



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

Abstract

The last 15 years have seen giant leaps in many fields of the computer and IT industry, not only in terms of performance but also when it comes to cost. This expansion has enabled many schools to provide staff and students with their devices, may it be a laptop or a tablet. Such a development has led to countless possibilities for improving education quality. Teachers worldwide have seized this opportunity and created interactive and engaging lessons for their students. Some countries and larger schools have also gone the extra way and have commissioned custom, often costly, software meant to enhance the performance of attendees even further.

However, this has also resulted in the widening of the difference in education quality nationally and internationally since many less-equipped institutions are reliant on already publicly available solutions. These are often developed by large multinational for-profit corporations. For that reason, many of them come with artificial limitations and privacy violations to push people into subscribing to often expensive monthly subscription services. This project aims to create a simple, free alternative to empower teachers worldwide.

Short version/task overview

Topic	Development of a web-based Quiz application	
Task (short)	The goal is to plan and implement a web-based quiz application for classrooms. The application will be run on an instructor's computer, which students can use to connect. A quiz is then run on the PC, which Students can take part in by reading the questions on a projector screen and then choosing the correct answer on their device.	
Candidates		Supervisors
Florian Piberger		Dipl.-Ing. (FH) Dietmar Winkler
		Prof. Dipl.-Ing. (FH) Markus Falkensteiner
External partners		
none		
Budget: none		
Planned Usage of the product: The source code will be openly viewable on GitHub. Commercial use of the Code or the product is not intended now or at any time in the future.		

Erklärung

Die unterfertigten Kandidaten haben gemäß § 34 (3) SchUG in Verbindung mit § 22 (1) Zi. 3 lit. b der Verordnung über die abschließenden Prüfungen in den berufsbildenden mittleren und höheren Schulen, BGBl. II Nr. 70 vom 24.02.2000 (Prüfungsordnung BMHS), die Ausarbeitung einer Diplomarbeit mit der umseitig angeführten Aufgabenstellung gewählt.

Die Kandidaten / Kandidatinnen nehmen zur Kenntnis, dass die Diplomarbeit in eigenständiger Weise und außerhalb des Unterrichtes zu bearbeiten und anzufertigen ist, wobei Ergebnisse des Unterrichtes mit einbezogen werden können.

Die Abgabe der vollständigen Diplomarbeit hat bis spätestens

05.04.2024

beim zuständigen Betreuer zu erfolgen.

Die Kandidaten nehmen weiters zur Kenntnis, dass gemäß § 9 (6) der Prüfungsordnung BMHS nur der Schulleiter bis spätestens Ende des vorletzten Semesters den Abbruch einer Diplomarbeit anordnen kann, wenn diese aus nicht beim Prüfungskandidaten (bei den Prüfungskandidaten) gelegenen Gründen nicht fertiggestellt werden kann.

Kandidaten / Kandidatinnen	Unterschrift
Florian Piberger	

Contents

1. List of abbreviations	1
2. Planning	2
2.1. Task definition	2
2.1.1. Initial idea	2
2.1.2. Status quo	2
2.2. Goals	2
2.2.1. Must-goals	3
2.2.2. Should-goals	3
2.3. Project organization	4
2.3.1. Roles	4
2.3.2. Task distribution	4
2.3.3. Milestones	4
3. Theoretical Foundations	6
3.1. Software used	6
3.1.1. Development	6
3.1.2. Other	7
3.2. Backend	11
3.2.1. Programming Languages	11
3.2.2. Frameworks/Libraries	11
3.2.3. PostgreSQL	14
3.2.4. Concepts	15
3.3. Frontend	17
3.3.1. Programming Languages	17
3.3.2. Frameworks/Libraries	18
3.3.3. Concepts	21
4. Implementation	22
4.1. Architecture	22
4.1.1. Backend	23
4.1.2. Frontend	24
4.2. Quiz Management	25
4.2.1. Backend	25
4.2.2. Frontend	33
4.3. Access Control with password	41
4.3.1. Backend	41
4.3.2. Frontend	42
4.4. Game Flow	44
4.4.1. Models/DTOs	45
4.4.2. Controller/Repositories	46

4.4.3. Initialize/Join Game	47
4.4.4. Gameplay	49
4.4.5. End Game	54
5. Conclusion	55
5.1. Results	55
5.1.1. Backend	55
5.1.2. Frontend	56
5.2. Potential future additions	56
Bibliography	57
Code Listings	59
Image Listings	60

1. List of abbreviations

API:	Application Programming Interface
CI/CD:	Continuous Integration/Continuous Deployment
CLI:	Command 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
VS Code:	Visual Studio Code
WYSIWYG:	What you see is what you get

2. Planning

This first section will cover all details relating to the planning of this project. It aims to give an extensive overview of the thought processes and ideas behind this diploma thesis.

2.1. Task definition

The first step when starting work on this thesis was to define the nature of the application precisely and to think about what was needed to achieve a completed state.

2.1.1. Initial idea

Creating a fun and exciting environment is essential for any instructor/teacher. A popular way to create engaging lessons for students is to utilize quizzes, which not only consolidate previously taught material but also create an opportunity to assess students' abilities without having to rely on often very unpopular and tedious exams/tests. That is exactly what I was pondering when I came to the realization that an application providing the feature described above would be an ideal choice for a diploma thesis. So, I began planning, and now I want to turn my plans into reality and hopefully create an application that will help many teachers make better lessons.

2.1.2. Status quo

Another reason for choosing this project is that existing solutions are often inadequate. They usually come with limitations, such as a maximum number of concurrent players, and irritating upselling, such as paid subscriptions and predefined lesson plans filled with irrelevant information and advertisements. They also collect and store user's data. A major problem with creating a similar application is that traditional solutions are often hosted on the Web and thus cost a lot to run. However, this project is not a problem, as it is run directly on the instructor's PC and, therefore, only accessible on a local network, which is a nonissue for classroom situations.

2.2. Goals

This project aims to create a functional prototype, which, while not 100% flawless, is still usable in a classroom setting.

Minimalism

The goal behind the application is not to create a full-fledged teaching suite, nor should it apply to as many distinct groups as possible. This product is mainly targeted toward educators, who desire a simple no-frills way to assist them in their class or educators, who are worried about the influence of large corporations on the field of education.

Simplicity & Ease of Use

Simplicity is another core principle behind the project. The goal is to create a user interface that, while not using the most up-to-date eye-catching design trends, is able to be used without having to spend more than a couple of minutes trying to figure out how to, for example create a quiz.

2.2.1. Must-goals

These goals must be completed by the final presentation in order to be able to call the project finished:

- **Ability to play quizzes**

The application must provide a simple interface for playing quizzes. The quizzes must be laid out on a central page where a game can be started. When playing a game, the current question is displayed on a projector screen, and a separate interface is provided for choosing the desired answer. After completing the quiz, The scoreboard must be displayed on the projector screen.

- **Ability to create/edit/delete quizzes**

The central page provides functionality for managing quizzes. There must also be a separate interface for creating quizzes. The quizzes must be stored on a local database.

- **Administrator**

The program must provide a way for the instructor to create an administrator account in order to prevent students from accessing the central page to alter created quizzes.

- **Mobile-friendly interface**

The student interface must be accessible from mobile devices, meaning that contents should be adjusted according to screen size.

- **Question properties**

The instructor must be able to customize questions in several ways: single choice or multiple choice, number of points on correct completion, time limit, and number of answers. (min. 2, max. 6)

2.2.2. Should-goals

These goals may be completed if all must-goals have already been completed before the hand-in date. Thus, they are considered optional:

- **Packaging/Containerization**

The application may be easily installed/run. This means it can be started with a simple click or a few CMD commands.

- **Exporting/Importing of quizzes**

The instructor may be able to import/export quizzes as text files that can be, when imported into another instance of the program running on an entirely different computer, fully converted back into a valid quiz, which can immediately be played.

- **Animations**

The application may contain animations in select parts of the User Interface (UI) (player view, question overview), which may help boost student engagement.

- **More question types**

The application may contain more than the two previously described (single choice, mul-

multiple choice) question types. Possible additions include text field entry (example: Math question), slider, sorting entries, ...

2.3. Project organization

This section seeks to inform about the project's organization and define specific tasks and milestones.

2.3.1. Roles

Main Supervisor	Dietmar Winkler
Secondary Supervisor	Markus Falkensteiner

Project leader Florian Piberger

Project team Florian Piberger

2.3.2. Task distribution

Florian Piberger

- Software Design
 - Backend Components
 - Frontend Components
 - Database Schema
 - Choosing used technologies
- Design Frontend interface
- Coding
 - Implementation of Backend Application Programming Interface (APIs)
 - Implementation of Frontend Components
 - Implementation of Real-time communication between Frontend and Backend
 - Creating Database Models
- Testing
- Documentation

2.3.3. Milestones

Until late 09/2023 - early 10/2023:

- Complete Scope Statement
- Start learning unknown technologies
- Enter details in the diploma thesis database

Until end 10/2023:

- Decide on major technologies
- Database schema for quizzes
- Initial work on the Backend
 - Architecture
 - 30% complete

Until end 11/2023:

- Backend - essential real-time communication
- Frontend - initial work

Until end 12/2023:

- Frontend - Design
- Backend - first prototype

Until end 01/2024:

- Documentation - research and start writing (40-50% complete)
- Frontend - first prototype

Until end 02/2024:

- Frontend/Backend - last polish and testing
- Documentation - 80-85% complete

Until end 03/2024:

- Documentation – complete
- Backend – complete
- Frontend – complete
- Start work on video and poster

3. 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 a reference point for the rest of the documentation. An explanation of what was achieved can be found in the Implementation section.

