Binod Panta

Full stack web app development using T3 stack

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

24 May 2024

Abstract

Author:	Binod Panta
Title:	Full Stack Web app development using T3 stack.
Number of Pages:	53 pages
Date:	24 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Ilkka Kylmäniemi, Head of Department

The growth and development of technologies uplifted and enhanced the scope of web application development. Among such a vast range of tech stacks, a growing stack scaffold named the T3 stack that comprises the core components such as TypeScript, Next.js, NextAuth.js, Prisma, tRPC, and tailwindCSS is gaining popularity among web developers due to its type-safe features.

The objective of this thesis was to explore and understand the potential of modern T3 stacks in web development through practical implementation in full-stack web applications and assess how they make web applications maintainable, scalable, and less error-prone. Moreover, it aimed at exploring the mandatory applicability of type safety in web applications from all fronts, be it frontend, backend, or full stack and analyze the flexibility a developer gets by adopting the T3 stack for web development.

The outcome of this thesis is that the T3 stack leveraged the power of TypeScript across the default and other stacks and made the application type safe. The use of the T3 stack prevented compile-time errors due to incorrect data types and enhanced the overall strength and maintainability of the application. The T3 stack was successfully applied to develop a responsive web application, QuizCard, by demonstrating the practicalities of type safety in the T3 stack.

Keywords: create-t3-app, T3 stack, web development, type safe

Contents

List of Abbreviations

API:	Application Programming Interface
CI/CD:	Continuous Integration Continuous Delivery
CLI:	Command Line Interface
CRUD:	Create Read Update Delete
CSS:	Cascading Style Sheet
DBMS:	Database management system
DOM:	Document Object Model
IDE:	Integrated Development Environment
LAMP:	Linux Apache MySQL PHP
MEAN:	Mongo Express Angular Node
MERN:	Mongo Express React Node
MEVN:	Mongo Express Vue Node
ORM:	Object-relational mapping
RPC:	Remote Procedure Call
SEO:	Search Engine Optimization
UI:	User Interface
JSX:	JavaScript Extensive Markup Language

UX:	User Experience

1 Introduction

The shift towards mobile devices has been significant in the past few years, yet web-based applications continue to hold a crucial role in today's digital landscape. With the rise of responsive and progressive web applications that offer functionalities on par with mobile apps, the importance of web applications is further emphasized. It's evident that web apps will persist and thrive for the foreseeable future. In the current modern world of technology, possessing a web application serves as a digital representation like that of a business card for business enterprises, government agencies, individuals, and others. With the prerequisite of launching and executing in a browser, web applications often serve as common desktop solutions. (Dzhangarov et al., 2021.)

Currently, there are diverse methods and stacks for web development. Among these pre-existing lists of web development stacks, with the motto that type safety is not optional, rose a web development stack called the T3 stack. Using a type safe stack improves productivity and tackles the shipment of bugs in any project. T3 Stack is a kind of CLI template built by developers that uses the setup of a modular T3 Stack app. The template comprises different options that can be generated based on the specific needs of a developer (T3 Stack, 2024.) The T3 stack offers a comprehensive and secure software stack tailored for Next.js and TypeScript development. It furnishes a suite of utilities and frameworks to enhance scalability and facilitate maintenance of applications, incorporating functionalities like inherent server-side rendering, automated code partitioning, and seamless TypeScript support within integrated development environments (IDEs). (Alpyca, 2023)

The fundamental stacks of the T3 stack are Next.js, TypeScript, Tailwind CSS, tRPC, NextAuth.js, and Prisma. Next.js is a React framework that offers a lightly opinionated, heavily optimized approach to creating applications using React. TypeScript assists a developer by offering real-time feedback, facilitating autocomplete suggestions, and alerting the attempt to access a non-existent property or provide a value of an incorrect type. TypeScript reduces the stress of debugging the application through its strict nature. For authentication setup, NextAuth.js provides secure and complete authentication solutions for NextAuth.js applications. NextAuth.js simplifies web development complexities such as session management, sign-in and sign-out procedures, and other aspects of authentication. NextAuth.js streamlines these once manually handled, time-consuming, and error-prone processes, offering unified solutions in NextAuth.js applications. tRPC enables developers to write end-to-end, type safe APIs without the need for code generation or additional runtime overhead. Leveraging TypeScript’s powerful type inference, tRPC automatically infers the API router’s type definitions and enables a developer to invoke API procedures from the frontend with complete type safety and auto-completion.

This thesis aims to create a full stack web application entirely dependent on type safe stacks such as TypeScript, Next.js, Next-Auth.js, and tRPC.
This thesis aims at understanding the potential of the modern T3 stack in web development through practical implementation in full stack web applications. A full stack responsive web application, QuizCard, will be developed implementing the CRUD concept to understand the practicalities of the T3 stack in web application development.

2 Theoretical Background

This chapter covers the theoretical background, brief history of web development, and research problem for this thesis.

2.1 Background

The history of web development dates back to 1989, when the World Wide Web was invented. Along with technological advancement, web development has leapfrogged in its usage and reach. Web development involves creating applications that operate over the internet, typically categorized into frontend and backend development, where the frontend interacts directly with clients and the backend handles server-side operations such as managing and storing data (T. Uppal et al, 2022.)

After the introduction of CSS and JavaScript in web development, the web experience´s creativity and design improved. After media queries started being adopted widely since 2012, it gave flexibility to the developers to implement breakpoints for responsive web design, which is still being used for web development (freecodecamp.org, 2021.) With the introduction of React by Facebook in 2012, React revolutionized the landscape for front-end developers, providing them with a wealth of opportunities to create user-friendly interfaces. The key features of React, such as reusable components, virtual DOM, lifecycle methods, JSX, and react hooks, facilitated developers with code readability, quick response time with sleek user interaction, ample resources, and strong development support (Medium, 2021.)

React is a JavaScript library that empowers developers to construct dynamic user interfaces (UI), defining the elements users see and interact with on-screen. While React JS offers helpful functions or APIs for UI creation, it also allows developers flexibility in integrating these functions into their applications. Its success is partly based upon its flexible nature regarding other application-building aspects, as it allows opportunities for third-party tools such as Next.js to foster. Next.js is a React framework that simplifies the development of web applications by managing the tooling and configuration required for React projects while offering additional structure, features, and optimizations (Next.js, 2024.) The current trends in web development involve single page applications, progressive web applications, AI development, serverless architecture, CI/CD, cloud technology, mixed reality, UX design, etc.(Medium, 2023) Web development has surged to a new height due to the introduction of numerous frameworks and libraries that support each other. Among the popular stacks of web development are MERN, MEAN, LAMP, MEVN, Ruby on Rails, Django, .NET, and JAMSTACK, which are widely adopted for creating web applications. (Codeless, 2023) Among the list of these stacks is a recently introduced stack that is steadily growing and gaining popularity among developers is T3 stack. Backed by the motto “Type safety is not optional, ” the T3 stack aims at creating a complete and secure web application by focusing on simplicity, modularity, and security.

2.2 Type safety in Web application development.

After its adoption by the tech community, JavaScript became a widely used language for web application development. However, in the case of larger and more complex projects, JavaScript has become prone to runtime errors due to the absence of type checking. As JavaScript is a dynamically typed language, it does not require type declarations or type inference. JavaScript executes nonsensical operations such as data conversion, which might cause runtime errors. JavaScript returns the value without error, even if a string is added to an integer. The type of variable is not known unless it has a concrete value at runtime (Destroy all software, 2024.)

Many errors can be identified early in the development process, closer to where they originate. Using Type Safety aids in application development by even automating parts of the coding process, which results in shorter code compared to languages that rely solely on dynamic typing. With the implementation of static typing, incompatible data errors are caught by the type checker and pointed out to the programmer at the location where the erroneous value is inserted. Using type in applications does not only serve to prevent errors but also conveys information to the machine, enabling it to assist the programmer further in their tasks. During refactoring, one can proceed with more assurance, as many errors introduced during this process in general are type errors. Types not only act as a form of documentation for programmers, but they also offer a roadmap indicating which terms are appropriate to use. For individuals familiar with them, types offer insights into what can be expressed through any given API. Types can be seen as defining a structure that is logical and coherent. In dynamic languages, information regarding the logical structure of a program should be conveyed through alternative means (P. Chiusano, 2016.)

The static type checker is equipped with a type system that forms the foundation for numerous beneficial features found in IDEs, including error detection, autocomplete, and automated code restructuring. By incorporating types into code and identifying types during compilation, the static type checker assists developers in writing code with fewer bugs. (Byby, 2021) Thus, to avoid the errors created by JavaScript, a long-time popular language of web development, an improved statically typed language, TypeScript, was introduced in 2012 by Microsoft. TypeScript is a superset of JavaScript that adds a layer of static typing to the language. In addition to bug detection, TypeScript enhances code completion, refactors, and supports class-based object-oriented programming. TypeScript has garnered extensive acceptance among developers for its enhanced readability and ease of maintenance (Reich Report, 2023.)

In programming languages, type safety is a control mechanism that ensures variables access only their designated memory locations in a defined and permissible manner. Essentially, type safety prevents code from executing invalid operations on the underlying object. Type safe variables are a strong foundation for a safe and robust program. (Baeldung, 2024.)

2.3 Research Problem

In this modern, fast-paced digital landscape, businesses must utilize the latest technological innovations to maintain competitiveness. Web application development stands out as a crucial advancement, offering various benefits such as enhancing the online presence, streamlining processes, improving customer engagement, efficient data management, and cost savings. Furthermore, web applications facilitate easy scalability, making them indispensable for businesses of all sizes and industries in the contemporary digital era (LinkedIn, 2024.) React, a JavaScript library initially developed by Facebook in 2013, has gained widespread adoption among developers globally for its ability to simplify UI development. By employing a declarative syntax and component-based approach, React JS enables the creation of complicated and interactive UI components effortlessly. Its popularity is backed up by features such as the capability to divide UI into reusable components, facilitating easier management of large codebases (A-team Global, 2023.)

Utilizing all the features of React with the additional layer of type safety features and strict data types, the T3 app helps to improve productivity and scalability, making code more readable, maintainable, and less error prone. This integrated toolkit offers developers a cohesive workflow and adherence to industry best practices. Renowned for its dynamic components, the T3 Stack seamlessly integrates into diverse projects, spanning from e-commerce platforms to complex data visualization tools (Dev Community, 2024.)

This thesis aims to explore the functionalities of T3 stack and assess how they make web applications maintainable, scalable, and less error prone. Moreover, it will explore the mandatory applicability of type safety in web applications from all fronts, be it frontend, backend, or full stack. It aims at analysing the flexibility a developer gets by adopting the T3 stack for web development.

