FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Systém pro vykazování pracovních záznamů a intergace s JIRA
Student:	Rastislav Zlacký
Vedoucí:	Ing. Jan Petr
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Time Mission je interní informační systém umožňující správu projektů, vykazování času a fakturaci. Ve společnosti Inventi je používán jako hlavní IS pro evidenci pracovní doby. Vývojové týmy pro svou každodenní práci používají systém JIRA, který rovněž vyžaduje vykazování pracovního času, čímž dochází k duplicitám a rozdílům ve výkazech. Cílem práce je navrhnout a implementovat nový systém pro vykazování pracovních záznamů, které budou uloženy v TM DB a integraci s JIRA, tak aby bylo možno čas strávený prací na projektu vykazovat pouze v jednom systému.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 7. listopadu 2018

Bakalárska práca

Systém pre vykazovanie pracovných záznamov a integrácia s JIRA

Rastislav Zlacký

Katedra softwarového inžinierstva
Vedúci práce: Ing. Jan Petr

27. júna 2019

Poďakovanie

Týmto by som sa rád poďakoval svojmu vedúcemu, Ing. Janovi Petrovi, za jeho ochotu a čas, ktorý mi venoval počas písania tejto bakalárskej práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ustanovením § 46 odst. 6 tohoto zákona týmto udeľujem bezvýhradné oprávnenie (licenciu) k užívaniu tejto mojej práce, a to vrátane všetkých počítačových programov ktoré sú jej súčasťou alebo prílohou a tiež všetkej ich dokumentácie (ďalej len „Dielo"), a to všetkým osobám, ktoré si prajú Dielo užívať.

Tieto osoby sú oprávnené Dielo používať akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela, a za akýmkoľvek účelom (vrátane komerčného využitia). Toto oprávnenie je časovo, územne a množstevne neobmedzené. Každá osoba, ktorá využije vyššie uvedenú licenciu, sa však zaväzuje priradiť každému dielu, ktoré vznikne (čo i len čiastočne) na základe Diela, úpravou Diela, spojením Diela s iným dielom, zaradením Diela do diela súborného či zpracovaním Diela (vrátane prekladu), licenciu aspoň vo vyššie uvedenom rozsahu a zároveň sa zaväzuje sprístupniť zdrojový kód takého diela aspoň zrovnateľným spôsobom a v zrovnateľnom rozsahu ako je zpřístupnený zdrojový kód Diela.

V Prahe 27. júna 2019 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Rastislav Zlacký. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Zlacký, Rastislav. *Systém pre vykazovanie pracovných záznamov a integrácia s JIRA*. Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Táto bakalárska práca sa zaoberá analýzou, návrhom a implementáciou systému pre vykazovanie pracovného času a správu projektov. Tento systém je integrovaný s nástrojom Jira, čo je softwarový nástroj pre evidenciu požiadaviek, chýb a problémov vyskytujúcich sa pri vývoji software alebo riadenia projektov. Jira taktiež umožňuje vykazovať pracovný čas k daným úloham projektu. Integrácia umožní, aby bolo možné vykázať čas len v jednom systéme a zabránila rozdielom vo výkazoch. Serverová časť systému je vyvíjaná v jazyku Java pomocou frameworku Spring, ktorá vystavuje REST API. Dáta ukladáme do databázy PostgreSQL a využívame technológiu Elasticsearch pre fulltextové vyľadávanie. Synchronizáciu dát zabezpečuje technológia Logstash. REST API je zabezpečené a užívateľ sa autorizuje pomocou protokolu OAuth 2.0 poskytovateľa Google. Zaoberať sa budeme aj implementáciou klientskej aplikácie, vyvíjanej v jazyku JavaScript pomocou frameworku React, ktorá bude slúžiť len ako koncept, demonštrujúci proces prihlasovania pomocou Google OAuth 2.0. Hlavným cieľom je vytvorenie plne funkčnej serverovej časti systému. Pre jednoduchú inštaláciu a spustenie všetkých častí systému využívame nástroj Docker, ktorý jednotlivé časti systému zostaví a zabalí do kontajnerov, ktoré sa následne pomocou nástroja Docker Compose jednotne spustia.

Klíčová slova pracovné výkazy, sledovanie času, správa projektov, integrácia, Java, Spring, Elasticsearch, Logstash, Docker, REST API, OAuth 2.0, Jira, React

Abstract

This bachelor thesis covers the analysis, design and implementation of time tracking and project management system. The system is integrated with Jira, a issue tracking software tool that allows bug tracking and project management. Jira also allows you to report working time to the issues. The integration allows time to be reported in only one system so it prevents having duplicated work records. The server part of the system is developed in Java using the Spring framework that exposes the REST API. Data are stored in the PostgreSQL database and we use the Elasticsearch for full-text search. Data synchronization is ensured by Logstash. The REST API is secured and the user must be authorized using the OAuth 2.0 protocol with the Google provider. We will also deal with the implementation of a client application developed in JavaScript using the React framework which will only serve as a concept demonstrating the login process with Google OAuth 2.0. The main goal is to create a fully functional server part of the system – backend. To easily install and run all parts of the system we use the Docker tool which builds and creates containers that run uniformly. That is ensured by Docker Compose tool.

Keywords work records, time tracking, project management, integration, Java, Spring, Elasticsearch, Logstash, Docker, REST API, OAuth 2.0, Jira, React

Obsah

Úvod **1**

1 Cieľ práce **3**

2 Analýza a návrh **5**
2.1 Analýza súčasného stavu riešenia problému 5
2.2 Jira Software 7
2.3 Analýza požiadaviek 7
2.4 Prípady použitia 9
2.5 Doménový model 13
2.6 Použité technológie a nástroje 17
2.7 Architektúra systému 21
2.8 Databázový model 24
2.9 Indexy v Elasticsearch 26

3 Realizácia **27**
3.1 Serverová časť systému 27
3.2 Logstash 41
3.3 Klientska aplikácia 43
3.4 Nasadenie a spustenie 46

4 Testovanie **51**
4.1 Server 51
4.2 Klientska aplikácia 54

Záver **57**

Literatúra **59**

A Zoznam použitých skratiek **63**

B Obsah priloženého USB **65**

Zoznam obrázkov

2.1 Aktéri 9
2.2 Use Case diagram správy pracovných výkazov 10
2.3 Use Case diagram správy projektov 11
2.4 Use Case diagram správy uživateľov a autentizácie 12
2.5 Matica sledovateľnosti požiadaviek 13
2.6 Doménový model 14
2.7 Logstash 18
2.8 Docker – aplikácie v kontajneroch 20
2.9 Virtuálne stroje 20
2.10 High-level architektúra 22
2.11 Diagram komponent serverovej časti systému 24
2.12 Databázový model 25
2.13 Databázový model – changelog schéma 26

3.1 Class diagram – moduly Data-api a Data 28
3.2 Sekvenčný diagram importovania pracovných záznamov z Jira . . . 35
3.3 Konfigurácia aplikácie Google OAuth 2.0 v Google konzole 39
3.4 Sekvenčný diagram autorizácie Google OAuth 2.0 40

