Progresivna web-aplikacija za alarmiranje u slučaju mraza

Jović, Josipa

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1169

PROGRESIVNA WEB-APLIKACIJA ZA ALARMIRANJE U SLUČAJU MRAZA

Josipa Jović

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1169

PROGRESIVNA WEB-APLIKACIJA ZA ALARMIRANJE U SLUČAJU MRAZA

Josipa Jović

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

ZAVRŠNI ZADATAK br. 1169

Pristupnica:	**Josipa Jović (0036529655)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Mario Kušek

Zadatak: **Progresivna web-aplikacija za alarmiranje u slučaju mraza**

Opis zadatka:

U današnjem Internetu značajno raste broj jednostavnih uređaja poput senzorskih i aktuatorskih čvorova koji prikupljaju i odašilju podatke s različitih senzora te primaju naredbe koje izvršavaju u fizičkom svijetu. Takvi čvorovi su obično vezani za pojedine stvari i tvore Internet stvari (Internet of Things - IoT). U laboratoriju za Internet stvari napravljena je platforma koja služi za povezivanje uređaja u Internetu stvari, spremanje njihovih podatka i kroz aplikacijsko programsko sučelje dohvaćanje podataka pomoću REST sučelja. Vaš zadatak je napraviti progresivnu web-aplikaciju za primanje alarma na pokretnom uređaju. Poslužiteljski dio je potrebno ugraditi u postojeću FER-ovu IoT platformu. Alarmi koji se šalju trebaju biti vezani za prognozu vremena npr. kada je u proljeće prognoziran mraz večer prije je potrebno alarmirati poljoprivrednika, a tijekom noći također ako temperatura na postavljenoj lokaciji padne blizu nule. Svu potrebnu literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

Rok za predaju rada: 9. lipnja 2023.

SADRŽAJ

1. Uvod

Platforma kao što je Web se konstantno razvija i mijenja te nam se zbog toga nudi bezbroj mogućnosti kada je u pitanju razvoj aplikacija za nju. Kao nova vrsta, osim tzv. *native* aplikacija i klasičnih web stranica, javljaju se progresivne web-aplikacije, PWA, koje pružaju iskustvo slično onome native aplikacija, ali s većim dosegom. PWA su dizajnirane s namjerom da ih se može koristiti na bilo kojem uređaju i na bilo kojem pregledniku koji je usklađen s odgovarajućim web-standardima.

Cilj ovog rada je razvoj progresivne web-aplikacije za alarmiranje u slučaju mraza. Mraz u sklopu ove aplikacije definiramo kao slučaj kada temperatura zraka pročitana na senzorima iznosi manje od 0°C. Svrha alarmiranja je upozoravanje korisnika na opasnost mraza koji može biti poguban za biljke. Alarmi se šalju na PWA u obliku *push* obavijesti, vrsta obavijesti koja se sastoji od naslova, poruke, slike i URL-a, a pojavljuje se na vrhu ili dnu desne strane zaslona.

U prvom dijelu se pobliže upoznajemo s pojmom PWA i istražujemo njene ključne karakteristike. Da bismo mogli klasificirati aplikaciju kao PWA, potrebno je zadovoljiti određene kriterije. To uključuje mogućnost pristupa aplikaciji putem web preglednika bez preuzimanja i instalacije, podršku za rad offline i slabo povezanim mrežama i mnoge druge.

U drugom dijelu razmatramo specifikacije zadanog zadatka, alarmiranje u slučaju mraza, te opisujemo rješenje i izradu aplikacije pomoću alata Spring Boot i biblioteke React.js. Spring Boot nam omogućuje brzo i jednostavno kreiranje RESTful API-ja te upravljanje podacima. S druge strane, React.js nam pruža efikasno korisničko sučelje, koje je sposobno dinamički reagirati na promjene podataka. Na poslijetku prikazujemo primjere korištenja aplikacije.

2. Progresivne web-aplikacije

Progresivne web-aplikacije (PWA) su inovacija u području razvoja aplikacija za web. Posjeduju značajke *native* aplikacija i klasičnih web stranica. Karakteristike PWA koje moramo istaknuti su mogućnost instaliranja, pouzdanost i responzivnost.

2.1. Značajke progresivne web-aplikacije

Najvažnija značajka PWA je mogućnost instaliranja. Možemo ih pokrenuti sa radne površine uređaja korisnika, te potražiti ih u izborniku aplikacija, što im daje isti osjećaj kao *native* aplikacija. Kriteriji koje moramo zadovoljiti kako bi aplikacija mogla biti instalirana na uređaj su:

- ispravno postavljen manifest.json,
- registriran *service worker*,
- korištenje HTTPS veze.

Osim mogućnosti instaliranja važna značajka je pouzdanost. Pouzdanost znači da aplikacija radi brzo i bez internetske veze. Aplikacija se ne smije srušiti ili postati neresponzivna kada se izgubi veza, već je potrebno imati i podatke koji se uvijek mogu učitati. Brzina učitavanja je također veliki čimbenik koji utječe na pozitivno korisničko iskustvo. Ako je aplikacija brza, veća je vjerojatnost da će korisnik nastaviti koristiti aplikaciju, jer je u današnjem svijetu većina korisnika već naviknuta na glatko i nesmetano korištenje aplikacija sa vrlo malim vremeom čekanja.

Na poslijetku imamo i responzivnost kao jednu od glavnih značajki PWA. Aplikacija mora biti kompatibilna s različitim veličinama uređaja kao npr. mobitel, tablet, laptop. To znači da aplikaciju možemo koristi bez poteškoća na svim našim uređajima, te osigurava izvrsno korisničko iskustvo za sve korisnike.

2.2. Karakteristični dijelovi progresivne web-aplikacije

Karakteristični dijelovi koje moramo dodati kako bi mogli nazvati svoju aplikaciju progresivnom web-aplikacijom su: Manifest datoteka i *service worker*.

2.2.1. Manifest datoteka

Manifest datoteka je datoteka tipa JSON koja se koristi za definiranje metapodataka i konfiguracije aplikacije, ona pruža informacije pregledniku i operativnom sustavu o tome kako aplikacija treba biti prikazana i ponašati se na korisničkom uređaju. Manifest datoteka mora sadržavati sljedeća svojstva:

- *short name* ili samo *name*,
    - *short name* se koristi na zaslonu korisnika, a *name* kod instalacije aplikacije. Ako *short name* nije postavljen, u oba slučaja se koristi *name*.
- *icons*,
    - svojstvo koje sadrži polje slika, potrebno je dodati ikone veličina 192x192 i 512x512 piksela.
    - za korištenje tzv. prilagodljivih, maskirajućih ikona, potrebno je dodati i *"purpose": "any maskable"* s svojstvu *icons*.
- *display*,
    - Definira kako će se aplikacija prikazivati korisnicima. Moguće vrijednosti uključuju *"fullscreen", "standalone", "minimal-ui", "browser"*.
- *start url*,
    - URL na kojem se aplikacija pokreće kada je instalirana ili pokrenuta
- *background color*.
    - Pozadinska boja aplikacije koja će se prikazivati tijekom pokretanja.

