SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

SYSTÉM RIADENIA TAXISLUŽBY

PROJEKT

Predmet:	I-ASOS – Architektúra softvérových systémov
Prednášajúci:	RNDr. Igor Kossaczký, CSc.
Cvičiaci:	Ing. Stanislav Marochok
Autori:	Bc. Lukáš Patrnčiak
	Bc. Kristián Danišovič
	Bc. Samuel Klement
	Bc. Peter Karas
	Bc. Patrik Karas

Bratislava 2024

Obsah

Úvod **1**

1 Inštalácia a spustenie **2**

2 Architektúra **4**

2.1 Technológie 4

2.2 Funkcionalita 5

2.3 Zabezpečenie 9

2.3.1 Validácia vstupov 9

2.3.2 Rate Limiting 10

2.3.3 Správa chýb a logovanie 10

2.3.4 Autentifikácia a autorizácia 11

2.3.5 Hashovanie hesiel 12

2.3.6 Kontajnerizácia 13

2.3.7 Migrácie databázy 13

2.3.8 Paginácia záznamov 14

3 Štruktúra back-end aplikácie **16**

3.1 Adresárová štruktúra 16

3.2 Architektúra modulov 17

3.3 Hlavné súbory aplikácie 17

3.4 Testovacia infraštruktúra 18

4 Štruktúra front-end aplikácie **19**

4.1 Hlavná štruktúra adresárov 19

4.2 Konfiguračné súbory 20

4.3 Organizácia kódu 20

5 Používateľské rozhranie **22**

6 Testovanie **28**

6.1 Architektúra testov 28

6.2 Príprava testovacieho prostredia 28

6.3 Príklad testovacieho scenára 29

6.4 Pokrytie testami 30

6.5 Spustenie testov 30

6.6 Automatizácia testov 31

Zoznam obrázkov a tabuliek

Obrázok 1	Možnosti použitia CRUD operácii prostredníctvom HTTP metód	7
Obrázok 2	Entitno-relačný diagram databázy systému	8
Obrázok 3	Diagram zabezpečenia aplikácie	9
Obrázok 4	Logovanie	10
Obrázok 5	Hlavná (prihlasovacia) stránka	22
Obrázok 6	Správa účtu používateľa	23
Obrázok 7	Pridanie taxislužby	24
Obrázok 8	Pridanie nového používateľa	24
Obrázok 9	Vytvorenie novej objednávky	25
Obrázok 10	História zákazníkov taxislužby	26
Obrázok 11	Generovanie PDF súboru objednávky	26
Obrázok 12	Vygenerovaný PDF súbor objednávky	27
Obrázok 13	Výsledky testovania	30

Úvod

Cieľom tohto projektu je vytvorenie webovej aplikáciu pre riadenie taxislužby s podporou jednoduchého a prehľadného front-end používateľského rozhrania, pričom manipulovať s jednotlivými údajmi je možné taktiež aj prostredníctvom HTTP metód. Aplikácia disponuje všetkými potrebnými funkcionalitami, ktoré prevádzkovatelia taxislužieb potrebujú. To zahŕňa evidenciu:

- spoločnosti, ktorá prevádzkuje taxislužbu,
- zamestnancov tejto spoločnosti,
- objednávok, ktoré sú zadané podľa požiadaviek zákazníkov
- zákazníkov, ktorí v minulosti využili taxislužbu.

Systém obsahuje možnosť upraviť svoje osobné údaje, ktoré sú zadané v systéme a taktiež aj možnosť generovania vytvorených objednávok do súboru vo formáte PDF.

1 Inštalácia a spustenie

Najskôr je potrebné naklonovať front-end aj back-end repozitáre, pričom je potrebné mať nainštalovanýé nástroje **Git**[1], **PostgreSQL**[2] databázu a **Docker**[3]. Potom je potrebné v otvorenom príkazovom riadku Git Bash spustiť nasledovné príkazy:

```
$ git clone https://github.com/nopo123/taxi-system-fe.git
$ git clone https://github.com/nopo123/taxi-system-be.git
```

Po naklonovaní týchto repozitárov je potrebné v každom z nich (taxi-system-fe aj taxi-system-be) premenovať súbor *.env.example* na *.env*. Tieto súbory obsahujú API nastavenia pre našu webovú aplikáciu, pričom je potrebné nastaviť konkrétne parametre v tomto súbore v taxi-system-be, ktorý obsahuje nasledovné údaje:

```
DATABASE_HOST=localhost
DATABASE_PORT={port}
DATABASE_NAME={name_of_database}
DATABASE_USER={user_name}
DATABASE_PASSWORD={password}
PORT={port_number}
GENERATED_TOKEN={token}
JWT_SECRET={token}
LOCAL_PASSWORD=pass123
```

Údaje v množinových zátvorkách je potrebné zmeniť a zátvorky odstrániť. Tento súbor pre front-end obsahuje zas nasledovné údaje:

```
VITE_APP_VERSION=v1.3.0
GENERATE_SOURCEMAP=false
VITE_BACKEND_URL=http://localhost:3000


## Backend API URL
VITE_APP_BASE_NAME = /
```

Tieto údaje nie je potrebné meniť. Slúžia na smerovanie API požiadaviek z front-end na back-end. Je potrebné ponechať správny port, to znamená, že ak v tomto prípade je pre

[1] https://git-scm.com/
[2] https://www.postgresql.org/
[3] https://www.docker.com/

localhost nastavený port 3000, aj v súbore .env v back-end musí byť *PORT=3000*. Po inštalácii a spustení bude mať front-end port 3001.

Vzhľadom na to, že naša webová aplikácia používa Docker, pre jej inštaláciu a spustenie stačí použiť príkaz

```
$ docker-compose up
```

Taktiež je možné použiť alternatívnu možnosť, použitím správcu balíkov pre JavaScript - npm[4]:

```
$ npm install
$ npm start
```

Následne je možné na stránke *localhost:port/login* vykonať prihlásenie administrátorským učtom:

- **Meno:** admin@admin.com
- **Heslo:** pass

Na záver tejto kapitoly je potrebné spomenúť, že naša aplikácia beží na JavaScript[5] runtime prostredí Node.js[6].

[4]https://www.npmjs.com/
[5]https://www.javascript.com/
[6]https://nodejs.org/

2 Architektúra

Taxi Systém je postavený na architekrúre REST (Representational State Transfer), ktorá je určená pre webové rozhranie. Nižšie budú popísané použité technológie a funkcionalita našej aplikácie.

2.1 Technológie

1. **Framework: NestJS** NestJS[7] je progresívny framework pre Node.js, ktorý je postavený na TypeScripte[8] a používa modularitu a objektovo-orientovaný prístup. V projekte je použitý v prípade modulov, controllerov, služieb, DTO a guardov.

2. **Programovací jazyk: TypeScript** Kód je písaný v TypeScripte, čo zabezpečuje typovú bezpečnosť a lepšiu čitateľnosť kódu.

3. **Swagger[9]** Bol použitý pre čitateľnú dokumentáciu.

4. **Databáza a ORM:** Projekt používa ORM (objektovo relačné mapovanie) (TypeORM[10]) v kombinácii s relačnou databázou PostgreSQL. Relačná databáza bola použíta vzhľadom na entity a DTO štruktúry.

5. **Autentifikácia a autorizácia:** Naša aplikácia používa pre autentifikáciu JWT (JSON Web Tokens)[11]. Táto technológia je použitá napríklad v prípade modulov auth, guard a strategy.

6. **Environmentálne premenné:** Konfigurácia aplikácie je flexibilná a riadená cez enviromentálne premenné (súbor .env).

7. **Kontajnerizácia: Docker** Dockerfile a docker-compose.yml umožňujú spustenie aplikácie v kontajneroch.

8. **Testovanie: Jest** *test/* adresár obsahuje E2E testy (end-to-end testovanie kompletnej funkcionality našej aplikácie), ktoré využívajú testovací framework Jest[12], integrovaný s NestJS.