4.1 1. Prihlasovacia stránka 54
4.2 2. Prihlasovacia stránka Google 55
4.3 3. Žiadosť o povolenia 55
4.4 4. Profil uživateľa 56

Zoznam tabuliek

2.1 Atribúty entity User 15
2.2 Atribúty entity Work Record 15
2.3 Atribúty entity Project 15
2.4 Atribúty entity Work Type 16
2.5 Atribúty entity Project Assignment 16
2.6 Atribúty entity User Role 16
2.7 Atribúty entity Project Role 17

3.1 Základné REST API metódy 37

Úvod

Sledovanie a vykazovanie pracovného času je esenciálnou činnosťou, ktorá nám, tímom či zamestnancom, môže pomôcť stať sa organizovanejším a efektívnejším. Systémy umožňujúce vykazovanie pracovného času sú neoddeliteľnou súčasťou firiem, ktorých príjmy sú založené na zúčtovateľných hodinách, ale môžu ho používať aj samostatne zárobkovo činné osoby, ktoré si definujú prácu v rámci celého dňa. Ako uvádza pán Zwilling, zakladateľ Startup Professionals, samotný počet odpracovaných hodín patrí k jedným z kľúčových obchodných metrík [1].

Údaje v pracovných výkazoch sú presné, ľahko dostupné a možnosť zoskupenia podľa rôznych parametrov umožňuje rýchlo skontrolovať výsledky za ľubovoľné časové obdobie. Takto zhrnuté údaje odhaľujú úplný obraz o časových nákladoch, výkonnosti a členení podľa pracovných dní. Porovnávaním údajov z rôznych období vieme jednoduchšie pochopiť dôležité trendy, vidieť dynamiku práce a prípadne odhaliť aké postupy spôsobujú to, že zamestnanci strácajú čas. Porovnaním stanovených termínov a odhadov so skutočnými výsledkami vieme zlepšiť techniku odhadov a tak vykonávať presnejšie odhady v budúcnosti. Údaje sa dajú ďalej využiť pre zdôvodnenie nákladov zákazníkom, sledovanie prekročenia nákladov na projekty s fixnými nákladmi, účasť a neprítomnosť zamestnancov, analýzu oddeleniami ľudských zdrojov a presné údaje o tom, koľko času sa strávilo na určitom projekte vieme využiť pre automatické generovanie faktúr zákazníkom či mzdy zamestnancom.

Vykazovací systém môže využívať firma s ľubovoľným zameraním, avšak tímy vo firmách zameraných aj na softvérový vývoj zvyknú pre svoju prácu používať softvér JIRA, nástroj pre evidenciu chýb, problémov pri vývoji softwaru, ktorý uľahčuje proces riadenia projektov a taktiež umožňuje vykazovať odvedenú prácu [2]. V rámci vývoja vykazovacieho systému v tejto bakalárskej práci sa zameriame aj na integráciu s týmto systémom, čo umožní vykázanie pracovného času stráveného na projekte len v jednom systéme.

V bakalárskej práci najskôr preskúmame niekoľko existujúcich systémov

pre vykazovanie pracovného času. Získané informácie zužitkujeme v analytickej časti, v ktorej sa zameriame na špecifikáciu požiadaviek systému. Následne prevedemie návrh a implementáciu a na záver otestujeme funkčnosť systému.

Kapitola 1

Cieľ práce

Hlavným cieľom práce je navrhnúť, implementovať, zabezpečiť a otestovať serverovú časť systému (backend) pre vykazovanie pracovného času a správu projektov, ktorý bude integrovaný so sftwarovým nástrojom Jira tak, aby bolo možné vykázať čas len v jednom systéme a zabráni rozdielom vo výkazoch.

K dosiahnutiu tohto hlavného cieľa bude potrebné splniť niekoľko priebežných cieľov. Prvým z nich je preskúmanie niekoľko existujúcich riešení, analyzovať ich funkčnosť, získať prehľad a na základe získaných informácií určiť kľúčové funkcionality systému.

Ďalším cieľom je analýza, v ktorej špecifikujeme funkčné a nefunkčné požiadavky, vymodelujeme prípady použitia a navrhneme doménový model. Ďalej sa zameriame na architektúru nového systému, predstavíme technológie, ktoré použijeme a vytvoríme detailný návrh systému.

Posledným cieľom bude samotná implementácia systému a jeho následné otestovanie a overenie funknčnosti. Výsledkom implementácie bude taktiež tzv. „Proof of Concept“ [3] klientska aplikácia, ktorá bude demonštrovať prihlasovanie pomocou účtu poskytovateľa Google použitím OAuth 2.0 protokolu, pomocou ktorého bude naša serverová časť systému zabezpečená.

Kapitola 2

Analýza a návrh

V tejto kapitole sa budeme venovať analýze a návrhu systému. Začneme analýzou už existujúcich riešení, ktoré nám následne pomôžu špecifikovať funkčné požiadavky. Ďalej špecifikujeme nefunkčné požiadavky kladené na systém a vytvoríme modely prípadov použitia. Navrhneme doménový model, architektúru systému a predstavíme technológie, pomocou ktorých celý systém zrealizujeme.

2.1 Analýza súčasného stavu riešenia problému

Na trhu je v súčasnosti možnosť výberu zo širokej škály nástrojov, ktoré riešia rovnaký problém ako táto bakalárska práca. Pri prieskume existujúcich riešení sme vybrali tri služby Clockify, DeskTime a Harvest. Jednotlivé služby krátko predstavíme a zhodnotíme ich funkčnú výbavu.

Získané informácie z tejto analýzy využijeme pri špecifikácii vlastných funkčných požiadaviek systému. Všetky nasledujúce aplikácie v podstate zdieľajú jednodlivé funkcie a v princípe sa od seba nelíšia. Detailné porovnanie vykazovacích aplikácii si môžeme pozrieť v tabuľke nachádzajúcej sa na adrese `https://clockify.me/best-time-tracking-apps`.

Výhodu, ktorú naše budúce riešenie bude poskytovať je, že budeme mať prístup k zdrojovému kódu a kedykoľvek si môžeme systém rozšíriť o ďalšie funkcie.

2.1.1 Clockify

Clockify je tzv. timer-based systém na sledovanie pracovného času [4], ktorý sa dá ľahko používať. Prípad užitia je jednoduchý, vytvoríme projekt, potom vytvoríme úlohu v rámci projektu a spustíme časovač. Neskôr časovač zastavíme a systém údaje spracuje. Jednotlivé pracovné výkazy sa v systéme dajú zadávať aj manuálne.

Po ukončení projektu alebo určitého obdobia máme možnosť vytvoriť faktúru pre svojho klienta a priamo ju zo systému odoslať. Okrem toho môžeme svoj celkový projektový výkaz zdieľať s klientom v softvére Excel, exportovať do CSV alebo PDF.

Ďalej nám ponúka rôzne nastavenia projektov, priradzovanie zodpovednosti a vytváranie odhadovaných časových harmonogramov a rozpočtov pre každý projekt a je integrovaný na viac než 50 iných webových služieb.

