Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Jan Alojz Gačnik

Brezkontaktni sistem za spremljanje delovnih ur s pomočjo platforme Raspberry Pi

Diplomsko delo

Maribor, julij 2022

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Jan Alojz Gačnik

Brezkontaktni sistem za spremljanje delovnih ur s pomočjo platforme Raspberry Pi

Diplomsko delo

Maribor, julij 2022

Brezkontaktni sistem za spremljanje delovnih ur s pomočjo platforme Raspberry Pi

Diplomsko delo

Študent:	Jan Alojz Gačnik
Študijski program:	Visokošolski strokovni študijski program
	Računalništvo in informacijske tehnologije
Mentor:	izr. prof. dr. Matej Črepinšek, univ. dipl. inž. rač. in inf.
Lektorica:	dr. Alenka Čuš, univ. dipl. slov.

Zahvala

Zahvaljujem se mentorju, izr. prof. dr. Mateju Črepinšku, za vso pomoč pri izdelavi diplomske naloge.

Prav tako se zahvaljujem staršem in dekletu, ki so mi med študijem vedno stali ob strani.

Brezkontaktni sistem za spremljanje delovnih ur s pomočjo platforme Raspberry Pi

Ključne besede: Raspberry Pi, Spring Boot, Android, Node-RED

UDK: 519.256:004.43(043.2)

Povzetek

Namen diplomskega dela je izdelava sistema, s katerim bi lahko brezkontaktno evidentirali delovne ure. To smo storili s pomočjo računalnika Raspberry Pi, na katerem smo z orodjem Node-RED preverjali prisotnost posameznika preko njegovih delovnih postaj in pametnih naprav. Za shranjevanje podatkov uporabimo dokumentno podatkovno bazo MongoDB, za prikaz informacij pa razvijemo Android aplikacijo. Vse to povezujemo s spletno aplikacijo Spring Boot, katera zbrane podatke tudi obdeluje, nato pa razvito rešitev primerjamo z že obstoječimi načini evidentiranja delovnih ur.

Contactless system for monitoring work-hours using the Raspberry Pi platform

Keywords: Raspberry Pi, Spring Boot, Android, Node-RED

UDC: 519.256:004.43(043.2)

Abstract

The purpose of this dipoma thesis is to create a system with whom we could monitor work hours without direct contact. For this matter we use a Raspberry Pi board with the Node-RED tool installed, on which we can examine the presence of an individual via their workstation and smart devices. For data storage we use the document-oriented database MongoDB and develop an Android app for displaying the collected data. We connect everything with a Spring Boot web application, which also process the collected data. After that we compare our solution to already existing ones.

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Koroška cesta 46
2000 Maribor, Slovenija

FERI

IZJAVA O AVTORSTVU ZAKLJUČNEGA DELA

Ime in priimek študent-a/-ke: Jan Alojz Gačnik
Študijski program: Visokošolski strokovni študijski program Računalništvo in informacijske tehnologije
Naslov zaključnega dela: Brezkontaktni sistem za spremljanje delovnih ur s pomočjo platforme Raspberry Pi
Mentor: izr. prof. dr. Matej Črepinšek, univ. dipl. inž. rač. in inf.
Somentor: _____

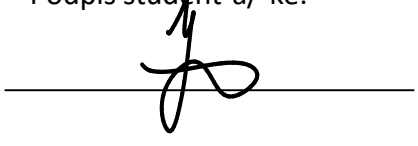
Podpisan-i/-a študent/-ka Jan Alojz Gačnik_____

- izjavljam, da je zaključno delo rezultat mojega samostojnega dela, ki sem ga izdelal/-a ob pomoči mentor-ja/-ice oz. somentor-ja/-ice;
- izjavljam, da sem pridobil/-a vsa potrebna soglasja za uporabo podatkov in avtorskih del v zaključnem delu in jih v zaključnem delu jasno in ustrezno označil/-a;
- na Univerzo v Mariboru neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico ponuditi zaključno delo javnosti na svetovnem spletu preko DKUM; sem seznanjen/-a, da bodo dela deponirana/objavljena v DKUM dostopna široki javnosti pod pogoji licence Creative Commons BY-NC-ND, kar vključuje tudi avtomatizirano indeksiranje preko spleta in obdelavo besedil za potrebe tekstovnega in podatkovnega rudarjenja in ekstrakcije znanja iz vsebin; uporabnikom se dovoli reproduciranje brez predelave avtorskega dela, distribuiranje, dajanje v najem in priobčitev javnosti samega izvirnega avtorskega dela, in sicer pod pogojem, da navedejo avtorja in da ne gre za komercialno uporabo;
- dovoljujem objavo svojih osebnih podatkov, ki so navedeni v zaključnem delu in tej izjavi, skupaj z objavo zaključnega dela.

Uveljavljam permisivnejšo obliko licence Creative Commons:_____(navedite obliko)

Kraj in datum: Maribor, 22. 7. 2022

Podpis študent-a/-ke:



KAZALO VSEBINE

KAZALO SLIK

KAZALO IZREZKOV KODE

KAZALO TABEL

SEZNAM UPORABLJENIH KRATIC

XML – označevalni jezik in podatkovni tip za shranjevanje podatkov (angl. Extensible Markup Language)

CRUD – osnovne operacije pri delu s podatkovno bazo (angl. Create, Read, Update, and Delete)

JSON – notacija JavaScript objekta (angl. JavaScript Object Notation)

JWT – JSON spletni žeton (angl. JSON Web Token)

KSR – vzorec za ločevanje odvisnosti na tri nivoje (krmilnik-storitev-repozitorij)

HTTP – protokol za prenos informacij na spletu (angl. Hypertext Transfer Protocol)

HTTPS – varnejša različica protokola HTTP (angl. Hypertext Transfer Protocol Secure)

TLS – kriptografski protokol za varno komunikacijo na spletu (angl. Transport Layer Security)

1 UVOD

Dandanes se na različnih delovnih mestih še vedno uporabljajo zastarele oblike spremljanja delovnih ur. To ponekod še delajo z ročnim zapisovanjem na liste papirja ali v računalniške tabele, kar pa lahko postane nepregledno ali nenatančno zaradi možnosti napak pri obračunanju delovnega časa. V te namene že obstajajo različne moderne rešitve, ki izboljšajo ta proces, a smo želeli ustvariti svojo rešitev in jo primerjati z drugimi.

Rešitev smo ustvarili z namenom primerjave lastne rešitve z obstoječimi. Preveriti nameravamo, kako natančne so te rešitve, koliko so uporabnikom prijazne s strani delavca in delodajalca ter stroške le-teh. S tem bomo preverili, ali je smiselen razvoj lastne rešitve, ki je posledično veliko bolj prilagodljiva specifičnim potrebam.

Pri razvoju bomo uporabljali različne tehnologije, kot so platforma Android [1], Spring Boot [2], Node-RED [3], dokumentna podatkovna baza MongoDB [4] in računalnik Raspberry Pi 4 [5]. Vse te bomo tudi podrobneje preučili. Preverili bomo drugi dve rešitvi, kot sta AllHours [6] in Clockify [7], ki se uporabljata v večjih organizacijah. Obravnavali bomo tudi vse dele našega sistema in njihove funkcionalnosti ter na koncu opravili primerjavo.

2 UPORABLJENE TEHNOLOGIJE IN ORODJA

Za realizacijo naše podane rešitve bomo uporabili različne tehnologije in orodja. Uporabljali bomo razne tehnologije in knjižnice za izdelavo čelnega in zalednega dela sistema ter razna orodja za razvoj in testiranje rešitev.

V naslednjih podpoglavjih bomo vse uporabljene tehnologije opisali in predstavili.

2.1 Platforma Android

Platforma Android se uporablja za razvoj aplikacij za različne naprave. Prvotno smo lahko razvijali z njo aplikacije za mobilne telefone Android, danes pa se lahko uporablja ne le za razvoj aplikacij za mobilnike, ampak tudi za pametne ure, televizorje ter pametne naprave za avtomobile. Prednost platforme Android je tudi ta, da je odprtokodni projekt, kar pomeni, da lahko izdelamo svoje distribucije operacijskega sistema. Njihovo maskoto lahko vidimo na Slika 2.1.

Slika 2.1 Android Robot grafika

Za platformo Android lahko razvijemo nativno ali hibridno aplikacijo. Za razvoj nativne aplikacije se uporablja razvojno orodje Android Studio [8], ki jo je razvilo podjetje Google in temelji na orodju IntelliJ IDEA [9] podjetja JetBrains'. Hibridni razvoj nam omogoča, da aplikacijo napišemo le enkrat in jo lahko uporabimo na različnih platformah. V te namene lahko uporabljamo različne tehnologije, kot so Flutter in Ionic. V primerjavi s hibridnimi aplikacijami so nativne bolje optimizirane in hitrejše. Slabost nativnih aplikacij je, da so stroški

in čas razvoja v primerjavi s hibridnimi daljši, če želimo razviti nativno aplikacijo za več različnih naprav (npr. iOS in Android). Ker je bil prvotni cilj diplomskega dela razviti le Android aplikacijo, smo se odločili, da bomo uporabili razvojno orodje Android Studio in ustvarjali v nativnem orodju.

2.1.1 Razvojno orodje Android Studio

Android Studio je odprtokodno orodje, s katerim lahko razvijamo nativne aplikacije za katerokoli različico operacijskega sistema Android. Kljub temu, da je rešitev odprtokodna, ima vgrajenih veliko funkcionalnosti, s katerimi nam olajša delo. Trenutna različica logotipa orodja (Slika 2.2) prav tako vsebuje maskoto Android operacijskega sistema.

Vgrajeno ima orodje za grafično oblikovanje, s katerim lahko direktno preko jezika XML manipuliramo izgled aplikacije ali pa to storimo preko grafičnega vmesnika, ki deluje na principu »povleci in spusti« (angl. drag and drop). Grafični vmesnik nam da na izbiro veliko elementov, ki jih lahko uporabimo pri oblikovanju, na primer polja za besedilo (brez in z možnostjo urejanja), elemente za prikaz videov, različne gumbe, elemente s storitvami Google (Maps, Adsense ipd.) ter mnoge druge.

Slika 2.2 Logotip razvojnega orodja Android Studio

Druge funkcionalnosti, ki jih orodje vsebuje, so razhroščevalnik, orodje za verzioniranje, Android emulator [10] in monitor aktivnosti naprav, s katerim lahko preverjamo porabo predpomnilnika ter procesorja. Vgrajen emulator je eden izmed najbolj uporabnih funkcionalnosti znotraj orodja. Z njim lahko emuliramo večino različic Android na virtualnih mobilnih napravah različnih velikosti, s tem pa nam tudi zmanjša stroške razvoja, saj nam ni potreben nakup različnih pametnih telefonov.

Razvoj lahko poteka v dveh programskih jezikih, v jeziku Java ali Kotlin [11]. Prvotno je bila Java edini in primarni jezik za razvoj, kasneje pa se mu je pridružil Kotlin, katerega je podjetje Google leta 2019 uradno razglasil kot uradni jezik za razvoj Android aplikacij.

2.1.2 Programski jezik Kotlin

Za izdelavo naše mobilne Android aplikacije smo izbrali odprtokodni programski jezik Kotlin. Tako smo se odločili zato, ker ima Kotlin številne prednosti pred jezikom Java in je teoretično za razvoj aplikacij hitrejši.

Slika 2.2.3 Logotip programskega jezika Kotlin

Precejšnja prednost Kotlin-a je količina kode, ki jo potrebujemo, da naredimo določeno stvar v primerjavi z Javo, saj se Kotlin izogne nepotrebnim vrsticam. Glede na uradno spletno stran jezika naj bi se število vrstic kode zmanjšalo za približno 40 odstotkov.

Ker je velik delež knjižnic in ogrodij v industriji namenjen razvoju z Javo, se je podjetje JetBrains odločilo, da bo njihov nov programski jezik popolno interoperabilen (angl. interoperable), kar pomeni, da je sposoben delovati z Javo in nam je s tem omogočeno v naše aplikacije kljub uporabi drugega jezika vključevanje Java ogrodja in knjižnice.

Kotlin ima tudi večjo varnost pred Null izjemami (angl. nullpointer exception), saj prevzeto spremenljivke ne morejo biti vrednosti »null«. Če bi želeli, da spremenljivka lahko ima takšno vrednost, moramo to posebej deklarirati s končnico »?«.

Kodo napisano v Kotlinu lahko trenutno prevajamo na tri načine. Prevede se lahko v Java bajtkodo za razvoj zalednih aplikacij, v ES5.1 za razvoj spletnih aplikacij ali pa v kodo za specifično platformo z uporabo nativnega prevajanja. Z uporabo nativnega načina lahko razvijamo za vgrajene sisteme, macOS in iOS, upoštevati pa moramo, da je ta način trenutno še v razvoju.

2.1.3 Knjižnica MPAndroidChart

V našem projektu smo nameravali grafično prikazati podatke o delovnem času, in sicer s pomočjo grafov. Da bi lahko vključili grafe v našo aplikacijo, smo morali uporabiti knjižnico, ki to omogoča, saj Android ne vsebuje elementov oz. funkcij za generiranje in prikazovanje grafov. S tem razlogom smo se odločili za javansko odprtokodno knjižnico MPAndroidChart [12] Philippa Jahoda. Knjižnica nam omogoča generiranje črtnih, tortnih, paličnih in mnogih drugih vrst grafov. Vsi grafi so zelo prilagodljivi, saj lahko spreminjamo velikosti, barve, orientacijo ter oznake. Grafom lahko prav tako dodajamo animacije in jih v realnem času posodabljamo.

Slika 2.4 Logotip knjižnice MPAndroidChart

2.1.4 Knjižnica Volley

Knjižnica Volley [13] je knjižnica HTTP, ki nam olajša delo z mrežnimi klici. Bila je razvita s strani podjetja Google in prvič predstavljena leta 2013.

Volley ponuja med drugim:

- avtomatsko vodenje zahtev HTTP,
- sočasno povezavo z več mrežnimi povezavami,
- podporo za izvršitev klicev po prioriteti,
- preklic in blokiranje posameznih zahtev,
- podporo za slike, nize in podatke JSON.

Ker bomo večino podatkov prenašali v formatu JSON in ker naši podatki ne bodo zavzemali veliko spomina, je ta za našo aplikacijo zelo priročna. Opozoriti je potrebno, da Volley ni primeren za prenos večjih podatkovnih paketov in operacij, kot na primer pretok video vsebin, saj vse odgovore shranjuje v spominu.

2.2 Ogrodje Spring in orodje Spring Boot

Za zalednji del, ki se bo uporabljal za obdelavo podatkov in komunikacijo med podatkovno bazo in aplikacijo, smo se odločili za ogrodje Spring [14] (glej Slika 2.5) in orodje Spring Boot (glej Slika 2.6).

Slika 2.5 Logotip ogrodja Java Spring

Java Spring je odprtokodno ogrodje, s katerim lahko razvijamo spletne aplikacije, ki delajo na Javanskem virtualnem stroju (JVM). Ogrodje je precej priljubljeno, saj omogoča vrivanje odvisnosti (angl. dependency injections), kar omogoča, da objekti definirajo svoje odvisnosti, katere nato ogrodje »vrine« v njih. Posledično so aplikacije, ki so napisane s tem ogrodjem modularne, kar je idealno za porazdeljene sisteme in mikro storitve (angl. microservices). Spring hkrati ponuja prevajanje podatkovnih tipov, validacijo, ravnanje z izjemami, opravljanje virov in dogodkov ter mnogo več. Ogrodje je kot večina Javanskih ogrodij po izidu Java različice 5.0 začela podpirati anotacije. Z njimi lahko definiramo zahtevane spremenljivke, določimo konfiguracijske razrede, razrede za direktno komunikacijo s podatkovno bazo (repozitorij), storitvene razrede, razne anotacije za »vrivanje« odvisnosti in več. Ker ogrodje temelji na programskem jeziku Java in raznih Java EE tehnologijah, so aplikacije, ustvarjene s tem ogrodjem, multiplatformne. Veliko razvijalcev se odloči, da bo uporabilo orodje Spring Boot, saj je potrebno veliko časa in znanja, da nastavimo, konfiguriramo in namestimo Java Spring aplikacije.

Glavne tri prednosti orodja Spring Boot so:

- avtomatska konfiguracija – orodje avtomatsko konfigurira aplikacijo in vse dodane knjižnice glede na uporabniške nastavitve in najboljše prakse, seveda pa lahko tudi ročno spreminjamo nastavitve glede na naše potrebe,
- mnenjski pristop (angl. opinionated approach) – orodje samo predvidi in namesti po svojem mnenju najboljše pakete in katere prevzete vrednosti naj bi uporabili, naše potrebe pa določimo preko spletnega obrazca, kjer izbiramo med raznimi začetnimi paketi, imenovanimi »Spring starters«, kateri nam izbrane pakete združi v aplikacijo, ki jo lahko nato začnemo programirati,
- ustvarjanje samostojnih aplikacij, kar pomeni, da lahko s tem orodjem ustvarimo aplikacijo, ki bo delovala brez odvisnosti od drugih spletnih strežnikov.

Slika 2.6 Logotip orodja Spring Boot

Z uporabo tega orodja izgubimo nekoliko fleksibilnosti ogrodja Java Spring, a v večini primerov nas to pri razvoju aplikacij ne bi oviralo. Oviralo nas bi le, če bi želeli ustvariti zelo specifični izdelek, kjer bi bila ta fleksibilnost velikega pomena.

2.2.1 Razvojno orodje IntelliJ

Java Spring aplikacije lahko razvijamo v različnih razvojnih orodjih. Med najbolj priljubljene spadata brezplačna in odprtokodna rešitev Eclipse organizacije Eclipse Foundation in licenčno orodje IntelliJ podjetja JetBrains. Slednje je zelo podobno orodju Android Studio, ki temelji na njem. Da bi uporabljali čim manj različnih orodij pri razvoju, smo se odločili za orodje IntelliJ. Uporaba tega orodja je za osebe v izobraževalnih ustanovah brezplačna, saj podjetje JetBrains ponuja brezplačne licence za študente, ki jih lahko vsako leto brezplačno podaljšamo, dokler

se izobražujemo. Orodje IntelliJ vsebuje podobne funkcionalnosti kot orodje Android Studio, le da je prilagojeno za razvoj aplikacij v jeziku Java. Vsebuje verzioniranje preko različnih spletnih portalov, kot je Github, razhroščevalnik, trgovino, preko katere lahko inštaliramo dodatna orodja ter podporo za različna ogrodja, med njimi tudi za Java Spring Framework in Spring Boot.

2.2.2 Knjižnica Project Lombok

Lombok [15] je javanska knjižnica, ki nam pomaga pri pisanju aplikacij, saj si lahko z njeno pomočjo zmanjšamo količino kode. Knjižnica nam omogoča, da določene metode nadomestimo z anotacijami, katere knjižnica nato prepozna in pretvori v zahtevano Java bajtkodo.

Knjižnica ponuja mnogo anotacij, med njimi so zanimivejše:

- @Getter/@Setter – generira getter in setter funkcije za vse spremenljivke razreda,
- @NoArgsConstructor, @RequiredArgsConstructor in @AllArgsConstructor – generirajo konstruktor brez parametrov, konstruktor le z določenimi zahtevanimi parametri ali z vsemi parametri razreda,
- @Data – generira vse metode, potrebne za razrede, ki delujejo kot model (getter, setter metode, toString metodo, primerjalno in hash metodo ter konstruktor z zahtevanimi spremenljivkami),
- @Value – generira vse metode, potrebne za nespremenljivi razred (immutable class).