3.1. 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.

3.1.1. 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 considerable influence in the decision process.

Visual Studio

Visual Studio is an Integrated Development Environment (IDE) developed by the Redmond, Washington-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 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: Open source 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.

Visual Studio Code

Visual Studio Code (VS Code) is a code editor developed by Microsoft with an open-source 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 potent tool thanks to its expansive and well-supported extension eco-system. It is thanks to these extensions that VS Code has become very popular in the field of web development, as many developers depend on VS Code extensions when working with popular frameworks and libraries. When some commonly utilized technologies 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 Code was primarily used when developing the application's Frontend.

Docker Desktop/Docker

Docker Desktop is a Graphical User Interface (GUI) application for Windows and macOS that is used to manage Docker Containers. It is available in both an open-source community edition and a paid commercial version. It allows the developer to create and maintain Docker Images and Kubernetes Clusters.

Docker itself is a system that gives developers the ability to deploy software quickly. It might be easier to imagine 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 and 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. Containers are isolated and self-contained. It is possible to run multiple instances of them simultaneously. 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 helpful 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]

3.1.2. Other

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

Mozilla Firefox

Mozilla Firefox is an open-source 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 significant bit of popularity by the time it was made open-source. Firefox, in turn, became a prevalent 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. Depending on the region, Firefox now hovers around 3 to 10% market share.

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]

Swagger UI

Swagger UI is an interface built upon the OpenAPI Specification for documenting an 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 initially known as the Swagger Specification. Swagger itself includes a set of tools that 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]

Typst

Typst is a newly developed Free and Open Source (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 can 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 must be marked by writing a “#” in front of them. [8] Typst provides both a Command 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]

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 requiring 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.

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 famous FOSS OS Linux in 2005 to be an entirely 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 unique 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 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:** Commits are snapshots of a repository’s current state of selected files. 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 free 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 storing a repository solely on a local computer is possible, 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 that 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 allows users to host simple Websites and track issues on repositories in addition to basic project management tools like a Kanban Board and CI/CD with GitHub Actions. [12]

3.2. Backend

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

3.2.1. Programming Languages

The following section outlines which Programming Languages were used in the Backend.

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 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 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. [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]

3.2.2. Frameworks/Libraries

In addition to C# as the primary programming language of the Backend, several frameworks, libraries, and packages were also used in cases where additional features were required.

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 on the older Asp.NET Framework, which, as 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, minor. 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 data transfer across the application.

On the other hand, web APIs, on the other hand, merely provides 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.

3.2.2.1.1. 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 like 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]

