ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Antosha** Jméno: **Andrii** Osobní číslo: **466900**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Webová aplikace pro kvíz v reálném čase

Název bakalářské práce anglicky:

Real-time quiz web application

Pokyny pro vypracování:

Cílem bakalářské práce je vytvořit webovou aplikaci pro multiplayerové kvízové hry v reálném čase (Kvízová hra je definována takto: hraje ji dva hráči, kde hráči v konkurenční formě odpovídají v reálném čase na náhodné triviální otázky).
Cílem této práce je prozkoumat existující řešení, studovat problémy webových aplikací v reálném čase včetně možných technologií a algoritmů, navrhnout a implementovat webovou aplikaci pro hraní trivia kvízů v reálném čase.
Aplikace by se skládala ze dvou částí (front-end a back-end), které budou komunikovat přes definované rozhraní pomocí protokolu HTTP / WebSocket.
Základní funkčnost aplikace: registrace uživatele, zobrazení seznamu dostupných kvízových her, ke kterým se můžete připojit, hraní kvízových her na různá témata v reálném čase s ostatními uživateli. Uživatelé mohou vytvářet vlastní kvízové hry s vybranými tématy a definovat počet otázek. Na konci hry mohou uživatelé zobrazit dashboard s osobními statistikami a historii her.
Aplikace by měla být navržena s ohledem na stolní i mobilní zařízení. Po vývoji aplikace by měly být provedeny různé druhy testování: testování funkčnosti, testování použitelnosti, testování výkonu, testování bezpečnosti.

Seznam doporučené literatury:

Andrew Lombardi: Lightweight Client-Server Communications, 1st edition (2015)
Addy Osmani: Learning JavaScript Design Patterns, O'Reilly Media 2017, ISBN 9781449331818
https://graphql.org/
https://www.apollographql.com/docs/react/
Eric Elliot: Programming JavaScript Applications: Robust Web Architecture With Node, Html5, And Modern Js Libraries, 1st Edition, ISBN-10: 149195029
Eric Elliot: Composing Software: An Exploration of Functional Programming and Object Composition in JavaScript, ISBN-10: 1661212565

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Ivan Jelínek, CSc., kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

doc. Ing. Ivan Jelínek, CSc. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Bachelor Thesis

Real-time quiz web application

Antosha Andrii

Supervisor: doc. Ing. Ivan Jelínek, CSc.

Study Program: Software engineering and technologies, Bachelor

May 21, 2020

Aknowledgements

I wish to show my gratitude to my supervisor, doc. Ing. Ivan Jelínek, CSc., who consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it. I also wish to acknowledge the support and great love of my family, they kept me going on and this work would not have been possible without their input.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 22, 2020 ....

Abstract

This thesis focuses on the design and implementation of web application for multiplayer real-time quiz games. Requirements for the application of technologies are addressed and existing solutions are analyzed. Technologies fulfilling the addressed requirements are then explored. Finally, a proof-of-concept application is designed and implemented.

Keywords: web application, real-time web, JavaScript, React.js, Node.js

Abstrakt

Táto bakalářská práce se věnuje návrhu a implementaci webové aplikace umožňující hrát multiplayerové kvizové hry ve skutečném čase. Rešeršní čast práce se nejdříve zabývá analýzou požádavků, zkoumá existující řešení a studuje problémy webových aplikací v reálném čase. Dále jsou představeny technologie které tyto požadavky splňují a umožňují komunikaci v reálném čase. Na základě těchto poznatků je navržen a implementován prototyp webové aplikace pro multiplayerové kvizové hry.

Klíčová slova: webová applikace, web ve skutečném čase, JavaScript, React.js, Node.js

Contents

Introduction **1**

1 Analysis and requirements **3**

1.1 Application introduction 3
1.2 Functional requirements 3
1.3 Use Cases 4
1.4 Non-functional requirements 6
1.5 Existing solutions 7
1.6 Summary 8

2 Design and proposed architecture **9**

2.1 Communication between client and server 9
2.2 Proposed architecture 11
2.3 Database schema 13
2.4 Technology stack 14
2.5 Summary 17

3 Implementation **18**

3.1 Project structure 18
3.2 Communication channels 19
3.3 Authentication 22
3.4 User Interface 23
3.5 Building the application 27
3.6 Summary 29

4 Testing **30**

4.1 Functional testing 30
4.2 Usability testing 31
4.3 Performance testing 36
4.4 Summary 37

Conclusion **38**

Bibliography **40**

A Acronyms **43**

B Contents of enclosed CD 44

List of Figures

1.1 Use case diagram 4

2.1 Three-tier architecture 12
2.2 Entity relationship diagram 14
2.3 Node.js event loop 15

3.1 Separation of communication channels 20
3.2 Stateful authentication flow 22
3.3 Token based authentication flow 23
3.4 User flow of the application 24
3.5 Preview of application home page 25
3.6 Preview of application Sign Up page 25
3.7 Preview of application Dashboard page 26
3.8 Preview of modal window for creating games 26
3.9 Preview of application Sign Up page 27
3.10 Preview of application Sign Up page 27
3.11 Preview of application Profile page 28

List of Tables

1.1 Matrix of Use Cases and Functional Requirements 6

3.1 Application REST API endpoints 20

4.1 Usability test results of participant 1 35
4.2 Usability test results of participant 2 35
4.3 Usability test results of participant 3 35
4.4 Overall latency distribution for connected users 36
4.5 Overall latency distribution for creating games 37

Introduction

In recent years web has been rapidly evolving and applications with real-time communication have been gaining in popularity. Many applications that we use on a daily basis are built upon real-time web technologies. These technologies have a number of common use cases such as displaying data, statistics, notifications or news as soon as it becomes available. Games are another example of such applications where these technologies allowing multiple users and systems to instantly communicate with each other which could bring better user experience and make the application much more functional.

Goals and objectives

The core goal of this work is to study the problem of real-time web applications and implement web application for playing multiplayer quiz games in real-time.
The first objective includes research on existing solutions and technologies for implementing real-time communication between server and client. We will analyze different techniques and approaches used in modern web applications.
The second objective of this work is to propose an architecture and implement proof-of-concept web application based on the analysis done in the research.

Motivation

As web technologies constantly evolve, seamless communication becomes a necessity while real-time data transmission is an inevitable outcome for the future of web-based information systems. These progressions along with the latest technologies generate several opportunities that will be explored in this project: advancements in web application and creating a working prototype for a web-based multiplayer game.

Structure

In the first chapter [1] of this work we will take a look at the existing solutions. We will also focus on real-time communication techniques used in modern web applications. Finally, application requirements would be defined. In the next chapter [2] we will describe the proposed architecture of the application. We will also review the technologies being used for implementation. The following chapter [3] describes the implementation details of client and server sides of the application. We will take a look at how core modules of the application work and document client-server communication. Finally, in the last chapter [4] the application will be tested. Based on the results of testing, we will verify the functionality and find weak points of the application.

Chapter 1

Analysis and requirements

This chapter introduces the application and outlines the requirements. In good alignment with basic software engineering practice, the system's design is to be presented at a reasonable level of abstraction. In this chapter we will describe both functional and non-functional requirements. This chapter also contains an introduction to the techniques used for real-time communication in modern web applications. Finally, we will research existing solutions.

1.1 Application introduction

The goal of this work is to create a web application for multiplayer real-time quiz games. The game is defined as the following: it is played by two players that answer random trivia questions in competitive form. The application only allows registered users to participate in games. Games are created by users and can be customized by different criteria such as difficulty, number of questions and topic. After a user has created a new game, it is displayed in the game lobby and other users can join it. When the game is being played, users get points for correct answers which are displayed along with their opponent's points. At the end of the game users can see their overall score and the winner of the game. The application also allows users to view their personal statistics and game history in their profile page.

1.2 Functional requirements

Functional requirements are useful for describing what the system must offer. Its purpose is to better understand the product capabilities.

- **F1** — Application should display list of online users and games user can join
- **F2** — Users should be able to customize a game when creating a new one
- **F3** — For one user login state should be either logged in or not logged in
- **F4** — New users can create an account in the system
- **F5** — Application should display user's and opponent's points during the game

- **F6** — Users should be able to view personal statistics
- **F7** — Application should handle errors and display notifications to user
- **F8** — Users should be able to join games created by other users

1.3 Use Cases

We will start with the use case diagram (see figure 1.1) that summarizes the relationships between use cases and actors. Each use case shown on this diagram will be described later for clearer specification.
The implemented application does not have any user roles with predefined set of permissions, therefore we would only distinguish between authenticated and non-authenticated user.

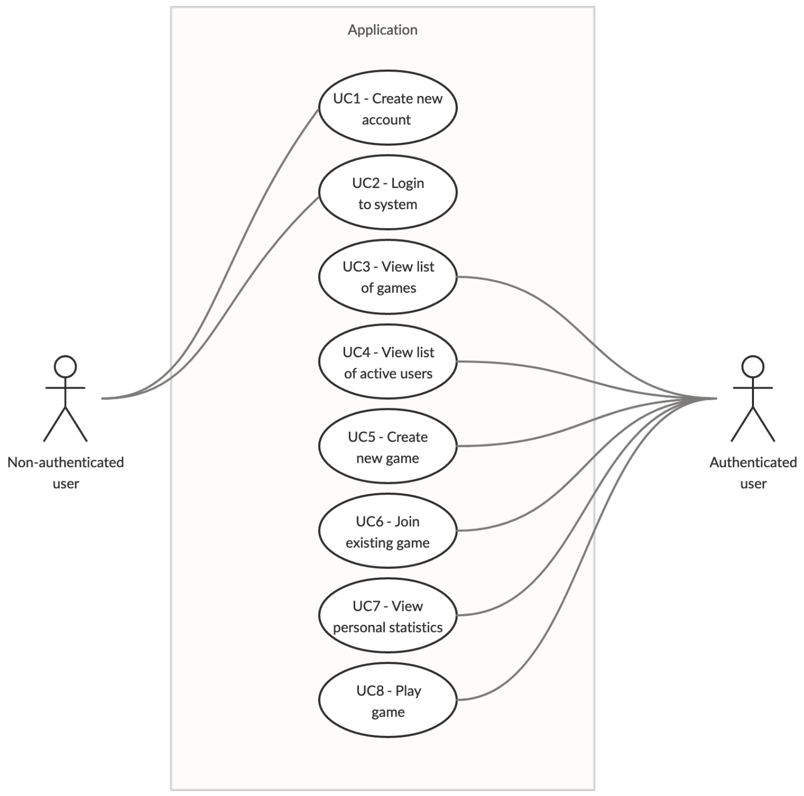


Figure 1.1: Use case diagram

UC1 - Create new account

Description: new user can create an account using the sign-up form
Actor: non-authenticated user
Event flow: after clicking the `Sign Up` button in the top right corner of the application user is redirected to sign-up page with form containing `Username` and `Password` fields. After

filling and submitting the form, application validates the form data and creates new user. The application redirects user to the dashboard page where the username is displayed in the top right corner. In case of error, notification is displayed.