2.2.3 Spring Security

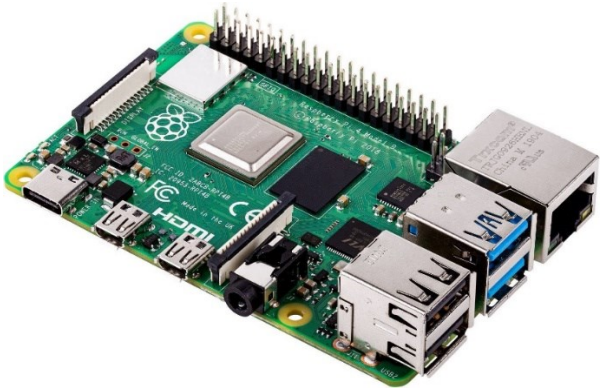
Ogrodje Spring Security [16] je standardno ogrodje za povečevanje varnosti za Java Spring aplikacije. Je zelo močno ogrodje, ki ga lahko prilagodimo glede na naše potrebe. Z njim lahko implementiramo v aplikaciji tako avtentikacijo kot avtorizacijo. Z implementacijo avtentikacije uporabnikov bomo znotraj aplikacije vedeli, kdo želi dostopiti do nje. Ko aplikacija ve, kdo dostopa do nje, lahko preveri, katere vloge oz. pravice uporabnik ima in s tem dovoli uporabniku dostop do delov aplikacije, za katere ima tudi avtorizacijo. Z ogrodjem lahko tako dovolimo, omejimo in preprečimo osebam dostop do naše aplikacije in povečamo predvsem varnost aplikacije in podatkov. V sklopu diplomske naloge s pomočjo ogrodja implementiramo žeton JWT in šifriranje z razredom BCrypt, ki je del ogrodja Spring Security.

2.3 Platforma Raspberry Pi 4

Raspberry Pi 4 Model B (glej Slika 2.8) je računalnik dobrodelne fundacije Raspberry Pi [17] (Slika 2.7) iz Velike Britanije. Njen cilj je, da omogoči čimveč osebam dostop do gradiv, s katerimi se lahko izobražujejo o računalništvu in razvoju programske opreme.

Slika 2.7 Logotip Raspberry Pi fundacije

S tem namenom ponujajo sponzorstva raznim organizacijam in tudi šolam. Gradiva, ki jih ponujajo, vključujejo učne načrte, vire in izobraževanja za učitelje. Najbolj znani so po računalniku Raspberry Pi, ki je v bistvu majhen, a zelo zmogljiv računalnik, ki deluje na Linux sistemu.



Slika 2.8 Razvojna ploščica Raspberry Pi 4 Model B

Računalnik ima že več revizij, najnovejša izmed njih je Raspberry Pi 4 Model B, njihov najnovejši izdelek pa je Raspberry Pi 400 (glej Slika 2.9), ki je pravzaprav tipkovnica z že vgrajenim računalnikom.

Slika 2.9 Raspberry Pi 400

Organizacija je prav tako razvila svoj operacijski sistem, imenovan Raspberry Pi OS (nekoč Raspbian). Temelji na operacijskem sistemu Linux Debian in je uradni operacijski sistem za Raspberry produkte, glede česa pa nas ne omejuje in lahko po želji uporabljamo tudi druge operacijske sisteme. V sklopu diplomske naloge smo uporabili računalnik Raspberry Pi 4 Model B s 4GB predpomnilnika (RAM) z operacijskim sistemom Raspberry Pi OS.

2.4 Orodje Node-RED

Node-RED je programsko orodje, ki deluje na okolju Node.js. Narejeno je tako, da je nezahtevno in je idealno za uporabo z nizkocenovno strojno opremo in za računalništvo v oblaku. Orodje se uporablja znotraj brskalnika, kjer v grafičnem urejevalniku z bloki sestavljamo toke (angl. Flows). Toke lahko primerjamo z nekakšnimi manjšimi programi, ki se izvršijo ob izvršitvi določenega dogodka, npr. klica HTTP. V toke lahko vključimo razne funkcije, pogoje, izhodne in vhodne operacije, klice HTTP in več. V orodje lahko dodajamo tudi dodatne bloke, ki jih napišemo sami ali naložimo iz Node.js paketnega repozitorija. Toki, ki jih ustvarimo, se shranjujejo v formatu JSON in jih lahko zato preprosto izvažamo in uvažamo v orodje Node-RED. V naši rešitvi smo uporabili dodaten modul Node-RED-contrib-arp [22], ki vsebuje blok »arp«, kateri preveri in vrne seznam naprav, povezanih na lokalno omrežje.

2.5 Dokumentna podatkovna baza MongoDB

MongoDB je dokumentna podatkovna baza, ki je vrsta NoSQL podatkovne baze. V takšnih bazah podatki niso shranjeni v tabelah, ki so med seboj povezane z relacijami, ampak v t. i. dokumentih. Ponavadi so podatki znotraj njih v formatu JSON, BSON ali XML in lahko zaradi tega v večini priljubljenih programskih jezikih dokumente brez dodatnih operacij prepišemo v objekte ter druge podatkovne strukture. Za razlikovanje med njimi vsakemu dokumentu dodelimo enolični identifikacijski niz. Vlogo tabel v dokumentnih bazah nadomestijo kolekcije, v katere shranjujemo dokumente, a to ne pomeni, da so vsi dokumenti iste strukture, temveč vsebujejo podobne podatke. Slednje hkrati omogoča veliko fleksibilnost, saj struktur ali shem ni potrebno točno določiti. Zaradi teh značilnosti so dokumentne podatkovne baze bolj intuitivne kot relacijske baze, kar tudi razloži njihovo priljubljenost.

V primeru MongoDB-ja so podatki formatirani v formatu JSON. Namenjen je razvoju internetnih in poslovnih rešitev, potrebe takšnih rešitev pa se lahko hitro spreminjajo, zato je še toliko bolj pomembno, da smo lahko fleksibilni s strukturo naših podatkov. Še ena izmed značilnosti MongoDB baz je tudi njihova hitrost operacij nad podatki, ne glede na velikost baze. Ker je MongoDB zelo fleksibilna in skalabilna podatkovna baza, je postala zelo priljubljena med razvijalci spletnih in drugih aplikacij.

2.6 Spletna platforma Github

Pri razvoju aplikacij je zelo pomembno, da našo kodo tudi verzioniramo. Slednje lahko storimo na različne načine, najpogosteje pa naredimo to s pomočjo različnih sistemov. Najbolj priljubljen med vsemi sistemi je sistem Git [18], s katerim lahko beležimo katerokoli spremembo v naši kodi s kratkim opisom. Vse te beležene spremembe lahko nato tudi primerjamo med seboj ali se vrnemo nanje v primeru napak v kodi. Te shranjujemo v t. i. repozitorije, kateri praviloma vsak vsebuje svojo aplikacijo. Obstajajo tudi spletni portali, kot so BitBucket in Github [19], na katerih lahko našo kodo shranjujemo in verzioniramo s pomočjo sistema Git.

V tem delu smo za verzioniranje izbrali portal Github in v te namene ustvarili več repozitorijev, v katerih smo beležili razvoj sistema za beleženje delovnih ur. Velika prednost Githuba je, da omogoča brezplačno ustvarjanje javnih in zasebnih repozitorijev, v veliko razvojnih orodjih pa ima tudi vgrajena orodja, s katerimi se lahko direktno povežemo na posamezne repozitorije iz portala Github.

2.7 Orodje Postman

Postman [20] je programsko orodje, ki nam omogoča kreiranje, testiranje, deljenje in dokumentiranje API klicev. Znotraj orodja lahko vsak klic posebej shranimo in poimenujemo s kratkim opisom. Ko si klic sestavimo, ga lahko preprosto izvršimo in se nam znotraj orodja izpiše odgovor strežnika oz. aplikacije. Prav tako lahko vsakemu klicu v glavo, telo ali v povezavo dodamo parametre ter avtentikacijo po različnih standardih. Če si ustvarimo svoj uporabniški račun, lahko svoje zbirke in klice še lažje prenašamo med napravami, saj se shranijo vsi klici na naš uporabniški račun. Osnovne funkcionalnosti orodja, kot je kreiranje in izvršitev klicev ter kreiranje zbirk, so povsem brezplačne; za dodatne funkcionalnosti, namenjene za večja podjetja, pa je potrebno plačevati mesečno naročnino.

3 ZASNOVA REŠITVE IN IMPLEMENTACIJA

Predvideli smo, da bo osnovna rešitev sestavljena iz treh različnih delov in podatkovne baze. Kot podatkovno bazo smo izbrali MongoDB, saj je namenjena za shranjevanje velikih količin podatkov in ima zelo fleksibilno strukturo podatkov. Sama rešitev je razdeljena na sprednji del (angl. frontend), katerega predstavlja Android aplikacija, zaledni del (angl. backend), izdelan v ogrodju Spring s pomočjo orodja Spring Boot in aplikacijo za preverjanje prisotnosti v orodju Node-RED na računalniku Raspberry Pi.

Vloga Android aplikacije je prikaz zabeleženih delovnih ur, upravljanje osebnih podatkov ter ročno beleženje časa, medtem ko bo zaledje skrbelo za prenos, obdelavo in shranjevanje uporabniških in izmerjenih podatkov. Aplikacija na računalniku bo skrbela za preverjanje prisotnosti naprav na omrežju, s katerimi bo možno samodejno beleženje oseb preko njihovih registriranih naprav.

3.1 Dokumentna podatkovna baza MongoDB

Podatkovno bazo MongoDB smo vzpostavili lokalno s pomočjo MongoDB Community strežnika. Da bi omogočili dostop do podatkovnih baz le določenim uporabnikom, smo v datoteki mongod.cfg dodali pod »security« vrstico »authorization: "enabled"«. Tako se lahko do podatkovne baze povežejo le uporabniki, ki so shranjeni v podatkovni bazi »admin«, katero MongoDB ustvari sam. Preden smo to omogočili, smo v podatkovno bazo dodali novega uporabnika, kar lahko storimo s pomočjo orodja MongoDBCompass ali preko orodne vrstice z ukazom »createUser«.

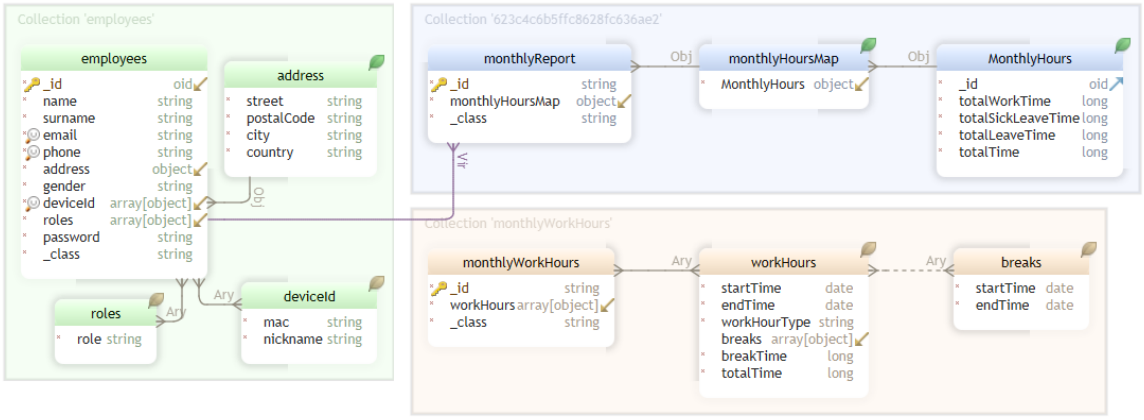
Za namene razvoja smo v našem primeru ustvarili preprostega uporabnika »admin« z geslom »admin« in prevzetimi vlogami oz. dovoljenji s pomočjo spodnjega ukaza.

```
1. db.createUser({ user: "admin",
2.                  pwd: passwordPrompt(),
3.                  roles: [ { role: "clusterAdmin", db: "admin" },
4.                           { role: "readAnyDatabase", db: "admin" },
5.                           "readWrite"] },
6.                { w: "majority" , wtimeout: 5000 })
```

Izrezek kode 3.1 Ukaz za dodajanje uporabnika

S tem ukazom ima uporabnik dostop do vseh podatkovnih baz znotraj našega strežnika ter dovoljenje do branja in pisanja. Da se naše geslo znotraj orodja ne prikaže, lahko uporabimo funkcijo passwordPrompt(). Funkcija nam po potrditvi ukaza prikaže polje, v katero vpišemo geslo in po potrditvi gesla iz orodne vrstice izbriše. Ko te nastavitve shranimo in strežnik znova zaženemo, se lahko do podatkovne baze povežejo le uporabniki, shranjeni v podatkovni bazi »admin«.

Za našo rešitev smo uporabili podatkovno shemo na Slika 3.1 Podatkovna struktura , ki smo jo generirali s pomočjo orodja DbSchema. Podatkovna baza je sestavljena iz treh dokumentnih zbirk. Imamo zbirko employees, v katero shranjujemo uporabnike oz. zaposlene, zbirko monthlyWorkHours, v katero shranjujemo podatke o delovnem času ter zbirko monthlyReport, v katero shranjujemo statistične podatke za vse zaposlene na mesečni ravni.

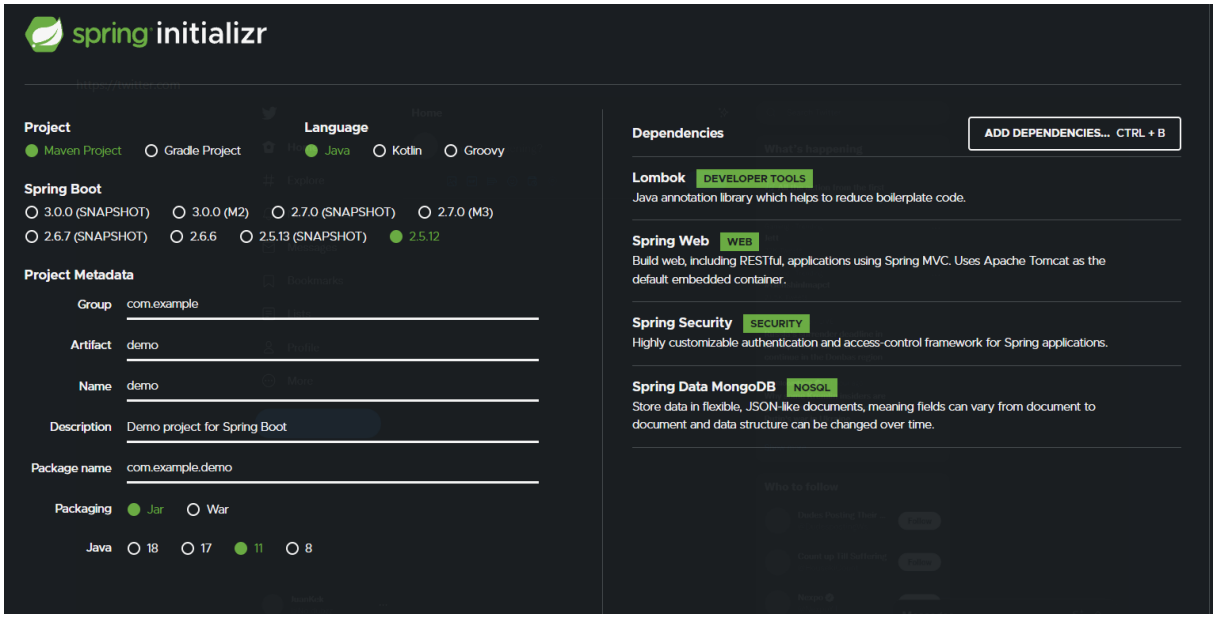


Slika 3.1 Podatkovna struktura implementirane rešitve

V zbirki employees so podatki o vsakem uporabniku shranjeni v svojem dokumentu. Ta vsebuje osnovne podatke, kot so ime, naslov in kontakte ter šifrirano geslo, registrirane naprave in vloge uporabnika. Zbirka monthlyHours vsebuje za vsakega uporabnika svoj dokument za vsak mesec posebej. Ta vsebuje polje delovnih ur, v katerih so shranjeni časi prihodov, odhodov ter odmorov, tip dela in skupno število časa dela in odmorov v minutah.
V zbirki monthlyHours se shranjujejo mesečni podatki o delovnem času. Tukaj se za vsak mesec generira nov dokument, ki vsebuje za vsakega zaposlenega število delovnih ur, bolniški stalež, dopust in skupno število vseh ur za ta mesec.

3.2 Razvoj spletne aplikacije z orodjem Spring Boot

Za zaledje smo razvili aplikacijo z orodjem Spring Boot. Pri tem smo se zgledovali po principu krmilnik-storitev-repozitorij in po tem principu ter s priporočenimi pristopi strukturirali našo aplikacijo.
Z razvojem spletne aplikacije začnemo na spletni strani https://start.spring.io/ (Slika 3.2 Spletna stran spring initializr), kjer si generiramo naš projekt. Na spletni strani lahko izberemo, ali bo aplikacija Maven oz. Gradle projekt, v katerem jeziku jo bomo pisali, določimo različne meta podatke ter katere odvisnosti želimo, da projekt vsebuje.



Slika 3.2 Spletna stran spring initializr

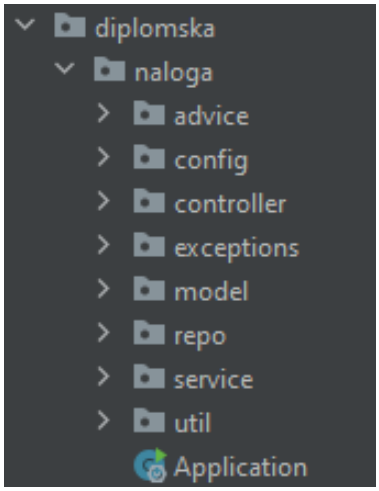
V našem primeru smo se odločili za naslednje odvisnosti:

- Lombok (dodatne anotacije za zmanjševanje količine kode),
- Spring Web (grajenje RESTful in spletnih aplikacij),
- Spring Security (prilagodljivo varnostno ogrodje za Spring aplikacije),
- Spring Data MongoDB (olajša delo z MongoDB podatkovno bazo).

Te odvisnosti lahko najdemo v pom.xml datoteki znotraj generiranega projekta. Možno je tudi dodati odvisnosti z dodajanjem le-teh v to datoteko. V našem primeru smo v projekt še dodali validation-api, s katerim lahko z anotacijami validiramo spremenljivke znotraj razredov, jjwt, s pomočjo katerega bomo implementirali žeton JWT in Log4j2 [21] za lažje beleženje.

3.2.1 Projektna struktura

Na Slika 3.3 Projektna struktura Spring Boot aplikacijevidimo strukturo našega zalednega projekta. Sestavljen je iz različnih podmap oziroma paketov. Pri razvoju smo uporabili princip krmilnik-storitev-repozitorij (controller-service-repository), s katerim lahko naredimo sam razvoj preglednejši in s tem tudi ločimo odvisnosti. Pri tem imamo tri nivoje, kjer je v najvišjem nivoju krmilnik, na najnižjem pa repozitorij, kar pomeni, da ne moremo dostopati do repozitorija direktno, ampak moramo prvo klicati krmilnik, ki je viden zunanjim entitetam. Na sliki vidimo tudi razred Application, ki je glavni razred v aplikaciji, s katerim le-to zaženemo. Nekoliko bolj podrobne opise vseh podmap oziroma paketov najdemo v Tabela 3.1 Opisi paketov znotraj Spring Boot aplikacije.



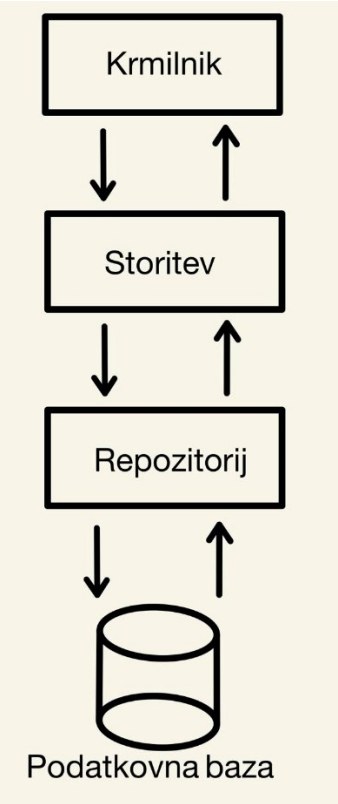
Slika 3.3 Projektna struktura Spring Boot aplikacije

Tabela 3.1 Opisi paketov znotraj Spring Boot aplikacije

Ime podmape/paketa	Opis
controller	So edini razredi, ki so dostopni zunanjim entitetam. Uporabljajo se za upravljanje klicev http (GET, POST, PUT, DELETE). Vsebujejo minimalno količino logike. Za pridobivanje podatkov kličejo »service« razrede.
service	So razredi srednjega nivoja pristopa krmilnik-storitev-repozitorij in vsebujejo glavno logiko aplikacije. Vsebujejo vso logiko, s katero obdelujemo podatke znotraj aplikacije. Podatke, ki jih obdelujejo, pridobivajo iz repozitorij razredov.
repo	V paketu repo (repository) so razredi, katerih edina naloga je komunikacija s podatkovno bazo in so po principu krmilnik-storitev-repozitorij v najnižjem nivoju. Ti razredi izvršijo CRUD operacijo nad našo podatkovno bazo.
exceptions	So razredi, ki jih ustvarimo po meri za upravljanje izjem znotraj naše aplikacije. Z njihovo pomočjo lahko natančneje ugotavljamo in dokumentiramo napake oziroma izjeme znotraj aplikacije.
advice	So razredi, ki lahko za določen krmilnik, paket ali celotno aplikacijo določijo, kakšen odgovor pošljemo nazaj uporabniku preko klica http, če pride do določene izjeme. S pomočjo teh razredov se lahko izognemo podvajanju kode.

model	V paketu model shranjujemo vse naše razrede, ki jih uporabljamo za modeliranje podatkov. Ti razredi ponavadi zrcalijo podatke, ki jih shranjujemo v naših bazah. Z njimi lahko preprosto podatke iz baze ali niza JSON priredimo v kodi in nato jih po meri obdelujemo, pošiljamo ali prejemamo.
util	So pomožni razredi, ki se lahko uporabljajo kjerkoli v aplikaciji. Takšen paket se lahko imenuje tudi »shared«, saj lahko razrede znotraj paketa delimo med vsemi razredi.