3 Stacks of T3 stack

This chapter will discuss the stacks that are used in the project. The section will elaborate as the project goes forward.

3.1 Next.js

Next.js, a React framework, empowers developers to create highly performant, SEO-friendly static websites and web apps effortlessly. Next.js is well-known for providing an outstanding developer experience. It comes with a range of features, such as hybrid static and server rendering, support for TypeScript, smart bundling, route prefetching, and more. No extra configuration is needed to use those features. (Next.js, 2024.) With Next.js, developers have several tools that help them move quickly from building to launching their projects. It's easy to learn, simple to use, and comes with powerful features to make development smooth and efficient. (What is Next.js, 2024) Next.js is a React framework that's built on top of Facebook's React library and the create-react-app package. It's designed to be flexible, easy to use, and ready for production, with features that go beyond the basics.

3.2 TypeScript

TypeScript is a compiled programming language that serves as a superset of JavaScript and improves tooling across projects of various sizes. The TypeScript project was led by Anders Hejlsberg, the mastermind behind C# at Microsoft, Often described as "JavaScript with syntax for types," it supplements JavaScript with additional functionalities (What is TypeScript, 2024.)TypeScript bridges JavaScript gaps with strong typing, enhancing app scalability. Its IDE support helps in error detection, offers inline assistance, and improves productivity. With static typing and type inference, it tackles the dynamic type problems of JavaScript, minimizing type-related errors during compilation. TypeScript's compilation process recognizes syntax errors, supporting prompt bug identification and minimizing runtime issues. (The New stack, 2024)

3.3 NextAuth.js

NextAuth.js is an open-source library created for integrating authentication and authorization features into Next.js applications. It provides developers with a collection of tools and APIs for implementing simple user registration, login/logout processes, managing user sessions, and safeguarding application routes and pages. NextAuth.js enables authentication in the Next.js application even with the minimal setup. (Building your application, 2024) It is compatible with various authentication providers, including platforms like Google, Discord, and GitHub, alongside conventional email, and password-based authentication methods (Aplyca, 2024.)

3.4 TailwindCSS

TailwindCSS is a utility-based CSS framework, offering an extensive array of CSS classes and tools to simplify the process of styling websites or applications. The core objective of TailwindCSS is to eliminate the trouble of dealing with traditional CSS approaches such as custom styles for each element. TailwindCSS addresses this challenge by providing a diverse range of CSS classes, each serving a specific purpose. For instance, instead of defining a generic `.btn` class with a multitude of CSS attributes, TailwindCSS encourages the application of focused utility classes directly to the button element, such as `bg-blue-500, py-2, px-4`, and rounded. Alternatively, developers can construct a custom `.btn` class by combining these utility classes, thereby restructuring the styling process. Tailwind CSS offers great support for responsive design. It provides various utility classes that help developers create interfaces that adjust to different screen sizes and devices. This makes it simple to design for mobile devices first and ensures that the interface looks good on various screens. (FreeCampcode, 2020.)

3.5 tRPC

tRPC (TypeScript-based Remote Procedure Call) is a framework designed for building microservices and distributed systems using TypeScript by offering simplicity and security in the applications. tRPC is one of the important stacks of the T3 app because of its type safety. Since TypeScript is used in tRPC, it ensures that the client and server-side use correctly typed requests and responses. Due to its type safe nature, the reliability and scalability of application are improved as the type-related bugs are detected at compile time (CleanCommit, 2022.)

tRPC provides APIs that support a wide range of data types and formats, such as JSON, binary data, and streaming data. The use of tRPC makes API development fast and simple. Due to TypeScript, tRPC automatically detects the data type, making development faster at the beginning of the project. Additionally, tRPC is compatible with most of the available IDEs, making development smoother (CleanCommit, 2022.)

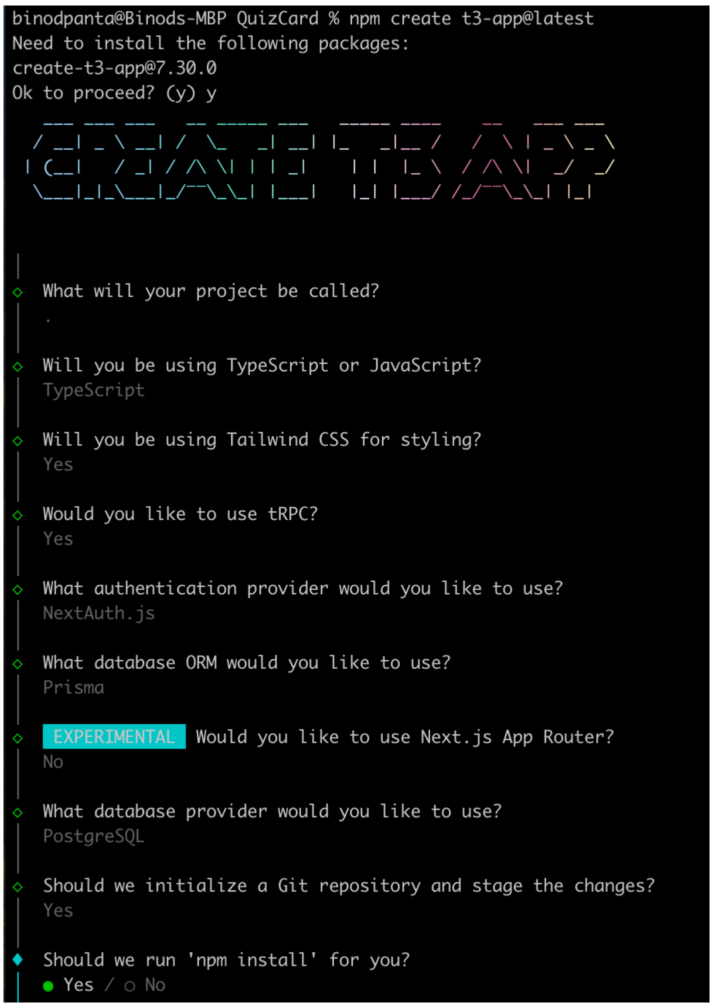
tRPC is a suitable option for the Next.js project as it helps to connect Next's backend and frontend as Next.js is a backend framework. Moreover, tRPC avoids all the unnecessary workload and allows developers to build a reliable and lightweight API that calls backend data in frontend (CleanCommit, 2022.) tRPC redefines API development, seamlessly connecting clients and servers with a type safe approach. It simplifies API creation, ensuring both server and client applications understand data and operations through TypeScript types. (Makeuseof, 2024.)

4 Project Implementation

The thesis will be backed up by a web application development project called QuizCard, created using the T3 stack. The application will allow users to create, update, delete, view, print, and share quiz cards. Authentication will be done using Google, Discord, and GitHub as authentication providers. NextAuth.js will be used for authentication handling in the project. The application will implement the basic CRUD feature, where styling will be backed by tailwindCSS, and the backend APIs will be written using the tRPC protocol. The T3 stack's default database management system, Prisma, will be used for applications' database access and management. Open-source alternatives such as Supabase will be used for storing the data, and UploadThing will be used for storing images.

4.1 Initial Application Setup

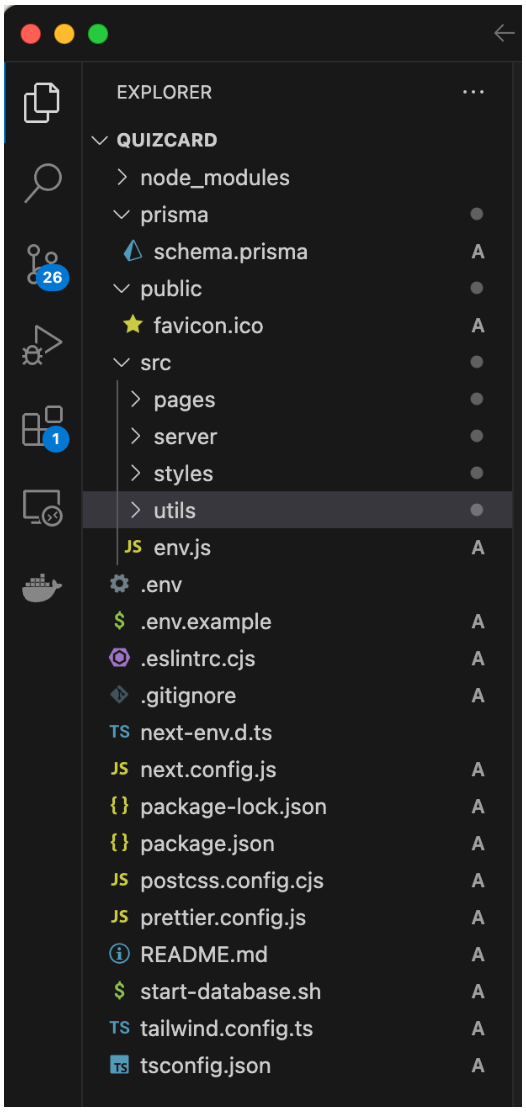
The T3 team created a template including some of the essential stacks of the full-stack TypeScript ecosystem for the application development setup. As mentioned above, it streamlines the setup of type safe Next.js apps without compromising modularity. The command `npm create t3-app@latest` creates a boilerplate code setup for nextAuth, Prisma, tailwindCSS, tRPC, dbContainer, envVariables, and esLint. It allows developers to select from a range of options within the stacks. Create-t3-app is neither a framework nor a stack, but a starter kit to help developers start immediately. Figure 1 below illustrates the template for initial project setup using just a single command and following up on the chain of questions. Based on the user's choice, relevant stacks are installed for the project.



```
binodpanta@Binods-MBP QuizCard % npm create t3-app@latest
Need to install the following packages:
create-t3-app@7.30.0
Ok to proceed? (y) y
   ___ ___ ___   __ _____ ___   _____ ____    __   ___ ___
  / __| _ \ __| /  \_   _| __| |_   _|__ /   /  \ | _ \ _ \
 | (__|   / _| / /\ \| | | _|    | |  |_ \  / /\ \|  _/  _/
  \___|_|_\___|_/‾‾\_\_| |___|   |_| |___/ /_/‾‾\_\_| |_|


│
◇  What will your project be called?
│  .
│
◇  Will you be using TypeScript or JavaScript?
│  TypeScript
│
◇  Will you be using Tailwind CSS for styling?
│  Yes
│
◇  Would you like to use tRPC?
│  Yes
│
◇  What authentication provider would you like to use?
│  NextAuth.js
│
◇  What database ORM would you like to use?
│  Prisma
│
◇  EXPERIMENTAL  Would you like to use Next.js App Router?
│  No
│
◇  What database provider would you like to use?
│  PostgreSQL
│
◇  Should we initialize a Git repository and stage the changes?
│  Yes
│
◆  Should we run 'npm install' for you?
│  ● Yes / ○ No
```

Figure 1: Template for creating a T3 application.

