Markus Nivasalo

Full stack development in TypeScript with tRPC and React Native

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

10 April 2023

Abstract

Author:	Markus Nivasalo
Title:	Full stack development in TypeScript with tRPC and React Native
Number of Pages:	39 pages + 4 appendices
Date:	10 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Ilkka Kylmäniemi, Senior Lecturer

The objective of this thesis was to research the strengths of full stack development utilizing only TypeScript, with the server and client being developed simultaneously in the same project.

The thesis was done by creating a mobile application project with a backend, with the tRPC library as the backbone of the front and backend, and React Native being used for the mobile application itself. PostgreSQL was used as the database, with the Prisma library being used for easier database integration. The server was run locally on a ArchLinux based machine and WebStorm IDE was used for the development process. An android emulator was used for testing the mobile application.

The outcome of this study is that it was determined that it was perfectly viable to develop both the server and client with TypeScript, and that tRPC especially was a great library to build upon that vastly cut down on development time by making it extremely easy to implement API endpoint calls to the frontend.

Keywords: TypeScript, React Native, full stack, mobile development, mobile application

Tiivistelmä

Tekijä:	Markus Nivasalo
Otsikko:	Full stack -kehitys TypeScriptillä käyttäen tRPC:tä ja React Nativea
Sivumäärä:	39 sivua + 4 liitettä
Aika:	10.4.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaaja:	Lehtori Ilkka Kylmäniemi

Insinöörityön tarkoituksena oli tutkia full stack -kehitystä ainoastaan TypeScript-ohjelmointikielellä ja sen vahvuuksia niin, että palvelin ja päätelaitteella käytettävä sovellus kehitettiin samaan aikaan samassa projektissa.

Insinöörityö tehtiin luomalla mobiilisovellusprojekti sekä sen taustalla toimiva palvelin. tRPC-kirjasto toimi sekä mobiilisovelluksen että palvelimen selkärankana, ja React Nativea käytettiin mobiilisovelluksen käyttöliittymän ja toiminnallisuuden kehittämiseen. PostgreSQL valittiin tietokannaksi, ja Prisma-kirjastoa käytettiin helpottamaan tietokantaintegraatiota. Palvelinta ajettiin paikallisesti ArchLinux-pohjaisella järjestelmällä, ja WebStorm-kehitysympäristöä käytettiin itse kehitystyöhön. Android-emulaattori toimi mobiilisovelluksen testiympäristönä.

Työssä selvisi, että TypeScriptin käyttäminen sekä palvelin- että sovelluskehitykseen yhtäaikaisesti oli täysin toimiva ratkaisu. Erityisesti projektin rakentaminen tRPC:n ympärille oli erittäin tehokasta ajan säästämiseksi, sillä se teki rajapintakutsujen tekemisestä mobiilisovelluksessa todella helppoa.

Avainsanat: TypeScript, React Native, full stack, mobiilikehitys, mobiilisovellus

Contents

List of Abbreviations

API: Application Programming Interface.

JWT: JSON Web Token.

SQL: Structured Query Language.

tRPC: TypeScript Remote Procedure Call.

URL: Uniform Resource Locator.

WORA: Write once, run anywhere.

1 Introduction

Since its inception in 2012, TypeScript has garnered massive attention, and its popularity has been ever-growing. In 2022, it was the fifth most popular programming language among developers worldwide, and in a developer survey done in 2020 it was ranked as the second most loved programming language only behind Rust. [1; 2] TypeScript has found its way to practically every field of programming, but it has especially cemented itself as the go-to choice for web development especially for larger projects. [3]

As for mobile development, TypeScript has become a popular choice alongside JavaScript. With constantly evolving cross-platform frameworks allowing a Write once, run anywhere (WORA) solution for mobile platforms, writing mobile software has never been more accessible. Out of the multiple that exist, React Native has been popular ever since its inception, being the most used cross-platform mobile framework from 2019 to 2021. [4] Originally it utilized JavaScript by default, but since its latest update 0.71 it now creates new projects in TypeScript only. [5]

The specification of this thesis project was to create a working full stack TypeScript project, where all the parts would work in conjunction. What this means is that all the parts of the stack would be in the same project, as one of the main aims of the project was to determine the usefulness of TypeScript Remote Procedure Call (tRPC). Though a full stack project in name, the project was to focus on the mobile side of frontend development, due to being for the Mobile Solutions major. Adding a fully functional website into the stack would also add another layer into an already large project.

Both the front and backend were developed side by side, adding features little by little and expanding upon them once everything was working. The plan was to start with a basic backend with local data in the code itself, and then retrieving that data successfully in the mobile application. Once that was working, the PostgreSQL database would be implemented into the backend and the data

stored into the database instead. Then it would be just a matter of implementing manipulation of the database into the mobile application.

Due to the nature of the project and what the goals of it were, it was rather inconsequential what the actual functionality of the application would be. It was therefore decided that the functionality should be something that could have real-world applications. The mobile application would allow the user to add in receipts which would be stored in the backend. An admin user could then approve or disapprove the receipts. Originally the idea was to have the receipts contain an image as well, but tRPC does not contain support for file management so it was left out from the scope of this project. [6]

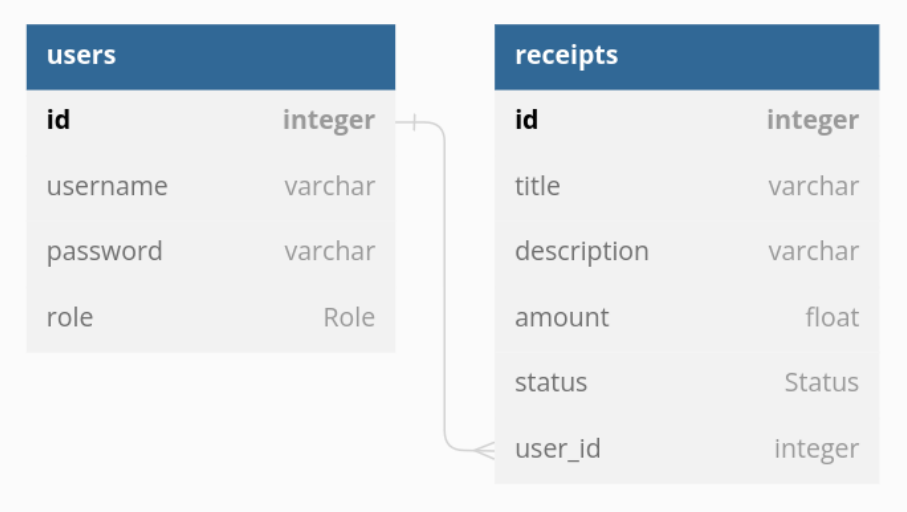

Figure 1: Initial data model for the project, displaying the users and receipts tables and their relations.

As seen in figure 1, the data model for the project was kept simple. This was to keep the scope of the project in check. There would be only two tables, consisting of the users and the receipts. The users would simply have an id, a username, a password and a role. The receipts would then have an id, a title, a description, an amount and a user id referencing the users table, status and a creation timestamp.

Having multiple users of course meant that user creation and identification would be necessary and therefore the passwords would obviously need to be encrypted and their hashes saved into the database. For this sort of authentication, it was decided that JSON Web Token (JWT) would be used, as it is an industry standard.

