Aplikacija za upravljanje pametnom kućom

Koščak, Kristijan

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:543138

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-27**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

APLIKACIJA ZA UPRAVLJANJE PAMETNOM KUĆOM

Diplomski rad

Kristijan Koščak

Osijek, 2021.

SADRŽAJ

1. UVOD 1

1.1. Zadatak diplomskog rada 1

2. PREGLED PODRUČJA 2

2.1. Općenito 2

2.2. Home Assistant 2

2.3. Homebridge 3

3. PRIMJENJENE TEHNOLOGIJE I ALATI 4

3.1. Spring Boot 4

3.1.1. Spring Data JPA i Hibernate 5

3.1.1.1. Jedan na prema jedan anotacija 6

3.1.1.2. Jedan na prema više anotacija 7

3.1.1.3. Više na prema više anotacija 7

3.1.2. Spring zaštita i JWT 8

3.1.3. REST API 8

3.1.4. MapStruct 9

3.1.5. Flyway 10

3.2. Angular 11

3.2.1. Komponente 11

3.2.2. Moduli 12

3.2.3. Usmjeravanje 12

3.2.4. Servisi 13

3.2.5. Presretači 13

3.2.6. Angular material 14

3.2.7. Direktive 14

3.2.8. Animacije 15

3.2.9. SCSS 15

3.3. Raspberry Pi 16

3.3.1. Općenito 16

3.3.2. Povezivanje sa udaljenim računalom 16

3.3.3. Biblioteka gpiozero 17

3.3.4. Flask 17

3.3.5. Ngrok 17

3.4. PostgreSQL 18

3.5. Docker 18

3.6. IntelliJ 18

4. REALIZACIJA RADA 19

4.1. Baza podataka 19

4.2. Serverski dio 21

4.2.1. Struktura projekta 22

4.2.2. Kontroleri 23

4.2.3. Servis 25

4.2.4. Repozitorij 26

4.2.5. Maperi ... 27
4.2.6. Zaštita aplikacije ... 27
4.3. Klijentski dio ... 29
4.3.1. Struktura projekta ... 29
4.3.2. Moduli ... 30
4.3.3. Usmjeravanje ... 31
4.3.4. Komponente ... 32
4.3.5. Servisi ... 35
4.3.6. Presretači ... 36
4.3.7. Zaštita aplikacije ... 38
4.4. Mikroračunalo ... 39
4.5. Izgled aplikacije ... 41
5. ZAKLJUČAK ... 45
LITERAURA ... 46
SAŽETAK ... 48
ABSTRACT ... 49
ŽIVOTOPIS ... 50
PRILOZI ... 51

1. UVOD

Zbog kontinuiranog razvoja računala, ona su postala sveprisutna. Različite tehnologije koje se danas primjenjuju nastale su zbog razvoja računala. Jedna od tih su i informacijske tehnologije koje su postale nezamisliv dio ljudskog života. Informacijske tehnologije predstavljaju upravljanje programskom opremom i računalnom sklopovskom podrškom gdje se računala koriste za prikupljanje, obradu, zaštitu i pohranu podataka [1]. Nadalje, razvojem računala pojavila su se i mikroračunala. To su mala, pristupačna računala koja se prema Von Neumannu sastoje od procesora, memorije i ulazno-izlaznog podsustava koji su međusobno povezani [2]. Jedno od takvih korišteno je u ovom diplomskom radu. Ono služi kao web aplikacija putem koje će se upravljati spojenim uređajima unutar pojedine kuće. Web aplikaciji pristupamo preko korisničkog sučelja koje će se kreirati i na kojemu korisnik može upravljati kućom. Podatke o svakoj kući, korisnicima, sobama, uređajima, statistiku, spremamo u bazu koristeći pozadinsku logiku cjelokupnog rada. Ideja je kreirati kompletnu aplikaciju za upravljanje pametnim kućama uz minimalne osnovne funkcionalnosti i jednostavno korisničko sučelje. Dokumentacija ovog diplomskog rada sadrži nekoliko poglavlja uključujući i ovo. Unutar poglavlja „PREGLED PODRUČJA“ daje se uvodna riječ o istom i već postojeća rješenja koja su slična ovome. Poglavljem „PRIMJENJENE TEHNOLOGIJE I ALATI“ opisuju se korištene tehnologije i alati u izradi ovoga rada, a strukturirane su u potpoglavlja. Svako od potpoglavlja se dijeli opet u potpoglavlja koja predstavljaju bitne koncepte svake tehnologije ili alata. Sljedeće poglavlje „REALIZACIJA RADA“ strukturno opisuje postupke izvedbe svakog dijela ovog rada (baza podataka, sučelje, pozadinska logika i mikroračunalo) i izgled konačnog rezultata. Nakon toga slijedi poglavlje „ZAKLJUČAK“ koje predstavlja rezimirano poglavlje unutar kojeg se daje osvrt na ciljeve i postignute rezultate.

1.1. Zadatak diplomskog rada

Zadatak diplomskog rada je razviti web aplikaciju za upravljanje pametnom kućom, koja će se moći pokretati na mobilnim i desktop uređajima. Započinje izradom korisničkog sučelja, a zatim se nastavlja kroz razvoj pozadinske logike same aplikacije. Mikroračunalo ćemo pretvoriti u internetski server, putem kojeg ćemo inicijalizirati uređaje i kontrolirati pojedini. Aplikacija ima jednu vrstu korisnika, koji može dodati kuću, i druge korisnike koji mogu pristupiti istoj, sobe te uređaje u pojedinoj sobi. Sučelje ima kontrolnu ploču preko koje korisnik može kontrolirati pojedini uređaj. Također, za svaku pojedinu kuću nudi se i statistički prikaz korištenih uređaja na dnevnoj, tjednoj i mjesečnoj razini.

2. PREGLED PODRUČJA

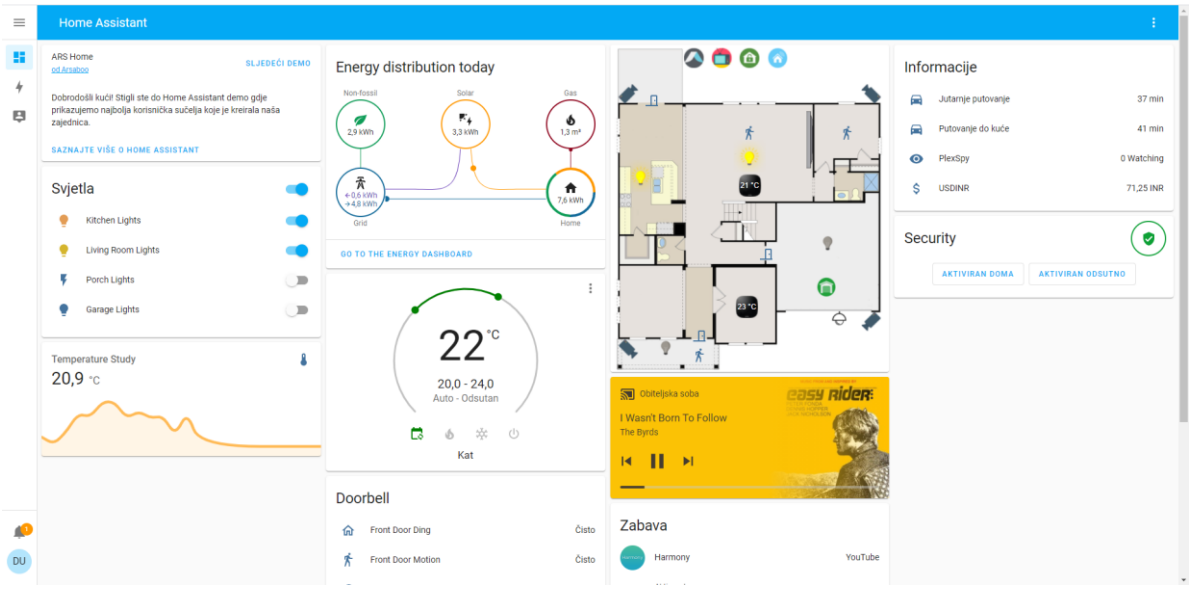
U ovom poglavlju, objasnit ćemo što je to pametna kuća te se dotaknuti postojećih sustava/aplikacija za upravljanje pametnim kućama.

2.1. Općenito

Pametna kuća je koncept koji korisnicima omogućava kontrolu i upravljanje kućanskim aparatima i uređajima. Upravljanje može biti ručno ili online upravljanje putem aplikacije [3]. Nastoji pružiti mogućnost pristupa s bilo kojeg mjesta koristeći bilo koji uređaj povezan na Internet. U pametnoj kući, svi uređaji su spojeni na jedan centralni uređaj koji predstavlja pristupnu točku same kuće [4].

2.2. Home Assistant

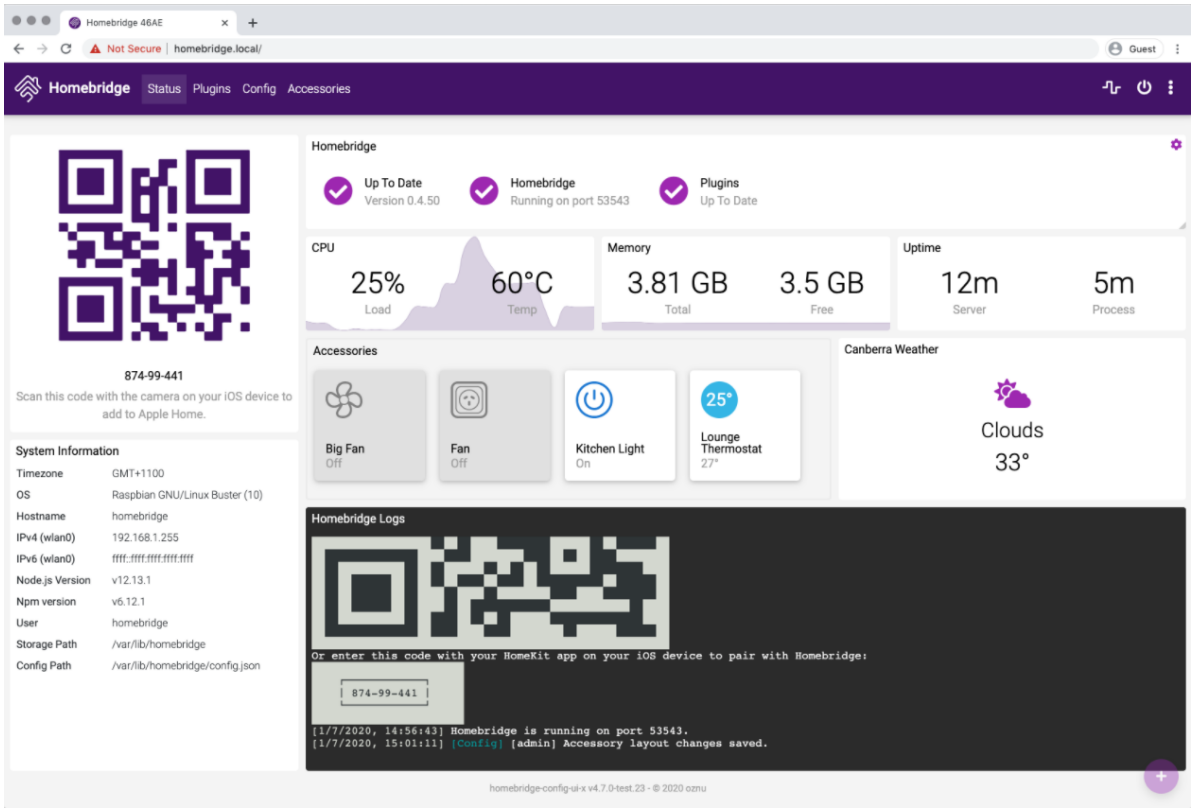
Home Assistant je besplatna aplikacija otvorenog koda koja se koristi za automatizaciju kuće. Pruža mogućnost lokalnog upravljanja kućnim uređajima i aparatima, bolju sigurnost, veću pouzdanost i veću fleksibilnost. Omogućava upravljanje bez oslanjanja na servere i internetsku konekciju što rezultira bržim operacijama i odzivima. Home Assistant pronalazi sve uređaje na Wi-Fi- mreži te se povezuje s onima koje može kontrolirati i pruža jasno sučelje za upravljanje. Pokreće se na lokalnom serveru ili na mikroračunalu Raspberry Pi koje zahtjeva Hassio operacijski sustav koji omogućuje softveru da iskoristi sve svoje resurse [5].



Slika 2.1. Prikaz demo sučelja Home Assistant-a [6]

2.3. Homebridge

Homebridge je NodeJS server koji je moguće pokrenuti na kućnom internetu. Oponaša iOS HomeKit API. Podržava mnoštvo dodataka koji predstavljaju most između HomeKit-a i raznih API-a pruženih od strane proizvođača pametnih uređaja [7]. HomeKit API predstavlja softverski okvir tvrtke Apple koji korisnicima omogućuje konfiguriranje, komunikaciju i upravljanje pametnim uređajima [8]. HomeBridge je moguće pokrenuti na Linux-u, macOS-u, Windows-u, ali i na mikroračunalu Raspberry Pi. Nakon uspješnog instaliranja, stvara se i korisničko sučelje, gdje korisnik može upravljati dodacima i uređajima.



Slika 2.2. Prikaz Homebridge korisničkog sučelja [9]

3. PRIMJENJENE TEHNOLOGIJE I ALATI

U ovom poglavlju, objašnjenje su tehnologije i alati koje se koriste u ovome radu. Kao softver za razvoj ovog diplomskog rada, korišten je IntelliJ integrirano razvojno okruženje. Sam rad sastoji se od dva dijela, korisničkog sučelja (engl. *Frontend*) i koda koji se odvija na serveru (engl. *Backend*). Korisničko sučelje je razvijeno koristeći Angular razvojno okruženje i programski jezik Typescript. Osim TS-a, korišten je i HTML prezentacijski jezik za kreiranje web stranica i SCSS. CSS je jezik koji se koristi za stiliziranje HTML elemenata, dok je SCSS nadogradnja na CSS.

3.1. Spring Boot

Spring je jedan od najpopularnijih razvojnih okruženja koje pruža opsežan model programiranja i konfiguracije za moderne Java aplikacije na bilo kojoj platformi za implementaciju [10]. Razvojem novih funkcionalnosti isti je postao kompleksan. To je zahtijevalo znatno ulaganje vremena da bi se kreirao novi projekt i postavila konfiguracija. Kako bi se skratilo to vrijeme i usmjerilo na sam razvoj aplikacija, predstavljen je Spring Boot [11].

