Browser's address bar. A BrowserRouter allows the developer to define routes and children routes by adding a path and a Component for the page like this:

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Root />,
    loader: quizLoader,
  },
  {
    path: "/editquiz/:quizId,
    element: <CreateEditQuiz />,
    loader: loader,
    action: addQuestionAction,
    children: [
      {
        path: "question/:questionId",
        element: <ShowQuestion />,
        loader: questionLoader,
      }},
```

The main Component then has to be wrapped in a Router Component in order for the routes to work.

In addition to simple routing, React Router also supports more advanced features such as loaders and actions, as shown in the example above. Loaders allow the developer to define functions that automatically get executed when the component loads. This is used to load data from an external data source (API). Actions on the other hand are more complicated. They emulate the standard HTML feature, where, when a form is submitted, a HTTP request is made. Actions intercept said request and relay the form data to a specifically defined function on a route. There, the data is processed and sent to a Backend Server. If there is only one action defined per route, said action is automatically executed on a form submit. Otherwise, the route of the action desired has to be defined on the Form.

React Router also allows for the creation of Links to redirect to another page. Redirecting is also possible as a return value of an action or via a React Hook called `useNavigate()`. [30] Hooks will be explained in the next section.

2.3.3. Concepts

The following concepts are vital to understanding the implementation of the Frontend.

2.3.3.1. React Component Types

React offers the ability to create two different types of Components: **Class** Components and **Functional** Components. Class Components are the older way, available since React was first released. They are essentially JavaScript Objects/Classes. They allow for the use of Lifecycle Methods, which perform actions at specific stages of the component's lifecycle (creation, update, deletion). They also allow for the management of the component state. This means that it is possible to store data specific to the component instance and to modify that data at any time. Functional Components, on the other hand, rely on functions for their implementation. When displayed, Functional Components act like a standard function and are executed once. They return JSX markup. Due to that behavior, they do not support Lifecycle Methods and dynamic state management. This made them by far the less popular option for a long time until Hooks were introduced. Hooks add the previously missing features to Functional Components. The by far most used Hook is the useState Hook. It allows for state management. It is initialized with a value and a setter function. Once the setter function is called, the entire Component gets rerendered with the updated data.

```
const [name, setName] = useState<string>("Simon")
```

In this example, the name is initialized with the value "Simon". The specification of the type is required only when working with TypeScript.

Functional Components are generally preferred nowadays since they require less Boilerplate code and are more performant. They also receive more support than Class Components, which are considered outdated at this point. Another reason for the popularity of Functional Components is that Classes behave very irregularly in JavaScript, which can lead to many bugs. Functions, on the other hand, are considered more reliable. [31]

3. Implementation

This chapter of the diploma thesis will cover the practical implementation of the project. It is divided into several chapters, where information about the architecture and code is separated into sections loosely based on the must-goals. It will also contain graphics and images, which aim to provide an even better look at the finished product.

3.1. Architecture

Even though the application is run locally, it still relies heavily on technologies commonly found in web development. A major deciding force for employing going this route was that the project is very similar to web applications in not only its concept (all available solutions, which were used as inspirations for this project are web-based) but also its general composition. The reason for this similarity is found in the fact that the application has to be accessible from not only a tutor's PC but from a student's device as well.

This sameness also extends to its general architectural Design, which like web applications is divided into a Frontend and a Backend. The former includes all Clients and GUIs accessed by students and tutors, and the latter contains a Server, which is responsible for storing and managing Quiz/Game data and for providing access to said data via multiple methods, which are outlined in Figure 1:
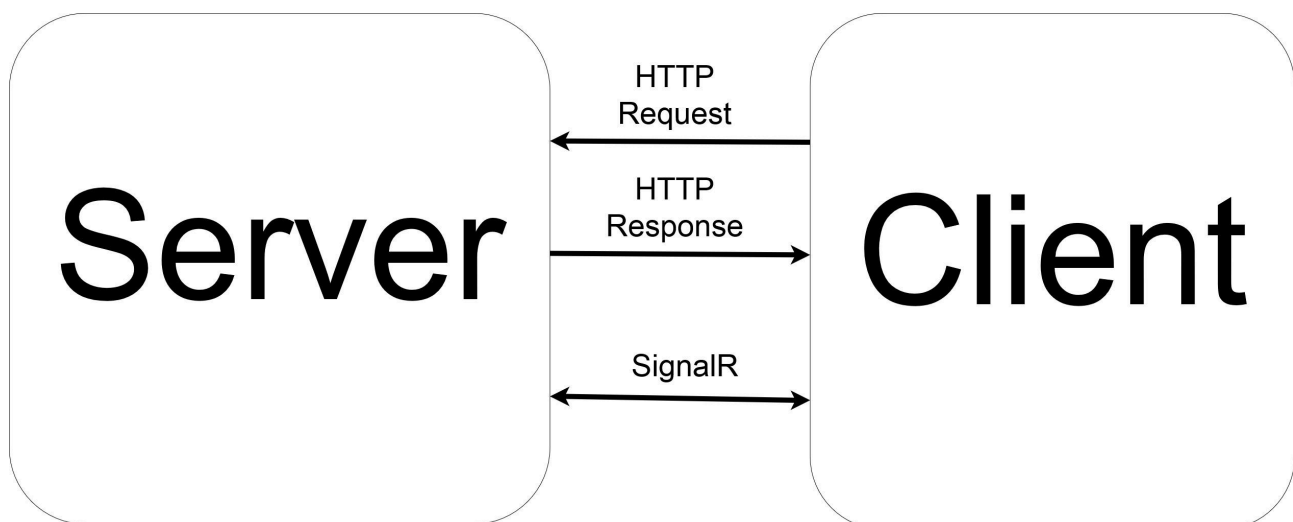
Figure 1: General Architecture Overview

As seen in the Figure 1, communication occurs both via HTTP, where data is requested by the Client and then subsequently provided/mutated/stored by the Server, as well as bidirectional SignalR WebSocket connections.

3.1.1. Backend
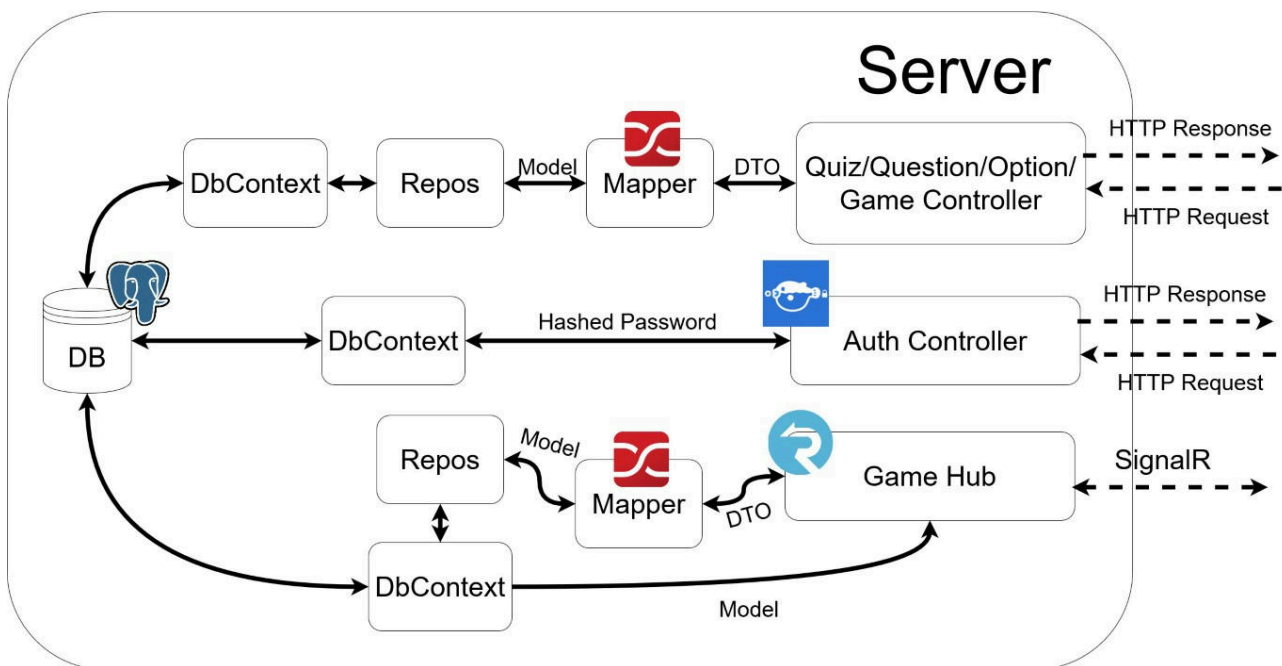
Figure 2: Backend Architecture Design

Logo Reference:

Figure 3: Postgres[32]   *Figure 4: AutoMapper[33]*   *Figure 5: SignalR[34]*   *Figure 6: Bcrypt.NET[35]*

Figure 2 shows the general composition of the Backend Server application. It is developed in Asp.NET Core and thus written with the C# programming language. It contains all Models, DTOs, Repositories, and Controllers needed for managing the persistent storage of Quizzes and Games in addition to basic user verification capabilities. All of these components will be explained in detail in their own sections. Information is stored in a PostgreSQL database, which is run as a Docker Container. It is initialized using a `docker-compose.yaml` file:

```
1  services:
2    postgres:
3      container_name: quap
4      image: postgres:latest
5      environment:
6        POSTGRES_USER: user
7        POSTGRES_PASSWORD: pwd
8      ports:
9        - 5432:5432
```

Code Snippet 1: Postgres Docker Compose

The difference between a `docker-compose.yaml` and a `dockerfile` is that the former is used to run and configure Docker Containers, while the latter is used to do the same for Docker Images. [36]. This is also evident from the code segment provided above, where an already created image providing the latest version of PostgreSQL is specified. The only thing left to

configure is the username and password for the database. Here, they are still hard-coded, as the must-goals do not require a more complex solution, but it would be better for larger projects to obfuscate that information. This, however, is outside the scope of this project.

The DbContext shown in Figure 2 is a class that utilizes Entity Framework Core. It is used to configure what to store in the database and how to do so. Further detail will follow in sections describing various Models. At the core of the Asp.NET Core project is the `Program.cs` file, which is used for configuration.

3.1.2. Frontend
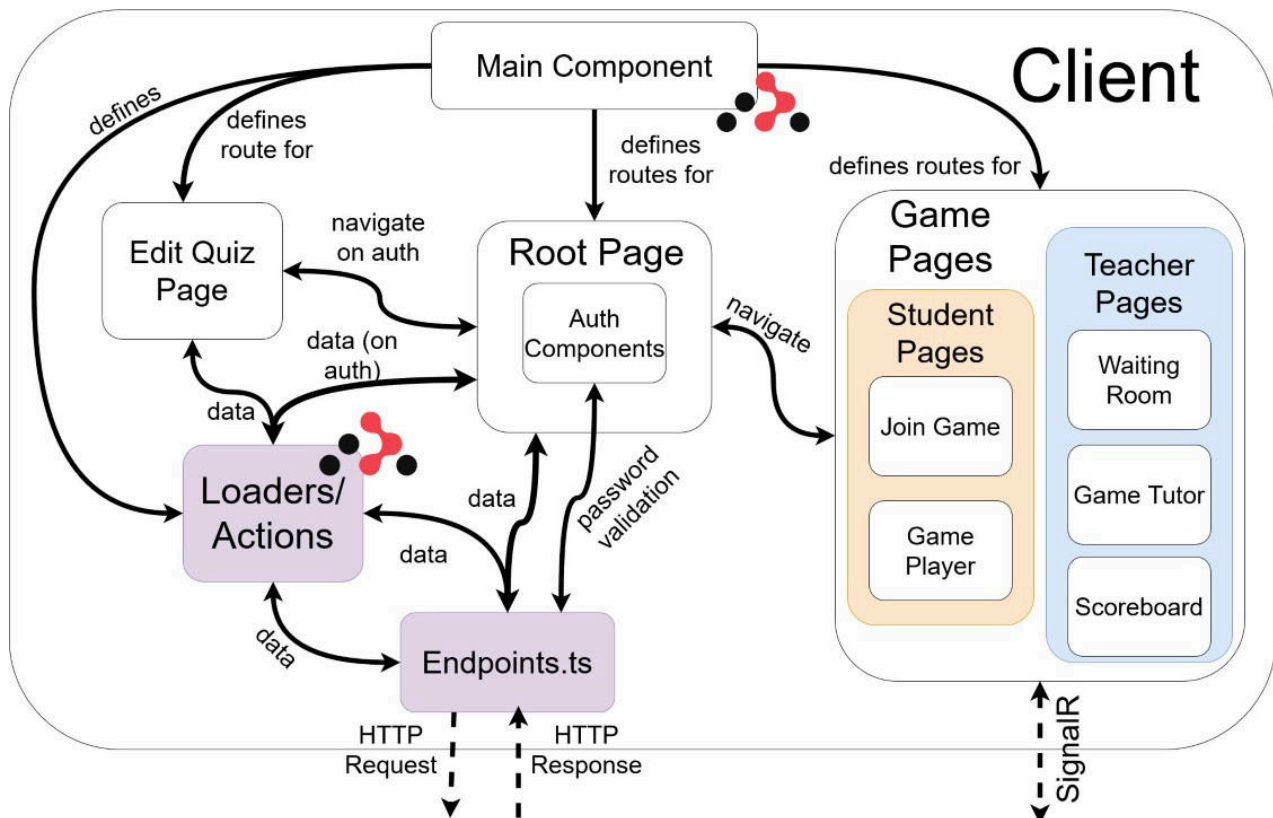
Figure 7: Frontend Architecture Design

Logo Reference:

Figure 8: React Router[37]

The Frontend of the project is a complex React application. It is divided into two main sections, which are different in both their implementation and use case. The first one is the Game Pages, which contain all the functionality needed for playing a game. They are shown in their own section in Figure 7. This part of the application mainly uses SignalR connections to communicate with the Server. The other section is the Quiz Management interfaces, entirely used by the tutor. Here, they are able to create new Quizzes, edit existing ones, or delete Quizzes, which are no longer used. These functions are separated into two pages:

- **Root Page:** Default page of the client. Displays all Quizzes and lets the tutor edit, create, delete, or start games with quizzes.

- **Edit Quiz Page**: This page is navigated when editing a Quiz. Here, individual Questions and Options are visible and editable.

Deleting and editing Quizzes requires the tutor to enter a password for verification. This segment of the Frontend relies on HTTP endpoints for interactions with the Backend, making significant use of React Router Loaders and Actions to manage data. All code needed to perform Requests to the HTTP API is located in the `endpoints.ts` file, from where it is imported into other parts of the application. All routes needed are defined in the Main Component.

3.2. Quiz Management

A major part of the planned project was a tutor's ability to manage Quizzes. This includes the creation, editing, and deleting of Quizzes and their containing Questions and Options. The aim was to create user-friendly interfaces to perform these tasks and for the Quizzes to be stored perpetually.

3.2.1. Backend

In the Server, this section includes a RESTful API providing several endpoints for managing Quizzes.

3.2.1.1. Models/DTOs

In order to properly store data, the Creation of multiple Models is required. A model is a class used as a reference by Entity Framework Core to create SQL queries to Initialize tables. Each Model represents one of these tables. This section includes three Models: Quiz, Question, and Option. Of those three, Option is the lowest in their shared hierarchical order as it is part of a Question, which in turn is part of a Quiz. The Option Model includes distinct fields that represent different essential data. The Model itself is created as a regular C# class:

```csharp
1   public class Option
2   {
3       [Key]
4       [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
5       public Guid OId { get; set; }
6       public string OptionText { get; set; } = String.Empty;
7       public bool IsCorrect { get; set; } = false;
8       public DateTime CreationDate { get; set; }
9       = DateTime.Now.ToUniversalTime();
10
11      public Guid QuestionId { get; set; }
12      public Question Question { get; set; }
13  }
```

Code Snippet 2: Option Model

The annotation `[Key]` is used to define the ID of an Option as the primary key and the line below tells Entity Framework Core to automatically generate a random ID when creating a new entry in the database. The CreationDate is needed for sorting Options. Note that an Option

also includes a reference to the Question containing it. This is done by adding fields for the Question ID and the Question itself. This is not strictly necessary when all a developer wants to do is use a model in another model as part of a composition dependency. It is however necessary for Entity Framework Core to properly assign foreign keys in the created database tables. Including both is considered a best practice, although it is not strictly necessary. An interesting fact was, however, discovered when testing several methods of assigning foreign keys as part of the planning stage of this project: Including both a reference to the parent object and its ID is necessary if Entity Framework Core should perform Cascading Deletes (Children of an Object are also deleted from the database when said object is deleted. This is useful since it removes the need for the developer to carefully manage the database to prevent no longer used data from being stored.). Not including any reference to the containing Model on the other hand is advantageous when Model data is referenced elsewhere or needed again at a later time. This observation was also made use of when storing Games.

As it was clear from the example given of the Option Model, Models tend to not be very complex as they simply contain definitions of fields. The Question Model is structured very similarly:

```
 1  public class Question
 2    {
 3      [Key]
 4      [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
 5      public Guid QuestionId { get; set; }
 6      public string QuestionName { get; set; } = string.Empty;
 7      public int NOptions { get; set; } = 0;
 8      public List<Option> Options { get; set; } = new List<Option>();
 9      public QType Type { get; set;} = QType.SingleChoice;
10      public int ?TimeLimit { get; set; }
11      public int ?Points { get; set; }
12      public DateTime CreationDate { get; set; }
13      = DateTime.Now.ToUniversalTime();
14
15      public Guid QuizId { get; set; }
16      public Quiz Quiz { get; set; }
17    }
```

Code Snippet 3: Question Model

There are, however, also several differences that deserve an examination. Line 8 shows the first example of how Models are used within other Models. It is evident from the example given that this is not very difficult to do. They can simply be used like any Class in OOP. The time limit and points are declared nullable by prefixing them with a ?. This is done because those properties are not set when creating a question and are added later. The biggest addition to the Question Model is the use of an Enum. An Enum is a set of constants, where each keyword is associated with an integer index. They are used when a variable can have multiple predefined states that can be switched between. [38]. The definition for the Question type Enum is done as follows:

```
1  public enum QType
2  {
3      SingleChoice, MultipleChoice
4  }
```

Code Snippet 4: Question type Enum

SingleChoice is assignable both with the integer: `Qtype[0]` and by using its name: `Qtype.SingleChoice`. The same goes for MultipleChoice. An Enum was used in this case, since it provides more flexibility if new Question types are introduced in the future and it improves the readability of the code.

The Quiz Model is structured very similarly to the Option model. It is, however, still important to the understanding of the rest of this documentation to know what is defined.

```
1  public class Quiz
2  {
3      [Key]
4      [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
5      public Guid QuizId { get; set; }
6      public string Name { get; set; }
7      public string Description { get; set; }
8      public int NQuestions { get; set; } = 0;
9      public List<Question> Questions { get; set; } = new List<Question>();
10     public DateTime CreationDate { get; set; }
11     = DateTime.Now.ToUniversalTime();
12  }
```

Code Snippet 5: Quiz Model

Each Model is also mapped to several DTOs to more efficient transfer data. The DTOs contain the same fields as the Models, although some of them are omitted as most use cases do not require all data. The mapping is done with AutoMapper and maps are configured in the `AutoMapperProfile.cs` file. Most maps can simply by letting AutoMapper assign each field of the DTO to its corresponding field on the Model and vice versa. Some, however, require the manual assignment of values:

```
1  CreateMap<Quiz, QuizDto>().ForMember(dest => dest.QuizId, opt => opt.MapFrom(src
   => src.QuizId));
2  CreateMap<Quiz, QuizDto>().ReverseMap().ForMember(dest => dest.QuizId, opt =>
   opt.MapFrom(src => src.QuizId));
```

Code Snippet 6: Manual AutoMapper assignment

In addition to configuring an AutoMapper Profile, there is another step that is required for the Models to become functional: A DbContext has to be created, which allows Entity Framework Core to work with the Models and allows the developer to store the Model data in the desired database by creating DbSets for each Model. It also allows for querying of data by performing

CRUD operations. The following example shows a simplified version of the DbContext used in this project. It only includes the definition of one DbSet and the needed configuration:

```csharp
1  public class QuizManagementDbContext : DbContext
2  {
3    public QuizManagementDbContext(DbContextOptions<QuizManagementDbContext>
   options) : base(options) { }
4    public DbSet <Question> Questions { get; set; }
5
6    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
7    {
8      optionsBuilder.UseExceptionProcessor();
9    }
10   protected override void OnModelCreating(ModelBuilder builder)
11   => builder.HasPostgresEnum<QType>();
12 }
```

Code Snippet 7: DbContext

The OnConfiguring method is run when a migration is created. A migration has to be created manually each time the Models are updated. They automatically generate code, that defines tables and relationships in the database. These are created without manual configuration with one exception. As Enums are only supported by PostgreSQL, they have to be manually assigned to the database.

This DbContext then has to be configured in the main `Program.cs` file. There, the Connection String, which includes all needed information needed to connect to the database (Username, Password), is used to initialize the connection to the database. The String is typically stored in a JSON, from where it is imported.

3.2.1.2. Repositories

The next step after the creation of the needed Models is to declare functions, which make use of said Models and their corresponding database tables to perform certain actions needed on the Frontend. As this project uses the Repository Pattern, the first task of this step is to create Repositories for each Model. Only the Question Repository will be shown in detail since they are all very similar.

Before creating the Repository itself, an interface has to be created first, which is then implemented by the Repository:

```csharp
1  public interface IQuestionRepository
2    {
3        Task<Question> GetQuestionById(Guid Id);
4        Task<Question> CreateQuestion(Question question);
5        Task<Question> UpdateQuestion(Guid Id, CreateUpdateQuestionDto question);
6        Task DeleteQuestion(Guid Id);
7        Task<Option> AddOption(Guid Id);
8        Task<IEnumerable<Question>> GetAll(Guid id);
9    }
```

Code Snippet 8: Question Repository Interface

This is done to create an even clearer separation between what the code needs (data access functionalities) and how this is achieved (specific implementation using a database or API). It is also essential because it allows for greater flexibility when creating the repositories since IRepositories simply have to be included in other parts of the application by using Dependency Injection. This is useful since it allows for the complete replacement of the specific repository implementation without having to change all other parts of the application and without having to delete the old implementation. Dependency Injection will be further explained in the next section.

After creating an IRepository, an implementation has to be created. There, built-in Entity Framework Core functions are used to mutate data stored on the database. The following shows the implementation of the DeleteQuestion method:

```
1  public async Task DeleteQuestion(Guid Id)
2  {
3      Question question = _context.Questions.FirstOrDefault(q => q.QuestionId
   == Id);
4      Quiz quiz = await _context.Quizzes.FirstOrDefaultAsync(quiz =>
   quiz.QuizId.Equals(question.QuizId));
5      _context.Questions.Remove(question);
6      quiz.NQuestions--;
7      await _context.SaveChangesAsync();
8  }
```

Code Snippet 9: DeleteQuestion implementation in QuestionRepository

This function takes an ID of a Question as its argument. This ID is then used to search for the specific Question and the containing Quiz in the database. The Question is then removed from the database and the NQuestions variable, showing how many Questions a Quiz has, is decreased by one. Finally, the modifications are stored by saving the changes in the DbContext. The function is declared as async because interacting with the database includes I/O operations, which can be slow and block the main thread. By making it asynchronous, the function allows other parts of the application to continue running while waiting for the database operation to complete. This means that overall responsiveness and performance are greatly improved. When declared to be asynchronous the function return type has to be a `Task`. It represents a unit of work that is performed in an asynchronous fashion. It tells other parts of the program that if awaited, some action will be run in the background and the program can wait for said action to complete or continue executing other code while waiting. While not strictly necessary when developing Web APIs, asynchronous functions are considered best practice. [39]

All other functions in the Question Repository and their equivalents in the Quiz or Option Repositories follow the same pattern as the DeleteQuestion method shown in the code snippet. Thus, they will not be shown here. One thing that has to be mentioned, however, is that not all functions make use of AutoMapper for Model conversion, since it is not very easy to use AutoMapper when updating an existing instance of an Entity, meaning that for example,

an already existing Quiz has to be updated with data from a DTO while retaining its ID. This is theoretically possible but it takes a lot of extra configuration, which is too much extra work for a project this size.

In order to work properly, the Repositories have to be registered in the main `Program.cs file.`

3.2.1.3. Controllers

The final step that has to be implemented before the REST API can be used in the Frontend is the creation of the API Controllers. There, specific HTTP endpoints are created, which provide an interface for the client to interact with the previously defined Repository methods. As all Controllers (Quiz Controller, Question Controller, Option Controller) are very similar in their implementation, only the Question Controller will be thoroughly explained in this document.

The first step when creating an API Controller is to declare the general route, where all endpoints can be accessed. This is done by using the following annotation about the class declaration: `[Route("api/[controller]")]`. The name of the Controller is automatically inserted. The next step is to include all dependencies needed to interact with the data. These might include Repositories, the DbContext, and an AutoMapperProfile. (The Repositories also use this method to inject the DbContext.) The following snippet shows such a configuration. (The snippet has been changed so that the data types of the variables are not shown in the constructor. This is done for space-saving reasons):

```
1  private readonly IOptionRepository _optionRepository;
2  private readonly IQuestionRepository _questionRepository;
3  private readonly IQuizRepository _quizRepository;
4  private readonly IMapper _mapper;
5
6  public QuestionController(optionRepository, questionRepository, quizRepository,
    mapper)
7  {
8      _optionRepository = optionRepository;
9      _questionRepository = questionRepository;
10     _quizRepository = quizRepository;
11     _mapper = mapper;
12 }
```

Code Snippet 10: Controller Dependency Injection

The dependencies are declared as read-only private variables and then retrieved using Dependency Injection. Dependency Injection is a Design Pattern, which states that higher-level components of the application should not be dependent on lower-level ones. Both should only be aware of each other via abstractions. An example of such an abstraction would be the previously discussed IRepositories. These abstractions themselves should also not depend on specific details in their implementations. The two principles together are referred to as the Dependency Inversion Principle, which is entirely theoretical in nature. The specific mechanism to apply said principle is called the Inversion of Control, which in turn is made use of by Dependency Injection. The specific approach used in Controllers is called Constructor Injection since the dependencies to be injected are passed into the Constructor. Other

methods include Method Injection and Property Injection. After they have been configured in `Program.cs`, the Controllers are automatically run with their constructor. [40]

Specific endpoints can then be created by writing a new function for each endpoint. These functions then use the functionality provided by the Repositories to interact with the database. The following shows how to get all Options from a Question:

```csharp
1  [HttpGet("{id}/options")]
2  public async Task<ActionResult<Option>> GetAllOptions(Guid id)
3  {
4      try
5      {
6          var result = _mapper.Map<IEnumerable<OptionDto>>(await
   _optionRepository.GetOptionsByQuestionId(id));
7          return Ok(result);
8      }
9      catch (Exception ex)
10     {
11         return BadRequest(ex.Message);
12     }
13 }
```

Code Snippet 11: GetAllOptions Endpoint

The annotation defines the specific HTTP request method and also defines the route, which would be `/api/Question/ID/options` in this case. In the function itself, a try-catch block is used to automatically return an error message in case something goes wrong. The list of Options is then gathered by using the Option Repository and converting them to an OptionDto using AutoMapper. This list is then sent to the client with a 200 (success) status code.

3.2.2. Frontend

On the Client, this section includes two pages, which provide the tutor the ability to create and manage Quizzes. The routes for the pages are defined using React Router.

3.2.2.1. Main page

The first page is the main page, also called the index page. It is configured on the default route. Fully loaded, the page looks like this:
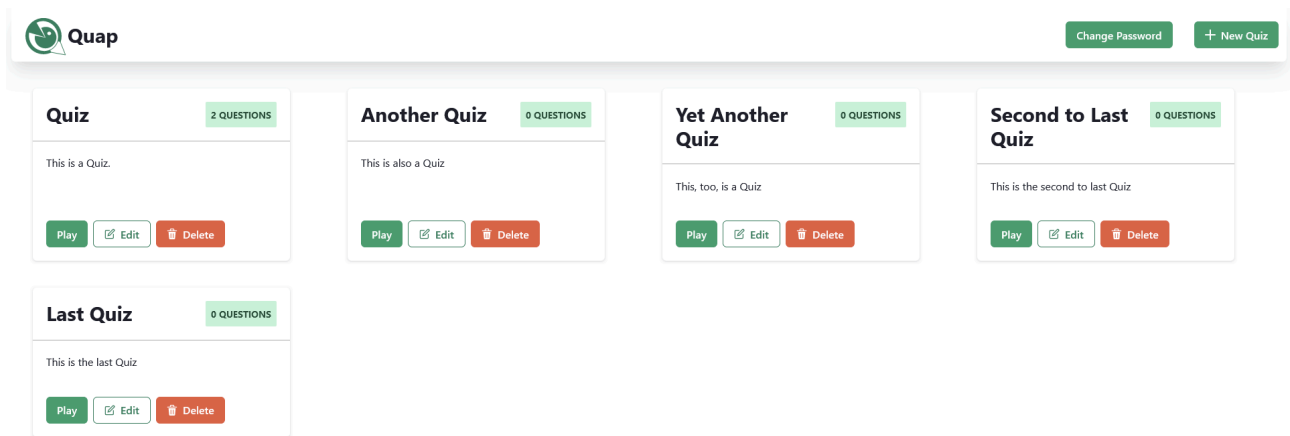
Figure 9: Main Page Screenshot

The first thing that happens when the page is loaded, is the loading of the Quizzes from the database. This is done using a ReactRouter Loader, which is declared on the route. Loaders and Actions are defined in their files. There, functions are created for each Loader/Action. The following Loader is used to fetch the Quizzes:

```
 1  export const quizLoader: LoaderFunction = async () => {
 2      const quizzes = await getQuizzes();
 3      if (quizzes == null) {
 4          throw new Response("", {
 5              status: 404,
 6              statusText: "Not Found",
 7          });
 8      }
 9      return quizzes;
10  }
```

Code Snippet 12: Quiz Loader Function

This function has the type LoaderFunction, which is provided by ReactRouter. The getQuizzes() method and all other methods containing fetch requests are contained in a separate file called `endpoints.ts`, where the JavaScript fetch API is used to make a HTTP Request to the Server. The URL constant and all Models are imported from another file:

```
1  type GetQuizzes = () => Promise<Quiz[]>
2  export const getQuizzes: GetQuizzes = async () => {
3      try {
4          const response = await fetch(`${URL}/Quiz`);
5          const data = await response.json();
6          return data as Quiz[];
7      } catch (e) {
8          throw e;
9      }
10 }
```

Code Snippet 13: getQuizzes() Method

The type has to be created in order to tell TypeScript which arguments of which types the function takes and what the return type of the function is.

The Loader can then be used in the Component of the page by using the `useDataLoader` hook. This hook is provided by ReactRouter. Once received, the Quizzes are passed as a Prop to another Component, where the Quizzes are laid out in Cards in a Grid layout. Each of these Cards in turn is its own Component. There, the name and description, as well as the number of Questions is laid out in an appealing way. Each Card also contains three Buttons to start a Game, edit a Quiz, and delete a Quiz. Two of these Buttons (Play, Delete) use Action Functions in order to execute API calls. The Edit button redirects to the **Edit Quiz Page**. The Start Game button is defined like so:

```
1  <Form method="post" action={`startgame/${id}`}>
2      <Button variant='solid' colorScheme='green' type="submit">
3          Play
4      </Button>
5  </Form>
```

Code Snippet 14: Start Game Button

`variant` and `colorScheme` are Chakra-Ui styling props. When pressing the other two buttons, the tutor has to verify their identity by entering their password. This, however, is explained later on.

When the tutor wants to create a new Quiz, they simply have to press the New Quiz Button in the top right corner of the page. When pressed, a dialog opens where they can enter a name and a description:
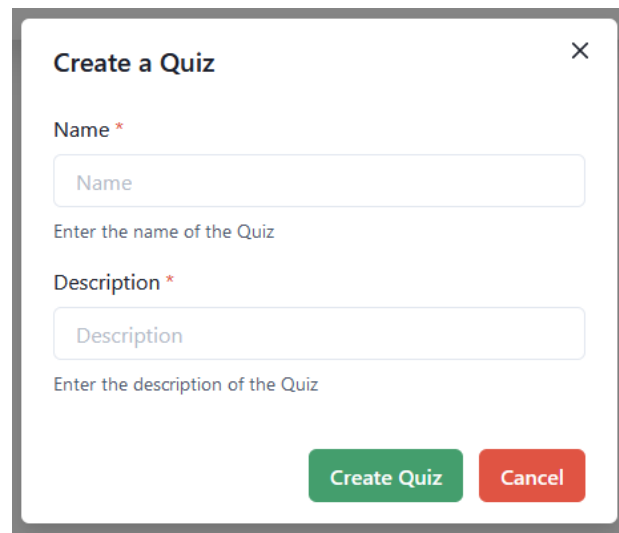
Figure 10: Create Quiz Dialog Screenshot

This is implemented using the Modal Component provided by Chakra-Ui. This Component can be controlled with the `useDisclosure` hook. This hook provides one boolean value and two functions, which are passed as props to the Modal (In this case the Modal is wrapped in its own Component so they are passed as a Prop of this Component). The value is true if the Modal is open and false if it is closed. The two functions are used to open and close the Modal respectively.

Inside the Modal, there are two input fields and a submit button, which first makes an API call to create a new Quiz and then navigates to the **Edit Quiz Page** of the created Quiz by using the `useNavigate` hook. This hook allows for redirection via code.

3.2.2.2. Edit Quiz Page

This page is used to edit the details of a Quiz. These details include the specific Questions and the various Options selectable on each Question. This page also gives the tutor the ability to rename a Quiz itself.

Figure 11: Edit Quiz Page

Sidebar:

This part of the page is configured as its own component, which is then included in the Edit Quiz Page. In the main Component of the Edit Quiz route, the questions are loaded via a ReactRouter Loader function:

```
 1  const loader: LoaderFunction = async ({ params }: QuestionsLoaderArgs) => {
 2      const quizId = params.quizId ?? "";
 3      const questions = await getQuestions(quizId);
 4      if (questions == null) {
 5          throw new Response("", {
 6              status: 404,
 7              statusText: "Not Found",
 8          });
 9      }
10      return questions;
11  };
```

Code Snippet 15: Question Loader function

This function first receives the Quiz ID through the URL parameters (The route of the Edit Quiz Page is /editQuiz/quizId). The first line of the function uses the null coalescing operator to check if the ID from the parameters is not undefined. The null coalescing operator first checks if the value before ?? is null/undefined. If it is not, the right value is used. If the right value is null, the left value is utilized instead. This seemingly pointless check is needed because even if there is theoretically no chance for the ID to be undefined in TypeScript, the compiler will still throw an error since it does not know for certain that the parameter is a string and not undefined. Next, a standard API call is made. The Questions array received through this Loader is then passed as a Prop to the Sidebar Component. There, the `array.map()` function is used to display a Link to the Question Details View for each Question. Other than the name of the Question, the Link also contains two Chakra-UI badge components displaying the Question type and the number of Options respectively. The button used to create a new Question employs a ReactRouter Action function. At the bottom of the Sidebar, there are two last buttons.

The second button is fairly simple as it merely contains a Link to the main route of the client. The first one, however, is more complicated as it opens a Modal where the name and the description of the Quiz can be altered. This Modal will not be shown here as it is almost entirely the same as the Create Quiz Dialog. It does, however, contain some interesting code: As the Modal Component needs to know the current Quiz details in order to populate the standard values of the text boxes, the name and description of the Quiz have to be loaded first. This seemed rather easy at first since it was assumed in the development process that instead of all Questions, the entire Quiz was loaded in the main Component of the route. This mistake was made due to the Edit Quiz details Modal being a late addition to the page made long after the rest. The problem was then solved by getting the Quiz ID from the Uniform Resource Locator (URL) Parameter. This was not very easy since the ReactRouter

documentation lacked any explanation on how to perform this task. It was eventually figured out through trial and error:

```
1    const { quizId } = useParams<'quizId'>();
2      useEffect(() => {
3          if (quizId) {
4              getQuizById(quizId)
5                  .catch(error => console.error(error))
6                  .then(data => {
7                      if (data) {
8                          setName(data.name);
9                          setDescription(data.description);
10                     }
11                 })
12         } else {
13             console.error("Could not load");
14         }
15     }, [])
```

Code Snippet 16: loading Quiz details in Modal

The Quiz ID can be loaded if the name of the URL field is specified. Afterwards, the React `useEffect` hook is used to perform the needed API. The useEffect Hook gives the developer the ability to perform side effects within function-based Components. Side effects are actions that need to happen after React has updated the DOM, such as fetching data, subscriptions, timers, or directly manipulating the DOM. By using useEffect, the developer essentially instructs React to execute specific code passed as a function in its first argument after the Component has been rerendered. The Hook also takes an array as its second argument. This array is used to specify when the code should be executed. This is useful since otherwise, the function would run every time the Component is rerendered, which happens a lot in React. If for example a state variable is passed as the second argument, the code is only executed if that specific state changes. If an empty array is passed as it is in the example given, the function is only called on the first render of the Component. [41]

Question Details View:

This part of the page encompasses all content to the right of the Sidebar. It can be displayed in two different versions depending on whether or not the Question is merely being viewed (display view) or actively edited (edit view). the views can be switched between by pressing the Edit button to start editing and by pressing the Save/Cancel buttons to mark the edit process as complete. Both views use the same underlying Components. The only difference is that in Edit mode, all buttons and text fields are enabled, which is achieved by declaring a boolean Prop on the Component and using said Prop to manage the state of input fields.

Both views are declared on their own subroutes of `/editquiz/quizId`. The edit view uses the route `/editquiz/quizId/editquestion/questionId` and the display view is available on `/editquiz/quizId/question/questionId`. As it is clear from the fact that the views have their individual URLs, they are separate pages. This means that pressing the Links in the Sidebar would, without additional configuration, redirect to another page containing the display view

of the desired Question. To display the contents of the route on the same page as the Side-bar, a ReactRouter Outlet Component needs to be included in the Component of the editquiz route. This tells ReactRouter to render the contents of subroutes as part of their parent. Once displayed, individual Questions are loaded using a ReactRouter Loader function. In the case when a new Question is created, the edit view is opened automatically. The Questions are always created without any data. The rest of this explanation will be focused on said edit view as it requires more attention due to the need to manage the State. This is not necessary on the display view since the user has no way of changing any values.

The edit view (and also the display view) contains a Component, which contains several input fields (disabled on display view) for entering data such as the Question name or the time limit and a dropdown list for the Question type. The following shows the input field for the time limit:

```
1  <Input type="number"
2  name="timeLimit"
3  value={timeLimit != 0 ? timeLimit : ""}
4  disabled={!isEditing}
5  onChange={(event) => setTimeLimit(+event.target.value)}
6  />
```

Code Snippet 17: Edit Question Time Limit Input Field

Its value is set to the time limit loaded with the Loader if the Question has a time limit specified. If not, the field is left empty. If the input changes as the tutor enters a number, said number is stored in State to be used in an API call to the server which performs a PATCH Operation to mutate the data of the Question. the plus in front of the expression `event.target.value` is used to convert a string to a number in JavaScript/TypeScript. The component also features an error message that is displayed above the input fields if there are empty fields. This also causes the save button to be disabled: **Image of error** This is done by declaring a boolean state, which is used to control the visibility of the alert and the ability to click the button. This is then updated by using the useEffect Hook to trigger when the state of the input field values changes.