Na Slika 3.4 Prikaz vzorca krmilnik-storitev-strežnik imamo prikaz delovanja principa krmilnik-storitev-strežnik, katerega smo uporabili v naši rešitvi. Krmilnik skrbi za zahteve HTTP. Če aplikacija zazna, da je prišla zahteva HTTP, izvrši metodo v krmilniku glede na zahtevo. Ker krmilniki ne vsebujejo logike, kličejo storitve, ki so razredi, v katerih je vsebovana vsa logika naše aplikacije. Te podatke, pridobljene iz repozitorijev, storitve obdelajo in jih pošljejo nazaj krmilniku – ta podatke pošlje nazaj v odgovoru uporabniku, ki je izvršil zahtevo.



Slika 3.4 Prikaz vzorca krmilnik-storitev-strežnik

Primer poteka operacij vzorca krmilnik-storitev-repozitorij:

1. Uporabnik pošlje aplikaciji zahtevo HTTP
2. Aplikacija glede na zahtevo izvrši metodo v krmilniku
3. Metoda kliče metodo v pravilni storitvi
4. Metoda v storitvi kliče repozitorij
5. Repozitorij izvrši CRUD operacijo in pošlje podatke nazaj storitvi
6. Storitev obdela podatke in jih pošlje nazaj krmilniku
7. Krmilnik podatke preko odgovora HTTP vrne uporabniku.

3.2.2 Protokol HTTPS in varnostni žeton JWT

Za varnost našega zalednega dela smo poskrbeli na dva načina. Najprej smo povezavo do aplikacije omogočili le preko protokola HTTPS, drugi način pa je bil, da smo za avtentikacijo implementirali žeton JWT.

Da omogočimo protokol HTTPS za našo aplikacijo, potrebujemo certifikat. Če ga še nimamo, ga lahko sami generiramo s pomočjo orodja keytool. Za našo aplikacijo smo uporabili PKCS12 in spodnji ukaz (Izrezek kode 3.2).

```
1. keytool -genkeypair -alias <vzdevek> -keyalg RSA -keysize 4096
   -storetype PKCS12 -keystore springboot.p12 -validity 3650 -
   storepass <geslo>
```

Izrezek kode 3.2 Ukaz za generiranje PCKS12 datoteke s ključem

S tem ukazom povemo orodju, da želimo generirati ključni par s podanim vzdevkom (alias), z izbranim kriptografskim algoritmom, velikostjo ključa, tipom ključa, imenom ključa, časom veljavnosti ključa v dneh in geslom. Ko izvršimo ta ukaz, nas orodje še vpraša za ime, ime enote v organizaciji (če obstaja), ime organizacije, mesto, regijo in dvomestno kodo za državo. Ko smo certifikat ustvarili, ga moramo dodati v našo aplikacijo tako, da generirano datoteko shranimo v mapo resources. Da omogočimo protokol HTTPS, moramo nato le še v nastavitveno datoteko applications.properties dodati spodnje vrstice (Izrezek kode 3.3).

```
# vrsta ključa
server.ssl.key-store-type=PKCS12
# ključa
server.ssl.key-store=classpath:keystore/diplomska.p12
# geslo ključa
server.ssl.key-store-password=diplomska2022
# vzdevek ključa
server.ssl.key-alias=diplomska-2022
# omogočimo https oz. ssl
server.ssl.enabled=true
```

Izrezek kode 3.3 Nastavitve za omogočanje protokola HTTPS

Da onemogočimo avtomatsko generiranje varnostnih nastavitev, dodamo v našo application.java datoteko le še anotacijo (Izrezek kode 3.4).

```
@SpringBootApplication(exclude = { SecurityAutoConfiguration.class })
public class Application {
  public static void main(String[] args) {
            SpringApplication.run(Application.class, args);
  }
}
```

Izrezek kode 3.4 Onemogočanje avtomatske konfiguracije varnosti

Naslednji korak je implementacija avtentikacije s pomočjo žetona JWT. Za slednje je potrebno kar nekaj logike. Med drugim se moramo odločiti, kaj bo skrivnost za JWT, kako dolgo bo veljaven žeton JWT in kako dolgo bo veljaven obnovitveni žeton (Izrezek kode 3.5). V našem primeru smo se odločili za skrivnost naključni dolgi niz, za veljavnost žetona JWT približno 31 dni in za veljavnost obnovitvenega žetona približno 41 dni. Veljavnost smo morali zapisati v milisekundah. Prav tako smo ustvarili razred za vloge uporabnika. V Izrezek kode 3.6 smo v ta razred dodali navadno USER vlogo ter administrativno vlogo ADMIN.

```
jwt.secret=717a2de65522dd8f7dba06114b1475c7b5b9cd077cee93905398430bf285f1
# cca. 31 days
jwt.expirationDateInMs=27000000000
# cca. 41 days
jwt.refreshExpirationDateInMs=3600000000
```

Izrezek kode 3.5 Nastavitve žetona JWT

```
1.  public class Roles {
2.      public static String ADMIN = "ADMIN";
3.      public static String USER = "USER";
4.  }
```

Izrezek kode 3.6 Razred za vloge uporabnika

Naslednje, kar bomo potrebovali, sta dva razreda za podatke za avtentikacijo uporabnika. En razred se bo uporabil za zahtevo, v kateri želimo uporabniško ime in geslo (Izrezek kode 3.7), drugi pa bo vseboval naša dva žetona (Izrezek kode 3.8).

```
1.  @Data
2.  @NoArgsConstructor
3.  @AllArgsConstructor
4.  public class JwtRequest implements Serializable {
5.      @Serial
6.      private static final long serialVersionUID = 6231579864324697L;
7.      private String username;
8.      private String password;
9.  }
```

Izrezek kode 3.7 Razred za podatke avtentikacijske zahteve

```
1.  @Data
2.  @AllArgsConstructor
3.  public class JwtResponse implements Serializable {
4.      @Serial
5.      private static final long serialVersionUID = -8091879091924046844L;
6.      private final String jwttoken;
7.      private final String refreshToken;
8.  }
```

Izrezek kode 3.8 Razred za podatke za odgovor avtentikacijske zahteve

Ker želimo uporabljati protokol HTTPS in avtentikacijo z žetonom JWT, potrebujemo še nekaj nastavitev. V ta namen smo izdelali nov konfiguracijski razred WebSecurityConfig.java (Izrezek kode 3.9). V njem smo nastavili, da bomo za šifriranje uporabili kodirnik BCrypt, ki je vključen v modulu Spring Security. S tem kodirnikom bomo šifrirali vsa gesla ter jih tudi med seboj primerjali za namene avtentikacije. V Izrezek kode 3.9 Konfiguracijski razred WebSecurityConfig.java smo od vrstice 22 naprej določili, kdo lahko dostopa do določenih povezav naše aplikacije. Poti »/authenticate« in »/refresh« se uporabljata za prijavo

uporabnika in sta dostopni vsem, vključno z neprijavljenimi uporabniki. Nato smo določili, do česa lahko dostopajo le prijavljeni uporabniki z vlogo ADMIN ter poti, do katerih imajo dostop vsi prijavljeni uporabniki. Nato smo se še odločili, da ne bomo uporabili seje za stanje uporabnika ter da se mora preveriti žeton JWT ob vsakem klicu, kjer je ta tudi potreben.

```
1. @Configuration
2. @EnableWebSecurity
3. @EnableGlobalMethodSecurity(prePostEnabled = true)
4. public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
5.           .
6.           .
7.           .
8.     @Autowired
9.     public void configureGlobal(AuthenticationManagerBuilder auth)
10.throws Exception {
11.          auth.userDetailsService(jwtUserDetailsService)
12.          .passwordEncoder(passwordEncoder());
13.    }
14.    @Bean
15.    public PasswordEncoder passwordEncoder() {
16.        return new BCryptPasswordEncoder();
17.    }
18.          .
19.          .
20.          .
21.    @Override
22.    protected void configure(HttpSecurity httpSecurity) throws Exception {
23.        httpSecurity.csrf().disable()
24.                .authorizeRequests()
25.                  .antMatchers("/authenticate**", "/refresh/**")
26.                  .permitAll()
27.          .antMatchers("/api/v1/admin/**", "/api/v1/nodered/**")
28.                  .hasAuthority(Roles.ADMIN.toString())
29.          .antMatchers("/api/v1/employees/**", , "/api/v1/hours/**",
30.                                                  "/api/v1/monthly/**")
31.          .hasAnyAuthority(Roles.USER.toString(),Roles.ADMIN.toString())
32.          .and().exceptionHandling()
33.                  .authenticationEntryPoint(jwtAuthenticationEntryPoint)
34.                  .and().sessionManagement()
35.          .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
36.        httpSecurity.addFilterBefore(jwtRequestFilter,
37.                  UsernamePasswordAuthenticationFilter.class);
38.    }
39.}
```

Izrezek kode 3.9 Konfiguracijski razred WebSecurityConfig.java

Žeton JWT je potrebno generirati, zato smo ustvarili pomožni razred JwtTokenUtil.java, ki nam bo z našimi podatki ustvaril žeton JWT ter preverjal njegovo veljavnost. Ta razred iz

application.properties pridobi potrebne informacije za generiranje žetona, ki smo jih v prejšnjih korakih nastavili. Najpomembnejši del razreda je generiranje samega žetona, ki slednje stori z dvema metodama na Izrezek kode 3.10.

```
1.      public String generateToken(UserDetails userDetails) {
2.          Map<String, Object> claims = new HashMap<>();
3.          Set<String> roles = new HashSet<>();
4.          for(GrantedAuthority role : userDetails.getAuthorities()) {
5.              roles.add(role.getAuthority());
6.          }
7.          claims.put("Roles", roles.toArray());
8.          return doGenerateToken(claims, userDetails.getUsername());
9.      }
10.
11.     private String doGenerateToken(Map<String, Object> claims, String subject) {
12.         return Jwts.builder().setClaims(claims).setSubject(subject)
13.             .setIssuedAt(new Date(System.currentTimeMillis()))
14.                 .setExpiration(new Date(System.currentTimeMillis() + jwtExperationInMs))
15.                 .signWith(SignatureAlgorithm.HS512, secret).compact();
16.     }
17. }
```

Izrezek kode 3.10 Implementacija generiranja žetona JWT

Metoda generateToken sprejme objekt, ki vsebuje naše uporabniško ime, geslo in njegove vloge, ki se bodo uporabile v namene avtorizacije. V našem primeru smo kot uporabniško ime uporabili e-poštni naslov uporabnika. Nato metoda iz objekta vzame vloge, jih mapira v set ter jih doda v claims objekt. Claims in uporabniško ime uporabnika kot argumente podamo funkciji doGenerateToken, katera s pomočjo razreda Jwts generira žeton. V žeton shrani claims objekt, naše uporabniško ime, čas kdaj je bil žeton ustvarjen, kako dolgo velja ter ga podpiše z našo skrivnostjo. Za prijavo uporabnika smo ustvarili nov krmilnik JwtAuthenticationController, ki bo skrbel za vse klice za prijavo. Uporabnik se lahko prijavi preko POST zahteve in mora poslati v zahteve objekt, ki vsebuje uporabniško ime in geslo.

```
1.  @PostMapping("/authenticate")
2.      public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtRequest
    authenticationRequest) throws Exception {
3.          authenticate(authenticationRequest.getUsername(),
    authenticationRequest.getPassword());
4.          final UserDetails userDetails = userDetailsService
5.                  .loadUserByUsername(authenticationRequest.getUsername());
6.          final String token = jwtTokenUtil.generateToken(userDetails);
7.          final String refreshToken = jwtTokenUtil.generateRefreshToken(userDetails);
8.          return ResponseEntity.ok(new JwtResponse(token, refreshToken));
9.      }
10.  private void authenticate(String username, String password) throws Exception {
11.         try {
12.             authenticationManager
13.             .authenticate(new UsernamePasswordAuthenticationToken(username, password));
14.         } catch (DisabledException e) {
15.             throw new Exception("USER_DISABLED", e);
16.         } catch (BadCredentialsException e) {
17.             throw new Exception("INVALID_CREDENTIALS", e);
18.         }
19.     }
```

Izrezek kode 3.11 Implementacija žetona JWT v krmilniku JwtAuthenticationController

Metoda createAuthenticationToken iz Izrezek kode 3.11 prvo preveri, če uporabnik obstaja ter če je vpisal pravilno geslo. To naredi z metodo authenticate, ki kliče metodo authenticate (Izrezek kode 3.12) iz razreda AuthenticationManager. Ta najprej poišče uporabnika iz naše baze in nato s kodirnikom BCrypt preveri, če se gesli ujemata. Če uporabnika ne najde ali se gesli ne ujemata, sproži izjemo. Če je bila avtentikacija uspešna, poišče uporabnika iz naše baze podatkov in vrne objekt, ki vsebuje e-poštni naslov, geslo in vloge uporabnika, za tem pa s pomočjo JwtTokenUtil razreda generira žeton in ga vrne uporabniku.

```
1.  @Override
2.      public Authentication authenticate(Authentication authentication) throws
    AuthenticationException {
3.          final String username = authentication.getName();
4.          final String password = authentication.getCredentials().toString();
5.          final Employee employee = employeeService.getEmployeeByEmailAuth(username);
6.          if(passwordEncoder.matches(password, employee.getPassword())) {
7.              return new UsernamePasswordAuthenticationToken(username, password,
    employee.getRoles());
8.          } else {
9.              throw new BadCredentialsException("1000");
10.         }
11.     }
```

Izrezek kode 3.12 Implementacija preverjanja prijavnih podatkov

Žeton JWT je v bistvu šifriran niz, ki vsebuje vse potrebne informacije uporabnika za njegovo nadaljnjo avtentikacijo in avtorizacijo. Ta niz mora biti vedno v glavi zahteve HTTPS s pripono »Bearer «, če želi uporabnik dostopati do določenih delov aplikacije in podatkov. Preden bomo obdelali zahtevo, bomo ta žeton preverili s pomočjo razreda JwtRequestFilter, ki vsebuje metodo doFilterInternal (Izrezek kode 3.13).

Metoda najprej preveri, če imamo v glavi zahteve žeton, ki je shranjen pod ključem »Authorization«, ter če se začne s pravilno pripono. Nato iz žetona preberemo uporabniško ime, poiščemo uporabnika v naši bazi ter preverimo veljavnost žetona. Če je preverjanje uspešno, označimo znotraj konteksta tudi avtentikacijo kot uspešno in z zahtevo nadaljujemo.

```
1.  @Override
2.      protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
    response, FilterChain chain)
3.              throws ServletException, IOException {
4.          final String requestTokenHeader = request.getHeader("Authorization");
5.          ObjectMapper mapper = new ObjectMapper();
6.          String username = null;
7.          String jwtToken = null;
8.          if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {
9.              jwtToken = requestTokenHeader.substring(7);
10.             try {
11.                 username = jwtTokenUtil.getUsernameFromToken(jwtToken);
12.             } catch (IllegalArgumentException e) {
13.                 System.out.println("Unable to get JWT Token");
14.             } catch (ExpiredJwtException e) {
15.                 response.setStatus(HttpStatus.UNAUTHORIZED.value());
16.                 response.setHeader(HttpHeaders.AUTHORIZATION, "expired");
17.                 response.getWriter().write(mapper.writeValueAsString(e));
18.             }
19.         } else {
20.             logger.warn("JWT Token does not begin with Bearer String");
21.         }
22.         if (username != null) {
23.             UserDetails userDetails = this.jwtUserDetailsService
24.                                         .loadUserByUsername(username);
25.
26.         if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {
27.         UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken
28.                         = new UsernamePasswordAuthenticationToken(
29.                           userDetails, null, userDetails.getAuthorities());
30.                 usernamePasswordAuthenticationToken
31.                         .setDetails(
32.                         new WebAuthenticationDetailsSource().buildDetails(request)
33.                         );
34.             SecurityContextHolder.getContext()
35.                         .setAuthentication(usernamePasswordAuthenticationToken);
36.             }
37.         }
38.         chain.doFilter(request, response);
39.     }
40.
```

Izrezek kode 3.13 Implementacija filtra za preverjanje žetonov JWT

V istem razredu še imamo metodo shouldNotFilter (Izrezek kode 3.14), ki se uporablja za preverjanje, katere točke dostopa ne filtriramo oziroma ne preverjamo prisotnosti žetona JWT. To je v našem primeru točka »/authenticate«, ki se uporablja za prijavo uporabnika.

```
1.  @Override
2.      protected boolean shouldNotFilter(HttpServletRequest request)
3.              throws ServletException {
4.          String path = request.getRequestURI();
5.          return "/authenticate".equals(path);
6.      }
```

Izrezek kode 3.14 Metoda za preverjanje točk brez preverjanja avtentikacije

3.2.3 Vzpostavitev povezave s podatkovno bazo

Vzpostavitev povezave z našo podatkovno bazo je precej preprost postopek, saj moramo v našo application.properties datoteko dodati le par vrstic iz Izrezek kode 3.15.

```
1.  spring.data.mongodb.authentication-database=admin
2.  spring.data.mongodb.username=admin
3.  spring.data.mongodb.password=admin
4.  spring.data.mongodb.database=main
5.  spring.data.mongodb.port=27017
6.  spring.data.mongodb.host=localhost
```

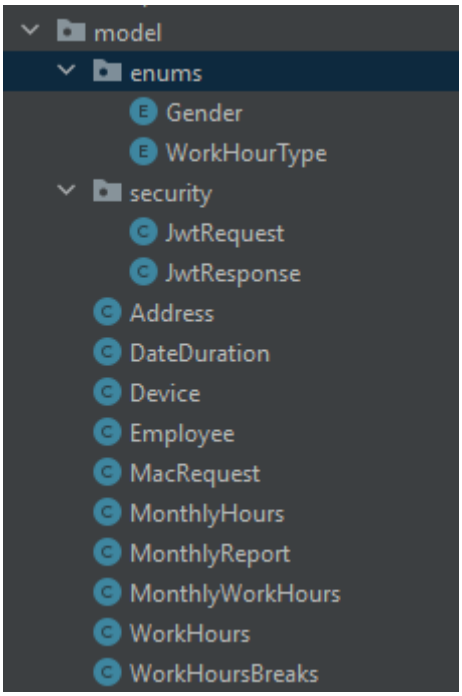
Izrezek kode 3.15 Konfiguracija povezave s podatkovno bazo

Določiti moramo, s katero bazo znotraj MongoDB gruče (angl. cluster) se bomo avtenticirali in s katerim uporabniškim imenom ter geslom. Nato določimo, katero podatkovno bazo bomo uporabili za potrebe aplikacije, kje se nahaja podatkovna baza ter vrata, preko katerih bomo komunicirali.