[7] Bcrypt would be used to handle the password encryption in the program.

2 Backend technologies and implementation

The backend of this thesis project was implemented utilizing TypeScript, with tRPC as the backbone handling type safety across the whole project. Express was used as an adapter to have the created Application Programming Interface (API) accessible via HTTP. PostgreSQL was used as the database, with Prisma handling the communication between the backend and the database.

For the scope of the project it was determined that it was not necessary to run a server in the cloud, but rather everything was done locally. Development was done with a Linux operating system, specifically EndeavourOS, and therefore everything related to server configuration in this thesis report will be handled from the viewpoint of doing it on a Linux machine.

In this chapter the development process of the backend is gone through from start to finish, explaining the various stages in more detail.

2.1 Backend initialization with local data

When writing a full stack application, the backend is often written separately from the frontend. This can cause time not being used as effectively as there is more going back and forth between the codebases. It can also be a cause for errors, as one has to keep track of the data being handled in both the front and back-end and that they match up. For this reason, tRPC was chosen as the backbone of this thesis project. It is a lightweight library that allows for easier communication between the front and backend. tRPC makes it easy to write endpoints that can be used in both the front and back-end with great type safety. Writing the endpoints and then referencing to them in the frontend gives you automatic type safety as well as auto completion for your code, which means you do not have to remember or check what was the correct way for the data to be sent. Thus, a basic tRPC structure was first laid out. [8]

The project started with the creation of a new React Native project. When the template application had been generated, tRPC was added to the project with npm, the package manager for node.js based projects. The necessary packages to get started were installed with `npm install @trpc/server`. The server side of tRPC only required the server package. Once everything was setup, a basic server application was setup with a testing router and some local test data. This required the creation of a context, which holds data all the tRPC procedures have access to. [9]

```
import * as trpcExpress from '@trpc/server/dist/adapters/express';
import {inferAsyncReturnType} from '@trpc/server';

export const createContext = ({
  req,
  res,
}: trpcExpress.CreateExpressContextOptions) => ({});
type Context = inferAsyncReturnType<typeof createContext>;
```

Listing 1: Creating a context for tRPC to utilize.

In listing 1, the `createContext` function is exported to be used in the backend. Since Express was used as the adapter, the options for the context were of type `ExpressContextOptions`. To utilize the context, a tRPC router was created.

```
import { initTRPC } from '@trpc/server';

const t = initTRPC.create();

const router = t.router;

interface User {
  id: number;
  name: string;
}

const userList: User[] = [
  {
    id: 0,
    name: 'Matt',
  },
  ...
];

const appRouter = router({
```

```
21      getUserById: publicProcedure
22        .input((val: unknown) => {
23          if (typeof val === 'number') {
24            return val;
25          }
26
27          throw new Error(`Invalid input: ${typeof val}`);
28        })
29        .query(req => {
30          const {input} = req;
31
32          return userList.find(u => u.id === input);
33        }),
34      getAllUsers: publicProcedure.query(() => {
35        return userList;
36      }),
37  });
38
39  export type AppRouter = typeof appRouter;
```

Listing 2: Creating local test data and a tRPC router for the server-

In listing 2 a tRPC instance was initiated by calling `initTRPC.create()`. There should always only be one instance of tRPC running in the application. [10] The router of the tRPC instance is then saved into the variable `router`. A very basic `User` type was also created for testing, as well as an array with multiple users in it.

With the router of the tRPC instance, a `appRouter` variable was then created and procedures were given to it, which are tRPC's API endpoints. [11] These can be basically any kind of API calls one wants to implement into the router. In this testing example, the appRouter is being given two query procedures, `getUserById` and `getAllUsers`. As the names suggest, the former gets a `User` with the given id from the test array, while the latter returns the entire list. Taking a closer look at the `getUserById` query, it can be seen that before the query happens, there is an input check with the `.input()` call. In this simple example, a simple type check of the given input value is done to see that it is of type `number` and if not, an error is thrown. If the check goes through, the value is returned to the `.query()` function which then finds the correct `User`.

The `type AppRouter` is exported as a type to be used in the frontend. This makes

it so that no actual backend code is imported in the frontend and all the necessary information the frontend needs is transferred via the `type AppRouter`. [10]

The procedures can obviously be a lot more complex, and usually ready-made input parsers are used instead of manually writing checks for everything. At this point, Zod was chosen as the parser for this project, as it is the one tRPC recommends. Thus, the procedures were modified as such.

```
import { z } from "zod";
  ...
const appRouter = router({
    getUserById: publicProcedure
      .input(
        z.number({
            invalid_type_error: "Input must be a number"
        })
      })
      .query(req => { ... } )
      ...
});
```

Listing 3: Using Zod to parse inputs.

In listing 3 Zod can be seen being used for input checking. Whilst simple in this example, Zod makes it way easier to have type checking for the inputs with its many built-in parsing functions. It also makes it easy to have certain fields optional with a simple `.optional()` call for the input. [12]

Finally, the Express adapter was created.

```
const express = require('express');
const app = express();

app.use(
  '/trpc',
  trpcExpress.createExpressMiddleware({
    router: appRouter,
    createContext,
  }),
);

app.listen(3000);
```

Listing 4: Creating the Express adapter and making it use the created tRPC router and context.

Creating the Express adapter with tRPC functionality doesn't differ much if at all from creating a regular one, since tRPC has such good built-in Express support. As can be seen in listing 4, Express was simply given the endpoint chosen to be used, in this case `/trpc`, and create an Express middleware with the previously created router and context. The application was then made to listen on the chosen port, in this case 3000. [13]

The nodemon package was chosen to be used for running the server application, as it allowed for easier development due to it automatically restarting the server when changes were made. [14] Once the server was running, tests were done with Postman to see if the API was working correctly. Sending a GET command to the endpoint at `http://localhost:3000/trpc/getUserById?input=0` resulted with the following response.

```
{
    "result": {
        "data": {
            "id": 0,
            "name": "Matt"
        }
    }
}
```

Listing 5: Response when sending a GET call to the getUserById procedure with id 0

The server was noted to be working correctly at this point, with incorrect calls getting caught and resulting in the correct errors. At this point, the mobile application was made to send the correct queries to the backend to receive the data as well, and from this point on most of the testing was done with the frontend instead of API testers such as Postman. This is because tRPC doesn't really support them properly and the HTTP calls have to be very specific. In August 2022, it was decided amongst the contributors of tRPC to leave it up to the software themselves to rather support tRPC than the other way around. [15] Due to this thesis project not being created as a REST API, it was decided that further

testing outside of the frontend itself was unnecessary.

2.2 Implementing the PostgreSQL database with Prisma

PostgreSQL was chosen as the database, due to its ease of use. Most of the heavy lifting was handled via Prisma, which drastically simplified everything regarding database access. It allows one to create database models writing easily understandable object-like code, and then generate the database schema based on that. [16] Prisma made it especially easy to keep the database up-to-date and make small adjustments if necessary with the use of its migrate functionality. [17]

2.2.1 PostgreSQL setup

Since the project was working fine with local data, implementing a database was the next move. Whilst MySQL with MariaDB was originally considered, PostgreSQL was ultimately chosen as it seemed to have the best compatibility with the chosen tools. The first step was to get a PostgreSQL service running on the server machine.