UC2 - Login to system

Description: user with existing account can login into the system
Actor: non-authenticated user
Event flow: after clicking `Login` button in the top right corner of the application user is redirected to login page with form containing `Username` and `Password` fields. After filling and submitting the form application validates the form data and logs user in. Application redirects already authenticated user to dashboard page where username is displayed in the top right corner. In case of an error notification is displayed.

UC3 - View list of games

Description: authenticated user can see a list of available games to join
Actor: authenticated user
Event flow: on dashboard page, authenticated user can see the list of games that he can join. User can see `Play` button next to all games except those that are created by this user.

UC4 - View list of active users

Description: authenticated user can see a list of online users
Actor: authenticated user
Event flow: on dashboard page, authenticated user can see the list of online users including himself.

UC5 - Create new game

Description: authenticated user can create a new game
Actor: authenticated user
Event flow: on dashboard page user clicks `Create new game` button. Application displays modal window with form containing editable game parameters and button `Create`. User can change individual game parameters and click `Create` to submit the form. After submitting the form, modal window will be closed and newly created game will be displayed in the list of active games. In case of an error notification is displayed.

UC6 - Join existing game

Description: authenticated user can join games created by other users
Actor: authenticated user
Event flow: on dashboard page user sees the list of available games. User can click `Play` button and join the game as an opponent.

UC7 - View personal statistics

Description: authenticated user can view personal statistics
Actor: authenticated user

Event flow: on the personal page, application displays user personal statistics such as total number of games played, number of wins, draws and losses, a list of recently played games and a list of categories with corresponding number of games played within each category.

UC8 - Play game
Description: authenticated user can create a new game
Actor: authenticated user
Event flow: after authenticated user joins a game, application displays game page with `Ready` button. After user clicks this button and game opponent does the same on his device application shows question, answer options and points of both players. After the game is finished application displays points of each player along with game result.

Table 1.1 shows the relationships between defined functional requirements and use cases. Each use case ticks a box under one or more functional requirements, indicating that the use case may be satisfied with a repository platform that supports the related functional requirement(s).

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
F1			X	X				
F2					X			
F3		X						
F4	X							
F5								X
F6							X	
F7	X	X			X			X
F8						X		

Table 1.1: Matrix of Use Cases and Functional Requirements

1.4 Non-functional requirements

Non-functional requirements specify the quality attributes of the system. They help us to define the performance attributes of the system along with system's reliability and availability.

- **NF1** — Application should be accessible [1]
- **NF2** — Application should be easy to deploy
- **NF3** — When a game is interrupted due to disconnection opponent should see the corresponding message
- **NF4** — Application should use WebSockets for real-time data exchange [2]

- **NF5** — All pages must be responsive and be viewable in different device sizes – mobile, tablets and desktop
- **NF6** — HTML is writtenin HTML5 standard and CSS in CSS3 standard

1.5 Existing solutions

This section analyzes existing solutions of multiplayer quiz games in which users can compete with each other. First we will review similar software solutions available and then compare them to the proposed application described in this work.

1.5.1 Platforms for multiplayer quiz games

QuizUp [3]

QuizUp is a popular mobile game for playing trivia online with friends or random opponents. The game provides a wide variety of topics to choose from and also features topic communities with each topic set within a category where users can play, post and interact with other users who share their interests. On top of that, users can view any user's profile and see their interests, follow them, play against them at a trivia game, or send them a direct message which makes QuizUp a social network.

TopQuizz [4]

TopQuizz is a web application for playing trivia quiz games against other players or as a single player. It lets users create their own quiz games as well as playing predefined ones, however it does not allow to customize the quiz except for setting the topic. For registered users it provides the ability to collect "neurons" which are points user gets for playing games, giving correct answers in the game or publishing the quiz.

Kahoot! [5]

Kahoot is game-based learning platform, mostly used as educational technology. It can be used to review student's knowledge, for formative assessment or playing trivia quizzes. Kahoot allows user to create their own quiz games or access millions of ready-to-play games on any topic in different languages. Besides the core quiz application, Kahoot also offers a few different applications for interactive presentations, online training and self-studying.

QuizWitz [6]

QuizWitz is a party quiz game that allows users to play on their phone, tablet or other smart device as a game controller. Other than party games, the application also allows to play as a single player, and similar to the other applications provides an option to create quizzes and publish them.

1.5.2 Evaluation of existing solutions

After analyzing a number of existing applications in this category, we can see that they offer users the option to create games for groups of people or the option to play as a singer player. The implemented application, however, focuses on the ability to play against random players, similar to **QuizUp**, however, it will be a web application instead of mobile. Additionally, most of the existing applications let users play quizzes created by other users or have the option to create their own quiz, which is not the case in the implemented application where we want games to be generated by the system and to give users the ability to customize the game parameters.

1.6 Summary

In this chapter we have introduced the idea of the implemented application. We have also determined the requirements for the application and defined the use cases that provide a higher-level view of the system. Finally, we have examined the existing solutions.

Chapter 2

Design and proposed architecture

This chapter discusses the architecture of the implemented application. First, we will describe the way server and client would communicate and what techniques could be used to implement real-time communication. Then, we will look into individual parts of the application: client, server and database. In this chapter we will also describe the domain model of the application. Finally, we will introduce the technologies being used for development.

2.1 Communication between client and server

Since the prototype application is web-based we have to address communication between clients and a game server. There are several possible ways for the implementation of client-server data exchange that we can use. In this section we will examine different options of client-server communication that meet the needs of the implemented application.

From the perspective of the data flow we can distinguish between *unidirectional* and *bidirectional* communication between a server and a client. In the case of unidirectional communication we're dealing with the data that flows only one way whether it be from client to server or vice versa. Conversely, in bidirectional communication data flows both ways and both sides of the communication channel can exchange the data simultaneously.
From [7] it follows that unidirectional communication is suitable for exchanging the data that does not have to be updated frequently. An example could be HTTP request-response cycle where after receiving the data there is no need to keep connection with server.
On the contrary, with bidirectional flow we can allow two-way simultaneous communication where server is capable of initiating a communication and send the data to the client when that data becomes available. This could be useful for communication in multiplayer games when one player takes an action that has to be relayed to other players. Possible implementation could be done with utilizing *WebSocket* protocol.

2.1.1 Server application interface

To enable the communication between server and client, the server should expose an API. Client could then send an API a request detailing the information it needs or to alter data on the server. At the time of writing this work, the two most popular paradigms for implementing an API are REST API and GraphQL API.

REST is an *architectural style* for providing standards between computer systems on the web, making it easier to communicate with each other. This is resource-oriented ([8] chapter 5.2.1) model where the resources are identified by unique URI (Uniform Resource Identifier). The most widely used protocol data messaging is HTTP, however REST itself is not limited to any specific protocol. For resource manipulation RESTful services use POST, GET, PUT, DELETE which are mapped to application create, read, update and delete(CRUD) operations.

GraphQL [9] is a *query language* that lets the client to specify what data it needs and makes it easier to aggregate data from multiple resources. Because GraphQL is a declarative data fetching specification and a query language, we only fetch what we need from the server by constructing our query to only include what we need. It offers more flexibility as opposed to REST APIs and is more suitable for applications with many different domain objects.

In the implemented application there will not be many domain objects which we might want to filter from and implementing GraphQL API would be complicated for this purpose. While RESTful API would be enough to create simple endpoints for authentication or fetching personal statistics.

2.1.2 Techniques for real-time communication

Information from the sever is often times needed instantly. For this use case there are different ways for client-side to interact with server-side. Here we will take a look at the common techniques used for implementing real-time communication on the web. In general, these techniques could be split into two groups: *client pull* and *server push*. Client pull asks the server for updates at certain regular intervals, whereas with server push the server is proactively pushing updates to the client.

Short Polling

Short polling is a technique that utilizes HTTP protocol in a way that client makes regular requests to the server at fixed delays. In case server does not have the data client requested the response will be empty. This approach may be viable for very small services, but generally it has a lot of drawbacks such as creating traffic and response delays.

Long Polling

Long polling provides a workaround to the delay in receiving data. In this method, the server receives a request but does not respond to it until it gets new data from another request. When new data appears - the server responds to the request with it. Long polling is more efficient than pinging the server repeatedly but it has some downsides. With long polling the server architecture must be able to work with many pending connections. Additionally, requests can time out and new request need to be issued periodically.

Server-Sent Events

Another technique for sending messages is the Server-Sent Events API [10], which allows the server to push updates to the client by leveraging the JavaScript EventSource [11] interface. It opens a persistent, unidirectional connection with the server over HTTP using a special text/event-stream header and listens for messages, which are treated like JavaScript events.

WebSockets

The WebSocket protocol, described in the specification [12] provides a way to exchange data between browser and server via a persistent connection over TCP. Although the communication still starts off as an initial HTTP handshake [13], it is further upgraded to follow the WebSockets protocol (i.e. if both the server and the client are compliant with the protocol). Once the connection is upgraded, the protocol switches from HTTP to WebSocket, and while packets are still sent over TCP, the communication now conforms to the WebSocket message format.