Spring Boot je okvir kreiran na već spomenutom razvojnom okruženju. Omogućava sve značajke kao i Spring, ali ga je znatno jednostavnije koristiti. Skraćuje kod i pruža jednostavniji način za kreiranje samostalnih aplikacija s ponešto ili gotovo ništa konfiguracije u kratkom vremenu [12]. Jedan od ugrađenih servera koji se nude i koji je korišten u ovome radu je Tomcat. Kod Spring-a to nije slučaj jer se zahtjeva postavljanje servera ručno. Automatsko konfiguriranje je jedna od glavnih značajki ovog okvira. Potrebno je koristiti odgovarajuću konfiguraciju za korištenje određene funkcionalnosti. Izgradnja i konfiguriranje aplikacije pojednostavljuje se početničkim ovisnostima [13]. Jedna od takvih je prikazana slikom ispod (Slika 3.1.). Ona nam je potrebna za izgradnju web aplikacija.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Slika 3.1. Starter web ovisnost

3.1.1. Spring Data JPA i Hibernate

JPA [1]se koristi u Java aplikacijama kao specifikacija za upravljanje relacijskim podacima. Prati objektno relacijsko mapiranje (ORM[2]). Pruža API za upravljanje entitetima (engl. *Entity manager*) za obradu upita i transakcije na objektima prema bazi podataka [14]. Omogućava jednostavno kreiranje spremnika podataka gdje možemo kreirati naše metode za pronalaženje podataka, a Spring će kompletnu implementaciju omogućiti automatski [15]. Na slici 3.2. prikazana je potrebna ovisnost koju smo uključili prilikom kreiranja projekta. Koristi se kako bi omogućila gore navedeno i hibernate funkcionalnosti.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Slika 3.2 Ovisnost za dodavanje Spring data JPA

Hibernate je besplatni objektno relacijski servis visokih performansi. Omogućava mapiranje iz Java klasa u entitete baze podataka (uključujući i tipove podataka) i obrnuto. Osim navedenog, pruža mogućnost upita i pretraživanja podataka [16]. Predstavlja ORM alat koji omogućava JPA. Za mapiranje entiteta u objekte Java klase koristimo anotaciju @Entity i @Table unutar koje je potrebno navesti ime tablice na koju se odnosi. Također, treba obratiti pozornost na mala i velika slova budući da je osjetljivo na to. Svojstvima objekata možemo pridružiti vrijednosti iz kolone u tablici koristeći @Column anotaciju i ime kolone. Primjer takve jedne klase koja predstavlja mapiranu klasu iz baze vidimo na slici 3.3.

[1] engl. *Java Persistance API* – niz specifikacija za upravljanje vezom Java objekt – entitet baze podataka.
[2] engl. *Object-Relation mapping* - objektno relacijsko mapiranje.

```java
@Entity
@Table(name = "khouseuser")
public class KHouseUser extends BaseEntity{

    @Column(name = "name")
    private String name;

    @Column(name = "surname")
    private String surname;

    @Column(name = "email")
    private String email;

    @Column(name = "password")
    private String password;

}
```

Slika 3.3. Mapirana klasa

Neke od korisnih anotacija, koje se također mogu koristiti odnose se na relacije između entiteta. Tu spadaju @OneToOne, @OneToMany i @ManyToMany anotacija.

3.1.1.1. Jedan na prema jedan anotacija

Anotacija jedan na prema jedan (engl. *One to one*) se ne koristi u ovome radu, stoga ćemo je objasniti uz pomoć jednostavnog primjera. Račun ima kolonu koja se odnosi na ID radnika. Unutar klase radnika koristimo anotaciju @JoinColumn(name="ime_kolone") te iznad toga koristimo @OneToOne anotaciju za svojstvo worker koje je tipa Worker. Unutar klase računa koristimo također @OneToOne anotaciju, ali sa atributom mappedBy koje predstavlja ime svojstva unutar klase račun (engl. *Worker*). Primjer je prikazan slikom 3.4.

```java
@{...}
class Bill extends BaseEntity {
    // other columns...
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "worker_id")
    private Worker worker;
}
@{...}
class Worker extends BaseEntity {
    // other columns...
    @OneToOne(mappedBy = "worker")
    private Bill bill ;
}
```

Slika 3.4. Primjer @OneToOne anotacije

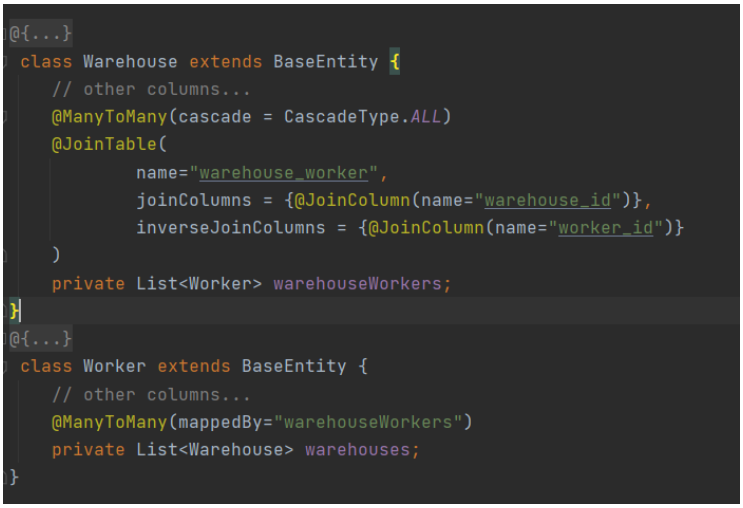
3.1.1.2. Jedan na prema više anotacija

Anotacija jedan na prema više (engl. *One to many*) se koristi u ovom radu i to na više entiteta. Svaki uređaj svojstvo room tipa Room, tj. sobe kojoj pripada. Nad tim svojstvom koristimo @ManyToOne anotaciju i @JoinColumn(name="room_id"). Unutar sobe se nalazi svojstvo lista uređaja iznad koje koristimo @OneToMany(mappedBy="room") anotaciju. Prema slici 3.5. vidimo kako vrijednost svojstva mappedBy odgovara imenu svojstva unutar uređaja (room).

Slika 3.5. Primjer @OneToMany anotacije

3.1.1.3. Više na prema više anotacija

Anotacija više na prema više (engl. *Many to many*) se također ne koristi u ovom radu, ali ćemo ju objasnit koristeći sličan primjer iz @OneToOne anotacije. Ova anotacija zahtijeva postojanje dodatne tablice koja će predstavlja vezu između dva entiteta, a sadrži glavne ključeve povezanih tablica. Unutar Warehouse entiteta koristimo anotaciju @JoinTable gdje se referenciramo na novu tablicu i nazive kolona unutar kojih su ključevi. Unutar Worker klase koristimo @ManyToMany anotaciju s atributom mappedBy koji sadrži ime varijable radnika iz klase skladišta. Navedeno je vidljivo na slici ispod (Slika 3.6.).



Slika 3.6. Primjer @ManyToMany anotacije

3.1.2. Spring zaštita i JWT

Spring zaštita (engl. *Spring Security*) je moćno i prilagodljivo razvojno okruženje za autentifikaciju i kontrolu pristupa. Predstavlja standard za zaštitu Spring aplikacija koji pruža autentifikaciju i autorizaciju [17]. Također je jedan od početnih ovisnosti koje se koriste kod kreiranja projekta. Uključen je koristeći sljedeću ovisnost prikazanu na slici 3.7. Uključivanjem, na raspolaganju imamo mnoštvo klasa i metoda koje se mogu primijeniti za autorizaciju i autentifikaciju.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Slika 3.7. Ovisnost Spring security

Radi dodatne sigurnosti ovaj rad uključuje JWT[3]. Prilikom uspješnog registriranja ili prijave, korisnik dobije token koji se kreira na osnovu email-a i definiranog tajnog ključa. Tajni ključ služi kako bi zaštitio token od izmjene ili nedozvoljenog izvlačenja podataka. Također, postavlja se i vrijeme istjecanja tokena. Taj token koristiti će se u zaštitnoj konfiguraciji kako bi se spriječio nedozvoljeni pristup kontrolerima. Kako bi ga koristili, potrebno je u pom.xml datoteku dodati ovisnost na slici 3.8.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

Slika 3.8. Prikaz ovisnosti za JWT

3.1.3. REST API

Spring Boot pruža jako dobru podršku za izgradnju web aplikacija. REST predstavlja arhitekturu koja se u projekt uključuje kroz već spomenutu ovisnost na slici 3.1. Zahvaljujući tome, možemo kreirati RESTful web servis. Glavna značajka spomenutih servisa su REST kontroleri koji se označavaju s @RestController anotacijom. Oni pružaju JSON, XML i prilagođeni odgovor. Druga bitna anotacija je @RequestMapping kojom se označavaju metode koje će se izvršiti kada se definirali link tj. pristupna točka posjeti [18].

[3] engl. *JSON Web Token* – standard za token .

3.1.4. MapStruct

Često se u web aplikacijama koriste DTO[4]-ovi koji predstavljaju objekt prijenosa podataka. Može poslužiti kako bi se sakrio model iz baze podataka. Također, može pružiti sve ili neke podatke na nekom udaljenom korisničkom sučelju ovisno o korisničkoj ulozi. To dovodi do potrebe za mapiranjem DTO-a u JPA entitet ili obrnuto. Kako bi to omogućili potrebno je uključiti sljedeću ovisnost (Slika 3.9.), ali i dodatak na slici 3.10.

```
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>${org.mapstruct.version}</version>
</dependency>
```

Slika 3.9. MapStruct ovisnost.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${org.mapstruct.plugin.version}</version>
    <configuration>
        <source>${java.version}</source>
        <target>${java.version}</target>
        <annotationProcessorPaths>
            <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${org.projectlombok.version}</version>
            </path>
            <!-- This is needed when using Lombok 1.18.16 and above -->
            <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok-mapstruct-binding</artifactId>
                <version>${lombok.mapstruct.binding.version}</version>
            </path>
            <!-- Mapstruct should follow the lombok path(s) -->
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>${org.mapstruct.version}</version>
            </path>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

Slika 3.10. Dodatak za MapStruct

Nakon uključivanja potrebnih stvari, možemo koristiti MapStruct ovisnost koja predstavlja alat za generiranje koda tj. mapera koji će pretvarati DTO u JPA entitet i obrnuto.

[4] engl. *Data transfer objekt* – objekti za prijenos podataka.

3.1.5. Flyway

Flyway je alat za kontroliranje postupnih promjena u bazi podataka. Omogućava jednostavno i pouzdano prebacivanje na novu verziju shema tablica [19]. Uključujemo ga koristeći ovisnost na slici 3.11. Vrlo je koristan kad se tijekom razvoja aplikacija pojavi potreba za dodatnim parametrima u već definiranim shemama tablica u bazi podataka. Osim toga, pogoduje razvojnim timovima da prilikom pokretanja aplikacije svi imaju identične tablice u bazi podataka.

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <version>6.4.4</version>
</dependency>
```

Slika 3.11. Flyway ovisnost

Nakon uključivanja, sql datoteke se kreiraju unutar mape src/resources/db/migration prateći konvenciju imenovanja. Imenuje se koristeći prefiks „V“ zatim dvije donje crte „__“, opis sheme i završava sa sufiksom „.sql“ [20].

3.2. Angular

Angular je razvojno okruženje bazirano na komponentama namijenjeno za razvoj učinkovitih SPA[5]. Komponente su blokovi od kojih se sama aplikacija sastoji [21]. Koristi Typescript programski jezik i dobro integrirane biblioteke za značajke kao što su usmjeravanje, upravljanje formama, klijent-server komunikacija itd. Jedna od značajki je Angular CLI[6] koji se koristi za inicijaliziranje, razvoj i održavanje same aplikacije. Pruža mogućnost kreiranja komponenti, servisa, presretača itd.

3.2.1. Komponente

Komponentu kreiramo koristeći CLI i naredbu na slici 3.12. Komponenta je ništa drugo već Typescript klasa sa @Component() dekoratorom (Slika 3.13), html predloškom i stilom. Dekoratorom klasu činimo dostupnom za uključivanje kao ovisnost npr. u modulima [22]. Unutar dekoratora definirana su pojedina svojstva same komponente kao što su selektor, HTML predložak i predložak za stil.

```
C:\Faks\Diplomski\kkoscak_diplomski\diplomski.web>ng g c ime-komponente
```

Slika 3.12. Naredba za kreiranje komponente

```
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent implements OnInit, OnDestroy {

  isAuthenticated: boolean;
  userName: string;
  subscription: Subscription;

  constructor(
    private userService: UserService,
    private tokenService: TokenStorageService,
    private router: Router
  ) { }
```

Slika 3.13 Prikaz komponente i korištenog @Component() dekoratora.

[5] engl. *Single-Page Applications* – arhitektura sučelja gdje postoji jedna stranica dok se sadržaj mijenja dinamički.
[6] engl. *Command Line Interface* – sučelje naredbenog retka.

Nakon uspješnog kreiranja dobijemo tri datoteke. Prva je html datoteka u kojoj kreiramo elemente, druga je datoteka u kojoj definiramo stilove i treća je datoteka u kojoj obavljamo logiku pojedine komponente.

3.2.2. Moduli

Angular aplikacije su modularne što se postiže koristeći NgModules. To su kontejneri koji mogu sadržavati komponente, servise, pružatelje usluga itd. Uključuju i izvoze određene funkcionalnosti za druge module. Svaka aplikacija ima barem jedan modul, a to je *app* modul koji predstavlja korijenski ili glavni modul [23]. On se prilikom pokretanja aplikacije prevodi te se učitava sve što je definirano u njemu. U ovom radu, postoji nekoliko modula koji predstavljaju bitne dijelove aplikacije. Možemo ih svrstati u dvije kategorije. Jedna predstavlja module komponenti, a druga module funkcionalnosti. Jedan od modula funkcionalnosti je app-routing modul koji je bitan za usmjeravanje. Ostale ćemo objasniti u poglavlju „Realizacija rada".

3.2.3. Usmjeravanje

Usmjeravanje (engl. *Routing*) je postupak usmjeravanja korisnika na određenu stranicu koristeći URL. Navedeno postižemo kreiranjem novog modula koji je potrebno uključiti u glavni aplikacijski modul.

```
@NgModule({
  imports: [RouterModule.forRoot(routes, config: {
    preloadingStrategy: PreloadAllModules,
  })],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Slika 3.14. Izgled app-routing modula

Na slici (3.14.) iznad možemo vidjeti kako izgleda spomenuti modul. On uključuje i izvozi RouterModul za definirane rute u varijabli *routes*. Rute smo definirali kao objekte, gdje smo naveli putanju i modul koji se treba učitati. Detaljnije ćemo objasniti u poglavlju „Realizacija rada".

3.2.4. Servisi

Komponente ne bi trebale dohvaćati i spremati podatke, već se fokusirati na prikaz podataka. Zbog toga koristimo servise. Oni predstavljaju odličan način dijeljenja informacija između komponenti. Servis kreiramo koristeći naredbu na slici 3.15.

```
C:\Faks\Diplomski\kkoscak_diplomski\diplomski.web>ng g s ime-servisa
```

Slika 3.15. Naredba za kreiranje servisa

Servis je također Typescript klasa, ali s dekoratorom @Injectable(). Slikom 3.16. prikazan je izgled korisničkog servisa. Možemo primijetiti kako dekorator ima definirano svojstvo providedIn. Ono je postavljeno na korijen (engl. *Root*). To znači da će kreirana klasa biti dostupna za uključivanje širom aplikacije.

```
import ...

@Injectable({
  providedIn: 'root'
})
export class UserService {...}
```

Slika 3.16. Izgled korisničkog servisa

3.2.5. Presretači

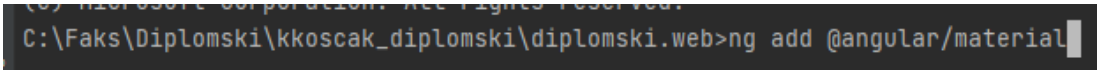
Kao što ime govori, presretači (engl. *Interceptors*) presreću i rukuju HTTP zahtjevima ili HTTP odgovorima. Predstavljaju Typescript klase s dekoratorom @Injectable. Implementiraju sučelje presretača HTTP zahtjeva koje omogućava korištenje metode intercept(). Većina ih preoblikuje izlazni zahtjev prije slanja sljedećem presretaču u lancu [24].