[7] https://nestjs.com/
[8] https://www.typescriptlang.org/
[9] https://swagger.io/
[10] https://typeorm.io/
[11] https://jwt.io/
[12] https://jestjs.io/

2.2 Funkcionalita

Taxi Systém sprístupňuje rôznu funkcionalitu pre používateľov na základe privilégii, pričom tie môžeme rozdeliť do troch kategórii, v závislosti od ktorých je umožnené danému používateľovi vykonávať CRUD operácie na nižšie popísaných funkcionalitách:

- **Administrátor**
- **Manažér**
- **Zamestnanec**

K týmto operáciam je možné pristupovať prostredníctvom používateľského rozhrania a HTTP metód (GET, POST, PUT a DELETE).

Administrátorské konto je už od začiatku nastavené (kapitola 1), pričom len administrátor a manažér je oprávnený pridávať používateľov a zobrazovať históriu zákazníkov. Zamestnanec je oprávnený spravovať jedine spravovať objednávky.

Všeobecný popis funkcionality:

1. *Spravovanie účtu:* Používateľovi je umožnené v prípade potreby zmeniť svoje krstné meno, priezvisko a email, prípadne heslo, s ktorým sa prihlasuje do systému.

2. *Spravovanie organizácii:* Je možné pridávať rôzne taxi spoločnosti na základe parametrov:

   - názov organizácie
   - adresa organizácie

   V prípade potreby je možnosť taktiež už existujúce spoločnosti upraviť alebo odstrániť.

3. *Spravovanie objednávok:* Vytvorenie novej objednávky je možné po vyplnení nasledujúcich parametrov:

   - krstné meno a priezvisko
   - trasa
   - dátum
   - počet km
   - počet vodičov
   - čakacia doba
   - celková cena (vypočítaná na základe počtu prejdených km, počtu vodičov a dĺžke čakacej doby)

- podpis vodiča
- podpis pasažiera

Všetky objednávky je umožnené vidieť iba administrátorovi. Manažér môže vidieť iba ním vytvorené objednávky a zamestnancom a zamestnanec môže spravovať iba ním vytvorené objednávky. V prípade potreby je možnosť taktiež už existujúce objednávky upraviť alebo odstrániť.

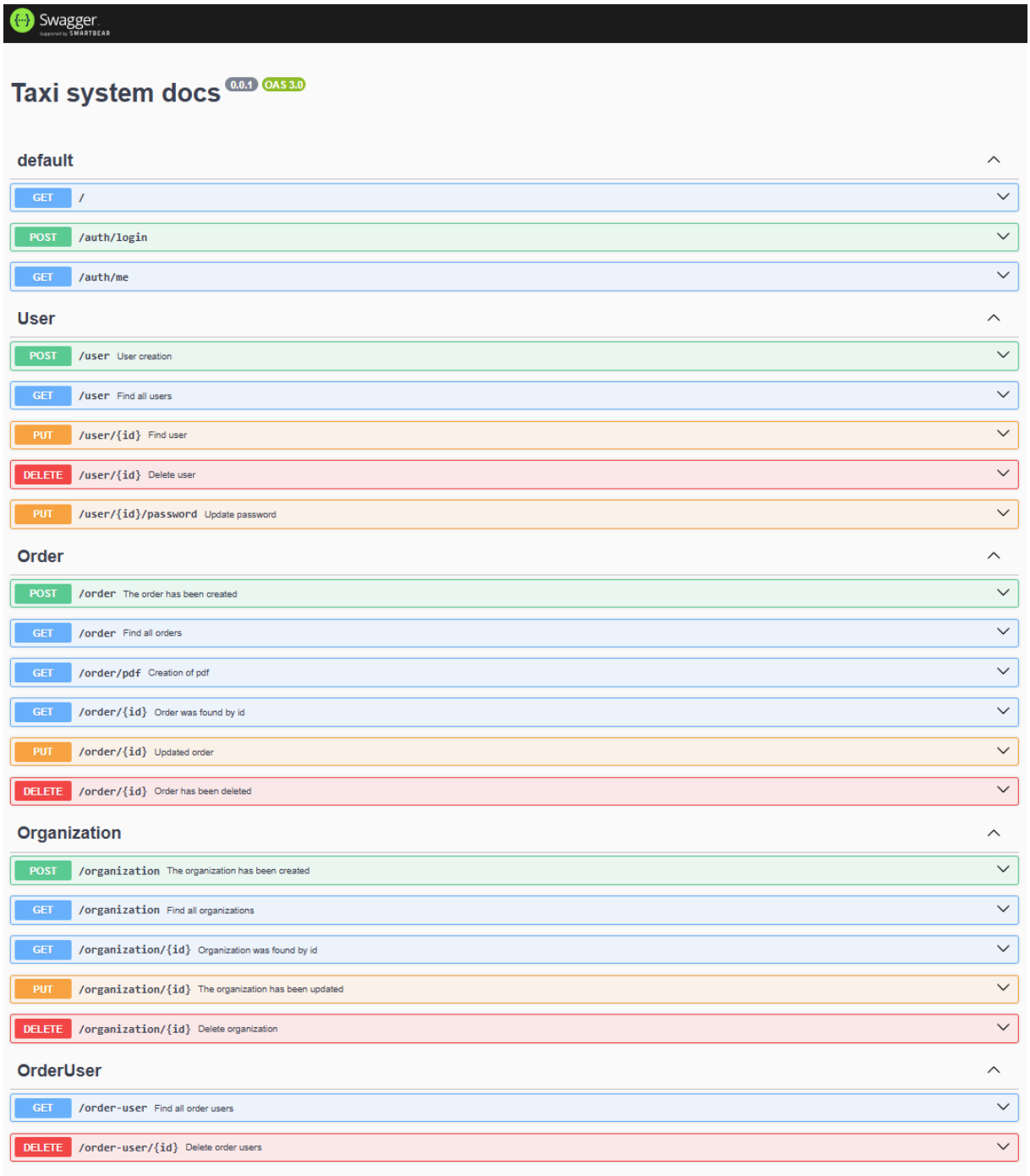
4. *Spravovanie používateľov:*

   - krstné meno
   - priezvisko
   - email
   - heslo
   - rola (manažér/zamestnanec)