Real-time communication is an omnipresent functionality in modern applications. There are different techniques that allow us to implement real-time communication in web applications such as polling or server-sent events. However, these models have their disadvantages and do not provide complete two-way communication. WebSocket protocol solves many problems of the alternate models as well as provides smooth biderectional communication where the data can be sent and received freely without any requests. Using WebSockets appears to be the most suitable solution for real-time multiplayer games and would be a communication technique of choice for real-time communication in the implemented application. We will also use Socket.io [14] library which uses WebSocket as a transport and adds some metadata to each packet. Using this library will simplify the setup compared to raw WebSockets, has simpler and more convenient API, and brings many advantages such as disconnection detection, ability to create separate communication channels that share the same connection.

2.2 Proposed architecture

A system architecture is the conceptual model that defines the structure of the system. In the implemented application we will use three-tier architecture (see figure 2.1) to separate it into three conceptual layers. According to [15] this allows more cohesive designs of each aspect, and makes these designs easier to implement. Here we will introduce each layer of the application as well as their responsibilities.

2.2.1 Client

The client side of the application represents user interface, often called presentation layer, and is responsible for showing information to the user and interpreting user's commands. It also communicates with the server using HTTP and WebSocket protocol for data manipulation. Later in 2.4 we will describe the technologies used for implementation and in 3.1 deep dive into the code structure.

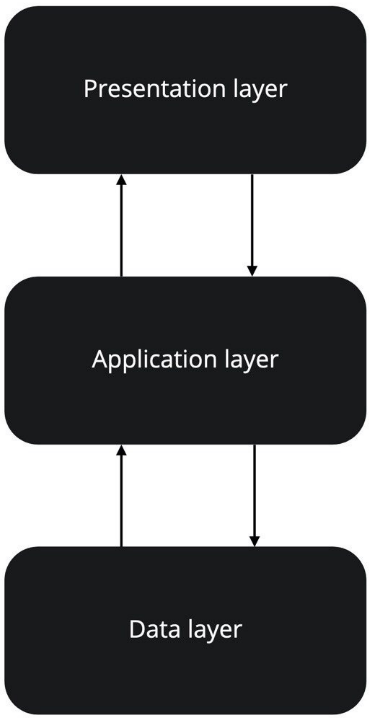


Figure 2.1: Three-tier architecture

2.2.2 Server

The server side is represented by application layer in three-tier architecture and defines the jobs the application is supposed to do. This tier also contains the functional business logic which drives an application's core capabilities. The server provides a REST API and WebSocket API for communication with the client side and is responsible for interacting with database by performing database queries.

2.2.3 Database

The database is represented by the data layer in three-tier architecture and includes the data persistence mechanisms and the data access layer that encapsulates the persistence mechanisms and exposes the data.
When it comes to choosing a database we usually decide between two options: *relational* and *non-relational*. These are also usually referred to as SQL and NoSQL (Not only SQL) databases and to determine the kind of database we will use primarily depends on the way the data is going to be structured. SQL databases have a definite schema structure which

contains tables and each table is comprised of columns and rows. Columns contain the the column name, data type and any other attribute for the column. Rows contain the records of data for the columns. NoSQL databases, on the contrary, are designed to be more flexible and could be either document-based, key-value pairs, graph databases or wide-column stores. This allows to store data unstructured data in many different ways without the need to define the structure first.
For the purposes of the implemented application SQL database would be a better fit because it is better suited for complex queries and are meant to work with pre-defined structures (see figure 2.2). Finally, the application does not have to stick to one type of database and could always be enhanced by other type of database for storing different data, but this is not necessary for the prototype.

2.3 Database schema

The idea is to design a database model that would store all the data related to a single game instance. The game is played via website, while the database is used to store information about players' actions and success rates. In this section we will describe each entity with their corresponding attributes and illustrate the domain model (see figure 2.2).

2.3.1 User

User table is where we will store a list of all our registered players. All attributes in the table are mandatory. The `username` is the unique key. Only one user can use a username and each username has to be unique. The `password` property is needed for user authentication and is stored in an encrypted form. Finally, `points` property represents user's overall score in all the games and is used for statistics.

2.3.2 Game

The game entity represents the games played by users. It contains `category`, `difficulty` and `numOfQuestions` properties that describe the individual game and are the properties that user can define when creating a new game.

2.3.3 Participant

The participant entity is where we will connect users and games. The properties `user` and `game` are references to the **User** and **Game** tables. Together they form an alternate (unique) key of the table. The `score` attribute is used to store a score player achieved in a given game. In the scope of the implemented application we will have three possible scores: 2 for a win, 1 for a draw and 0 for losing the game. The default score value is 0, so if a user loses due to disconnection we will already store the correct score. That is also why this attribute is mandatory.

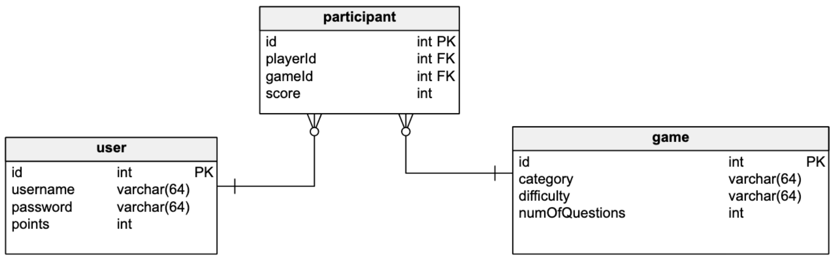


Figure 2.2: Entity relationship diagram

2.4 Technology stack

This section describes the technologies used for the implementation of the application. It also reviews possible alternatives to selected technologies.

2.4.1 Implementation language

For the implementation of the application I have decided to use TypeScript [16] on both client-side and server-side, which is a language developed and maintained by Microsoft Corporation. It is a *superset* of JavaScript and contains all of its elements. TypeScript provides an *optional type system* for JavaScript along with planned futures from JavaScript edition to current JavaScript engines. As mentioned in [17] the main motivation for bringing types to JavaScript is that they have the ability to to enhance code quality and understandability.

2.4.2 Client

There are many frameworks and libraries available for frontend development. One of the most popular libraries for building UIs is React [18] which will be a technology of choice for implementing the client-side part of the application. The main building block of React is a component which is essentially a class or a function that optionally accepts inputs, usually called *props* (short for properties), and returns a React element that describes how a section of the UI should appear. Props that a component receives are only being passed from the parent component to its children and this core pattern used in React is called **one-way data flow**. Usually, the props that are passed down the component tree have to be defined and stored, and for that we use *state* which is another important concept in React. The state could be seen as the representation of the of the app at a given point in time. Each component can have its own local state which can be updated. Even though, managing application state only with React is entirely possible, it can quickly become ineffective due

to scaling complexity and performance issues.
In the implemented application we would make use of local component's state as well as add a library for *global* state management called Redux Toolkit [19]. This would allow us to have a state that is shared between all the components and not localized to only one component.

2.4.3 Server

For the implementation of the server-side part of the application I decided to choose Node.js [20] runtime environment. The core idea behind Node.js is using non-blocking, event-driven IO to remain efficient and lightweight in the face of data-intensive real-time application which makes it perfect fit for the implemented application.
Event-driven programming is a programming paradigm where the flow of the program is determined by events [21]. Under the hood Node.js listens to events and invokes callbacks in an *asynchronous* manner. To do this, Node.js uses event loop (see figure 2.3) which is essentially a loop for handling events that usually runs forever.

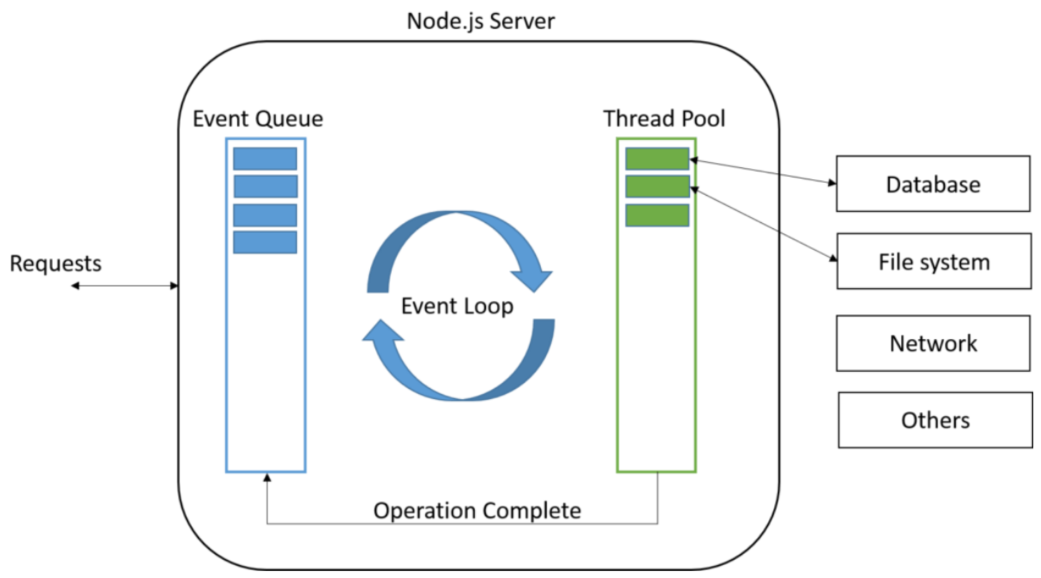


Figure 2.3: Node.js event loop

Source: https://www.pabbly.com/tutorials/node-js-event-loops/

2.4.3.1 Node.js frameworks

Node.js framework is a set of tools, libraries and helpers that simplify the process of building the base layer of the web application. We will review the most popular frameworks and select one for the implemented application.

Express.js [22]

Express is a popular flexible Node.js framework that provides a set of useful and performant set of features for creating scaleable web applications. It was launched in 2009 and is an established project with big community support. It offers a simple way to setup a server and promotes code reuse with its built-in router. Express has a rich API that permits to configure different routes to send or receive requests between the frontend and database. It is also fast and has many built-in functions, utility methods and middlewares easier development of robust API.

Koa [23]

Koa is a framework that was created by developers behind Express. It is a minimal framework that does not bundle any middleware within its core hence being a very lightweight framework that could be easily extended. Main selling points of Koa is leveraging modern JavaScript syntax such as `async` functions and generators. However, this frameworks still in development and some its features are not supported by Node.js yet.