Navedena svojstva su ključna za ispravan rad PWA, ali je moguće dodati još razna svojstva. Manifest datoteku dodajemo u aplikaciju na način da u izvornom direktroiju stvorimo direktorij *public*, te je dodamo unutra zajedno sa slikama koje smo naveli u datoteci u polju *icons*. Unutar istog direktorija se nalazi index.html datoteka koja služi kao početna stranica, kada korisnik prvo pristupi određenoj domeni, poslužitelj će prvo poslati index.html kao odgovor na taj zahtjev.

2.2.2. Service worker

Service worker je najvažnija značajka koja omogućuje rad progresivnih web-aplikacija. To je JavaScript radnik koji djeluje kao posrednik između naše aplikacije i mreže. Upravlja priručnom memorijom (*cache*) i bavi se presretanjem mrežnih zahtjeva, kako bi omogućio nesmetan rad bez obzira na povezanost s mrežom.

Rad *service workera* unutar PWA odvija se kao ciklus, a sastoji se od sljedećih stanja:

- Registracija: Web stranica registrira *service worker* putem JavaScript koda. Registracija se obično događa pri učitavanju stranice ili pri ponovnom posjetu web aplikaciji.
- Instalacija: Nakon registracije, Service Worker skripta se preuzima i instalira u pozadini web preglednika. Prilikom instalacije, skripta može izvršiti inicijalizaciju, postaviti *cache* memoriju i preuzeti statičke resurse potrebne za *offline* podršku.
- Aktivacija: Kada je *service worker* spreman preuzeti kontrolu nad upravljanjem aplikacije, prelazi u stanje aktivacije. Ovo stanje se često koristi za čišćenje priručne memorije.
- Slušanje događaja: Aktivirani Service Worker osluškuje različite događaje koji se događaju u web pregledniku. Na primjer, osluškuje *fetch* događaje koji se pokreću kada se šalju zahtjevi za resursima.
- Obrada zahtjeva i događaja: Kada se dogodi odgovarajući događaj, *service worker* može intervenirati i obraditi taj zahtjev ili događaj. Na primjer, može preusmjeriti zahtjeve na lokalno pohranjene resurse iz cache memorije kako bi omogućio offline podršku.

Primjer koda za registraciju *service workera*:

```
window.addEventListener('load', async () => {
if ('serviceWorker' in navigator) {
try {
  await navigator.serviceWorker.register('./sw.js');
} catch (err) {
  console.log('ServiceWorker registration failed', err);
}
}
});
```

Za mogućnost instalacije dodajemo sljedeću liniju koda:

```
self.addEventListener('install', async installEvent => {});
```

Nakon toga prelazi u stanje aktivacije:

```
self.addEventListener('activate', async activateEvent => {});
```

Nakon što je *service worker* instaliran i aktiviran, on kontrolira stranicu kako bi pružio poboljšanu pouzdanost i brzinu.

2.3. Podrška za PWA

Jedna od ključnih prednosti progresivnih web-aplikacija je njihova sposobnost prilagodbe, što znači da uvijek pružaju optimalno korisničko iskustvo temeljeno na karakteristikama ekrana na kojem se prikazuju i mogućnostima preglednika na kojem se izvršavaju. PWA su relativno nova inovacija u području razvijanja aplikacija za web, pa tako se podrška za njih razlikuje od preglednika do preglednika. Na tablici 2.1 možemo vidjeti podršku PWA na najpoznatijim preglednicima i operacijskim sustavima.

Tablica 2.1: Podrška preglednika za PWA [7]

Preglednik	Windows	macOs	Linux	Android	iOS
Chrome	Da	Da	Da	Da	-
Safari	-	Da	-	-	Da
Edge	Da	Da	-	Da	-
Firefox	Da	Da	Da	Da	Ne
Opera	Da	Da	Da	Da	-

Tablica 2.2: Podrška preglednika za web *push* obavijesti [7]

Preglednik	Windows	macOs	Linux	Android	iOS
Chrome	Da	Da	Da	Da	Ne
Safari	-	Da	-	-	Ne
Edge	Da	Da	-	Da	Ne
Firefox	Da	Da	Da	Da	Ne
Opera	Da	Da	Da	Da	Ne

Podrška za PWA može se periodično mijenjati kako pružatelji preglednika objavljuju ažuriranja i uvode promjene u svoje preglednike. Bitno je biti informiran o izdanjima preglednika i kompatibilnosti kako bi se osiguralo dosljedno i ažurirano iskustvo PWA-a na različitim platformama i preglednicima.

Iako određeni preglednici podržavaju PWA, to ne znači i da podržavaju sve njene mogućnosti i karakteristike. U slučaju aplikacije s alarmiranjem, važno je istražiti koji preglednici podržavaju primanje *push* obavijesti (Tablica 2.2). Kao što se može uočiti, svi operacijski sustavi osim iOS-a, podržavaju primanje web *push* obavijesti.

3. Razvoj progresivne web-aplikacije za alarmiranje

3.1. Pregled zahtjeva

U laboratoriju za Internet stvari razvijena je platforma koja omogućuje povezivanje i upravljanje uređajima u Internetu stvari. Ova platforma osigurava skladištenje podataka uređaja i omogućuje korisnicima da preko jednostavnog REST sučelja dohvaćaju te podatke. Razvijamo aplikaciju koja dohvaća te podatke i na temelju njih šalje alarme. Aplikacija koju razvijamo mora zadovoljavati sljedeće zahtjeve:

- Web-aplikacija mora biti PWA koju možemo koristiti na pokretnom uređaju, što znači da moramo implementirati *service workera* i dodati manifest.json
- Alarmi se mogu postavljati, brisati i izmjenjivati posebno za svakog korisnika,
- Alarme je moguće uključiti i isključiti, bez da ih nužno obrišemo iz baze alarma,
- Alarmi se šalju na temelju očitanja senzora za temperaturu iz IoT vrta,
- Autentifikacija na stranici pomoću Keycloak *logina*.

3.2. Arhitektura sustava

Naša aplikacija za alarmiranje izgrađena je od sljedećih dijelova:

- Klijent, tj. PWA napravljena u React.js-u,
- Poslužitelj, napravljen u razvojnom okviru Spring Boot,
- Influx baza podataka koja sadrži očitanja senzora,
- Keycloak, koji koristimo za autentifikaciju i sigurnost aplikacije.