As seen in Figure 1, just with a single command, it gives a range of options for a developer for the application's initial setup. It provides a preconfigured project structure, build scripts, and other necessary dependencies needed to get the project up and running. Upon the completion of the selection setup, the setup builds the folder structure as seen in figure 2. Figure 2 illustrates the folder structure created for the project after all the required installations are complete.



```
EXPLORER                         ...
∨ QUIZCARD
  > node_modules
  ∨ prisma                         ●
      schema.prisma                A
  ∨ public                         ●
      favicon.ico                  A
  ∨ src                            ●
    > pages                        ●
    > server                       ●
    > styles                       ●
    > utils                        ●
      env.js                       A
    .env
    .env.example                   A
    .eslintrc.cjs                  A
    .gitignore                     A
    next-env.d.ts
    next.config.js                 A
    package-lock.json              A
    package.json                   A
    postcss.config.cjs             A
    prettier.config.js             A
    README.md                      A
    start-database.sh              A
    tailwind.config.ts             A
    tsconfig.json                  A
```

Figure 2: Folder structure from the initial setup

As seen in Figure 2, a preconfigured project structure is created where most of the source code is within the src directory. The frontend part of the application is configured in the pages directory, whereas the backend is configured in the server directory. Database models are created in the schema.prisma file, which is located in the Prisma directory. In a T3 template, by default, the authorization for the application is done through the Discord provider, but developers have the flexibility to use the authentication provider of their choice; for example, other popular authentication providers are Google and GitHub. For this thesis project, the database is stored remotely on the Supabase platform. Since tRPC does not support direct file management, an open-source library called

UploadThing was used for storing images remotely, and tRPC handled the images in terms of string variables.

4.2 Initial implementation of other libraries

4.2.1 Shadcn/ui

Apart from the default stacks of the T3 application, some other prominent stacks are used in the application for the thesis project. Shadcn/UI is an open-source collection of beautiful and re-usable UI components. UI component form shadcn/ui collection is used in some portion of the project that allowed the author to access and customize UI components for the thesis project. Each of the components was installed using CLI, such as `npx shadcn-ui add card` . The implementation of shadcn/ui gives developers flexibility as it is not a component library but more like a building block.

4.2.2 UploadThing

UploadThing is an open-source library used in the thesis project for adding file uploads. UploadThing is made to be compatible with type safe applications that harness the power of TypeScript. UploadThing functions similarly to a S3 bucket, serving as a storage solution for application files. It provides an open-source API for authenticating and signing uploads in the project backend without the server processing the file. Listing 1 displays the boilerplate type safe API available for uploading an image.

```
imageUploader: f({ image: { maxFileSize: "4MB" } })
  .middleware(({ req }) => auth(req))
  .onUploadComplete((data) => console.log("file", data)),
```

Listing 1. Boiler plate type safe API for uploading profile picture in UploadThing.

In Listing 1, `imageUploader` is a simple `FileRoute` like an endpoint that has permitted type, max file size, option middleware for authentication, and a callback function `onUploadComplete` for when the image upload is complete. The API can be used in the frontend using the inbuilt UploadThing Button Component, as shown in Listing 2.

```
<UploadButton<OurFileRouter>
  endpoint="imageUploader"
  onClientUploadComplete={(files) => console.log("files",
files)}
  onUploadError= {(error)=>{
alert(`ERROR!! ${error.message}`)}
/>
```

Listing 2. Image upload button for frontend implementation

The `@uploadthing/react` package contains inbuilt “UploadButton” component that handles the uploading of file with a simple click function. In Listing 2, UploadButton component interacts with the server through the imageUploader endpoint , handles upload completion, and displays errors in case any error occurs.

5 Backend Implementation

For the backend implementation of the T3 apps, the default and only type safe option is tRPC. Thus, for this project, tRPC was used as it is TypeScript compatible and handles type safety in the whole application. As seen in Figure 2, the server folder inside the `src` folder implements the backend code of the application. Prisma was used for database management and communication between the frontend and backend, whereas PostgreSQL was used as the database, which was hosted remotely on an open-source database platform called Supabase. Even the initial thought of the project was to run the application on a remote server such as Vercel, Netlify, or Railway, but the scope of this thesis project limited it, and everything was setup locally.

This chapter will deal with the practical backend implementation of the project from scratch and the theoretical aspects related to it.

5.1 tRPC setup

tRPC is a library that makes it easy to create and use fully secure APIs in web development without creating schema or code. It addresses the challenge of statically writing API endpoints in TypeScript and aims to make the process simple and efficient. (Medium, 2023). Most of the backend boilerplate tRPC code is provided by the T3 stack CLI used in the initial setup. Listing 3 below shows customer router `quizCardRouter` containing sample backend APIs of the application.

```
import {
  createTRPCRouter,
  protectedProcedure,
  publicProcedure,
} from "~/server/api/trpc";
import { TRPCError } from "@trpc/server";

export const quizCardRouter = createTRPCRouter({
  createNewCard: protectedProcedure
    .input(cardSchema)
    .mutation(async ({ ctx, input }) => {
      const userId = ctx.session.user.id;
      const createdCards = await ctx.db.deck.create({
        data: {
          name: input.name,
          description: input.description,
          difficulty: input.difficulty,
          image: input.image,
          user: { connect: { id: userId } },
          cards: {
            create: input.cards,
          },
        },
      });

      return createdCards;
    }),
getAllCards: publicProcedure.query(({ ctx }) => {
    const allCards = ctx.db.deck.findMany({
      include: {
        _count: {
          select: {
            cards: true,
          },
        },
        user: true,
      },
    });
    return allCards;
  }),
});
```

Listing 3. Custom router that contains a mutation procedure createNewCard and getAllCards.

The first step to creating a tRPC API is to define the procedure. In general, procedures are functions that are used to build the backend. The procedures can be queries, mutations, or subscriptions, and a single router can contain multiple procedures based on the scale of the backend. Procedures play a crucial role in structuring the API and handling data interaction (tRPC, 2024). In Listing 3, a `quizCardRouter` variable is created, and a protected procedure `createNewCard`, and public procedure `getAllCards`, tRPC's API endpoints were defined within it. As the name suggests, the `createNewCard` procedure creates a card and stores it in the database, whereas `getAllCards` returns the deck of cards as an array containing the cards of all the users. `createNewCard` is a protected procedure; thus, only an authenticated user can create a card on the frontend, whereas `getAllCards` is a public procedure accessible to all users. Each procedure used in a router serves a specific purpose, and it facilitates scalability as new procedures can be added for additional functionalities without affecting current code bases.

As seen in Listing 3, the schema validation open-source library Zod is used to validate the client-side input so that it matches the expectations of the `createNewCard` procedure. Zod ensures that the data sent to and received from API endpoints relies on the predefined schema. Zod helps prevent errors and maintain data integrity by validating input and output data against the schema. If the user input in the frontend does not match the Zod schema, it throws a validation error. Zod has built-in parsing functions that make input schema validation easier and simpler. For example, it gives us flexibility to limit the count of inputs using the inbuilt functions such as `.min(5).max(100)`. Additionally, it gives us the freedom to add optional input if needed with the function `optional()`. The function returns the cards created, which include the user input data. Listing 4 below illustrates the connection between the frontend part of application with backend tRPC API endpoints.

```
import {type AppRouter } from "~/server/api/root";

export const api = createTRPCNext<AppRouter>({
  config() {
    return {
      transformer: superjson,
      links: [
        loggerLink({
          enabled: (opts) =>
            process.env.NODE_ENV === "development" ||
            (opts.direction === "down" && opts.result
instanceof Error),
        }),
        httpBatchLink({
          url: `${getBaseUrl()}/api/trpc`,
        }),
      ],
    };
  },
  ssr: false,
});

//Front-end
const {data: entriesData} = api.card.getAllCards.useQuery();
```

Listing 4. Connecting the API endpoint to the frontend through type safe react-query hooks.

As depicted in Listing 4, These API functions are used in the frontend code through the import of AppRouter as a type. The AppRouter type is later used in type safe react-query hooks before calling in the frontend. The frontend interacts with the API endpoints defined in the backend using type safe react-query hooks. As seen in Listing 4, the `useQuery` hook is used to fetch data for the `getAllCards` endpoint `const { data: entriesData } = api.card.getAllCards.useQuery().` This hook adheres to type safety by providing autocomplete and type checking features based on the types defined for the API responses. `entriesData` receives and stores the array of cards, which is later used in the JSX component to display it to the users. There was no need to use the API endpoint testing platforms such as PostMan, SwaggerUI, Insomnia etc. as the testing was done using the frontend of the application.