Výhodou Clockify je, že je zadarmo pre neobmedzený počet ľudí, ale funkcionalita je obmedzená. Platené funkcie sa delia do ďalších troch kategórií PLUS, PREMIUM a SERVER, pričom cena predstavuje 10 $, 30 $ a 450 $ mesačne.

2.1.2 DeskTime

DeskTime je aplikácia, ktorá kombinuje tri kľúčové funkcie - monitorovanie zamestnancov, riadenie projektov a analýzu produktivity [5]. Tento softvér je navyše navrhnutý tak, aby pomohol manažérom a ich tímom identifikovať ich neproduktívne návyky, triedením webových stránok a aplikácií do „produktívnej“ časti a „neproduktívnej“ časti.

DeskTime nielenže sleduje čas, ale tiež automaticky vypočíta dennú produktivitu a efektivitu na základe kategorizácie adries URL, programov a aplikácií. Umožňuje vytvárať skupiny zamestnancov a riadiť produktivitu aplikácií individuálne pre každú skupinu. Fakturáciu, počítanie nákladov a správu projektov umožňuje podobne ako Clockify.

Narozdiel od Clockify, DeskTime je zadarmo len pre jedného uživateľa a funkcionalita je taktiež obmedzená. Za plnú funkcionalitu je potrebná mesačná platba v závislosti na počte ľudí v tíme či firme.

2.1.3 Harvest

Harvest [6] je aplikácia, ktorá umožňuje taktiež sledovať čas strávený na každom projekte alebo jednotlivých úlohách. Potom tieto údaje zbiera a vytvára intuitívne vizuálne správy, ktoré znázorňujú, na čom jednotlivé tímy pracujú.

Podobne ako Clockify, po ukončení určitého obdobia, umožňuje posielať faktúry svojim klientom priamo z aplikácie. Navyše posielanie faktúr prebieha pomocou systému Stripe [7] alebo PayPal [8]. Týmto spôsobom nie je potrebné platiť za dodatočný fakturačný a platobný softvér. Harvest dokáže klientom posielať aj automatickú pripomienku v prípade, že platba faktúr je oneskorená. V základných funkcionalitách sa to od predošlích softvérov nelíši.

Pre jedného uživaľa je táto aplikácia s plnou funkcionalitou zadarmo, ale počet projektov je obmedzený dvomi. Pre neobmedzený počet projektov sa platí 12 $ mesačne. Pre tímy, taktiež s neobmedzeným počtom projektov sa platí 12 $ mesačne za osobu.

2.2 Jira Software

Jira [2] je softwarový nástroj pre evidenciu požiadaviek, chýb a problémov vyskytujúcich sa pri vývoji software alebo riadenia projektov, vyvíjaný spoločnosťou Atlassian. Jira podporuje a uľahčuje proces riadenia projektov a požiadaviek, ponúka flexibilné uživateľské nástroje pre riadenie a sledovanie pracovníkov a ich výkonnosti pri plnení úloh.

Jira umožňuje vytvárať projekty, ku ktorým sa jednotlivé požiadavky či chyby evidujú. Tieto požiadavky môžu byť následne priradené k vývojárovi, ktorý má možnosť výkazať čas, strávený prácou na danej úlohe. Jira ďalej umožňuje nastaviť vlastný pracovný postup (workflow) vývoja.

2.3 Analýza požiadaviek

Požiadavky sú základom všetkých systémov a mali by byť jediným vyjadrením toho, čo by mal systém robiť a nie ako by to mal robiť [9, str. 78].

V tejto časti sa budeme zaoberať funkčnými a nefunkčnými požiadavkami kladenými na systém. Funkčné požiadavky určujú aké funkcie bude systém ponúkať a ako sa bude systém správať. Nefunkčné požiadavky majú určiť obmedzenia a dodatočné vlastnosti systému, ktoré majú dopad na voľbu technológií a architektúru.

2.3.1 Funkčné požiadavky

F1 – Vykazovanie pracovného času

Systém bude umožňovať vytváranie, úpravu a mazanie pracovných výkazov. Táto funkčnosť bude dostupná všetkým zamestnancom (uživateľom), ktorí sa úspešne autentizovali a sú priradení k aspoň jednému projektu. Pracovný výkaz sa vzťahuje vždy na projekt, na ktorom zamestnanec v aktuálnej dobe pracoval. Uživateľ bude svoju prácu vykazovať ručne pomocou aplikácie s grafickým uživateľským rozhraním, ktorej vývoj je mimo rozsahu tejto práce. Systém ďalej zaistí konzistentnosť pracovného času v jednotlivých výkazoch uživateľa, aby nedošlo k ich prekrývaniu.

Na základe oprávnenia prihlaseného uživateľa bude možné vytvárať, upravovať a mazať pracovné výkazy iných uživateľov.

F2 – Vykazovanie pracovného času zo systému Jira

Systém bude integrovaný so systémom Jira a zaistí, aby pracovné výkazy zamestnanca v systéme Jira boli uchované aj v tomto systéme. Tento proces môže prebiehať buď automaticky, alebo manuálne.

F3 – Správa projektov

Systém bude umožňovať správu projektov. Zahrňuje to vytváranie, úpravy a mazanie projektov. Každý projekt bude mať špecifikované typy práce,

ktoré zjednodušia proces vytvorenia pracovného výkazu. Môže sa jednať o vývoj, analýzu apod. Systém ďalej umožní na základe oprávnenia uživateľa priradzovať iných uživateľov k projektom, odoberať z projektov a meniť ich práva v rámci projektu.

F4 – Autentizácia a autorizácia
Vstup do systému bude podmienený prihlásením. Systém tak na základe oprávnenia prihláseného uživateľa a jeho oprávnení vzťahujúcich sa na konkrétne projekty sprístupní funkcie, ktoré bežne nebudú dostupné.

F5 – Správa úživateľov
Systém sám o sebe nebude umožňovať registráciu nových uživateľov a nebude uchovávať heslá. Uživateľ sa vytvorí automaticky počas autentizácie s poskytovateľom tretej strany. Podmienkou užívania systému je už vytvorený účet u tohto poskytovateľa – konkrétna špecifikácia sa nachádza v nefunkčnej požiadavke N1. Systém ďalej sprístupní oprávneným uživateľom možnosť zmeny oprávnení iných uživateľov.

F6 – Vyhľadávanie
Systém bude umožňovať fulltextové vyhľadávanie pracovných záznamov, projektov a uživateľov. Na základe oprávnení uživateľa bude funkcia vyhľadávnia obmedzená.

F7 – Reporting
Systém bude umožňovať reporting – oprávnený uživateľ bude môcť získavať denné, mesačné a ročné prehľady práce jednotlivých uživateľov vykonanej na určitom projekte.

2.3.2 Nefunkčné požiadavky

N1 – Zabezpečenie
Autentizácia a autorizácia bude prebiehať pomocou protokolu OAuth 2.0 [10] s externým poskytovateľom Google.

N2 – Výkonnosť
Predpokládaný počet uživateľov je 500, súčasne prihlásených 20 a odozva systému musí byť do 500 ms.