AutoMapper

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

```
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 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 match the corresponding fields automatically. Should this fail, the developer will be able to create maps manually 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]

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 specific use cases, but it is too limited for others, such as chats or games. Instead, these applications may benefit from using 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 challenging for developers to use and implement. The developer must also maintain the connection and manage clients, which adds even more difficulty.

To solve these problems, Microsoft released the SignalR library for .NET and .NET Core. It enables developers to work with WebSockets efficiently. At the core of SignalR are so-called Hubs. There, functions can be created that correspond to a particular task performed by the application. Business logic is defined within them, 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 easily combined with Microsoft Identity to associate specific clients with logged-in users. For 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 call the previously discussed functions on the server by invoking them with their name and, if needed, adding data required 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, SignalR also supports real-time communication technologies other than WebSockets. They are, however, excluded from this documentation since they are not used. An example of one of them would be Server Side Events. [18]

Bcrypt.NET

Bcrypt.NET is a .NET library that implements 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, created by American cryptographer Bruce Schneider. Although newer ciphers and algorithms exist, Bcrypt is still considered secure and widely used, thanks to its availability on many platforms. Blowfish even has a successor called Twofish.

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 significant 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 in the following way: 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 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]

3.2.3. PostgreSQL

The database used for this project is PostgreSQL, a FOSS relational database. Similarly 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. It supports concepts like custom data types and inheritance typically found in Object Oriented Programming (OOP). It also allows multiple transactions to be run simultaneously by assigning a snapshot of the database to each one and for reusable queries. [21]

This project uses some of these features since it uses Enums to store specific 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.

3.2.4. Concepts

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

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 a well-designed APIs is essential to the modern web, where developers not only work with their APIs but also with several others hosted on other servers on the web. Therefore, all APIs must be 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 minor 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 intermediaries, such as proxies between the client and the server, as long as they are not visible to the client. They may, however, still block specific 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 temporarily transfer executable code to the client. The server also has to ensure that the client can execute said code. An example would be JavaScript files. [22]

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 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 whether or not an Option is correct or not is omitted to prevent students from cheating.

Repository Pattern

The Repository pattern in ASP.NET is a design pattern where so-called Repositories are created, which 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 Controller 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 data access details, 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]

3.3. 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.

3.3.1. Programming Languages

Following Programming Languages were used in the Frontend.

JavaScript

When working on websites, developers have a limited number of technologies they can use, as most Web Browsers only support a small 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 as a Programming Language, although it is often referred to as a scripting language. 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, for which it was invented. 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 element's CSS properties 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]

TypeScript

As valuable as JavaScript is to Web Development, it is also widely considered flawed. One significant aspect often criticized is its lack of a proper typing system. JavaScript uses dynamic typing, meaning a developer does not have to specify a data type when creating a variable. This alone is not the main reason 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` checks if the correct value is the same as the left one and 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 consider 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 easy to make mistakes when writing JavaScript, which is often quite tricky to debug. This also makes JavaScript 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's main addition is static typing, meaning it makes it possible to define the data type of a variable. However, it is not strictly necessary because TypeScript automatically tries to figure out the type of a variable on creation if it is initialized with a value. It also offers advanced type-checking capabilities and an improved class system. TypeScript also allows for creating 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 similar to an interface, although it cannot directly inherit other types. It also supports many advanced typing features not found on interfaces. [26]

This project uses TypeScript as opposed to regular JavaScript.

3.3.2. 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.

React

When creating larger web applications, using the same elements more than once is widespread. 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 might not seem like much work for a small element such as this. However, as 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.

However, many libraries and frameworks 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 allows the developer to write HTML markup directly in the JavaScript/TypeScript file using JavaScript Extensible Markup Language (XML) (JSX). JSX also allows for seamless code injection into the markup by wrapping all code to be executed in “{}”. [27]

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 be compiled to technologies understood by Browsers. This step also requires including or bundling in all necessary dependencies. The traditional tool of choice for this task was WebPack, but in recent years, another option, 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, which increases the efficiency of the bundling process because code no longer has to be bundled into a single file, greatly reducing complexity. This also makes it possible for Vite to update only the modules for certain parts of the application since code is bundled into multiple files, making it particularly useful for development, as it decreases reload times when making small changes to code. Vite can also determine if certain parts of code are essential to the application and then remove said code from the bundle, reducing its file size. [28]

Chakra UI

When working with React or another Component-based framework/library, it soon becomes evident that many different Components appear 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 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 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 shown in the example, it is straightforward 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. Afterwards, Chakra UI can be used anywhere in the application. [29]

React Router

React alone is often insufficient when working on more complex projects 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 must then be wrapped in a Router Component 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 only one action

is declared 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.

3.3.3. Concepts

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

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, as they have been available since React was first released. They are 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 it is possible to store data specific to the Component instance and 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 the less popular option for a long time until Hooks were introduced. Hooks add the previously missing features to Functional Components. The most used Hook by far 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]

4. 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 to provide an even better look at the finished product.

4.1. 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:

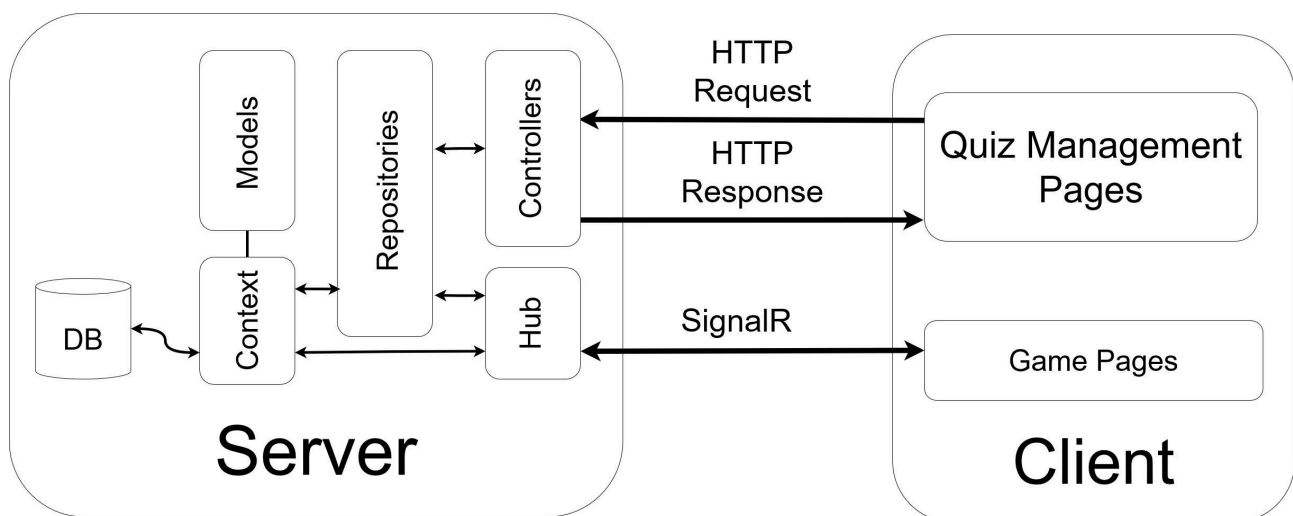


Figure 1: General Architecture Overview

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.

4.1.1. Backend

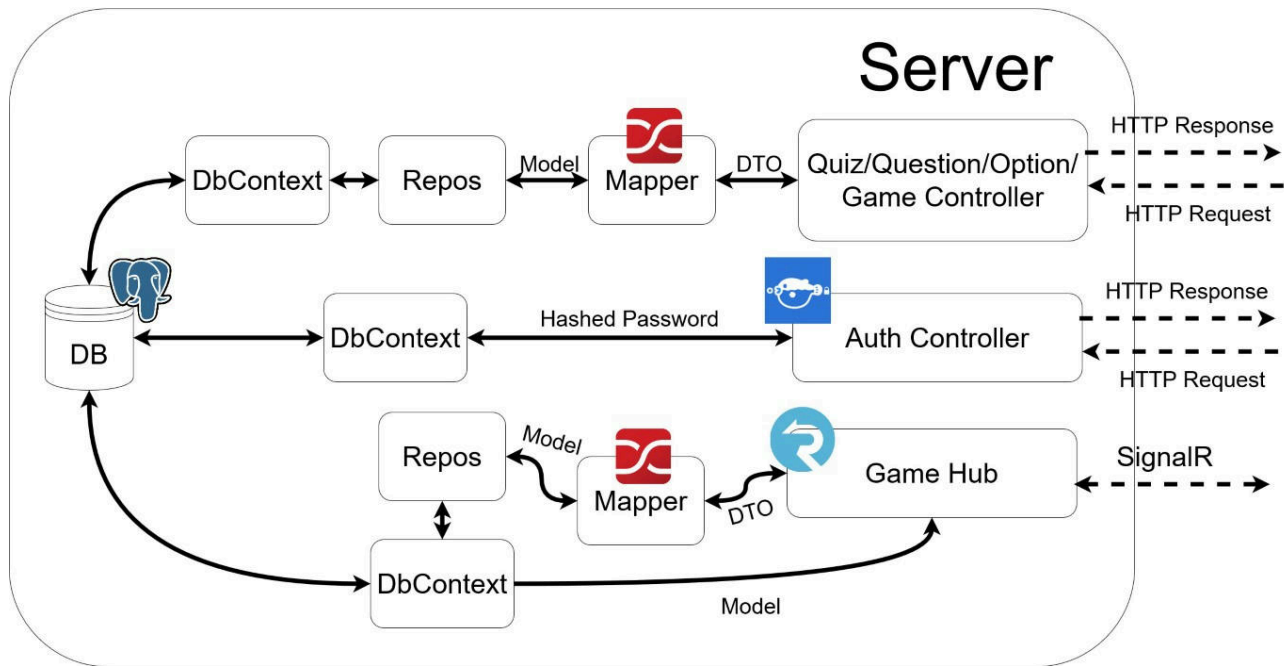


Figure 2: Backend Architecture Design

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 to manage 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.yml` 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 above, where an Image that has already been created providing the latest version of PostgreSQL is specified. The only thing left to configure is the username and password for the database. They are still hard-coded here, 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 details 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.

4.1.2. Frontend

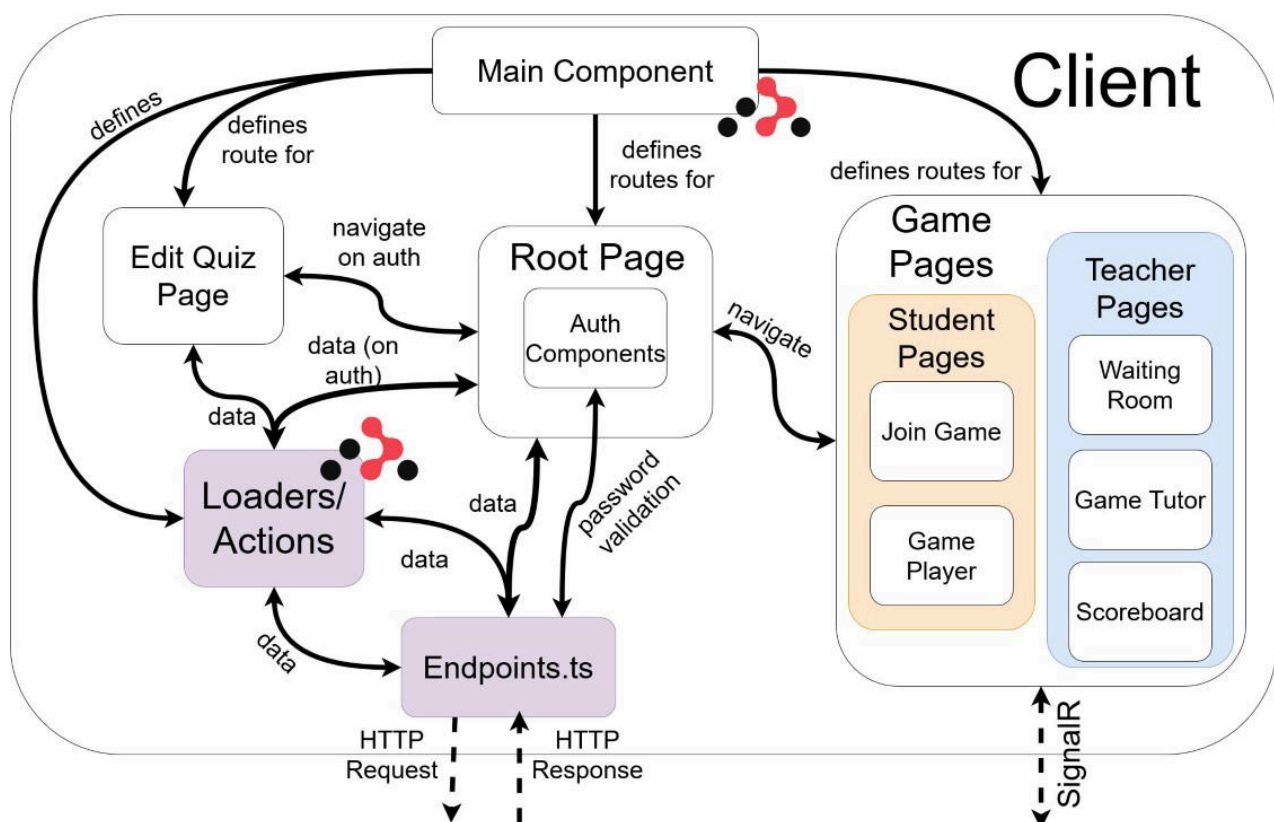


Figure 7: Frontend Architecture Design

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 differ in 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 com-

municate with the Server. The other section is the Quiz Management interfaces, which are entirely used by the tutor. Here, they can create new Quizzes, edit existing ones, or delete Quizzes that 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.

4.2. Quiz Management

A significant 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.

4.2.1. Backend

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

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 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:

```
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; } = DateTime.Now.ToUniversalTime();
9     public Guid QuestionId { get; set; }
10    public Question Question { get; set; }
11 }
```

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. Doing so is not strictly necessary when a developer only wants to use a Model in another Model as part of a composition dependency. It is, however, necessary for Entity Framework Core to correctly 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.). On the other hand, not including any reference to the containing Model is advantageous when Model data is referenced elsewhere or needed again later. This observation was also made use of when storing Games.

As it was clear from the example given of the Option Model, Models tend not to be very complex as they 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; } = DateTime.Now.ToUniversalTime();
13
14    public Guid QuizId { get; set; }
15    public Quiz Quiz { get; set; }
16 }
```

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 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 crucial to understand 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; } = DateTime.Now.ToUniversalTime();
11 }
```

Code Snippet 5: Quiz Model

Each Model is also mapped to several DTOs to transfer data more efficiently. The DTOs contain the same fields as the Models, although some 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 DbSet for each Model. It also allows for data querying 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:

```
1 public class QuizManagementDbContext : DbContext
2 {
3     public QuizManagementDbContext(DbContextOptions<QuizManagementDbContext>
4         options) : base(options) { }
5     public DbSet <Question> Questions { get; set; }
6
7     protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
8     {
9         optionsBuilder.UseExceptionProcessor();
10    }
11    protected override void OnModelCreating(ModelBuilder builder)
12    => builder.HasPostgresEnum<QType>();
13 }
```

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 must be manually assigned to the database.

This DbContext must be configured in the main Program.cs file. The Connection String, which includes all the 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.

Repositories

The next step after creating the needed Models is to declare functions that use said Models and their corresponding database tables to perform specific 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:

```
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 more apparent 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 must be included in other parts of the application using Dependency Injection, which is helpful 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 explained further 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 Quizzes and the Questions contained within Quizzes 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 asynchronous, the function return type must be a `Task`. It represents a unit of work that is performed asynchronously. 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 be completed 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 correctly, the Repositories must be registered in the main `Program.cs` file.