V našem primeru imamo podatkovno bazo kar na lokalnem računalniku in je zato pot do nje localhost, vrata pa 27017, kar so privzeta vrata za to podatkovno bazo. Pri razvoju smo uporabili preprosto konfiguracijo za bazo, če pa bi želeli to aplikacijo uporabljati znotraj organizacij, bi zagotovo uporabili naprednejše prijavne podatke ter kakšno dodatno knjižnico, kot na primer Jasypt [23], s katero lahko podatke znotraj application.properties šifriramo. Ko smo te podatke pravilno vnesli v našo datoteko, se bo aplikacija ob zagonu sama povezala na to podatkovno bazo. V primeru, da se aplikacija ne more povezati s podatkovno bazo, nam bo ta vrnila napako.

3.2.4 Podatkovni modeli

Znotraj aplikacije uporabljamo različne razrede za shranjevanje in obdelovanje podatkov. V našem projektu vse te razrede shranjujemo v mapi »model« (glej Slika 3.5 Podatkovni razredi znotraj mape model).



Slika 3.5 Podatkovni razredi znotraj mape model

Znotraj podmape enums imamo dva razreda, Gender (Izrezek kode 3.16 Razred Gender) in WorkHourType (Izrezek kode 3.17 Razred WorkHourType), ki se uporabljata znotraj drugih razredov za določanje spola delavca ter tipa ur.

```
1.  public enum Gender {
2.      MALE, FEMALE, OTHER
3.  }
```

Izrezek kode 3.16 Razred Gender

```
1.  public enum WorkHourType {
2.      WORK, SICK_LEAVE, LEAVE
3.  }
```

Izrezek kode 3.17 Razred WorkHourType

Ustvarili smo tudi razred Address (Izrezek kode 3.18) in Device (Izrezek kode 3.19), ki ju uporabljamo znotraj razreda Employee (Izrezek kode 3.20) za beleženje naslova uporabnika in njegovih registriranih naprav.

```
1.  @Data
2.  @AllArgsConstructor
3.  public class Address {
4.      private String street;
5.      private String postalCode;
6.      private String city;
7.      private String country;
8.  }
```

Izrezek kode 3.18 Razred Address

```
1.  @Data public class Device {
2.     String mac;
3.     String nickname;
4.  }
```

Izrezek kode 3.19 Razred Device

V razredu Employee najdemo vse pomembne informacije, ki jih potrebujemo za posameznega uporabnika.

```
1.  @Data
2.  @Document(collection = "employees")
3.  @JsonIgnoreProperties(ignoreUnknown = true)
4.  @JsonInclude(JsonInclude.Include.NON_NULL)
5.  public class Employee {
6.      @Id
7.      private String uuid;
8.      @NotNull
9.      private String name;
10.     @NotNull
11.     private String surname;
12.     @Indexed(unique = true)
13.     @NotNull
14.     private String email;
15.     @Indexed(unique = true)
16.     @NotNull
17.     private String phone;
18.     @NotNull
19.     private Address address;
20.     @NotNull
21.     private Gender gender;
22.     @Indexed(unique = true)
23.     private ArrayList<Device> deviceId;
24.     private ArrayList<SimpleGrantedAuthority> roles;
25.     private String password;
26. }
```

Izrezek kode 3.20 Razred Employee

Z anotacijo »@Data« smo določili, da nam knjižnica Lombok ob zagonu generira setter, getter, toString in konstruktor metode. Z naslednjo anotacijo smo določili, v kateri zbirki naj bi se shranjevali objekti tipa Employee znotraj naše baze. Z »@JsonIgnoreProperties« smo določili, da ob serializaciji ignoriramo neznane spremenljivke, z anotacijo »@JsonInclude« pa povemo, da pri serializaciji naj ne zapisujemo le »null« vrednosti. Znotraj modela najdemo nato razne spremenljivke za osebne podatke uporabnika. Za shranjevanje registriranih naprav in vlog uporabnika smo uporabili podatkovno strukturo ArrayList, saj lahko ima uporabnik znotraj podjetja več naprav in/ali vlog.

Nad spremenljivkami najdemo tudi nekaj dodatnih anotacij:

- @Id – označimo spremenljivko, s katero identificiramo objekt,
- @NotNull – iz validation-api, s katero označimo spremenljivke, ki ne smejo vsebovati vrednosti »null«,
- @Indexed – označimo spremenljivke, ki jih želimo indeksirati z uporabo MongoDB funkcij. Če nastavimo pri anotaciji »unique = true«, slednje pomeni, da je ta vrednost znotraj naše podatkovne baze enolična.

Delovni čas beležimo s pomočjo razreda MonthlyWorkHours (Izrezek kode 3.21). Ta vsebuje identifikacijski niz in spremenljivko tipa Map, v katero shranimo delovne ure tipa WorkHours (Izrezek kode 3.22) pod numeričnim ključem, ki predstavlja dan v mesecu. Identifikacijski niz je sestavljen iz ID uporabnika, trenutnega meseca in leta. Z anotacijo »@NoArgsConstructor« in »@AllArgsConstructor« določimo, da se razredu generira konstruktor brez parametrov in z vsemi parametri.

```
1.  @Data
2.  @Document(collection = "monthlyWorkHours")
3.  @NoArgsConstructor
4.  @AllArgsConstructor
5.  public class MonthlyWorkHours {
6.      @Id
7.      private String uuid; // employeeId + _ + month + _ + year
8.      private Map<Integer, WorkHours> workHours;
9.  }
```

Izrezek kode 3.21 Razred MonthlyWorkHours

```
1.  @Data
2.  @AllArgsConstructor
3.  @NoArgsConstructor
4.  public class WorkHours {
5.      @Id
6.      private String uuid; // id is day of the month
7.      @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
8.      private LocalDateTime startTime;
9.      @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
10.     private LocalDateTime endTime;
11.     private WorkHourType workHourType;
12.     private ArrayList<WorkHoursBreaks> breaks;
13.     private Long breakTime;
14.     private Long totalTime; // in minutes for ease of calculations
15. }
```

Izrezek kode 3.22 Razred WorkHours

Razred WorkHours, ki ga uporabljamo znotraj Map strukture razreda MonthlyWorkHours, je namenjen beleženju delovnega časa posameznih dni v mesecu. Vsebuje čas začetka in konca delovnega časa, tip ur, čas odmorov v dnevu in celotni čas odmorov ter delovnega časa v minutah. Z anotacijo »@JsonFormat« tukaj določimo točen format, v katerega naj LocalDateTime spremenljivki ob serializaciji formatirata. Razred WorkHoursBreak (Izrezek kode 3.23), katerega uporabljamo znotraj WorkHours, vsebuje le spremenljivki za začetek in konec odmorov. Oba razreda vsebujeta še nekaj dodatnih konstruktor metod, ki se razlikujejo le po parametrih, ne vsebujejo pa dodatne logike.

```
1.  @Data
2.  @AllArgsConstructor
3.  @NoArgsConstructor
4.  public class WorkHoursBreaks {
5.      @Id
6.      private String id;
7.      @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
8.      private LocalDateTime startTime;
9.      @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
10.     private LocalDateTime endTime;
11. }
```

Izrezek kode 3.23 Razred WorkHoursBreak

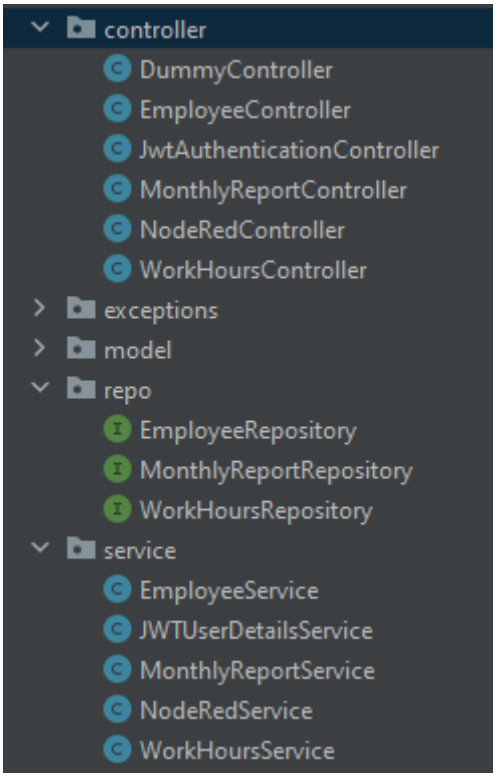
Zadnji model, katerega uporabljamo, je model MacRequest (Izrezek kode 3.24) za podatke, ki jih pridobivamo od Node-RED aplikacije. Ta nam vrača dva polja z MAC naslovi, eno polje za nove naslove v omrežju in eno polje za naslove, ki niso več povezani z našim lokalnim omrežjem.

```
1.  @Data
2.  @AllArgsConstructor
3.  @NoArgsConstructor
4.  public class MacRequest {
5.      private String[] newDevices;
6.      private String[] oldDevices;
7.  }
```

Izrezek kode 3.24 Razred MacRequest

3.2.5 Implementacija načrtovalskega vzorca krmilnik-storitev-repozitorij

Za komunikacijo z Android aplikacijo in možnimi drugimi aplikacijami smo znotraj spletne aplikacije implementirali komplete vzorca KSR za avtentikacijo ter prenos in obdelavo različnih podatkov (Slika 3.6). Pridobivali in obdelovali se bodo podatki o delavcu, delovnem času, mesečnih poročilih delovnega časa in podatki, prejeti od aplikacije Node-RED. Vsak tak komplet je sestavljen iz treh razredov: krmilnika, storitve in repozitorija. Znotraj teh se prav tako uporabljajo razni modelni razredi.



Slika 3.6 Krmilnik-storitev-repozitorij razredi

Z implementacijo krmilnika EmployeeController smo omogočili različne načine pridobivanja podatkov o uporabnikih. Možno je pridobivanje podatkov o vseh uporabnikih, trenutnega uporabnika in uporabnikov glede na ID, ime in priimek. Prav tako je možno dodati nove uporabnike, posodobiti podatke posameznih uporabnikov, odstraniti uporabnike in dodajanje ali odstranjevanje delovnih naprav za posameznega delavca. Ker smo vgradili precej različnih funkcionalnosti za manipulacijo uporabnikov, bomo nadalje obravnavali le najpomembnejše.

V nadaljnjem bomo za povezave uporabljali le končnice le-teh, saj smo se pri razvoju sklicevali do aplikacije preko localhost naslova.

V tem razredu imamo eno metodo za ustvarjanje novih uporabnikov, ki jo lahko pokličemo preko povezave “localhost/api/v1/employees/new” in v telesu zahteve pošljemo Employee objekt v obliki niza JSON. Nato ta metoda pokliče addNewEmployee metodo (Izrezek kode 3.25) iz razreda EmployeeService. Če ta metoda uspešno ustvari in doda novega uporabnika v našo podatkovno bazo, zapišemo v konzolo ime in priimek novega uporabnika ter nato kot odgovor vrnemo HTTP status OK (200).

```
1.      @PostMapping(value = "/new",consumes = {MediaType.APPLICATION_JSON_VALUE})
2.      public ResponseEntity<Void> addNewEmployee(@RequestBody Employee employee) {
3.          employeeService.addNewEmployee(employee);
4.          log.info("New user was created with name: {} {}",
5.                                  employee.getName(), employee.getSurname());
6.          return new ResponseEntity<>(HttpStatus.OK);
7.      }
```

Izrezek kode 3.25 Končna točka za ustvarjanje novih uporabnikov

Metoda addNewEmployee (Izrezek kode 3.26) preveri, če je pridobljen objekt tipa Employee pravilen glede na pravila, ki smo jih definirali v modelu (npr. ali je vrednost null). To naredi s pomočjo validatorja, ki nam vrne Set kršitev pravil, če obstajajo. Če objekt ni pravilen, izvrši izjemo in se uporabnik ne doda. Nato metoda preveri, ali uporabnik z e-poštnim naslovom ali telefonsko številko ter registriranimi napravami v naši podatkovni bazi že obstaja. Če uporabnik z enim ali več podatki že obstaja, se novi uporabnik ne ustvari in se izvrši izjema. Metoda prav tako preveri, če ima uporabnik določeno geslo. Če ga nima, nastavi kot geslo kombiniran niz imena in priimka uporabnika ter ga šifrira s knjižnico bycrpt. V primeru, da je geslo že nastavljeno, se to tudi šifrira. Preden uporabnika dodamo v bazo, mu še dodamo navadno USER vlogo; v primeru, da objekt ne vsebuje nobenih vlog.

```
1.   public void addNewEmployee(Employee employee) {
2.         Set<ConstraintViolation<Employee>> violation = validator.validate(employee);
3.         if (!violation.isEmpty())
4.             throw new ConstraintViolationException(violation);
5.         if (employeeRepository.findFirstEmployeeByEmailOrPhone(employee.getEmail(),
6.              employee.getPhone()).isEmpty() && (employee.getDeviceId() == null ||
7.             employeeRepository.findEmployeeByDeviceIdMac(
8.                       employee.getDeviceId().get(0).getMac()).isEmpty())) {
9.             if (employee.getPassword() == null) {
10.                employee.setPassword(passwordEncoder.encode(
11.                                 employee.getName() + employee.getSurname()));
12.            } else {
13.                employee.setPassword(passwordEncoder.encode(employee.getPassword()));
14.            }
15.            if(employee.getRoles() == null) {
16.                ArrayList<SimpleGrantedAuthority> roles = new ArrayList<>();
17.                roles.add(new SimpleGrantedAuthority(Roles.USER.toString()));
18.                employee.setRoles(roles);
19.            }
20.            employeeRepository.insert(employee);
21.            return;
22.        }
23.        throw new EmployeeNotCreatedException
24.            (" employee with E-mail, Phone number or Device ID already exists");
25.    }
26.
```

Izrezek kode 3.26 Metoda za dodajanje uporabnikov

Repozitorij razredi že vsebujejo vse osnovne CRUD operacije, ki jih potrebujemo za določen model, zato vanje ne rabimo dodajati veliko novih operacij. V tem primeru smo dodali operacijo »findFirstEmployeeByEmailOrPhone« (Izrezek kode 3.27), ki poišče prvega uporabnika, da vsebuje podan e-poštni naslov ali telefonsko številko. Za to nam ni potrebno v repozitorij pisati veliko dodatne kode, temveč moramo samo pravilno sestaviti definicijo metode s pomočjo ključnih besed, kot so na primer And, Or, Between, get, find, Less, By in imen spremenljivk v modelnem razredu, katerega uporabljamo.

```
1.  Optional<Employee>findFirstEmployeeByEmailOrPhone(String email, String phone);
```

Izrezek kode 3.27 Definicija nove metode za izvršitev CRUD operacij

Za dostop do osebnih podatkov ima uporabnik na voljo povezavo »localhost/api/v1/employees/new« (Izrezek kode 3.28). Ker prijavljen uporabnik pri vsakem klicu pošlje žeton JWT, ga lahko identificiramo preko tega in njegove podatke najdemo preko informacij, zapisanih v žetonu. Pri tem pokliče metodo getEmployeeByEmail (Izrezek kode 3.29) iz storitvenega razreda in poda e-poštni naslov uporabnika, ki ga razbere iz žetona JWT

s pomočjo JwtTokenUtil razreda. Če metoda najde uporabnika, ga vrne skupaj s Statusom 200 v odgovoru.

```
1.    @GetMapping("/user")
2.      public ResponseEntity<Employee> fetchUserData(
3.              @RequestHeader (name="Authorization") String token) {
4.          return new
5.              ResponseEntity<>(employeeService.getEmployeeByEmail(
6.                      jwtTokenUtil.getUsernameFromToken(token.substring(7))),
7.                      HttpStatus.OK);
8.      }
```

Izrezek kode 3.28 Krmilniška metoda za pridobivanje osebnih podatkov uporabnika

V storitvi metoda preko repozitorija poišče uporabnika in preveri, če ta ni v vrednosti null. V primeru, da uporabnika ne najde, vrne izjemo »EmployeeNotFoundException«. Če uporabnik s tem e-poštnim naslovom obstaja v naši bazi, ga metoda vrne. Takšno metodo bi lahko naredili za katerokoli vrednost v našem modelu. V repozitoriju smo enako kot v prejšnjem primeru ustvarili novo definicijo »findEmployeeByEmail«, s katero metoda išče v bazi po e-poštnem naslovu.

```
1.  public Employee getEmployeeByEmail(String email) {
2.          Employee returnEmployee = employeeRepository.findEmployeeByEmail(email);
3.          if (returnEmployee != null) {
4.              returnEmployee.setPassword(null);
5.              return returnEmployee;
6.          }
7.          throw new EmployeeNotFoundException("e-mail: " + email);
8.      }
```

Izrezek kode 3.29 Metoda za iskanje uporabnikov preko e-pošte

V tem delu aplikacije smo implementirali posodobitev podatkov uporabnika (Izrezek kode 3.30). To storimo preko HashMap-a, v katerem shranimo vse spremembe, ki jih želimo shraniti. Nato krmilnik slednje poda storitvi, ki shrani spremembe v podatkovno bazo. To smo implementirali kar preprosto, saj smo omogočili posodobitev le določenih podatkov.

```
1.  public Employee updateEmployeeData(HashMap<String,String> changes) {
2.          Optional<Employee> employeeOptional =
    employeeRepository.findById(changes.get("id"));
3.          if (employeeOptional.isEmpty()) {
4.              throw new EmployeeNotFoundException("uuid: " + changes.get("id"));
5.          }
6.          Employee employee = employeeOptional.get();
7.          employee.setSurname(changes.get("surname"));
8.          employee.setName(changes.get("name"));
9.          Address address = employee.getAddress();
10.         address.setCity(changes.get("city"));
11.         address.setStreet(changes.get("street"));
12.         address.setPostalCode(changes.get("postalCode"));
13.         employee.setAddress(address);
14.         return employeeRepository.save(employee);
15.     }
```

Izrezek kode 3.30 Metoda za posodabljanje uporabniških podatkov

Metoda v storitvi iz spremenljivke »changes« pridobi spremenljivko »id« in preko nje poišče uporabnika – če tega ne najde, sproži izjemo – če uporabnika najdemo, mu spremenimo naprej določena polja in posodobimo uporabniške podatke preko funkcije »save«, katero vsebuje naš repozitorij. Ker podamo tej funkciji naš celoten Employee objekt, se ne ustvari novi dokument znotraj kolekcije, temveč se posodobi objekt preko spremenljivke id.

Druge metode, ki se uporabljajo za uporabnika, so v večini primerov podobne že opisanim in delajo v večini preprostejše operacije, kot na primer iskanje in brisanje uporabnikov preko različnih spremenljivk ter dodajanje in odstranjevanje naprav posameznega uporabnika.

3.2.6 Implementacija načrtovalskega vzorca KSR za obdelavo delovnih ur

Za osnovno implementacijo obdelave delovnih ur smo morali implementirati beleženje začetka in konca delovnika in odmora. Prav tako smo omogočili pridobivanje podatkov o statusu uporabnika (ali dela, je na odmoru ipd.) ter število delovnih ur za trenuten dan in teden.