N3 - Rozšíriteľnosť
Dá sa očakávať, že sa v budúcnu objavia nové požiadavky na funkcionality serverovej časti. Systém preto musí byť ľahko rozšíriteľný.

2.3.3 Splniteľnosť požiadaviek

Splniteľnosť požiadaviek bude definovaná testovacími prípadmi (test cases), ktorým sa v tejto bakalárskej práci nebudeme venovať a ktoré budú dodané

test analytikmi. Testovacie prípady sa otestujú a na základe výsledku sa zhodnotí splniteľnosť daných požiadaviek.

2.4 Prípady použitia

V tejto časti sa budeme zaoberať modelovaním prípadov použitia, ktoré je doplnkovým spôsobom získavania a dokumentovania požiadaviek. Výstupom bude model, ktorého obsah budú tvoriť aktéri, hranice systému a prípady použitia:

aktéri predstavujú roly, pridelené osobám používajúcim daný systém

hranice systému je vyznačenie územia alebo hraníc modelovaného systému

prípady použitia sú činnosti, ktoré môžu aktéri so systémom vykonávať

2.4.1 Aktéri

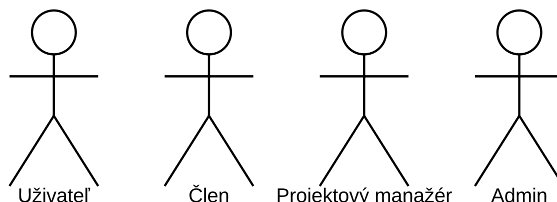
V systéme sa nachádzajú štyri druhy aktérov:

Uživateľ predstavuje základnú rolu v systéme s minimálnym oprávnením. Uživateľ si môže získať svoj profil a všetky jeho informácie, vyhľadávať, získať historické pracovné výkazy a prehľady s nimi spojené.

Člen je rozšírená rola Uživateľa, ktorá je vytvorená vo chvíli priradenia uživateľa na projekt a stáva sa týmto členom projektu. Člen má možnosť vykazovať pracovnú činnosť na projektoch, na ktorých je priradený.

Projektový manažér predstavuje rozšírenú rolu Člena. Projektový manažér má kontrolu nad určitým projektom a jeho členmi. Môže upravovať projektové informácie, získavať štatistické informácie a upravovať pracovné výkazy iných členov projektu.

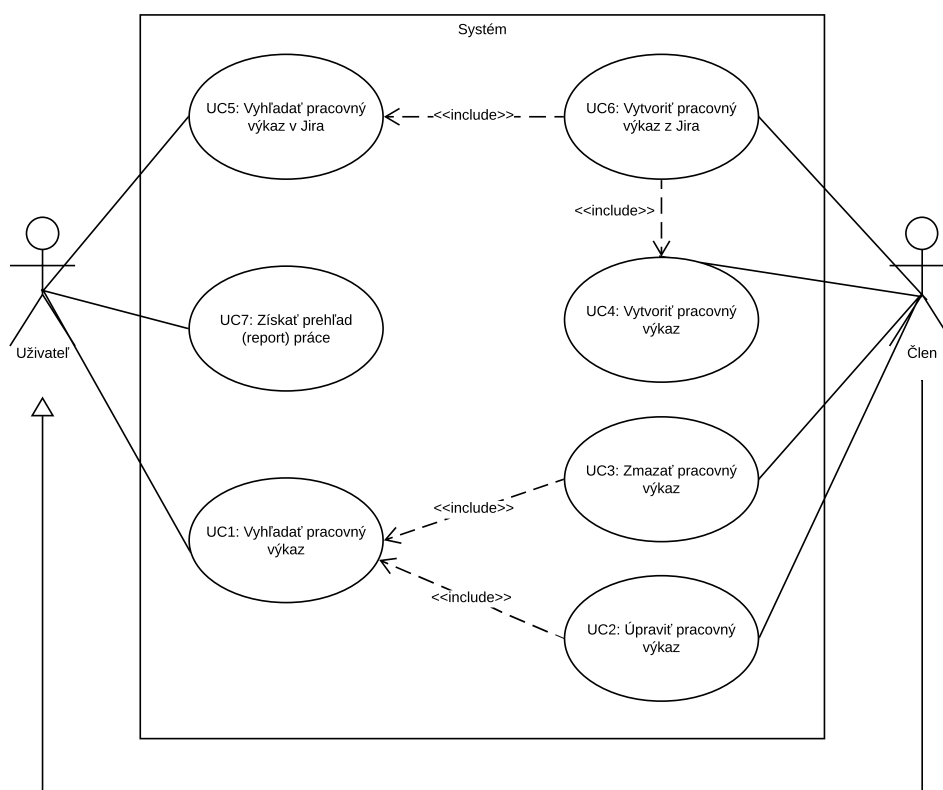
Admin má prístup ku všetkým zdrojom systému. Môže spravovať projekty, spravovať uživateľov a ich role, spravovať pracovné výkazy a získavať všetky štatistické informácie projektov či uživateľov.



Obr. 2.1: Aktéri

2.4.2 Správa pracovných výkazov

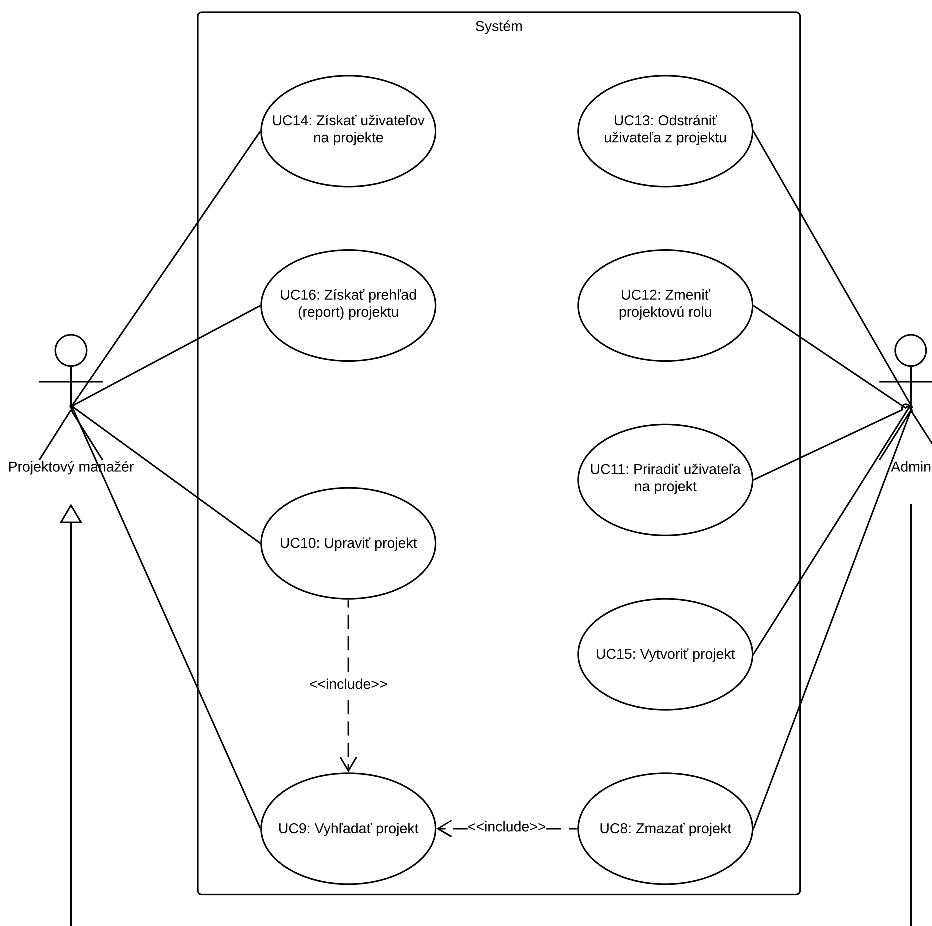
Na obrázku 2.2 sú znázornené prípady použitia, ktoré popisujú akcie spojené hlavne so správou pracovných výkazov uživateľa. Pre jednoduchosť, role Projektový manažér, Admin a ich komunikačné relácie nie sú v diagrame zahrnuté. Tieto role dedia z role Člen a použitie systému sa týmto nemení, avšak funkcie týchto rolí sú rozšírené. Projektový manážer má možnosť v rámci projektu, na ktorom túto rolu vykonáva, spravovať pracovné výkazy aj ostatných členov. Rola Admin môže spravovať pracovné výkazy ľubovoľného uživateľa, bez ohľadu na jeho pridelený projekt.