As mentioned in the beginning of this chapter, the development was done on an Arch-based Linux machine. First the `postgresql` package was installed with the `pacman` package manager via `pacman -S postgresql`. This installed the package and created a PostgreSQL user, and `sudo -iu postgres` would swap to the user. The database then had to be initialized as `postgres` user, using the command `initdb -D /var/lib/postgres/data` which initialized the database cluster into the parameter location. The database service could then be started via `systemctl start postgresql.service`. Once the database service was up and running, a user was created for the service via `createuser markus`, and then the actual project database was created with `createdb receipt_scanner`. At this point the database setup itself was complete, and the rest would be handled with the use of Prisma. [18]

2.2.2 Setting up Prisma

The Prisma package was installed via npm using `npm i prisma`. A Prisma folder was then created into the root directory of the project with a `schema.prisma` file inside of it. The schema tells Prisma everything it needs to know about the database and all the data models are written there. [19]

```
// .env

DATABASE_URL="postgresql://markus@localhost:5432/receipt_scanner"

// schema.prisma

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

model User {
  id Int @id @default(autoincrement())
  username String
  password String
  role Role @default(USER)
  receipts Receipt[]
}

model Receipt {
  id Int @id @default(autoincrement())
  title String
  description String
  amount Float
  status Status @default(OPEN)
  user User? @relation(fields: [userId], references: [id])
  userId Int?
}

enum Role {
  USER
  ADMIN
}

enum Status {
  OPEN
  CLOSED
}
```

Listing 6: Project's .env file and the schema for Prisma

In listing 6, a `.env` file is created into the root of the project. `DATABASE_URL` is then declared, which holds the information Prisma needs to correctly handle the previously created database. The format it expects is while using PostgreSQL is `postgresql://USER:PASSWORD@HOST:PORT/DATABASE`. For the user that was created earlier while setting up the database, no password was set so that was leaved out in the Uniform Resource Locator (URL). Host was set as the localhost since everything was ran locally, and the port was left at PostgreSQL's default 5432. [20]

In the `schema.prisma` file, the created environment variable was then given to the `datasource` object to be used as the URL to which Prisma connects to. Models were then created in accordance to the data model shown in figure 1. To build a relation between the tables, the `User` model was given an array of receipts as a parameter, whilst the `Receipt` model was given a user as a parameter where the relation was declared via the usage of the `@relation` keyword. [17]

Once the setup was complete, Prisma Migrate could then be utilized to update the `receipt_scanner` database with the created models. Migrate creates Structured Query Language (SQL) commands based on the schema it has been given and runs them on the database by using the URL set on the `datasource` object. The command `prisma migrate dev --name init` was ran to initiate the process, and the outcome was checked via the terminal.

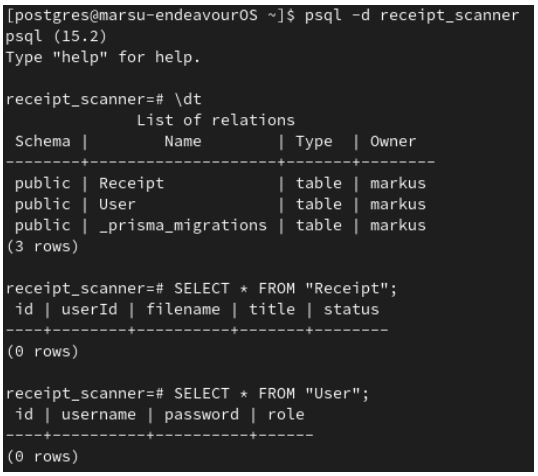

Figure 2: Utilizing terminal commands to display the PostgreSQL tables having been created succesfully through Prisma

As can be seen in figure 2, even though no setting up of tables was done previously while setting up the database, all the tables have been created as they were modeled in the Prisma schema. This simplifies the database handling process tremendously, as at this point changes could simply be made into the schema and the `migrate` command ran again to have those changes apply to the database.

At this point, the database was ready to be implemented into the backend code. For this as well, Prisma offers functionality to automate the process in the form of generators. These can be implemented into the schema to have code generated specifically meant for the tools being used in the project. In the case of this thesis project, the `prisma-trpc-generator` was used. As the name suggests it allows for tRPC specific code to be generated to create backend routers that connect directly to Prisma, which in turn connects to the database. After installing the generator package with `npm i prisma-trpc-generator`, it could then be inserted into the Prisma schema. [21; 22]

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

generator trpc {
  provider = "prisma-trpc-generator"
  withZod = true
```

```
  withMiddleware = false
  withShield = false
  output="../src/server/prisma"
  contextPath = "../src/server/context"
  generateModelActions =
      "aggregate,count,create,delete,findMany,findUnique"
}
```

Listing 7: tRPC generator added into the `schema.prisma` file

Prisma's own proprietary generator `prisma-client-js` is a requirement for the `prisma-trpc-generator` to work, and therefore it needs to be imported into the schema as well as seen in listing 7. For the tRPC generator, multiple parameters can be set. Zod input parsing can be set on or off, but as mentioned in 2.1 in this project it as decided to be used for stronger type safety. A middleware can also be attached to be run before all procedures if desired but it was left off here for now, as was the option for tRPC Shield which helps with for example restricting certain procedures to only certain user roles. The most important part was to enter the wanted output location for the generated files, as well as the path to the context file of the application. The `generateModelActions` parameter was also important, as it determined which database actions were to be created as procedures. [22]

To utilize Prisma in the routers, changes were made to the context file of the server.

```
const prisma = new PrismaClient();

export const createContext = ({}:
    trpcExpress.CreateExpressContextOptions) => ({
  prisma,
});
export type Context = inferAsyncReturnType<typeof createContext>;
```

Listing 8: tRPC context utilizing Prisma

Giving the context access to the `PrismaClient` as shown in listing 8 enables the routers to access it as well. After the package setup and context changes, `prisma generate` was ran which created a new folder in the determined output location. Routers were generated in accordance of the model set in the Prisma schema, to be used for database actions in the frontend.

```
export const usersRouter = t.router({
  aggregateUser: publicProcedure
    .input(UserAggregateSchema)
    .query(async ({ctx, input}) => {
      const aggregateUser = await ctx.prisma.user.aggregate(input);
      return aggregateUser;
    }),
  createOneUser: publicProcedure
    .input(UserCreateOneSchema)
    .mutation(async ({ctx, input}) => {
      const createOneUser = await ctx.prisma.user.create(input);
      return createOneUser;
    }),
  deleteOneUser: publicProcedure
    .input(UserDeleteOneSchema)
    .mutation(async ({ctx, input}) => {
      const deleteOneUser = await ctx.prisma.user.delete(input);
      return deleteOneUser;
    }),
  findManyUser: publicProcedure
    .input(UserFindManySchema)
    .query(async ({ctx, input}) => {
      const findManyUser = await ctx.prisma.user.findMany(input);
      return findManyUser;
    }),
  findUniqueUser: publicProcedure
    .input(UserFindUniqueSchema)
    .query(async ({ctx, input}) => {
      const findUniqueUser = await
          ctx.prisma.user.findUnique(input);
      return findUniqueUser;
    }),
});
```

Listing 9: Generated router for the User table in the PostgreSQL database

The type of router seen in listing 9 was also generated for the Receipt table of the database. As can be seen here, procedures for all the database actions stated in listing 7 were generated, with various schemas set as the input parsers for them all. Taking a look at a schema reveals the Zod checks given to the procedure.