V krmilniku »WorkHoursController« smo določili kot glavno pot »api/v1/hours« in za vsako metodo določili enolično pripono. V primeru za beleženje začetka delavnika smo izbrali pripono »/new/{uuid}/{type}« (Izrezek kode 3.31), kjer v poti dodamo dve spremenljivki, in sicer id uporabnika ter tip ure. Ta tip je lahko »WORK« za navadni delavnik, »SICK_LEAVE« za bolniški stalež in »LEAVE« za namene beleženje dopusta. V trenutni implementaciji se uporablja le tip »WORK«, smo pa dodali druge tipe za namene nadgradenj. V primeru

zaključka delovnika uporabljamo pripono »/end/{uuid}« (Izrezek kode 3.32), kjer nam je potrebno podati le id uporabnika. Tukaj bi lahko naredili tudi izboljšavo, kjer bi podatke pošiljali preko telesa zahteve in za identifikacijo uporabnika uporabili podatke iz žetona JWT zahteve.

```
1.  @PostMapping(value = "/new/{uuid}/{type}")
2.      public ResponseEntity<String> addNewEntry(@PathVariable String uuid, @PathVariable
    WorkHourType type) {
3.          return new ResponseEntity<>(workHoursService.addNewEntry(uuid, type),
    HttpStatus.OK);
4.      }
5.
```

Izrezek kode 3.31 Končna točka za beleženje začetka delovnika

```
1.      @PutMapping("/end/{uuid}")
2.      public ResponseEntity<MonthlyWorkHours> endEntry(@PathVariable String uuid) throws
    Exception {
3.          return new ResponseEntity<>(workHoursService.endEntry(uuid), HttpStatus.OK);
4.      }
```

Izrezek kode 3.32 Končna točka za beleženje konca delovnika

Za vpis začetka delovnega časa v metodi “addNewEntry” (Izrezek kode 3.33) znotraj WorkHoursService razreda pridobimo instanco Calendar objekta, ki se uporablja ta čas. S pomočjo nje sestavimo identifikacijski niz za dokument, v katerem shranjujemo delovni čas za tekoči mesec. Identifikacijski niz je sestavljen iz id uporabnika, trenutnega meseca in leta, ki so ločeni s podčrtajem. Ta dokument nato vzame Map spremenljivko workHours, doda novi vnos s ključem, ki je enak numeričnemu dnevu v mesecu, kot vrednost pa vnese spremenljivko tipa WorkHours, v katero vpiše trenuten čas in tip ur. V primeru, da dokument ne obstaja, pred vnosom ustvari novo spremenljivko MonthlyWorkHours, ki predstavlja dokument in ga skupaj z vnosom začetka shrani v bazo.

```
1.  public String addNewEntry(String id, WorkHourType workHourType) {
2.          Calendar calendar = Calendar.getInstance();
3.          calendar.setTime(new Date());
4.          log.warn(id + "_" + (calendar.get(Calendar.MONTH)+1) + "_" +
    calendar.get(Calendar.YEAR));
5.          Optional<MonthlyWorkHours> monthlyWorkHoursOptional =
    workHoursRepository.findById(id + "_" + (calendar.get(Calendar.MONTH)+1) + "_" +
    calendar.get(Calendar.YEAR));
6.          if (monthlyWorkHoursOptional.isPresent()) {
7.              MonthlyWorkHours monthlyWorkHours = monthlyWorkHoursOptional.get();
8.              Map<Integer, WorkHours> workHours = monthlyWorkHours.getWorkHours();
9.
10.             workHours.put(calendar.get(Calendar.DATE), new
    WorkHours(LocalDateTime.now(), workHourType));
11.             monthlyWorkHours.setWorkHours(workHours);
12.             return workHoursRepository.save(monthlyWorkHours).toString();
13.         } else {
14.             MonthlyWorkHours monthlyWorkHours = new MonthlyWorkHours();
15.             monthlyWorkHours.setUuid(id + "_" + (calendar.get(Calendar.MONTH)+1) + "_" +
    calendar.get(Calendar.YEAR));
16.             Map<Integer, WorkHours> workHours = new HashMap<>();
17.             workHours.put(calendar.get(Calendar.DATE), new
    WorkHours(LocalDateTime.now(), workHourType));
18.             monthlyWorkHours.setWorkHours(workHours);
19.             return workHoursRepository.save(monthlyWorkHours).toString();
20.         }
21.     }
22.
```

Izrezek kode 3.33 Metoda za shranjevanje in obdelavo začetka delovnika

Osnovna logika za beleženje konca delovnika (Izrezek kode 3.34) je kar preprosta. Najprej preverimo, če sploh obstaja dokument za podanega delavca ter če za trenuten dan že obstaja vnos za začetek delovnika. Če katero od preverjanj ni uspešno, vrnemo uporabniku izjemo. Ker beležimo tudi odmore, preverimo, če še obstajajo kateri nedokončani odmori in jih končamo, pri tem vpišemo kot končni čas odmora trenuten čas. V metodi izračunamo še celoten čas odmorov in delovnika. Pri računanju časa delovnika odštejemo čas odmorov, v primeru, da presegajo 30 minut. Ker ima v večini primerov delavec 30 minut odmora pri osem-urnem delovniku, pri tem izračunu odštejemo vse, kar je nad 30 minut. Po vseh izračunih shranimo vse podatke v bazo podatkov.

```
1.       public MonthlyWorkHours endEntry(String id) throws Exception {
2.          Calendar calendar = Calendar.getInstance();
3.          calendar.setTime(new Date());
4.          Optional<MonthlyWorkHours> monthlyWorkHoursOptional =
    workHoursRepository.findById(id + "_" + (calendar.get(Calendar.MONTH)+1) + "_" +
    calendar.get(Calendar.YEAR));
5.          if (monthlyWorkHoursOptional.isEmpty()) {
6.              throw new EmployeeNotFoundException("No start entry found for this
    employee");
7.          }
8.          MonthlyWorkHours monthlyWorkHours = monthlyWorkHoursOptional.get();
9.          Map<Integer, WorkHours> workHours = monthlyWorkHours.getWorkHours();
10.         if (workHours.get(calendar.get(Calendar.DATE)) == null) {
11.             throw new Exception("Work hours not started for today"); // create new
    expetion
12.         }
13.         WorkHours today = workHours.get(calendar.get(Calendar.DATE));
14.         today.setBreakTime(0L);
15.         if(today.getBreaks()!=null && today.getBreaks().size() > 0) {
16.             if (today.getBreaks().get(today.getBreaks().size()-1).getEndTime() == null)
    {
17.                 today.getBreaks().get(today.getBreaks().size()-
    1).setEndTime(LocalDateTime.now());
18.             }
19.
    today.setBreakTime(WorkHoursUtils.calculateTotalBreakTime(today.getBreaks()));
20.         }
21.         today.setEndTime(LocalDateTime.now());
22.         if(today.getBreakTime() > 30L) {
23.             today.setTotalTime(WorkHoursUtils.calculateWorkTime(today.getStartTime(),
    today.getEndTime()) + 30L - today.getBreakTime());
24.         } else {
25.             today.setTotalTime(WorkHoursUtils.calculateWorkTime(today.getStartTime(),
    today.getEndTime()));
26.         }
27.         workHours.put(calendar.get(Calendar.DATE), today);
28.         monthlyWorkHours.setWorkHours(workHours);
29.         return workHoursRepository.save(monthlyWorkHours);
30.     }
31.
```

Izrezek kode 3.34 Metoda za beleženje zaključka delovnika

Pri beleženju imamo precej podobne metode za beleženje začetka in konca odmorov. Ko želimo začeti odmor, aplikacija preveri, če je uporabnik že začel delovnik, nato pa če ima uporabnik še kakšen aktiven odmor. V primeru neobstoječega vpisa za današnji dan ali aktivnega odmora, vrnemo izjemo. V obratnem primeru dodamo nov odmor v polje odmorov za trenuten dan. V primeru zaključka odmora prav tako preverimo, če je delovnik uporabnika aktiven ter obratno, nato pa kot pri začetku preverimo, če obstaja še aktiven odmor. Če je odmor še aktiven, ga zaključimo, v nasprotnem primeru vrnemo izjemo. Vse začetke in konce seveda beležimo v konzolo aplikacije. Vsi časi začetka in konca se nastavijo kot trenuten čas klica oziroma izvršitve metode.

Uporabnik ima tudi dostop do drugih uporabniških informacij, kot so status uporabnika, trenutno število delovnih ur in število delovnih ur za trenuten teden. Te informacije se uporabljajo znotraj mobilne aplikacije, ki jih grafično predstavlja.

V primeru izračunave trenutnega delovnika (Izrezek kode 3.35) se kot v prejšnjih primerih preveri, če vpis obstaja. Nato se s pomožnim statičnim razredom WorkHoursUtils izračuna delovni čas do trenutka izvršitve metode. Ta razred stori to s pomočjo razreda ChronoUnit, ki nam izračuna razlike med dvema časoma v določeni časovni enoti. V našem primeru so to minute. Če je potrebno, metoda še od delovnega časa odšteje odmore in nato vrne delovni čas v minutah uporabniku.

```
1.  public String calculateCurrentWorkTime(String id) {
2.          Calendar calendar = Calendar.getInstance();
3.          calendar.setTime(new Date());
4.          Optional<MonthlyWorkHours> monthlyWorkHoursOptional =
    workHoursRepository.findById(id + "_" + (calendar.get(Calendar.MONTH) + 1) + "_" +
    calendar.get(Calendar.YEAR));
5.          if (monthlyWorkHoursOptional.isPresent()) {
6.              WorkHours workHours =
    monthlyWorkHoursOptional.get().getWorkHours().get(calendar.get(Calendar.DATE));
7.              if (workHours.getTotalTime() != null) {
8.                  return workHours.getTotalTime().toString();
9.              }
10.             long t = WorkHoursUtils.calculateWorkTime(workHours.getStartTime(),
    LocalDateTime.now());
11.             long breakTime = 0;
12.             if(workHours.getBreaks() != null &&
    workHours.getBreaks().get(workHours.getBreaks().size()-1).getEndTime() !=null){
13.                 for (WorkHoursBreaks breaks : workHours.getBreaks()) {
14.                     breakTime += WorkHoursUtils.calculateWorkTime(breaks.getStartTime(),
    breaks.getEndTime());
15.                 }
16.             }
17.             if (breakTime > 30L) {
18.                 return Long.toString(t + 30L - breakTime);
19.             }
20.             return Long.toString(t);
21.         } else {
22.             throw new EmployeeNotFoundException("Not found");
23.         }
24.     }
25.
```

Izrezek kode 3.35 Metoda za izračunavo trenutnega delovnega časa

V primeru izračuna delovnega časa za tekoči teden (Izrezek kode 3.36) se lahko pojavi težava. Možno je, da so podatki od delovnega časa v dveh dokumentih, saj so lahko datumi znotraj tedna v dveh različnih mesecih. Zato metoda preveri, če je prvi in zadnji dan v tednu znotraj istega meseca. Če je, metoda mapira podatke po dnevu (od ponedeljka do nedelje) znotraj

enega dokumenta in slednje vrne uporabniku. Kot ključ se uporabi enum DayOfTheWeek, ki predstavlja dan v tednu – v nasprotnem primeru mora metoda pridobiti dokument za oba meseca. Pri prvem mesecu nato poiščemo vse dni od ponedeljka do konca meseca, v drugem dokumentu pa vse dni od začetka meseca do nedelje tekočega tedna.

```
1.  public Map<DayOfWeek, Long> getWorkhoursOfCurrentWeek(String id) {
2.          Calendar calendar = Calendar.getInstance();
3.          calendar.setTime(new Date());
4.          MonthlyWorkHours monthlyWorkHoursPrev;
5.          MonthlyWorkHours monthlyWorkHours;
6.          Map<DayOfWeek, Long> hours = new HashMap<>();
7.          // get LocalDate of current weeks monday and Sunday
8.          LocalDate monday =
    LocalDate.now().with(TemporalAdjusters.previousOrSame(DayOfWeek.MONDAY));
9.          LocalDate sunday =
    LocalDate.now().with(TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY));
10.         // if months differ
11.         if (monday.getMonth() != sunday.getMonth()) {
12.             // fetch hours of previous month depending on monday date
13.             Optional<MonthlyWorkHours> monthlyWorkHoursPrevOpt =
    workHoursRepository.findById(id + "_" + monday.getMonth().getValue() + "_" +
    monday.getYear());
14.             // map all dates with dayOfMonth >= monday.dayOfMonth
15.             if (monthlyWorkHoursPrevOpt.isPresent()) {
16.                 monthlyWorkHoursPrev = monthlyWorkHoursPrevOpt.get();
17.                 Map<Integer, WorkHours> mapPrev =
    monthlyWorkHoursPrev.getWorkHours().entrySet().stream().filter(x -> x.getKey() >=
    monday.getDayOfMonth())
18.                         .collect(Collectors.toMap(Map.Entry::getKey,
    Map.Entry::getValue));
19.                 // map key as dayOfWeek, value as totalHours
20.                 mapPrev.forEach((k, v) -> hours.put(LocalDate.of(monday.getYear(),
    monday.getMonth(), k).getDayOfWeek(), v.getTotalTime()));
21.             }
22.             // map all dates with dayOfMonth >= sunday.dayOfMonth
23.             Optional<MonthlyWorkHours> monthlyWorkHoursNextOpt =
    workHoursRepository.findById(id + "_" + sunday.getMonth().getValue() + "_" +
    sunday.getYear());
24.             if (monthlyWorkHoursNextOpt.isPresent()) {
25.                 monthlyWorkHours = monthlyWorkHoursNextOpt.get();
26.                 Map<Integer, WorkHours> mapNext =
    monthlyWorkHours.getWorkHours().entrySet().stream().filter(x -> x.getKey() <=
    sunday.getDayOfMonth())
27.                         .collect(Collectors.toMap(Map.Entry::getKey,
    Map.Entry::getValue));
28.                 // map key as dayOfWeek, value as totalHours
29.                 mapNext.forEach((k, v) -> hours.put(LocalDate.of(sunday.getYear(),
    sunday.getMonth(), k).getDayOfWeek(), v.getTotalTime()));
30.             }
31.             return hours;
32.         }
33.         // if months dont differ, fetch current month hours and map them to hours
34.         Optional<MonthlyWorkHours> monthlyWorkHoursOptional =
    workHoursRepository.findById(id + "_" + (calendar.get(Calendar.MONTH)+1) + "_" +
    calendar.get(Calendar.YEAR));
35.         monthlyWorkHoursOptional.ifPresent(workHours ->
    workHours.getWorkHours().forEach((k, v) ->
    hours.put(LocalDate.of(calendar.get(Calendar.YEAR), calendar.get(Calendar.MONTH),
    k).getDayOfWeek(), v.getTotalTime())));
36.         return hours;
37.     }
```

Izrezek kode 3.36 Metoda za izračun delovnega časa za tekoči teden

S pomočjo tega dela aplikacije lahko preverimo, kakšen status ima v tem trenutku uporabnik. V ta namen smo ustvarili metodo checkStatus (Izrezek kode 3.37), ki kliče dve podmetodi, metodi isBreakActive in isEmployeeWorking, ki vrneta boolean vrednost. V primeru, da vrne ena od njiju true, vrnemo uporabniku odgovor, da je uporabnik na odmoru ali da dela. V nasprotnem primeru pomeni, da uporabnik še ni začel delovnika.

```
1.  public String checkStatus(String id) {
2.          if(isBreakActive(id)) {
3.              return "On break";
4.          } else if (isEmployeeWorking(id)) {
5.              return "Working";
6.          } else {
7.              return "Inactive";
8.          }
9.      }
10.
```

Izrezek kode 3.37 Metoda za preverjanje status uporabnika

Za namene generiranja poročil tukaj uporabljamo še razrede MonthlyReport in MonthlyTask. MonthlyReport je enako kot Workhours implementiran po vzorcu KSR. MonthlyTask (Izrezek kode 3.38) pa je razred, ki je namenjen za implementacijo opravil, ki se naj bi izvršile po določenih intervalih ali času. Z anotacijo »@EnableScheduling« omogočimo znotraj razreda časovno načrtovanje. V trenutni implementaciji imamo le eno metodo, ki je nastavljena tako, da vsak dan ob polnoči kliče metodo iz MonthlyReport storitve, ki skrbi za zaključevanje izmen, če se še niso zaradi kakršnegakoli razloga zaključile ter beleženje časa dela za vse uporabnike kot mesečno poročilo. To smo storili z anotacijo »@Scheduled«, kateri smo podali kot argument cron izraz za izvršitev vsak dan ob polnoči.

```
1.  @EnableScheduling
2.  public class MonthlyTasks {
3.      @Autowired
4.      private MonthlyReportService monthlyReportService;
5.
6.      @Scheduled(cron = "0 0 0 * * ?") // everyday at midnight
7.      public void calculateMonthlyHours() throws Exception {
8.          monthlyReportService.calculateMonthlyHours();
9.      }
10. }
11.
```

Izrezek kode 3.38 Razred MonthlyTask za časovno načrtovanje opravil

Metoda calculateMonthlyHours (Izrezek kode 3.39) glede na trenuten mesec in leto ustvari ali pridobi dokument za mesečno poročilo. Nato pridobi vse dokumente o delovnem času delavcev za trenuten mesec in za vsak dokument sešteje število ur. Pri tem deli ure glede na delovne, ure bolniškega staleža, ure dopusta ter celotno število ur za ta mesec v minutah. Nato te podatke o vseh delavcih shrani v enoten dokument, posamezni podatki o delavcu pa se shranijo pod njegovim identifikacijskim nizom. Ker je možno, da je kateri od delavcev pozabil končati svoj delovni čas, znotraj metode poskrbimo, da se časi zaključijo. Metoda endEntryForWorkDay poskrbi, da se časi ustrezno beležijo. Če je uporabnik pozabil končati delovnik, avtomatsko nastavi kot končni čas osem ur po začetku izmene. V primeru, da ima uporabnik aktivno malico, se beleži kot konec izmene začeten čas malice.

```
1.  public void calculateMonthlyHours() throws Exception {
2.          Calendar calendar = Calendar.getInstance();
3.          calendar.setTime(new Date());
4.          MonthlyReport monthlyReport = new
5.              MonthlyReport("monthlyReport_"+calendar.get(Calendar.MONTH) + "_" +
6.                                          calendar.get(Calendar.YEAR), new HashMap<>());
7.          Map<String, MonthlyHours> monthlyHoursMap = new HashMap<>();
8.          ArrayList<MonthlyWorkHours> monthlyWorkHours =
    workHoursRepository.findAllByUuidContaining((calendar.get(Calendar.MONTH)+1) + "_" +
    calendar.get(Calendar.YEAR));
9.          for (MonthlyWorkHours hours : monthlyWorkHours
10.         ) {
11.             String[] employeeUuid = hours.getUuid().split("_");
12.             workHoursService.endEntryForWorkDay(employeeUuid[0]);
13.             Map<Integer, WorkHours> workHoursMap = hours.getWorkHours();
14.             long work = 0, sick = 0, leave = 0, total = 0;
15.             for (Map.Entry<Integer, WorkHours> entry : workHoursMap.entrySet()) {
16.                 switch (entry.getValue().getWorkHourType()) {
17.                     case WORK -> work += entry.getValue().getTotalTime();
18.                     case LEAVE -> leave += entry.getValue().getTotalTime();
19.                     case SICK_LEAVE -> sick += entry.getValue().getTotalTime();
20.                 }
21.                 total += entry.getValue().getTotalTime();
22.             }
23.             MonthlyHours monthlyHours = new MonthlyHours(employeeUuid[0], work, sick,
    leave, total);
24.             monthlyHoursMap.put(employeeUuid[0],monthlyHours);
25.         }
26.         monthlyReport.setMonthlyHoursMap(monthlyHoursMap);
27.         monthlyReportRepository.save(monthlyReport);
28.     }
29.
```

Izrezek kode 3.39 Metoda za generiranje mesečnih poročil

Znotraj storitvenega razreda imamo prav tako metodo, ki za določenega uporabnika pridobi podatke iz mesečnih poročil za trenutno leto. Te podatke prikazujemo znotraj mobilne

aplikacije, hkrati pa uporabljamo aplikacijo Node-RED za beleženje delovnika, da nam na določene intervale pošilja informacije o napravah na lokalnem omrežju. Aplikacija nam ob vsaki zahtevi poda podatke o napravah, ki so nove na omrežju ter o napravah, ki niso več povezane na omrežje (Izrezek kode 3.40). V ta namen uporabljamo storitev NodeRedService in krmilnik NodeRedController, ki nam je dostopen preko pripone »api/v1/nodered«.

```
1.  @PostMapping("/device")
2.      public ResponseEntity<Void> checkDevices(@RequestBody MacRequest macs) throws
    Exception {
3.          log.info("\nnew devices: {}, \n disconected devices {}"
4.                  ,macs.getNewDevices(), macs.getOldDevices());
5.          nodeRedService.recordTimeByNetworkDevices(macs);
6.          return new ResponseEntity<>(HttpStatus.OK);
7.      }
8.
```

Izrezek kode 3.40 Krmilnik za pridobivanje podatkov od aplikacije Node-RED

Krmilnik kliče metodo recordTimeByNetworkDevices (Izrezek kode 3.41), kateri poda objekt, ki vsebuje dve polji naprav.

```
1.  public void recordTimeByNetworkDevices(MacRequest macRequest) throws Exception {
2.          for(String macAddress: macRequest.getNewDevices()) {
3.              Optional<List<Employee>> employeeList =
    employeeRepository.findEmployeeByDeviceIdMac(macAddress);
4.              if(employeeList.isPresent()) {
5.                  if(workHoursService.isBreakActive(employeeList.get().get(0).getUuid()))
    {
6.                      workHoursService.endBreak(employeeList.get().get(0).getUuid());
7.                  } else {
8.                      workHoursService.addNewEntry(employeeList.get().get(0).getUuid(),
    WorkHourType.WORK);
9.                  }
10.             }
11.         }
12.         for (String macAddress: macRequest.getOldDevices()) {
13.             Optional<List<Employee>> employeeList =
    employeeRepository.findEmployeeByDeviceIdMac(macAddress);
14.             if(!employeeList.isEmpty()) {
15.                 if(workHoursService.isFirstBreak(employeeList.get().get(0).getUuid())) {
16.                     workHoursService.addNewBreak(employeeList.get().get(0).getUuid());
17.                 } else if
    (!workHoursService.isBreakActive(employeeList.get().get(0).getUuid())){
18.                     workHoursService.endEntry(employeeList.get().get(0).getUuid());
19.                 }
20.             }
21.         }
22.     }
23.
```

Izrezek kode 3.41 Metoda za beleženje časa glede na napravo na lokalnem omrežju

Ta metoda najprej preveri vse naslove, ki so se na novo povezali na omrežje. Preveri, na katerega uporabnika je naprava registrirana in če še zaposleni delovnika ni začel, mu začne beležiti delovne ure. V primeru, da je uporabnik šel na odmor izven omrežja in se vrnil, pa mu beleženje odmora konča. Nato preveri še naprave, ki so prekinile povezavo z omrežjem. V primeru, da uporabnik odmora še ni imel, mu začne beležiti odmor, v nasprotnem primeru pa konča beleženje delovnega časa. To smo naredili zato, da v primeru, če uporabnik pozabi beležiti odmor, mu sistem ne konča delovnika avtomatsko. V primeru, da se uporabnik ne vrne, mu sistem ob polnoči zaključi delovnik in mu kot konec delovnika beleži čas začetka nezaključenega odmora.