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor{
  constructor(
    private tokenStorage: TokenStorageService
  ) { }

  // tslint:disable-next-line:typedef
  intercept(req: HttpRequest<any>, next: HttpHandler) {...}
}
```

Slika 3.17. Primjer jednog presretača

3.2.6. Angular material

Angular material je biblioteka komponenti koje se koriste za izgradnju korisničkog sučelja. Istu je potrebno dodati u projekt koristeći CLI i naredbu prikazanu slikom 3.18.



Slika 3.18. Naredba za dodavanje Angular material-a

Biblioteka pruža mnoštvo kvalitetnih komponenti koji se mogu koristiti u aplikacijama. Jako dobro su testirane kako bi osigurale performanse i pouzdanost. Većina komponenti omogućava stiliziranje koristeći direktive [25]. Kako bi koristili te komponente, potrebno ih je u jednom od modula uključiti. Na slici 3.19. vidimo primjer uključivanja modula za dugme komponentu u kojoj se nalazi nekoliko vrsta dugmića.

```
import {MatButtonModule} from '@angular/material/button';



const matModules = [...];

@NgModule({
  declarations: [],
  imports: [...matModules],
  exports: [...matModules]
})

export class MaterialModule {}
```

Slika 3.19. Uključivanje komponente za dugmiće

3.2.7. Direktive

U Angular aplikacijama ponašanje se elementima mijenja koristeći direktive, koje predstavljaju klase. Direktive dijelimo na komponente, atributne i strukturalne direktive.

Komponente su direktive sa predloškom. Predstavljaju najčešće korištenu direktivu. Atributne direktive omogućavaju mijenjanje izgleda ili ponašanja elemenata komponenti ili neke druge direktive. Najčešće korištene ugrađene atributne direktive su NgClass, NgStyle i NgModel. Strukturalne direktive utječu na DOM [7]tako što dodaju i brišu DOM elemente. Najčešće korištene ugrađene strukturalne direktive su NgIf, NgFor i NgSwitch [26].

[7] engl. *Document object model* – sučelje za web dokumente.

3.2.8. Animacije

Jedna od značajki Angular-a su i animacije. HTML elementi mijenjaju stil tijekom vremena, stoga primjena animacija može aplikaciju učiniti zanimljivijom i jednostavnijom za korištenje. Poboljšavaju korisničko iskustvo i samu aplikaciju na nekoliko načina. Bez animacija, promjene u prikazu neke veće liste podataka mogu biti trenutne te ju korisnik neće uočiti. Upotrebom animacija korisnik može primijetiti promjene što doprinosi korisničkom iskustvu. Kako bismo ih koristili, potrebno je u aplikacijski modul uključiti BrowserAnimationsModule. Animacije se definiraju unutar @Component dekoratora same komponente pod svojstvom animations [27]. Detaljnije ćemo objasniti u poglavlju „Realizacija rada“.

3.2.9. SCSS

CSS je jezik koji se koristi za stiliziranje web stranica, konkretnije HTML elemenata. Kod većih aplikacija teško je održavati takve datoteke. SCSS je nadogradnja na CSS koja uključuje nekoliko značajki koje ne postoje kod CSS-a. Jedna od njih je ugnježđivanje, nasljeđivanje, definiranje varijabli itd [28]. Na sljedećim slikama prikazane su neke od razlika između SCSS-a i CSS-a.

```scss
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

```css
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

Slika 3.20. Razlika u primjeni varijabli [28]

```scss
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

```css
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

Slika 3.21. Razlika u ugnježđivanju [28]

3.3. Raspberry Pi

3.3.1. Općenito

Raspberry Pi je jeftino, malo računalo koje ima mogućnost spajanja na TV ili monitor i korištenja tipkovnice i miša. Zbog svoje jednostavnosti omogućava ljudima istraživanje mogućnosti i učenje programskih jezika kao što je npr. Python. Ima funkcionalnosti kao što ima i obično desktop računalo. Može komunicirati s vanjskim svijetom i koristi za u širokom rasponu projekata kao što su pametne kuće, detektori vremena, strojno učenje itd [29]. Budući da se radi o mikroračunalu, za uspješno funkcioniranje potreban je operacijski sustav. Na službenoj stranici dostupno je nekoliko operacijskih sustava. Najjednostavniji za korištenje bi bio Raspberry Pi Desktop koji pruža grafičko korisničko sučelje. Za instalaciju se koristi Raspberry Pi Imager. Spomenuti OS dolazi sa određenim značajkama koje su već instalirani te nema potrebe za dodatnim instaliranjem. Jedna od tih je programski jezik Python. Tvorci Raspberry Pi-a su ga odabrali kao glavni jezik zbog njegove moći i jednostavnosti korištenja [30]. Raspberry Pi pločica posjeduje i red GPIO pinova cijelom svojom dužinom. Broji četrdeset pinova koji se dijele u nekoliko kategorija. Jedna je GPIO pin, dva pina koji pružaju izlaz od 5 volti, dva pina od 3.3 volti te osam pinova koji predstavljaju uzemljenje. GPIO pinovi su također 3.3 volti, ali oni se koriste tokom programiranja te se pale ovisno o postavljenom uvjetu.

3.3.2. Povezivanje sa udaljenim računalom

Ponekad nije zgodno svaki puta koristiti poseban monitor, miš i tipkovnicu kako bi mogli upotrebljavati Raspberry Pi. Desktop računala zbog toga često koriste softver za povezivanje s udaljenim računalom (engl. *Remote Desktop Control*). Time se omogućava kontrola nad udaljenim računalom. Softver koristi RDP[8], Microsoft-ov protokol koji omogućava pristup našem uređaju. S druge strane, udaljeno računalo zahtijeva dodatan softver za povezivanje, koji se zove xrdp. Xrdp je RDP server otvorenog koda sa grafičkim sučeljem za prijavu na udaljeno računalo. Prihvaća povezivanje od različitih klijenata među kojima je i već spomenuti softver za povezivanje s udaljenim računalom [31]. Korištenjem spomenutih tehnologija omogućava se pristup Raspberry Pi računali sa desktop računala.

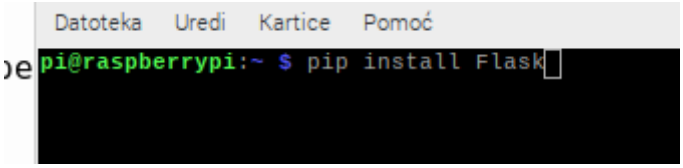
[8] engl. *Remote desktop protocol* – protokol za povezivanje s udaljenim računalom.

3.3.3. Biblioteka gpiozero

Na već spomenute pinove mogu se spojiti razni elektronički uređaji kao što su led diode, dugmići, senzori pokreta, svjetlosni senzori itd. Radi jednostavnijeg upravljanja spomenutim uređajima koristi se Gpiozero biblioteka. Služi za jednostavnije kodiranje GPIO pinova tokom čitanja i pisanja. Sadrži definirane klase za svaku od gore spomenutih elektroničkih uređaja te sadrži podatke o istima.

3.3.4. Flask

Flask je web razvojno okruženje u Pythonu. Još se naziva i mikro razvojno okruženje jer ne zahtijeva dodatne alate ili biblioteke [32]. Kako bi ga mogli koristit unutar Pythona potrebno ga je instalirati koristeći naredbu prikazanu na slici 3.22.



Slika 3.22. Prikaz naredbe za instalaciju Flask-a

Nakon instaliranja, u datoteku se uključuje kao i uobičajene biblioteke. Sada je moguće kreirati web aplikaciju te definirati metode koje će se aktivirati kada se pojedina pristupna točka posjeti. Detaljnije je opisano u poglavlju „Realizacija rada“.

3.3.5. Ngrok

Kreirane aplikacije iz prethodne kategorije (3.3.4.) dostupne su samo unutar okvira kućne mreže. Kako bi iste napravili dostupnima na internetu, potrebno je koristiti ngrok alat. To je alat kojim lokalnu web stranicu možemo izložiti. Kako bi to omogućili potrebno je preuzeti sa službene stranice, a zatim instalirati. Korištenje ovog alata zahtjeva registraciju na njihovoj stranici čime se korisniku pruža sučelje putem kojeg može pratiti pristup web aplikaciji. Nakon što je web aplikacija pokrenuta lokalno, naredbom na slici 3.23. ju činimo dostupnom na internetu

Slika 3.23. Naredba kojom aplikaciju činimo dostupnom na internetu [33].

3.4. PostgreSQL

PostgreSQL je objektno relacijska baza podataka otvorenog koda. Zbog svoje postojanosti ona je pouzdana, pretrpana mogućnostima i visokih performansi. Podržava veliki dio SQL standarda i pruža mnoge značajke kao što su kompleksni upiti, strani ključevi, okidači itd [34].

3.5. Docker

Docker je platforma za razvoj, slanje i pokretanje aplikacija. Omogućava odvajanje aplikacija u izolirana okruženja koja se zovu kontejneri. Zbog izoliranja omogućava se pokretanje više kontejnera istovremeno. Takvi kontejneri se mogu pokretati na računalu programera, stvarnom ili virtualnom stroju u podatkovnom centru, u oblaku ili miješanim okruženjima [35].

3.6. IntelliJ

IntelliJ je integrirano razvojno okruženje pisano u Javi za razvoj računalnih softvera. Pruža mogućnost rada s Java i drugim jezicima kao što su Kotlin, Scala i Groovy. Velika prednost mu je što omogućava razvoj full-stack web aplikacija. Tome pridonose integrirani alati, podrška za JavaScript i povezane tehnologije. Izuzetno je kompatibilan s razvojnim okruženjima kao što su Spring i Spring Boot za koje pruža kvalitetnu podršku. Dodaci razvijeni od JetBrains tima, pridonose radu s jezicima kao što su Python, SQL, PHP, itd. Jedna od značajki je i sučelje za povezivanje s različitim bazama podataka te upravljanje istima [36].

4. REALIZACIJA RADA

Kod ovog rada, cjelokupni posao podijeljen je na četiri dijela. Prvi se odnosi na PostgreSQL bazu podataka koju ćemo koristiti i potrebne tablice. Drugi dio se odnosi na serverski dio koji smo kreirali koristeći Spring Boot. Njegova svrha je komunikacija s bazom podataka, obrada podataka te dostava istih. Klijentska strana predstavlja treći dio i ujedno korisničko sučelje koje je kreirano kao SPA pomoću Angular razvojnog okruženja. Zadnji dio odnosi se na mikroračunalo koje predstavlja web aplikaciju putem koje upravljamo uređajima spojenim na isto.

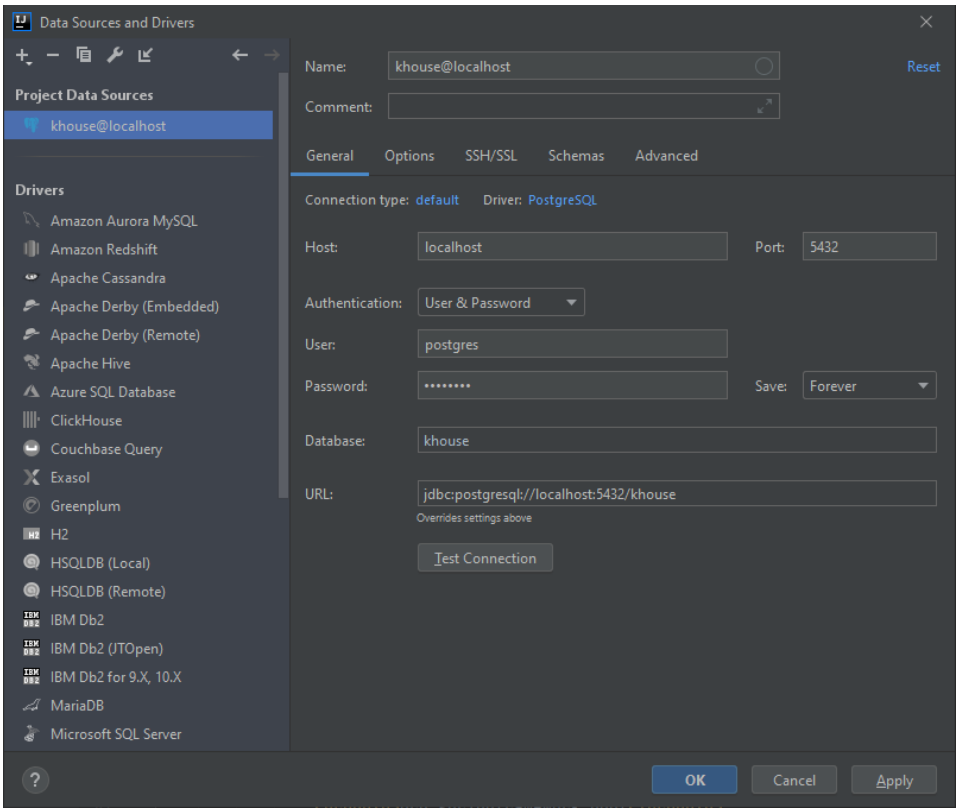
4.1. Baza podataka

Baza podataka postavlja se koristeći spomenuti alat Docker. Njime kreiramo i pokrećemo kontejner PostgreSQL servera. U terminalu unosimo naredbu prikazanu na slici 4.1. kako bi pokrenuli spomenuti server.