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.

When creating an API Controller, the first step 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, which 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 that 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 be independent of specific details in their implementations. The two principles are called the Dependency Inversion Principle, which is entirely theoretical. The specific mechanism to apply said principle is called the Inversion of Control, which is used 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 configuring them 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:

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

Code Snippet 11: GetAllOptions Endpoint

The annotation defines the specific HTTP request method and 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.

4.2.2. Frontend

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

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:

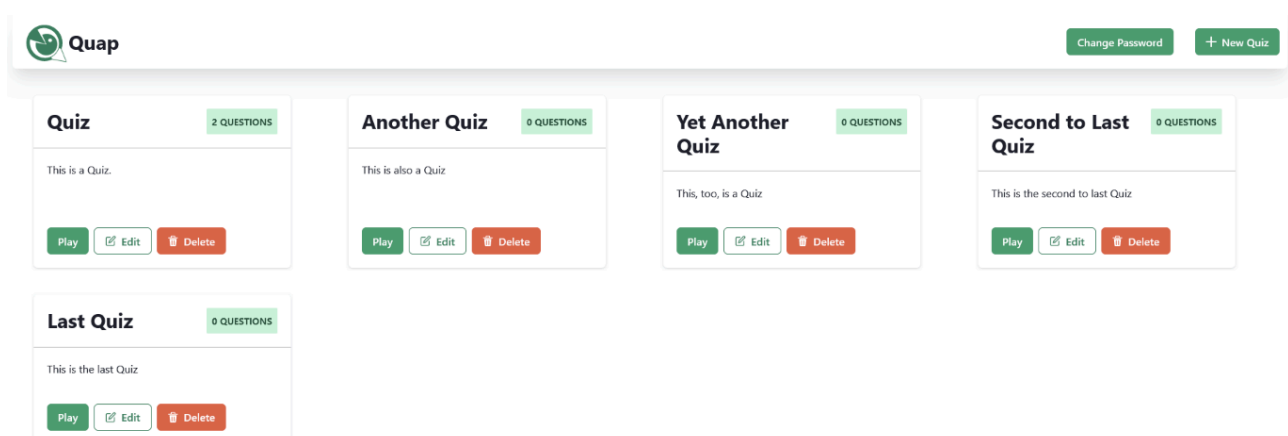


Figure 9: Main Page Screenshot

The first thing that happens when the page is loaded is loading the Quizzes from the database. This is done using a ReactRouter loader 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 Uniform Resource Locator (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 to tell TypeScript which arguments of which types the function takes and what the function's return type is.

The loader can then be used in the Component of the page 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 with 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, are laid out appealingly. 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 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:

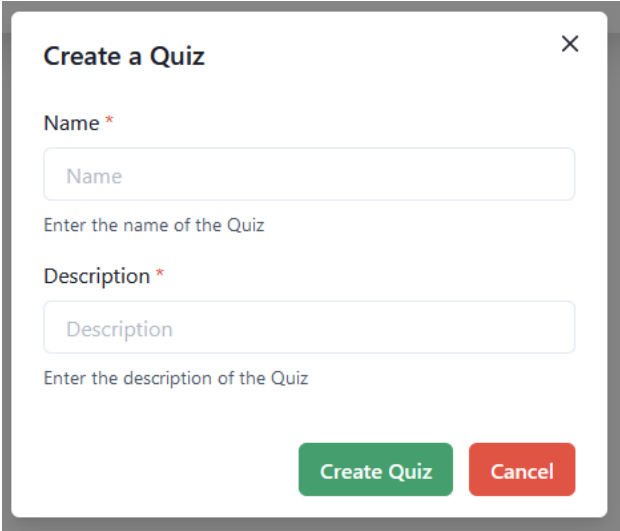


Figure 10: Create Quiz Dialog Screenshot

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.

Edit Quiz Page

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

The screenshot shows a web interface for editing a quiz. On the left is a sidebar titled 'Quap' containing a list of quiz questions. Each question entry shows the question text, a difficulty level (e.g., 'SC'), the number of options (e.g., '3 OPTIONS'), and a trash icon. At the bottom of the sidebar are buttons for '+ New', 'Edit Details', and 'Close'. The main area is titled 'Question' and contains a form for editing a specific question. The form includes fields for 'Question Name' (with a preview of 'How much is 2+2'), 'Question Type' (a dropdown menu currently showing 'SingleChoice'), 'Time Limit' (a field with '50'), and 'Points' (a field with '5'). There are also checkboxes for 'Enter your Question' and 'Enter the Time Limit for your question'. On the right side of the form, there are three input fields for multiple choice options: '4', '2', and '24', each with a checkmark icon. At the bottom of the main form is an 'Edit' button.

Figure 11: Edit Quiz Page

The page is split up into two sections:

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 editquiz 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 correct 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 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 displays a Link to the Question Details View for each Question. Besides the Question's name, the Link contains two Chakra UI badge Components displaying the Question type and the number of Options. A 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 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 to populate the standard values of the text boxes, the Quiz's name and description must 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 because the Edit Quiz details Modal was a late addition to the page made long after the rest. The problem was solved by getting the Quiz ID from the URL Parameter, which was difficult 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 allows the developer 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. Using `useEffect`, the developer 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 often 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 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 pressing the "Save"/"Cancel" buttons to mark the edit process as complete. Both views use the same underlying Components. The only difference is that all buttons and text fields are enabled in Edit mode, achieved by declaring a boolean prop on the Component and using said prop to manage the state of input fields.

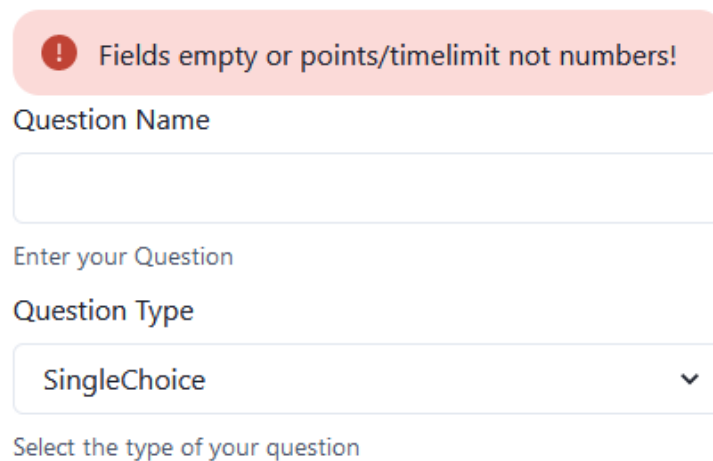
Both views are declared on their own subroutes of `/editquiz/quizId`. The edit view uses the route `/editquiz/quizId/editquestion/questionId` and the display view is available on `/editquiz/quizId/question/questionId`. As it is clear from the fact that the views have their individual URLs, they are separate pages, which 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 Sidebar, 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. 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 focus on the edit view as it requires more attention due to the need to manage the state. Doing so is unnecessary on the display view since the user cannot change any values.

The edit view (and the display view) contains a Component, which contains several input fields (disabled on the display view) for entering data such as the Question name or the time limit, as well as 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` converts a string to a number in JavaScript/TypeScript. The Component also features an error message displayed above the input fields if there are empty fields and also causes the save button to be disabled:



The screenshot shows a form with a red error message at the top: "Fields empty or points/timelimit not numbers!". Below the message is a text input field labeled "Question Name" with the placeholder text "Enter your Question". Underneath is a dropdown menu labeled "Question Type" with "SingleChoice" selected and a downward arrow. At the bottom, there is a label "Select the type of your question".