3.2.7 Upravljanje z izjemami

V ogrodju Spring lahko upravljamo z izjemami zelo elegantno. Ker se pri vsakem klicu našega API-ja lahko izvrši izjema, bi bilo zelo zapravljivo upravljati izjeme znotraj vsake metode krmilnikov. Posledično lahko v te namene uporabljamo t. i. »Advice« razrede, ki so namenjeni za upravljanje izjem za enega ali več krmilnikov ali pa za celotno aplikacijo. Znotraj njih moramo definirati t. i. »Handler« metode, ki se izvršijo ob določeni izjemi in vrnejo uporabniku vnaprej definiran odgovor z določenim statusom HTTP.

```
1.   @ControllerAdvice(assignableTypes = {EmployeeController.class})
2.   public class EmployeeAdvice {
3.      @ResponseBody
4.      @ExceptionHandler(EmployeeNotCreatedException.class)
5.      @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
6.      String employeeNotCreatedHandler(EmployeeNotCreatedException ex){
7.          return ex.getMessage();
8.      }
9.
10.     @ResponseBody
11.     @ExceptionHandler(EmployeeNotFoundException.class)
12.     @ResponseStatus(HttpStatus.NOT_FOUND)
13.     String employeeNotFoundHandler(EmployeeNotFoundException ex) {
14.         return ex.getMessage();
15.     }
16.
17.
18.     @ResponseBody
19.     @ExceptionHandler(ConstraintViolationException.class)
20.     @ResponseStatus(HttpStatus.BAD_REQUEST)
21.     String constraintViolationExceptionHandler(ConstraintViolationException ex) {
22.         return ex.getMessage();
23.     }
24.
25.     @ResponseBody
26.     @ExceptionHandler(DeviceAlreadyAssignedException.class)
27.     @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
28.     String deviceAlreadyAssignedExceptionHandler(DeviceAlreadyAssignedException ex) {
29.         return ex.getMessage();
30.     }
31.
32.     @ResponseBody
33.     @ExceptionHandler(DeviceNotFoundException.class)
34.     @ResponseStatus(HttpStatus.NOT_FOUND)
35.     String deviceNotFoundExceptionHandler(DeviceNotFoundException ex) {
36.         return ex.getMessage();
37.     }
38. }
```

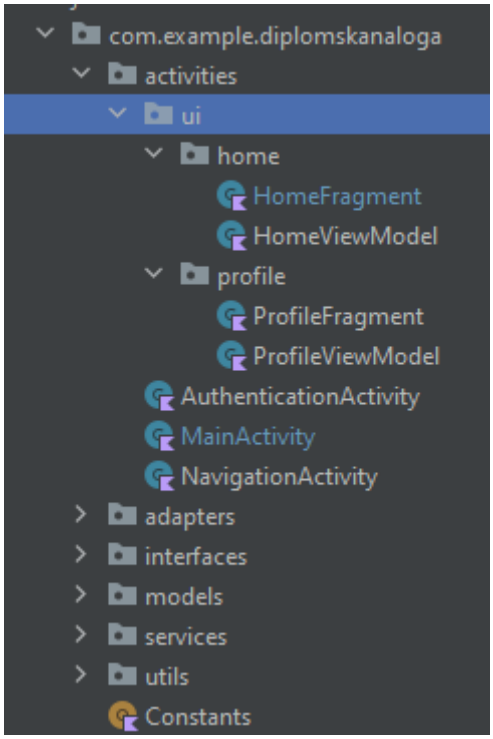
Izrezek kode 3.42 EmployeeAdvice razred za uporavljanje z izjemami

Za naš krmilnik EmployeeController lahko v Izrezek kode 3.42 vidimo, kako izgleda takšen advice razred. Tukaj imamo tudi nekaj po meri narejenih izjem za primere, kadar uporabnik ali naprava ne obstajata, če uporabnika ni mogoče najti, kadar uporabnika ni mogoče ustvariti in kadar je naprava že registrirana na obstoječega uporabnika. Da metoda znotraj Advice razreda ve, kdaj naj izvrši katero metodo, moramo pri metodi z anotacijo »@ExceptionHandler« definirati izjemo, ob kateri se naj metoda izvrši in z anotacijo »@ResponseStatus« podati status kodo, katero naj vrne. Z anotacijo »@ResponseBody« povemo, da se naj sporočilo vrne znotraj telesa odgovora.

Ko pride do katerih od teh izjem znotraj krmilnika EmployeeController, to Advice razred prepozna in glede na izjemo vrne odgovor s sporočilom in status kodo HTTP.

3.3 Razvoj mobilne aplikacije za platformo Android

Pri razvoju mobilne aplikacije smo želeli omogočiti avtentikacijo, pregled delovnega časa, manualno beleženje delovnika in odmorov ter urejanje osebnih podatkov. Za vizualni prikaz podatkov smo v aplikacijo vključili MPAndroidChart knjižnico, za komunikacijo s strežnikom pa uporabili knjižnico Volley. Aplikacija sama je bila napisana v jeziku Kotlin.

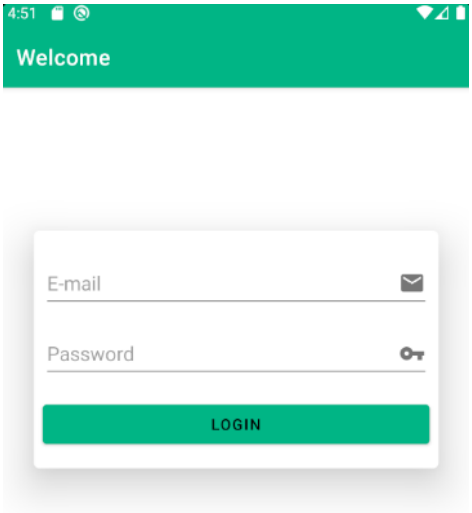


Slika 3.7 Projektna struktura mobilne aplikacije

Sama aplikacija je sestavljena iz Activity razredov (kateri vsebujejo grafični vmesnik in logiko, povezano z vmesnikom), adapterjev, vmesnikov, modelov, storitev in pomožnih razredov (Slika 3.7).

3.3.1 Avtentikacija uporabnika

Aplikacija je sestavljena tako, da brez pravilnih uporabniških podatkov do njenih funkcionalnosti ni mogoče dostopati. Zato moramo ob vstopu v aplikacijo vpisati uporabniške podatke, s katerimi se uporabnik avtenticira (Slika 3.8). Če je to uspešno, pridobi aplikacija s strani API-ja uporabniški žeton JWT, katerega shranimo v SharedPreferences. To se stori zato, da ob vsakem vstopu ni potrebno vpisati uporabniških podatkov (Izrezek kode 3.43).



Slika 3.8 Prijavni obrazec v mobilni aplikaciji

```
1.  fun isUserLoggedIn() {
2.      val sharedPreferences = this.getSharedPreferences(getString
3.              (R.string.preference_file_key), Context.MODE_PRIVATE)
4.          // if user isn't logged in forward to auth activity else dashboard
5.          if (!sharedPreferences.contains(getString(R.string.preference_token))) {
6.              openAuth()
7.          } else {
8.              employeeRestService.getUserData(this, object : VolleyResponse {
9.                  override fun onSuccess(response: Any?) {
10.                     userData = gson.fromJson(response.toString(), Employee::class.java)
11.                     Log.w("Response", userData.toString())
12.                     openHome()
13.                 }
14.
15.                 override fun onError(error: VolleyError?) {
16.                         openAuth()
17.                 }
18.             })
19.         }
20.     }
```

Izrezek kode 3.43 Metoda za preverjanje prisotnosti prijavnih podatkov

V primeru, da žeton JWT ni shranjen ali ni več veljaven, razred MainActivity preusmeri uporabnika v AuthenticationActivity, v katerem lahko uporabnik vpiše njegov elektronski naslov in geslo.

```
1.  fun login(view: View) {
2.          val usernameText = findViewById<EditText>(R.id.inputEmail).text.toString()
3.          val passwordText = findViewById<EditText>(R.id.inputPassword).text.toString()
4.          val loginCridentials = LoginCridentials(usernameText, passwordText)
5.          employeeRestService.login(this, loginCridentials, object: VolleyResponse {
6.              override fun onSuccess(response: Any?) {
7.                  openMainActivity()
8.              }
9.              override fun onError(error: VolleyError?) {
10.             }
11.         })
12.     }
13.
```

Izrezek kode 3.44 Metoda za prijavo uporabnika v aplikacijo

Ob potrditvi uporabniških podatkov se izvrši login metoda (Izrezek kode 3.44), katera iz elementov pridobi elektronski naslov in geslo uporabnika, ter jih združi v objekt, da bi lahko lažje serializirali te podatke v niz JSON. Ta objekt metoda posreduje razredu EmployeeService, ki z metodo login preko knjižnice Volley avtenticira uporabnika preko API klica (Izrezek kode 3.45).

```
1.    fun login(context: Context, credentials: LoginCridentials, callback: VolleyResponse) {
2.          var toast = Toast.makeText(context, "", Toast.LENGTH_SHORT)
3.          HttpsTrustManager.allowAllSSL()
4.          val data = JSONObject(gson.toJson(credentials))
5.          val loginUserRequest = JsonObjectRequest(
6.              Request.Method.POST,
7.              Constants.baseUrl + "/authenticate",
8.              data,
9.              Response.Listener { response ->
10.                 toast.setText("LOGIN SUCCESSFUL")
11.                 toast.show()
12.                 saveToken(
13.                     response.getString("jwttoken"),
14.                     response.getString("refreshToken"),
15.                     context
16.                 )
17.                 callback.onSuccess(response)
18.             },
19.             { error ->
20.                 toast.setText("LOGIN UNSUCCESSFUL")
21.                 toast.show()
22.                 callback.onError(error)
23.             })
24.         RestQueueService.getInstance(context).addToRequestQueue(loginUserRequest)
25.     }
26.
```

Izrezek kode 3.45 Klic HTTP avtentikacijske kode v mobilni aplikaciji

Metoda login pridobljen podatek pretvori v niz JSON in ga preko POST klica pošlje strežniku za avtentikacijo. V primeru, da so podatki pravilni, pridobimo žeton JWT, katerega preko pomožne metode shranimo v SharedPreferences s ključem “jwt”. Nato preusmeri uporabnika v NavigationActivity, ki uporabniku prikaže HomeFragment, ki vsebuje razne grafično vizualizirane podatke o delovniku uporabnika. V primeru neuspešne prijave se s pomočjo Toast elementa prikaže sporočilo o neuspešnosti.

Ker za komunikacijo preko protokola HTTPS uporabljamo svoj certifikat, ki smo ga sami ustvarili in podpisali, nastane pri Android aplikacijah manjši problem. Privzeto Android omogoča le uporabo certifikatov, ki so bili podpisani preko pristojnih ustanov – če pa bi želeli uporabiti svojega, pa je slednje potrebno omogočiti ročno. To smo v našem primeru uredili preko razreda HttpTrustManager in metode allowAllSSL, katero je potrebno klicati ob vsaki izvršitvi zahteve HTTPS. Slednjo smo implementirali po rešitvi s spletne strani StackOverflow uporabnika MaxMxx [24]. Če to ne bi storili, nam z našim certifikatom ne bi bilo mogoče izdelati zahteve preko protokola HTTPS. Čeprav rešitev deluje, bi bila precej nevarna za uporabo izven lokalnega razvoja, saj nam metoda sama omogoči ustvarjanje zahtev, certifikata pa ne validira, zato se zaradi varnostnih razlogov ne bi uporabila v produkciji. To

rešitev smo implementirali zato, da smo lahko razvili rešitev brez uradnega SSL certifikata. Ker knjižnica Valley za pošiljanje zahtev uporablja t. i. RequestQueue (Izrezek kode 3.46), je potrebno vsako zahtevo, katero želimo poslati, podati v to vrsto, ki nato zahteve izvrši.

```
1.  class RestQueueService constructor(context: Context) {
2.      companion object {
3.          @Volatile
4.          private var INSTANCE: RestQueueService? = null
5.          fun getInstance(context: Context) = INSTANCE ?: synchronized(this) {
6.              INSTANCE ?: RestQueueService(context).also { INSTANCE = it }
7.          }
8.      }
9.      val requestQueue: RequestQueue by lazy {
10.         Volley.newRequestQueue(context.applicationContext)
11.     }
12.     fun <T> addToRequestQueue(req: Request<T>) {
13.         requestQueue.add(req)
14.     }
15. }
16.
```

Izrezek kode 3.46 Singleton razred RestQueueService

Na koncu vsake metode za pošiljanje zahtev kličemo singleton razred RequestQueueService. Ker je razred singleton, imamo za celo aplikacijo eno samo instanco naše vrste RequestQueue in se izognemo ponovni inicializaciji RequestQueue-ja ob vsakem klicu HTTP. V kotlinu se to uredi s companion objektom, saj kotlin sam ne vsebuje “static” oznake. Implementacijo tega razreda smo izdelali po primeru iz uradne strani knjižnice.

Ker moramo pri vseh zahtevah razen avtentikacije podati v glavi zahteve žeton JWT, ga moramo tudi na določen način dodati v Valley zahtevo. To storimo tako, da prepišemo getHeader metodo (Izrezek kode 3.47) posamezne zahteve. Namen te metode je, da vrne vse dodatne vrednosti glave zahteve, ki jih moramo poslati z zahtevo.

```
1.  @Throws(AuthFailureError::class)
2.              override fun getHeaders(): Map<String, String> {
3.                  val headers = HashMap<String, String>()
4.                  headers["Authorization"] = "Bearer $jwt"
5.                  return headers
6.              }
```

Izrezek kode 3.47 Metoda za dodajanje žetona JWT v glavo zahteve

V projektu smo tudi ustvarili vmesnik VolleyResponse (Izrezek kode 3.48), ki nam pomaga pri obdelavi zahtev izven storitvenih razredov. To smo storili zato, ker po klicu Volley zahteve

odgovor ne moremo vrniti kot rezultat klica funkcije. Da bi lahko pridobljene podatke lažje pridobili znotraj Activity razreda in jih grafično prikazali, je bila potrebna implementacija vmesnika.

```
1.  interface VolleyResponse {
2.      fun onSuccess(response: Any?)
3.      fun onError(error: VolleyError?)
4.  }
5.
```

Izrezek kode 3.48 Vmesnik VolleyResponse

```
1.  fun getWeeklyReport(root: View) {
2.          workHoursRestService.getWeeklyReport(this.requireContext(), object :
    VolleyResponse {
3.              override fun onSuccess(response: Any?) {
4.                  val type = object : TypeToken<Map<String, Long>>() {}.type
5.                  weeklyReport = gson.fromJson(response.toString(), type)
6.                  initWeeklyBarChart(weeklyReport)
7.              }
8.
9.              override fun onError(error: VolleyError?) {
10.                 if (error != null) {
11.                     Log.e("Response Error", error.localizedMessage.toString())
12.                 }
13.             }
14.         })
15.     }
16.
```

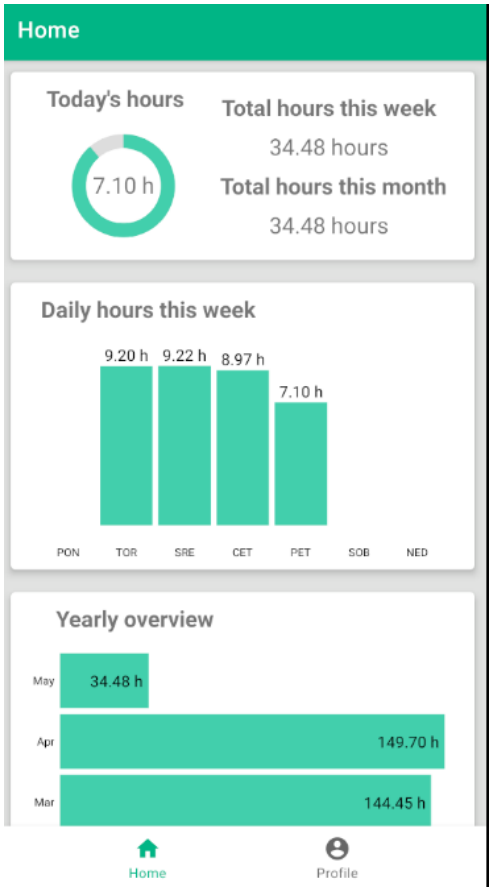
Izrezek kode 3.49 Primer obdelave uspešnega in neuspešnega Volley klica

V metodi iz Izrezek kode 3.49 s pomočjo tega vmesnika v getWeeklyReport metodi prepišemo klica onSuccess in onError, ki Volley izvrši ob uspešnem ali neuspešnem klicu HTTP. Metodi vsebujeta kot argument odgovor zahteve, katerega lahko nato po želji obdelamo.

3.3.2 Prikaz in urejanje podatkov

Prijavljenemu uporabniku se ob odprtju aplikacije vedno prikažejo podatki o njegovem delovnem času. S tem namenom smo uporabili NavigationActivity. Ta vsebuje v spodnjem delu aplikacije dva zavihka, za domačo stran s podatki o delovniku in za uporabnikov profil z njegovimi osebnimi podatki. Da imamo lahko znotraj enega Activity razreda dva različna pogleda, smo uporabili šablono za Navigation Drawer Activity, ki vsebuje zavihke, ter

Fragment razrede, ki predstavljajo vsak svoj pogled. Posledično imamo v projektu HomeFragment (Slika 3.9) za domačo stran in ProfileFragment (Slika 3.10) za stran o profilu uporabnika.



Slika 3.9 Domača stran mobilne aplikacije

Na domači strani ima uporabnik tri predele. Prvi predel vsebuje grafični prikaz o delovnih urah za trenuten dan in informacije o številu delovnih ur za trenuten teden ter mesec. Nato vidimo dva predela s stolpčnimi diagrami. Prvi prikazuje delovni čas za posamezen dan trenutnega tedna, drugi pa celoten čas delovnih ur za posamezni mesec.

Prikaz o trenutnem delovnem času smo uredili s po meri izdelanim elementom, ki predstavlja prazen krog in ProgressBar elementom. V primeru diagramov smo jih programsko ustvarili s pomočjo knjižnice MPAndroidChart. V primeru te knjižnice moramo element le dodati v želeno »layout« datoteko in ta element lahko nato znotraj kode grafično uredimo ter povežemo nanj podatke, ki jih želimo prikazati.