```
docker run -e POSTGRES_PASSWORD=postgres -e POSTGRES_USER=postgres -e POSTGRES_DB=testdb -p 5432:5432 postgres:12.4
```

Slika 4.1. – Naredba za pokretanje PostgreSQL server kontejnera

Nakon toga, koristeći sučelje baze podataka u IntelliJ-u se povezujemo s bazom podataka i podacima prikazanim na slici 4.2.



Slika 4.2. Podaci za spajanje na kreiranu bazu podataka

Kreiramo tablice, za entitete koje ćemo koristiti kao što su: khouseuser, house, house_user, room, device i device_statistic.

khouseuser predstavlja korisnike koji će se registrirati u aplikaciju i upravljati kućama. Sadrži podatke o korisniku kao što su: ime, prezime, email i lozinka.

house je tablica kuće, ujedno i glavni entitet ove aplikacije. Sadrži podatke o kući kao što su: grad, ulica, pristupna poveznica, broj uređaja, oznaka vlasnika.

house_user predstavlja korisnika koji se dodaje u kuću, sadrži email i strani ključ na kuću.

room – je tablica sobe unutar pojedine kuće, a sadrži ime sobe i strani ključ na kuću.

device – predstavlja uređaj u pojedinoj sobi, a sadrži sljedeće atribute: vrsta, pin, status, ostale informacije i strani ključ na sobu.

device_statistic je tablica statistike o pojedinom uređaju, sadrži podatke o trenutku paljenja i gašenja uređaja te strani ključ na uređaj.

Tablice kreiramo na identičan način kao i sljedeću prikazanu slikom 4.3. Jedan od uvjeta u tablici odnosi se na „id“, a to je da on predstavlja glavni ključ tablice. Drugi uvjet odnosi se na „device_id“ koji predstavlja strani ključ na tablicu uređaja

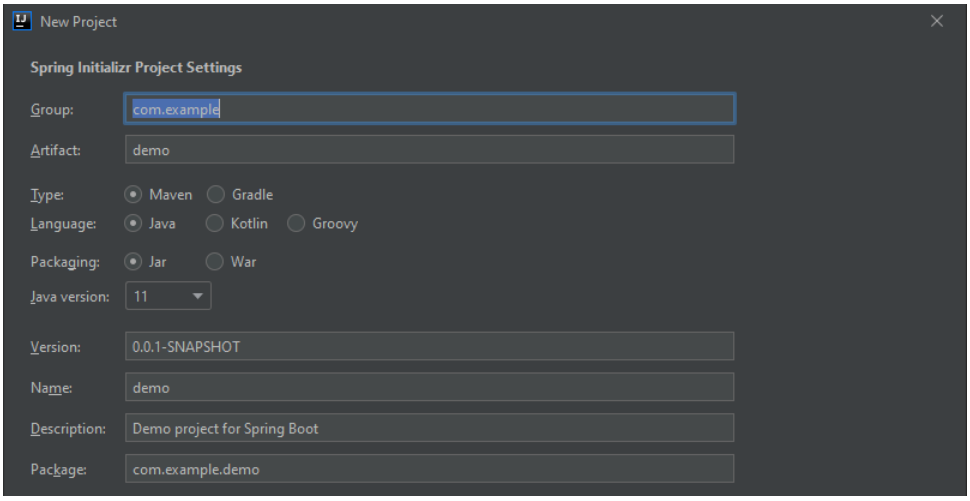
```
CREATE TABLE device_statistic (
                              id bigserial NOT NULL,
                              turned_on timestamp NULL,
                              turned_off timestamp NULL,
                              device_id int8 NOT NULL,

                              CONSTRAINT device_statistic_pkey PRIMARY KEY (id),
                              CONSTRAINT device_statistic_fkey FOREIGN KEY (device_id) REFERENCES device(id)
);
```

Slika 4.3. Primjer kreiranja tablice u PostgreSQL

Zbog korištenja Fylway ovisnost, imamo mogućnost verzioniranja postojećih tablica. Isto tako, ne moramo ručno kreirati tablice svaki puta kada se projekt preuzme i pokrene. U bazi podataka kreira se dodatna tablica „flyway_schema_history“ unutar koje se nalaze sve izvršene sheme iz mape migracije (unutar mape resursi, prema strukturi na slici 4.6.). Prilikom pokretanja spomenuta tablica se gleda kako bi se provjerilo koja je zadnja i postoji li neka koja nije izvršena.

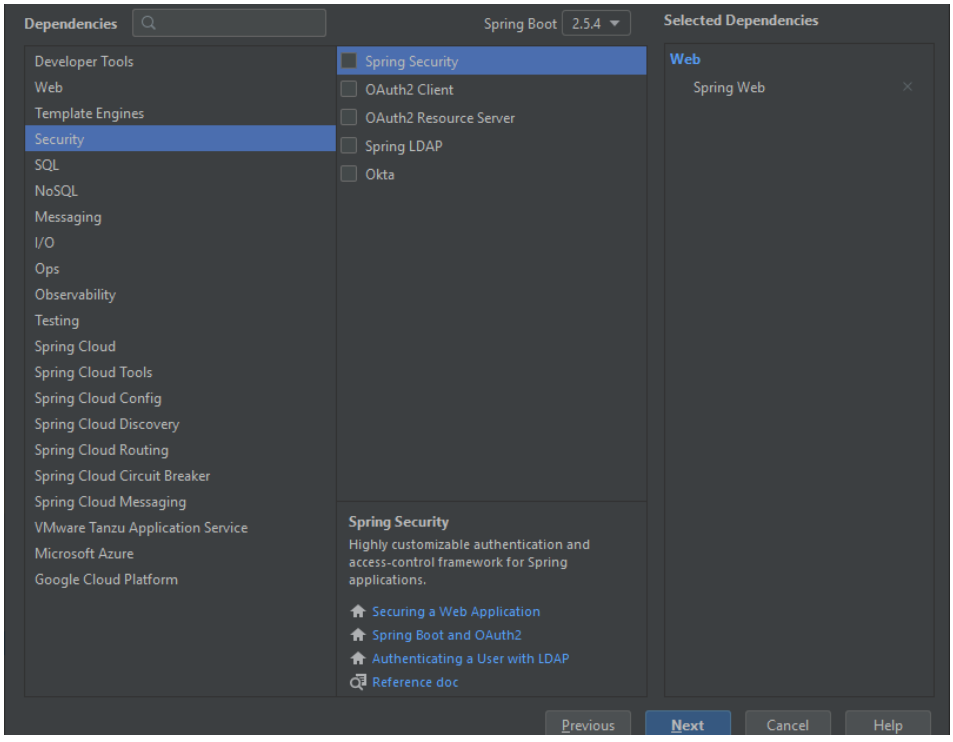
4.2. Serverski dio

Projekt kreiramo koristeći Spring Initializr (Slika 4.4.). On pruža API za generiranje Java projekata i za pregled meta podataka koji se koriste za generiranje Spring Boot projekta. Unutar njega unosimo potrebne podatke kao što su vrsta projekta, jezik, Java verzija itd. Budući da smo kao vrstu projekta odabrali Maven, imamo pom.xml datoteku unutar koje se dodaju ovisnosti za određene funkcionalnosti.



Slika 4.4. Kreiranje projekta koristeći Spring Initializr

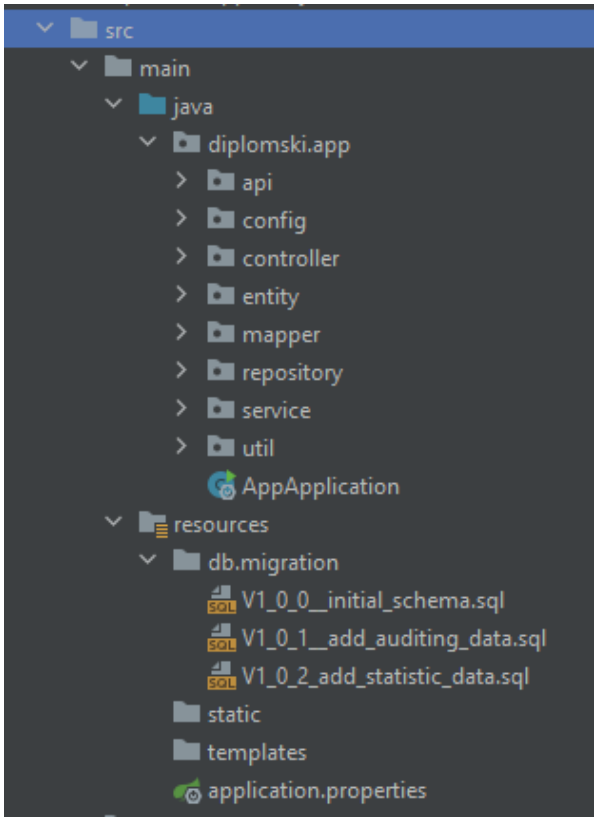
Osim navedenog, možemo uključiti i određene ovisnosti prikazane slikom 4.5.



Slika 4.5. Dodavanje početnih ovisnosti

4.2.1. Struktura projekta

Struktura aplikacije je podijeljena u nekoliko paketa radi lakšeg snalaženja. Istovremeno, izbjegavalo se što veće grananje u paketima. Struktura je prikazana slikom 4.6.



Slika 4.6. Struktura aplikacije

Pristupnu točku definiranoj logici predstavlja paket kontroler (engl. *Controller*). Unutar njega nalazi se nekoliko kontrolera koji implementiraju sučelja definirana u *api* paketu. Također, unutar tog paketa nalaze se *dto* paket gdje su definirani DTO objekti. Unutar *config* paketa nalazi se konfiguracija zaštite aplikacije, pratitelj prijavljenog korisnika i kodiranja lozinke. Paket entitet (engl. *Entity*) sadržava entitete navedene u poglavlju „Baza podataka“. Unutar *mapper* paketa nalaze se sučelja definiranih mapera koje možemo koristiti zahvaljujući MapStruct ovisnosti koju smo uključili. Sljedeći paket skladišta (engl. *Repository*) sadrži sučelja koja izvode CRUD[9] operacije nad entitetima. Paket servis (engl. *Service*) sadrži sučelja i implementaciju servisa, kojima se izvode operacije nad bazom podataka. Zadnji paket *util* sadrži potrebne klase za korištenje tokena. Unutar mape migracija (engl. *Migration*) nalaze se sheme koje kreiraju tablice prilikom prvog pokretanja aplikacije te se pamti koja shema je zadnja pokrenuta. Jako bitan dokument projekta je *application.properties* dokument. On predstavlja spremnik unutar kojeg se

[9] engl. *Create, read, update and delete* – četiri osnovne metode koje se obavljaju nad podacima.

mogu definirati određene varijable te koristiti širom projekta. Mnoge ovisnosti ga koriste za postavljanje pojedinih varijabli konfiguracije. Detaljnije ćemo opisati u poglavljima „Servis“ i „Repozitorij“.

4.2.2. Kontroleri

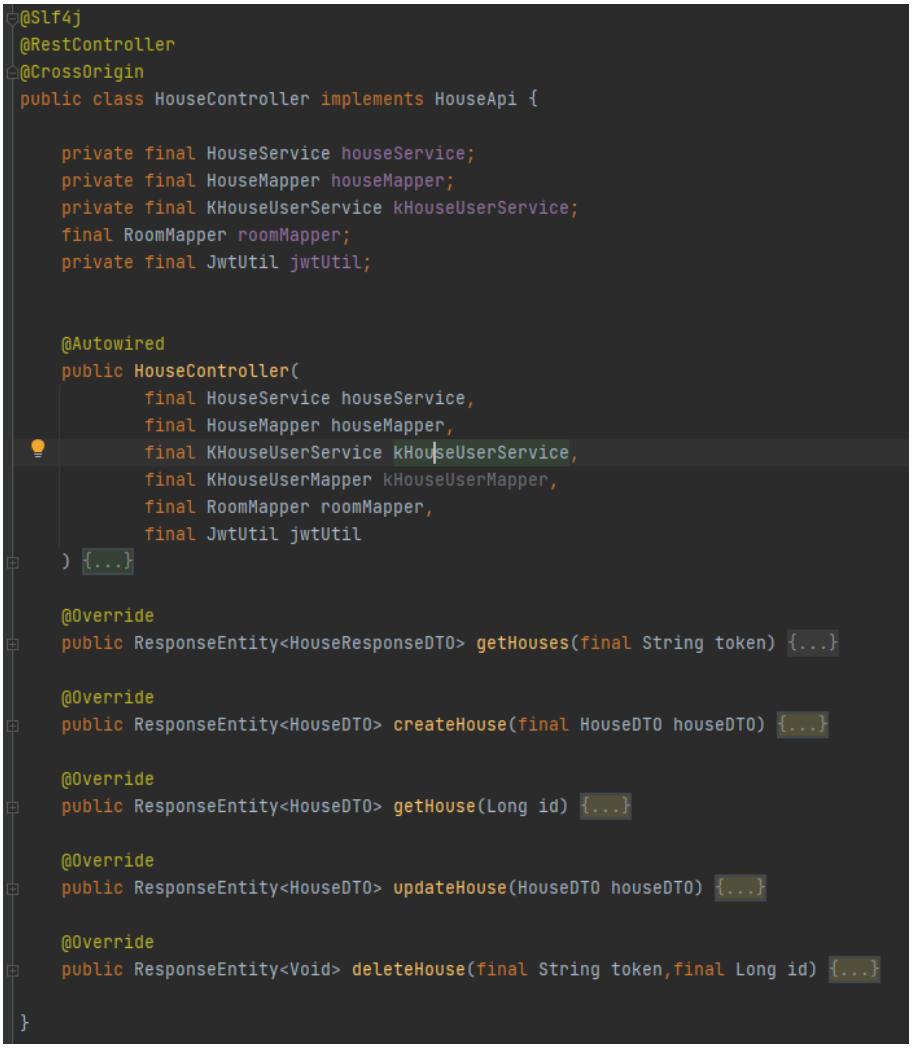
Kontroleri predstavljaju pristupne točke naše aplikacije budući da se u njima odvija primanje i rukovanje korisničkih zahtjeva. Svaki od kontrolera implementira sučelje unutar kojeg definiramo što to sučelje predstavlja. Jedno od takvih sučelja prikazano je na slici 4.7.

```java
@RestController
@RequestMapping(HouseApi.BASE_URL)
public interface HouseApi {

    String BASE_URL = "/api/house";

    @GetMapping(value = "/")
    ResponseEntity<HouseResponseDTO> getHouses(@RequestHeader (name="Authorization") String token);

    @PostMapping(value = "/")
    ResponseEntity<HouseDTO> createHouse(@RequestBody HouseDTO houseDTO);

    @PutMapping(value = "/")
    ResponseEntity<HouseDTO> updateHouse(@RequestBody HouseDTO houseDTO);

    @GetMapping(value = "/{id}")
    ResponseEntity<HouseDTO> getHouse(@PathVariable("id")Long id);

    @DeleteMapping(value = "/{id}")
    ResponseEntity<Void> deleteHouse(@RequestHeader (name="Authorization") String token, @PathVariable("id")Long id);

}
```

Slika 4.7. Primjer sučelje kuće

Sučelje ima anotaciju @RestController koje označava da se radi o REST kontroleru. Također, vidimo da posjeduje i @RequestMapping anotaciju unutar kojeg navodimo rutu putem koje ćemo moći pristupiti tome kontroleru, a ona počinje sa /api/house. Nadalje, unutar sučelja definirali smo metode. Svaka od metoda sadrži određenu anotaciju koja predstavlja vrstu HTTP zahtjeva i rutu za koju će se ta metoda izvršiti. Korištenje dinamičkih atributa rute omogućeno je korištenjem vitičastih zagrada unutar kojih navodimo ime atributa. Istome možemo pristupiti tako što prilikom definiranja metode, predamo metodi varijablu određenog tipa i imena ispred kojeg koristimo @PathVariable anotaciju sa imenom atributa definiranog u ruti. Za pristup određenom dijelu zahtjeva kao što je npr. tijelo zahtjeva koristimo @RequestBody anotaciju. Ona se većinom koristi kada nekakve JSON podatke šaljemo sa korisničkog sučelja te se definira također kao parametar koji se predaje metodi. Možemo primijetiti još jednu anotaciju koja se koristi, a to je @RequestHeader anotacija koja pristupa određenom parametru zaglavlja zahtjeva. Koristimo ju kod brisanja i dohvaćanja kuća, za dohvaćanje tokena.

Nakon definiranja sučelja kreiramo kontroler klasu koja nasljeđuje spomenuto sučelje. Iznad klase također koristimo @RestController anotaciju kao što je prikazano na slici 4.8.



Slika 4.8. Primjer implementacije kontrolera kuće

Unutar njega koristimo @Autowired anotaciju iznad konstruktora kontrolera. Unutar istog predajemo određene servise, mapere i klasu koja rukuje tokenom. Zbog spomenute anotacije pravilno će se povezati predani parametri sa kreiranima u projektu. Možemo primijetiti kako su sve metode definirane u sučelju prepisane i sadrže određenu logiku. Ista se obavlja koristeći definirane mapere i servise. Na slici 4.9. možemo vidjeti logiku dohvaćanja kuće.