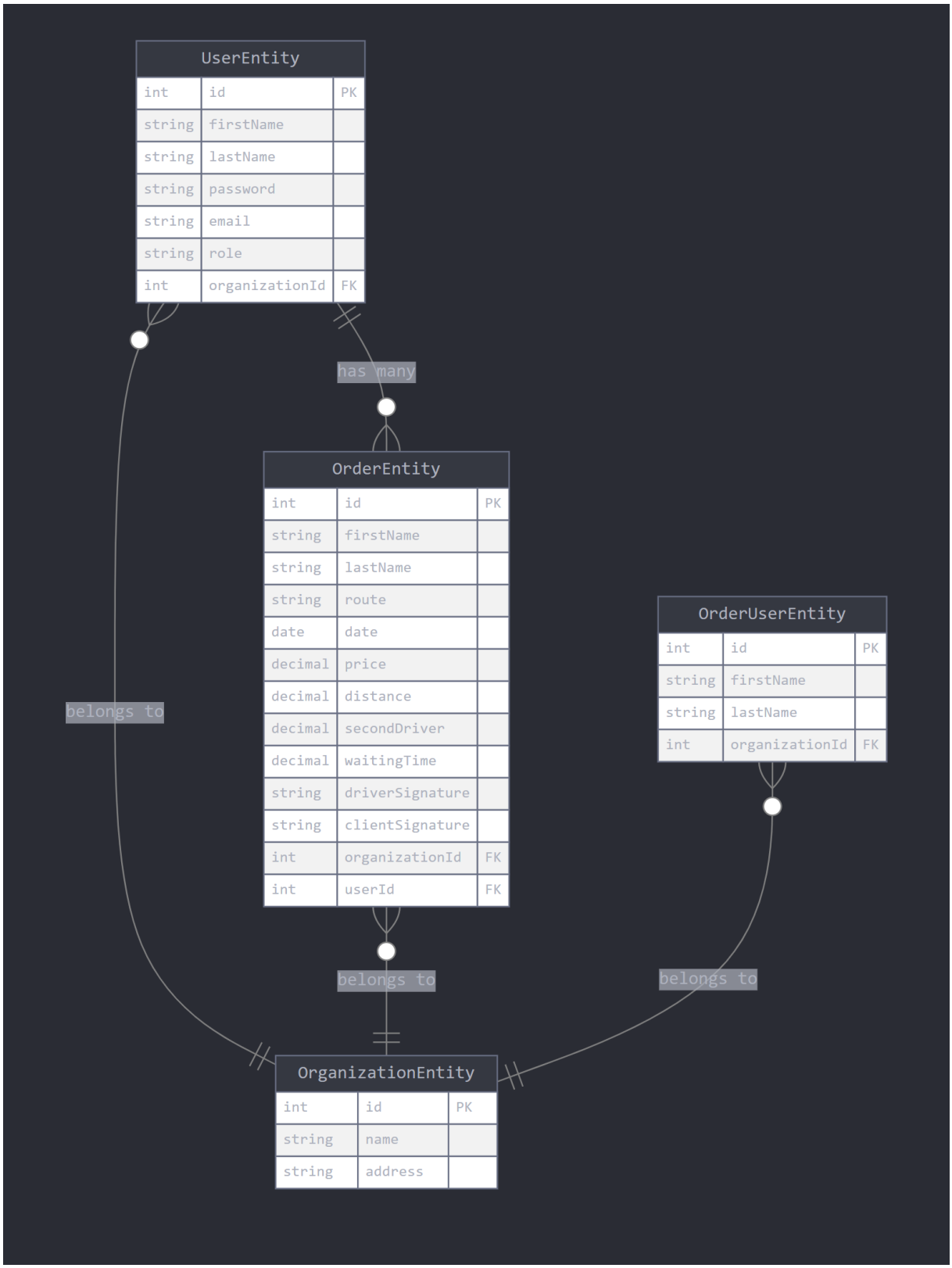
   Všetkých používateľov je umožnené vidieť iba administrátorovi. Manažér môže vidieť iba ním vytvorených používateľov. V prípade potreby je možnosť taktiež už existujúce objednávky upraviť alebo odstrániť.

5. *Zobrazenie histórie zákazníkov:* V tejto sekcií je zoznam zákazníkov na základe ich mena a priezviska. V prípade potreby je možné existujúci záznam zákazníkov odstrániť.

6. *Generovanie objednávky:* V prípade potreby je umožnené používateľom (administrátorom a manažérom) generovať výstupný súbor so zadanou objednávkou vo formáte PDF.

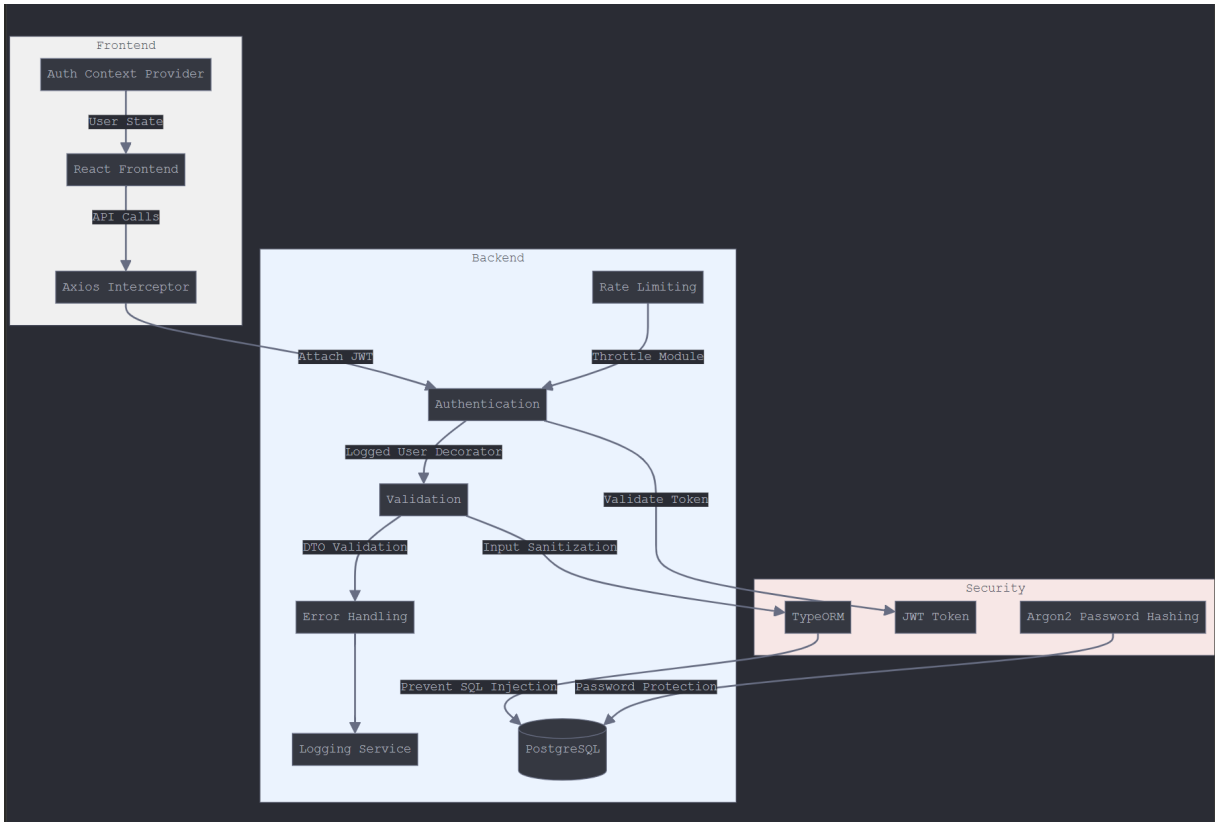


Obrázok 1: Možnosti použitia CRUD operácii prostredníctvom HTTP metód



Obrázok 2: Entitno-relačný diagram databázy systému

2.3 Zabezpečenie



Obrázok 3: Diagram zabezpečenia aplikácie

Systém implementuje viacero úrovní zabezpečenia a validácie pre zaistenie spoľahlivosti a bezpečnosti aplikácie:

2.3.1 Validácia vstupov

Validácia vstupných dát je implementovaná pomocou dekorátorov v DTO (Data Transfer Objects):

```
@IsNumber({ allowNaN: false, allowInfinity: false, maxDecimalPlaces: 2 })
@Min(0)
@Type(() => Number)
readonly price: number;

@IsNumber({ allowNaN: false, allowInfinity: false, maxDecimalPlaces: 2 })
@Min(0)
```

```
@Type(() => Number)
readonly distance: number;
```

2.3.2 Rate Limiting

Aplikácia používa ThrottlerModule pre obmedzenie počtu požiadaviek:

```
ThrottlerModule.forRoot([
    {
        name: 'short',
        ttl: 1000,
        limit: 3,
    },
    {
        name: 'medium',
        ttl: 10000,
        limit: 5,
    },
    {
        name: 'long',
        ttl: 60000,
        limit: 50,
    },
]);
```