5.2 Prisma setup

As mentioned in Chapter 5, Prisma is a powerful ORM tool that simplifies database management and interactions in web development. It assists developers in working with databases using type safe and intuitive APIs. As seen in Figure 2, a folder called Prisma was created using the T3 CLI, that contains `schema.prisma` file. Listing 5 below displays a sample Prisma schema of the application.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model QuizDeck {
  id              String       @id @default(cuid())
  user            User         @relation(fields: [userId], 
references: [id], onDelete: Cascade)
  userId          String
  name            String       @db.VarChar(50)
  description     String       @db.VarChar(200)
  image           String?
  cards           QuizCard[]
  createdAt       DateTime     @default(now())
  updatedAt       DateTime     @updatedAt
  submissionCount Int?         @default(0)
  submissions     Submission[]
  difficulty        String?      @db.VarChar(20)

  @@index([userId])
}

model QuizCard {
  id        String   @id @default(cuid())
  deck      QuizDeck     @relation(fields: [deckId], 
references: [id], onDelete: Cascade)
  deckId    String
  front     String   @db.VarChar(200)
  back      String   @db.VarChar(500)
  createdAt DateTime @default(now())

  @@index([deckId])
}
```

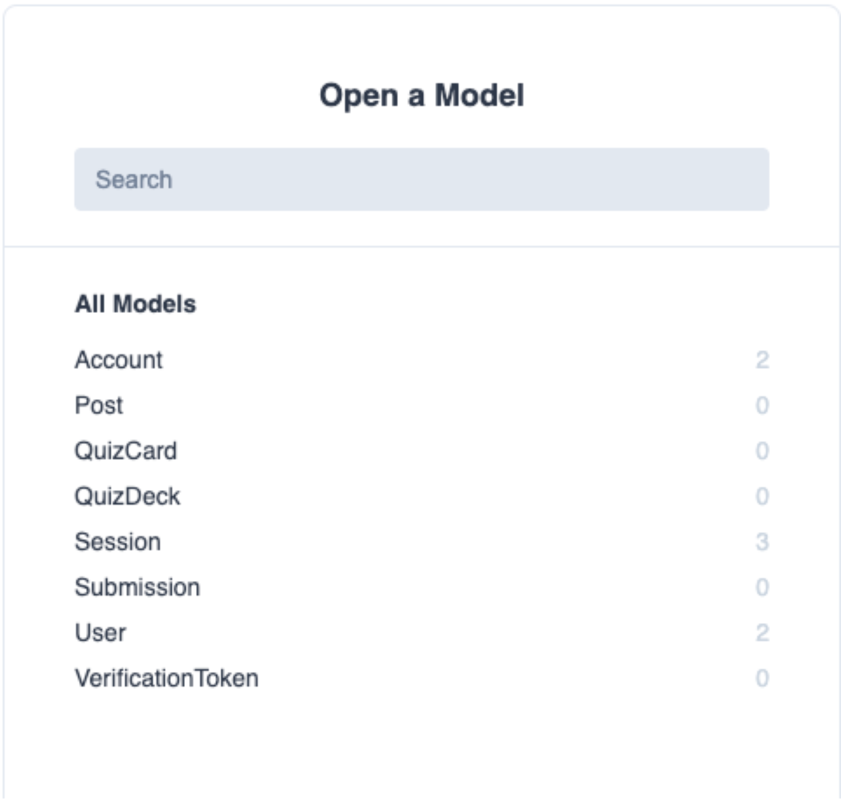
Listing 5. Prisma schema containing sample database model of the project.

The Prisma schema file acts as a blueprint for the database structure. It contains data sources that specify the database provider and connection details, a generator that defines which Prisma client should be generated based upon the available data model, and data models that describe the application’s data entities and the relationship between them. In general, there is a single data source in the Prisma schema (Prisma, 2024.) Once the Prisma command is invoked, the CLI typically reads some information from the schema file. As seen in Listing 5, the schema file specified the data source as PostgreSQL, the generator as Prisma-Client and the data models as `QuizDeck` and `QuizCard`. When used with TypeScript , Prisma Client generates type definitions for those above models and makes database access entirely type safe. (Prisma, 2024)

The environment variable `DATABASE_URL` is accessed using the function `env()`. `DATABASE_URL` is used for connecting the Prisma data schema to the remote database. As mentioned earlier in this thesis report, supabase was used for storing databases remotely. The general PostgreSQL connection string format is `postgresql://USER:PASSWORD@HOST:PORT/DATABASE`. Since the project is using supabase for the database storage, the connection string format looks like:
`postgres://postgres.gtqamrsyeyouzhznmxyv:[PASSWORD]@aws-0-eu-central-1.pooler.supabase.com:5432/postgres`.

Several Prisma models were created based on application's requirements. The models were connected using the `@relation` keyword. For example, to establish a relationship between the Deck table and the Card table, the Deck model was given an array of cards as a parameter, while the Card model was given decks as a parameter. The relationship between other models, User and Deck, Deck and Submission, and User and Card, was also created and applied using the `@relation` keyword. After the models and other setups are completed, the modified schema needs to be pushed to the supabase Postgres database using the command `npx prisma db push`. This command pushes the changes made in the schema file to the database, and the database schema is synchronized with the changes made in the Prisma schema. Prisma provides an in-built visual interface called Prisma Studio that lists all the models defined in the Prisma schema file. It allows developers to perform CRUD operations on the data from the database table.

Prisma Studio is a visual data editor tool that runs by using the command `npx prisma studio` in the terminal. Once the command runs, Prisma Studio gets hosted by default at `localhost:5555`. In the event that port 5555 is unavailable, it will take the nearest available port to display the interface. In figure 3, Prisma Studio displays all the model used in the application, with the data content in the respective tables. Any changes made to the schema would reflect immediately in the database by running the command `npx prisma migrate dev.` Figure 3 below illustrates a simple tabular interface that allows developers to test the data in a local database and validate if the application is working properly.



Open a Model

Search

All Models

Account	2
Post	0
QuizCard	0
QuizDeck	0
Session	3
Submission	0
User	2
VerificationToken	0

Figure 3: Visual interface showing Prisma models in localhost:5555.

As illustrated in Figure 3, some default models, such as Account, Post, User, and Verification Token, are created by default, whereas project specific QuizCard and QuizDeck data models are seen in Prisma Studio. This visual interface gives flexibility and easy access to the data locally. Details can be viewed by clicking on a specific data model. Listing 6 below is about an inbuilt code snippet used for database interaction.

```
import { PrismaClient } from "@prisma/client";

import { env } from "~/env";

const createPrismaClient = () =>
  new PrismaClient({
    log:
      env.NODE_ENV === "development" ? ["query", "error",
"warn"] : ["error"],
  });

const globalForPrisma = globalThis as unknown as {
  prisma: ReturnType<typeof createPrismaClient> |
undefined;
};

export const db = globalForPrisma.prisma ??
createPrismaClient();

if (env.NODE_ENV !== "production") globalForPrisma.prisma =
db;
```

Listing 6. Function creating Prisma Client for interacting with the database.

In listing 6, createPrismaCLient ensures there's only one instance of the Prisma client for interacting with the database. It creates a new client with logging configured based on the environment. The Prisma Client `db` obtained in Listing 6 is used later within the tRPC service to interact with the database and perform database operations. Listing 7 below shows a sample function termed as tRPC procedures.

```
import { createTRPCRouter, protectedProcedure } from
"@/server/api/trpc";
import { TRPCError } from "@trpc/server";
import {
  type RankingResult,
  profileSchema,
} from "@/utils/appUtil/CommonInterface";
```

```
export const userRouter = createTRPCRouter({
  updateUser: protectedProcedure
    .input(profileSchema)
    .mutation(async ({ ctx, input }) => {
      const updatedUser = await ctx.db.user.update({
        where: { id: input.userId },
        data: {
          name: input.userUpdate.username,
          image: input.userUpdate.image,
        },
      });
      return updatedUser;
    }),
  getUserCards: protectedProcedure.query(({ ctx }) => {
    const userCard = ctx.db.quizDeck.findMany({
      include: {
        _count: {
          select: {
            cards: true,
          },
        },
      },
      where: {
        userId: ctx.session.user.id,
      },
    });
    return userCard;
  }),
});
```

Listing 7. Sample procedures used in user router.

A sample of procedures used for the user table that included the tRPC queries and mutation composable can be seen in Listing 7. One of the procedures gets the user profile, while another updates the user profile. Zod schema validator checks whether the inputs are user id and an object comprising username and image. `profileSchema` is passed as an input prop in the `updateUser` procedure. Router generation saves developers' time by automating repetitive tasks in different projects. Any changes that happen in the data model are reflected in the tRPC procedures after running the prisma migration command. The change of data model in the schema file will be reflected, and the tRPC code will be updated based on the new schema.

5.3 NextAuth.js setup

NextAuth.js gives simple and accessible authentication to web applications through an extensive list of authentication providers. In the T3 application, the integration of NextAuth.js, tRPC, and Prisma is handled through the automated initial setup created after running the create-t3 command. (NextAuth.js, 2024) The Prisma schema will have a preconfigured model ready to be used with NextAuth.js. Listing 8 below shows a default Prisma schema generated to handle authentication using NextAuth,js.

```
// Necessary for Next auth
model Account {
  id                String  @id @default(cuid())
  userId            String
  type              String
  provider          String
  providerAccountId String
  refresh_token     String? // @db.Text
  access_token      String? // @db.Text
  expires_at        Int?
  token_type        String?
  scope             String?
  id_token          String? // @db.Text
  session_state     String?
  user              User    @relation(fields: [userId],
references: [id], onDelete: Cascade)

  @@unique([provider, providerAccountId])
}

model Session {
  id           String   @id @default(cuid())
  sessionToken String   @unique
  userId       String
  expires      DateTime
  user         User     @relation(fields: [userId],
references: [id], onDelete: Cascade)
}
```

Listing 8. Preconfigured prisma model that integrates with NextAuth.js.

As depicted in Listing 8, preconfigured models Session and Account are set up automatically once NextAuth.js and Prisma are chosen in the create-t3-app CLI options. When adding new fields to any of the models, those fields are automatically created when a new user signs up and logs in. This is done to handle possible errors. The default value should be provided as the Prisma adapter is not aware of the updated fields. As mentioned in Listings 4 and 7, the protected procedures are accessible to authenticated users only. The integration of NextAuth.js and tRPC helps with easy access to the session object within authenticated procedure (T3 Docs, 2024.)

In order to implement NextAuth.js in a web application, after the relevant installation, the application's entry point is wrapped in the `SessionProvider` component as seen in Listing 9 below.

```
<SessionProvider session={session}>
        <Toaster toastOptions={{ duration: 3000 }} />
        <Navbar />
        <Component {...pageProps} />
</SessionProvider>
```

Listing 9. Application's entry point is wrapped by SessionProvider.

As illustrated in Listing 9, utilizing the provided `<SessionProvider>` facilitates the sharing of the session object among components through React Context. It manages the synchronization of the session across tabs and windows, ensuring it remains updated. Session data and session status are accessible through the `useSession()` hook throughout the application. Listing 10 displays the authentication provider options in the application.