Hapi [24]

Hapi is a rich framework for building web applications and services. It is a configuration-driven pattern, traditionally modeled to control web server operations. It also provides provides a deeper control over request handling. Hapi has many bundles with a big set of extensions for authentication, authorization and validation. This framework is suitable for enterprise applications but could include too much boilerplate for smaller applications.

2.4.3.2 Evalution of Node.js frameworks

In 2.4.3.1 we can see that Express and Koa are very similar frameworks, whereas Hapi comes with its own structure, ecosystem and set of features. For implementation of the application I have decided to use Express.js framework because it is the most stable framework for creating server-side applications using JavaScript at the moment of writing this work. Both Express and Hapi are very reliable frameworks but Express has better community support, more open-source packages for enhancing applications and allows more flexibility.

2.4.3.3 Database

As the database we would use PostgreSQL [25] which is an open-source object-relational database system developed by PostgreSQL Global Development Group. In the application we will also need to have a way to interact with the database. For that we can use raw SQL queries or Object Relational Mapper (ORM) [26]. Raw SQL queries can be hard to construct but these tend to be more performant. ORMs, on the other hand, are designed to simplify the process of managing databases and map entities from the code in a relational database. Because the database of the application is not complex and performance is not a key requirement, we would benefit more from using ORM. This would allow us to have less repetiotion in the code, abstract away SQL queries and still have small performance tax because many ORMs provide objects caching. The technology of choice will be TypeORM [27] which is an open-source ORM tool with rich set of features and TypeScript support.

2.5 Summary

In this chapter we have defined the architecture of the application which will be three-tier. We have defined the responsibilities of each layer of the application as well as the database schema. This chapter also described different techniques for implementing real-time communication between server and client. Based on this analysis we have selected the protocol for the communication. Finally, the technologies used for the implmentation were introduced.

Chapter 3

Implementation

In this chapter we will describe the implementation of the application. We will review the project structure and take a look at the core modules of the application. This chapter also documents client-server communication using REST API and real-time communication using WebSockets. Finally, we will take a look at the UI implementation and application configuration.

3.1 Project structure

This section describes the organization of application's project structure. The correct organization will help us to avoid code duplication, improve stability and could potentially help us with the future scaling of the application.

3.1.1 Server code organization

In the following directory tree we can see the code organization of the server-side application. We will explain the responsibility of each component and the data processing flow in 3.2.

```
/
├── dist ........................................ compiled version of code
└── src
    ├── config ............................ environmnet variables and configuration
    ├── controllers ........................... route controllers for API endpoints
    ├── interfaces ................................................ shared interfaces
    ├── middleware ................ fuctions for intercepting requests and responses
    ├── models ...................................................... database models
    ├── routes .................................................. REST API endpoints
    ├── services ........................................ application business logic
    ├── sockets ........................... tools for managing WebSocket connections
    ├── utils ......................................... set of reusable functions
    └── index.ts ................................... entry point of the application
```

Dist stands for distribution and is a set of compiled and minified modules that are used for deployment of the application. **Src** stands for source and is a folder that contains all the modules of the application.

3.1.2 Client code organization

In the following directory tree we can see the code organization of the client-side application.

```
/
├─ public ............................ index.html page with compiled version of code
└─ src
   ├─ api ........................................ functions for calling server API
   ├─ app ............................................ main application container
   ├─ assets .............................................. application images
   ├─ components .................................. generic reusable components
   ├─ features ................................ domain features of the application
   ├─ theme ..................................................... color palettes
   ├─ utils ........................................... set of reusable functions
   └─ index.tsx ................................... entry point of the application
```

Similar to **dist** in server code structure, **public** folder contains `index.html` file which contains the compiled React code in `script` tag. **Src** contains application's modules and is separated into a few folders itself. The directory tree described above is a high-level overview of the responsibilities of each folder, but the most important folder is features. It contains the components with state and also Redux state separated into *slices* [19]. Each slice could be seen a piece of global state of the application and includes the initial state values, defines how the state is being updated and defines what specific actions result in state updates.

3.2 Communication channels

As described in chapter 2, we are going to utilize two different communication protocols for the application implementation. Part of the application would be providing REST API for the client and the other part would be communication via WebSocket protocol. While REST API would be used for user authentication (see 3.3) and personal statistics, WebSockets will be used for handling games.

The separation of the communication channels would allow us to create different modules for handling application logic and split the implementation into well-defined pieces. Figure 3.1 shows how the server application actually consists of two different servers: HTTP server for handling REST API requests and WebSocket server for handling sockets connections and managing them. One big difference between these modules of the application is that HTTP server communicates with the database for retrieving and updating the data, although WebSocket server handles data manipulation on the server within its memory. This is done to not add unnecessary complexity to the database model e.g. adding new attributes to the models. Another reason for that is to have faster updates for real-time features of the

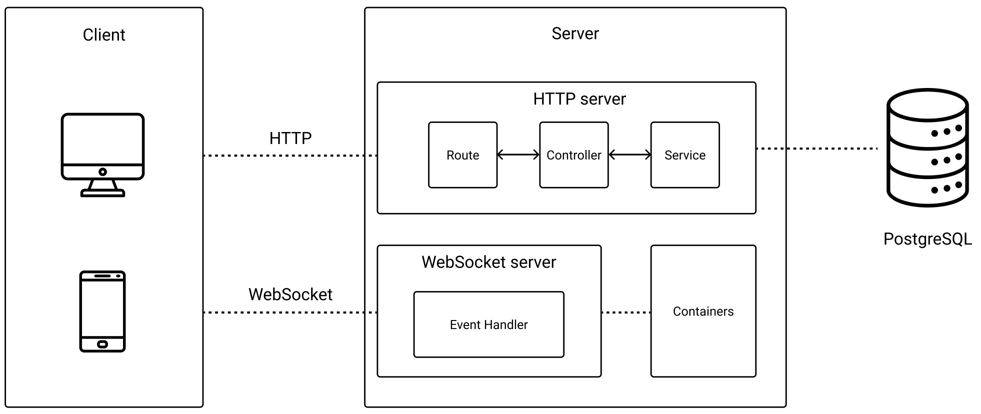


Figure 3.1: Separation of communication channels

application because we are not going to execute SQL queries for each update which could decrease the performance of the application since these updates would be taking place a lot more often than regular HTTP requests. We will take a more detailed look at handling WebSocket connection in 3.2.2.

3.2.1 HTTP server

As mentioned earlier, HTTP server of the application is responsible for handling REST API request response cycle. Here we will describe the flow of this cycle and how the application handles it.
When a user makes a request to the server the first module that is responsible for procesing the request is **routes**. This module holds the names of endpoints paths and calls corresponding function when certain enpoint is accessed. The function that is called by router is called a **controller** and accepts request and response objects [28] as parameters, processess the request and returns response to the user. The individual processing of the request is not being done directly within the controller but is being delegated to **services** which is a collection of classes with clear purposes [29]. The service layer contains business logic of the application and is responsible for communicating with the database. The described flow of the request response cycle is depicted on figure 3.2. In table 3.1 we can see the available REST endpoints of the application.

Method	Resource	Description
POST	/auth/signup	Create new account in the system
POST	/auth/login	Login to an existing account
GET	/auth/me	Validate user's JWT token
GET	/user/points/:id	User's points statistics
GET	/user/gamesData/:id	User's overall games statistics
GET	/user/recentGames/:id	User's recent games and results
GET	/user/categoryStatistics/:id	User's statistics by categories

Table 3.1: Application REST API endpoints

3.2.2 WebSocket server

WebSocket server is responsible for handling real-time communication processes in the application. As described earlier in this section, this server does not handle data manipulation using the database but only uses server's memory. However, this is not the case for all the games because we still need to have an option of saving games and results in a persistent manner. For that purpose we would be using **containers** which are represented by an array data structure. Because in JavaScript array is mutable data structure, we are going to make use of functional programming principles for keeping the containers immutable [30]. For implementing immutable data structures and processing the data in the containers in an immutable fashion we will use Ramda [31], which is a JavaScript library for functional programming. Here we will describe the processes of handling real-time communication within the application and define the events the application supports.

3.2.2.1 Online users

One of the requirements (see 1.2) of the application is the ability to show the list users who are currently online. For this purpose we have a container that is responsible for storing user objects with their respective id and username. Storing this data would allow us to identify each user and not displaying multiple users with the same username.
When a new user connects to the application via WebSocket protocol, the server identifies this user and pushes a new object to the container. Additionally, the server emits an event called `players` which sends a list of all the connected users stored in the container. We also need to control the updates for already connected users, this is done by sending a list of online users whenever the container is updated which happens when a new user connects or disconnects (this is handled by Socket.io library `disconnect` event).

3.2.2.2 Managing a game

For users to play games against each other we need to have an option for creating one. This is done by sending an event from the client called `game created`. After receiving this event the server checks if a user does not have an already created game or if the game with the parameters that user has defined could not be created. Because the application uses open-source third party API [32] that generates questions for a particular game, there could be situations when it is responding with error code in case it could not provide questions based on these parameters. In either case if the game could not be created server responds with an error message that is displayed to user.
If the game is created successfully then this new game instance is being added to `games` container. After the game is created server emits an event `new game` that notifies other users about newly created game. After new game is being sent to other users they can join it by sending an event `join game`. This adds a new opponent to the game instance and removes the game from the list of available games other users can join because one game could be only played by two users. Also, we have to remove all the games previously created by user who joined the game.
Before users start playing a new game they have to confirm that they are ready by sending an event `player ready`. After both participants of the game are ready, server sends a new

question and repeats this process until the last game question is answered. Moreover, when both participants confirm that they are ready to play the game is added to the database so that we have a record of that game for later use in statistics. When all the questions were sent, server emits an event `game ended` that also contains data about the winner.

3.3 Authentication

In this section we will review two common types of authentication on the web. We will also compare them and describe how authentication is being implemented in the application. Today there are two common ways to implement the authentication on the web which are usually referred to as *stateful* and *stateless* authentication.