2.3.3 Správa chýb a logovanie

```
[Nest] 61  - 11/21/2024, 9:48:18 AM     LOG [OrderService] Finding all orders for user
[Nest] 61  - 11/21/2024, 9:48:18 AM     LOG [OrderService] Found 0 orders
[Nest] 61  - 11/21/2024, 9:48:20 AM     LOG [OrganizationService] Returning 1 organizations
[Nest] 61  - 11/21/2024, 9:48:27 AM     LOG [UserService] Finding all users
[Nest] 61  - 11/21/2024, 9:48:31 AM     LOG [OrganizationService] Returning 1 organizations
[Nest] 61  - 11/21/2024, 9:48:42 AM     LOG [OrganizationService] Creating organization with name: Druha spolocnost
[Nest] 61  - 11/21/2024, 9:48:42 AM     LOG [OrganizationService] Returning 2 organizations
[Nest] 61  - 11/21/2024, 9:48:48 AM     LOG [OrderService] Finding all orders for user
[Nest] 61  - 11/21/2024, 9:48:48 AM     LOG [OrderService] Found 0 orders
[Nest] 61  - 11/21/2024, 9:48:52 AM     LOG [UserService] Finding all users
[Nest] 61  - 11/21/2024, 9:48:53 AM     LOG [OrganizationService] Returning 2 organizations
[Nest] 61  - 11/21/2024, 9:48:55 AM     LOG [UserService] Finding all users
[Nest] 61  - 11/21/2024, 9:49:01 AM     LOG [UserService] Updating user with id: 2
[Nest] 61  - 11/21/2024, 9:49:01 AM     LOG [UserService] User with id: 2 updated successfully
```

Obrázok 4: Logovanie

Implementovaný je komplexný systém spracovania chýb využívajúci vlastný HttpExceptionFilter:

```
@Catch(HttpException)
export class HttpExceptionFilter<T extends HttpException>
implements ExceptionFilter {
    constructor(private readonly errorService: ErrorService) {}
    private readonly logger = new Logger(HttpExceptionFilter.name);

    catch(exception: T, host: ArgumentsHost) {
        // Logovanie chyby
        this.logger.error('Error message:', exception.message);

        // Spracovanie odpovede
        const ctx = host.switchToHttp();
        const response = ctx.getResponse<Response>();

        // Odoslanie štruktúrovanej chybovej správy
        const status = exception.getStatus();
        const exceptionResponse = exception.getResponse();
        const error = typeof response === 'string'
            ? { message: exceptionResponse }
            : (exceptionResponse as object);

        response.status(status).json({
            ...error,
            timestamp: new Date().toISOString(),
        });
    }
}
```

2.3.4 Autentifikácia a autorizácia

Frontend používa Context API pre správu stavu autentifikácie a Axios interceptor pre automatické pripájanie JWT tokenov:

```
const axiosInstance = axios.create({
    baseURL: import.meta.env.VITE_BACKEND_URL
});

axiosInstance.interceptors.request.use(async (config) => {
    const accessToken = AppService.getToken();
    if (accessToken) {
        config.headers.Authorization = `Bearer ${accessToken}`;
    }
    return config;
});
```

Backend využíva vlastný LoggedInUser dekorátor pre prístup k autentifikovanému používateľovi:

```
export const LoggedInUser = createParamDecorator(
    (data: unknown, ctx: ExecutionContext) => {
        const request = ctx.switchToHttp().getRequest();
        return request.user;
    },
);
```

2.3.5 Hashovanie hesiel

Pre bezpečné ukladanie hesiel je použitá knižnica Argon2 s nasledujúcou konfiguráciou:

```
return await argon2.hash(password, {
    type: argon2.argon2id,
    memoryCost: 2 ** 16,
    timeCost: 4,
    parallelism: 2,
    hashLength: 32,
});
```

2.3.6 Kontajnerizácia

Aplikácia je kontajnerizovaná pomocou Docker s nasledujúcimi konfiguráciami pre frontend a backend:

```
version: "3.8"
services:
  frontend:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3001:3001"
    volumes:
      - .:/app
      - /app/node_modules
    environment:
      - CHOKIDAR_USEPOLLING=true
```

Backend Docker konfigurácia:

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE ${PORT}
CMD ["sh", "-c", "npm run typeorm:migration:run && npm run start:dev"]
```

Táto architektúra zabezpečuje komplexnú ochranu aplikácie na viacerých úrovniach, od validácie vstupov až po bezpečné ukladanie citlivých údajov.

2.3.7 Migrácie databázy

Pre správu databázových zmien a inicializáciu systému sú použité TypeORM migrácie. Príkladom je migrácia pre vytvorenie superadmin účtu:

```
export class CreateSuperAdmin1729937655585 implements MigrationInterface {
    public async up(queryRunner: QueryRunner): Promise<void> {
```

```
        const hashedPassword = await passwordHash('pass');
        await queryRunner.query(`
            INSERT INTO "user" (
                "firstName",
                "lastName",
                "email",
                "password",
                "role"
            )
            VALUES (
                'Super',
                'Admin',
                'admin@admin.com',
                '${hashedPassword}',
                'superAdmin'
            );
        `);
    }

    public async down(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(`
            DELETE FROM "user"
            WHERE "email" = 'admin@admin.com';
        `);
    }
}
```

Táto migrácia zabezpečuje:

- Vytvorenie počiatočného superadmin účtu v systéme
- Bezpečné hashovanie hesla pred uložením do databázy
- Možnosť rollback-u pomocou `down` metódy

2.3.8 Paginácia záznamov

Pre efektívne zobrazovanie väčšieho množstva dát je implementovaná paginácia záznamov. Napríklad pri objednávkach:

```
orders = await this.orderRepository.find({
    order: { date: 'DESC' },  // zoradenie podľa dátumu zostupne
    skip: page,               // preskočenie predchádzajúcich strán
    take: 20,                 // počet záznamov na stránku
});
```

Paginácia poskytuje:

- Obmedzenie počtu záznamov na stránku (20 záznamov)
- Zoradenie záznamov podľa dátumu od najnovších
- Efektívne načítavanie dát z databázy
- Optimalizovaný výkon pri veľkom množstve záznamov

3 Štruktúra back-end aplikácie

Back-end aplikácia je postavená s modulárnou architektúrou. Štruktúra a jej organizácia je popísaná v nasledovných v nasledovných podkapitolách.

3.1 Adresárová štruktúra

- **/src** - Hlavný zdrojový kód
  - **/auth** - Autentifikačný modul
  - **/common** - Zdieľané komponenty a utility
  - **/error** - Spracovanie chýb
  - **/migrations** - Databázové migrácie
  - **/order** - Modul pre správu objednávok
    - `/dto` - Data Transfer Objects
    - `/entities` - Databázové entity
    - `/mappers` - Mapovanie medzi DTO a entitami
    - `order.controller.ts` - REST endpointy
    - `order.service.ts` - Biznis logika
    - `order.module.ts` - Konfigurácia modulu
  - **/order_user** - Modul pre vzťahy medzi objednávkami a používateľmi
  - **/organization** - Modul pre správu organizácií
  - **/roles** - Správa rolí a oprávnení
  - **/setup** - Konfigurácia a inicializácia
  - **/user** - Modul pre správu používateľov
- **/test** - E2E testy
  - `/orders` - Testy pre objednávky
  - `/organizations` - Testy pre organizácie
  - `/users` - Testy pre používateľov
  - `app.e2e-spec.ts` - Hlavné integračné testy

3.2 Architektúra modulov

Každý modul je štruktúrovaný podľa NestJS konvencií a obsahuje:

1. **Controller** (`*.controller.ts`)
    - Definuje REST API endpointy
    - Spracováva HTTP požiadavky
    - Validuje vstupné dáta
2. **Service** (`*.service.ts`)
    - Obsahuje biznis logiku
    - Komunikuje s databázou
    - Spracováva dáta
3. **Module** (`*.module.ts`)
    - Definuje závislosti modulu
    - Konfiguruje providery
    - Spája komponenty
4. **DTO** (`/dto/*.dto.ts`)
    - Definuje štruktúru vstupných/výstupných dát
    - Obsahuje validačné pravidlá
5. **Entity** (`/entities/*.entity.ts`)
    - Reprezentuje databázové modely
    - Definuje vzťahy medzi entitami

3.3 Hlavné súbory aplikácie

- `main.ts` - Vstupný bod aplikácie
- `app.module.ts` - Hlavný modul aplikácie
- `app.controller.ts` - Hlavný controller

- `app.service.ts` - Hlavná service
- `data-source.ts` - Konfigurácia databázového pripojenia

3.4 Testovacia infraštruktúra

Backend obsahuje komplexnú testovaciu infraštruktúru:

- End-to-end testy pre každý modul
- Jednotkové testy pre služby a controllery
- Konfigurácia v `jest-e2e.json`
- Testovacie helpery v `/setup` adresári

Táto architektúra zabezpečuje:

- Vysokú modularitu a znovupoužiteľnosť kódu
- Jednoduchú údržbu a testovateľnosť
- Čistú separáciu záujmov (Separation of Concerns)
- Škálovateľnosť aplikácie
- Prehľadnú organizáciu kódu

4 Štruktúra front-end aplikácie

Frontend aplikácie je organizovaný do logicky oddelených adresárov, ktoré zabezpečujú prehľadnú a udržateľnú architektúru:

4.1 Hlavná štruktúra adresárov

- **/src** - Hlavný zdrojový kód aplikácie
  - **/api** - API klienti a integrácie
  - **/assets** - Statické súbory (obrázky, fonty)
  - **/components** - Znovupoužiteľné React komponenty
  - **/contexts** - React Context API definície
  - **/hooks** - Vlastné React hooks
  - **/layout** - Komponenty pre rozloženie stránky
  - **/lib** - Zdieľané knižnice a utility
  - **/menu-items** - Konfigurácia navigačného menu
  - **/pages** - React komponenty pre jednotlivé stránky
  - **/routes** - Definície a konfigurácia smerovania
  - **/services** - Biznis logika a služby
  - **/styles** - CSS štýly a témy
  - **/themes** - Konfigurácia tém aplikácie
  - **/utils** - Pomocné funkcie a nástroje
- **Konfiguračé súbory**