Obr. 2.2: Use Case diagram správy pracovných výkazov

2.4.3 Správa projektov

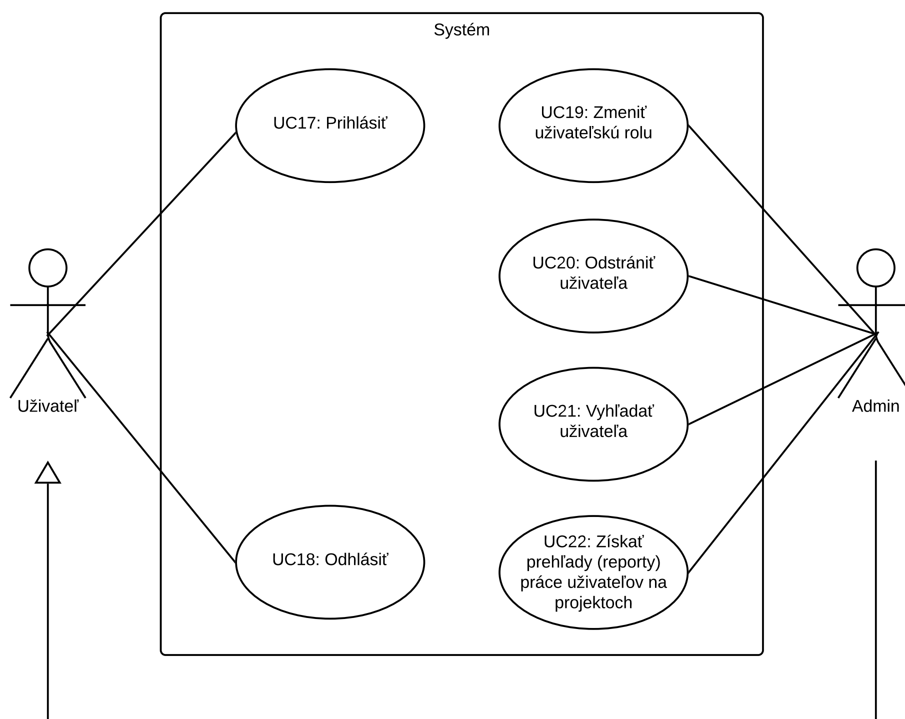
Táto sekcia pokrýva funkčné požiadavky týkajúce sa najmä správy projektov. V nasledujúcom diagrame sú znázornené odpovedajúce prípady použitia.



Obr. 2.3: Use Case diagram správy projektov

2.4.4 Správa uživateľov a autentizácia

Diagram prípadov užitia v tejto sekcii pokrýva funkčné požiadavky týkajúce sa správy uživateľov a autentizácie.



Obr. 2.4: Use Case diagram správy užívateľov a autentizácie

2.4.5 Prehľad realizácie požiadaviek

Pre overenie konzistencie funkčných požiadaviek a prípadov použitia využijeme maticu sledovateľnosti požiadaviek [9, str. 111]. Ak by v matici existovala funkčná požiadavka, ktorá nie je vo vzťahu so žiadnym prípadom použitia, znamenalo by to, že prípad použitia v modele chýba a je potrebné ho špecifikovať. V opačnom prípade to platí taktiež. Ak by v matici existoval prípad použitia, ktorý nie je vo vzťahu so žiadnou požiadavkou, znamenalo by to, že množina požiadaviek nie je úplná.

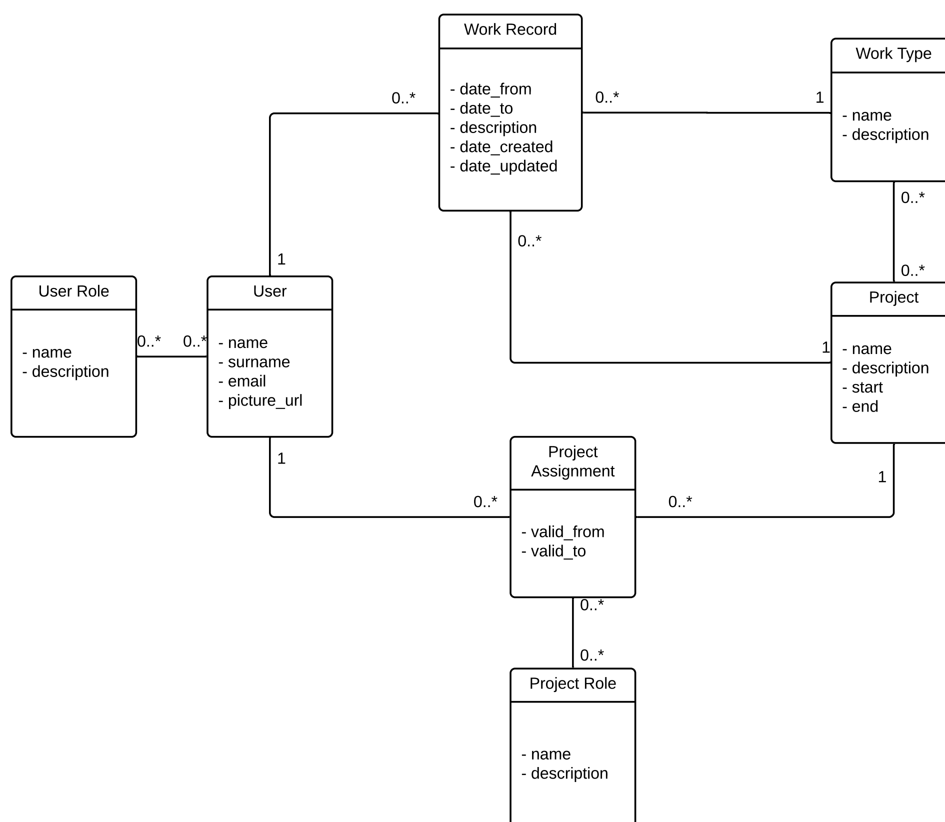
Na obrázku 2.5 je táto matica znázornená a môžme si všimnúť, že každý prípad použitia je namapovaný na niektorú z požiadaviek a každá požiadavka je namapovaná na aspoň jeden prípad použitia. Týmto spôsobom sme overili konzistentnosť našich funkčných požiadaviek a prípadov použitia.