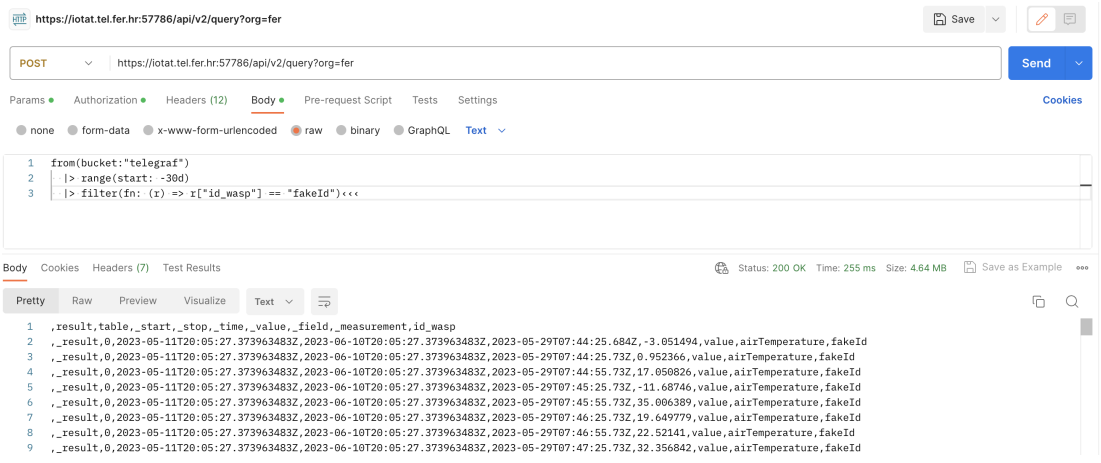
Klijent se spaja na poslužitelj pomoću REST API-ja definiranih na poslužitelju. Poslužitelj zatim šalje zahtjeve na Influx bazu podataka i vraća odgovore poslužitelju tj.

u našem slučaju šalje nazad alarme u obliku *push* obavijesti. Klijent se također spaja na Keycloak, te pri prijavi korisnika je preusmjeren na Keycloak login. Poslužitelj se također spaja na Keycloak kako bi mogao autentificirati zahtjeve klijenta. Klijent na poslužitelj, uz svoj zahtjev, šalje i tzv. *acces token* dobijen od strane Keycloaka.

3.3. InfluxDB

InfluxDB je TSDB (*Time Series Database*) otvorenog koda, koja se koristi za pohranu i dohvat serija podataka u područjima poput senzorskih podataka Internet of Things (IoT) koje koristimo u našoj aplikaciji. Vremenski serijalizirani podaci (*time series data*) su podaci koji su organizirani prema vremenu, skup promatranja za određeni entitet u različitim vremenskim intervalima. InfluxDB omogućuje pohranu, praćenje, vizualizaciju, prikupljanje podataka i pružanje upozorenja u vremenskim serijama podataka.

Podaci koje provjeravamo s namjerom slanja alarma, dobivamo iz InfluxDB baze podataka. Primjer načina na koji šaljemo zahtjev prikazan je na slici 3.1. Uz podatke koji su prikazani, moramo u Headers moramo poslati i token koji nam omogućuje pristup podacima u bazi. Zaglavlja "Accept" i "Content-type" moraju biti promjenjena kako bi kako bi odgovarala prihvatljivim formatima podataka zahtijevanim od strane InfluxDB-a. "Accept" naznačuje format koji želimo prihvatiti kao odgovor od poslužitelja, a to je "application/csv". "Content-type" definira format podataka koje šaljemo na poslužitelj, u našem slučaju je to "application/vnd.flux".



Slika 3.1: *Slanje POST zahtjeva InfluxDB bazu*

3.4. Poslužiteljska strana

Poslužiteljsku stranu aplikacije izrađujemo kao Spring Boot aplikaciju. Spring Boot pruža alate i okvir za razvoj poslužiteljske logike koja omogućuje aplikaciji da primi i obrađuje zahtjeve klijenata na učinkovit način. U ovom poglavlju opisat ćemo rad RESTful API-ja, upravljanje ruta i putanja te obradu ulaznih zahtjeva i generiranje odgovora. Također prikazujemo rad sa WebSockets, njihova konfiguracija i sposobnost slanja poruka na različite kanale. Obraditi ćemo i konfiguraciju Keycloak autentifikacije kako bi osigurali sigurnost i autentifikaciju korisnika u aplikaciji.

3.4.1. Arhitektura Spring Boot aplikacije

Arhitektura naše aplikacije je izrađena po poznatom modelu MVC (Model - View - Controller). MVC je obrazac softverske arhitekture koji služi organizaciji koda ovisno o njegovoj namjeni. U našoj aplikaciji nismo koristili View aspekt ove arhitekture, već imamo podjelu na Model, Controller i Service.

Model se odnosi na podatke, objekte koji predstavljaju stvarne entitete koje koristimo u aplikaciji. U našoj aplikaciji možemo vidjeti pod Model spadaju tri klase: *AlarmData, Data* i *PushNotification*. *Data* predstavlja podatke koje klijent šalje na poslužitelj, kako bi stvorio novi alarm. *AlarmData* predstavlja podatke o alarmima koji su, nakon zahtjeva klijenta, zapisani u text datoteku *alarms*. *PushNotification* je entitet koji predstavlja potisnu obavijesti, sadrži atribute naziv i poruka.

Service sadrži metode koje obavljaju složene poslove, koristi se za izvršavanje određenih poslova većinom iz kontrolera. U našoj aplikaciji nalazimo klase *WebSocketConfig, WebSocketSecurityConfig, AlarmFileReader, WebSecurityConfiguration, PushNotificationService*.

Klase *WebSocketConfig i WebSocketSecurityConfig* koriste se za konfiguraciju *Websocket* protokola. *WebSecurityConfiguration* koristimo za konfiguraciju sigurnosti naše aplikacije. Metoda filterChain konfigurira sigurnosni filter lanac za HTTP zahtjeve. U ovom primjeru, konfiguracija uključuje onemogućavanje CSRF(*Cross-Site Request Forgery*) zaštite, omogućavanje CORS (*Cross-Origin Resource Sharing*) podrške te definiranje pravila za odobravanje pristupa određenim URL-ovima. Također se konfigurira OAuth2 Resource Server za provjeru i dekodiranje JWT (*JSON Web Token*) tokena.

CSRF je sigurnosna ranjivost na webu koja napadaču omogućuje da navede korisnike da izvrše radnje koje nisu namjeravali izvesti [5]. Spring Boot ima ugrađenu

CSRF zaštitu koja pomaže u sprečavanju ovakvih napada. Takvu zaštitu za ovakvu vrstu napada isključujemo jer koristimo autentifikaciju s tokenima JWT kako bi spriječili neželjene zahtjeve.