```
const Schema: z.ZodType<Prisma.UserCreateInput> = z
  .object({
    username: z.string(),
    password: z.string(),
    role: z.lazy(() => RoleSchema).optional(),
```

```
    receipts: z
      .lazy(() =>
          ReceiptCreateNestedManyWithoutUserInputObjectSchema)
      .optional(),
  })
  .strict();

export const UserCreateInputObjectSchema = Schema;
```

Listing 10: Zod schema for creating a new User into the database table.

As shown in listing 10, the input is checked for the parameters of the User model shown in listing 6. As seen here, a new user can be created with receipts already under them as well, though it is optional. Also the role is by default set to `USER`, so therefore it is optional.

The usefulness of the router generation cannot be understated. It cuts down on time-consuming manual labor which doesn't really differ from project to project. As only the specifications change and those can be altered via modifying the Prisma schema, there really isn't any reason to not utilize them. Not only that, after adding the generator into the schema, it is automatically ran every time a Prisma migrate process is done. This means any time a change is made into the data models and thus the database, the code is also automatically updated to match the new schema. [17]

The generation process isn't flawless, though. Manual changes can and should be expected at times, as not everything that was generated for this project was perfect. For example, for this project there was a need for a SQL query that selects everything from a table without any filtering, and as can be seen in listing 9 such a call does not exist as the `findMany` query expects an input. Therefore a custom router for this is in order.

```
export const customUserRouter = t.router({
  findMany: publicProcedure.query(async ({ctx}) => {
    return await ctx.prisma.user.findMany();
  }),
});
```

Listing 11: Custom router with a procedure for selecting everything in the User table

In listing 11, the `findMany` procedure is very similar as in the generated router in listing 9, though it is lacking the input checking as there should be no inputs for this call. Having a separate location for custom routers is recommended, as the Prisma generator process can alter folder structures and remove files that are not part of the process. This happened to the thesis project once, though the damage was minimal due to version control.

2.3 User authentication

For the purposes of this project, a relatively simple authentication system was decided to be put in. A personal JWT token is generated when the user logs into the service and saved encrypted onto the device, and this allows them to continue using the application without having to log in every time. For the purposes of this project, the JWT tokens did not have an expiration time set on them, but rather they would last forever. Passwords were hashed using `bcryptjs` before storing them into the database, and then decrypted when logging in.

To get started, `bcryptjs` and `jsonwebtoken` packages were installed via npm, and then a controller class for the authentication was created that would contain the registration and login functionalities.

```
export default class AuthController {
  async signupUser(ctx: Context, input: {username: string; password:
      string}) {
    try {
      const {username, password} = input;
      const hashedPassword = await bcrypt.hash(password, 10);
      const noUsername = username === '';
      const noPassword = password === '';

      if (noUsername || noPassword) {
        return {
          code: ERROR,
          message: 'Please fill all fields',
        };
      }

      // Check if user already exists
      const user_exist = await ctx.prisma.user.findUnique({
        where: {
```

```
19            username,
20          },
21        });
22
23        if (user_exist) {
24          return {
25            code: ERROR,
26            message: 'Username already exists',
27          };
28        }
29
30        const user = await ctx.prisma.user.create({
31          data: {
32            username,
33            password: hashedPassword,
34          },
35        });
36
37        const accessToken = jwt.sign(
38          {id: user.id, username: user.username, role: user.role},
39          ACCESS_SECRET_KEY,
40        );
41
42        return {
43          message: 'Signup Successful, logging in.',
44          status: 'success',
45          user,
46          accessToken,
47        };
48      } catch (error: any) {
49        throw new trpc.TRPCError(error);
50      }
51    }
52    ...
53  }
```

Listing 12: AuthController class's registration functionality

Listing 12 demonstrates the AuthController class' registration function `signupUser`. It simply receives the `username` and `password` inputs, encrypts the password and then checks if the inputs are empty and calls the `User` router's `findUnique` function to check if the username already exists. If everything checks out, it calls the router's `create` function to add the user into the database. After this, the JWT access token is generated, with the user's id, username and role

being stored into the JWT payload and a secret key being used to generate the token itself. After this, an object is returned containing the user data and the access token so they can be stored on the client device, automatically logging in the user after registration. If the user already has an account however, they need to obviously be able to login without creating a new one. [23]

```
export default class AuthController {
  async loginUser(ctx: Context, input: {username: string; password:
      string}) {
    try {
    ...
      const user = await ctx.prisma.user.findUnique({
        where: {username},
      });

      const passwordMatch = await bcrypt.compare(
        password,
        (user as Record<string, any>).password,
      );
    ...
      return {
        message: 'Login successful',
        status: 'success',
        user,
        accessToken,
      };
    } catch (error: any) {
      throw new trpc.TRPCError(error);
    }
  }
  ...
}
```

Listing 13: AuthController class's login functionality

As seen in listing 13, the login process is very similar, except instead of creating a new user the database is queried for the input's username and a `bcrypt` comparison is done on the password to check it against the hashed one. The same type of response object is also returned to the client device.

Once the controller class was done, an authentication router was created so the client could access its functionality.

```
export const authRouter = t.router({
```

```
  authentication: publicProcedure
    .input(
      z.object({
        id: number(),
        accessToken: string(),
      }),
    )
    .mutation(async ({ctx, input}) => {
      let sessionState = SessionState.FAILED;

      try {
        const response = jwt.verify(
          input.accessToken,
          ACCESS_SECRET_KEY,
        ) as jwt.JwtPayload;

        const checkedUser = await ctx.prisma.user.findUnique({
          where: {
            id: response.id,
          },
        });

        if (checkedUser?.id === input.id) {
          sessionState = SessionState.SUCCESS;
        }
        return {sessionState, token: response};
      } catch (error: any) {
        console.error(error);
      }
    }),
  login: publicProcedure
    .input(
      z.object({
        username: z.string(),
        password: z.string(),
      }),
    )
    .mutation(async ({ctx, input}) => {
      return await new AuthController().loginUser(ctx, input);
    }),
  register: publicProcedure
    .input(
      z.object({
        username: z.string(),
        password: z.string(),
      }),
    )
```

```
49      .mutation(async ({ctx, input}) => {
50        return await new AuthController().signupUser(ctx, input);
51      }),
52  });
```

Listing 14: The authentication router, displaying its three procedures: authentication, login and register.

The authentication router shown here in listing 14 differs a bit from previous router examples shown. The login and register procedures are rather straightforward, with each of them calling their respective `AuthController` functions after verifying with Zod that the inputs are correct. The authentication procedure however is used with the token object that's been stored in the client device. It receives as input the stored user id and the access token, and then calls a JWT `verify` function on it. The received payload response can then be compared against the saved user data to confirm the identity. In this case the check is done with the user id. [23]

2.4 Final router setup

At this point, only the receipt functionality remained to be implemented into the frontend. However, the generated router for the Receipt table from section 2.2.2 was not working properly for some reason though the user router was. For this reason, a custom router was set up for the receipts.