Za pridobivanje podatkov na začetni strani naredi HomeFragment preko Volley razreda tri zahteve HTTP. Te so za pridobivanje trenutnega števila delovnih ur (Izrezek kode 3.50) ter za tedenski in mesečni pregled. Slednje opravimo z metodami znotraj WorkHoursRestService razreda, ki deluje po zelo podobnem principu, kot že prej prikazano za avtentikacijo. Vse te zahteve obdelamo, kot že omenjeno, s pomočjo VolleyResponse vmesnika in različnih klicev HTTP. Edina večja razlika med klici za prijavo uporabnika in drugimi zahtevami je, da se pri teh doda v glavo zahteve žeton JWT.

```
1.  fun getCurrentTime(root: View) {
2.          workHoursRestService.getCurrentWorkTime(this.requireContext(), object :
    VolleyResponse {
3.              override fun onSuccess(response: Any?) {
4.                  var temp = response as String
5.                  val time = temp.toBigDecimal().setScale(2,
    RoundingMode.HALF_EVEN).div(BigDecimal(60))
6.                  currentWorkHoursTextView.text = "$time " + "h"
7.                  dailyProgressBar.progress = temp.toInt()
8.                  Log.e("Response Error", response.toString())
9.              }
10.
11.             override fun onError(error: VolleyError?) {
12.                 if (error != null) {
13.                 }
14.             }
15.         })
16.     }
```

Izrezek kode 3.50 Pridobivanje podatkov o trenutnem delovnem času

Za prikaz podatkov o trenutnem delovnem času se izvrši klic HTTP, ki nam vrne delovni čas v minutah kot niz. Ta niz pretvorimo v decimalno število, katero s pomočjo BigDecimal razreda zaokrožimo na dve decimalki. Izračunan podatek nato znotraj praznega kroga prikažemo kot niz. Elementu dailyProgressBar podamo vrednost v minutah in glede na to vrednost napolnimo krog. Maksimalna vrednost tega elementa je nastavljena na 480, kar predstavlja 8 ur. Če čas presega to vrednost, ostane krog v celoti pobarvan, točen čas pa lahko vedno vidimo znotraj tega kroga.

Za grafe moramo prav tako najprej pridobiti podatke preko zahtev HTTP. Pri tem moramo po pridobitvi podatkov le-te urediti v pravilno obliko, inicializirati graf ter povezati podatke.

V primeru stolpčnega grafa (Izrezek kode 3.51) moramo podatke pretvoriti v BarEntry objekte.

```
1.  fun initWeeklyBarChart(values: Map<String, Long>) {
2.          val entries: MutableList<BarEntry> = ArrayList()
3.          Constants.WEEKDAY_LABEL.forEach { v ->
4.              if (values[v] != null) {
5.                  val fVal = values.get(v)!!.toFloat()
6.                  var time: Float = fVal.div(60f)
7.                  entries.add(BarEntry(Constants.WEEKDAY_LABEL.indexOf(v).toFloat(),
    time))
8.              } else {
9.                  entries.add(BarEntry(Constants.WEEKDAY_LABEL.indexOf(v).toFloat(), 0f))
10.             }
11.         }
12.         val valueFormatter = ChartValueFormatter(
13.             "",
14.             " h",
15.             Constants.MONTHS_LABEL
16.         )
17.         val barData = BarDataSet(entries, null)
18.         barData.color = resources.getColor(R.color.secondaryColor)
19.         val data = BarData(barData)
20.         data.setValueFormatter(valueFormatter)
21.         data.barWidth = 0.9f
22.         val chart: BarChart = ChartUtils.setBarChartDefaults(barChart)
23.         chart.data = data
24.         chart.animateXY(1000, 1000)
25.         // set text size of bar values
26.         chart.data.setValueTextSize(14f)
27.
28.         chart.animate()
29.         var totalMin: Long = 0
30.         for ((key, value) in values) {
31.             totalMin += value
32.         }
33.         var totalHour  = totalMin.toBigDecimal().setScale(2,
    RoundingMode.HALF_EVEN).div(BigDecimal(60))
34.         hoursThisWeekTextview.text = "$totalHour hours"
35.     }
36.
```

Izrezek kode 3.51 Inicializacija grafa iz knjižnice MPAndroidChart

V primeru podatkov o tednu pridobimo podatke v obliki Map objekta. Podatki so v tem objektu shranjeni pod ključem, ki je krajšava za posamezni dan v tednu. Podatke pretvorimo tako, da gremo skozi celotni objekt in če vsebuje vrednost, jo prepišemo v BarEntry, v nasprotnem primeru pa vzamemo vrednost 0. Te podatke nato med seboj seštejemo in jih prikažemo ob trenutnem delovnem času. Vsak BarEntry objekt vsebuje dve vrednosti. Prva vrednost predstavlja pozicijo v grafu oziroma vrstni red, druga pa dejansko vrednost podatka, ki ga želimo prikazati. Nato povežemo podatke z grafom, definiramo barve, širino grafov, animacije, velikost pisave ter ga grafično uredimo s pomočjo statičnih metod razreda ChartUtils (Izrezek kode 3.52). Metode znotraj tega razreda poskrbijo, da so grafi čim bolj minimalistični. Te skrijejo vse nepotrebne mreže in črte znotraj grafa, nepotrebne oznake in definirajo kje in kakšne bodo oznake. Po želji lahko tudi pokažemo ali skrijemo legende grafov.

```
1.  fun setBarChartDefaults(chart: BarChart): BarChart {
2.              // hide all lines of chart
3.              chart.axisLeft.setDrawLabels(false)
4.              chart.axisLeft.setDrawGridLines(false)
5.              chart.axisLeft.setDrawAxisLine(false)
6.              chart.axisRight.setDrawLabels(false)
7.              chart.axisRight.setDrawGridLines(false)
8.              chart.axisRight.setDrawAxisLine(false)
9.              chart.xAxis.setDrawAxisLine(false)
10.             chart.xAxis.setDrawGridLines(false)
11.             // set labels for bars
12.             chart.xAxis.granularity = 1f
13.             chart.xAxis.isGranularityEnabled = true
14.             chart.xAxis.setCenterAxisLabels(false)
15.             chart.xAxis.position = XAxis.XAxisPosition.BOTTOM
16.             chart.xAxis.valueFormatter =
    IndexAxisValueFormatter(Constants.WEEKDAY_LABEL_SLO)
17.             // hide description
18.             chart.description.text = ""
19.             // hide legend
20.             chart.legend.isEnabled = false
21.             chart.setFitBars(false)
22.             return chart
23.         }
24.
```

Izrezek kode 3.52 Metoda za grafično urejanje grafov

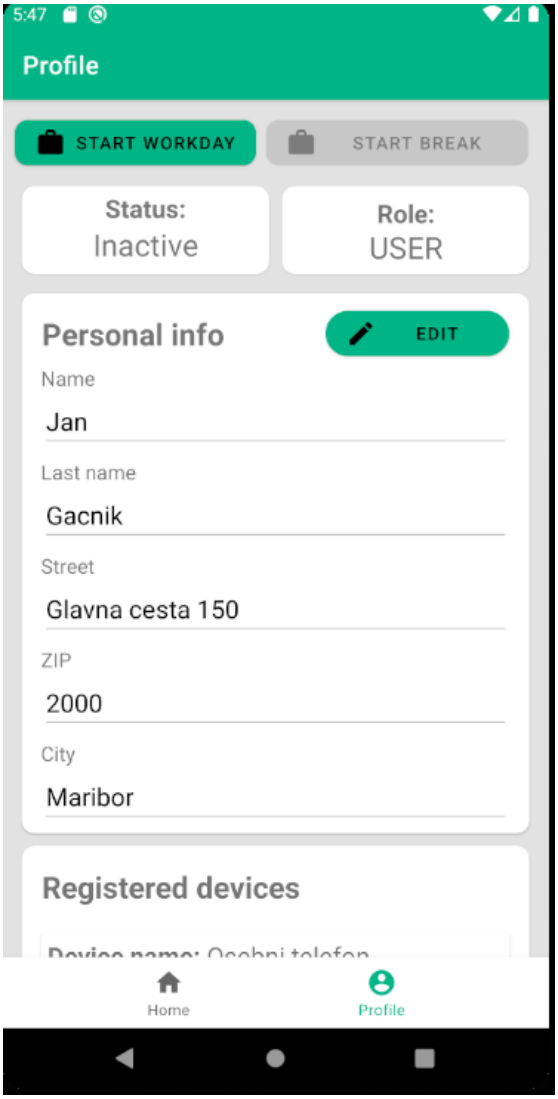
Da smo lahko oznake še bolj priredili svojim potrebam, smo napisali svoj pretvornik za oznake (Izrezek kode 3.53).

```
1.  class ChartValueFormatter(val valuePrefix: String, val valueSuffix: String,val
    labelValues: List<String>) : ValueFormatter() {
2.      private var mFormat: DecimalFormat? = null
3.
4.      init {
5.          mFormat = DecimalFormat("###,###,##0.00")
6.      }
7.
8.      override fun getFormattedValue(value: Float): String {
9.          if(value <= 0f) {
10.             return ""
11.         }
12.         return this.valuePrefix + mFormat!!.format(value)  + this.valueSuffix
13.     }
14.
15.     override fun getAxisLabel(value: Float, axis: AxisBase): String {
16.           return this.labelValues[value.toInt()]
17.     }
18. }
19.
```

Izrezek kode 3.53 Pretvornik oznak za grafe

Naš pretvornik poskrbi za to, da so vrednosti znotraj grafov omejene na dve decimalki, s čimer povemo, da so podatki v urah, ter na koncu vseh dodamo črko »h«. V primeru, da je vrednost 0, je ne prikažemo. Ta razred prav tako poskrbi, da se pridobijo pravilne oznake iz polja oznak, katerega podamo grafu.

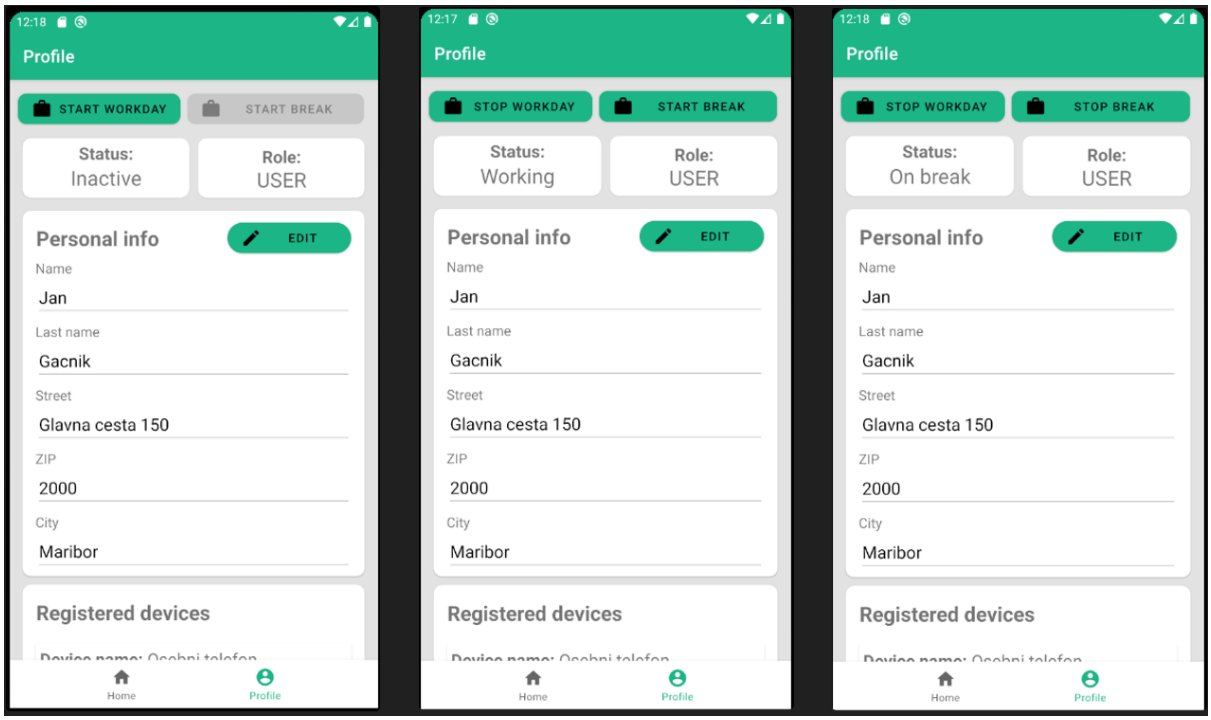
V zavihku Profile (Slika 3.10) prikazujemo le osebne podatke uporabnika ter njegove vloge in trenutni status. Uporabnik lahko tukaj tudi uredi svoje osebne podatke ter ročno začne in konča delo oziroma odmor. Te podatke pridobimo s strani naše Spring Boot aplikacije.



Slika 3.10 Stran profila uporabnika v mobilni aplikaciji

Uporabnik lahko ima trenutno tri različne statuse: »Working«, »On break« in »Inactive« (Slika 3.11). Prvi nam pove, da uporabnik trenutno dela, drugi da je uporabnik na odmoru in tretji da uporabnik ne dela. Ob statusu uporabnika vidimo tudi njegove dodeljene vloge. V našem

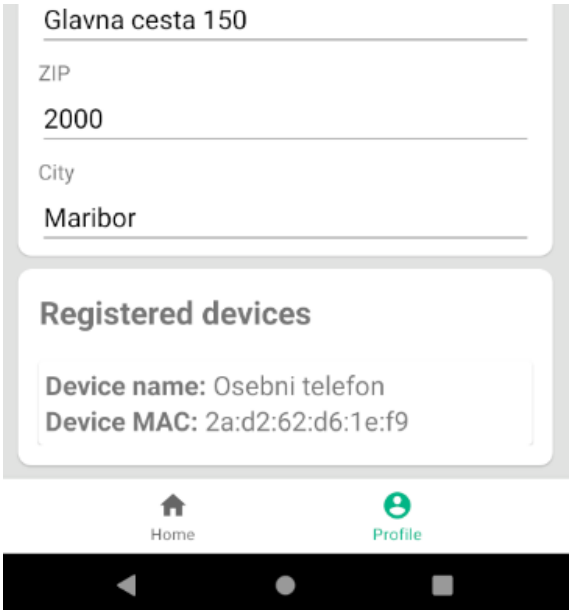
primeru ima vsak uporabnik vsaj vlogo »USER«, kar pomeni, da je navadni uporabnik. Če bi bil administrator, bi mu lahko tudi dodali vlogo »ADMIN«. Nismo omejeni na le ta dva tipa, saj lahko za naše potrebe definiramo vloge, kolikor jih želimo. Podatek o vlogah uporabnika pridobimo od strežnika skupaj z njegovimi osebnimi podatki. Ob teh podatkih ima uporabnik gumb »Edit«, s pritiskom katerega lahko nato ureja svoje podatke. Ob ponovnem pritisku nanj pa se urejeni podatki pošljejo spletni aplikaciji, ki podatke posodobi.



Slika 3.11 Različne možnosti in status uporabnika znotraj aplikacije

Čisto na vrhu opazimo še dva gumba, »START WORKDAY« in »START BREAK« (Slika 3.11). Namenjena sta ročnemu beleženju v primeru dela na terenu ali od doma. Prvi začne oziroma konča delovni dan, drugi pa je namenjen beleženju odmorov. Seveda uporabnik ne more začeti odmora, če se njegov delovni dan še ni začel. Komaj po začetku delovnika je uporabniku omogočeno beleženje odmorov. Ob pritisku posameznega gumba aplikacija preveri status uporabnika in glede na to pošlje spletni aplikaciji zahtevo ter spremeni besedilo na gumbu. Ob uspešnem klicu nato onemogoči ali omogoči gumb »START BREAK« ter posodobi status uporabnika znotraj mobilne aplikacije.

Pod osebnimi podatki še najdemo seznam vseh naprav, ki so registrirane pod uporabnikom (Slika 3.12). Za vsako napravo lahko uporabnik vidi vzdevek ali krajši opis ter MAC naslov naprave. Prav tako smo nameravali uporabniku še omogočiti možnost registracije mobilnika znotraj mobilne aplikacije. Žal pa je to v operacijskem sistemu Android omogočeno le sistemskim aplikacijam in zaradi tega te funkcionalnosti nismo mogli implementirati.



Slika 3.12 Prikaz registriranih naprav uporabnika

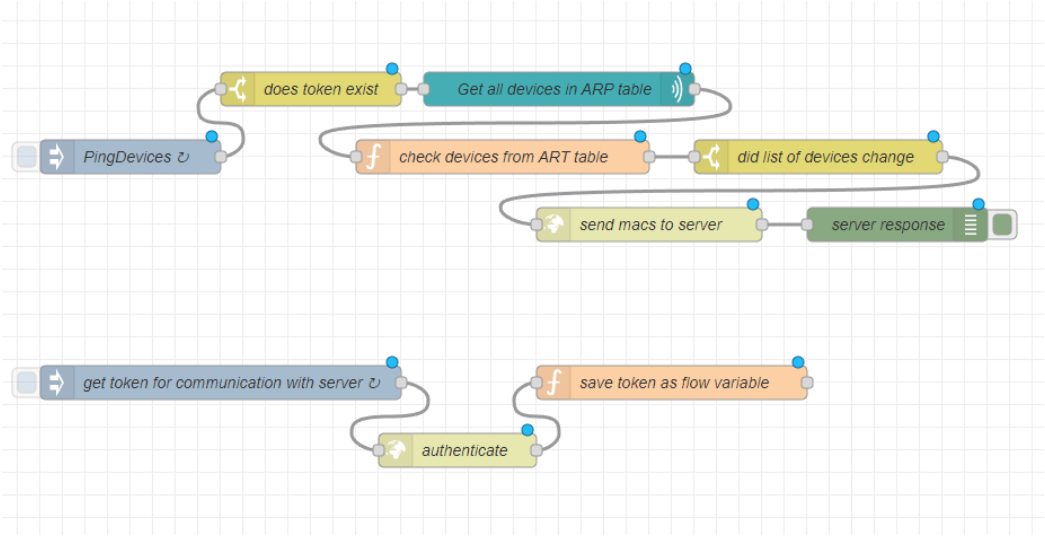
3.4 Integracija v okolje Node-RED

Odločili smo se, da bomo aplikacijo Node-RED namestili na računalnik Raspberry Pi. Zato smo na računalnik namestili operacijski sistem Raspberry Pi OS in nato v operacijskem sistemu še Node.js in Node-RED. Po namestitvi Node-RED-a lahko zaženemo aplikacijo preko terminala z ukazom »Node-RED-start«, za zaustavitev aplikacije ali ponovni zagon pa lahko uporabimo ukaz »Node-RED-stop« ali »Node-RED-restart«. Če bi želeli, da se aplikacija ob zagonu sistema zažene sama, lahko to omogočimo z ukazom »sudo systemctl enable nodered.service«. Ko je aplikacija aktivna, lahko odpremo brskalnik in preko povezave »http://localhost:1880« začnemo z ustvarjanjem tokov. Rutina je določen potek operacij, ki si sledijo ena za drugo. Sestavljene so iz različnih elementov, imenovanih vozlišča, ki imajo vsak svojo funkcijo.

V našem primeru uporabljamo naslednja vozlišča:

- inject vozlišče (pošlje sporočilo v tok manualno ali na določen interval),
- switch vozlišče (preusmeri tok glede na določeno vrednost in pogoj),
- ARP vozlišče (iz knjižnice Node-RED uporabnika fchanson. Pridobi ARP tabelo za trenutno lokalno omrežje),
- function vozlišče (omogoča pisanje funkcij v jeziku JavaScript),
- request vozlišče HTTP (omogoča pošiljanje zahtev HTTP),
- debug vozlišče (uporablja se za razhroščevanje).