Figure 12: Error Message Empty Question fields

This is done by declaring a boolean state, which controls the alert's visibility and the ability to click the button. The variable is then updated 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, another one shows 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 challenging 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 that is accessible anywhere within the Component tree. When a Component needs a specific piece of data, `useContext` allows it to tap into that Context directly, retrieving the current value. In this example, the Hook is used to store all Options in an array of objects. For this, the TypeScript Index signa-

ture 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

The code 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 determining whether or not the parent Question is SingleChoice or MultipleChoice. If it is the former, additional code is needed to allow only one Option to be correct at one time. The checkbox itself is made with a Chakra UI IconButton Component, which is filled out if the Option is marked correctly. 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: !
13        prevOptions[id].isCorrect } };
14 });
```

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 combined with the Question data to make an API call to the server to save the changes to the Questions/Options.

4.3. Access Control with password

Another must-goal required adding functionality, which allows the tutor to restrict specific actions from being executed without verification, which 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 personal device and modify data without the tutor's consent. The solution for this problem is a verification system using a password. This means the tutor is not required to provide login data like a username and an email address to use the application.

4.3.1. Backend

In the Backend, the password is stored similarly to the Quiz data, which means it is stored in a hashed form in the same database as the rest of the application data. Doing so reduces complexity. A password Model was also created containing an ID set to "1" automatically since there can only 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, verifying 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.

4.3.2. Frontend

For the password verification to be helpful to the tutor, the previously described endpoints must be used in the Frontend.

When first loading the main page of the React application, the endpoint providing information, whether or not a password is set, is accessed using the `useEffect()` Hook. If no password is set, a red “Register” button is displayed next to the “New Quiz” button on the top of the page. If, however, a password has already been set and the corresponding hashed string is found in the database, the button is replaced with a green one, allowing the tutor to change the password. The latter is shown in Figure 9. Both buttons, when clicked, open their own Modals/Dialogs (contained in separate Components), where input fields are provided for the necessary data. They are laid out almost identically to the Modal shown in Figure 10.

On the other hand, the Modal used for verification is more involved in its implementation, the reason for this being the necessity to execute particular code or redirect to other pages on a successful verification attempt. To be more precise, verification is needed when trying to edit or delete a Quiz. It would be trivial if React provided a way to freeze all other Components until verification is complete. This way, using the Component like a JavaScript Dialog on a simple HTML page would be possible. However, this requires a lot of extra code, which could even include the manual implementation of a custom Hook. Thus, this approach was abandoned.

The easier solution, which was eventually discovered, is that passing a function as a prop to the Modal Component is feasible, which would then be executed on a successful password entry. This works very well when redirecting to the Edit Quiz Page since this only requires a change of route, which is easily done with the following function:

```
1 const onVerificationSuccessEdit = () => {  
2   navigate(`editquiz/${id}`);  
3 }
```

Code Snippet 23: redirect to Edit Quiz Page function

A problem arises when applying the same approach to deleting Quizzes, which uses a `ReactRouter` action function. There is no real way to run an action via code. It is possible to redirect to the route of the action, but doing so merely loads the corresponding page. In this case, the `deletequiz` route does not have a page, so an error occurs. There is, however, a way in React to submit a form using code instead of a submit button, which includes using the `useRef()` Hook. The core concept behind this Hook can easily be explained using its name: It provides a reference to some value or object. It can also be used to store the state itself. The difference between this Hook and the `useState()` Hook is that a change of the state or referenced data does not cause the Component to rerender. Another distinction of the `useRef()` Hook is how the state is accessed or mutated. Other than the `useState()` Hook, it does not use a function to update the state, as this would trigger a reload of the Component. Instead, it stores the value in a mutable object, which can be accessed or changed using its `value.current` property. This Hook also allows the developer to reference elements in JSX. [43]

The first thing that was attempted in this project was to create a reference to the form and to submit the form via that reference like so:

```
1  const deleteRef = useRef<HTMLFormElement>(null);
2
3  const onVerificationSuccessDelete = () => {
4    deleteRef.current?.submit();
5  };
6
7  <Form method="DELETE" action={`deletequiz/${id}`} ref={deleteRef}>
```

Code Snippet 24: useRef Hook attempt with Form

Note that this code has been condensed and is not executable in its current state, which should work, but it does not. Even after several hours of research, reading the React documentation, and studying the behavior of submitting a Form in JavaScript, no explanation for this oddity could be found. The most likely problem is that ReactRouter does not support calling an action without a submit button. The solution, which was eventually developed, is to create a button in the form that is not visible to the tutor. This button is then referenced and clicked via code. This solution is not optimal, but since the underlying problem is likely due to a bug in a library, it remains the most elegant way to fix this problem.

```
1  const deleteBtnRef = useRef<HTMLButtonElement>(null);
2
3  const onVerificationSuccessDelete = () => {
4    deleteBtnRef.current?.click();
5  };
6
7  <Button type="submit" display={"none"} ref={deleteBtnRef}></Button>
```

Code Snippet 25: useRef successful attempt with invisible button

4.4. Game Flow

The last central area to explore in the implementation section is the logic behind the Game itself. While also making use of a small REST API, the majority of the communication between the various participants in a Game is handled with SignalR. Said communication is handled via a series of messages, which occur throughout the different stages of a Game. The following sequence diagram shows all messages in the chronological order they occur in. Details about the implementations of specific messages will be shown further down in this section.

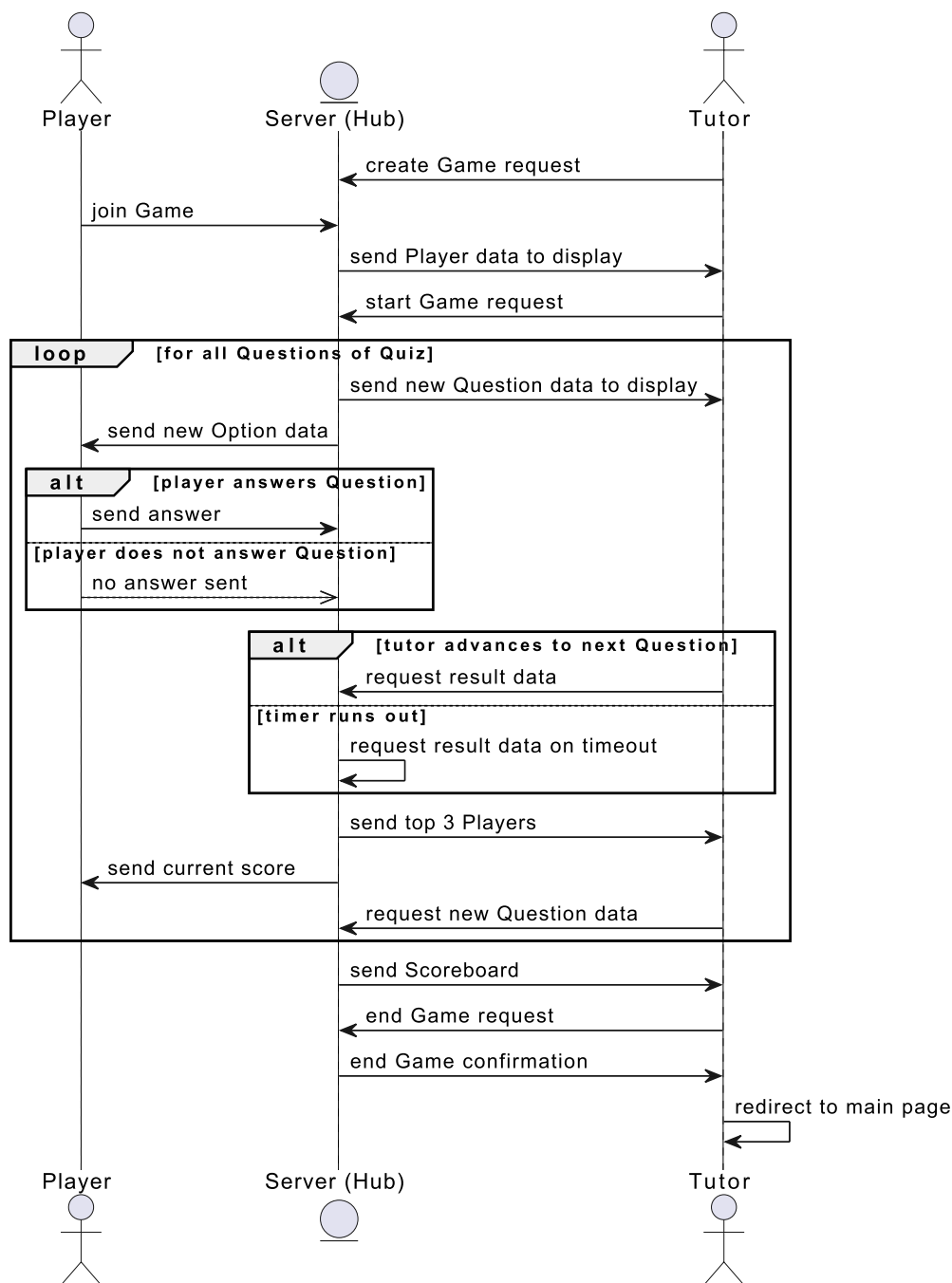


Figure 13: Game Flow Sequence Diagram

4.4.1. Models/DTOs

Similar to the Quiz data, information about Games is also stored in Models and transmitted using DTOs. To be specific, two new Models were required to structure the necessary data properly.

The first of these is a Model for a Player of a Game. Its general composition is not that different from the Models shown in the Quiz Management section. One potentially noteworthy aspect is that the icon of a Player is not a custom image, as is the case with competing Quiz Game solutions. Instead, a string is used to store an Emoji as the icon, which is possible since they are no different from other Unicode characters. Doing so makes the application more versatile as it does not rely on a third-party library.