```
1  useEffect(() => {
2      setEmptyError(name == "" || points == 0 || timeLimit == 0);
3  }, [name, points, timeLimit])
```

Code Snippet 18: Edit Question Error Check

Next to the Component for displaying the information about the Question there is another one showing all the Options. This Component uses the `array.map()` function to display one Component for each option, where the Option title can be changed (in the edit view), be marked as correct, or be deleted. As it would be extremely difficult to manage the state of the Options through that many layers of Components, the `useContext()` Hook is used on the main Component of the editquestion route. This hook provides an elegant way to share data across various components within an application, eliminating the need for "prop drilling" (-

Passing Props through multiple layers). A Context can be thought of as a container, which is accessible anywhere within the Component tree. When a Component needs a specific piece of data, useContext allows it to directly tap into that Context, retrieving the current value. In this example, the Hook is used to store all Option in an array of objects. For this, the TypeScript Index signature is used. This allows the developer to define a key for each entry in the array similar to a dictionary in other languages. The following shows the signature of the state used here: `{ [key: string]: OptionDto; }`. The key in this example is the ID of an Option. Existing Options are loaded into the State as follows:

```
1  const [options, setOptions] = useState<{ [key: string]: OptionDto }>(() => {
2  if (question.options.length > 0) {
3    return question.options.reduce((acc:{[key: string]:OptionDto;}, option) => {
4       acc[option.oId] = {
5           optionText: option.optionText,
6           isCorrect: option.isCorrect
7       };
8       return acc;
9    }, {});
10 } else return {}
11    });
```

Code Snippet 19: Load Options into Context State

This uses the `array.reduce()` function with two arguments, the dictionary where all options should be stored and the options themselves. The function is then run for every Option to add them to the dictionary. If no Options exist, the dictionary is left empty.

In order to use the Context State in other Components, they have to be wrapped in a Context Provider. This Provider takes the State and SetState functions as arguments. Inside those Components, the Context State can be accessed using the useContext Hook by passing the name of the Context [42]:

```
1  const { options: contextOptions } = useContext(OptionsContext);
```

Code Snippet 20: useContext Hook example

In an Option Component, the most complicated problem is to determine whether or not the parent Question is SingleChoice or MultipleChoice. If it is the former, additional code is needed to only allow one Option to be correct at one time. The checkbox itself is made with a Chakra-UI IconButton Component which is filled if the Option is marked correct. The following code shows how an Option is marked as correct:

```
1  setOptions((prevOptions: { [key: string]: OptionDto }) => {
2      const correctOptionId = Object.keys(prevOptions).find(
3      key => prevOptions[key].isCorrect);
4
5      if (correctOptionId
6          && correctOptionId !== id
7          && !prevOptions[id].isCorrect
8          && type == QType.SingleChoice) {
9              console.error(`Error Message`);
10             return prevOptions;
11     }
12     return { ...prevOptions, [id]: { ...prevOptions[id], isCorrect: !
   prevOptions[id].isCorrect } };
13 });
```

Code Snippet 21: check if there is already a correct Option

It first checks whether or not there already is a correct Option. Afterward, it checks if the ID of the current Option is not the ID of the Option that is marked correct and if the current Option is marked as not correct in the state. Finally, it checks the type of the Question. Said type is also provided to the Component with a Context. If all of these conditions are true, an error message is displayed. Otherwise, the state is updated. The Component containing all Options also contains Alerts for when an Option is empty and for the situation where there are not the required 2-6 Options. The Options from the Context State are then used in combination with the Question data to make an API call to the Server to save the changes to the Questions/Options.

3.3. Access Control with password

Another must-goal required the addition of functionality, which allows the tutor to restrict certain Actions from being executed without verification. This is necessary since the application is run locally and accessible through the tutor's IP address. Without verification, it would be possible for a student to access the page on their own device and modify data without the tutor's consent. The solution that was implemented for this problem is a verification system using a password. This means that the tutor is not required to provide any login data like a username and an Email-address to use the application.

3.3.1. Backend

In the Backend, the password is stored similarly to the Quiz data. This means that it is stored in a hashed form in the same database as the rest of the application data. This is done in order to reduce complexity. A password Model was also created containing an ID that is automatically set to "1" since there can only ever be one password and a string for the password hash.

This Model is then used in a dedicated authentication Controller, which provides endpoints for checking if a password is set, registering a password, verification of an entered password, and changing the password set. The following shows the verification endpoint:

```
 1   [HttpPost("verify")]
 2  public async Task<IActionResult> Verify([FromBody] RegisterVerifyPasswordDto
    model)
 3  {
 4      var storedPasswordHash = await _context.Passwords.FindAsync(1);
 5      if (storedPasswordHash == null) return NotFound("Password not set");
 6
 7      if (BCrypt.Net.BCrypt.Verify(model.Password,
    storedPasswordHash.PasswordHash))
 8      {
 9          return Ok("Password verified successfully");
10      }
11      else
12      {
13          return Unauthorized("Incorrect Password");
14      }
15  }
```

Code Snippet 22: Verify Password endpoint

This code first retrieves the password hash stored in the database. Then, it uses Bcrypt.NET to check whether or not the password passed in the Body and the stored one corresponding to the hash in the database are the same. If they are, a HTTP Response with the status code 200 (success) is sent. If they are not, the Server sends a Response with the status code 401 (unauthorized). The endpoint to change the password is similar with the only difference being that in addition to sending a 200 status code, the Server also updates the password hash on the database. The other two endpoints are also structured very similarly. Thus, they will be omitted in this documentation.

3.3.2. Frontend

3.4. Game Flow

3.4.1. Models/DTOs

3.4.2. Repositories

3.4.3. Controller

3.4.4. Initialize Game

3.4.5. Join Game

3.4.6. Start Game

3.4.7. Gameplay

3.4.8. End Game

4. Conclusion

4.1. Results

4.1.1. Backend

4.1.2. Frontend

4.2. Potential future additions

4.3. Reflection on the project

Bibliography

[1]   M. Hall, G. P. Zachery, and Karl Montevirgen, "Microsoft Corporation". Accessed: Feb. 13, 2024. [Online]. Available: https://www.britannica.com/topic/Microsoft-Corporation

[2]   anandmeg, ghogen, and GitHubber17, "What is Visual Studio?". Accessed: Feb. 13, 2024. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022

[3]   Microsoft, "Why did we build Visual Studio Code?". Accessed: Feb. 13, 2024. [Online]. Available: https://code.visualstudio.com/docs/editor/whyvscode

[4]   Microsoft, "Emmet". Accessed: Feb. 13, 2024. [Online]. Available: https://code.visualstudio.com/docs/editor/emmet

[5]   P. Chandrayan, "What is Docker Desktop: Getting Started". Accessed: Feb. 13, 2024. [Online]. Available: https://www.knowledgehut.com/blog/devops/docker-desktop

[6]   W. L. Hosch, Y. Chauhan, and G. Lotha, "Firefox". Accessed: Feb. 15, 2024. [Online]. Available: https://www.britannica.com/technology/Firefox-Web-browser

[7]   D. R. Eliis, "What is Swagger? A Beginner's Guide". Accessed: Feb. 15, 2024. [Online]. Available: https://blog.hubspot.com/website/what-is-swagger

[8]   L. Mädje, "Typst - A Programmable Markup Language for Typesetting", 2022. [Online]. Available: https://www.user.tu-berlin.de/laurmaedje/programmable-markup-language-for-typesetting.pdf

[9]   Typst, "Typst Home Page". Accessed: Feb. 15, 2024. [Online]. Available: https://typst.app/

[10]   JabRef, "JabRef Documentation". Accessed: Feb. 15, 2024. [Online]. Available: https://docs.jabref.org/

[11]   P. K. Ponuthorai, *Version control with Git*, Third edition. Sebastopol, CA: O'Reilly Media, Inc, 2022.

[12]   GitHub, "GitHub Features". Accessed: Feb. 15, 2024. [Online]. Available: https://github.com/features

[13]   GeeksforGeeks, "Introduction to C#". Accessed: Feb. 15, 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-c-sharp/

[14]   gewarren, zipperer, jivanf, richlander, and IEvangelist, "Introduction to NET". Accessed: Feb. 15, 2024. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/introduction#

[15]   P. Rout, "Introduction to ASP.NET Core Framework". Accessed: Feb. 16, 2024. [Online]. Available: https://dotnettutorials.net/lesson/introduction-to-asp-net-core/

[16]   I. V. Abba, "What is an ORM – The Meaning of Object Relational Mapping Database Tools". Accessed: Feb. 16, 2024. [Online]. Available: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/

[17] F. Nguyen, "How to use AutoMapper in C# Net Core?". Accessed: Feb. 16, 2024. [Online]. Available: https://medium.com/knowledge-pills/how-to-use-automapper-in-c-6f949402be05

[18] J. M. Aguilar, *SignalR programming in Microsoft ASP.NET*, First Edition. in Professional. Redmond, Washington: Microsoft Press, a division of Microsoft Corporation, 2014.

[19] CodeMaze, "How to Secure Passwords with BCrypt.NET". Accessed: Feb. 16, 2024. [Online]. Available: https://code-maze.com/dotnet-secure-passwords-bcrypt/

[20] D. Arias, "Hashing in Action: Understanding bcrypt". Accessed: Feb. 16, 2024. [Online]. Available: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

[21] R. Burdiuzha, "What is PostgreSQL?". Accessed: Feb. 16, 2024. [Online]. Available: https://gartsolutions.medium.com/what-is-postgresql-2bbd8e4ada6a

[22] M. H. Massé, *REST API design rulebook*, Second Edition. Sebastopol: O'Reilly, 2012.

[23] M. Fowler, "Data Transfer Object". Accessed: Feb. 16, 2024. [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html

[24] M. Murugan, "Repository Pattern in ASP.NET Core – Ultimate Guide". Accessed: Feb. 19, 2024. [Online]. Available: https://codewithmukesh.com/blog/repository-pattern-in-aspnet-core/#Whats_a_Repository_Pattern

[25] MDN, "What is JavaScript?". Accessed: Feb. 18, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[26] Kinsta, "What Is TypeScript? A Comprehensive Guide". Accessed: Feb. 18, 2024. [Online]. Available: https://kinsta.com/knowledgebase/what-is-typescript/

[27] D. Herbert, "What is React.js? Uses, Examples, & More". Accessed: Feb. 18, 2024. [Online]. Available: https://blog.hubspot.com/website/react-js

[28] T. Davidson, "What is Vite, And Why Is It Awesome?". Accessed: Feb. 18, 2024. [Online]. Available: https://cleancommit.io/blog/what-is-vite/

[29] O. Alaiya, "Demystifying Chakra UI". Accessed: Feb. 19, 2024. [Online]. Available: https://odafe.hashnode.dev/demystifying-chakra-ui#heading-so-what-is-chakra-ui

[30] Remix, "React Router - Feature Overview". Accessed: Feb. 19, 2024. [Online]. Available: https://reactrouter.com/en/main/start/overview

[31] L. Kong, "React component guide: Class vs functional". Accessed: Feb. 19, 2024. [Online]. Available: https://www.educative.io/blog/react-component-class-vs-functional#class-component

[32] D. Lundin, Accessed: Feb. 21, 2024. [Online]. Available: https://de.wikipedia.org/wiki/PostgreSQL?useskin=vector#/media/Datei:Postgresql_elephant.svg

[33] Automapper, Accessed: Feb. 21, 2024. [Online]. Available: https://avatars.githubusercontent.com/u/890883?s=280&v=4

[34] Microsoft, Accessed: Feb. 21, 2024. [Online]. Available: https://ably.com/topic/signalr-deep-dive

[35] Bcrypt.NET, Accessed: Feb. 21, 2024. [Online]. Available: https://avatars.githubuserc
    ontent.com/u/22194067?s=200&v=4

[36] C. McKenzie, "Dockerfile vs docker-compose: What's the difference?". Accessed: Feb.
    22, 2024. [Online]. Available: https://www.theserverside.com/blog/Coffee-Talk-Java-
    News-Stories-and-Opinions/Dockerfile-vs-docker-compose-Whats-the-difference

[37] Remix, Accessed: Feb. 22, 2024. [Online]. Available: https://reactrouter.com/brand

[38] H. Manwani, "How to use Enums in C#". Accessed: Feb. 22, 2024. [Online]. Available:
    https://www.loginradius.com/blog/engineering/enum-csharp/

[39] T. Ugurlu, "Asynchronous Database Calls With Task-based Asynchronous Program-
    ming Model (TAP) in ASP.NET MVC". Accessed: Feb. 25, 2024. [Online]. Avail-
    able: https://www.tugberkugurlu.com/archive/asynchronous-database-calls-with-task-
    based-asynchronous-programming-model-tap-in-asp-net-mvc-4

[40] A. Chiarelli, "Understanding Dependency Injection in .NET Core". Accessed: Feb. 25,
    2024. [Online]. Available: https://auth0.com/blog/dependency-injection-in-dotnet-core/

[41] S. Weber, "A complete guide to the useEffect React Hook". Accessed: Feb. 27, 2024.
    [Online]. Available: https://blog.logrocket.com/useeffect-react-hook-complete-guide/

[42] D. Ceddia, "React useContext Hook Tutorial (with Examples)". Accessed: Feb. 28,
    2024. [Online]. Available: https://daveceddia.com/usecontext-hook/

Code Listings

Image Listings