```
1  export const customReceiptRouter = t.router({
2    findAllReceipts: publicProcedure.query(async ({ctx}) => {
3      return await ctx.prisma.receipt.findMany();
4    }),
5    findUserReceipts: publicProcedure
6      .input(z.number())
7      .query(async ({ctx, input}) => {
8        return await ctx.prisma.receipt.findMany({
9          where: {
10           userId: input,
11         },
12       });
13     }),
14   createOneReceipt: publicProcedure
15     .input(
16       z.object({
17         title: z.string(),
18         description: z.string(),
```

```
19          amount: z.number(),
20          userId: z.number(),
21        }),
22      )
23      .mutation(async ({ctx, input}) => {
24        const createOneReceipt = await
              ctx.prisma.receipt.create({data: input});
25        return createOneReceipt;
26      }),
27    closeReceipt: publicProcedure
28      .input(
29        z.object({
30          receiptId: z.number(),
31          accessKey: z.string(),
32        }),
33      )
34      .mutation(async ({ctx, input}) => {
35        const response = jwt.verify(
36          input.accessKey,
37          ACCESS_SECRET_KEY,
38        ) as jwt.JwtPayload;
39
40        const checkedUser = await ctx.prisma.user.findUnique({
41          where: {
42            id: response.id,
43          },
44        });
45
46        if (checkedUser?.role === Role.ADMIN) {
47          const closeReceipt = await ctx.prisma.receipt.update({
48            where: {id: input.receiptId},
49            data: {status: Status.CLOSED},
50          });
51          return closeReceipt;
52        }
53        return {
54          status: 'ERROR',
55          message: 'Not an admin',
56        };
57      }),
58  });
59
```

Listing 15: The custom receipt router with its four procedures

As shown in listing 15, procedures were added for retrieving all receipts, retrieving a single user's receipts, creating a receipt and closing a receipt. The

`closeReceipt` procedure is the most unique here, as it receives the session access token stored on the device as input, and verifies it with JWT. After that, it finds the user stored onto the token and verifies that it is indeed an admin user.

3 Frontend technologies and implementation

The frontend of this thesis project was implemented utilizing TypeScript and React Native, with tRPC used to allow for easier communication between the front and backend.

For the mobile application itself, React Native was chosen as it is the most popular cross-platform framework for mobile development, and it integrates well with tRPC. [4; 24] As mentioned in the introduction, starting with version 0.71 new React Native projects will be in TypeScript by default. It also added new more accurate TypeScript declarations, making using TypeScript with React Native more comfortable. For these reasons, it was seen as the best choice for this project.

Due to the restrictions set by Apple on iOS development, the mobile application portion of this project was entirely created for Android as testing for iOS compatibility would have been impossible. Using Expo to combat this was was considered, but due to it adding another layer to the project and not always having the most up-to-date functionality of React Native, it was decided that React Native CLI would be used instead. The application should, however, theoretically run fine on both operating systems.

In this chapter the development process of the frontend is gone through from start to finish, explaining the various stages in more detail.

3.1 Frontend initialization and retrieving data from the backend

The project started with the initialization of a new React Native project. Once that was done, necessary packages for tRPC utilization were installed via npm with `npm install @trpc/client @trpc/react-query @tanstack/react-query`. The client package handles universal communication with the tRPC server package. The react-query packages are specifically needed since the frontend will be React Native based. The hooks used by tRPC's react-query are a thin wrapper around

the TanStack implementation of it and therefore both are needed. [24]

The initial goal was to create a simple single-page application, where the implemented test data from the backend would be displayed in a list. This was done to get used to the way communication between front and backend is handled with tRPC. To get started with this, in the main application file of the React Native project several clients were established.

```
// trpc.ts
import type {AppRouter} from '../server/routers';

export const trpc = createTRPCReact<AppRouter>();

// App.tsx
function App(): JSX.Element {
  const [queryClient] = useState(() => new QueryClient());
  const [trpcClient] = useState(() =>
    trpc.createClient({
      links: [
        httpBatchLink({
          url: 'http://10.0.2.2:3000/trpc',
          async headers() {
            return {};
          },
        }),
      ],
    }),
  );
  ...
}
```

Listing 16: Creating a QueryClient and a TRPCClient

As shown in listing 16, a client instance of tRPC is first created with the use of the `createTRPCReact<>()` function, where it is given the type of the `AppRouter` that was created in listing 2. This type holds all the information that is needed in the frontend. By importing a type, the reference is stripped at compile-time which means server-side code isn't inadvertently imported into the client. [25] A `QueryClient` is then created alongside `TRPCClient` to handle the communication between the client and the server. [26] The `TRPCClient` is simply given a `links` parameter which is an array containing link objects referencing the server. In this case, only an URL to the default API endpoint of the server has to be given, as

there are no headers needed. The URL here is referencing to the localhost of the device running the Android emulator used for testing, which is of course the development PC running the server.

The whole application then needed to be wrapped around providers handling the communication so that queries could actually happen from React Native components.

```
  ...
  return (
    <trpc.Provider client={trpcClient} queryClient={queryClient}>
      <QueryClientProvider client={queryClient}>
          <DashboardScreen />
      </QueryClientProvider>
    </trpc.Provider>
  );
}
```

Listing 17: The main React Native application file's rendering output

The providers in listing 17 are given the previously created clients as props. Wrapping up the entire application allows for tRPC React Query functionality from anywhere in the application. [26] A simple screen component was then created for testing purposes to see if the queries could be sent properly.

```
...
import {trpc} from '../trpc';

export default const DashboardScreen = () => {
  const userList = trpc.getAllUsers.useQuery();

  return (
    <View style={{alignItems: 'center'}}>
      <Text>Testing tRPC</Text>
      <FlatList
        data={userList}
        renderItem={({item}) => <Text>{item.name}</Text>}
      />
    </View>
  );
};
```

Listing 18: The DashboardScreen component's rendering output

The tRPC instance created in listing 16 is now imported here in listing 18. A

simple reference to the router procedure `getAllUsers` created in listing 2 allows for calling of the query, and the data becomes available in the client. The users names were thus rendered on the screen.

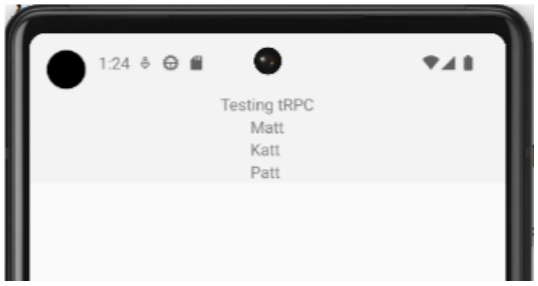

Figure 3: All the users retrieved from the backend displaying correctly on the Android emulator.

As seen in figure 3, retrieving the data from the backend proved extremely simple once the necessary background work was done. This was helped by the fact that when writing the client code after creating the backend fundamentals, code suggestions made it practically impossible to write the client code inaccurately.

After this, the PostgreSQL database with Prisma was setup as discussed in chapter 2.2. Changes to the frontend after this were minimal, as the logic doesn't really change there.