```
1 public class Player
2 {
3     [Key]
4     [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
5     public Guid PlayerId { get; set; }
6     public string Name { get; set; }
7     public string Icon { get; set; }
8     public int Score { get; set; } = 0;
9
10    public Guid GameId { get; set; }
11    public Game Game { get; set; }
12 }
```

Code Snippet 26: Player Model

The Player Model is then included in the larger Game Model. It also includes a Quiz, but other than the previous times a Model was made part of another, the Quiz Model does not contain any reference to the Game Model. By doing this, cascading delete is not triggered when removing a Game from the database, which makes it possible to play multiple Games with the same Quiz. The Players are, however, still deleted.

```
1 public class Game
2 {
3     [Key]
4     [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
5     public Guid GameId { get; set; }
6     public List<Player> Players { get; set; } = new List<Player>();
7     public Quiz Quiz { get; set; }
8 }
```

Code Snippet 27: Game Model

Various DTOs were also created for these Models.

4.4.2. Controller/Repositories

Even though the Game functionality is mainly centered around real-time communication requiring SignalR, certain pieces of functionality can also be performed with a HTTP API. Thus, a separate Controller was created for the Game. The two main ways this Controller is used are to create and delete a Game once it is finished. These endpoints are almost identical to their counterparts in the Quiz, Question, or Option Controllers. Thus, they will not be shown here to prevent the spotlighting of duplicate code.

Repositories were also implemented for both the Game and the Player Models. They are used both in the Game Controller and the Game Hub. The repos provide basic functionality for interacting with Games/Players by, for example, adding a Player to a Game or increasing the Player's score.

4.4.3. Initialize/Join Game

The first thing a tutor has to do to provide a Game for their students to play is select a Quiz for the Game, which is done on the main page (Figure 9) by pressing the “Play” button on a Quiz Card. This then makes an API call to the server, where a new Game is created in the Game Controller. Subsequently, the tutor is redirected to the Waiting Room page. A link is displayed for the page where students can join the Game. This link is `location.hostname` browser property.

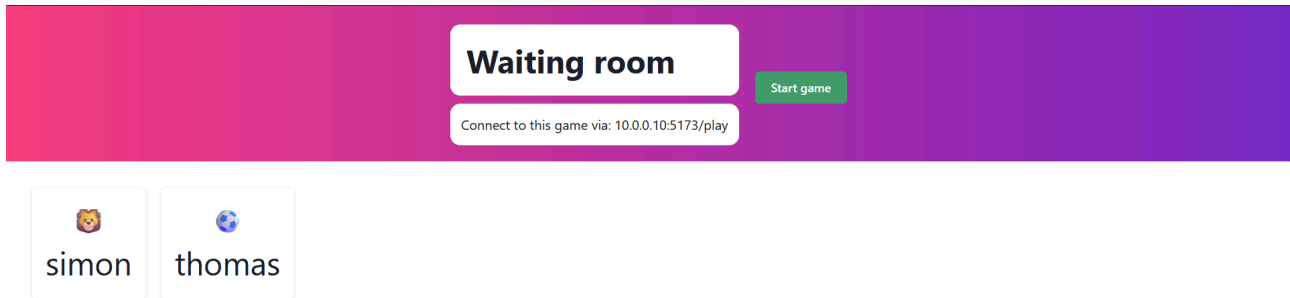


Figure 14: Game Waiting Room

Once started, Players can join a Game immediately. They can do so by navigating to the application's `/play` route. There, they are met with an opened Modal where they can enter their name and select their preferred Emoji as an icon. This is done using a predefined Emoji picker Component, which is available on the NPM and installed using the command `npm i emoji-mart`.

```
1 <Picker theme="light" data={Data} onEmojiSelect={(event: { native:
  SetStateAction<string>; }) => {
2   setIcon(event.native);
3   setPickerVisible(!isPickerVisible);
4 }} />
```

Code Snippet 28: Emoji Picker Component

Once selected, the local state of the icon is changed, and the Picker is not visible. If both a name and icon have been selected, the Player can join the Game. Before that, a connection to the Game Hub has to be created. In the case of the join Game page, this connection is declared with the `useRef()` Hook. Doing so is necessary since a state change, and thus a Component rerender, is triggered every time the Player enters a new character in a text field, which would cause the connection to be restarted each time this occurs.

```
1 useEffect(() => {
2     connection.current = new signalR.HubConnectionBuilder()
3         .withUrl(`${sURL}/Game`)
4         .build();
5
6     connection.current.start()
7         .then(() => console.log('Connection started!'))
8         .catch(() => console.log('Error while establishing connection :('));
9
10    return () => {
11        if (connection.current) {
12            connection.current.stop()
13                .then(() => console.log('Connection stopped!'))
14                .catch(() => console.log('Error while stopping connection :('));
15        }
16    };
17 }, []);
```

Code Snippet 29: Start Connection to Hub in Frontend

This is done inside the `useEffect()` Hook to make the connection available as soon as the Component loads. First, the connection is created, and messages are subsequently outputted to the browser console on a successful connection or a failure. The return statement of the `useEffect()` Hook is simply executed when the Component is unloaded, meaning when the page is changed, which is done to dispose of the connection before bugs can occur.

If the connection is started correctly and no error appears, the Player can join the Game by simply pressing the “Join” button. This button then sends a message to the server. The following closes the Model.

```
1 connection.current.invoke("RegisterPlayer", dto)
2 .then(() => {
3     onClose();
4 })
5 .catch(err => console.error(err));
```

Code Snippet 30: Join Game Message

Once a Player is added to the Game, the Hubs send a message to said Player containing their ID. At the same time, the Player is stored in the database. If no Game is open for joining, the Player will receive an error message saying so. There are many options available to send individualized messages to Players. The one chosen for this project is to add the Player's connection ID, which is unique to each connection created to the Hub. In said group, the connection ID and the Player ID are associated. This is necessary since the connection ID

of a Player changes once they are redirected to a new page. There, the new connection ID has to be added to the group, and the Player can continue to receive messages.

```
1 public async Task RegisterPlayer(CreatePlayerDto dto)
2 {
3     await Console.Out.WriteLineAsync(_currentQuestionIndex.ToString());
4     if (_currentQuestionIndex == 0 && _game != null)
5     {
6         Player createdPlayer = await _playerRepositoryCreatePlayer(dto);
7         await _gameRepository.AddPlayer(createdPlayer);
8
9         await Groups.AddToGroupAsync(Context.ConnectionId,
10            createdPlayer.PlayerId.ToString());
11
12         await Clients.All.SendAsync("playerAdded", _mapper
13            Map<PlayerDto>(createdPlayer));
14
15         await Clients.Group(createdPlayer.PlayerId.ToString())
16            .SendAsync("addedYou", createdPlayer.PlayerId);
17     }
18     else
19     {
20         await Clients.Client(Context.ConnectionId).SendAsync("gameRunning");
21     }
22 }
```

Code Snippet 31: Player Registration on Server

Messages can be received by the client by creating listeners for specific messages. The following one listens to the “addedYou” message and stores the Player’s ID in LocalStorage. From there, it can be accessed whenever it is required, for example, when submitting an answer.