```java
@Override
public ResponseEntity<HouseDTO> getHouse(Long id) {
    log.info("Getting House with id: {}", id);
    final House houseFromDB = houseService.findHouseById(id);
    if(houseFromDB == null){
        return new ResponseEntity( body: "House with that ID does not exist!", HttpStatus.NOT_FOUND);
    }
    return ResponseEntity.ok(houseMapper.toDTO(houseFromDB));
}
```

Slika 4.9. Primjer logike za dohvaćanje kuće.

Koristimo servis kuće koji istu dohvaća prema identifikacijskoj oznaci. Budući da on radi s entitetima baze, potrebno je prilikom vraćanja podataka korisniku istu prebaciti u DTO objekt kuće. Detaljnije o maperima ćemo objasniti u poglavlju „Maperi“.

4.2.3. Servis

Servisi su Java klase koje obavljaju određenu logiku nad podacima prije nego što se proslijede kontroleru. Označava se s anotacijom @Service koja omogućava vidljivost širom aplikacije. Budući da servisi većinom obavljaju identične operacije kao što su kreiraj, dohvati pojedini, dohvati sve, uredi ili obriši podatke, koristili smo nasljeđivanje i implementaciju kako izbjegli dupliciranje koda. Jedan od servisa prikazan je na slici 4.10.

```java
@Slf4j
@Service
public class HouseServiceImpl extends AbstractCrudServiceImpl<House, Long, HouseRepository> implements HouseService {

    private final HouseUserService houseUserService;

    @Autowired
    public HouseServiceImpl(final HouseUserService houseUserService) { this.houseUserService = houseUserService; }

    @Override
    public List<List<House>> getHouses(KHouseUser kHouseUser) {...}

    @Override
    public House findHouseById(final Long id) throws ResourceNotFoundException {
        return genericRepository.findById(id)
                .orElseThrow(()->{
                    log.info("There is no house with id -> {}", id);
                    return new ResourceNotFoundException(House.class.getSimpleName());
                });
    };

    @Override
    public House findHouseByAccessApi(String accessApi) { return genericRepository.findByAccessApi(accessApi); }
}
```

Slika 4.10. Implementacija servisa kuće

Spomenuti servis nasljeđuje AbstractCrudServiceImpl apstraktnu klasu unutar koje se nalazi logika za već spomenute uobičajene metode. Osim spomenutih prepisujemo metode koje su definirane u sučelju HouseService i kreiramo logiku, a predstavljaju izvanredne metode koje nisu pokrivene u naslijeđenoj klasi. Jedna od njih je i findHouseById prikazana na slici 4.10. Budući

da naša klasa servisa nasljeđuje drugu klasu, ona ima sva svojstva i metode kao i ta roditeljska klasa. Prema tome, možemo pristupiti generičkom repozitoriju koji dohvaća određeni entitet za predanu identifikacijsku oznaku. Detaljnije ćemo objasniti u sljedećem poglavlju.

4.2.4. Repozitorij

Spring Data JPA smanjuje veličinu koda potrebnu za rad s bazom podataka. Omogućava nam definiranje repozitorija i metoda za izvođenje jednostavnih CRUD operacija nad podacima. Da bi mogli pristupiti bazi podataka potrebno je definirati određene varijable konfiguracije unutar *application.properties* datoteke. One su prikazane slikom. 4.11.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/khouse
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Slika 4.11 Varijable konfiguracije za povezivanje s bazom

JpaRepository je sučelje koje posjeduje metode za izvođenje jednostavnih operacija nad podacima. Stoga, naši repozitoriji nasljeđuju JpaRepository koji zahtjeva vrstu entiteta i identifikacijske oznake. Jedno od takvih sučelja je prikazano na slici 4.12.

```
public interface HouseRepository extends JpaRepository<House, Long> {

    House findByAccessApi(String accessApi);
}
```

Slika 4.12. Repozitorij kuće

Unutar njega vidimo kako smo definirali vlastitu metodu za dohvaćanje kuće prema predanom pristupnom linku. Ono će kreirati upit prema imenu varijable, dakle za pretraživanje kuća prema odgovarajućem pristupnom linku gdje je povratni tip, tipa kuće. Repozitorij kuća posjeduje metodu za dohvaćanje prema predanoj identifikacijskoj oznaci koja vraća objekt tipa House, a koja se koristi u servisu na slici 4.10. Svi ostali repozitoriji u projektu su kreirani na identičan način i posjeduju iste metode. Razlika je jedino u tipu objekta s kojim rukuju.

4.2.5. Maperi

Nakon dohvaćanja kuće na slici 4.9. vidimo kako se koristi maper za prebacivanje entiteta u DTO. DTO su objekti koji se koriste za prijenos podataka u određenoj strukturi. Radi jednostavnosti potrebno je da struktura kuće na korisničkom sučelju odgovara strukturi spomenutog objekta na serveru. Time skraćujemo posao prilagođavanja podataka na sučelju i obrnuto. Budući da se entiteti odnose na strukturu podataka koji su spremljeni u bazi podataka, nije baš zgodno istu poslati na sučelje. Također, koristeći DTO možemo kreirati nekoliko različitih prikaza podataka pojedinog entiteta. Zahvaljujući MapStruct ovisnosti izbjegava se potreba za pisanjem logike koja će DTO prebaciti u entitet i obrnuto. Maper predstavlja sučelje unutar kojega je potrebno definirati metode koje će vršiti pretvorbu. Sučelje sadrži @Mapper anotaciju unutar koje je potrebno navesti ostale mapere koji se koriste. Nakon pokretanja aplikacije, generirati će se implementacije spomenutih sučelja sa svim metodama i njihovom logikom. Na slici 4.13. prikazan je jedan od mapera.

```
@Mapper(componentModel = "spring",
        uses = {RoomMapper.class, HouseUserMapper.class},
        injectionStrategy = InjectionStrategy.CONSTRUCTOR)
public interface HouseMapper {

    House toEntity(HouseDTO houseDTO);

    HouseDTO toDTO(House house);

    List<House> toListEntity(List<? extends HouseDTO> houseDTOs);

    List<HouseDTO> toListDTO(List<? extends House> house);

    void update(House newHouse, @MappingTarget House existingHouse);

}
```

Slika 4.13. Primjer mapera kuće

Vidimo kako postoji nekoliko metoda za pretvorbu i metoda za izmjenu postojeće kuće. Unutar nje koristimo @MappingTarget anotaciju kako bi znali koju kuću konkretno izmjenjujemo. Prema slici 4.9. ‚maper koristi metodu toDTO jer se podaci kuće šalju nazad na klijentsku stranu nakon upita.

4.2.6. Zaštita aplikacije

Kako bi onemogućili nedozvoljene pristupe pojedinim kontrolerima, potrebno je postaviti konfiguraciju zaštite. Ista se nalazi u paketu *config* kao što je navedeno u poglavlju „Struktura projekta“. Ondje se nalazi klasa koja nasljeđuje WebSecurityConfigurerAdapter. Zbog nje

moramo prepisati metodu configure i napisati logiku koja se odvija nakon pokretanja aplikacije. Naša konfiguracijska klasa je prikazana slikom 4.14.

```
@Configuration
@EnableWebSecurity
public class KHouseSecurityConfig extends WebSecurityConfigurerAdapter {

    private static final String[] AUTH_WHITELIST = {
            // Other public API endpoints
            "/api/**"
    };

    @Autowired
    private AuthEntryPointJwt unauthorizedHandler;

    @Bean
    public AuthTokenFilter authenticationJwtTokenFilter() {return new AuthTokenFilter();}

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
                .cors().and().csrf().disable() HttpSecurity
                .antMatcher("/**").authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
                .antMatchers(AUTH_WHITELIST).permitAll()
                .anyRequest().authenticated()
                .and() HttpSecurity
                .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        httpSecurity.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    }
}
```

Slika 4.14. Zaštitna konfiguracija aplikacije

Sadrži @Configuration anotaciju koja naglašava da se radi o konfiguracijskoj klasi. Osim spomenute konfiguracije koristi se i @EnableWebSecurity konfiguracija koja omogućava primjenu sigurnosne zaštite širom aplikacije. Pristup bilo kojem od kontrolera zahtjeva autorizaciju. Zaštita radi u kombinaciji sa JWT-om kako bi provjerili je li korisnik valjan. Navedeno je omogućeno upotrebom filtera kojeg smo definirali kao klasu AuthTokenFilter. Unutar njega radimo provjeru, zahtjeva tako što izvlačimo token iz zaglavlja zahtjeva. Nakon toga koristimo klasu JwtUtil klasu koja je odgovorna za kreiranje, dohvaćanje emaila i provjeru ispravnosti tokena. Nakon provjere tokena, odlučuje se što dalje. Jedna od opcija je da korisnik može pristupiti kontroleru, a druga je da korisnik nije valjan. U tom slučaju koristi se rukovatelj neovlaštenog pristupa koji baca grešku o nedozvoljenom pristupu. Isti je definiran klasom AuthEntryPointJwt. Spomenute klase JwtUtil, AuthTokenFilter i AuthEntryPointJwt smo mi kreirali i nalaze se unutar paketa *util*

4.3. Klijentski dio

Klijentski dio projekta razvijen je koristeći Angular razvojno okruženje, Typescript programski jezik, HTML i SCSS. Koristeći terminal i komandu prikazanu slikom ispod (Slika 3.1.) kreiramo SPA aplikaciju. Nakon pokretanja, korisnika se pita želi li uključiti neke od značajki aplikacije kao što su stroža provjera tipa podataka, usmjeravanje i format stilova. Koristimo

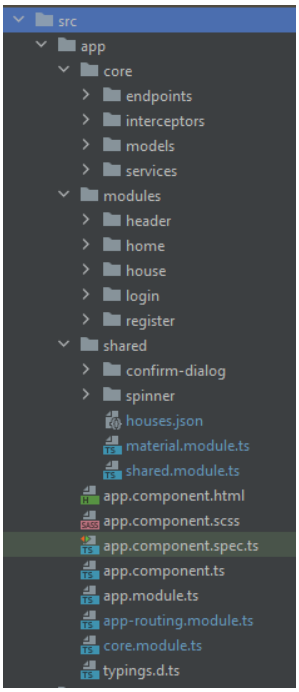
```
C:\Faks\Diplomski\dokumentacija>ng new ime-aplikacije
```

Slika 4.15. Naredba za kreiranje projekta

Nakon kreiranja, potrebno je u CLI-u radni direktorij prebaciti u sami projekt kako bi mogli pokretati server, dodavati vanjske biblioteke, kreirati komponente itd.

4.3.1. Struktura projekta

Struktura aplikacije podijeljena je u tri bitna dijela. Radi lakšeg snalaženja i bolje preglednosti kreirali smo glavnu (engl. *Core*), moduli (engl. *Modules*) i dijeljenu (engl. *Shared*) mapu. Također smo izbjegavali „dubinu“ unutar svake. Struktura je prikazana slikom 4.16.



Slika 4.16. Struktura aplikacije

Unutar *core* mape nalaze se pristupne točke za pozadinsku logiku (engl. *Backend*), prestretači, modeli i servisi. Mapa moduli (engl. *Modules*) sadržava glavne komponente projekta dok dijeljena (engl. *Shared*) mapa sadrži dijelove aplikacije koje se koriste na različitim mjestima širom aplikacije.

4.3.2. Moduli

Glavni modul aplikacije je aplikacijski modul. Unutar njega smo uključili i druge kao što je modul za rute, dijeljeni modul, temeljni modul ,modul navigacijske trake, modul za animacije itd. Izgled aplikacijskog modula prikazan je slikom 4.17.

```
const appRoutes: Routes = [...]

@NgModule({
  declarations: [...],
  imports: [
    // Angular modules
    BrowserModule,
    BrowserAnimationsModule,
    HttpClientModule,
    FontAwesomeModule,
    RouterModule.forRoot(appRoutes, config: {relativeLinkResolution: 'legacy'}),

    // App modules
    AppRoutingModule,
    SharedModule,
    CoreModule,
    HeaderModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Slika 4.17. Aplikacijski modul

Svi moduli i komponente koje su definirani u aplikacijskom modulu će se učitati prilikom pokretanja aplikacije. Radi optimizacije smanjili smo njihov broj, dijeljenjem na više modula koji su odgovorni za glavne komponente aplikacije. Nadalje opisat ćemo neke od modula koji su uključeni u aplikacijskom modulu.

AppRoutingModule je odgovoran za usmjeravanje. Unutar njega smo definirali rute kao objekte koje pozivaju odgovarajuće module.

SharedModule je odgovoran za module i komponente koje se koriste širom aplikacije. Time smo spriječili dupliciranje koda.

CoreModule modul je odgovoran za presretanje zahtjeva, rukovanje greškama i tokenima.

4.3.3. Usmjeravanje

Usmjeravanje koristimo kako bi korisnika preusmjerili na odgovarajuću stranicu. Rute smo kao što je već spomenuto definirali kao objekte (Slika 4.18.). Time smo izbjegli učitavanje svih komponenti iz korijenskog modula te smo to delegirali na pojedine module komponenti. Taj postupak dijeljenja aplikacije u module zove se lijeno učitavanje (engl. *Lazy loading*) strategija učitavanja. Spomenutu strategiju smo koristili kako bi optimizirali samu aplikaciju. Osim navedenog, potrebno je i u samome modulu odrediti strategiju prethodnog učitavanja. Nju smo postavili na PreloadAllModules, što znači da će se u pozadini dijelovi aplikacije učitati i pozvati kada budu potrebni.

```
const routes: Routes = [
  {
    path: '',
    loadChildren: () => import("./modules/home/home.module").then(m => HomeModule),
    pathMatch: 'full'
  },
  {
    path: 'login',
    loadChildren: () => import("./modules/login/login.module").then(m => LoginModule)
  },
  {
    path: 'register',
    loadChildren: () => import("./modules/register/register.module").then(m => RegisterModule)
  },
  {
    path: 'house',
    loadChildren: () => import("./modules/house/house.module").then(m => HouseModule)
  }
];
```

Slika 4.18. Definirane rute unutar modula za usmjeravanje

Ukoliko korisnik posjeti stranicu na kojoj se prikazuju kuće /house, uključit će se HouseModule koji je odgovoran za nekoliko komponenti. Unutar njega su definirane rute za prikaz pojedine kuće, dodavanje kuće, izmjenu postojeće itd. Također uključuje dijeljeni modul i modul koji sadrži različite module Angular komponenti. Primjer jedne od ruta koja se nalazi unutar kuće dan je slikom 4.19.

```
{
  path: '',
  component: HouseComponent,
  canActivate: [AuthGuard],
  pathMatch: 'full',
  resolve: {data: HouseService...}
},
```

Slika 4.19. Prikaz rute za početnu stranicu kuća

Kada korisnik posjeti rutu /house, učitati će se glavna komponenta kuća. Ostale parametre iz rute prikazane slikom 4.19. ćemo objasniti u sljedećim kategorijama.