```
export const authOptions: NextAuthOptions = {
  callbacks: {
    session: ({ session, user }) => ({
      ...session,
      user: {
        ...session.user,
        id: user.id,
      },
    }),
  },
  adapter: PrismaAdapter(db) as Adapter,
  providers: [
    DiscordProvider({
      clientId: env.DISCORD_CLIENT_ID,
      clientSecret: env.DISCORD_CLIENT_SECRET,
    }),
    GithubProvider({
      clientId: env.GITHUB_CLIENT_ID,
      clientSecret: env.GITHUB_CLIENT_SECRET,
    }),
    GoogleProvider({
      clientId: env.GOOGLE_CLIENT_ID,
      clientSecret: env.GOOGLE_CLIENT_SECRET,
    }),
  ],
};
```

Listing 10. Authentication option setup using Discord, Google, and GitHub providers.

Client Id and Client secret are necessary keys that are needed for authentication in the NextAuth.js setup. These key values are stored in the `.env` file and accessed across all other files and folders. Different providers have different setup processes and after the setup, the providers give access to the client Id and client secret that can be used in the project for authentication. As depicted in Listing 10, authOptions is ensured to be type safe by setting its type to NextAuthOptions. The NextAuthOptions interface contains providers as a mandatory field, whereas other fields such as session, secret, and jwt are optional. If the parameters used in authOptions are not compatible as types, it throws a type error. Figure 4 below shows different sign in options implemented utilizing NextAuth.js in this thesis project.

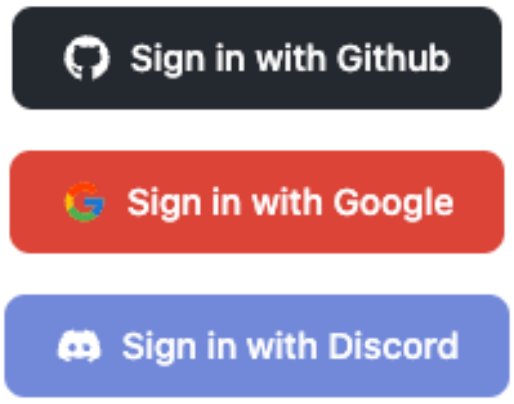


Figure 4: Sign in options as GitHub, Google, and Discord is used for authentication.

The setup done in Listing 1 allows the application to implement three different authentication providers in the project. As seen in Figure 4, users have three different options, such as GitHub, Google, and Discord, to select for authentication. Once authenticated using any of the options, the user data is stored in the session. Listing 11 below illustrates the usage of `sessionData` and `sessionStatus` in the application.

```
const { status: sessionStatus } = useSession();
const { data: sessionData } = useSession();

  useEffect(() => {
    if (sessionStatus === "unauthenticated") {
      void replace("/redirectSignIn");
    }
  }, [replace, sessionStatus]);

{sessionData && (
      <>
        <div className={`${menuToggle ? "md:hidden" :
"hidden md:flex"}`}>
         {!menuToggle && <ThemeSwitcher />}
          </div>
           <UserDropdownMenu
             handleConfirmLogout={handleConfirmLogout}
             avatarSrc={getUpdatedUser?.image ?? ""}
             handleProfileRoute={handleProfileRoute}
            />
        </>
  ) }
```

Listing 11. Sample usage of sessionStatus and sessionData in the project

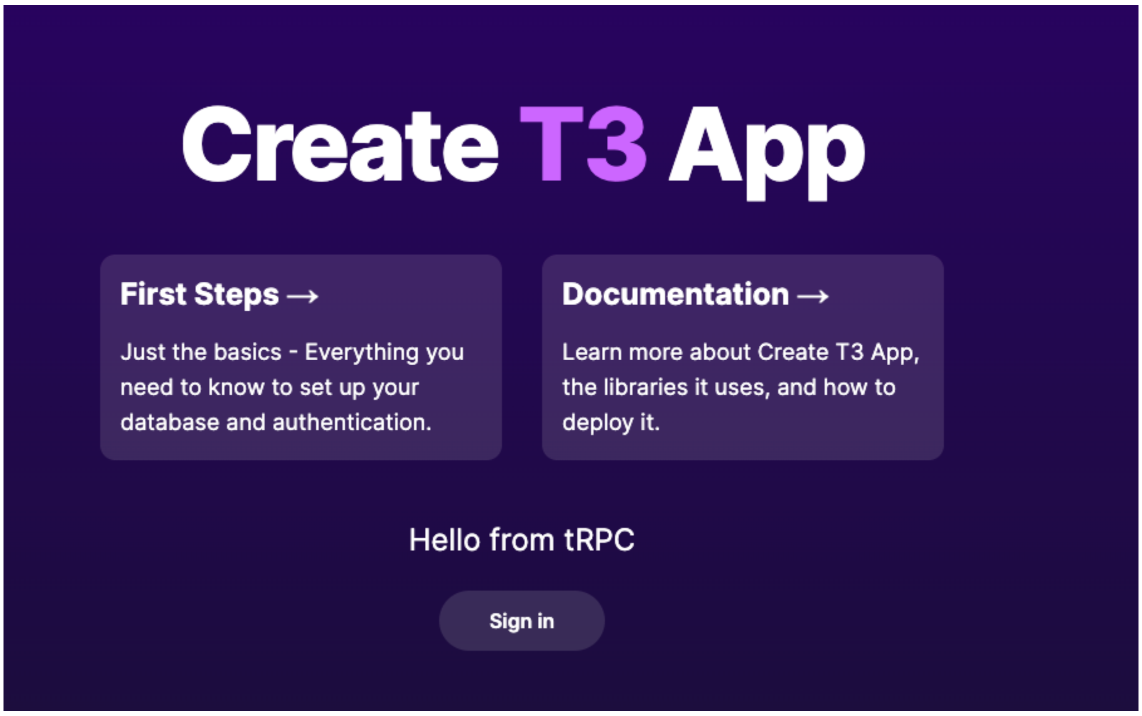
As seen in Listing 11, authenticated users' session data and status are stored in the variable sessionData and sessionStatus and are used in different scenarios with the utmost ease with the implementation of NextAuth.js.

6 Frontend Implementation

This chapter will deal with the practical frontend implementation of the project and the theoretical aspects related to it.

6.1 Initial Frontend Setup

Once the initial backend setup is done fulfilling the minimal requirements, it is time to test the features in the frontend of the application. The T3 application can be run using the command npm run dev, and with the command, the application is hosted locally at http://localhost:3000, and in case the port is unavailable, it takes the closest available port to host the application. The first look of the application is as seen in Figure 5 on hosted in the localhost 3000. Figure 5 shows a ready-made template UI featured in the T3 application. The default sign-in functionality uses Discord as an authentication provider.


Create T3 App

First Steps →

Just the basics - Everything you need to know to set up your database and authentication.

Documentation →

Learn more about Create T3 App, the libraries it uses, and how to deploy it.

Hello from tRPC

Figure 5: Ready made template for T3 Apps

As seen in Figure 5, a simple user interface launches once the application is run using the command npm run dev. The template includes a basic sign-in feature that can be used as a starting point for authentication usage.

6.2 Route setup

In Next.js application pages router has a filesystem-based router setup. Any files added to pages directory are available as a route. Next.js supports both static routes as well as dynamic routes. Static routes are accessed through the `index.tsx` file inside the folder, as shown in Listing 12, below.

```
//Use of index.tsx
pages/index.tsx -----> route to the root route(/)
pages/create/index.tsx ------> route to the create route
http://localhost:3000/create

//Nested Route
pages/blog/first-post.ts -----> route to /blog/first-post
http://localhost:3000/blog/first-post

pages/dashboard/settings/username.ts    --->    route    to
/dashboard/settings/username
http://localhost:3000/setting/username

//Dynamic Route
pages/mycards/[pid].tsx ----> route to dynamic routes such
as /mycards/123, mycards/124 ...
http://localhost:3000/mycards/123
http://localhost:3000/mycards/124
```

Listing 12. Route setup in Next.js application

As presented in Listing 12, different page routes can be accessed by appending the relevant route to localhost:3000/. The files are routed in a nested sequence in case a nested folder structure is created. A single page can be deconstructed into multiple reusable components. By default, Next.js pre-renders each page to enhance the speed and user experience of the application. It utilizes the Link component from the next/link package to facilitate smooth transitions between different routes (Next.js, 2024). For the scope of this thesis project, both static and dynamic routes have been used for routing among pages, as seen in Listing 12.

6.3 Navigation setup

For the navigation setup, a responsive navbar for the web application consisting of the application logo, navigation links such as Home, My Cards, Create, Statistics, theme options, and a user profile drop-down menu is created. State and routing are managed using React hooks such as `useState` and `useRouter`. The navbar adjusts its layout based on the screen size, displaying a hamburger menu on a mobile-based screen. It integrates with authentication by displaying a user dropdown menu when a user is logged in, allowing them to access their profile and logout. At initial setup, the user dropdown menu was not used in smaller screen sizes. The application theme is yet to be implemented for this project. There was not any extra setup needed to connect the backend to the frontend. Just a simple API call made the backend and frontend connections smooth. As seen in Figure 3, all the frontend related code is located in `pages` and `component` files.

Figures 6 and 7 demonstrate the navigation view in the web application and mobile application. These figures show different navigation routes implemented in the application.

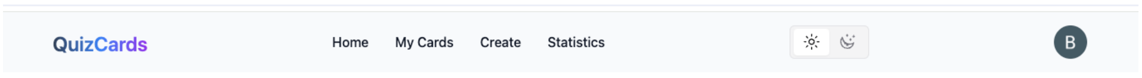


Figure 6: Navigation bar in web view

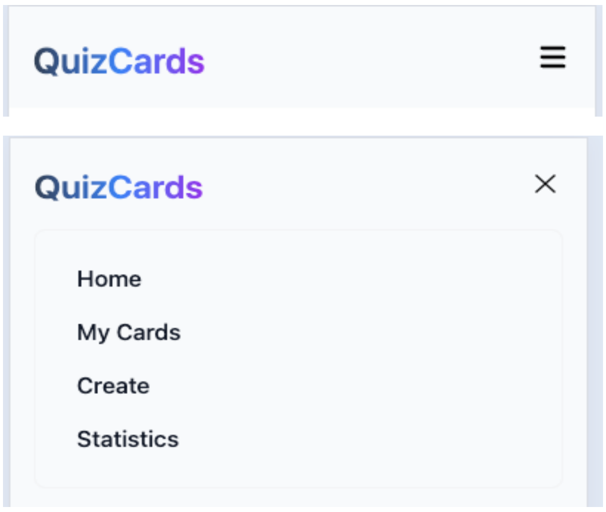


Figure 7: Navigation bar in mobile view

As depicted in Figures 6 and 7, navigation routes contain Home, My Cards, Create, and Statistics routes, where Home displays all the quiz cards created by different users, My Cards is the specified section that displays the card owned by the logged-in user, Create allow the user to create a new card, and the Statistics route displays the basic gamified statistics related to the application.

6.4 CRUD setup

As mentioned earlier in this thesis, the web application implemented the CRUD feature to get a deeper insight into the type safety in different scenarios. For that purpose, a user would be able to create a deck of quiz cards, read, observe, and interact with other users' quiz cards, update, and delete the quiz card exclusively owned by the user. Apart from that, a basic read and update feature is added to the user profile as well. Since the authentication was carried out using nextauth.js, no separate registration or login process was needed for this application.

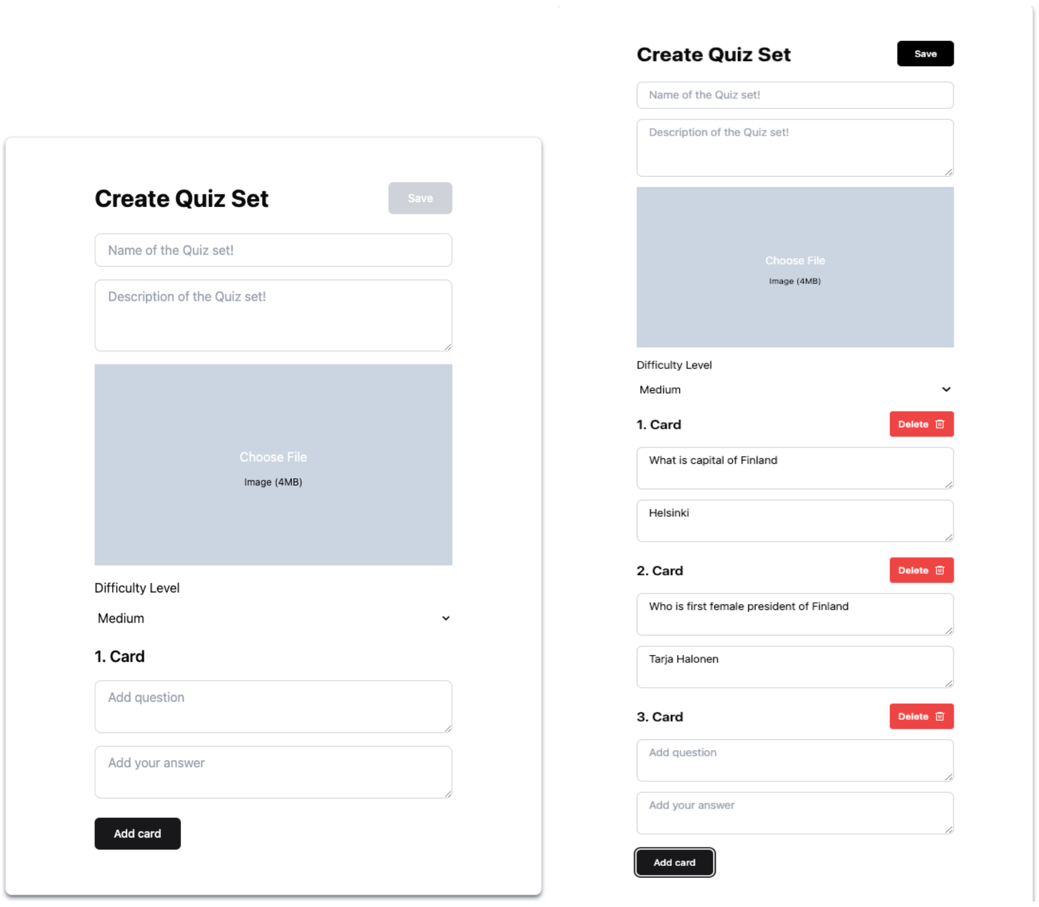
6.4.1 Create quiz card.

The first step of the application usage for a freshly logged-in user is to create a quiz cards. For the creation of a deck of quiz card, components of the `react-hook-form` library were used. The basic components of react-hook-form used for the card creation were, `input`, `textarea`, `select`, `button`, and `form`. As the create component was integrated into the backend using the mutation procedure that required the inputs mentioned in Listing 13.