```
1 connection.current.on("addedYou", (id: string) => {
2     localStorage.setItem('playerId', id);
3 })
```

Code Snippet 32: Message listener

At the same time, another message is sent to all clients. This message is listened to in the Waiting Room and is used to display new Players as they join automatically.

4.4.4. Gameplay

As soon as all students have joined, the tutor can start the Game by pressing the “Start Game” button on the Waiting Room page, which sends the “StartGame” message to the SignalR Hub on the server. There, the first Question is sent to all clients. This is done separately

from the StartGame function since sending Questions is not just required when starting the Game. On the tutor's computer, which screen is projected to a projection screen or shown on a digital whiteboard, a redirection to the page of the /gameTeacher route occurs. There, all relevant information about the Question is displayed straightforwardly:

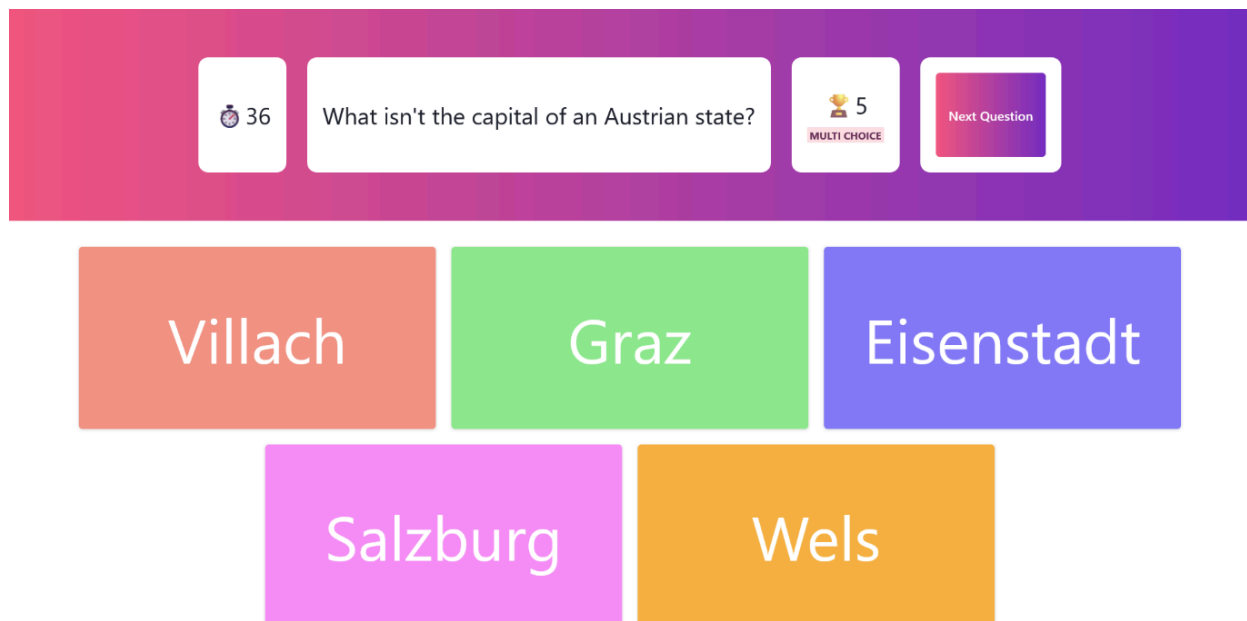


Figure 15: Game Teacher View

The top part of the page features a pleasing gradient background with multiple Cards for important information. On the first Card, the timer is displayed. It continuously goes down until it reaches zero, at which point the Question is closed for answering. Further detail on the implementation of the timer in the Frontend and Backend will follow further down this section since it requires further information to be comprehended in its entirety. Next to the timer, the name of the Question is displayed. The last two Cards display the points received if the correct Option or Options are selected and a button to advance to the next Question, respectively. The bottom part of the page features the individual Options displayed in a grid layout. The colors shown are predefined in an array and assigned to the Cards using a previously unknown feature of the `array.map()` function. As previously explained, this function allows for an action to be performed on each element of an array, which is used here to display a Card for each Option. This has already been done similarly in the Sidebar on the Edit Quiz page. The new functionality used here allows the developer to define an incrementing number similar to a for loop. This number is then used to select the correct color from the colors array.

On the Player side, the student is redirected to the page of the /gameStudent route. There, a different layout is displayed depending on whether the Question is single or multiple-choice. The difference between the two is that Options are selected and submitted via a separate button on a multiple-choice Question. The styling of the Options is also different. This page

also features a mobile-friendly responsive design, as it is likely that a student might want to play on their phone instead of a laptop or computer.

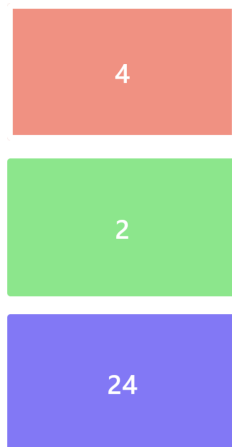


Figure 16: Game Student View Single Choice



Figure 17: Game Student View Multi Choice

The way answers are submitted differs slightly between the two. Both rely on the same function to send the IDs of selected Options to the Hub alongside the Player ID stored in LocalStorage, which is done comparably to any other SignalR message described until now. The two differ in how this function is used. In a single-choice Question, each Option contains a click event handler, which runs this function with the ID of the Option as an argument:

```
1  const submitSingle = (event: React.MouseEvent<HTMLButtonElement>, id: string)
  => {
2      event.preventDefault();
3      sendToHub([id]);
4  }
```

Code Snippet 33: Submit single Answer

Even though only a single ID needs to be sent to the Hub, an array is still used to make the sendToHub function usable for both use cases. On the Backend, the sent IDs are used to create a list of Options. This list is then checked, and if all Options are correct, the player's score is updated. If the Options are part of a multiple-choice Question, the IDs are added to an array and submitted via a submit button.

```
1  const selectMulti = (event: React.MouseEvent<HTMLButtonElement>, id: string) =>
  {
2      event.preventDefault();
3      multiIds.includes(id) ?
4      setMultiIds([...multiIds.filter((m) => m !== id)]) :
5      setMultiIds([...multiIds, id]);
6  }
```

Code Snippet 34: Submit multiple Answers

This function first checks if the ID has already been added to the array. If it has, it is removed by filtering the array. Otherwise, it is simply added. This is done to allow for an Option to be both selectable and deselectable.

Once all Players have submitted an answer, the tutor can press the “Next Question” button in the top portion of the teacher view. A message is then sent to the Hub requesting the scores of the students, which leads to the following function being called:

```
1 public async Task RequestScores()
2 {
3     _game.Players = await _context.Players.ToListAsync();
4
5     var playersWithScore = _game.Players.Where(p => p.Score > 0).ToList();
6     var topPlayers = playersWithScore.Count < 3
7         ? playersWithScore.Concat(_game.Players.Except(playersWithScore)
8             .OrderBy(p => Guid.NewGuid()).Take(3 - playersWithScore.Count)).ToList()
9         : playersWithScore.OrderByDescending(p => p.Score).Take(3).ToList();
10
11     await _hubContext.Clients.All.SendAsync("topPlayers", topPlayers);
12
13     foreach (Player player in _game.Players)
14     {
15         await _hubContext.Clients.Group(player.PlayerId.ToString())
16             .SendAsync("resultReceived", player.Score);
17     }
18 }
```

Code Snippet 35: Request Scores function

This function uses a hubContext instead of sending messages normally, which is necessary since it is executed once the timer has run out. In that case, the current instance of the Hub at the time of the timer creation is no longer active once it reaches zero. This is due to a Hub having a very short lifecycle as it is often disposed of and then created again to improve memory usage. The hubContext is needed because it allows access to the Hub anytime. The way it is used here is not intended as it is usually used outside a Hub, but its characteristics also make it ideal for this use case. The line `_game.Players = await _context.Players.ToListAsync();` fetches the latest list of Players from the database and assigns it to `_game.Players`, which is done to ensure that `_game.Players` always contains the most up-to-date data from the database.

Next, a list of the top three Players is created. First, all Players who have a score greater than zero are added to a list. This list is then sorted by score, and the three best students are selected. If there are fewer than three Players with a score, the list is extended by adding

as many scoreless Players as needed to reach three again. After that, the list is sent to the tutor's computer, where a Modal is shown with a table showing its data:

Current Leaderboard



PLACEMENT	PLAYER	CURRENT SCORE	Next Question
1.	 thomas	5	
2.	 simon	10	

Figure 18: Top three Players

Each Player also receives their current score via a Modal. The student is also informed whether or not the Question has been answered correctly by storing each score received in an array and comparing the last two scores. If they are identical, a mistake was made when answering. The tutor can then advance to the next Question by pressing the button in the Modal, at which point the server sends the new data to the clients where the contents of the pages are updated.

Timer

The timer was first created using the Hub's `SartGame()` method. There, the time limit of the first Question is assigned to it, and the `RequestScores()` function is designated to be executed when the timer runs out. The timer is simulated on the Frontend using the `useEffect()` Hook.