	Funkčné požiadavky						
Prípady použitia	F1	F2	F3	F4	F5	F6	F7
UC1						■	
UC2	■						
UC3	■						
UC4	■						
UC5		■					
UC6		■					
UC7							■
UC8			■				
UC9						■	
UC10			■				
UC11			■				
UC12			■				
UC13			■				
UC14			■				
UC15			■				
UC16							■
UC17				■			
UC18				■			
UC19					■		
UC20					■		
UC21						■	
UC22							■

Obr. 2.5: Matica sledovateľnosti požiadaviek

2.5 Doménový model

V tejto sekcii sa zameriame na doménový model, ktorého diagram sa nachádza na obrázku 2.6. V diagrame sme zachytili entity súvisiace s analyzovanou doménou, ich relácie a kardinalitu. Jednotlivé entity ďalej popíšeme a atribúty týchto entít, pre prehľadnosť, popíšeme v tabuľkách.



Obr. 2.6: Doménový model

2.5.1 Entita User

Entita reprezentujúca uživateľa. Uživateľ je osoba, ktorá systém používa a uchovávame si o nej potrebné informácie. Môže mať niekoľko uživateľských rolí, ktoré jej umožňujú využívať rôzne funkcie systému. K ďalším rolám patria aj projektové role, ktoré uživateľ získava až po pridelení na projekt. Tieto role sme bližšie špecifikovali v sekcii 2.4.1.

Uživateľ môže byť pridelený na niekoľko projektov a môže vykazovať svoju prácu na každom z nich.

2.5.2 Entita Work Record

Entita reprezentujúca pracovný výkaz. V pracovnom výkaze musíme mať vždy informáciu, aký uživateľ výkaz vytvoril a taktiež k akému projektu a typu práce sa výkaz vzťahuje. Samotné atribúty sú popísané v následujúcej tabuľke.

Názov atribútu	Popis atribútu
name	Krstné meno
surname	Priezvisko
email	Email
picture_url	URL fotografie užívateľa

Tabuľka 2.1: Atribúty entity User

Názov atribútu	Popis atribútu
date_from	Dátum a čas začiatku práce
date_to	Dátum a čas konca práce
description	Popis práce
date_created	Dátum a čas vytvorenia pracovného výkazu
date_updated	Dátum a čas aktualizácie pracovného výkazu

Tabuľka 2.2: Atribúty entity Work Record

2.5.3 Entita Project

Entita reprezentujúca projekt. Každý projekt môže mať definované vlastné typy práce, pod ktorými užívateľ vykazuje svoju prácu. Nie je nutné, aby projekt mal definovaný typ práce, avšak užívateľ v takomto momente nebude môcť na tento projekt vykazovať svoju prácu.

K projektu sa ďalej môžu vzťahovať pridelenia užívateľov, popísané v sekcii 2.5.5.

Názov atribútu	Popis atribútu
name	Názov projektu
description	Popis projektu
start	Dátum a čas začiatku projektu
end	Dátum a čas konca projektu

Tabuľka 2.3: Atribúty entity Project

2.5.4 Entita Work Type

Entita reprezentujúca typ práce. K typu práce sa môžu vzťahovať pracovné výkazy a jeden typ práce môže patriť k viacerým projektom. Pod typom práce si môžme predstaviť napríklad vývoj, analýzu apod.

Názov atribútu	Popis atribútu
name	Názov typu práce
description	Popis typu práce

Tabuľka 2.4: Atribúty entity Work Type

2.5.5 Entita Project Assignment

Entita reprezentujúca pridelenie užívateľa na projekt. Vďaka informáciám o čase (doba platnosti) nástupu užívateľa na projekt a odstupu užívateľa z projektu máme dostupnú aj históriu užívateľov a ich pridelení na projekt. Dátum a čas odstupu užívateľa evidovať nemusíme. V takomto prípade to znamená, že užívateľ je v aktuálnej dobe stále aktívny na projekte.

Ku každému prideleniu užívateľa na projekt sa môžu vzťahovať projektové role, ktoré sú platné len v dobe platnosti.

Názov atribútu	Popis atribútu
valid_from	Dátum a čas začiatku nástupu užívateľa na projekt
valid_to	Dátum a čas odstupu užívateľa z projektu

Tabuľka 2.5: Atribúty entity Project Assignment

2.5.6 Entita User Role

Entita reprezentujúca užívateľskú rolu. K týmto rolám patrí rola Užívateľ a rola Admin, popísané v sekcii 2.4.1.

Názov atribútu	Popis atribútu
name	Názov užívateľskej roly
description	Popis užívateľskej roly

Tabuľka 2.6: Atribúty entity User Role

2.5.7 Entita Project Role

Entita reprezentujúca projektovú rolu. K týmto rolám patrí rola Člen a rola Projektový manažér, popísané v sekcii 2.4.1.

Rozdiel medzi užívateľskou rolou a projektovou rolou je hlavne v tom, že projektová rola sa vždy vzťahuje len na užívateľa a k nemu pridelený projekt. Užívateľská rola sa na žiaden projekt nevzťahuje a platí globálne pre celý systém.

Názov atribútu	Popis atribútu
name	Názov projektovej roly
description	Popis projektovej roly

Tabuľka 2.7: Atribúty entity Project Role

2.6 Použité technológie a nástroje

V nasledujúcich sekciách v krátkosti predstavíme jednotlivé technológie a nástroje, ktoré využijeme v realizačnej časti.

2.6.1 Spring Framework

Spring Framework je open-source aplikačný framework pre platformu Java, ktorého úlohou je zjednodušiť vývoj podnikových (enterprise) aplikácií [11]. Je to modulárny framework a umožňuje využiť len tú časť, ktorá sa hodí k riešeniu daného problému.

Jadro frameworku Spring je tvorené IoC (Inversion Of Control) kontajnerom. Princíp IoC je postavený na presunutí zodpovednosti za vytváranie, previazanie a spravovanie životného cyklu objektov na kontajner [12]. Takéto objekty sa nazývajú Beans a kontajner ich injektuje do miesta, kam ich potrebujeme. Takáto injektáž je známa pod termínom DI (Dependency Injection) [13]. Hlavnou výhodou tohto princípu je nízka previazanosť objektov v aplikácii, čím sa aplikácia stáva viac udržiavateľná a jednoduchá testovateľnosť.