4.3.4. Komponente

Komponente su predstavljaju blokove od kojih se Angular aplikacija sastoji. U ovoj aplikaciji ih ima nekoliko. Jedna od njih je i HouseComponent komponenta koja se pokreće prilikom dolaska na rutu /house. Unutar HTML dokumenta nalazi se komponenta kartice (engl. *Tabs*) iz Angular Material biblioteke kao i element kojim se prikazuje učitavanje dokumenata. Spomenute dvije komponente se izmjenjuju ovisno o atributu isLoading. Osim spomenute, koristi se i komponenta dugmića iz iste biblioteke. Izgled iste s kodom za jednu karticu prikazan je na slici 4.20.

```
<div fxLayout="row" fxFlex="100" class="house-container">
  <div fxLayout="row" fxFlex="100" fxLayoutAlign="center center" *ngIf="isLoading">
    <app-spinner></app-spinner>
  </div>
  <div fxLayout="row" fxFlex="100" fxLayoutAlign="center start" class="user-houses-container" *ngIf="!isLoading">
    <mat-tab-group mat-align-tabs="center" fxFlex="100">
      <mat-tab label="Own houses" fxLayoutAlign="center center" >
        <ng-container>
          <div fxLayout="row" fxFlex="100" class="add-house-container" >
            <div fxLayout="column" fxLayoutAlign="center center" fxFlex="100">
              <span class="text-centered" *ngIf="ownHouses.length === 0 ">You didn't add a house yet. </span>
              <button mat-raised-button class="add-button" title="Add house" (click)="openAddHouseDialog()">
                <span>Add {{ownHouses.length > 0 ? 'another' : ''}} house</span>
              </button>
            </div>
          </div>
        </ng-container>
        <ng-container>
          <div fxLayout="column" fxLayout.gt-xs="row" fxLayoutAlign="start center"
               class="houses-container"
               [ngClass.gt-xs]="{'house-container-gt-xs': true,'house-container-gt': true}"
               [ngClass.gt-sm]="{'house-container-gt-sm': true,'house-container-gt': true}"
               [ngClass.gt-md]="{'house-container-gt-md': true,'house-container-gt': true}">
            <ng-container  *ngFor="let house of ownHouses; ">
              <app-house-item [house]="house" [isOwner]="true"></app-house-item>
            </ng-container>
          </div>
        </ng-container>
      </mat-tab>
      <mat-tab label="Other houses" fxLayoutAlign=" center"...>
    </mat-tab-group>
  </div>
</div>
```

Slika 4.20. Izgled house.component.html datoteke

HTML elementi imaju neuobičajene atribute koje se koriste. Primjer takvog je fxLayout. To je svojstvo koje nam omogućava Angular Flex-Layout modul. Pruža jednostavniji način korištenja Flexbox CSS koda i mediaQuery-a koji se koriste za stiliziranje istih. Spomenuti atribut koristi se za određivanje smjera elemenata, odnosno hoće li elementi unutar njega biti stavljeni u red ili kolonu. Također koriste se strukturalne direktive kao ngIf i ngFor i ugrađena direktiva ngClass.

Osim HTML dokumenta, komponenta se sastoji i od Typescript dokumenta unutar kojeg se nalazi logika same komponente. Ista je prikazana slikom 4.21.

```typescript
@Component({
  selector: 'app-myhouse',
  templateUrl: './house.component.html',
  styleUrls: ['./house.component.scss']
})
export class HouseComponent implements OnInit, OnDestroy {

  // @ts-ignore
  ownHouses: House[];
  // @ts-ignore
  otherHouses: House[];
  // @ts-ignore
  isLoading: boolean;
  // @ts-ignore
  housesSubscription: any;

  constructor(
      public dialog: MatDialog,
      private route: ActivatedRoute,
      private houseService: HouseService
  ) { this.setSubscription(); }

  ngOnInit(): void {...}

  resetHouses(): void{...}

  initStartData(): void{...}

  checkIsEditing(): void{...}

  setSubscription(): void{...}

  openAddHouseDialog(): void{...}

  ngOnDestroy(): void {...}
}
```

Slika 4.21. Prikaz house.component.ts dokumenta

Unutar nje kreirali smo dvije varijable koje predstavljaju polja kuća, varijablu koja provjerava učitavaju li se još podaci i varijablu koja prati promjene u servisu. Koristeći konstruktor uključujemo određene vrijednosti koje koristimo za obavljanje određene funkcionalnosti. Svaka komponenta nasljeđuje OnInit klasu. Zbog nje možemo koristiti ngOnInit metodu kojom definiramo logiku koja se obavlja kod kreiranja same komponente. Jedna od njih je postavljanje varijable isLoading na logičku istinu, zatim varijable kuća postavljamo na prazna polja. Nakon toga provjeravamo postoji li parametar u ruti. Parametrima rute možemo pristupiti koristeći ruta

(engl. *Route*) varijablu uključenu kroz konstruktor. Ukoliko postoji, korisnika preusmjeravamo na stranicu za uređivanje kuće. Logika spomenutih funkcionalnosti prikazana je na slici 4.22.

```
ngOnInit(): void {
  this.initStartData();
  this.checkIsEditing();
}

resetHouses(): void{...}

initStartData(): void{
  this.isLoading = true;
  this.resetHouses();
}

checkIsEditing(): void{
  const houseId = this.route.snapshot.paramMap.get('id');
  if (houseId){
    const dialogRef = this.dialog.open(UpdateHouseDialogComponent, config: {height:'80%', width:'80%', data: {id: houseId}})
  }
}

setSubscription(): void{
  this.housesSubscription = this.houseService.onHousesChanged.subscribe( next: data => {
    this.ownHouses = data.ownHouses;
    this.otherHouses = data.otherHouses;
    this.isLoading = false;
  });
}
```

Slika 4.22. Logika pojedinih funkcionalnosti.

Unutar konstruktora pokrenuli smo metodu kojom se pretplaćujemo na promjene podataka o kućama unutar servisa odgovornog za kuće. Drugim riječima, čim se podaci o kući dohvate, komponenta će biti o tome obaviještena te će im moći pristupiti. Zadnji, ali jednako bitan dokument komponente, je SCSS dokument. Neke od kreiranih klasa unutar kojeg se nalazi CSS kod kao i primijenjene značajke SCSS-a možemo vidjeti na slici 4.23.

```
@use './src/styles';

.house-container{
  position:absolute;
  height:90%;
  width:100%;

  .user-houses-container {
    overflow-y: scroll;

      .add-house-container{...}

      .houses-container{...}
      .house-container-gt{...}
      .house-container-gt-xs{...}
      .house-container-gt-sm{...}
      .house-container-gt-md{...}
  }
}
```

Slika 4.23. Prikaz stilova komponente kuća

4.3.5. Servisi

Servisi su klase čija je odgovornost rukovanje podacima unutar aplikacije. Ovaj rad sadrži nekoliko servisa. Jedan od njih prikazan je slikom 4.24.

```
@Injectable({
  providedIn: 'root'
})
export class HouseService implements Resolve<any>{
  onHousesChanged: BehaviorSubject<any>;
  ownHouses: House[];
  otherHouses: House[];

  constructor(
    private http: HttpClient,
    private tokenStorage: TokenStorageService
  ) {...}

  initStartData(): void{...}

  resetHousesValue(): void{...}

  createHouse(house: House): Observable<any>{...}

  getHouses(): Observable<any> {...}

  getHouseById(id: number): Observable<any>{...}

  updateHouse(house: House): Observable<any>{...}

  removeById(id: number): Observable<any>{...}

  getStatisticsForToday(deviceIds: number[]): Observable<any>{...}

  getStatisticsForLastSevenDays(deviceIds: number[]): Observable<any>{...}

  getStatisticsForLastMonth(deviceIds: number[]): Observable<any>{...}

  resolve(): void {...}
}
```

Slika 4.24. Primjer servisa za rukovanje kućama

Vidimo kako spomenuti servis implementira riješenost (engl. *Resolve*) sučelje. To implementiranje zahtijeva prepisivanje metode riješi (engl. *Resolve*) koja se okida neposredno prije učitavanja stranice. Na taj način možemo definirati logiku za dohvaćanje kuća kako bi na vrijeme bile dostupne unutar komponente koja ih zahtijeva i skratili vrijeme čekanja na iste. Na

slici 4.19. ruta posjeduje parametar riješi (engl. *Resolve*) koji se povezuje sa servisom za kuće. Na taj način, prilikom posjeta rute okida se logika definirana u metodi riješi (engl. *Resolve*) na slici 4.25. Unutar servisa definirali smo varijable kuća i varijablu putem koje pretplaćene komponente obavještavamo da je došlo do promjene podataka.

```
getHouses(): Observable<any> {
  this.resetHousesValue();
  return this.http.get<any>(
    Urls.HOUSE_API()
  );
}

resolve(): void {
  this.getHouses().subscribe( next: data => {
    this.ownHouses = data.ownHouses;
    this.otherHouses = data.otherHouses;
    this.onHousesChanged.next(data);
  });
}
```

Slika 4.25. Logika metode za dohvaćanje kuća

Nakon što se podaci o kućama dohvate, pohranjujemo ih i šaljemo svim komponentama koje su pretplaćene na promjene varijable onHousesChanged. Konkretno izvršava se logika na slici 4.22. u metodi setSubscription.

4.3.6. Presretači

Presretači su klase koje implementiraju sučelje presretanja zahtjeva. Ono definira metodu presretni (engl. *Intercept*) unutar koje se definira logika presretanja zahtjeva. Unutar ove aplikacije nalazi se nekoliko presretača. Jedan od njih je i presretač autentifikacije koji je prikazan slikom 4.26. Unutar njega dohvaćamo token koristeći servis za tokene i provjeravamo kamo je zahtjev upućen. Tu provjeru radimo kako bi izbjegli dodavanje tokena u zaglavlje zahtjeva za rute gdje token nije potreban, kao npr. rute za prijavu i registraciju. Drugi presretač hvata i baca pogreške nakon slanja zahtjeva, ukoliko one postoje. Zbog toga u aplikaciji postoji globalni rukovatelj greškama koji implementira sučelje rukovatelja greškama i prepisuje metodu handleError. Unutar njega koristimo servis za greške i servis za notifikacije. Servis za greške provjerava o kojoj je vrsti greške riječ (greška sa strane servera ili klijenta) te prema tome vraća odgovarajuću poruku. Servis za notifikacije koristi komponentu iz Angular Material biblioteke kojom se prikazuju nastale pogreške.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor{
  constructor(
    private tokenStorage: TokenStorageService
  ) { }

  // tslint:disable-next-line:typedef
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const authToken = this.tokenStorage.getToken();
    const loginApiIsTriggered = Urls.AUTH_API() + 'login' === req.url;
    const registerApiIsTriggered = Urls.AUTH_API() + 'register' === req.url;
    const initDevicesApiIsTriggered = req.url.endsWith('ngrok.io/init');
    const toggleDeviceApiIsTriggered = req.url.includes('ngrok.io/device');

    if (authToken && !loginApiIsTriggered && !registerApiIsTriggered && !initDevicesApiIsTriggered && !toggleDeviceApiIsTriggered) {
      return next.handle(
        req.clone( update: {
          headers: req.headers.set('Authorization', `Bearer ${authToken}`),
        })
      )
    }
    return next.handle(req.clone())
  }
}
```

Slika 4.26. Prikaz AuthInterceptor-a

Prikaz globalnog rukovatelja pogreškama prikazan je na slici 4.27.

```
@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  constructor(
    private injector: Injector,
    private zone: NgZone,
    private router: Router
  ) { }

  handleError(error: Error | HttpErrorResponse): void {
    const errorStorage = this.injector.get(ErrorService);
    const notificationStorage = this.injector.get(NotificationStorageService);
    const userStorage = this.injector.get(UserService);

    let message;
    let stackTrace;

    if (error instanceof HttpErrorResponse){
      console.log(error)
      errorStorage.errorOccurred.next( value: true);
      message = errorStorage.getServerErrorMessage(error);
      notificationStorage.showError(message);

      if(error.status === 401){...}
      if(error.status === 404 ){...}
    }
    else {
      console.log(error)
      message = errorStorage.getClientErrorMessage(error)
      stackTrace = errorStorage.getClientErrorStackTrace(error)
      notificationStorage.showError(message)
    }
  }
}
```

Slika 4.27. Globalni rukovatelj pogreškama

4.3.7. Zaštita aplikacije

Bilo bi nezgodno kada bi korisnici mogli pristupati bilo kojoj stranici bez obzira na to jesu li trenutno prijavljeni ili ne u aplikaciju. Zbog toga koristimo zaštitnika (engl. *Guard*). Ako se vratimo na sliku 4.19. , možemo uočiti kako ruta ima parametar canActivate. Njoj smo pridružili klasu AuthGuard. AuthGuard jedna vrsta servisa koja implementira sučelje CanActivate te zbog toga prepisuje metodu canActivate. Ista je prikazana slikom 4.28.

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate{

  constructor(
    private router: Router,
    private userService: UserService
  ){ }


  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    if (!this.userService.isAuthenticated()){
      this.userService.logout();
      this.router.navigate( commands: ['login']);
      return false;
    }
    return true;
  }

}
```

Slika 4.28. Zaštitnik ruta

Kroz konstruktor dodajemo dvije varijable među kojima je i korisnički servis. Unutar njega je definirana ruta koja provjerava je li token istekao te prema tome vraća logičku istinu ili neistinu kao rezultat. Na osnovu tog rezultata, zaštitnik određuje hoće li korisnika pustiti na stranicu ili će ga odjaviti koristeći metodu iz korisničkog servisa i preusmjeriti na stranicu za prijavu.

4.4. Mikroračunalo

Nakon pokretanja mikroračunala, u isto se prijavljujemo koristeći RDC[10] sa našeg desktop računala. Nakon toga otvara se *xrdp* sučelje unutar kojeg je potrebno unijeti ime i lozinku koju smo postavili prilikom instalacije xrdp-a. Pokretanje lokalnog servera oslanja se na dvije Python datoteke koje smo kreirali. Jedna je app.py koja predstavlja aplikaciju (Slika 4.29.), dok je druga store.py (Slika 4.30.) koja nam služi kao rukovatelj uređajima.

```
from flask import Flask, jsonify, request, Response, make_response
from flask_cors import CORS
from gpiozero import LED
from time import sleep
import string
import random
import store

app = Flask(__name__)
CORS(app)

@app.route('/init', methods=['POST'])
def initDevices():
    data = request.json
    if(len(data) != 0):
        store.initDevices(data)
        devices = store.houseDevices
        print(devices)
        return Response(headers={'Access-Control-Allow-Origin':'*'})
    else:
        return Response('Something went wrong ops.',400)

@app.route('/device/<devicePin>')
def action(devicePin):
    print(store.houseDevices)
    device = store.fetchDevice(int(devicePin))
    if(device):
        device.toggle()
        return Response(headers={'Access-Control-Allow-Origin':'*'})
    else:
        return Response('There is no device with that pin.',400)

if __name__ == '__main__':
    app.run(debug=True, port=5005, host='0.0.0.0')
```

Slika 4.29. Datoteka app.py

Unutar aplikacijske datoteke kreiramo *app* varijablu koja predstavlja Flask objekt. Nakon toga definiramo dvije metode iznad kojih koristimo @app.route anotaciju. Jedna od metoda je metoda za inicijalizaciju uređaja koji se postavljaju kada se kuća kreira. Iznad nje koristeći spomenutu anotaciju definiramo rutu za koju će se spomenuta metoda pokrenuti. Unutar metode dohvaćamo podatke koji se šalju u tijelu zahtjeva. Nakon toga, provjerava se veličina podataka budući da se radi o polju soba unutar kojih su uređaji. Ukoliko podaci postoje, koristimo „store.py“ skriptu preko koje inicijaliziramo uređaje. Druga metoda je predstavlja metodu kojom palimo i gasimo uređaje. Iznad nje koristimo spomenutu anotaciju s definiranom rutom koja ima parametar devicePin. Tome parametru pristupamo tako što u metodi predamo varijablu istog imena kao parametar u ruti. Također koristimo „store.py“ skriptu kojom dohvaćamo uređaj te ukoliko on postoji mijenjamo mu status, a ukoliko ne postoji vraćamo odgovor s odgovarajućom porukom.