```
...
import {trpc} from '../trpc';

export default const DashboardScreen = () => {
  const userList = trpc.customUser.findMany.useQuery();
  const userCreation = trpc.user.createOne.useMutation();

  return (
    <View style={{alignItems: 'center'}}>
      <Text>Testing tRPC</Text>
      <Button
        title={'Add a user'}
        onPress={() => {
          userCreation.mutate({
            data: {username: 'Test User', password: 'Testpassword'},
          });
        }}
      />
      <FlatList
        data={userList.data}
```

```
21          renderItem={({item}) => <Text>{item.username}</Text>}
22        />
23      </View>
24    );
25  };
26
```

Listing 19: Alterations made to the DashboardScreen component to utilize the newly created backend routers.

As shown in listing 19, the custom made `customUser` router from listing 11 is being used to fetch the entirety of the `User` table from the database. The generated router from listing 9 is then being used for mutations, which in the context of tRPC mean any sort of alterations done to the data, e.g. creating entries, updating them or deleting them. [27] In this case, the `mutate` function is being used to add a new user into the `User` table of the database utilizing some test data. This was then tested to be working.

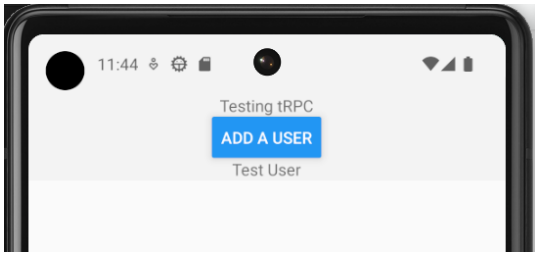

Figure 4: A newly created user created by calling the `mutate` function being displayed in the list fetched from the database.

Running the application and testing both the `query` and `mutation` functionality yielded successful results as can be seen in figure 4, where by pressing the button a new user was able to be created into and displayed from the database.

3.2 Navigation implementation

To handle navigation between different screens in the application, the React Navigation package was chosen. Utilizing the package, it was rather simple to build a basic navigation stack that held all the screens needed. Installation command `npm install @react-navigation/native react-native-screens react-native-safe-area-context` was ran to install the package and its

dependencies, after which a navigator file as well as a navigation type file were created.

```
// types.ts

export type RootStackParamList = {
  Login: undefined;
  Receipts: undefined;
  AddReceipt: undefined;
};

export type LoginScreenProps = NativeStackScreenProps<
  RootStackParamList,
  'Login'
>;
...

// navigator.tsx

const Stack = createStackNavigator<RootStackParamList>();

export default function NavigationStack() {
  return (
    <Stack.Navigator screenOptions={{headerShown: false}}>
      <Stack.Screen name={'Login'} component={LoginScreen} />
      <Stack.Screen
        name={'Receipts'}
        component={ReceiptScreen}
        options={{gestureEnabled: false}}
      />
      <Stack.Screen name={'AddReceipt'} component={AddReceiptScreen}
          />
    </Stack.Navigator>
  );
}
```

Listing 20: The Stack Navigator holding all the different screens of the application, as well as the type file for it containing all the parameters and prop types each screen can have.

As shown in listing 20, the newly created Stack navigator takes the `type RootStackParamList` so that it knows what sort of parameters each screen can be given. The different prop types like `type LoginScreenProps` shown here are created so that TypeScript knows what sort of props each screen is expected to have. [28]

The final structure of the application was also set here. It was determined that only the three screens shown here would be needed.

3.3 Authentication implementation

After the authentication on the backend side was completed as discussed in section 2.3, the functionality was put into place into the mobile application. A login screen that has fields for both logging in and registering was created.

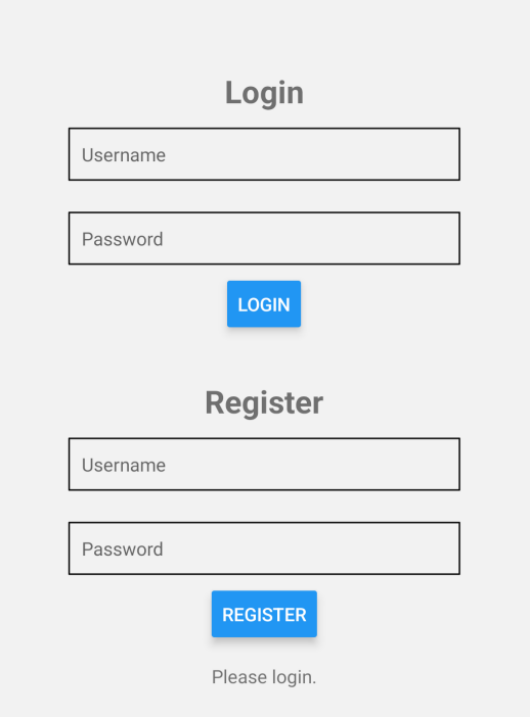

Figure 5: The login screen user interface.

As shown in figure 5, the user interface for the login screen was kept very plain. The login functionality itself was the more interesting part.

```
1   ...
2   const [bearerToken, setBearerToken] = useState('');
3   const authentication = trpc.auth.authentication.useMutation();
4   const registerUser = trpc.auth.register.useMutation();
5   const loginUser = trpc.auth.login.useMutation();
6
7     useEffect(() => {
8         const data = loginUser.data ?? registerUser.data;
9         const accessToken = data?.accessToken;
10        const username = data?.user?.username;
11        const id = data?.user?.id;
```

```
12          const role = data?.user?.role;
13
14          async function setToken() {
15            try {
16              await EncryptedStorage.setItem(
17                USER_SESSION,
18                JSON.stringify({
19                  accessToken,
20                  username,
21                  role,
22                  id,
23                }),
24              );
25            } catch (e) {
26              setError(e as string);
27            }
28          }
29
30          if (accessToken) {
31            setToken().then(() => {
32              setBearerToken(accessToken);
33            });
34          } else {
35            if (data?.code === ERROR) {
36              setError(data.message);
37            }
38          }
39      }, [registerUser.data, loginUser.data]);
40    ...
```

Listing 21: The `useEffect` function that gets called each time a login or registration attempt happens.

In listing 21, constants `registerUser` and `loginUser` are set to connect with the backend authentication router. Not shown here is that when the corresponding buttons from figure 5 for login or registering are pressed, the tRPC mutations are called with the inputted information. The `useEffect` shown here is ran every time the data received from these mutations changes which means every time the controller from listings 12 and 13 returns something. If the data is valid, the received token object is then saved onto the device into encrypted storage and is also set into the `bearerToken` state. This is then used to move forward with the authentication process.