CORS je mehanizam zasnovan na HTTP zaglavljima koji omogućava poslužitelju da naznači druge izvore (domene, sheme ili portove) osim vlastitog, s kojih preglednik treba dopustiti učitavanje resursa [1]. CORS omogućava web aplikacijama na jednoj domeni da komuniciraju i dijele resurse s aplikacijama na drugim domenama. CORS podrška je potrebna kako bi *frontend* mogao komunicirati s *backendom*.

Klasa *PushNotificationService* implementira zakazano provjeravanje podataka pročitanih na senzorima i slanje alarma pomoću Spring anotacije *scheduling* i klasa PushNotificaton, SimpMessagingTemplate među ostalim. Ova klasa implementira glavni dio zadatka naše aplikacije te ćemo njen rad objasniti detaljnije u potpoglavlju 3.4.3.

Controller tj. Upravitelj, upravlja korisničkim zahtjevima. Prima HTTP zahtjeve, obrađuje ih i komunicira s modelom i servisom kako bi generirao odgovor. U našoj aplikaciji imamo *TemperatureController* koji predstavlja REST kontroler, a sadrži metode addData, getAlarms, deleteAlarm i updateAlarm. Metoda addData predstavlja POST zahtjev na '/create' endpointu. Ova metoda prima JSON podatke koji se parsiraju i nakon toga se dodaju u datoteku alarms.txt. Na taj način izrađujemo nove alarme za pojedine korisnike. Metoda getAlarms predstavlja POST zahtjev pomoću kojega korisnik na klijentu dohvaća sve alarme koji su namijenjeni za njega. Metoda deleteAlarm je DELETE zahtjev na '/id' endpointu, pri čemu je id jedinstveni id alarma kojeg želimo obrisati. Slično metodi deleteAlarm, updateAlarm je PUT zahtjev na endpointu '/update/id', ali se ona koristi za ažuriranje podataka postojećih alarma.

3.4.2. REST API za rad s alarmima

REST API (*Representational State Transfer Application Programming Interface*) je arhitekturalni stil i skup pravila za razvoj web usluga koje omogućuju komunikaciju i razmjenu podataka između različitih računalnih sustava. REST API se temelji na principima HTTP protokola i koristi HTTP metode kao što su GET, POST, PUT, DELETE za manipulaciju podacima. U aplikaciji za alarmiranje, kao što smo naveli u prethodnom potpoglavlju 3.4.1, imamo tri metode, addData, updateAlarm i deleteAlarm.

Metoda addData, koju vidimo na slici 3.3, uzima podatke poslane pomoću POST zahtjeva, a primjer poslanog zahtjeva možemo vidjeti na slici 3.2. Podatke koje zahtjev treba sadržavati su sljedeći:

- *alarmId* - jedinstveni broj dodijeljen svakom alarmu

– *deviceId* - jedinstveni id dodijeljen uređaju na kojem se nalazi senzor

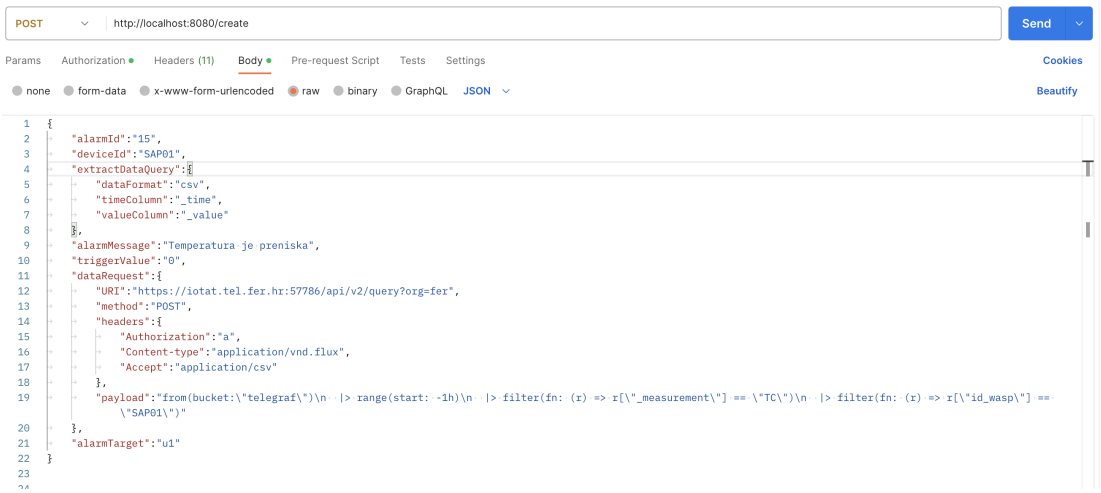
– *extractDataQuery* - govori nam o tipu podataka i vrijednostima koje dohvaćamo iz baze

– *alarmMessage* - poruka koja se šalje zajedno s alarmom

– *triggerValue* - temperatura koja okida slanje alarma

– *dataRequest* - zahtjev koji se šalje na bazu gdje su zapisana mjerenja

– *alarmTarget* - ime korisnika za koga je namijenjen alarm



Slika 3.2: *Slanje POST zahtjeva na endpoint 'create'*

Nakon što metoda parsira podatke, koristeći klasu ObjectMapper i Jdk8Module modul, potraži ako već postoji alarm sa istim alarmId, pomoću metode readDataFromFile iz klase AlarmFileReader. Ako postoji vraća HTTP odgovor *BAD REQUEST*, sa porukom "*Alarm with this alarmId already exist.*" koja daje doznanja korisniku kako već postoji alarm s odabranim alarmId. U suprotnom podaci o alarmu se dodaju u datoteku alarms.txt te se vraća HTTP odgovor *OK*. Podaci se zapisuju u sljedećem redoslijedu: *alarmId, deviceId, dataRequest, alarmTarger, alarmMessage, triggerValue*.

Metoda updateAlarm radi na sličan način kao metoda addData. Na jednak način pronađe alarm s traženim alarmId, te ako takav alarm ne postoji vraća status *NOT FOUND*. U suprotnom briše postojeće podatke o alarmu te ih mijenja sa novo unesenima. Kako bi ažurirali podatke o alarmu, moramo opet poslati sve podatke, ne samo one koje želimo izmijeniti.