[10] engl. *Remote Desktop Control* – softver za povezivanje s udaljenim računalom.

```
from gpiozero import LED, Button
import json

houseDevices = []

def initDevices(rooms): # '{"type": "LED", "pin":17}'
    if (len(houseDevices)> 0):
        houseDevices.clear()
    for room in rooms:
        for device in room["devices"]:
            devicePin = device["pin"]
            deviceType = device["type"]
            if(deviceType == "Bulb" or deviceType == "Door"):
                if(not isExisting(devicePin)):
                    newDevice = LED(device["pin"])
                    houseDevices.append(newDevice)

def isExisting(devicePin):
    state = False
    for device in houseDevices:
        if(device.pin.number == devicePin):
            state = True
            return state
    return state


def fetchDevice(devicePin):
    for device in houseDevices:
        if(device.pin.number == devicePin):
            return device
```

Slika 4.30. Datoteka store.py

Datoteka „store.py“ sadrži varijablu koja predstavlja polje uređaja te tri metode. Prva metoda se odnosi na inicijalizaciju uređaja. Započinje čišćenjem varijable uređaja ukoliko oni postoje te nakon toga prolazimo kroz sobe kako bi dohvatili sve uređaje. Dodajemo one uređaje čiji pin nije zauzet od strane drugog uređaja, a tu provjeru vršimo pomoću jedne od metoda. Zadnja metoda dohvaća uređaj iz varijable uređaja na temelju predanog pina.