Okrem Spring Frameworku nám Spring ponúka niekoľko ďalších projektov na ňom postavených, ktoré poskytujú podporu pre rôzne architektúry aplikácií. Zahrňuje to napríklad zasielanie správ (messaging – Spring AMQP), servletový resp. reaktívny web framework (Spring MVC resp. Spring WebFlux), perzistenciu dát (Spring Data) atď. [14]

Pri realizácii, okrem Spring Frameworku, využijeme tieto Spring projekty:

Spring Data poskytuje konzistentný prístup k uloženým údajom v relačnej či nerelačnej databáze, prístup ku cloudovým datovým službám apod.

Spring MVC je framework, postavený na Servlet API [15] a uľahčuje vývoj webu. Zahrňuje vývoj RESTful webovej služby.

Spring Security je prispôsobiteľný framework, ktorý sa zameriava na autorizáciu a autentifikáciu Spring aplikácií

Spring Boot uľahčuje celkovú konfiguráciu Spring aplikácií. Pri jeho použití nie je potrebné jednotlivé Spring moduly konfigurovať, Spring Boot si ich dokáže automaticky nakonfigurovať. Využíva Tomcat ako predvolený vstavaný kontajner [16].

2.6.2 PostgreSQL

PostgreSQL je open-source objektovo-relačný databázový systém. PostgreSQL si získal silnú reputáciu vďaka svojej osvedčenej architektúre, spoľahlivosti, robustnému nastaveniu funkcií, rozšíriteľnosti a open-source komunity, ktorá neustále poskytuje výkonné a inovatívne riešenia [17].

2.6.3 Elasticsearch

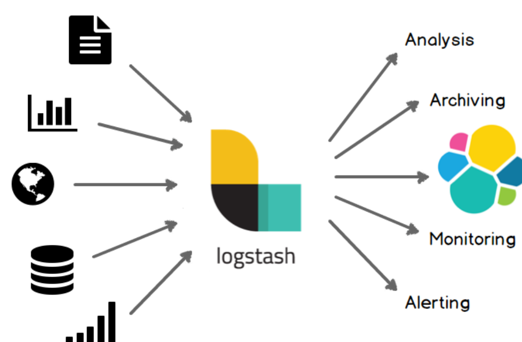
Elasticsearch je distribuovaný, RESTful vyhľadávač (search engine) založený na knižnici Apache Lucene [18].

Elasticsearch môžeme použiť na vyhľadávanie všetkých druhov informácií. Ku kľúčovým vlastnostiam tejto technológie patrí hlavne jeho rýchlosť a uložené údaje vieme získať takmer v réalnom čase. Tým, že Elasticsearch je distribuovaný systém, vieme ho ľahko horizontálne škálovať, pričom automaticky riadi, ako sa indexy a dotazy distribuujú v rámci clusteru. Ďalej dokáže zistiť zlyhania systému a pomocou replík zabezpečiť dostupnosť údajov bez výpadku [19].

Elasticsearch je najpopulárnejší vyhľadávací nástroj [20].

2.6.4 Logstash

Logstash je open-source, server-side systém pre spracovanie dát (data processing pipeline) [21]. Dokáže prijímať dáta z viacerých zdrojov súčasne, transformovať ich a potom ich transportovať napríklad do technológie Elasticsearch, Amazon S3, MongoDB, RabbitMQ apod. Tieto technológie môžu slúžiť aj ako zdroj dát pre Logstash. Celý zoznam sa nachádza na dokumentačných stránkach [22] a [23]. Zjednodušene povedané, Logstash dáta na vstupe príjme, transformuje a prefiltruje podľa našich podmienok a na výstupe dáta odovzdá.



Zdroj: `https://www.elastic.co/guide/en/logstash/current/static/images/logstash.png`

Obr. 2.7: Logstash

2.6.5 React

React je open-source JavaScript knižnica pre tvorbu uživateľského rozhrania webových a mobilných aplikácií. Na rozdiel od rôznych kompletných frameworkov (napr. Angular) sa React sústredí na jednu špecifickú oblasť a ak sa na to pozrieme z hľadiska MVC architektúry, tak tvorí práve vrstvu, ktorá prezentuje dáta uživateľovi – View. Preto sa v praxi používa v kombinácii s ďalšími knižnicami, ktoré zabezpečujú napr. smerovanie (routing), interakciu s API apod [24].

2.6.6 SonarQube

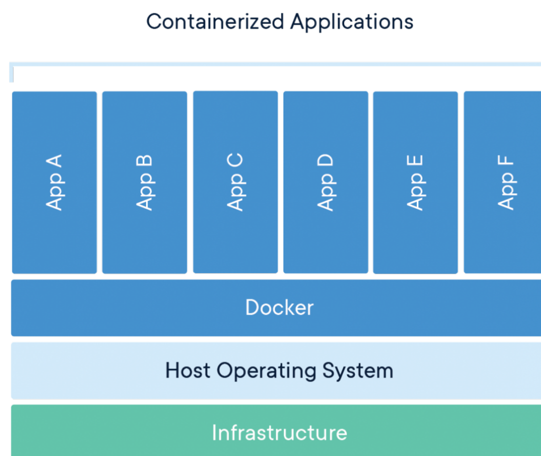
SonarQube je nástroj pre kontrolu kvality zdrojového kódu. Vykonáva statickú analýzu kódu, kód vyhodnocuje, vyhľadáva duplicity, detekuje potenciálne chyby a bezpečnostné zraniteľnosti. Podporuje viac ako 25 jazykov vrátane jazyku Java, C#, JavaScript, C, C++ a ďalších.

SonarQube dokáže spracovať výsledky testov vyvíjanej aplikácie a vytvárať prehľady, obsahujúce úspešnosť testov a pokrytie zdrojového kódu testami.

2.6.7 Docker

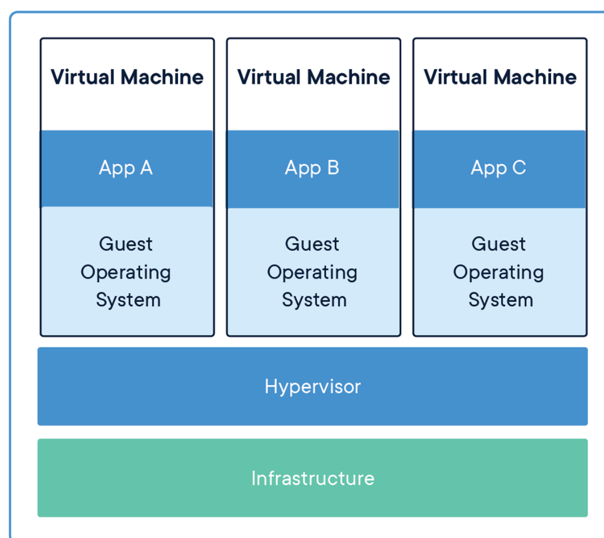
Docker je nástroj, ktorý uľahčuje vývoj, nasadzovanie a spúšťanie aplikácií pomocou kontajnerov. Kontajnery umožňujú vývojárovi zostaviť a zabaliť aplikáciu so všetkými časťami, ktoré potrebuje, ako sú knižnice a iné závislosti. Takto vytvorené kontajnery zabezpečia, že aplikácia bude spustiteľná na akomkoľvek Linux systéme bez ohľadu na aktuálne nastavenia systému, ktoré by sa mohli líšiť od systému, na ktorom bola aplikácia vyvíjaná [25].