```
1     ...
```

```
2   const [bearerToken, setBearerToken] = useState('');
3   const authentication = trpc.auth.authentication.useMutation();
4   ...
5   useEffect(() => {
6     retrieveUserSession().then(result => {
7       if (result !== undefined) {
8         if (bearerToken === '') {
9           setBearerToken(result);
10        }
11        authentication.mutate({id: result.id, accessToken:
              result.accessToken});
12      }
13    });
14  }, [bearerToken]);
15
16  useEffect(() => {
17    switch (authentication.data?.sessionState) {
18      case SessionState.SUCCESS:
19        const {id, username, role} = bearerToken;
20        navigation.navigate('Receipts', {id, username, role});
21        break;
22      case SessionState.FAILED:
23        setError('Authentication failed, please try logging in
              again.');
24        break;
25      default:
26        setError('Please login.');
27        break;
28    }
29  }, [authentication.data]);
30  ...
```

Listing 22: The `useEffect` functions handling the actual logging in.

When either launching the application or when having the `bearerToken` state change via the `useEffect` function in listing 21, the session retrieval from the device storage happens as shown in listing 22. The `retrieveUserSession` function simply makes a similar call as the `setToken` function in listing 21, but rather than doing a save it does a retrieval. This means that if a session is already active from previous usage of the application, it is retrieved here as the call is done once on the first render and then only if the `bearerToken` state changes. If a session is retrieved successfully, it is then sent to the authentication router's authentication procedure shown in listing 14.

When the authentication procedure returns the data, the other `useEffect` function gets called, and if the authentication has been succesful the user is then navigated to the `Receipts` screen, with the id, username and role being taken as props.

3.4 Creating and displaying receipts

The final two screens were added at this point, with `ReceiptScreen` being the main screen while logged in. If the logged in user is a regular user, it displays a list of their own receipts. If they are an admin, it displays receipts of all the users. The screen has a floating action button at the bottom to navigate to the `AddReceiptScreen`, in which the regular users can create new receipts. The button is hidden from admins, as they do not need to add any receipts. Prop parameters were added for these screens into the navigator type list.

```
1 export type RootStackParamList = {
2   Login: undefined;
3   Receipts: {id: number; username: string; role: Role; accessToken:
        string};
4   AddReceipt: {id: number; username: string; role: Role};
5 };
```

Listing 23: The modified `RootStackParamList` showing the new prop types for the Receipts and AddReceipt screens.

As shown in listing 23, the Receipt and AddReceipt screen receive all the session data that was retrieved in the Login screen upon logging in. These are used to display the correct information on the screens, and the `accessToken` parameter in the Receipt screen is sent to the receipt router when closing receipts to check once more if the token is really for an admin, as seen in listing 15.

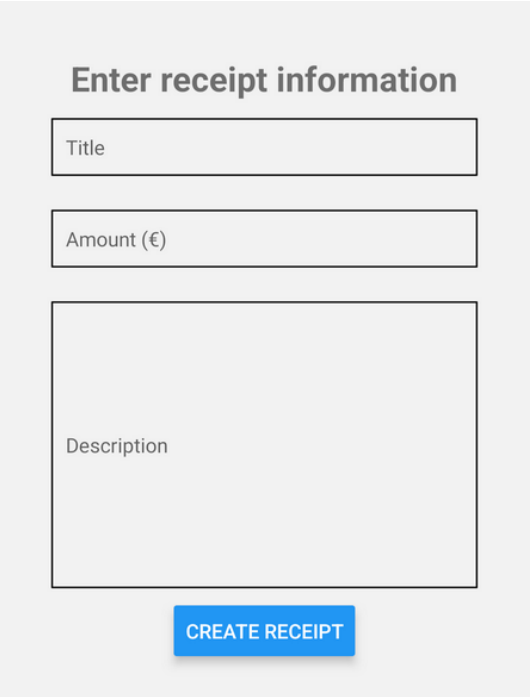

Figure 6: The receipt creation user interface.

The receipt creation was made very simple as can be seen in figure 6, with a few input fields for the title, amount and description. A button was then used to call the receipt creation mutation from the receipt router. This was noted to be working, as receipts were found when checking the database through the terminal.

```
const ReceiptScreen = ({navigation, route}: ReceiptScreenProps) => {
  const {id, username, role, accessToken} = route.params;