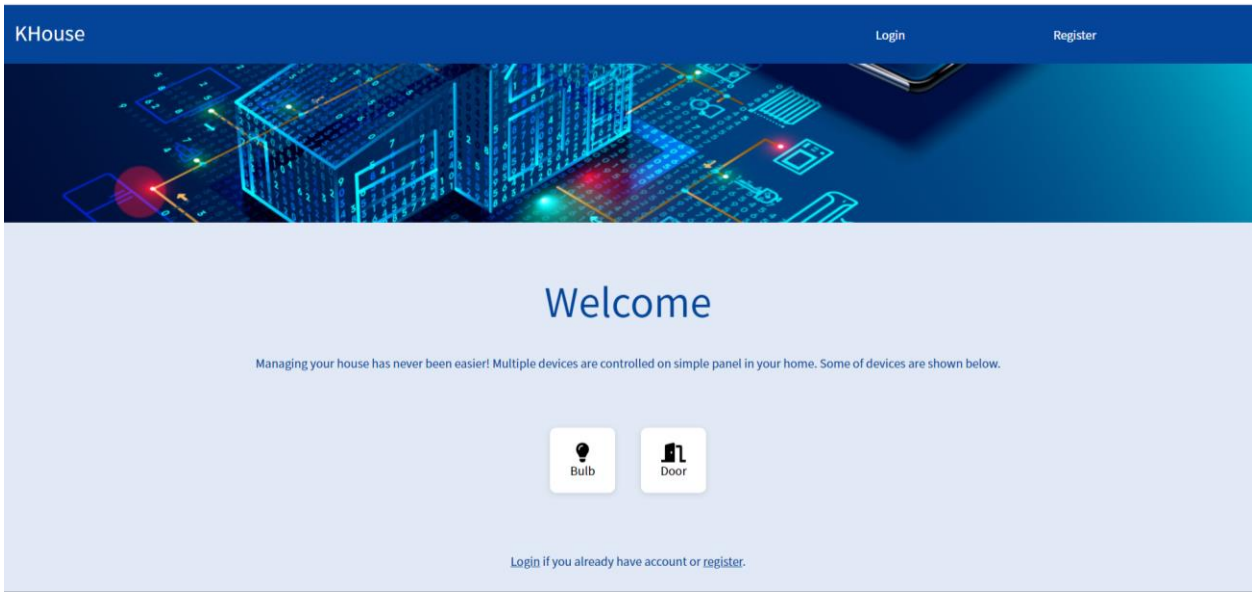
Nakon pokretanja app.py skripte kreira se lokalni web server na portu 5005. Kako bi njega učinili vidljivim i izvan lokalne mreže koristimo ngrok i naredbu spomenutu u poglavlju „Ngrok“. Nakon toga naš server je dostupan putem interneta koristeći jednu od poveznica prosljeđivanja.

```
ngrok by @inconshreveable                                                    (

Session Status                online
Account                       kristijan.koscak16@gmail.com (Plan: Free)
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://9d81-95-168-121-110.ngrok.io -> http://localhost:5005
Forwarding                    https://9d81-95-168-121-110.ngrok.io -> http://localhost:5005

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Slika 4.31. Rezultat pokretanja ngrok naredbe

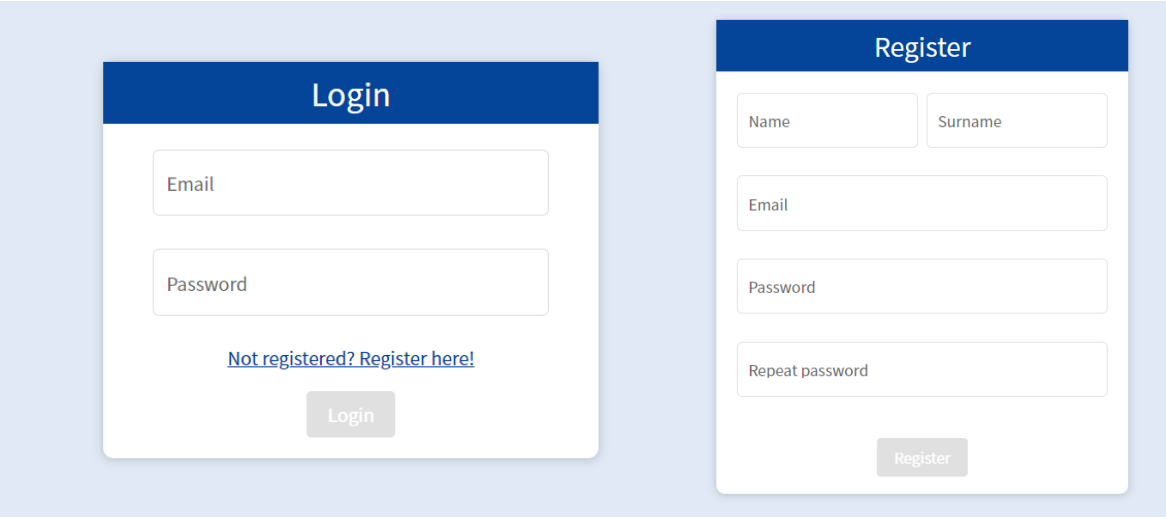
4.5. Izgled aplikacije

Nakon pokretanja serverskog dijela i klijentskog dijela možemo pristupiti web stranici lokalno koristeći sljedeću poveznicu: http://127.0.0.1:4200/ . Korisnika se preusmjerava na početnu stranicu koja je prikazana slikom 4.32.



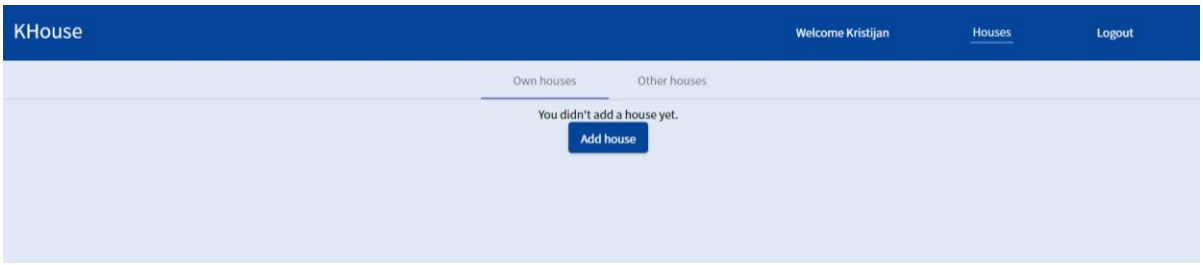
Slika 4.32. Prikaz početne stranice

Nakon toga korisnik ima mogućnost prijave ili registracije koje su prikazane slikom 4.33. Nakon uspješne registracije ili prijave, korisnika se preusmjerava na početnu stranicu, ali nešto drugačijom navigacijskom trakom.



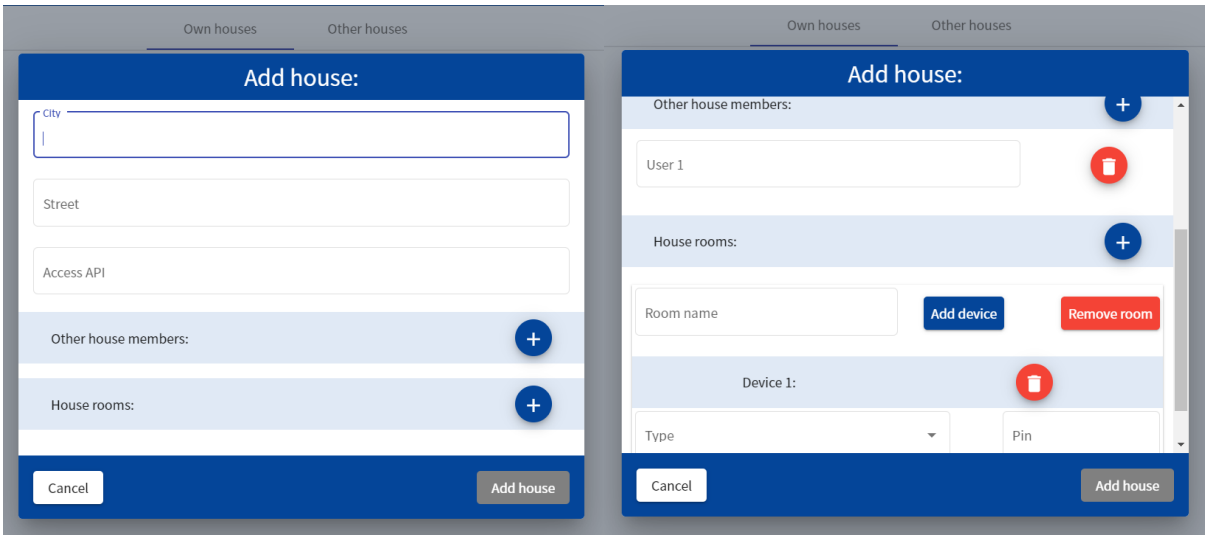
Slika 4.33. Prikaz forme za prijavu i registraciju

Korisnik može otići na stranicu kuće gdje mu se prikazuje sljedeća stranica (Slika 4.34.). Korisnik može pregledati svoje kuće kao i kuće kojima on ima pristup, ali i dodati novu kuću.



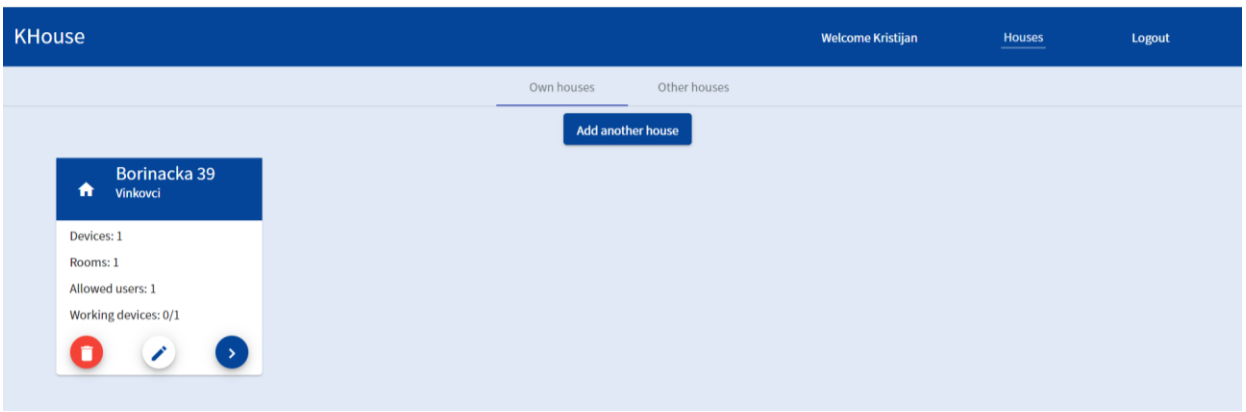
Slika 4.34. Prikaz stranice kuća

Klikom na dodaj kuću ,otvara se sljedeća forma prikazana slikom 4.35. Identična forma se koristi i za izmjenu postojećih kuća. Korisnik može dinamički dodavati i brisati polja za druge ukućane ili sobe, a unutar svake sobe polje za uređaj. Unutar polja Access API se unosi poveznica prosljeđivanja sa slike 4.31. Korisnik može dodati kući bez drugih korisnika i uređaja, ali ne može bez osnovnih podataka o kući.



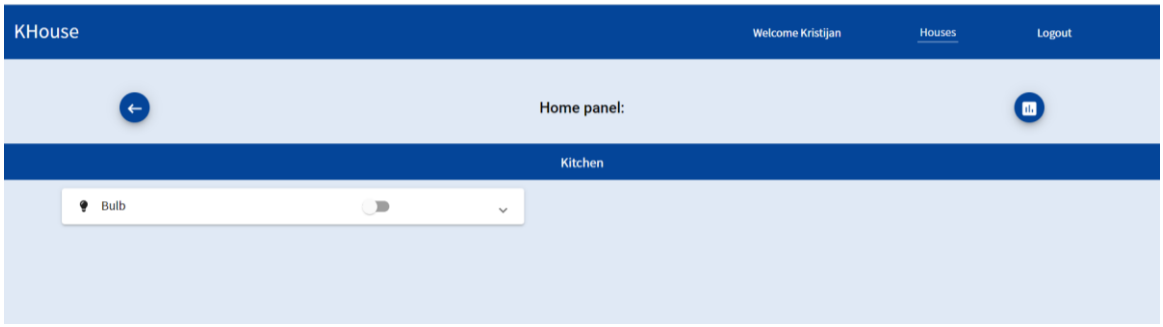
Slika 4.35. Prikaz forme za dodavanje i izmjenu kuća

Nakon uspješnog dodavanja kuće korisnika se preusmjerava na stranicu kuća koja je identična stranici na slici 4.34. ,ali ovaj puta ona ima podatke o kućama budući da smo ju dodali. Sada ta stranica izgleda ovako (Slika 4.36).



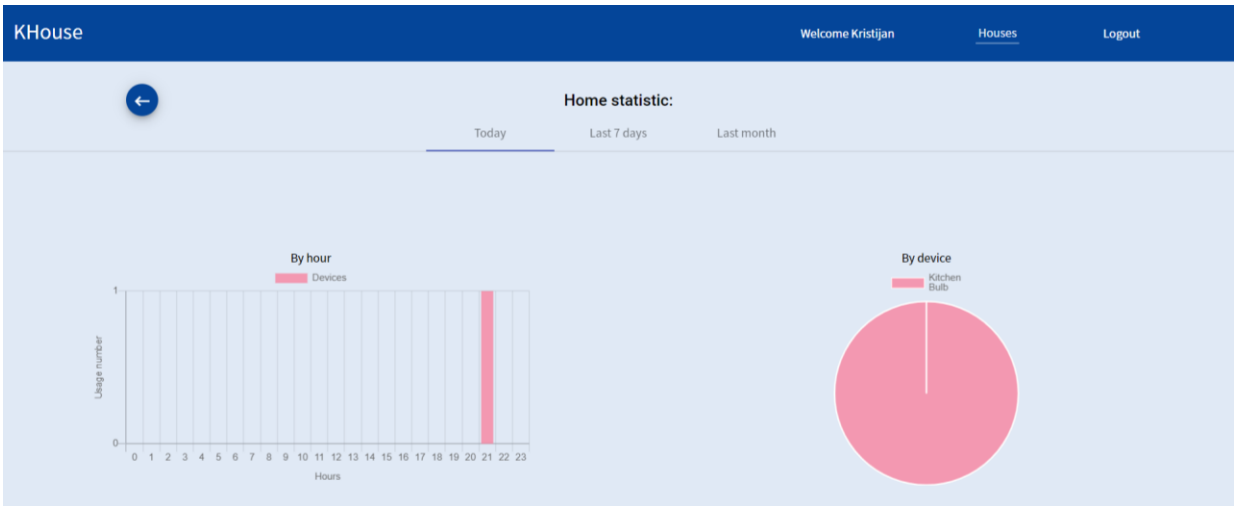
Slika 4.36. Prikaz stranice kuća sa podacima

Korisnik može pojedinu kuću obrisati, izmijeniti ili ući na kontrolnu ploču. Radi boljeg korisničkog iskustva prilikom brisanja od korisnika se zahtijeva da potvrdi tu akciju. Prilikom izmjene podataka unutar kuće, otvara se identična forma kao na slici 4.35. ,ali sa popunjenim informacijama unutar polja. Odlaskom na kontrolnu ploču otvara se stranica kao na slici 4.37. Na njoj se prikazuju sve sobe koje postoje unutar kuće kao i svaki uređaj unutar pojedine sobe. Svaki od uređaja korisnik može paliti i gasiti koristeći klizač.



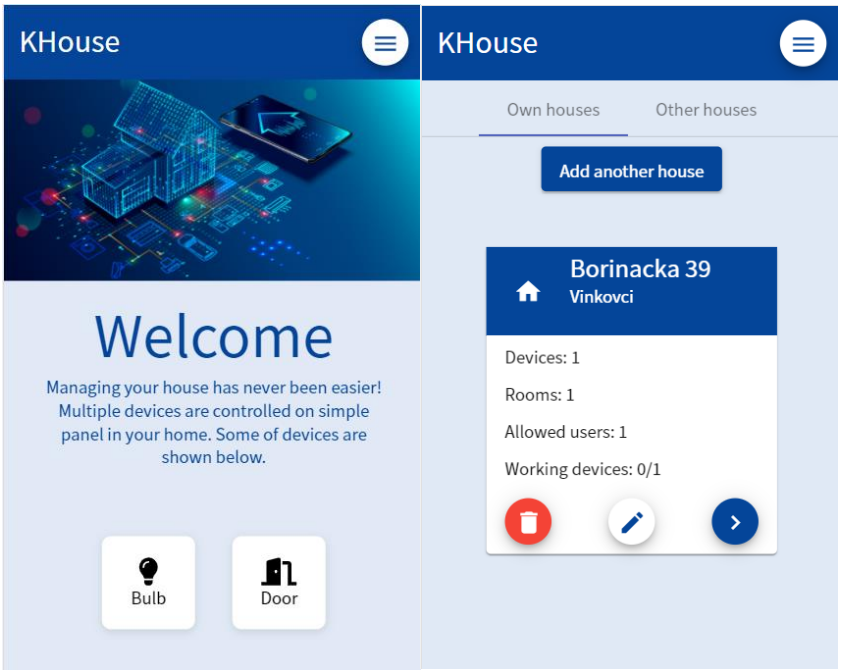
Slika 4.37. Kontrolna ploča pojedine kuće.

Sa kontrolne ploče možemo otići na stranicu sa statistikom pojedine kuće. Ta stranica prikazuje podatke o paljenju i gašenju uređaja na dnevnoj, tjednoj i mjesečnoj razini. Na dnevnoj razini prikazuje se broj korištenja uređaja za svaki sat kao i podatak o tome koji je uređaj koliko puta korišten. Na tjednoj i mjesečnoj razini prikazuje se broj korištenja uređaja tokom tjedna i podatak koji je uređaj koliko puta korišten. Sve je to prikazano 4.38. Budući da nemamo trenutno podatke na tjednoj i mjesečnoj razini prikazat ćemo podatke za dnevnu razinu nakon paljenja i gašenja uređaja.



Slika 4.38. Prikaz statistike na dnevnoj razini

Cijelo sučelje je je prilagođeno mobilnim i tablet uređajima te ćemo prikazati slike za neke od stranica. Sljedećom slikom prikazano je sučelje za mobilne uređaje (Slika 4.39).



Slika 4.39. Prikaz aplikacije na mobilnim uređajima

5. ZAKLJUČAK

U ovom diplomskom radu opisan je postupak kreiranja web aplikacije „Aplikacija za upravljanje pametnom kućom“. Cilj je bio realizirati cjelokupnu projekt koji se sastoji od korisničkog sučelja, servera na kojem se odvija sva logika vezana za podatke i hardvera na koji povezujemo elektroničke uređaje. Opisane su primijenjene tehnologije i alati koje tvore ovu aplikaciju. Korisničko sučelje omogućava korisnicima potpunu kontrolu nad kućom i uređajima koji su spojeni na istu. Sučelje pruža animacije kako bi se omogućilo bolje korisničko iskustvo. Serverski dio omogućava pristupne točke koje obavljaju određene operacije na temelju korisničkih zahtjeva. Kako bi se onemogućio neovlašteni pristup, koristimo Spring Security paket. Naše mikroračunalo Raspberry Pi omogućava korisnicima spajanje različitih uređaja kao što su npr. diode. One simuliraju žarulje koje korisnik može paliti i gasiti. Koristeći sučelje korisnik za sada može dodati dva uređaja, a to su žarulja ili vrata. Broj uređaja ovisi o broju pinova na mikroračunalu. Jedna od prednosti ovog projekta je responzivnost koja omogućava korištenje aplikacije na mobilnim, tablet i desktop uređajima. Glavna ideja je bila pružiti što jednostavnije sučelje za korištenje, kako bi se korisnici bilo koje dobi snašli. Aplikacija je otvorena za razna proširenja. Jedno od njih je mogućnost prijave koristeći Google ili Facebook račun. Druga, ozbiljnija mogućnost je pretvorba mikroračunala u pristupnu točku koristeći Wi-Fi. Time bismo mogli koristiti postojeće Wi-Fi uređaje i povezati na mikroračunalo te više ne bi morali voditi brigo o tome koji je uređaj spojen na kojem pinu. S druge strane, to zahtijeva znatnu promjenu implementacije.

LITERAURA

[1] What is Information Technology (IT) [online], dostupno na: https://www.snhu.edu/about-us/newsroom/2018/07/what-is-information-technology [Rujan 2021.]

[2] Microcomputer [online], dostupno na: https://www.britannica.com/technology/microcomputer [Rujan 2021.]

[3] Smart home [online], dostupno na: https://www.investopedia.com/terms/s/smart-home.asp [Rujan 2021.]

[4] Home automation [online], dostupno na: https://www.security.org/home-automation/ [Rujan 2021.]

[5] Home Assistant [online], dostupno na: https://futurehousestore.co.uk/what-is-home-assistant-and-what-it-can-do [Rujan 2021.]

[6] Home Assistan demo sučelje [online], dostupno na: https://demo.home-assistant.io/#/lovelace/0 [Rujan 2021.]

[7] Homebridge [online], dostupno na: https://github.com/homebridge/homebridge [Rujan 2021.]

[8] HomeKit API [online], dostupno na: https://www.programmableweb.com/api/apple-homekit [Rujan 2021.]

[9] Homebridge primjer sučelja [online], dostupno na: https://raw.githubusercontent.com/oznu/homebridge-config-ui-x/master/screenshots/homebridge-config-ui-x-status.png [Rujan 2021.]

[10] Spring Framework [online], dostupno na: https://spring.io/projects/spring-framework#overview [Rujan 2021.]

[11] Spring Framework [online], dostupno na: https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot [Rujan 2021.]

[12] Spring Boot [online], dostupno na: https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot [Rujan 2021.]

[13] Spring Boot [online], dostupno na: https://www.javatpoint.com/spring-boot-tutorial [Rujan 2021.]

[14] Spring Data JPA [online], dostupno na: https://www.javatpoint.com/spring-boot-jpa [Rujan 2021.]

[15] Spring Data JPA [online], dostupno na: https://spring.io/projects/spring-data-jpa [Rujan 2021.]

[16] Hibernate [online], dostupno na: https://www.tutorialspoint.com/hibernate/orm_overview.htm [Rujan 2021.]

[17] Spring security [online], dostupno na: https://spring.io/projects/spring-security [Rujan 2021.]

[18] Spring Boot REST API [online], dostupno na: https://www.tutorialspoint.com/spring_boot/spring_boot_building_restful_web_services.htm [Rujan 2021.]

[19] Flyway [online], dostupno na: https://www.callicoder.com/spring-boot-flyway-database-migration-example/ [Rujan 2021.]

[20] Flyway konvencija imenovanja [online], dostupno na: https://medium.com/@tejozarkar/configure-flyway-with-spring-boot-9493aebf336b [Rujan 2021.]

[21] Angular [online], dostupno na: https://angular.io/guide/what-is-angular [Rujan 2021.]

[22] Komponenta [online], dostupno na: https://angular.io/api/core/Component [Rujan 2021.]

[23] Modul [online], dostupno na: https://angular.io/guide/architecture-modules [Rujan 2021.]

[24] Presretač [online], dostupno na: https://angular.io/api/common/http/HttpInterceptor [Rujan 2021.]

[25] Angular mateiral [online], dostupno na: https://material.angular.io/ [Rujan 2021.]

[26] Direktive [online], dostupno na: https://angular.io/guide/built-in-directives [Rujan 2021.]

[27] Animacije [online], dostupno na: https://angular.io/guide/animations [Rujan 2021.]

[28] SCSS [online], dostupno na: https://sass-lang.com/ [Rujan 2021.]

[29] Raspberry Pi [online], dostupno na: https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/ [Rujan 2021.]

[30] Python [online], dostupno na: https://realpython.com/python-raspberry-pi/ [Rujan 2021.]

[31] xdrp [online], dostupno na: http://xrdp.org/ [Rujan 2021.]

[32] Flask [online], dostupno na: https://www.geeksforgeeks.org/python-introduction-to-web-development-using-flask/ [Rujan 2021.]

[33] Ngrok [online], dostupno na: https://ngrok.com/docs [Rujan 2021.]

[34] PostgreSQL [online], dostupno na: https://www.postgresql.org/ [Rujan 2021.]

[35] Docker [online], dostupno na: https://docs.docker.com/get-started/overview/ [Rujan 2021.]

[36] IntelliJ [online], dostupno na: https://www.jetbrains.com/idea/features/ [Rujan 2021.]

SAŽETAK

Ideja ovog diplomskog rada bila je stvoriti web aplikaciju koja će omogućiti jednostavnije upravljanje pametnim kućama. Pojedina postojeća rješenja imaju sučelja koja su često nepregledna. Za izradu je potrebno poznavanje Angular razvojnog okruženja, HTML, CSS, TypeScript-a, Java programskog jezika kao i Spring Boot okvira. Osim navedenih poželjno je poznavanje osnova Python programskog jezika kao, prethodno korištenje Raspberry Pi-a i Docker-a koji smo koristili za postavljanje i pokretanje baze podataka. Realizacijom ovog rada kreirali smo jednostavnu web aplikaciju sa jednom stranicom na kojoj se izmjenjuju komponente. Omogućili smo prijavu i registraciju korisnika. Registrirani korisnici mogu pregledavati, dodavati, uređivati i brisati kuće iz baze podataka. Na svakoj kući vlasnici i omogućeni korisnici mogu upravljati kućanskim uređajima kao i pregledavati statistiku za pojedinu kuću. Statistika prikazuje podatke o ukupnom broju korištenju uređaja kao i broj korištenja pojedinih uređaja na dnevnoj, tjednoj i mjesečnoj razini.

Ključne riječi: Angular, Docker, PostgreSQL, Raspberry Pi, Spring Boot

ABSTRACT

Application for smart home management

The idea of this graduate thesis was to create web application which will provide easier management of smart homes. Some of existing solutions have interfaces that are often opaque. For developing it is necessary knowledge of Angular framework, HTML, CSS, TypeScript, Java programming language as well as Spring Boot framework. Except these it is also necessary having knowledge of Python programming language basics, previous usage of Raspberry Pi and Docker which we used for setting and running database. Realization of this application we created simple web application with one page on which components are changed. We enabled user login and registration. Registered user can watch, create, update and delete houses from database. Each house enables owner and its users can manage house devices as well as see statistics for each house. Statistics show usage data of devices and number of usage for each device daily, weekly and monthly.

Keywords: Angular, Docker, PostgreSQL, Raspberry Pi, Spring Boot

ŽIVOTOPIS

Kristijan Koščak rođen je 16.02.1998. godine u Vinkovcima. Nakon završenog osnovnog obrazovanja koje je započelo 2004. godine u Osnovnoj školi Antun Gustav Matoš, Vinkovci upisuje srednju školu. Obrazovanje nastavlja 2012. godine u Tehničkoj školi Ruđera Boškovića u Vinkovcima gdje upisuje smjer elektrotehnika. 2016. godine upisuje preddiplomski studij, smjer računarstva na Fakultetu elektrotehnike, računarstva, elektrotehnike i informacijskih tehnologija u Osijeku koji je u sklopu Sveučilišta Josipa Jurja Strossmayera. Nakon uspješnog završetka preddiplomskog studija uspije diplomski studij računarstva, smjer Programsko inženjerstvo na istom fakultetu. Tijekom studija postaje stipendist tvrtke DICE Digital Innovation Center d.o.o. U sklopu iste firme odrađuje potrebnu studentsku praksu.

_____

KRISTIJAN KOŠČAK

PRILOZI

Na CD-u priloženom uz Diplomski rad nalaze se dokumenti:

diplomski_kkoscak.docx

diplomski_kkoscak.pdf

WebAplikacija.zip