  - `App.jsx` - Hlavný React komponent
  - `config.js` - Globálna konfigurácia aplikácie
  - `constants.js` - Konštanty aplikácie
  - `index.jsx` - Vstupný bod aplikácie
  - `vite-env.d.js` - TypeScript deklarácie pre Vite

4.2 Konfiguračné súbory

- `.env` a `.env.example` - Environmentálne premenné
- `.eslintrc` - Konfigurácia ESLint
- `.prettierrc` - Konfigurácia Prettier
- `docker-compose.yml` a `Dockerfile` - Kontajnerizácia
- `package.json` - NPM závislosti a skripty
- `vite.config.mjs` - Konfigurácia Vite

4.3 Organizácia kódu

Aplikácia využíva modulárny prístup s nasledujúcimi princípmi:

1. **Komponentová architektúra**
    - Znovupoužiteľné komponenty v `/components`
    - Stránkové komponenty v `/pages`
    - Layoutové komponenty v `/layout`
2. **Oddelenie zodpovedností**
    - Biznis logika v `/services`
    - API volania v `/api`
    - Pomocné funkcie v `/utils`
3. **State Management**
    - React Context API v `/contexts`
    - Vlastné hooks v `/hooks`
4. **Routing a navigácia**
    - Definície ciest v `/routes`
    - Konfigurácia menu v `/menu-items`

Táto štruktúra podporuje:

- Jednoduchú údržbu a rozšíriteľnosť
- Prehľadnú organizáciu kódu
- Efektívnu spoluprácu v tíme
- Rýchle nájdenie relevantných súborov
- Jasné oddelenie zodpovedností

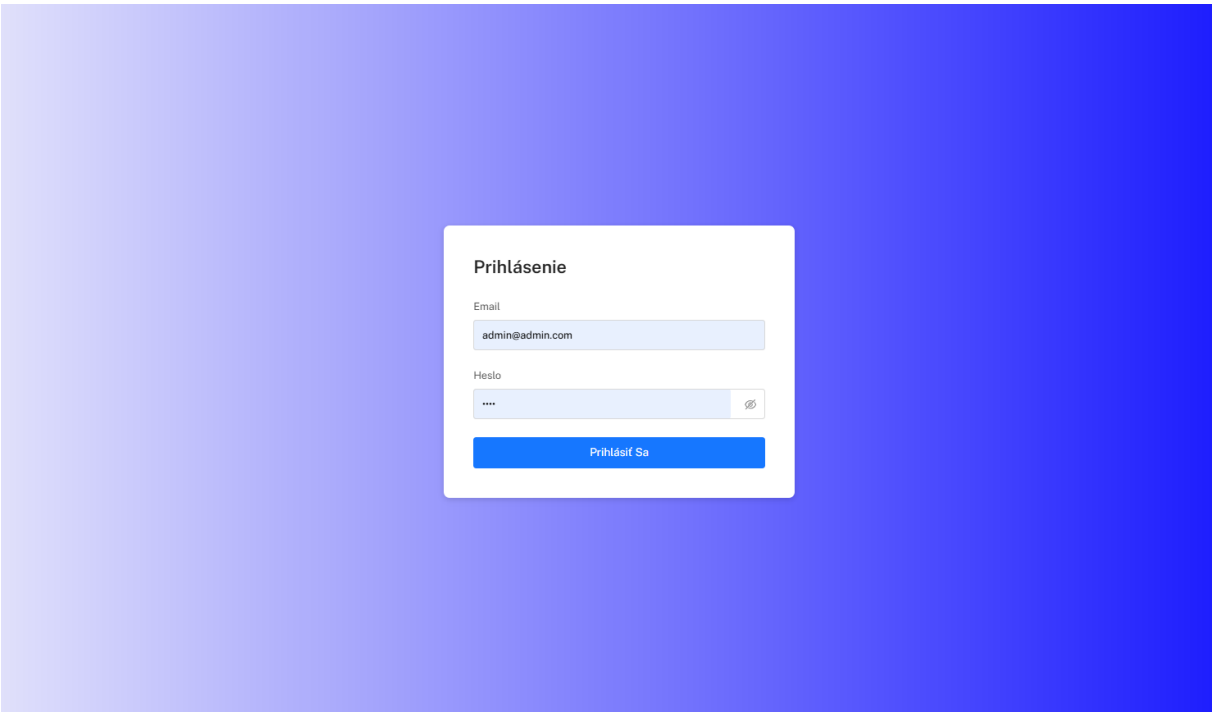
5 Používateľské rozhranie

Používateľské rozhranie, resp. front-end časť nášho systému, bolo vytvorené prostredníctvom JavaScript s použitím frameworku React.js[13], ktorý bol použitý pre rozdelenie do komponentov, pre lepšiu správu a údržbu kódu.

Pre UI (user interface) dizajn bola použitá React.jd knižnica Material-UI, ktorá uľahčuje prácu s UI dizajnom bez nutnosti písania veľkého množstva vlastného CSS a ktorá je založená na filozofii Material Design od spoločnosti Google.

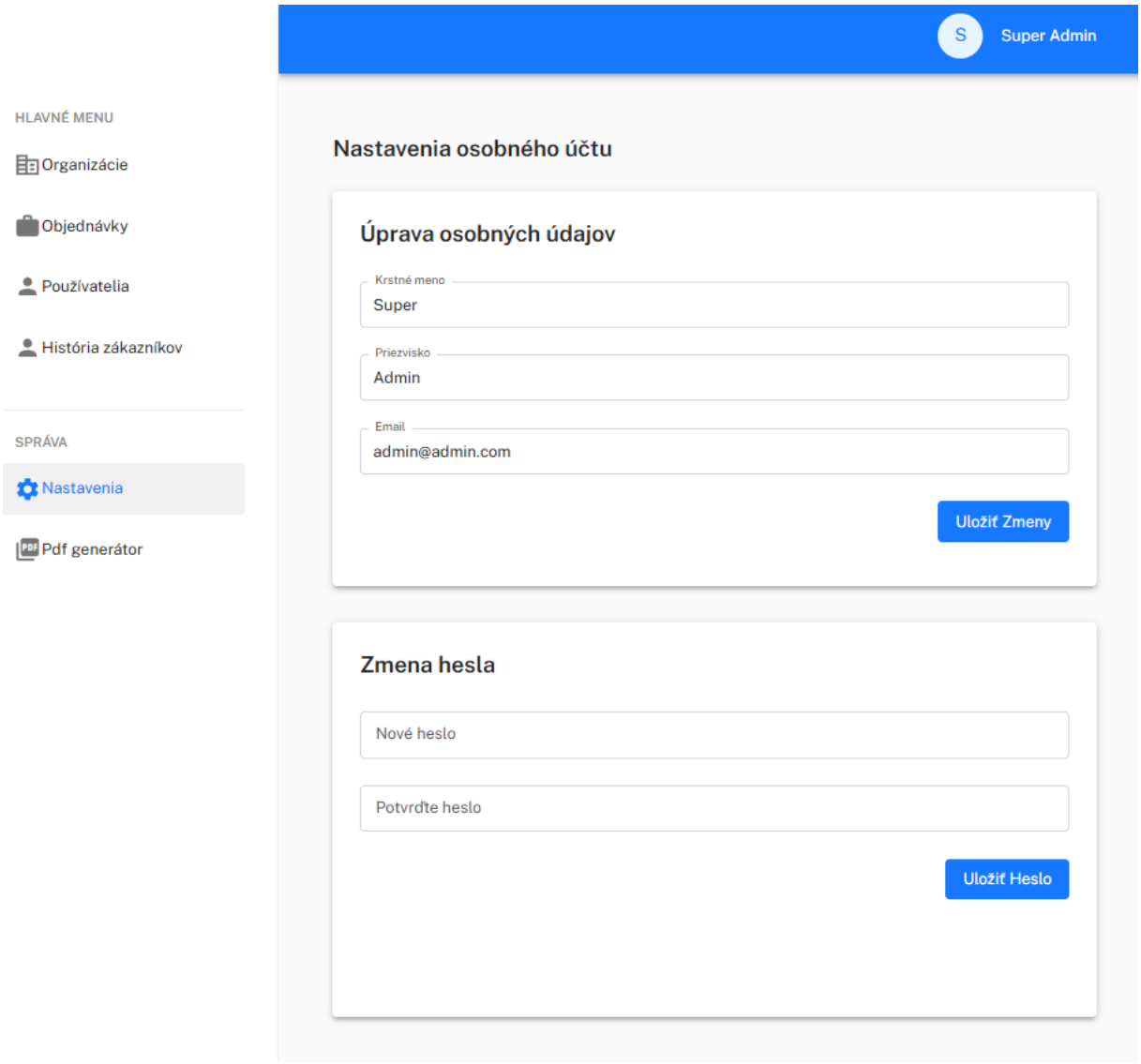
Pre správu našej aplikácie bol použitý nástroj Vite[14], ktorý umožňuje efektívnejší vývoj aplikácie.

Taktiež je potrebné dodať, že naša aplikácia je responzívna a prispôsobená na zobrazovanie aj na mobilných zariadeniach. Nižšie ukážeme zopár obrázkov používateľského rozhrania.

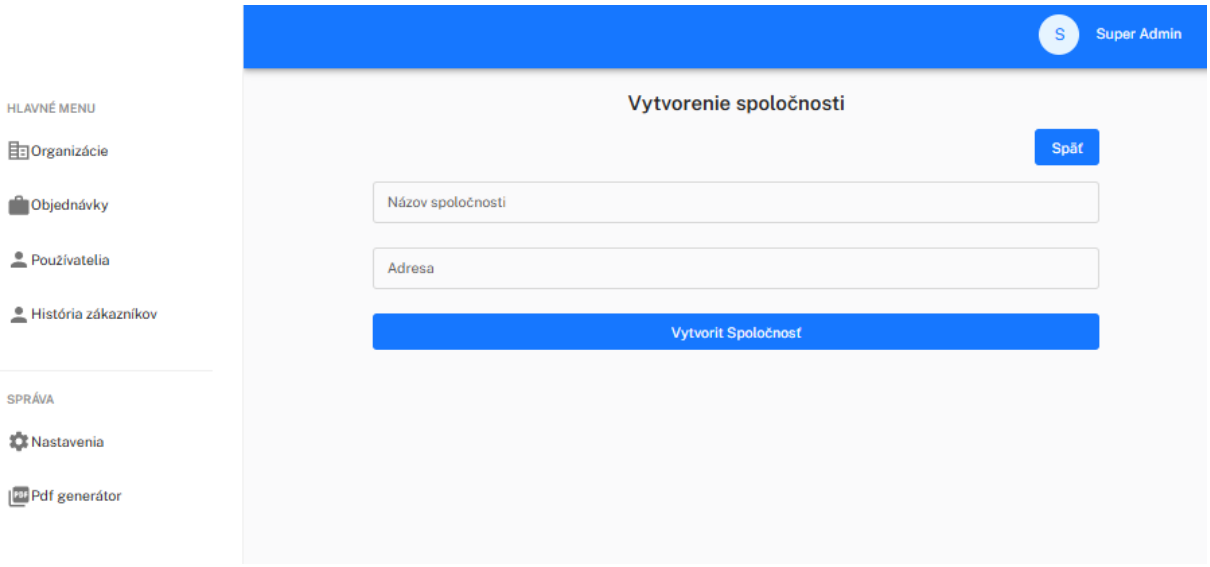


Obrázok 5: Hlavná (prihlasovacia) stránka

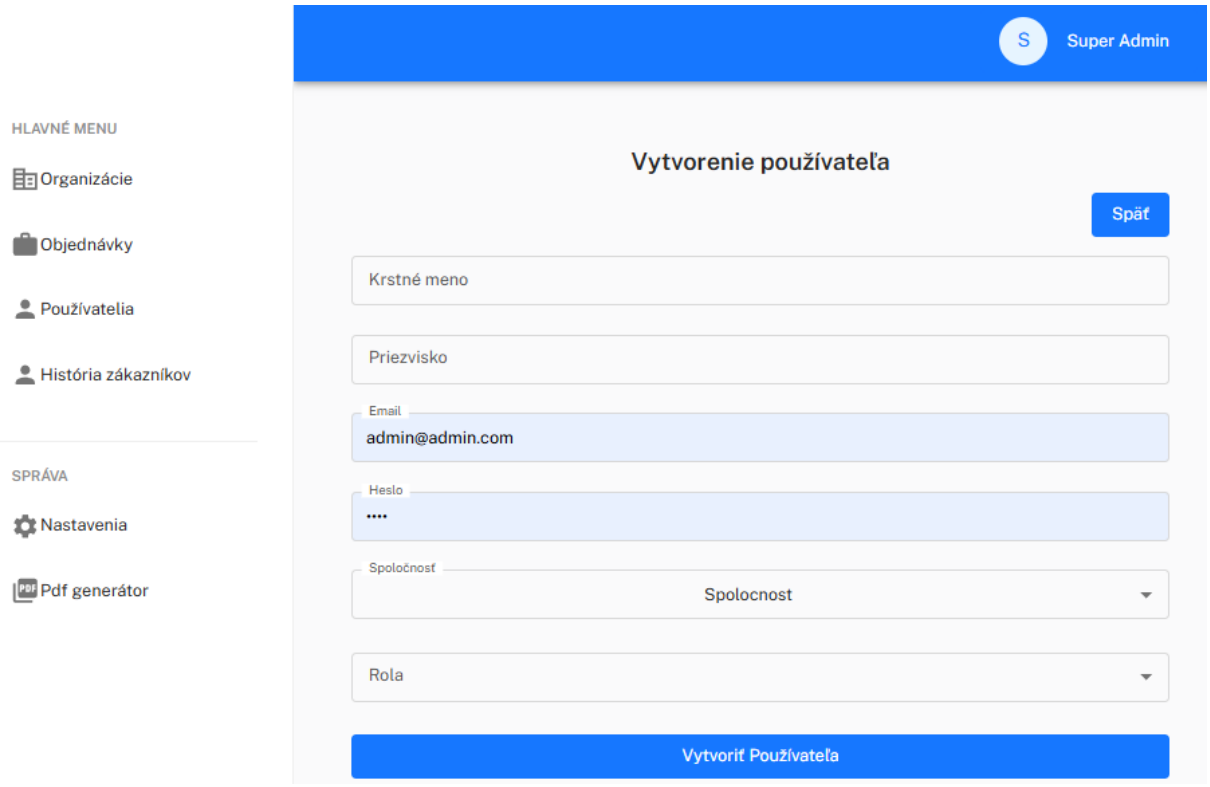
[13]https://react.dev/
[14]https://vite.dev/



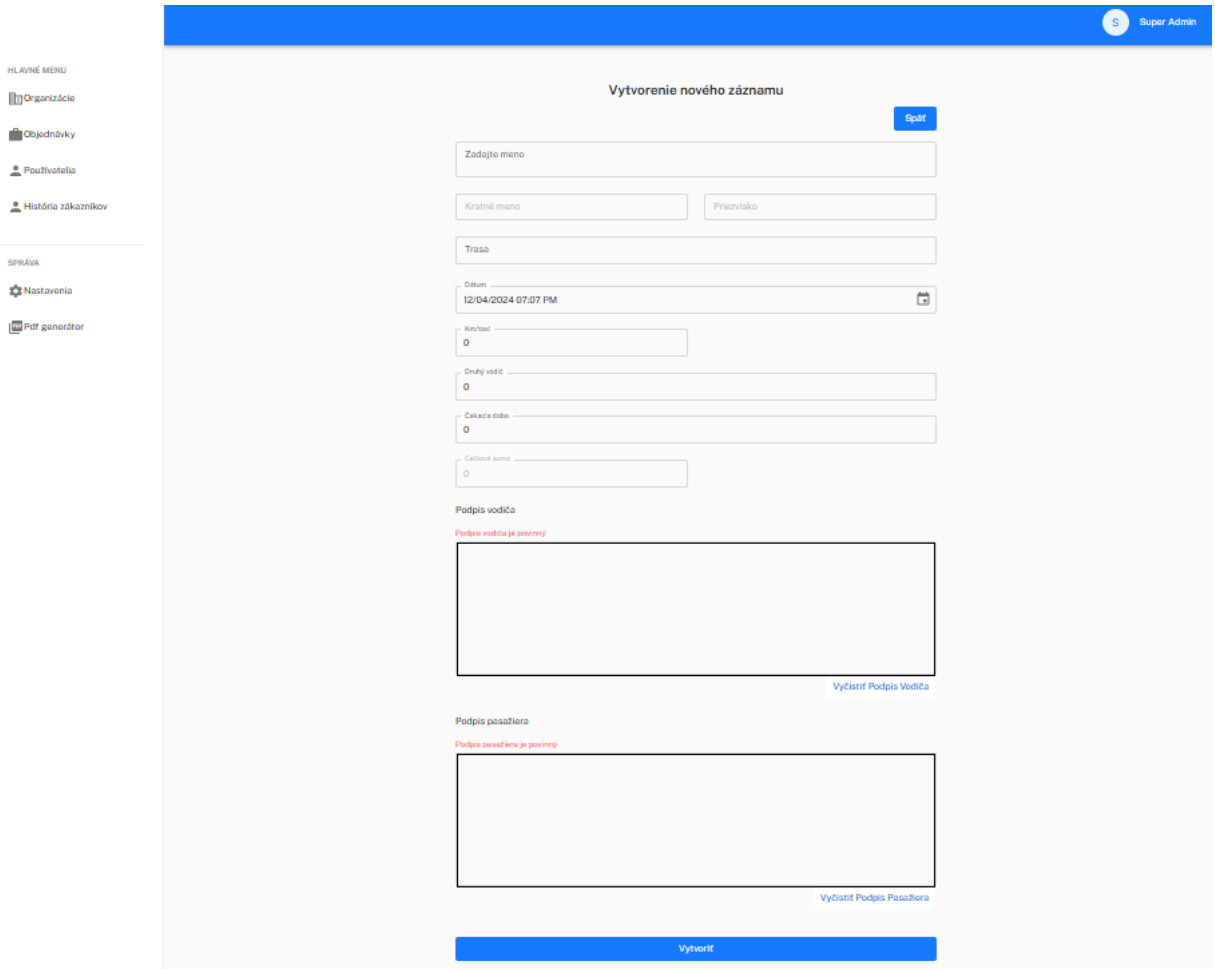
Obrázok 6: Správa účtu používateľa



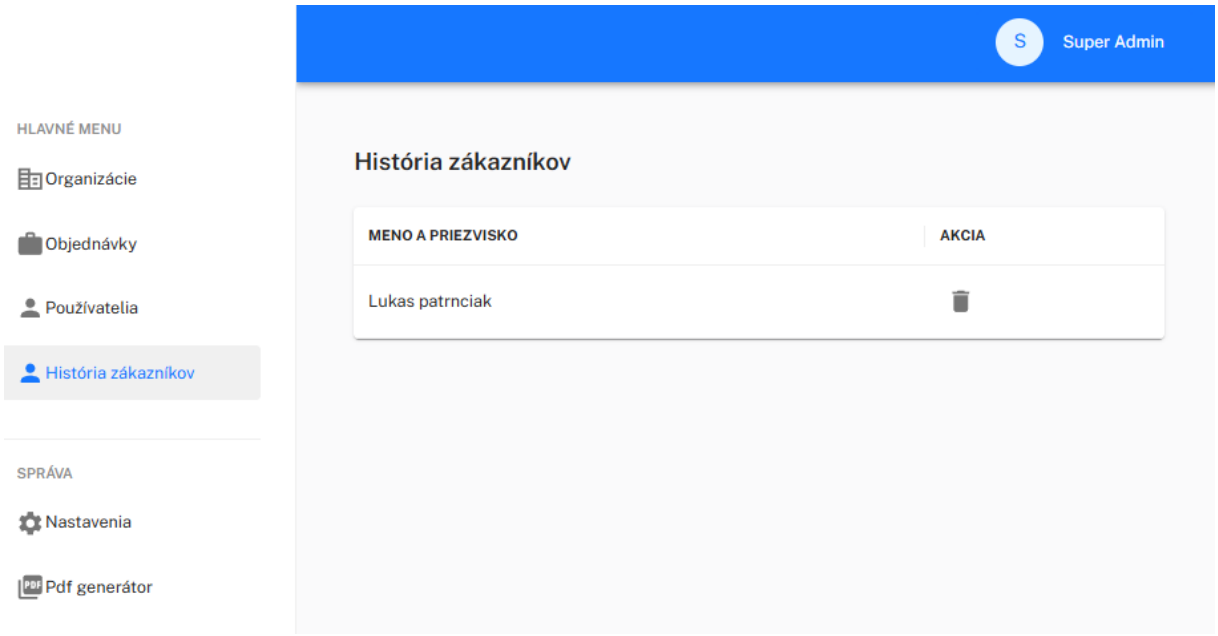
Obrázok 7: Pridanie taxislužby



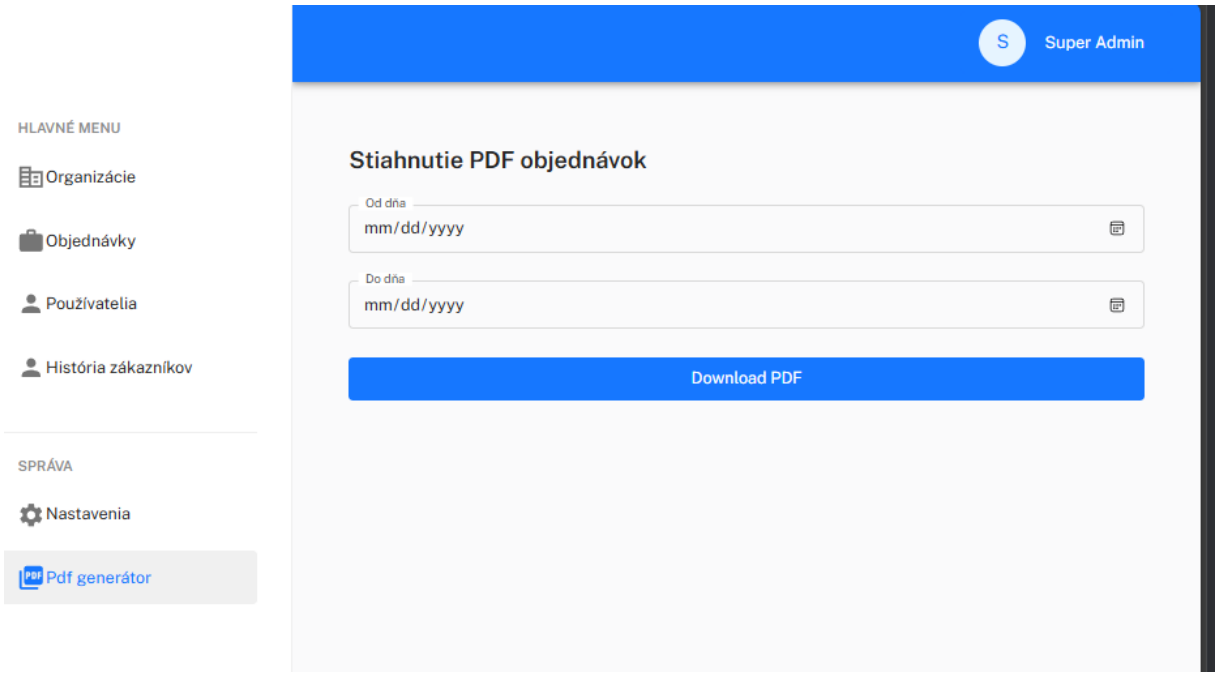
Obrázok 8: Pridanie nového používateľa



Obrázok 9: Vytvorenie novej objednávky



Obrázok 10: História zákazníkov taxislužby



Obrázok 11: Generovanie PDF súboru objednávky

Taxi sluzba FEI

Výkaz za mesiac 11.2024

Objednávka/Order č: 1

Dátum/Date: 14. 11. 2024
Čas/Time: 12:54
Hlavný vodič/Main driver: fei fei
Pasažier/Passenger: fei fei
Celková suma/Total price: 4.00€
Km/taxi: 2.00 km
Trasa/Route: Tokyo - Osaka
Druhý vodič/Second driver: 2.00
Čakanie/Waiting: 0.00 min
Podpis pasažiera/Passenger signature:

Podpis vodiča/Driver signature:

Obrázok 12: Vygenerovaný PDF súbor objednávky

6 Testovanie

Testovanie aplikácie je implementované pomocou end-to-end (E2E) testov, ktoré overujú funkcionalitu celého systému od frontendu až po backend. Testy sú písané v TypeScript s využitím frameworku Jest a knižnice Supertest pre HTTP požiadavky.

6.1 Architektúra testov

Testovacia architektúra je postavená na niekoľkých kľúčových komponentoch:

1. **SetupHelpers** - pomocná trieda pre inicializáciu testovacieho prostredia
2. **SetupTestingEntitiesHelpers** - trieda pre vytváranie testovacích entít
3. **TestingModule** - NestJS modul špecifický pre testovanie

6.2 Príprava testovacieho prostredia

Pred spustením testov je potrebné pripraviť testovacie prostredie. Toto zahŕňa:

1. Vytvorenie testovacieho modulu
2. Inicializáciu databázy
3. Vytvorenie testovacích užívateľov a tokenov
4. Prípravu testovacích dát