3.3.1 Stateful authentication

Stateful authentication is a way to verify users by having the server or backend store much of the session information, such as user properties. In stateful authentication, whenever the client sends a request to the server, the server must look up session information such as user properties and match them against its identity provider (IdP). This usually involves receiving the client's reference ID and marching it against stored authentication data that is has on all users (see figure 3.2). Cookies are a common way for storing session identifiers on the client.

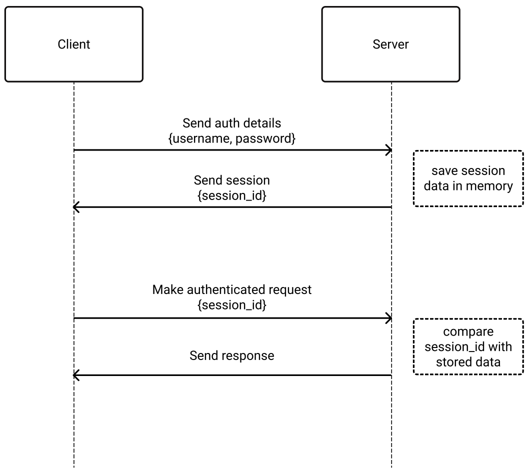


Figure 3.2: Stateful authentication flow

This way of authentication has some disadvantages. Because the sessions are stored in the server's memory, scaling becomes an issue when there is a huge number of users using the system at once.

3.3.2 Stateless authentication

Stateful authentication, also often called token based authentication, is one in which user state is stored on the client. In token based authentication the server creates JWT [33] with a secret and sends the JWT to the client. The client stores the JWT and includes JWT in the header with every request. The server would then validate the JWT with every request from the client and sends response (see figure 3.3). Even though token based authentication is better at scaling and requires no special table in database, it also has disadvantages e.g. JWT tokens are easier to compromise and require some management for expiration time and refreshing.

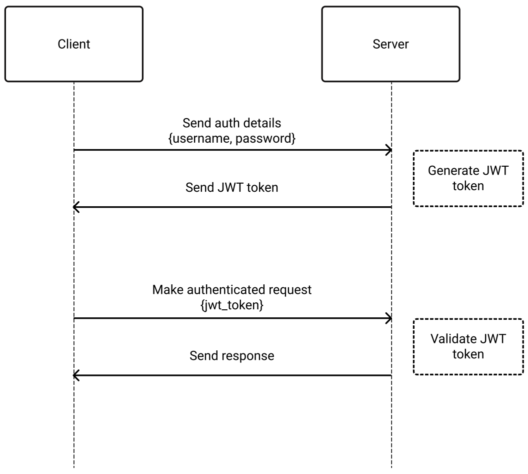

Client
Server
Send auth details {username, password}
Generate JWT token
Send JWT token
Make authenticated request {jwt_token}
Validate JWT token
Send response

Figure 3.3: Token based authentication flow

3.3.3 Choosing authentication method

For the implemented application I have decided to use token based authentication. This will allow us to keep the database cleaner and have less queries in the backend. Additionally, we may embed custom data in the token for simplifying the flow of certain operations. Finally, it will also let us to secure our WebSocket connections without the need to implement separate authentication strategies.

3.4 User Interface

User Interface (UI) is the point of human-computer interaction and communication in a system. Design of the UI focuses on anticipating what users might need to do and ensuring that the interface has elements that are easy to access, understand, and use to facilitate those actions. For designing UI of the implemented application we would want to set certain rules to consider:

- Interface should be simple and clean
- Interface should be consistent and use common UI elements
- Interface should be available in dark and light mode

For creating the UI of the application I have decided to use open-source library React Rainbow [34]. This will provide a collection of styled React components that are tested and will help us to implement the application interface according to the rules defined above. Nonetheless, we will still need to create application layout that will be responsive [35] and for that we will use CSS media-queries. In addition, we will also implement dark theme for the application which is not only a design trend but also provides a better user experience. For general overview of the application UI and better understanding of how a user will interact with the application we will illustrate (see figure 3.4) general overview of the application screen flow. Later we will take a closer look at the individual screens of the application.

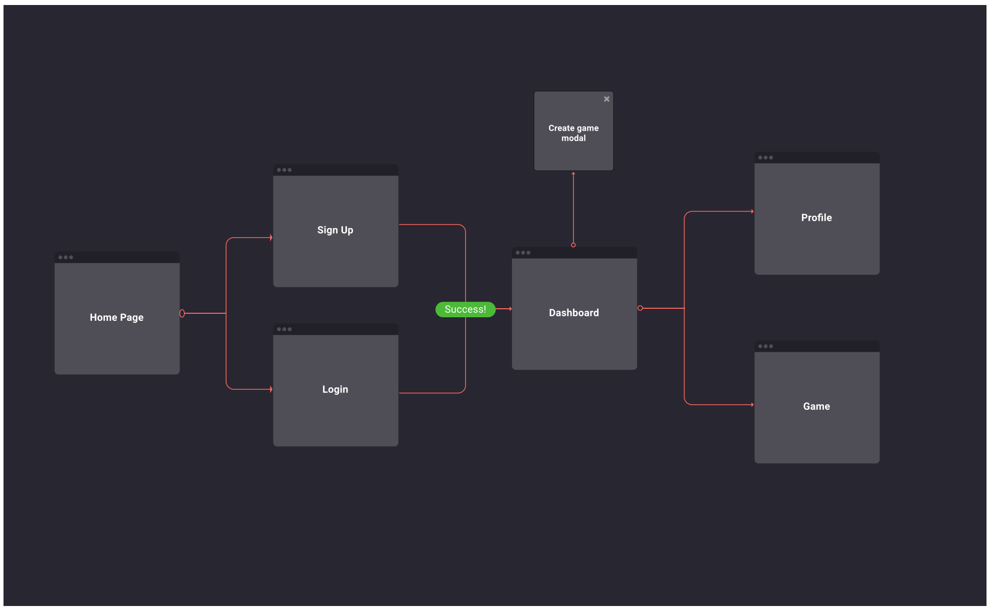


Figure 3.4: User flow of the application

3.4.1 Application screens

3.4.1.1 Home page

Home page (see figure 3.5) represents an entry point of the application for an unauthenticated user. It serves as a welcome page for new users and contains a description of the primary features of the application.

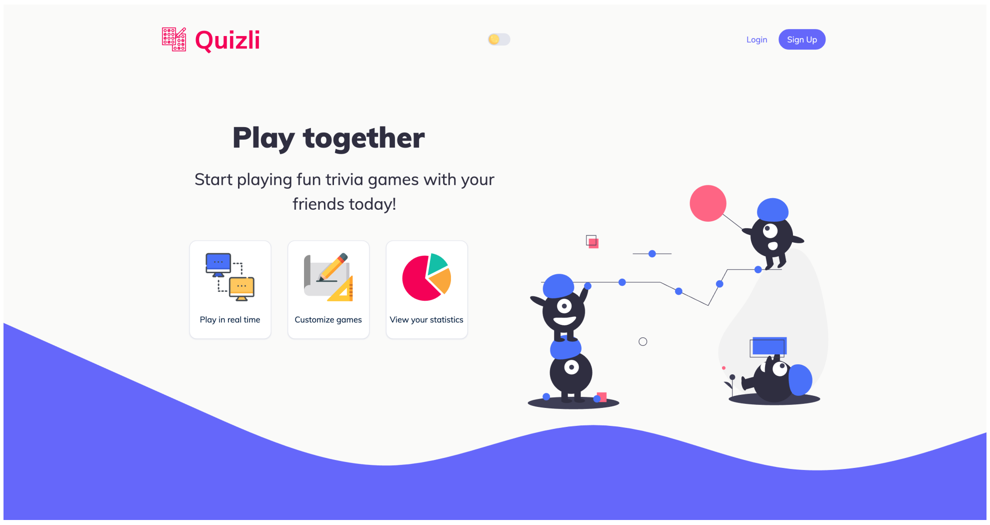


Figure 3.5: Preview of application home page

3.4.1.2 Sign Up page

Sign Up page (see figure 3.6) allows new users to create an account by filling and submitting the form.

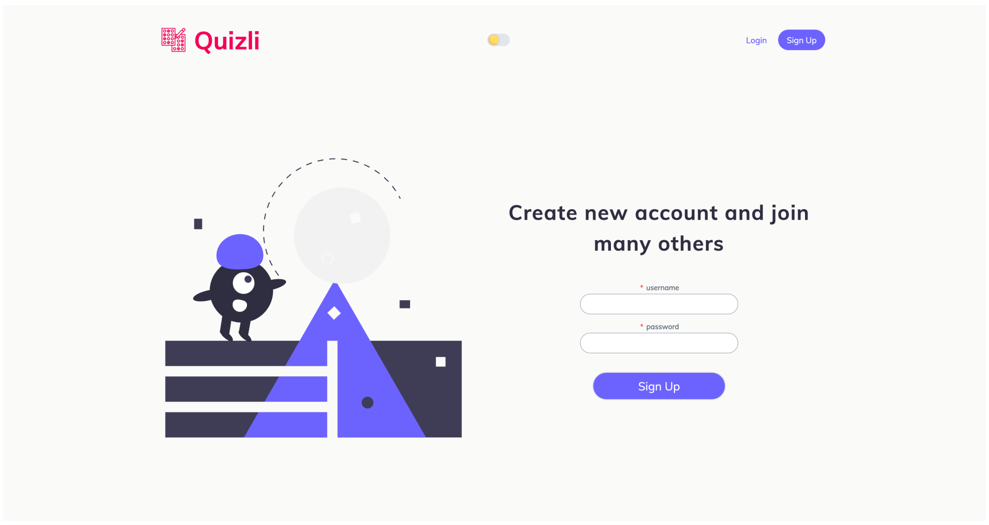
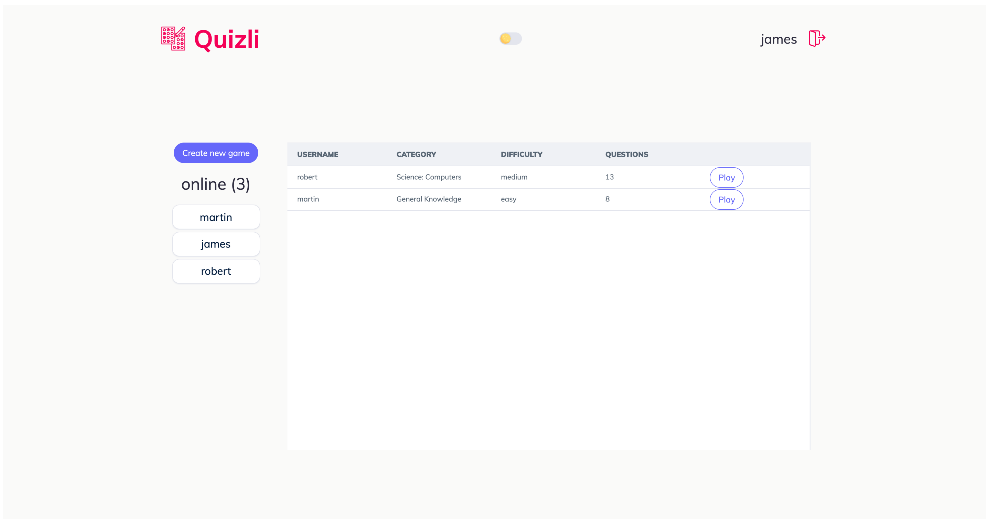