```
export const setSchema = z.object({
  name: z.string().min(1).max(50),
  description: z.string().min(1).max(200),
  difficulty: z.string().min(1).max(25),
  image: z.string().optional(),
  cards: z
    .array(
      z.object({
        front: z.string().max(200),
        back: z.string().max(500),
      }),
    )
    .min(1),
});
```

Listing 13. Backends' input schema for the create quiz card component.

As illustrated in Listing 13, the form for creating a quiz card involves several input components, such as name, description, and card content, whose information is collected through text areas; the difficulty level is chosen using a select dropdown, and images are uploaded via the UploadButton, a component from UploadThing. Zod schema validation ensures that react-hook-form components only accept values that meet the predefined schema criteria, thereby reducing the likelihood of input errors. Figure 8 below, demonstrates the user interface for creating a quiz card in both web and mobile views. This interface contains a form to create a new quiz card.


Create Quiz Set

Save

Name of the Quiz set!

Description of the Quiz set!

Choose File

Image (4MB)

Difficulty Level

Medium

1. Card

Add question

Add your answer

Add card

Create Quiz Set

Save

Name of the Quiz set!

Description of the Quiz set!

Choose File

Image (4MB)

Difficulty Level

Medium

1. Card

Delete

What is capital of Finland

Helsinki

2. Card

Delete

Who is first female president of Finland

Tarja Halonen

3. Card

Delete

Add question

Add your answer

Add card

Figure 8: User interface for creating a new quiz card.

Figure 8 displays the create quiz card screen of the application, where a user can create a deck of quiz cards. Once the user adds the question and answer for the card, the Save button is enabled, allowing the user to add the deck of quiz cards. Listing 14 illustrates a sample component that handles card creation.

```
//sent the user input data to the backend
  const userEntry = api.card.create.useMutation({
    onSuccess: () => {

//sent the user input data to the backend
```

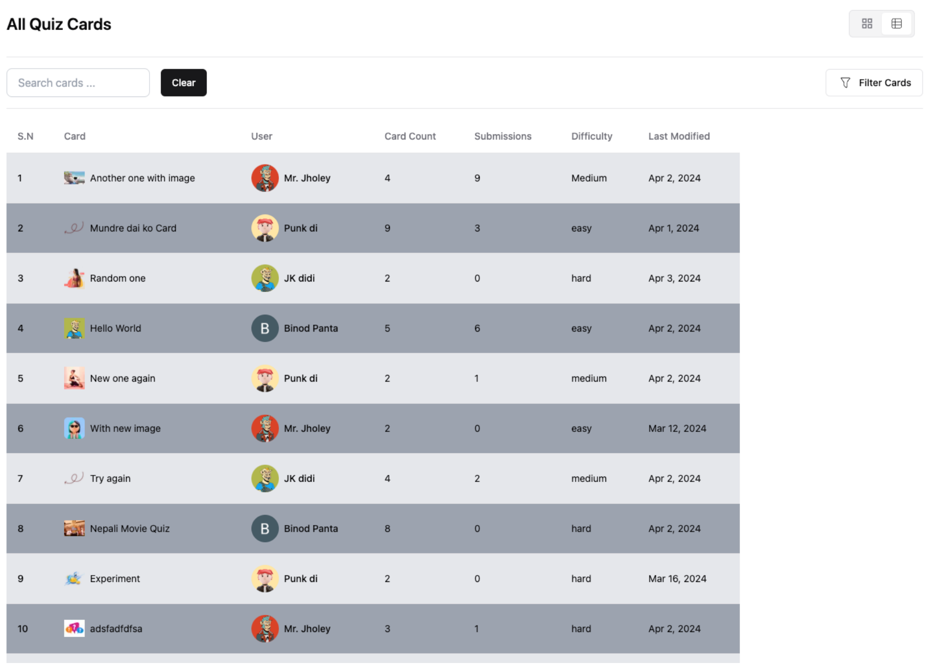
```
  const userEntry = api.card.create.useMutation({
    onSuccess: () => {
      toast.success("Congrats your card is added");
      void replace("/mycards");
    },
onError: () =>
toast.error("Failed to create card Set! Please try again
later."),
  });
const onSubmit = async (formData: SetSchema) => {
try {