```
1  useEffect(() => {
2      let interval: number = 0;
3      if (timerActive && timerSeconds > 0) {
4          interval = setInterval(() => {
5              setTimerSeconds((timerSeconds) => timerSeconds - 1);
6          }, 1000);
7      } else if (!timerActive && timerSeconds != 0) {
8          clearInterval(interval);
9      }
10     return () => clearInterval(interval);
11 }, [timerActive, timerSeconds]);
```

Code Snippet 36: Timer Frontend

The Hook takes a function and an array of dependencies (`timerActive` and `timerSeconds`) as arguments. The function inside the Hook is executed every time one of the dependencies changes.

In this function, an interval is set up to decrement `timerSeconds` by one every second, but only if the timer is active and has not yet reached zero. If the timer is inactive and `timerSeconds` is not 0, the interval is cleared, effectively pausing the timer. The return statement clears the

interval, ensuring the timer stops when the Component is unmounted or before it updates, which prevents potential memory leaks.

Back on the server side, the timer is stopped when the scores are requested and starts again with the new time limit of the next Question.

4.4.5. End Game

After the last Question of a Quiz has been reached and the tutor has clicked the “Next Question” button in the Modal, the server sends a list of all Players sorted by their score, which is then displayed on a new route.

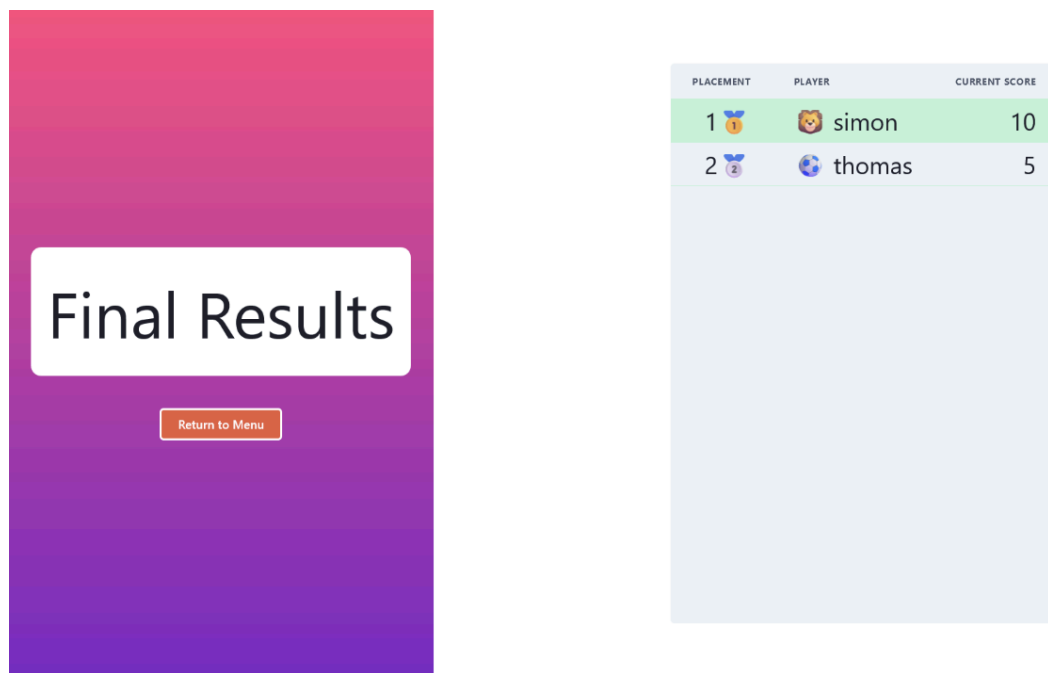


Figure 19: Scoreboard

While this process works, an error occurs because the tutor’s client still tries to request a new Question. The most interesting part of this problem is when it occurs: The error message is displayed long after the redirection to the scoreboard has already taken place. Due to this, it is very likely that the reason lies within the way React dismantles its Components. Somehow, this process seems to send a redundant message to the Hub. While certainly not ideal, this error was ignored because it does not affect the flow of the Game itself.

Finally, the tutor has to press the “Return to Menu” button, which sends a message to the server requesting that the Game be deleted from the database. Once this is done, the server sends a confirmation back, and the tutor is redirected to the main page, where they can either edit a Question, create a new one, or play another Game.

5. Conclusion

The implementation of the must-goals, with the exception of a few bug fixes, was completed on February 7, 2024. It took approximately 110 hours of time entirely spent on coding. The planning and research were completed in about 40 hours, and the documentation was written in another 40 hours. In total, the project required 190 hours to complete.

As previously alluded to, all must-goals were completed. However, no should-goals were completed because of time constraints.

5.1. Results

The results of this diploma thesis are presented and explained below.

5.1.1. Backend

The project offers a high-performing and well designed server, which provides both a highly useful REST API for the seamless management of Quizzes and their underlying Questions and a responsive SignalR Hub used to ensure real-time communication between clients during a game. Both components provide outstanding security and efficiency thanks to the robust ecosystem of ASP.NET Core. Data is persistently stored in a database and represented in well-structured Models, enhancing code readability and making it more readable.

Overall, the project's Backend development proved more efficient and streamlined than anticipated. The implementation of the SignalR Hub was completed incredibly quickly, as it had been assumed that it would be the most complicated section of the whole project at the start. In reality, SignalR was easier to work with than anticipated when starting the project.

The REST API development progressed smoothly due to clear RESTful design principles, resulting in an intuitive and easy-to-understand API. While Entity Framework Core with PostgreSQL was chosen for the database needs, the Enum configuration required additional research. This minor hurdle did not significantly impact the development timeline. Data is also transferred using well-desired DTOs powered by the powerful yet easy-to-use tool AutoMapper. They ensure that the data transfer between the client and the server is done in an efficient and logical manner.

A minor challenge arose when the initial API design was not fully REST compliant, necessitating a complete redesign, which proved to be a valuable learning experience, leading to improved API design skills.

The server additionally offers a simple yet effective API for adding passwords, securely hashed with Bcrypt.NET and stored in the database. A verification mechanism ensures authorized user access to specific functionalities, adding an extra security layer and protecting data integrity.

Overall, the application boasts a robust, efficient, and secure Backend, providing a solid foundation for a smooth and enjoyable user experience.

5.1.2. Frontend

In addition to the previously described server, the application also provides several GUI pages, which are pleasing to the eye thanks to the well-designed but still very functional Components from Chakra UI. They are also highly user-friendly, allowing even inexperienced tutors to create engaging Quizzes without hassle. It also includes a design for the player pages, which is remarkably versatile as it is styled to look and function well on full-sized desktop monitors and small smartphones. The fact that the Frontend is implemented entirely with web technologies also makes it not only efficient but also cross-platform. Additionally, the tutor has the ability to set a password to protect their precious Quizzes from meddling students, which is an excellent addition to classroom use.

The implementation of the Frontend went significantly less smoothly than the Backend implementation. A significant factor for the unexpected complexity was the need for an extensive state management solution for the Edit Quiz page to achieve a great layout. Initially, it had not been anticipated that this would need as many complicated Components as it ended up needing. It also had to be rewritten due to the initial implementation not meeting the high standards set for this project. An enormous breakthrough occurred when the `useContext()` Hook was first used. After this, everything fell right into place and resulted in a well-designed and intuitive page, which, as a whole, represents the best this application has to offer.

Consequential delays in the development process also occurred when implementing the Game pages as they needed the highest amounts of polish in the entire Frontend due to being interacted with by both students and tutors. However, this effort paid off thanks to the great design and ease of use provided by the pages in their current state.

5.2. Potential future additions

In its current state, the project already offers a feature-rich Quiz application with a great user-friendly design. The connection between the Frontend and the Backend works without significant errors, and the experience of playing a game Game is very enriching and informative thanks to the ability to create exciting Quizzes.

Even though all must-goals were completed, there is still room for improvements, including containerizing the application with Docker, making it even more versatile. Additionally, minor bugs and other small problems could be ironed out. Finally, the ability to export and import Quizzes could help build a community and foster the spirit of learning within everyone willing to expand their horizons.

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. Ponuthurai, *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/eaCatalog/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.githubusercontent.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 Programming Model (TAP) in ASP.NET MVC". Accessed: Feb. 25, 2024. [Online]. Available: <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/>
- [43] A. Isiaka, "A Thoughtful Way To Use React's useRef() Hook". Accessed: Mar. 01, 2024. [Online]. Available: <https://www.smashingmagazine.com/2020/11/react-useref-hook/>

Code Listings

Code Snippet 1: Postgres Docker Compose	23
Code Snippet 2: Option Model	26
Code Snippet 3: Question Model	27
Code Snippet 4: Question type Enum	27
Code Snippet 5: Quiz Model	28
Code Snippet 6: Manual AutoMapper assignment	28
Code Snippet 7: DbContext	29
Code Snippet 8: Question Repository Interface	30

Code Snippet 9: DeleteQuestion implementation in QuestionRepository	30
Code Snippet 10: Controller Dependency Injection	32
Code Snippet 11: GetAllOptions Endpoint	33
Code Snippet 12: Quiz loader Function	34
Code Snippet 13: getQuizzes() Method	34
Code Snippet 14: Start Game Button	35
Code Snippet 15: Question loader function	36
Code Snippet 16: loading Quiz details in Modal	37
Code Snippet 17: Edit Question Time Limit Input Field	38
Code Snippet 18: Edit Question Error Check	39
Code Snippet 19: Load Options into Context State	40
Code Snippet 20: useContext Hook example	40
Code Snippet 21: check if there is already a correct Option	41
Code Snippet 22: Verify Password endpoint	42
Code Snippet 23: redirect to Edit Quiz Page function	43
Code Snippet 24: useRef Hook attempt with Form	44
Code Snippet 25: useRef successful attempt with invisible button	44
Code Snippet 26: Player Model	46
Code Snippet 27: Game Model	46
Code Snippet 28: Emoji Picker Component	47
Code Snippet 29: Start Connection to Hub in Frontend	48
Code Snippet 30: Join Game Message	48
Code Snippet 31: Player Registration on Server	49
Code Snippet 32: Message listener	49
Code Snippet 33: Submit single Answer	51
Code Snippet 34: Submit multiple Answers	51
Code Snippet 35: Request Scores function	52
Code Snippet 36: Timer Frontend	53

Image Listings

Figure 1: General Architecture Overview	22
Figure 2: Backend Architecture Design	23
Figure 3: Postgres[32]	23
Figure 4: AutoMapper[33]	23
Figure 5: SignalR[34]	23
Figure 6: Bcrypt.NET[35]	23

Figure 7: Frontend Architecture Design	24
Figure 8: React Router[37]	24
Figure 9: Main Page Screenshot	33
Figure 10: Create Quiz Dialog Screenshot	35
Figure 11: Edit Quiz Page	36
Figure 12: Error Message Empty Question fields	39
Figure 13: Game Flow Sequence Diagram	45
Figure 14: Game Waiting Room	47
Figure 15: Game Teacher View	50
Figure 16: Game Student View Single Choice	51
Figure 17: Game Student View Multi Choice	51
Figure 18: Top three Players	53
Figure 19: Scoreboard	54