Figure 3.6: Preview of application Sign Up page

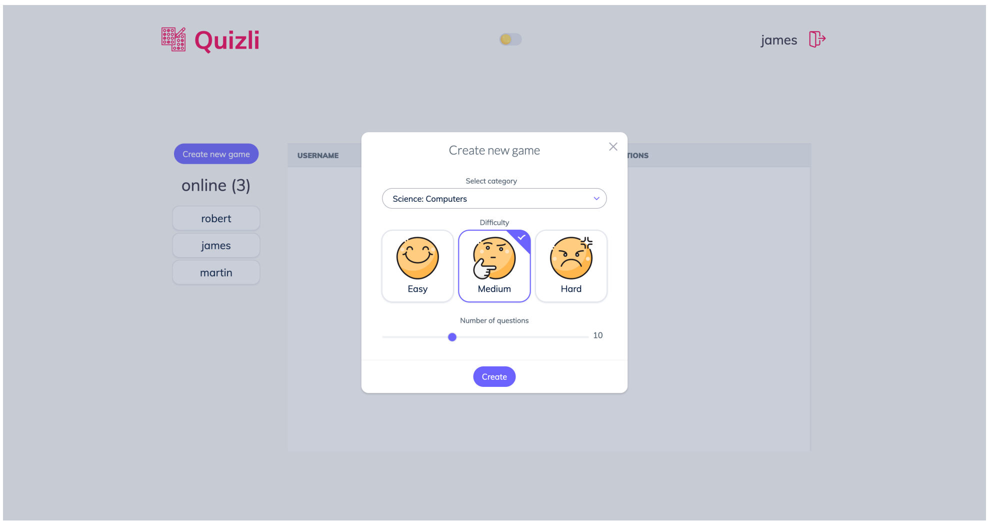
3.4.1.3 Dashboard page

Dashboard page (see figure 3.7) is the main page for authenticated users. Here, users can see a list of online users along with a list of games they can join. It also contains a button for creating a new game that opens a modal window with the corresponding form (see figure 3.8).



USERNAME	CATEGORY	DIFFICULTY	QUESTIONS	
robert	Science: Computers	medium	13	Play
martin	General Knowledge	easy	8	Play

Figure 3.7: Preview of application Dashboard page



Select category: Science: Computers

Difficulty: Easy [✓ Medium] Hard

Number of questions: 10

Figure 3.8: Preview of modal window for creating games

3.4.1.4 Profile page

Game page (see figure 3.9) is only displayed when a user is playing a game and contains current question of the game along with options for answers. On the sides it also displays personal progress of a user and his opponent. At the end of a game the page displays the results containing the score of each participant and a username of a winner in case one of participant scored more points than the other or text *It's a draw!* if both participant have scored the same amount of points.

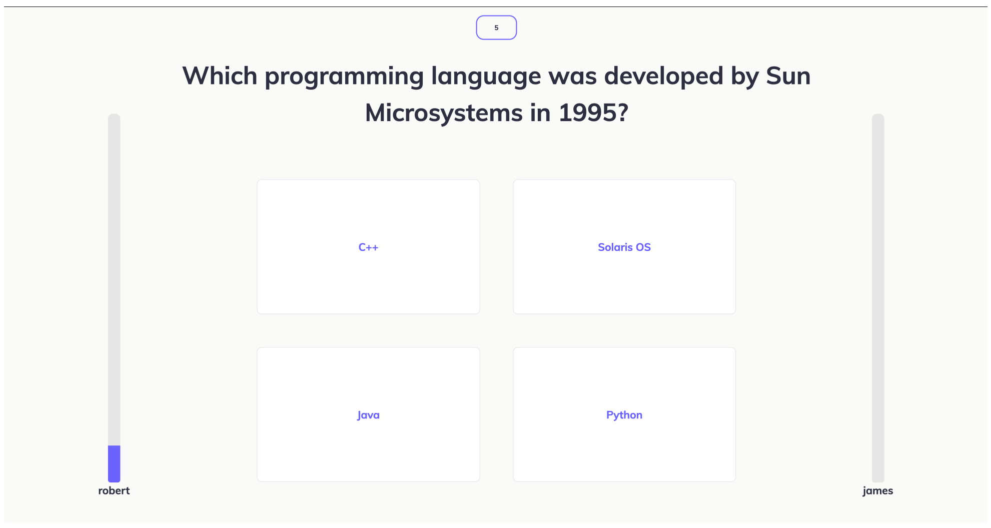

5
Which programming language was developed by Sun Microsystems in 1995?
C++
Solaris OS
Java
Python
robert
james

Figure 3.9: Preview of application Sign Up page

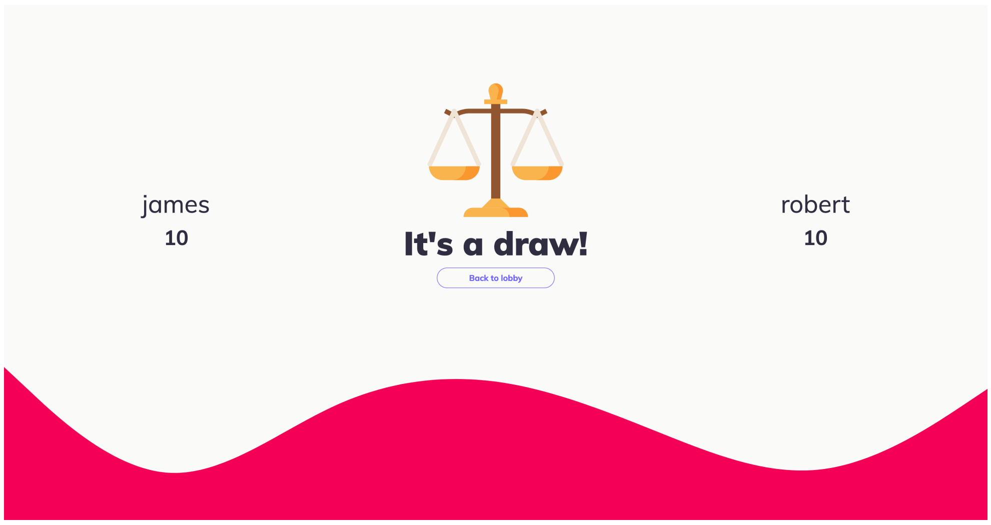


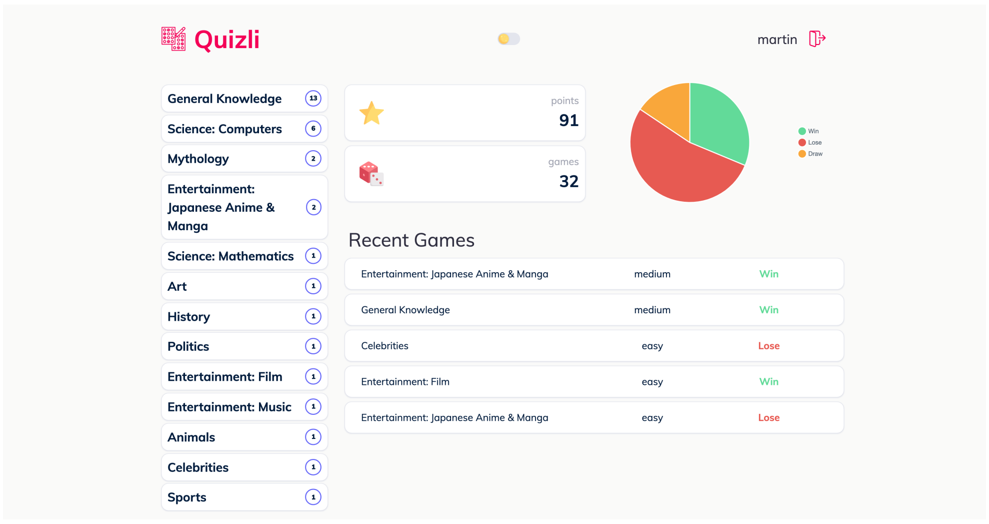
Figure 3.10: Preview of application Sign Up page

3.4.1.5 Profile page

Profile page (see figure 3.11) displays user's statistics and achievements. On the left it shows a list of categories with corresponding number of games for each category that a user has ever participated in. On the right of the list it displays a total number of games, points, chart with win-lose-draw ratio and a list of recent games with their results.