      await userEntry.mutateAsync({
        ...formData,
        image: imageUrl,
      });
      await replace("/mycards");
    } catch (error) {
      console.error("Error:", error);
      toast.error("Error in submission.");
    }
  };
```

Listing 14. Sample component that handled the creation of quiz card and sent the data to the backend.

As seen in Listing 14, the input data from the user is sent to the backend through the `onSubmit` function, which receives `formData` as props, and the function mutates the received data through the `userEntry` constant. Image data is appended separately as uploadthing library handles image uploads as strings. On successful mutation or storage of data, the user receives a success toast message and is routed to my cards route. In the case of a failed mutation, a failure toast message is displayed on the screen.

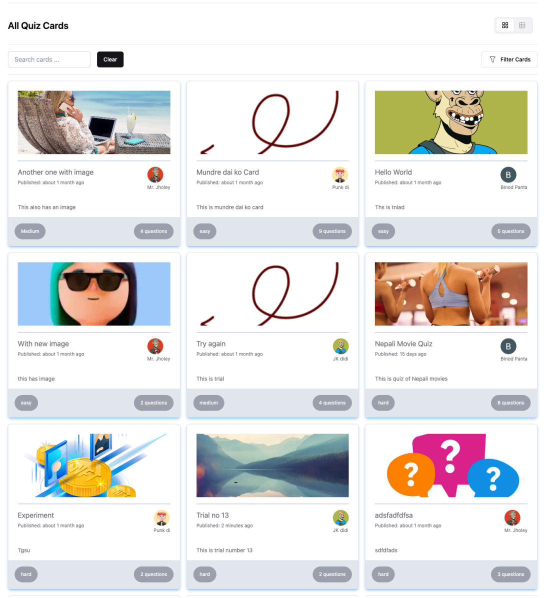
6.4.2 Read quiz cards.

Quiz cards are displayed in two different routes, one in the Home route that displays all the cards and the other in My Cards route that displays the cards owned by the logged-in user. There is the possibility to display the cards in grid and table view and filter the cards for easy access based upon the difficulty, number of cards in the deck, and owner of the card. The search input allows the user to search the card using the card name. Figures 9 and 10 display the list of quiz cards in table and grid views. The user can switch between the view options by clicking the table and grid icons to the left of the screen.



S.N	Card	User	Card Count	Submissions	Difficulty	Last Modified
1	Another one with image	Mr. Jholey	4	9	Medium	Apr 2, 2024
2	Mundre dai ko Card	Punk di	9	3	easy	Apr 1, 2024
3	Random one	JK didi	2	0	hard	Apr 3, 2024
4	Hello World	Binod Panta	5	6	easy	Apr 2, 2024
5	New one again	Punk di	2	1	medium	Apr 2, 2024
6	With new image	Mr. Jholey	2	0	easy	Mar 12, 2024
7	Try again	JK didi	4	2	medium	Apr 2, 2024
8	Nepali Movie Quiz	Binod Panta	8	0	hard	Apr 2, 2024
9	Experiment	Punk di	2	0	hard	Mar 16, 2024
10	adsfadfdfsa	Mr. Jholey	3	1	hard	Apr 2, 2024

Figure 9: List of quiz cards in table view.



Another one with image
Published: about 1 month ago
Mr. Jholey
This also has an image
Medium
4 questions

Mundre dai ko Card
Published: about 1 month ago
Punk di
This is mundre dai ko card
easy
9 questions

Hello World
Published: about 1 month ago
Binod Panta
Ths is tniad
easy
5 questions

With new image
Published: about 1 month ago
Mr. Jholey
this has image
easy
2 questions

Try again
Published: about 1 month ago
JK didi
This is trial
medium
4 questions

Nepali Movie Quiz
Published: 15 days ago
Binod Panta
This is quiz of Nepali movies
hard
8 questions

Experiment
Published: about 1 month ago
Punk di
Tgsu
hard
2 questions

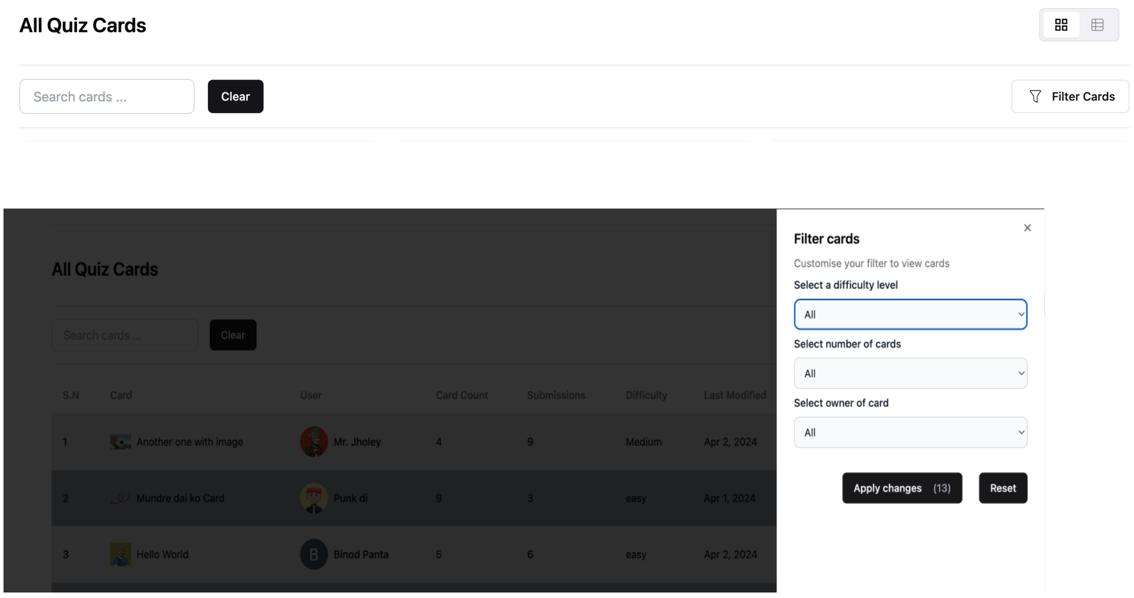
Trial no 13
Published: 2 minutes ago
JK didi
This is trial number 13
hard
2 questions

adsfadfdfsa
Published: about 1 month ago
Mr. Jholey
sdfdfads
hard
3 questions

Figure 10: List of quiz cards in grid view.

As displayed in Figures 9 and 10, users can observe and interact with the quiz cards in two different view options, such as table view and grid view. Table view shows the cards in a simple table, whereas grid view displays the quiz cards in a grid view containing 3 cards in a row. In grid view, the horizontal number of cards changes based on the screen size. For the search option, the `useState` hook is used to hold the current value of the search query entered by the user, where the `handleSearchInputChange` function sets and updates the search query entered by the user. User input is handled by the input component of `react-hook-form`.

For easy access to a set of cards, filter functionality is implemented, where the user can filter the cards based on personal requirements. `filteredEntries` stores the entries data from the backend and filters the data based upon the number of cards, creator of the card, and difficulty level of the card. The user can have quick access to the required card by applying the available filter options. Figure 11 below, displays different filtering options implemented in the Quiz Card project. Filter options include difficulty level, number of cards, and creator of the quiz card.



S.N	Card	User	Card Count	Submissions	Difficulty	Last Modified
1	Another one with image	Mr. Jholey	4	9	Medium	Apr 2, 2024
2	Mundre dai ko Card	Punk di	9	3	easy	Apr 1, 2024
3	Hello World	Binod Panta	5	6	easy	Apr 2, 2024

Select a difficulty level: All
Select number of cards: All
Select owner of card: All

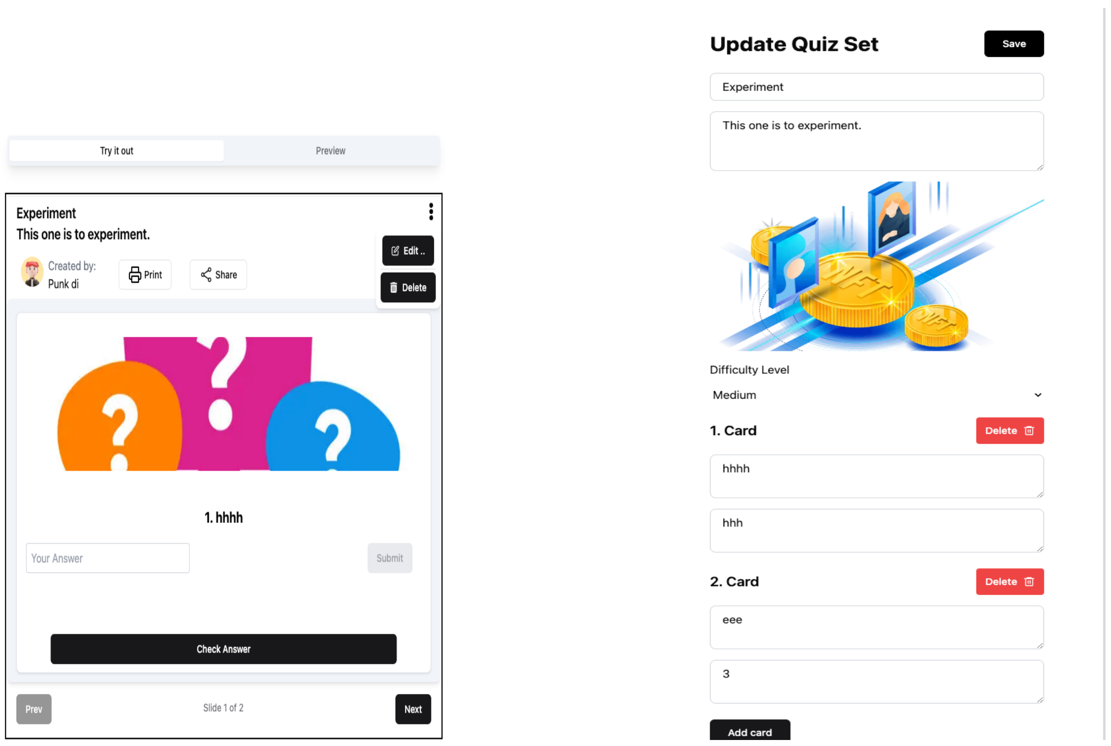
Figure 11: Filter options for filtering the cards.

As seen in figure 11, three different filter options are available to filter the quiz cards. The number of cards available after filtering is displayed in the Apply

Changes button. The user can reset the filter that gives the list of all the available cards.

6.4.3 Update quiz card

An update feature is added to the quiz card and user profile. The user can edit the cards owned by him or her. Using the navigation option for My Cards, users get a grid view of cards. Once a particular card is clicked, the user can view a vertical ellipsis icon that gives the option to either modify or delete the card. Figure 12 below displays the update feature of the quiz card implemented in the project.



Experiment
This one is to experiment.
Created by:
Punk di
1. hhhh
Your Answer
Slide 1 of 2

Update Quiz Set
Experiment
This one is to experiment.
Difficulty Level
Medium
1. Card
hhhh
hhh
2. Card
eee
3

Figure 12: Edit option of single card owned by the user.

Figure 12 shows that the user can modify the card name, description, and card contents, but for now, the image update feature has not been introduced. The update feature is personalised based on the ownership of the quiz card. The cards are visible to all the logged-in users but modifiable by only the owner of the cards. Similarly, users can modify their personal profiles. The user profile has a minimal feature applied that allows the user to modify their image and username. The user can access their profile from the navigation bar by clicking the user image. Within the user profile, users get to update their user image and username, as seen in figure 13.

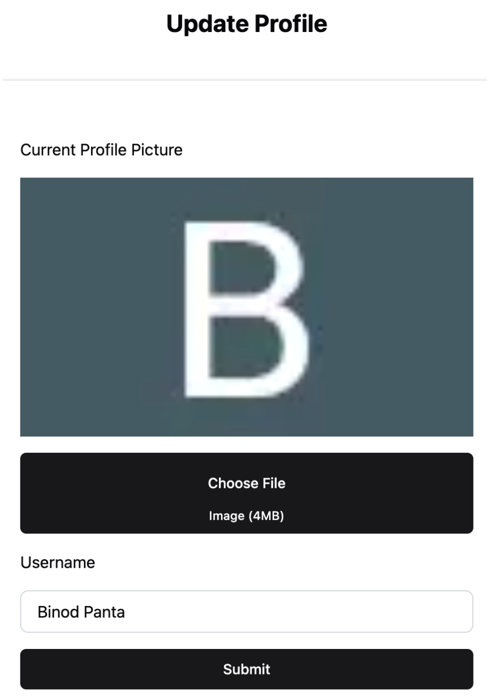
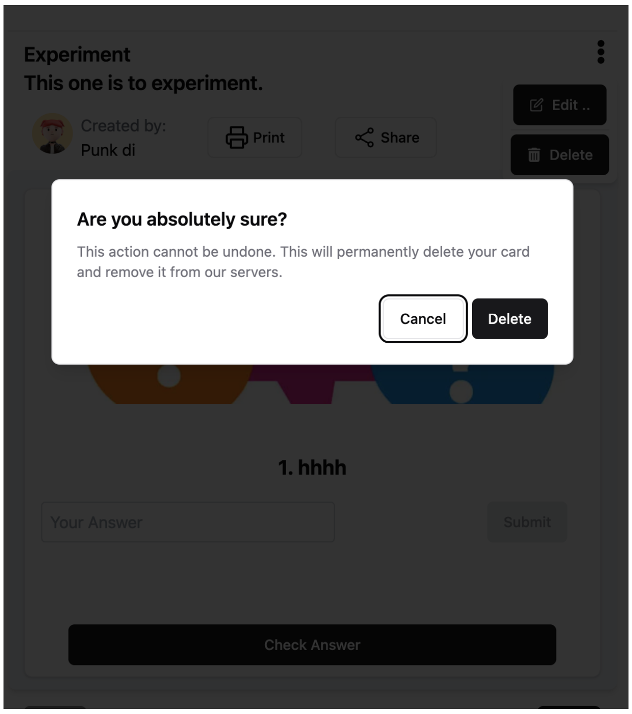


Figure 13: Update profile allows user to modify profile picture and username.

As illustrated in figure 13, user can modify username and profile picture. It is a minimalistic feature that uses UploadThing component within a form component to handle the file update.

6.4.4 Delete quiz card

As the last part of the CRUD application, the delete feature is implemented to delete the cards owned by the user himself or herself. Figure 14 below illustrates the delete alert that pops up for confirmation from the user to delete a card owned by the user.


Are you absolutely sure?

This action cannot be undone. This will permanently delete your card and remove it from our servers.

Figure 14: Prompt initiating the deletion of quiz card.

Users can modify and delete the cards they own. As seen in Figure 11, once the vertical ellipsis icon is clicked, the user can modify or delete the selected card. Once delete is clicked, the user is prompted to confirm if the deletion is intentional, as seen in figure 13. Once confirmed by the user the card is permanently deleted by calling the backend function `deleteQuizCard` as shown in Listing 15 below.