Metoda deleteAlarm pronalazi alarm koji želimo izbrisati po alarmId u datoteci alarms.txt, na jednak način kao što je prikazan u metodi addData na slici 3.3. Ako

```java
@JsonCreator
@PostMapping(value="/create")
public ResponseEntity<String> addData(@RequestBody String jsonData) {

    ObjectMapper mapper = registerJdkModuleAndGetMapper();
    Data data;
    try {
        data = mapper.readValue(jsonData, Data.class);
    } catch (JsonProcessingException e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Invalid input data");
    }

    AlarmFileReader alarmFileReader = new AlarmFileReader();
    List<String> alarmEntries = alarmFileReader.readDataFromFile(FILE_PATH);

    Optional<String> alarmEntry = alarmEntries.stream()
            .filter(entry -> entry.startsWith(data.getAlarmId()))
            .findFirst();

    if(alarmEntry.isPresent()){
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Alarm with this alarmId already exists");
    }
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_PATH,  append: true))) {
        String formattedJson = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(data);
        writer.write(String.format("###\n%s\n%s\n---\n%s\n---\n%s\n%s\n%s\n", data.getAlarmId(), data.getDeviceId(),
                data.getDataRequest().getPayload(), data.getAlarmTarget(), data.getAlarmMessage(), data.getTriggerValue()));
        return ResponseEntity.ok( body: "Data added successfully");
    } catch (IOException e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Failed to add data");
    }
}
```

Slika 3.3: *Metoda addData*

alarm ne postoji, vraća se status *NOT FOUND*, a ako postoji ga se briše iz liste pročitanih alarma, *alarmEntries*, te se takva lista ponovo upisuje u datoteku.

3.4.3. Implementacija funkcionalnosti slanja alarma

Implementaciju slanja alarma klijentu ostvarujemo sa korištenjem Spring WebSocket modula. WebSockets je protokol komunikacije koji omogućuje komunikaciju između klijenta i poslužitelja putem jedne TCP veze. Kada je Websocket veza uspostavljena, omogućeno nam slanje različitih poruka na različite kanale, na koje se klijent može pretplatiti da prima poruke.

Modul Spring WebSocket omogućuje jednostavno definiranje *WebSocket endopointa*, rukovanje porukama, podršku za različite transportne slojeve te integraciju s drugim Spring komponentama poput kontrolera i servisa. U našoj Spring Boot aplikaciji potrebno je prvo uključiti navedeni modul na način da dodamo u datoteku pom.xml:

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

Nakon toga moramo napraviti klasu *WebSocketConfig* koja nam služi za postavljanje

endpointa na kojem će se *WebSocket* veza uspostavljati, definiranje rukovatelja porukama koji će se koristiti za obradu dolaznih i odlaznih poruka. Možemo vidjeti kako ova klasa izgleda na slici 3.4. Zbog korištenja Keycloak autentifikacije, potrebno je i

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    no usages  josipajovic
    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker( ...destinationPrefixes: "/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }

    no usages  josipajovic
    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint( ...paths: "/ws").setAllowedOriginPatterns("*").withSockJS();
    }
}
```

Slika 3.4: *WebSocketConfig.java*

dodati konfiguraciju za sigurnost WebSockets na način prikazan na slici 3.5.

```
 josipajovic
@Configuration
public class WebSocketSecurityConfig extends AbstractSecurityWebSocketMessageBrokerConfigurer {

    no usages  josipajovic
    @Override
    protected boolean sameOriginDisabled() { return true; }
}
```

Slika 3.5: *WebSocketSecurityConfig.java*

Nakon postavljanja ovih konfiguracija možemo pobliže objasniti kako klasa *PushNotificationService* implementira slanje alarma. Prvi korak je čitanje svih alarma iz datoteke u kojoj su zapisani pomoću servisa AlarmFileReader. Svaki alarm zapisan u datoteci sadrži sljedeće podatke: vlastiti id, id uređaja, *payload* koji se dalje šalje kao upit na bazu, korisničko ime korisnika kojem je alarm namijenjen, poruka koja se šalje alarmom te minimalna vrijednost na senzoru koja će potaknuti slanje alarma tzv. *triggerValue*. Pomoću POST metode, *payload* alarma tj. upit se šalje na bazu s očitanjima senzora. Kada dobijemo odgovor iz baze, prvo provjeravamo ima li uopće

kakvih očitanja, u tom slučaju šaljemo poruku da obavijestimo korisnika da nema nedavnih očitanja. Ako smo dobili očekivani odgovor od baze, tj. očitanja, parsiramo ih kako bi pročitali najnovije. Provjeravamo je li pročitano stanje manje od *triggerValue*, ako je šaljemo PushNotification pomoću *SimpMessagingTemplate* metode *convertAndSend*. Metoda šalje obavijest na *WebSocket endpoint* '/topic/alarmId', gdje je *alarmId* jedinstven za svaki alarm.

3.5. Klijentska strana

U ovom poglavlju ćemo se posvetiti klijentskoj strani aplikacije koja je razvijena pomoću React.js biblioteke i alata poput Node.js i npm. React.js je popularan JavaScript okvir koji se koristi za izgradnju modernih korisničkih sučelja. Node.js je serverska platforma koja omogućuje izvršavanje JavaScript koda izvan web preglednika, dok je npm (Node Package Manager) alat za upravljanje paketima i ovisnostima projekta.

3.5.1. Arhitektura React.js aplikacije

U izgradnji aplikacije, koristimo komponentnu arhitekturu što znači da imamo direktorije *components* i *pages* u kojima se nalaze komponente poput lista, navigacijske trake, odnosno specifične rute ili stranice u aplikaciji. Korištenje komponentne arhitekture pruža organiziran pristup izgradnji aplikacija. Razdvajanje funkcionalnosti u komponente olakšava održavanje koda, ponovnu upotrebu, testiranje i poboljšava čitljivost i razumljivost aplikacije.

U direktoriiju *pages* nalazimo *SecuredPage*, stranica koja u sebi poziva komponentu *DevicesList* koja prikazuje alarme za prijavljenog korisnika.

U direktoriju *components* možemo naći sljedeće komponente: *DeviceForm, DevicesList, Nav, PrivateRoute*. Komponenta *PrivateRoute* osigurava da samo autentificiran korisinik može vidjeti sadržaj *SecuredPage*, u suprotnom vraća sljedeći natpis: *"Login to see alarms!"*. Komponenta *Nav* predstavlja navigacijsku traku koji se prikazuje na vrhu aplikacije. Na njoj se nalazi gumb za prijavu korisnika, te ako je korisnik ulogiran, njegovu odjavu. Komponente *DeviceForm* i *DevicesList* sadrže glavni dio naše aplikacije, a njihovim opisom ćemo se baviti u potpoglavlju 3.5.2.

U korijenskom direktoriju imamo dvije glavne komponente aplikacije *index.js* i *App.js*. Komponenta *index.js* poziva funkciju za registriranje *service workera* i učitava glavnu komponentu aplikacije *App.js* što možemo vidjeti na slici 3.6. Ova datoteka služi kao ulazna točka (*entry point*) za aplikaciju.

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
serviceWorkerRegistration.register();
```