3.5 Building the application

Build process includes multiple steps that optimize the application for the best performance. In JavaScript applications this process usually consists of the following steps: transpiling,


Quizli

martin

General Knowledge	13
Science: Computers	6
Mythology	2
Entertainment: Japanese Anime & Manga	2
Science: Mathematics	1
Art	1
History	1
Politics	1
Entertainment: Film	1
Entertainment: Music	1
Animals	1
Celebrities	1
Sports	1

points 91

games 32

Recent Games

Entertainment: Japanese Anime & Manga	medium	Win
General Knowledge	medium	Win
Celebrities	easy	Lose
Entertainment: Film	easy	Win
Entertainment: Japanese Anime & Manga	easy	Lose

Figure 3.11: Preview of application Profile page

bundling, minifying and packaging. Transpiling lets us the advantage of the newest language features with maintaining browser compatibility. Moreover, because TypeScript is not compatible with modern browsers at the moment of writing this work, it has to be transpiled to plain JavaScript. Bundling is responsible for turning all the modularized files of the application into one executable and deliverable file. Minifying reduces the final size of the application by removing unnecessary or redundant data without affecting how the resource is processed by the browser. Finally, packaging is responsible for putting the bundled and minified version of the application into specific folder or file.

3.5.1 Building the client application

Client-side application requires the build process before every run in a browser since it is written in TypeScript and utilizes language features that are not natively supported by modern browsers. For buidling the application I have used pre-configured module bundler Webpack [36], which comes with `create-react-app` package that was used for implementing the client-side of the application. Besides a production build, this package also allows to build the application in development mode with page reload after each edit.
Production build of the application can be done by running `yarn build` which creates a bundled and minified application in the directory `build`. This allows us to create a production build of the application and deploy it because the output of this command generates static files with one main HTML file called `index.js`.

3.5.2 Building the server application

Server-side of the application is meant to be run in Node.js environment (see section 2.4.3). For building the application that will be running in Node.js environment, we will use Webpack with setting the `target` property in `webpack.config.js` to `node`. To build and run the application we will also need to create a `.env` file which contains all the environment variables. After setting the environment, the application can be build by running `yarn build` command. This will compile TypeScript code into executable and minified JavaScript.

3.6 Summary

This chapter describes implementation details of the application. It provides an overview of the code structure and demonstrates how core modules of the application work. Furthermore, this chapter documents client-server communication implementation exploring the idea of separation of communication channels using REST API and WebSockets. Finally, it also highlights the implementation of the UI and describes application build process.

Chapter 4

Testing

In this chapter we will focus on testing the implemented application. After performing the tests we will evaluate the implemented application prototype and explore the potential improvements.

4.1 Functional testing

Functional testing is a type of software testing where the system is tested against the functional requirements (see 1.2) to confirm that the functionality of a system behaves as expected. This type of tests helps us to test the application from the user's perspective.
For basic functional testing of the application we have used Cypress [37] which is a JavaScript framework for end-to-end testing. However, Cypress does not support multiple tabs or browsers for testing collaboration of the users, therefore to test functionality of the application we have launched the server locally and simulated multiple users using multiple browsers. These are individual steps to take for testing the application on localhost:

1. Run server on localhost using command `npm run start`
2. Run client application on locahost using command `npm run start`
3. Open two different browsers and login with different accounts
4. Create new game in one of the browsers
5. Another user should see a game in his browser and be able to join the game
6. After playing the game users are able to see the results
7. Users can see their statistics by clicking on the username in the top right corner

The test was taken a few times with different deviation from the scenario described above and in all cases the application was behaving as expected. We have tested that users can login to their accounts, create new games with customization, join the games other users have created, play against each other in real-time and view their personal statistics. The only disadvantage of this test is testing client and server connection on the same machine which may differ in real world scenario when there could occur potential network problems.

From the above test it follows that the application is capable of providing the same functionality as defined in 1.2. We have verified that the application is acting in accordance with the requirements and validatad that it works as intended.

4.2 Usability testing

Usability testing is a method used to evaluate how easy a system is to use. It provides a way to see how easy it is to use the application by testing it with real users. The goal is to reveal areas of confusion and uncover opportunities to improve the overall user experience. Usability testing starts with explaining the goal of the test to a participant which is followed by completing the test. During the test we will observe the participant's behaviour and listen for a feedback. After a test is completed we may also ask the followup questions and, finally, collect the insights and findings.

4.2.1 Task scenarios

For usability testing we want to introduce task scenarios that reflect functional requirements 1.2. Before a participant starts the test we would explain the objective and the goal of test. Each task scenario contains the goal, conditions, time limit and individual steps to take. After a participant completes the test and the followup questions are answered, we would process the feedback and compare the result with the expected outcome of the task.

T1 — Creating a new account
Goal: New user can create an account
Conditions: User has no account in the system and home page is displayed
Time limit: 5 minutes
Task steps:

1. Click on *Sign Up* button
2. Fill the registration form
3. Submit the form by clicking on Sign Up

Expected outcome of the test:

1. New account is created in the system
2. New user is logged in
3. Application displays dashboard and shows username in the top right corner
4. Application displays username in the list of online users

T2 — Logging into the application
Goal: User with existing account can log in
Conditions: User has an existing account and home page is displayed
Time limit: 5 minutes
Task steps:

1. Click on *Login* button
2. Fill the login form
3. Submit the form by clicking on Login

Expected outcome of the test:

1. User is logged into the system
2. Application displays dashboard and shows username in the top right corner
3. Application displays username in the list of online users

T3 — Creating a game
Goal: Authenticated user can create games and customize them
Conditions: User is logged in and dashboard page is displayed
Time limit: 5 minutes
Task steps:

1. Click on *Create new game* button
2. Change different game parameters
3. Click on *Create* button to submit

Expected outcome of the test:

1. Modal window with game parameters automatically closes
2. New game is displayed in the list of active games on dashboard page

T4 — Playing a game
Goal: Authenticated user can join other available games to play
Conditions: User is logged in and at least one game is displayed in the list of active games
Time limit: 15 minutes
Task steps:

1. Click on *Play* button next to any available game in the list
2. Confirm ready state by clicking on *Ready* button
3. Answer questions as they are displayed on the screen

4. View game points during the game
5. View game results after it ends
6. Return to dashboard page

Expected outcome of the test:

1. Application connects two users for playing a game
2. Points for correct answers are calculated and displayed during the game
3. Application displays new questions with the same time interval
4. Application displays game results at the end of the game
5. Application saves a game and results for statistics

T5 — View personal statistics
Goal: Authenticated user can view games history and statistics
Conditions: User is logged in at least one game is displayed in the list of active games
Time limit: 5 minutes
Task steps:

1. Click on *Play* button next to any available game in the list
2. Confirm ready state by clicking on *Ready* button
3. Answer questions as they are displayed on the screen
4. View game points during the game
5. View game results after it ends
6. Return to dashboard page

Expected outcome of the test:

1. Application connects two users for playing a game
2. Points for correct answers are calculated and displayed during the game
3. Application displays new questions with the same time interval
4. Application displays game results at the end of the game
5. Application saves a game and results for statistics

4.2.2 Participants

Participants are realistic users of the application. During the test they are asked to narrate and thoughts as they perform the tasks. To uncover the majority of the most common problems in the application we will perform the qualitative usability testing. This type of testing focuses on collecting insights and findings about how users use the application.
Four people have participated in usability testing of the implemented application. To protect personal privacy of the participants their names were replaced by a numerical equivalent. Each participant has completed all the tests defined in 4.2.1.

Participant 1 game designer, 25 years old, interested in programming and data science. Has a good command of advanced modeling and rendering software. Often plays real-time applications on the web, mostly table or card games.

Participant 2 student at Scholastika, 21 years old, studying graphical design and interested in art. Has experience with collaborative interface design tools but has never played multiplayer games on the web.

Participant 3 mobile developer, 26 years old, his hobbies are competitive sports and games. Has experience with advanced web services and a good command of developing complicated software. Mostly plays multiplayer games on mobile phone and has a good understanding of real-time games mechanics.

4.2.3 Test results

This section describes the results of the individual tests that were completed by the participants. Before each test a participant was presented a goal of the test and how the testing experiment was going to take place. Additionally, before each test application prototype was set to the state of test conditions. Participant were not receiving any help to achieve test goals except for the situations when it was needed to complete the test.
After all the tests were completed by a participant, he was asked to rate the experience on scale 1 — 5, where 1 means that UX is unintuitive (participant was not able to finish the test without help), and 5 means that UX is intuitive and easy to use (participant was able to finish the test with no problems).

Participant 1

Participant had no problem with creating an account (T1), logging into the application (T2) and creating a game (T3). When playing a game (T4), participant mentioned that it was not clear how game points are calculated and pointed out that countdown timer for each question will improve game experience. Participant would also prefer to have a more detailed overview of personal statistics (T5), more specifically having an option to view statistics for each recent game such as number of scored points, correct answers and opponent username. The evaluation of test results can be seen in table 4.1.

Test scenario	Rate (1 - 5)
T1	5
T2	5
T3	5
T4	3
T5	3

Table 4.1: Usability test results of participant 1

Participant 2

Participant pointed out good implementation of the interface and seamless navigation through the application. No problems were discovered when going through tests on creating an account (T1), logging into the application (T2), creating a new game (T3) and viewing game statistics (T5). Some negative observations were touching the game process where it would be good to have a countdown timer for each question and getting information about correct answers. Participant was assuming that the outlined answer was correct while it was highlighting the selected one. The evaluation of test results can be seen in table 4.2.

Test scenario	Rate (1 - 5)
T1	5
T2	5
T3	5
T4	4
T5	5

Table 4.2: Usability test results of participant 2

Participant 3

Participant had no issues with creating an account (T1), logging into an account (T2) and creating new game (T3). Participant found game process to be less intuitive because individual question did not have countdown timer and were replaced by new ones unexpectedly. Additionally, it was not clear to know whether the answer was right or wrong because it was always highlighted with the same color. To view personal statistics some little help was required to find a way to navigate to profile page. The evaluation of test results can be seen in table 4.3.

Test scenario	Rate (1 - 5)
T1	5
T2	5
T3	5
T4	3
T5	4

Table 4.3: Usability test results of participant 3

4.2.4 Evaluation of usability testing

During the usability we have received a feedback on the interface of the application and evaluated whether it is easy to use it. All the participants pointed out that the application interface is intuitive and easy to navigate. Main issues for the participants were lack of countdown timer for individual questions during the game and lack of detailed statistics for each game. These issues are not stumbling blocks for using the application but can affect user experience in a bad way, therefore it is important to take the findings into account and solve these problems in the future development.