```
beforeAll(async () => {
    testHelper = new SetupHelpers();
    const setupData: SetupTestingData = await testHelper.createSetupApp({
        testName: TEST_NAME,
        organizations: {
            count: ORGANIZATIONS_COUNT,
        },
        users: {
            count: 3,
        },
    });
```

```
    app = setupData.app;
    httpServer = app.getHttpServer();
    data = setupData.data;
    superAdminToken = setupData.tokens.superAdminToken;
});
```

6.3 Príklad testovacieho scenára

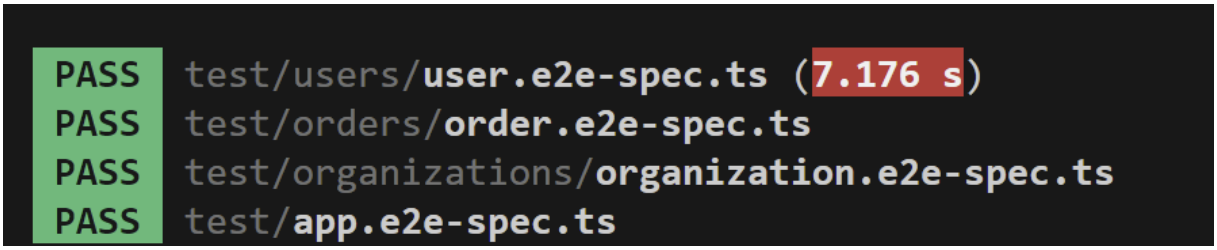
Nižšie je uvedený príklad testovania CRUD operácií pre organizácie:

```
describe('Organization Testing', () => {
    it('Create Organization [POST /organization]', async () => {
        const createOrganizationDto: CreateOrganizationDto = {
            name: ORGANIZATION_NAME,
            address: ORGANIZATION_ADDRESS,
        };

        return request(app.getHttpServer())
            .post('/organization')
            .send(createOrganizationDto)
            .auth(superAdminToken, { type: 'bearer' })
            .expect(HttpStatus.CREATED);
    });

    it('Find All Organizations [GET /organization]', async () => {
        return request(app.getHttpServer())
            .get('/organization')
            .auth(superAdminToken, { type: 'bearer' })
            .expect(HttpStatus.OK);
    });
});
```

6.4 Pokrytie testami



Obrázok 13: Výsledky testovania

Aplikácia obsahuje komplexné E2E testy pre nasledujúce moduly:

- `user.e2e-spec.ts` - testovanie používateľských operácií
- `order.e2e-spec.ts` - testovanie objednávok
- `organization.e2e-spec.ts` - testovanie organizácií
- `app.e2e-spec.ts` - testovanie základnej funkcionality aplikácie

6.5 Spustenie testov

Spustenie end-to-end testov je možné nasledovným príkazom:

```
$ npm run test:e2e
```

Testy využívajú vlastnú testovaciu databázu, ktorá je pred každým spustením vymazaná a nanovo inicializovaná, čím sa zabezpečuje konzistentnosť testovacieho prostredia:

```
if (process.env.NODE_ENV === 'test') {
    const connection = app.get(Connection);
    await connection.dropDatabase();
    await connection.synchronize(true);
}
```

6.6 Automatizácia testov

Každý test automaticky:

- Vytvorí čisté testovacie prostredie
- Pripraví potrebné testovacie dáta
- Vykoná testovací scenár
- Overí očakávané výsledky
- Vyčistí testovacie prostredie

Toto zabezpečuje izoláciu testov a konzistentnosť výsledkov pri opakovanom spúšťaní.