```
  const { mutate: deleteEntryMutations } =
api.card.deleteQuizCard.useMutation({
    onSuccess() {
      void router.push("/mycards");
    },
  });
deleteQuizCard: protectedProcedure
    .input(z.object({ id: z.string() }))
    .mutation(async ({ ctx, input }) => {
      await ctx.db.deck.delete({ where: { id: input.id }
});
      return { message: "Success" };
    }),
```

Listing 15. Delete function that remove the cards permanently.

In Listing 15, the `deleteQuizCard` function removes the card using the card id as an input. As mentioned before, only the user owning the card can delete the card, as it is a protected procedure.

7 Conclusion

The aim of this thesis was to create a responsive full stack web application by utilizing the type-safe feature of a T3 stack. This thesis aimed at understanding the practicalities of type safety in the modern T3 stack in web development through the practical implementation of the CRUD concept in full stack web application. Keeping this thesis aim at the center, a full stack Quiz Card application was developed.

As mentioned above in Chapter 2, using stacks such as Typescript, Next.js, NextAuth.js, and tRPC, ensured the type safety of the web application developed for this thesis. Typescript itself is a type safe language built on top of JavaScript. It is utilized by Next.js, tRPC, Prisma, and NextAuth.js for type safety, making the use of other stacks type safe as well. Next.js supports TypeScript, allowing developers to write type safe React components. The Next.js page component uses TypeScript to ensure type safety by making sure the data passed to the component is correctly typed. Use of prop interface ensures that any prop passed to a React functional component are correctly typed, preventing compile time errors due to incorrect prop types. In case the data type is mismatched, it flags the error at compile time. This way, the error can be avoided at the location where it originated.

tRPC leverages TypeScript to ensure that the data passed between client and server is type-checked. To ensure type safety, the shape of input and output was defined for each API endpoint using the TypeScript types. As seen in Listing 3, `createNewCard`, the tRPC endpoint uses a TypeScript-first Schema validation library, Zod, to ensure that the data passed to the query is correctly typed, providing type safety at the API level. If the user input does not align with the input type schema, it throws a type error. Moreover, using protected procedures protects the endpoint in a simple way without the need for additional codes.

Integration of TypeScript with Prisma in Listing 3 demonstrates the power of TypeScript in Prisma to provide type safe database access that enhances the robustness and maintainability of the application. Based upon the database schema, as seen in Listing 6, Prisma ensured type safe results. In the `getAllCards` function, as seen in Listing 3, the `findMany` method is a promise that resolves to an array of decks, with each deck being a type safe object that matches the structure of the database scheme in Listing 5. The returned `allCards` is a type safe array of Quizdecks, where each deck complies with the structure defined in the database schema. This property ensured the expected data, reducing the likelihood of compile-time errors. The database operations used In this thesis project included creating, reading, updating, and deleting records in the database.

By default, in NextAuth.js, certain types or interfaces are shared across submodules. Session is created at `/src/server/api/auth.ts,` and TypeScript picks it up in every location where it is referenced. Through the concept of module augmentation of NextAuth.js, shared interfaces like sessions are defined in a single place, ensuring type safety across the whole application. As seen in Listing 10, the session interface was extended to include the id property inside the user object, which allowed to define shared types in a single place and have TypeScript automatically recognize them through the applications adhering to type safety.

In a nutshell, the T3 stack leveraged the power of TypeScript across the default and other stacks and made the application type safe. While creating this application, the use of the T3 stack prevented compile time errors due to incorrect data types and enhanced the overall strength and maintainability of the application. Type errors were detected by the type checker and highlighted to the developer at the location where the incorrect value was introduced. In conclusion, the T3 stack has been successfully applied to develop a responsive web application, QuizCard, by demonstrating the practicalities of type safety in the T3 stack. Since the application built for the thesis was a basic application, a thorough investigation could not be made regarding the type safety considering the complex setup. A more comprehensive and large-scale application should be built in order to test the overall scope of the T3 stack as a type safe stack for web application development.

References

Aplyca. (2023). Creating scalable applications with T3. [online] Available at https://www.aplyca.com/en/blog/blog-T3-scalable-aplications. [Accessed 20 March 2024]

Baeldung. (2024). Type Safety in Programming Languages. [online] Available at https://www.baeldung.com/cs/type-safety-programming. [Accessed 3rd April 2024]

Byby. (2021). Type Safety in JavaScript. [online] Available at https://byby.dev/js-type-safety. [Accessed 3rd April 2024]

CleanCommit (2022). tRPC vs GraphQL: How to choose the best option for your next project. [online] Available at https://cleancommit.io/blog/trpc-vs-graphql-how-to-choose-the-best-option-for-your-next-project/. [Accessed 4 April 2024]

Codeless (2023). 8 Best Web Development Stacks for 2024. [online]. Available at https://codeless.co/web-development-stacks/. [Accessed 22 March 2024]

Destroy All Software (2024). Types for anyone who knows a programming language. [online]. Available at https://www.destroyallsoftware.com/compendium/types?share_key=baf6b67369843fa2. [Accessed 3rd April 2024]

Dev Community (2024). Unleash the potential of T3 Stack. [online] https://dev.to/4r7ur/unleash-the-potential-of-t3-stack-m4k. [Accessed 20 March 2024]

FreeCampCode.org. (2020). What is tailwindcss. [online] Available at https://www.freecodecamp.org/news/what-is-tailwind-css-and-how-can-i-add-it-to-my-website-or-react-app/. [Accessed 21 March 2024]

Freecodecamp.org. (2021). A Brief History of Responsive Web Design. [online] Available at https://www.freecodecamp.org/news/a-brief-history-of-responsive-web-design/. [Accessed 22 March 2024]

LinkedIn.com (2024). Why Is Web Application Development Valuable for Businesses? [online] https://www.linkedin.com/pulse/why-web-application-development-valuable-businesses-wp6we. [Accessed 24 March 2024]

Medium 2021. ReactJS: A brief history. [online] Available at https://medium.com/@sjarancio/reactjs-a-brief-history-3c1e969a477f. [Accessed 20 March 2024]

Medium 2023. Dominating Web Development Trends 2023. [online] Available at https://medium.com/quick-code/dominating-web-development-trends-2021-94a8e86ba416. [Accessed 21 March 2024]

Medium 2023. Getting started with the T3-Stack. [online] Available at https://medium.com/@shagilislam786/getting-started-with-the-t3-stack-part-1-3faf3dbe186a. [Accessed 4 April 2024]

Prisma 2024. Data sources [online] Available at https://www.prisma.io/docs/orm/prisma-schema/overview/data-sources. [Accessed on 4 April 2024]

Prisma 2024. Data sources [online] Available at https://www.prisma.io/docs/orm/prisma-schema/data-model/models [Accessed on 4 April 2024]

Reich Report 2023. The evolution of NextJS and Type safety. [online] Available at https://www.erichreich.com/the-evolution-of-nextjs-and-type-safety/. [Accessed on 3 April 2024]

T3 Docs. NextAuth.js [online] Available at https://create.t3.gg/en/usage/next-auth. [Accessed 4 April 2024]

The new stack. (2022). What is TypeScript. [online] Available at https://thenewstack.io/what-is-typescript/. [Accessed 21 March 2024]

The React Framework for the Web.(2021). Next.js. Available at https://nextjs.org/. [Accessed 24 March 2024]

tRPC (2024) Homepage. [online] Available at https://trpc.io/. [Accessed on 4 April 2024]

What is Next.js? A look at popular JavaScript framework. (2024). [online] Available at https://kinsta.com/knowledgebase/next-js/. [Accessed 22 March 2024]

What is TypeScript? A comprehensive guide.(2024). [online] Available at https://kinsta.com/knowledgebase/what-is-typescript/ [Accessed 20 March 2024]

Next.js. (2024). About React and Next.js. [online] Available at https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs. [Accessed 21 March 2024]

NextAuth.js. (2024). Home. [online] Available at https://next-auth.js.org/getting-started/example. [Accessed 4 April 2024]

P. Chiusano, Functional progamming, UX, tech, 2016. The advantage of static typing, simpy stated. [online] Available at https://pchiusano.github.io/2016-09-15/static-vs-dynamic.html. [Accessed 3rd Wed 2024]

T. Uppal, S. Srivastava and K. Saini, "Web Development Framework : Future Trends," 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Greater Noida, India, 2022, pp. 2181-2184, doi: 10.1109/ICAC3N56670.2022.10074105. keywords: {Market research;Libraries;Faces;Front-End;E-Commerce;JavaScript;Framework;HTML;DOM}, [online] Available at https://ieeexplore-ieee-org.ezproxy.metropolia.fi/document/10074105. [Accessed 22 March 2024]

Makeuseof.com. (2024). What Is tRPC and Why Should You Use It? [online] Available at https://www.makeuseof.com/trpc-what-why-use/ [Accessed 24 March 2024]