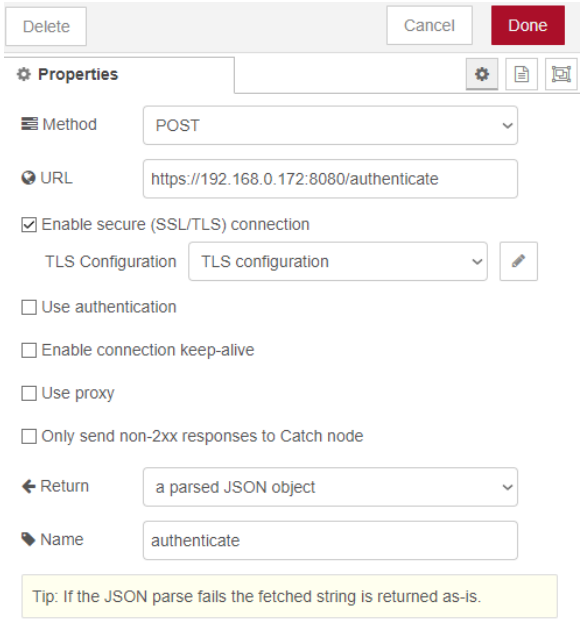
Za našo rešitev uporabljamo dejansko le dva toka (Slika 3.13). En je za pridobivanje žetona JWT, tako da lahko aplikacija komunicira z zaledjem, drugi tok pa se uporablja za pridobivanje naprav, povezanih na omrežje.



Slika 3.13 Uporabljeni toki Node-RED

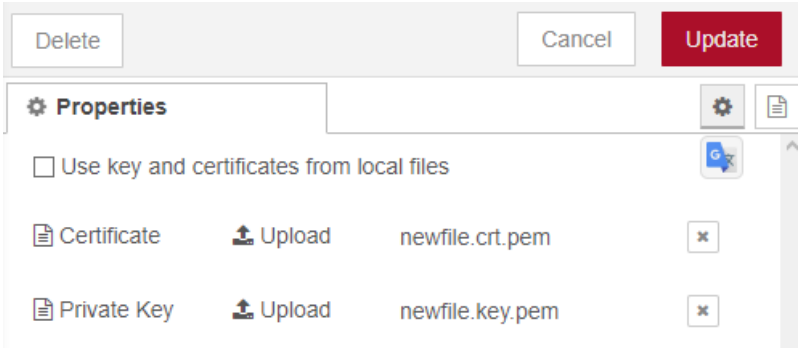
3.4.1 Avtentikacije

Ker lahko pošiljamo zahteve na naš API le z veljavnim žetonom JWT za aplikacijo Node-RED, moramo še pridobiti veljaven žeton. Zato smo vstavili dodaten tok, katerega smo implementirali tako, da vsak dan naredi API klice za avtentikacijo, in imamo vedno veljaven žeton. Za izvršitev tega toka poskrbi inject vozlišče, ki vsakih 24 ur pošlje http request vozlišču prijavne podatke, ustvarjene specifično le za aplikacijo Node-RED. V tem vozlišču moramo izbrati pravilni tip zahteve, url in omogočiti TLS, saj morajo biti naše zahteve narejene vedno preko protokola HTTPS (Slika 3.14).



Slika 3.14 Nastavitve request vozlišča HTTP za avtentikacijo

Da omogočimo varno povezavo preko klica, moramo v TLS nastavitvah podati naš certifikat in privaten ključ (Slika 3.15). Zato potrebujemo naš certifikat in zasebni ključ. Te lahko s pomočjo keygen orodja in našega .p12 certifikata generiramo ter jih vstavimo v to vozlišče. Slednje moramo opraviti tudi za vse druge zahteve HTTP znotraj drugih tokov.



Slika 3.15 TLS nastavitve za zahteve HTTP znotraj aplikacije Node-RED

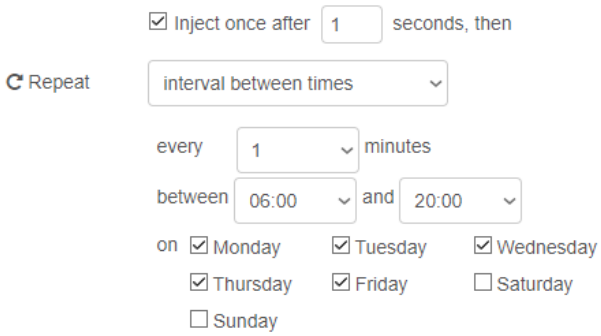
Ko pridobimo žeton JWT od API klica, ga s pomočjo function vozlišča s preprosto JavaScript kodo shranimo kot lokalno spremenljivko »token«, ki je dostopna vsem tokom (Izrezek kode 3.54).

```
1.  flow.set("token",msg.payload);
2.  node.warn(flow.get("token",msg.payload));
3.  return msg.payload;
```

Izrezek kode 3.54 Metoda za shranjevanje žetona JWT v lokalno spremenljivko

3.4.2 Preverjanje naprav na lokalnem omrežju

Preverjanje naprav na lokalnem omrežju sprožimo z inject vozliščem v določenem intervalu. Pri tem lahko določimo, na kateri dan v tednu bo to aktivno, med katerimi urami naj se stvari izvršijo ter na koliko minut. V našem primeru smo to nastavili tako, da se izvrši vsako minuto med 6.00 in 20.00 od ponedeljka do petka (Slika 3.16). Sporočilo v tem primeru ni pomembno, zato pošljemo naprej le trenuten čas, kar je privzeta vrednost.



Slika 3.16 Nastavitev intervala inject vozlišča

Prva stvar, ki jo ta tok naredi je preverjanje s pomočjo switch vozlišča, če lokalna spremenljivka »token« obstaja, saj v njej shranjujemo naš žeton JWT. Če spremenljivka obstaja, se izvrši ARP vozlišče, ki nam pridobi tabelo ARP, v kateri najdemo vse naslove MAC naprav na lokalnem omrežju.

Koda iz Izrezek kode 3.55 v novo polje zapiše vse naslove MAC, pridobljene od vozlišča ARP. Nato pridobi lokalno spremenljivko lastSavedDevices, v kateri so shranjene vse naprave zadnjega preverjanja ARP tabele. V primeru, da spremenljivka ne obstaja, shrani vse naslove kot nove naprave v omrežju in jih shrani v objekt JSON, kateri se nato poda naprej za pošiljanje k Spring Boot aplikaciji. V primeru, da spremenljivka obstaja, se preveri prvo, če se je seznam naprav spremenil od zadnjega klica. Če se ni spremenil, ne pošlje zahteve API, saj to ni potrebno. V drugem primeru s primerjanjem starega in novega polja naprav pridobimo polje novih naprav in naprav, ki so prekinile povezavo z omrežjem. Ta polja shranimo v skupen objekt JSON in ga podamo naprej v tok, hkrati pa pri vseh nastavimo v glavo zahteve naš žeton JWT.

Naslednje vozlišče preveri, če se je seznam naprav spremenil in v primeru, da se je, pošljemo podatke o tem aplikaciji Spring Boot. Za preverjanje uspešnosti te zahteve na koncu s pomočjo debug vozlišča izpišemo odgovor zahteve.

```
var netDevices = [];
for(var device of msg.payload){
    netDevices.push(device.mac);
}
const savedDevices = flow.get("lastSavedDevices");
netDevices = netDevices.sort();
if(savedDevices) {
    if(JSON.stringify(savedDevices) !== JSON.stringify(netDevices)){
        // get new devices connected to LAN/WAN
        const newDevices = netDevices.filter(device => !savedDevices.includes(device));
        // get devices that left LAN/WAN
        const oldDevices = savedDevices.filter(device => !netDevices.includes(device));
        node.warn("new " + newDevices.toString());
        node.warn("old " + oldDevices.toString());

        flow.set("lastSavedDevices", netDevices);
        msg.payload = {};
        msg.payload = {newDevices: newDevices, oldDevices: oldDevices};
        msg.headers = {};
        msg.headers["Authorization"]="Bearer " + flow.get("token").jwttoken;
        node.warn(msg);
        return msg;
    }
} else {
    flow.set("lastSavedDevices", netDevices);

        msg.payload = {};
        msg.payload = {newDevices: netDevices, oldDevices: []};
        msg.headers = {};
        msg.headers["Authorization"]="Bearer " + flow.get("token").jwttoken;
    return msg;
}
return {payload:{}};

```

Izrezek kode 3.55 Funkcija za preverjanje naprav na lokalnem omrežju

4 PRIMERJAVA REŠITEV

Delovne ure lahko evidentiramo na različne načine. Najbolj preprost način je izpolnjevanje časovnika na list papirja ali digitalno tabelo, kjer vsak posameznik dnevno vpisuje čase prihodov in odhodov iz delovnega mesta ter po možnosti tudi količino časa prisotnosti. V primeru digitalne tabele, ki jo lahko uredimo v programu Microsoft Excel ali odprtokodnih rešitvah, kot recimo OpenOffice, lahko z uporabo formul uredimo, da se delovni čas posameznega dneva v tabeli izračuna povsem avtomatsko. Nekoč so se uporabljale časovne kartice, ki bi ob prihodu ali odhodu v določeno napravo, nameščeno pri vhodu, le-te vstavili in ta bi nato fizično na kartico napisala trenutni čas. Takšne kartice se danes v večini ne uporabljajo več, temveč jih je digitalizacija podjetij nadomestila z RFID čipi oz. karticami ali z ročno prijavo v delovni sistem z identifikacijsko številko ali uporabniškim imenom. Princip delovanja je ostal enak, le vrsta beleženja se je spremenila iz analognega v digitalno beleženje. S takim principom lahko naredimo beleženje bolj točno in preprečimo človeške napake.
Imamo še modernejše rešitve s pomočjo različnih aplikacij, kot sta All Hours in Clockify. Te aplikacije nam ne omogočajo le vodenje delovnih ur, temveč tudi odsotnosti, ustvarjanje delovnega urnika, pregled plač in vodenje časa uporabe delovnih aplikacij. Takšne aplikacije so lahko zelo priročne, sploh za delodajalca, saj se vse potrebne informacije izračunajo same ter nam omogočajo preprosto načrtovanje delovnika ter optimizacijo dela z generiranjem raznih poročil.

4.1 Moderne rešitve za evidentiranje

Moderne rešitve se lahko med seboj precej razlikujejo, so pa po osnovnih funkcionalnostih precej podobne. Nekatere so bolje avtomatizirane, druge manj. Med vsemi rešitvami smo si izbrali tri, ki jih obravnavamo podrobneje: našo rešitev, aplikacijo All Hours slovenskega podjetja Špica ter tujo rešitev po imenu Clockify, ki se uporablja v multinacionalnih podjetjih, kot so Amazon, Disney, Microsoft, Google idr.

4.1.1 Storitev Allhours

Rešitev All Hours vsebuje spletno in mobilno aplikacijo in terminale, s katerimi si lahko način registracije in evidentiranja delovnega časa prilagodimo glede na naše potrebe in način dela. Znotraj mobilne aplikacije lahko ročno začnemo meriti svoj delovni čas, hkrati pa tudi beleži GPS lokacijo naprave. Če bi delodajalec želel ravnati s temi meritvami bolj strogo, lahko s pomočjo mobilne aplikacije omeji lokacije, na katerih se delovni čas lahko meri. V primeru, da za nas navadna GPS lokacija ni dovolj točna, lahko te omejitve bolj natančno določimo s pomočjo dodatne naprave, imenovane Bluetooth žolna (beacon). S to napravo nam nato aplikacija sama registrira naš delovni čas, ko smo v njenem dosegu. Hkrati mobilna aplikacija še ponuja urejanje odsotnosti in pregled osnovnih informacij, povezanih z delovnim časom.

Medtem ko je mobilna aplikacija bolj priročna za delavca, je spletna aplikacija z njenimi funkcionalnostmi bolj priročna za delodajalca. Omogoča obračun delovnih ur, fleksibilno spreminjanje pravil glede malic, rednega dela, nadur, dopustov in drugih stvari ter vsak mesec s pritiskom na en gumb izvažanje vseh podatkov za plače. Znotraj spletne aplikacije lahko prav tako odobrimo ali zavrnemo zahteve o odsotnosti, nadzorujemo prisotnost v realnem času, vključno z lokacijami registracije, vidimo statistike o celem podjetju in imamo omogočen kronološki pregled vseh vpogledov, prijav, izvozov in urejanj podatkov.

Podjetje ne ponuja le spletno in mobilno aplikacijo, temveč tudi dva različna terminala, prenosnega in stacionarnega. Ta omogočata, da svojo osebno identifikacijsko kartico prislonimo nanjo in nam nato začne teči delovni čas. Stacionarni terminal se namesti na zid, ponavadi nekje v bližini vhoda oz. izhoda ter se na omrežje poveže žično. Prenosni terminal je precej manjši in namenjen za uporabo na terenu, glede funkcionalnosti pa ni veliko razlik.

4.1.2 Storitev Clockify

Rešitev Clockify je druga večja rešitev, ki se uporablja v podjetjih po celem svetu. Za beleženje časa se uporablja štoparica, ki jo mora posameznik začeti in končati sam. Posamezna opravila lahko podrobneje opredelimo in definiramo, če spadajo v določeno kategorijo ali projekt. Meritve časa lahko urejamo celo v različnih vpogledih, v tabeli, koledarju ali obrazcih. Vsaki osebi lahko točno določimo pozicijo in skupino, v katero spada ter tudi urno postavko ter osebne podatke. Podobno kot v rešitvi AllHours nam aplikacija omogoča beleženje lokacije, urejanje urnika, generiranje raznih poročil, dodatno pa Clockify omogoča še beleženje stroškov in izdelavo računov. Dodatna prednost rešitve je aplikacija za namizne in prenosne računalnike, ki lahko beleži čas avtomatsko in zazna, koliko časa se določena aplikacija na računalnikih uporablja.

4.1.3 Primerjava rešitev

V Tabela 4.1 Primerjava rešitev za evidentiranje delovnega časasmo primerjali rešitvi All Hours, Clockify in našo rešitev ter glavne funkcionalnosti in možnosti za evidentiranje delovnika.

Tabela 4.1 Primerjava rešitev za evidentiranje delovnega časa

	AllHours	Clockify	Naša rešitev
Podprte platforme z uporabniškim vmesnikom	Android, iOS, spletna aplikacija	Android, iOS, spletna aplikacija	Android
Ročno beleženje časa	✓	✓	✓
Avtomatsko beleženje časa	✓	✗	✓
Možnosti urejanja delovnega časa	✓	✓	✗
Možnost deljenja časa na različna opravila	✗	✓	✗
Beleženje lokacije	✓	✓	✗
Generiranje poročil in statistik	✓	✓	✓
Beleženje časa porabe posameznih aplikacij	✗	✓	✗
Možnost izdelave urnika	✗	✓	✗

Iz tabele lahko vidimo, da ima Clockify največ funkcij, s katerimi lahko olajšamo beleženje in evidentiranje časa. Med drugim omogoča tudi beleženje uporabe posameznih aplikacij,

deljenje časa na različna opravila in izdelavo urnika, kar AllHours in naša rešitev ne podpirata. Kar pa Clockify aplikaciji manjka, je možnost avtomatskega beleženja časa, kar drugi dve rešitvi podpirata. Razen teh razlik sta si v večini AllHours in Clockify precej podobna in vsebujeta vse potrebne funkcionalnosti za optimiziranje evidentiranja.

Če izpostavimo našo rešitev, ta prav tako vsebuje vse osnovne funkcije, potrebne za beleženje, čeprav je precej manj razvita kot drugi dve, seveda pa bi lahko z dodatnim razvojem slednjo še izboljšali in omogočili uporabnikom več možnosti kot pri drugih dveh. Največja pomanjkljivost v trenutnem stanju je manjkajoča nadzorna plošča za administratorje, kjer bi lahko preverjali poročila in urejali podatke.

5 ZAKLJUČEK

Cilj diplomskega dela je bila izdelava rešitve za evidentiranje delovnih ur, v katero smo želeli vključiti različne tehnologije. Ta cilj smo dosegli in razvili rešitev, s katero lahko preverjamo delovni čas brezkontaktno na ročni ali samodejni način. Vanjo smo kot predvideno uspešno vključili platformo Android, ogrodje Spring, orodje Spring Boot in Node-RED ter računalnik Raspberry Pi 4.

Izdelava mobilne aplikacije za platformo Android je bila precej lahka v primerjavi z drugimi deli rešitve. Orodje Android Studio nam je pri tem zelo olajšalo delo, saj je namenjeno izdelavi prav takšnih aplikacij in ima veliko dodatnih orodij za pomoč pri razvoju. Ker smo uporabili programski jezik Kotlin, smo nekoliko zmanjšali količino kode in se izognili raznim težavam, ki se pogosto pojavijo pri razvoju z jezikom Java.

Razvoj zaledja v ogrodju Spring in programskem jeziku Java je bil največji izziv, s katerim smo se soočili. Tehnologija nam je bila še precej neznana, saj še nismo imeli opravka z njo, kljub temu pa nam je uspelo ustvariti aplikacijo, s katero smo lahko realizirali vse osnovne predpostavke. Pri tem smo se naučili ustvarjanja API-ja z vzorcem krmilnik-storitev-repozitorij, novega načina upravljanja izjem ter podrobneje spoznali ogrodje Spring in orodje Spring Boot.

V delo smo prav tako vključili računalnik Raspberry Pi 4, na katerem smo razvili in izvajali aplikacijo Node-RED, a v manjšem obsegu kot predvideno. Kljub temu je imela pri tem ta velik vpliv, saj brez nje ne bi mogli preprosto in z malimi stroški omogočiti samodejno preverjanje prisotnosti na določenem delovnem mestu preko lokalnega omrežja.

Prav tako smo opisali vse uporabljene tehnologije, orodja in knjižnice ter raziskali že obstoječe rešitve za naš problem in jih primerjali z našo. Pri tem smo obravnavali delo z orodjem Spring Boot za razvoj API-ja, nakazali, kako lahko implementiramo varnejši protokol HTTPS za komunikacijo preko interneta in dokazali, da je vse analizirane tehnologije možno združiti v delujoč produkt.

6 VIRI

[1] Platforma Android. Dostopno na: https://developer.android.com/about [8. 11. 2021]

[2] Orodje Spring Boot. Dostopno na: https://spring.io/projects/spring-boot [17. 12. 2021]

[3] Node-RED. Dostopno na: https://nodered.org/ [20. 12. 2021]

[4] Podatkovna baza MongoDB. Dostopno na: https://www.mongodb.com/ [8. 11. 2021]

[5] Raspberry Pi 4 Model B. Dostopno na: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ [20. 12. 2021]

[6] Allhours. Dostopno na: https://allhours.com/sl [4. 3. 2022]

[7] Clockify. Dostopno na: https://clockify.me/ [4. 3. 2022]

[8] Razvojno orodje Android Studio. Dostopno na: https://developer.android.com/studio [8. 11. 2021]

[9] Razvojno orodje IntelliJ IDEA. Dostopno na: https://www.jetbrains.com/idea/ [8. 11. 2021]

[10] Android emulator. Dostopno na: https://developer.android.com/studio/run/emulator?gclid=CjwKCAiAg6yRBhBNEiwAeVyL0O-CgdxK5zC69JwvixzbXSz8XTUn8Vo2K9-9XgFuBGAVX4THQOOpWBoC1V0QAvD_BwE&gclsrc=aw.ds [8. 11. 2021]

[11] Programski jezik Kotlin. Dostopno na: https://kotlinlang.org/ [8. 11. 2021]

[12] Knjižnica MPAndroidChart. Dostopno na: https://github.com/PhilJay/MPAndroidChart [15. 11. 2021]

[13] Knjižnica Volley. Dostopno na: https://google.github.io/volley/ [15. 11. 2021]

[14] Ogrodje Java Spring. Dostopno na: https://spring.io/ [17. 12. 2021]

[15] Knjižnica Lombok. Dostopno na: https://projectlombok.org/ [17. 12. 2021]

[16] Spring Boot Security. Dostopno na: https://spring.io/projects/spring-security [17. 12. 2021]

[17] Organizacija Raspberry Pi. Dostopno na: https://www.raspberrypi.com/ [20. 12. 2021]

[18] Git. Dostopno na: https://git-scm.com/ [5. 12. 2021]

[19] Github. Dostopno na: https://github.com/ [5. 12. 2021]

[20] Orodje Postman. Dostopno na: https://www.postman.com/ [23. 12. 2021]

[21] Apache Log4J2. Dostopno na: https://logging.apache.org/log4j/2.x/ [17. 12. 2021]

[22] Vozlišče Node-RED ARP. Dostopno na: https://flows.nodered.org/node/Node-Red-contrib-arp [21. 12. 2021]

[23] Jasypt. Dostopno na: http://www.jasypt.org/ [21. 12. 2021]

[24] Omogočanje HTTPS klica v mobilni aplikaciji. Dostopno na: https://stackoverflow.com/questions/17045795/making-a-https-request-using-android-volley [1. 12. 2021]