  const receiptQuery =
    role === 'USER'
      ? trpc.customReceipt.findUserReceipts.useQuery(id, {
          refetchInterval: 3000,
        })
      : trpc.customReceipt.findAllReceipts.useQuery(null, {
          refetchInterval: 3000,
        });
    ...
};
```

Listing 24: The main functionality of the `ReceiptScreen`

As noted previously, the user's role is brought as a prop from the login process to the Receipt screen. As seen in listing 24, this information is then used to determine which query to call for in the screen. Options were also added to the

queries for the first time in the form of the `refetchInterval` parameters. This determines how often the data is refetched. The receipt items themselves are displayed in a simple `Flatlist` component, and they show a button for closing the receipt if the user is an admin. When pressed, the button calls the `closeReceipt` procedure from the router.

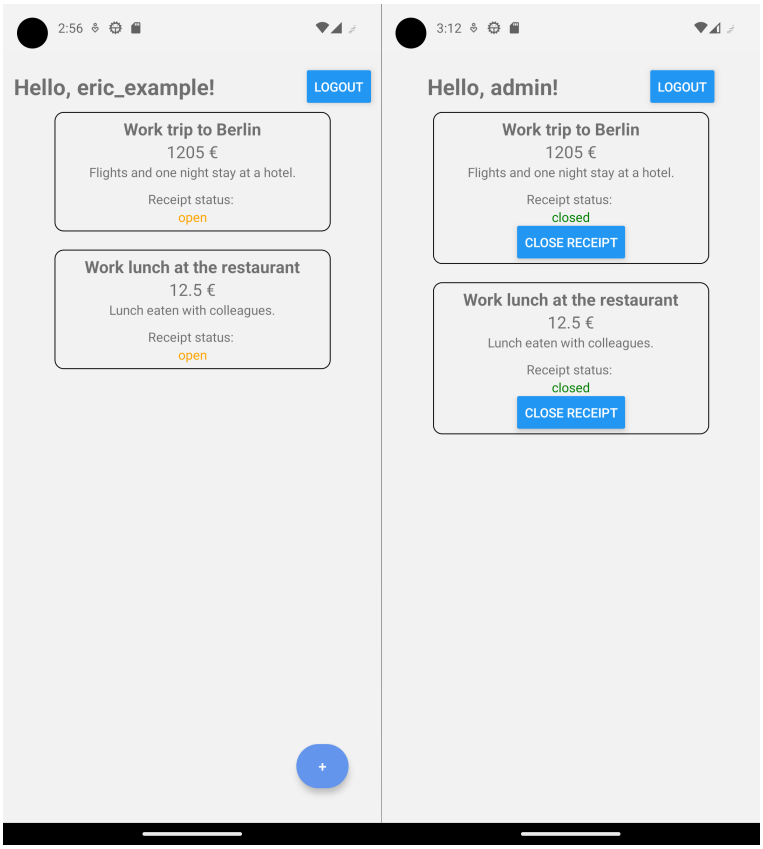

Figure 7: Comparison screenshots displaying the differences between a regular user and an admin.

The screenshots shown in figure 7 were taken with a regular user on the left, and the admin user on the right. The admin screenshot was taken after the receipts were closed using the closing button. As can be seen, the regular user does not have the receipt closing button displayed here, whilst the admin does not have the floating action button for navigating to the receipt creation visible.

4 Analysis and conclusions

As mentioned in the beginning, the objective of this thesis was to create full stack mobile application with TypeScript utilizing tRPC and React Native and analyze the strengths of the taken approach. A mobile application allowing the creation of a user account and uploading of receipts was thus created.

Starting out, a basic backend and frontend were created with dummy test data stored locally in code being used for everything. This made it easy to get used to the basics of tRPC, and also showed arguably its biggest strength. While things were added to both the front and backend constantly throughout the project, many of the functionalities stayed the same. Were one to plan the project out carefully from the start, the frontend code would not need to be changed at all throughout the development, as the usage of the router procedure calls does change at all. As long as the function calls are named correctly with the correct parameters, the same calls work with the local dummy data as well as with actual data from a database. tRPC is not completely perfect for everything, however. As was noted in the introduction, tRPC does not include a support for file management so the photo uploading portion of the application that was planned had to be scrapped.

Once everything was working correctly with the local data, the PostgreSQL database with Prisma was created and connected to the backend. Though it wasn't initally planned to be used and rather came into the project by a mere coincidence, Prisma was most likely the biggest time saver of the entire project. Not having to write the SQL commands to generate and control the database saved tremendous amount of hours throughout the project's lifespan. The migrate functionality also made it quite easy to keep the database synced up. The usage of Prisma generators also saved a lot of time, even though there were issues with some of the generated routers as discussed in section 2.4. Regardless, they still gave a good point of reference from which to create custom routers.

After the database, authentication was added into the project to allow for user

specific content to be shown. Arguably the most difficult part of the project, it turned out alright. There were things that could have been done differently, such as utilizing the tRPC Shield package for handling authentication checks and for only allowing certain roles to have access to certain router procedures. [29] However, for the scope of this thesis project, the simple authentication created here was deemed sufficient. It prevented users from accessing receipts they were not supposed to see, and from closing the receipts if they were not the admin.

As far as the actual user interface is concerned, React Native was deemed a successful choice. tRPC's React Query implementation made it very easy to integrate into the application, and everything regarding the functionality was extremely responsive and quick. This made it a very pleasant development experience. It is worth noting though, that not a lot of documentation was found for React Native specifically, so there was quite a bit of trial and error involved in some parts even though most of the React functionality still applied.

Looking at the thesis analysis, the goals were achieved. The chosen tools were noted as being a great choice, and they would make it easy to keep expanding upon the project. While there would be a need to expand into other frameworks for some functionality such as file uploading, tRPC could still be used as the backbone for everything else.

There are plenty of routes to take when it comes to expanding upon the project. The application does not have any sort of deeper user management, so promoting someone to admin status had to be done via the terminal. Also the aforementioned photo taking and uploading functionality for the receipts would be a good step, as actual receipts being handled in companies usually require photo attachments. As mentioned in the analysis, the authentication could also be expanded upon, for example by actually utilizing tRPC more, thus making the actual procedures shielded from users who should not have access to them.

References

1 Stack Overflow. 2022a. Most used programming languages among developers worldwide as of 2022. Online. <https://www-statista-com.ezproxy.metropolia.fi/statistics/793628/worldwide-developer-survey-most-used-languages/>. Visited on 03/31/2023.

2 — 2022b. Most Loved, Dreaded, and Wanted Languages. Online. <https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted>. Visited on 03/31/2023.

3 Shapel, Maryia. 2023. TypeScript vs JavaScript: Which Is Best in 2023. Online. <https://www.sam-solutions.com/blog/typescript-vs-javascript/>. Visited on 03/31/2023.

4 Stack Overflow. 2021. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. Online. <https://www-statista-com.ezproxy.metropolia.fi/statistics/869224/worldwide-software-developer-working-hours/>. Visited on 03/31/2023.

5 Carroll, Matt; Gerleman, Nick; Corti, Nicola & Sciandra, Lorenzo. 2023. React Native 0.71: TypeScript by Default, Flexbox Gap, and more... Online. <https://reactnative.dev/blog/2023/01/12/version-071>. Visited on 03/31/2023.

6 Is there a way to handle multipart/form-data requests? 2022. Online. <https://github.com/trpc/trpc/discussions/658>. Visited on 04/06/2023.

7 JSON Web Token (JWT) 2015. Internet Engineering Task Force (IETF). <https://www.rfc-editor.org/rfc/rfc7519>. Visited on 01/04/2023.

8 tRPC. 2023a. tRPC homepage. Online. <https://trpc.io/>. Visited on 03/31/2023.

9 — 2023b. tRPC documentation, Context. Online. <https://trpc.io/docs/server/context>. Visited on 04/01/2023.

10 — 2023c. tRPC documentation, Routers. Online. <https://trpc.io/docs/server/routers>. Visited on 04/01/2023.

11 — 2023d. tRPC documentation, Procedures. Online. <https://trpc.io/docs/server/procedures>. Visited on 04/01/2023.

12 Zod Documentation 2023. Online. <https://zod.dev/>. Visited on 04/01/2023.

13 tRPC. 2023e. tRPC documentation, Express Adapter. Online. <https://trpc.io/docs/server/adapters/express>. Visited on 04/01/2023.

14 rem. 2023. nodemon homepage. Online. <https://nodemon.io/>. Visited on 04/01/2023.

15 tRPC contributors. 2022. RFC: Easier calling of tRPC queries with Postman etc. Online. <https://github.com/trpc/trpc/issues/2480>. Visited on 04/01/2023.

16 Prisma Data, Inc. 2023a. What is Prisma? Online. <https://www.prisma.io/docs/concepts/overview/what-is-prisma>. Visited on 03/31/2023.

17 — 2023b. Prisma documentation, Get started with Prisma Migrate. Online. <https://www.prisma.io/docs/concepts/components/prisma-migrate/get-started>. Visited on 03/31/2023.

18 Arch Linux documentation, PostgreSQL 2023. Online. <https://wiki.archlinux.org/title/PostgreSQL>. Visited on 04/02/2023.

19 Prisma Data, Inc. 2023c. Prisma documentation, Installing the Prisma CLI. Online. <https://www.prisma.io/docs/concepts/components/prisma-cli/installation>. Visited on 04/03/2023.

20 — 2023d. Prisma documentation, PostgreSQL. Online. <https://www.prisma.io/docs/concepts/database-connectors/postgresql>. Visited on 04/03/2023.

21 — 2023e. Prisma documentation, Generators. Online. <https://www.prisma.io/docs/concepts/components/prisma-schema/generators>. Visited on 04/03/2023.

22 Dulaimi, Omar. 2023a. prisma-trpc-generator. Online. <https://github.com/omar-dulaimi/prisma-trpc-generator>. Visited on 04/03/2023.

23 Ogunsusi, Temitayo. 2022. Implementing JWT Authentication with TRPC and Express. Online. <https://temitayoogunsusi.hashnode.dev/implementing-jwt-authentication-with-trpc-and-express>. Visited on 04/05/2023.

24 tRPC. 2023f. tRPC documentation, React Query Integration. Online. <https://trpc.io/docs/reactjs/introduction>. Visited on 03/31/2023.

25 — 2023g. tRPC documentation, Set up a tRPC Client. Online. <https://trpc.io/docs/client/setup>. Visited on 04/02/2023.

26 — 2023h. tRPC documentation, Set up the React Query Integration. Online. <https://trpc.io/docs/reactjs/setup>. Visited on 04/02/2023.

27 TanStack Query Docs, Mutations 2023. Online. <https://tanstack.com/query/v3/docs/react/guides/mutations>. Visited on 04/05/2023.

28 React Navigation Docs, Type checking with TypeScript 2023. Online. <https://reactnavigation.org/docs/typescript>. Visited on 04/06/2023.

29 Dulaimi, Omar. 2023b. prisma-trpc-generator. Online. <https://github.com/omar-dulaimi/trpc-shield>. Visited on 04/06/2023.