Slika 3.6: *Index.js*

Komponenta *App.js* je glavna komponetna aplikacije, obično djeluje kao kontejner koji sadrži i upravlja ostalim komponentama koje čine aplikaciju. To je mjesto gdje se definira struktura i funkcionalnost aplikacije, što se da uočiti na slici 3.7. Za autentifikaciju koristimo *ReactKeycloakProvider* iz biblioteke *@react-keycloak/web*. Za postavljanje ruta koristimo *BrowserRouter, Route, Routes* iz biblioteke *react-router-dom*.

```
function App() {
 return (
   <div>
     <ReactKeycloakProvider authClient={keycloak}>
      <Nav />
      <BrowserRouter>
        <Routes>
          <Route exact path="/" element={<PrivateRoute>
                  <SecuredPage />
                </PrivateRoute>} />
        </Routes>
      </BrowserRouter>
      </ReactKeycloakProvider>
   </div>
 );
}
export default App;
```

Slika 3.7: *App.js*

3.5.2. Implementacija *push* obavijesti u Reactu

Implementacija primanja *push* obavijesti se nalazi unutar dvije komponente *DeviceForm* i *DevicesList*. Koristimo biblioteke *stompjs, sockjs-client* i *react-push-notification* za povezivanje na *WebSocket* kanal i prikazivanje potisnih obavijesti.

DevicesList šalje POST zahtjev na poslužitelj, kojim dobiva nazad listu alarma za trenutno prijavljenog korisnika. Potrebno je poslati *Authorization Header* koji sadrži *Bearer Token* trenutnog korisnika kako postigli pristup podacima. Svaki alarm na listi se zatim prikazuje pomoću komponente *DeviceForm*. *DeviceForm* je komponenta koja prikazuje alarm i omogućuje korisniku da uključi i isključi alarm. U ovoj komponenti koristimo *useState* i *useEffect* kako bismo pratili stanje provjere alarma za uređaj i uspostavili *WebSocket* vezu za primanje obavijesti. Na početku, koristimo *useState*

```
useEffect(() => {
    fetchDevices();
  }, []);
const fetchDevices = async () => {
    try {
      const response = await fetch("http://localhost:8080/alarms",{
        method: 'POST',
        headers: {
          'Authorization': 'Bearer ' + accessToken,
        }
      });
      const data = await response.json();
      const devices = [];
      for (const key in data) {
        const device = {
          alarmId: key,
          ...data[key]
        };

        devices.push(device);
      }
      setDevices(devices);
    } catch (error) {
      console.error('Error fetching devices:', error);
    }
  };
```

Slika 3.8: *Isječak iz DevicesList.js*

```
useEffect(() => {
    let stompClient = null;
    if (localStorage.getItem('alarm') === 'true') {

      const socket = new SockJS("http://localhost:8080/ws", {
        headers: {
          'Authorization': 'Bearer ' + accessToken,
        },});
      stompClient = Stomp.over(socket);
      stompClient.connect({}, async () => {
        console.log("CONNECTED");
          stompClient.subscribe("/topic/" + props.alarmId, (notification) => {
            const pushNotification = JSON.parse(notification.body);
      addNotification({
        title: pushNotification.title,
        message: pushNotification.message,
        native:true,
      });
    });

      });}
    return () => {
      if (stompClient) {
        stompClient.unsubscribe("/topic/" + props.alarmId);
        stompClient.disconnect();
        console.log("disconnected")
      }
    };
  }, [checked]);
```

Slika 3.9: *Isječak iz DeviceForm.js*

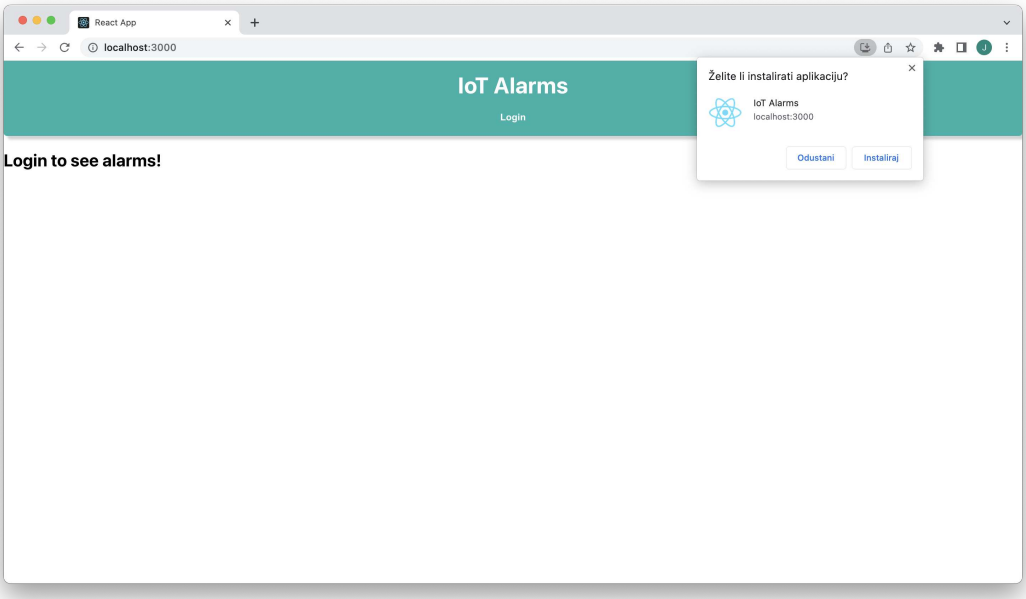
kako bi inicijalizirali stanje alarma, je li uključen ili nije. Vrijednost se čita iz lokalne memorije pomoću *localStorage.getItem* funkcije i postavlja se kao početna vrijednost za *checked* stanje. Također koristimo *useKeycloak* kako bi dohvatili informacijame o autentifikaciji, odnosno *access token*. Nakon toga se klijent uspostavlja *WebSocket* vezu koristeći *SockJs*. Prilikom uspostavljanja veze, pretplaćujemo se na željeni alarm ("/topic/" + props.alarmId) kako bismo primili obavijesti za taj određeni alarm. Kada primimo obavijest od poslužitelja prikazujemo je u klijentu funkcijom *addNotification*.