4.3 Performance testing

Performance testing is a form of testing that focuses on how a system behaves under expected workload. In this section we will focus on load testing, which is a subset of performance testing. It measures system performance as the workload increases ensuring the application can perform as expected in the production.
Because the core functionality of the application is real-time communication between users, we would be focusing on testing WebSocket connections. For testing the connection we will use Artillery.io [38], a tool for performing load testing. This tool has native support for Socket.io engine and allows to create test scenarios that could be run as a script.

4.3.1 Testing strategy

First, we would test user connections without creating games. Each test contains different amount of connected users and examines application performance under different conditions. The test scenario is the following: in the space of 5 minutes every second 3 virtual users are establishing the connection and keep it opened for 30 seconds. The results could of these test be seen in table 4.4.

Concurrent virtual users	Max	Median	p95	p99
30	2.2 ms	0.1 ms	0.2 ms	0.3 ms
50	1.4 ms	0.2 ms	0.3 ms	0.4 ms
100	2.0 ms	0.2 ms	0.3 ms	0.8 ms

Table 4.4: Overall latency distribution for connected users

We will also run similar tests for creating games on the server. Here, in the space of 5 minutes we would connect a new user which creates a new game with the probability of 80%. The results of the tests could be seen in table 4.5. These tests rather focus on the implementation of creating new games and storing them on the server efficiently under large workloads instead of supporting large numbers of connections.

4.3.2 Evaluation of performance testing

It is important to notice that the results of the tests may differ. The results could depend on machine that is generating the loading test, server configuration and network connection.

Concurrent virtual users	Max	Median	p95	p99
30	1.9 ms	0.2 ms	0.2 ms	0.4 ms
50	2.4 ms	0.2 ms	0.3 ms	0.5 ms
100	2.5 ms	0.3 ms	0.3 ms	1.1 ms

Table 4.5: Overall latency distribution for creating games

For a complete testing of the application, when it is deployed, we might use a distributed load testing approach to test server scalability, network delays and bigger workloads. Taking this into account, in our case all the measurements were made on localhost and the following machine:

- Machine: MacBook Pro (16-inch, 2019)
- Processor: 2,3 GHz 8-Core Intel Core i9
- Memory: 32 GB 2667 MHz DDR4

Results of the tests described in this section have proven that the application handles bigger workloads well without a noticeable delay in performance. We can also assume that more concurrent users of the application will not cause drastic decrease of performance after deploying and hosting the server.

4.4 Summary

In this chapter we have described testing of the application prototype. In section 4.1 we have tested the application against functional requirements and verified that specified user flows work as expected. Then, in section 4.2 we have conducted usability testing to evaluate application functionality with real users. Finally, in the last section 4.3, we examined application behaviour under a significant load from which we can understand reliability and stability of the application.

Conclusion

The primary goal of this thesis was to design, implement and test a proof-of-concept web application for multiplayer real-time quiz games. The thesis is composed of four chapters, each of them dealing with different aspect of software development life cycle.
First chapter of this thesis is introductory and outlines the concept of the application. Next, it defines functional and non-functional requirements explaining the core functionality. Furthermore, similar existing solutions are researched and evaluated.
In the following chapter studies the problem of real-time communication on the web including the research on possible techniques for its implementation. This chapter also provides an overview of the proposed architecture and illustrates the domain model of the application. Finally, second chapter introduces the technologies used for the implementation.
Chapter three concentrates on problems resulting from the previous chapter and explains individual implementation details. There, project structure is illustrated and separation of communication channels is described.
The last chapter of this thesis deals with testing of the implemented application. It is separated into three parts. Part one describes functional testing of the application and examines the correct functionality of the application based on the requirements outlined in first chapter. Part two deals with usability testing and evaluates the application functionality from a perspective of real users. Finally, last part describes performance testing strategy and evaluates the results.
The main goals of the thesis have been reached. We have researched existing solutions and technologies for implementing real-time communication between server and client. Additionally, we have analyzed different techniques and approaches used in modern web applications. Finally, a proof-of-concept web application was implemented aligned with software development best practices, providing a requirement analysis, software architecture design and software testing.

Future work

For the future work, the implemented application could be deployed to production environment with some potential improvements such as preventing a user from connecting more than once via web sockets. This is important because enabling login onto the same account multiple times could create unfair scenarios in games. Another potential problem of the application is using too many server resources for storing online users. Both of the problems could be solved by adding external cached data store. This improvement will be the next development milestone and could be easily added thanks to modular code structure and separation of concerns of the application. Furthermore, the application could be extended by a set of new features such as letting users to communicate with each other, play private games by introducing game links, implementing a detailed statistics for each game and creating PWA version of the application.

Bibliography

[1] Shawn Lawton Henry and Liam McGee. Accessibility. [online] `https://www.w3.org/standards/webdesign/accessibility`, accessed 22 April 2020.

[2] MDN. The WebSocket API (WebSockets). [online] `https://developer.mozilla.org/docs/WebSockets`, Feb 3, 2020.

[3] QuizUp - the biggest Trivia Game in the world. [software] `https://www.quizup.com/`, accessed 22 April 2020.

[4] TopQuizz - multiplayer quiz game. [software] `https://en.topquizz.com/`, accessed 22 April 2020.

[5] Kahoot! Learning games. [software] `https://kahoot.com/`, accessed 22 April 2020.

[6] QuizWitz. [software] `https://www.quizwitz.com/en/`, accessed 22 April 2020.

[7] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of Push and Pull techniques for AJAX. In *2007 9th IEEE International Workshop on Web Site Evolution*, pages 15–22, 2007.

[8] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine, 2000.

[9] GraphQL: A query language for APIs. `https://graphql.org/`, accessed 22 April 2020.

[10] MDN. Server-sent events. [online] `https://developer.mozilla.org/docs/Web/API/Server-sent_events`, Apr 18, 2019.

[11] Shawn Lawton Henry and Liam McGee. EventSource. [online] `https://developer.mozilla.org/docs/Web/API/EventSource`, Aug 1, 2019.

[12] Ian Fette and Alexey Melnikov. The websocket protocol. RFC, Standards Track, 2011. Available at `https://tools.ietf.org/html/rfc6455`.

[13] Andrew Lombardi. *WebSocket: Lightweight Client-Server Communications*, chapter 8. O'Reilly Media, first edition, sep 2015.

[14] Socket.io contributors. Socket.io. [software]
`https://socket.io/`, accessed 23 April 2020.

[15] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley, 2014. ISBN: 0321125215.

[16] Microsoft. TypeScript. [software]
`https://www.typescriptlang.org/`, accessed 24 April 2020.

[17] Basarat Ali Syed. TypeScript Deep Dive. [online]
`https://basarat.gitbook.io/typescript/`, accessed 24 April 2020.

[18] Facebook Inc. A javascript library for building user interfaces. [online]
`https://reactjs.org/`, accessed 24 April 2020.

[19] Dan Abramov and the Redux documentation authors. Redux Toolkit. [online]
`https://redux-toolkit.js.org/`, accessed 24 April 2020.

[20] NODE.JS FOUNDATION. Node.js. [software]
`https://nodejs.org/`, accessed 24 April 2020.

[21] Anton Kovalyov. *Beautiful JavaScript: Leading Programmers Explain How They Think*, chapter 10. O'Reilly Media, first edition, aug 2015.

[22] StrongLoop. Express.js. [software]
`https://expressjs.com/`, accessed 24 April 2020.

[23] Koa contributors. Koa. [software]
`https://koajs.com/`, accessed 24 April 2020.

[24] Sideway Inc. Hapi. [software]
`https://hapi.dev/`, accessed 24 April 2020.

[25] The PostgreSQL Global Development Group. PostgreSQL. [software]
`https://www.postgresql.org/`, accessed 24 April 2020.

[26] Brian Cline. What is ORM? [Online]
`https://www.brcline.com/blog/what-is-orm`, Mar 7, 2018.

[27] TypeORM Contributors. TypeORM. [software]
`https://typeorm.io/`, accessed 24 April 2020.

[28] StrongLoop, Inc. Express documentation. [Online]
`https://expressjs.com/api.html`, accessed 26 April 2020.

[29] Eric Elliott. *Programming JavaScript Applications*. O'Reilly Media, first edition, jul 2014. ISBN: 9781491950289.

[30] Eric Elliott. *Composing Software: An Exploration of Functional Programming and Object Composition in JavaScript*. Independently published, first edition, dec 2018. ISBN: 1661212565.

[31] Ramda contributors. Ramda - Practical functional JavaScript. [Online] https://ramdajs.com/, accessed 27 April 2020.

[32] PIXELTAIL GAMES LLC. Open Trivia DB. [Online] https://opentdb.com/, accessed 27 April 2020.

[33] Michael Jones, John Bradley, and Nat Sakimura. Json Web Token (JWT). RFC, Standards Track, 2015. Available at https://tools.ietf.org/html/rfc7519.

[34] React Rainbow Components Contributors. React Rainbow Components. [Online] https://react-rainbow.firebaseapp.com/, accessed 27 April 2020.

[35] Ethan Marcotte. *Responsive Web Design*. A Book Apart, first edition, 2011. ISBN: 9780984442577.

[36] Webpack. [Online] https://webpack.js.org/, accessed 5 May 2020.

[37] Cypress.io. Javascript end to end testing framework. [Online] https://www.cypress.io/, accessed 28 April 2020.

[38] Shoreditch Ops Ltd. Artillery.io. [Online] https://artillery.io/, accessed 3 May 2020.

Appendix A

Acronyms

API	Application Programming Interface
CSS	Cascading Style Sheets
HTTP	Hypertext Transfer Protocol
JWT	Json Web Token
TCP	Transmission Control Protocol
SQL	Structured Query Language
ORM	Object-Relational Mapping
PWA	Progressive Web Application
IO	Input Output
UI	User Interface
UX	User Experience

Appendix B

Contents of enclosed CD

`README.md`		CD contents description
`src`		directory with source code
	`client`	client application source code
	`server`	server application source code
	`README.md`	instructions for building and running the application
`thesis`		
	`src`	LaTeX source code
	`thesis.pdf`	thesis text in PDF format