4. Primjer korištenja aplikacije

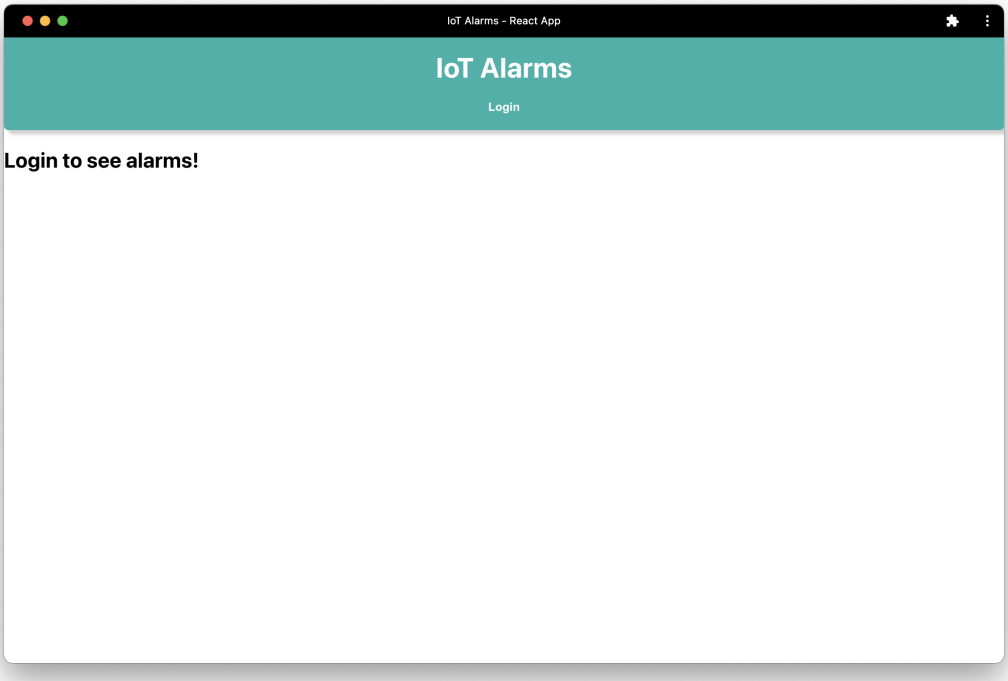
Sljedeće slike prikazuju primjer korištenja aplikacije za alarmiranje. Na slici 4.1 vidimo početnu stranicu kada prvi put pristupimo aplikaciji putem preglednika. Pojavi se prozor koji nas navodi da instaliramo aplikaciju lokalno na uređaj.

Na slici 4.2 prikazana je aplikacija koju smo instalirali. Nismo prijavljeni, pa imamo podsjetnik da se potrebno prijaviti kako bi mogli vidjeti alarme. Klikom na gumb *Login* preusmjereni smo na Keycloak login stranicu, što možemo vidjeti na slici 4.3. Nakon prijavljivanja ispod naslova *IoT Alarms* imamo opciju odjavljivanja, te je isto tako ispisano ime korisnika koji je trenutno prijavljen. Na slici 4.4 vidimo i ispisane sve alarme koji su stvoreni za trenutnog korisnika.

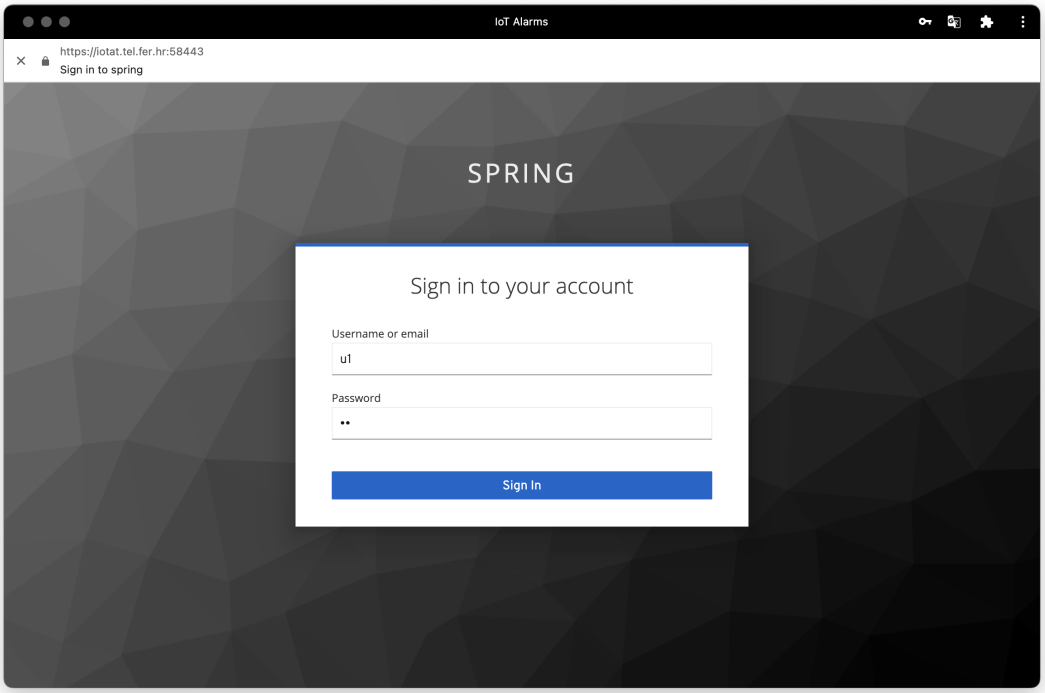
Alarme za koje želimo primati obavijesti označimo, te primjer pristizanja obavijesti je prikazano na slici 4.5. Obavijest ima naslov koji sadrži informaciji o trenutnoj temperaturi i poruku koja je odabrana prilikom stvaranja samog alarma.



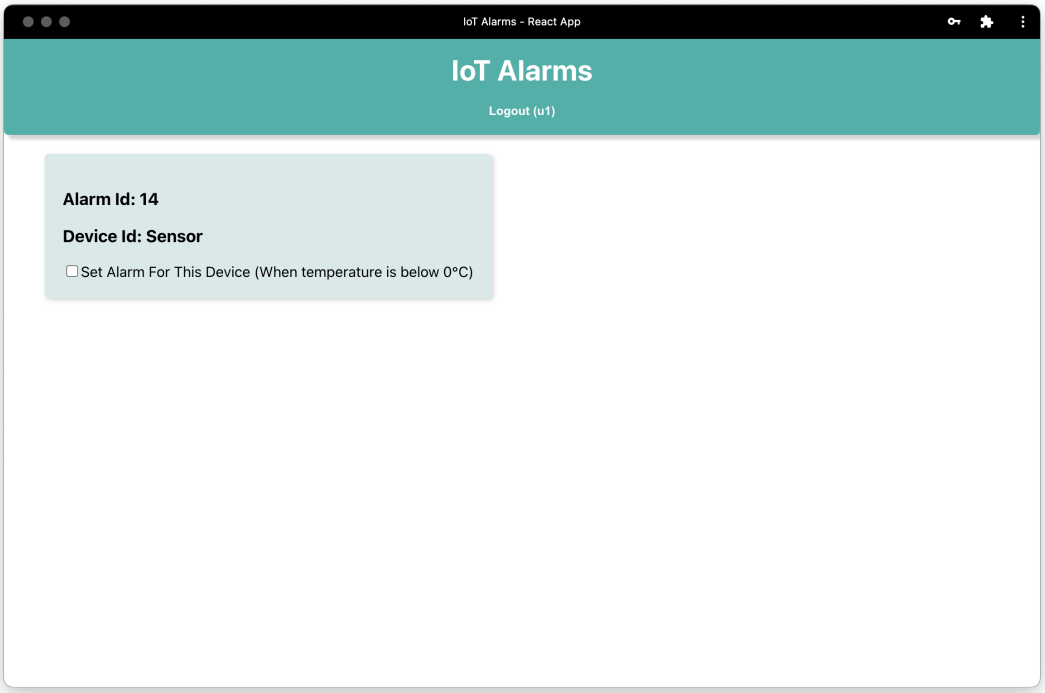
Slika 4.1: *Instaliranje PWA na uređaj*



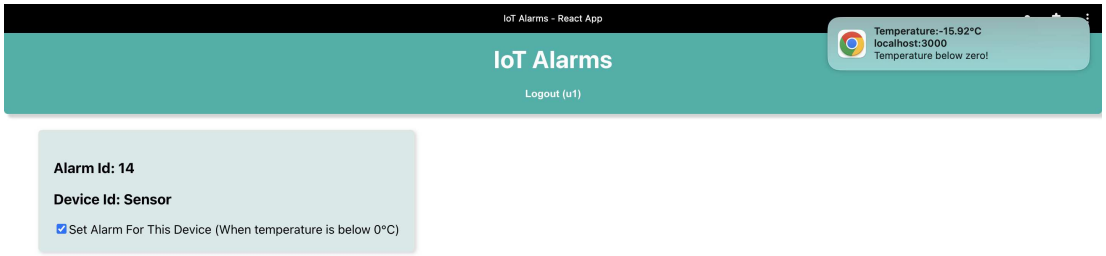
Slika 4.2: *Početna stranica*



Slika 4.3: *Keycloak login*



Slika 4.4: *Dohvaćena lista alarma za korisnika*



Slika 4.5: *Primjer potisne obavijesti*

5. Zaključak

Progresivne web-aplikacije su zaista najinovativniji način izrade aplikacija za web. Njihova mogućnost prilagodbe svakom uređaju i platformi im daje prednost nad aplikacijama koje su specifične za određenu platformu, tzv. *native* aplikacije. Kao PWA, ova aplikacija omogućuje korisnicima instalaciju na mobilnim uređajima i desktop računalima te pristupanje funkcionalnostima bez internetske veze. To pruža korisnicima fleksibilnost i pristupačnost u korištenju aplikacije u različitim uvjetima.

Jedna od ključnih prednosti PWA aplikacija je njihova prilagodljivost svim vrstama uređaja i platformi. Koristeći responzivni dizajn i tehnologije poput React.js-a, korisničko sučelje se prilagođava veličini zaslona i mogućnostima uređaja na kojem se koristi. To omogućuje korisnicima konzistentno iskustvo bez obzira na kojem ju uređaju koriste.

PWA aplikacija za alarmiranje omogućuje korisniku dobivanje obavijesti na temelju očitanja podataka. Korištenjem tehnologija poput WebSockets-a, SockJS-a i Stomp-a, omogućeno je uspostavljanje komunikacije u stvarnom vremenu između klijenta i poslužitelja, što omogućuje brzo i pouzdano primanje obavijesti. Ova komunikacija se ostvaruje preko WebSocket veze, koja se aktivira kada korisnik uključi određeni alarm putem korisničkog sučelja na PWA.

Zaključno, ova Progresivna Web-aplikacija nudi korisnicima izuzetnu fleksibilnost, prilagodljivost i mogućnost komuniciranja s poslužiteljem. Kombinacija web tehnologija poput React.js-a i naprednih funkcionalnosti PWA-a omogućuje izgradnju modernih i efikasnih aplikacija koje korisnicima pružaju bogato iskustvo bez obzira na uređaj ili platformu koju koriste. Ovaj inovativan pristup razvoju aplikacija otvara nove mogućnosti za alarmiranje i druge scenarije upotrebe, nudeći korisnicima intuitivno sučelje i pouzdane funkcionalnosti.

LITERATURA

[1] Cross-origin resource sharing (cors). URL https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[2] Push notifications. URL https://vwo.com/push-notifications/.

[3] Progressive web applications. URL https://vaadin.com/pwa.

[4] What are progressive web apps?, 2020. URL https://web.dev/what-are-pwas/.

[5] PortSwigger. Cross-site request forgery (csrf). URL https://portswigger.net/web-security/csrf.

[6] Sheriff Quadri. Push notifications. 2021. URL https://blog.logrocket.com/implement-keycloak-authentication-react/.

[7] Muriel Santoni. Progressive web apps browser support and compatibility. Kolovoz 2021. URL https://www.goodbarber.com/blog/progressive-web-apps-browser-support-compatibility-a883/.

[8] Wikipedia. Influxdb, . URL https://en.wikipedia.org/wiki/InfluxDB.

[9] Wikipedia. Progressive web app. . URL https://en.wikipedia.org/wiki/Progressive_web_app.

Progresivna web-aplikacija za alarmiranje u slučaju mraza

Sažetak

Ovaj rad se bavi temom razvijanja progresivne web-aplikacije za alarmiranje u slučaju mraza. U radu će se definirati progresivne web-aplikacije, opisati njene značajke i načine njihove izrade. Opisat će se implementacija alarma koristeći *push* obavijesti koje dolaze na progresivnu web-aplikaciju. Izrada aplikacije se događa na IoT platformi, podaci se dobivaju iz InfluxDB baze podataka. Tehnologije koje opisujemo i koristimo u izradi aplikacije su Spring Boot te React.js.

Ključne riječi: progresivna web-aplikacija, pwa, alarmi, push obavijesti

A progressive web application for frost alarm

Abstract

This thesis deals with the development of a progressive web application alarming in case of frost. The thesis will define progressive web applications, describe their features, and methods of development. It will describe the implementation of an alarm using push notifications that are delivered to the progressive web application. The application is built on an IoT platform, and the data is obtained from an InfluxDB database. The technologies described and used in the application development are Spring Boot and React.js.

Keywords: pwa, progressive web app, alarms, push notifications