Na obrázku 2.8 môžme vidieť infraštruktúru aplikácií bežiacich v kontajneroch v technológii Docker. Od virtuálnych strojov sa kontajnery líšia hlavne tým, že kontajnery sú abstrakciou v aplikačnej vrstve, zatiaľ čo virtuálne stroje (VMs) sú abstrakciou fyzického hardwaru [26]. Virtuálne stroje sú znázornené na obrázku 2.9.



Zdroj: `https://www.docker.com/resources/what-container`

Obr. 2.8: Docker – aplikácie v kontajneroch



Zdroj: `https://www.docker.com/resources/what-container`

Obr. 2.9: Virtuálne stroje

2.7 Architektúra systému

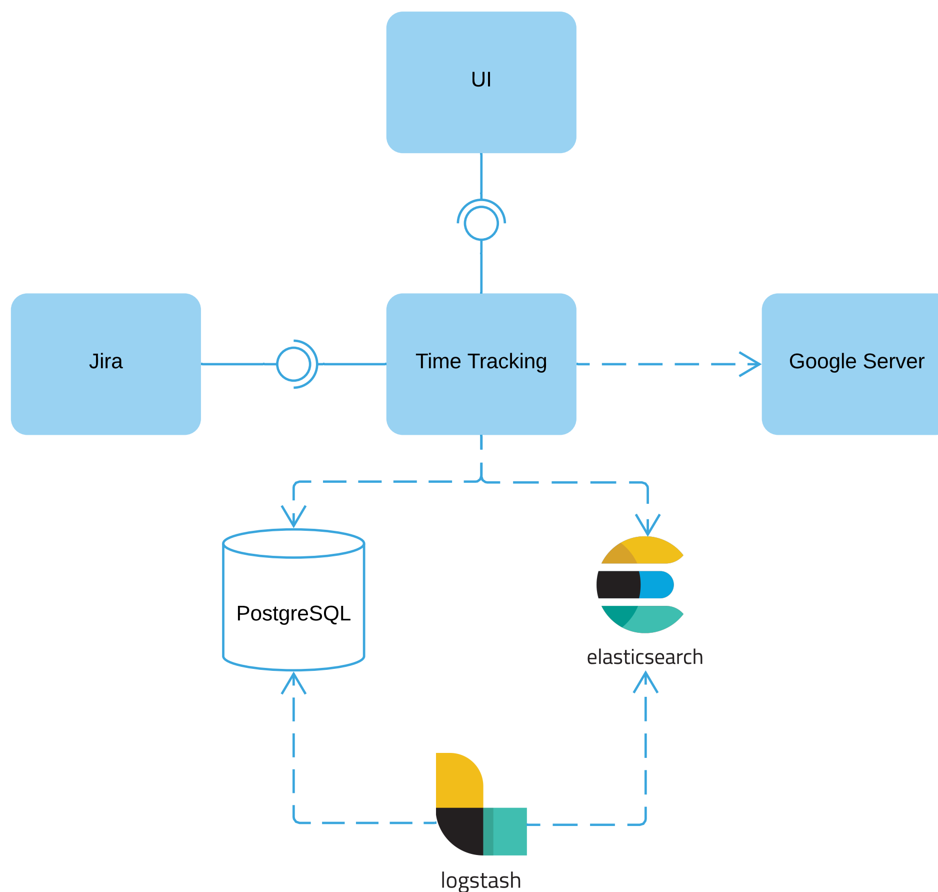
V nasledujúcich sekciách si predstavíme architektúru systému. Zameriame sa na architektúru systému ako celok (high-level architektúra), ktorá je popísaná v sekcii 2.7.1. Následne v sekcii 2.7.2 popíšeme architektúru serverovej časti systému.

2.7.1 High-level architektúra

Na obrázku 2.10 môžeme vidieť high-level architektúru systému, na ktorej sú znázornené všetky komponenty využívané priamo alebo nepriamo serverovou časťou aplikácie. Serverová časť aplikácie sa na obrázku nachádza pod názvom Time Tracking.

Time Tracking komponenta využíva servery Google kvôli autorizácii pomocou protokolu OAuth 2.0 a software Jira pre získavanie pracovných výkazov. Pre perzistentné ukládanie dát využijeme objektovo-relačný databázový systém PostgreSQL a pre účely rýchleho fulltextového vyhľadávania používame technológiu Elasticsearch. Aj keď samotný systém PostgreSQL poskytuje funkčnosť fulltextového vyhľadávania a bol by v súčasnosti postačújuci, pri zložitejších dotazov, agregácií a prípadných analyzátorov už by to nestačilo a pri prípadnom rozšírení nášho systému o tzv. business intelligence nástroje [27] by bolo potrebné uskutočniť technologickú migráciu. Z tohto dôvodu sme zvolili Elasticsearch a pre synchronizáciu dát medzi Elasticsearch a PostgreSQL využívame technológiu Logstash. Podľa oficiálnej dokumentácie technológie Elasticsearch [28] a diskusie [29] môžu nastať prípady, v ktorých by mohlo dôjsť k nekonzistencii údajov. Z tohto dôvodu využívame PostgreSQL ako primárne dátove úložisko a Elasticsearch ako sekundárne, len pre účel fulltextového vyhľadávania.

Komponenta UI je webová aplikácia s užívateľským rozhraním, ktorá využíva API Time Tracking komponenty.



Obr. 2.10: High-level architektúra

2.7.2 Architektúra serverovej časti

Architektúra serverovej časti systému je modulárna, pričom každý modul zodpovedá za určitú časť systému. Na obrázku 2.11 je táto architektúra zachytená. Jednotlivé moduly a ich funkcie sú:

Data
: Umožňuje perzistentne ukladať a načítať údaje z databázy. Viď. sekcia 2.8.

Search
: Integruje technológiu Elasticsearch a poskytuje rozhranie pre fulltextové vyhľadávanie uživateľov, pracovných záznamov a projektov.

Jira Integruje software Jira a poskytuje rozhranie pre získavanie pracovných záznamov zo systému Jira.

User
Zodpovedá za správu uživateľských účtov a správu ich uživateľských rolí.

Project
Zodpovedá za správu projektov, priradzovanie uživateľov k projektom a správu ich projektových rolí.

Work Record
Zodpovedá za správu pracovných výkazov a využíva modul Jira, z ktorého získava pracovné výzákazy a poskytuje rozhranie pre ich správne uchovanie v systéme.

Report
Poskytuje rozhranie pre získavanie reportov – prehľady odpracovanej doby za určité obdobie, celkové prehľady projektov apod.

Rest
Vystavuje REST API [30], validuje vstupné parametre a využíva vyššie uvedené moduly, do ktorých deleguje jednotlivé volania API. Tento modul predstavuje bránu do aplikácie.

Security
Modul, úzko spojený s modulom Rest, ktorý obsahuje konfiguráciu, potrebnú pre zabezpečenie REST API a logiku pre dodatočné riadenie prístupu. Autorizácia uživateľa prebieha pomocou protokolu OAuth 2.0 poskytovateľa Google.

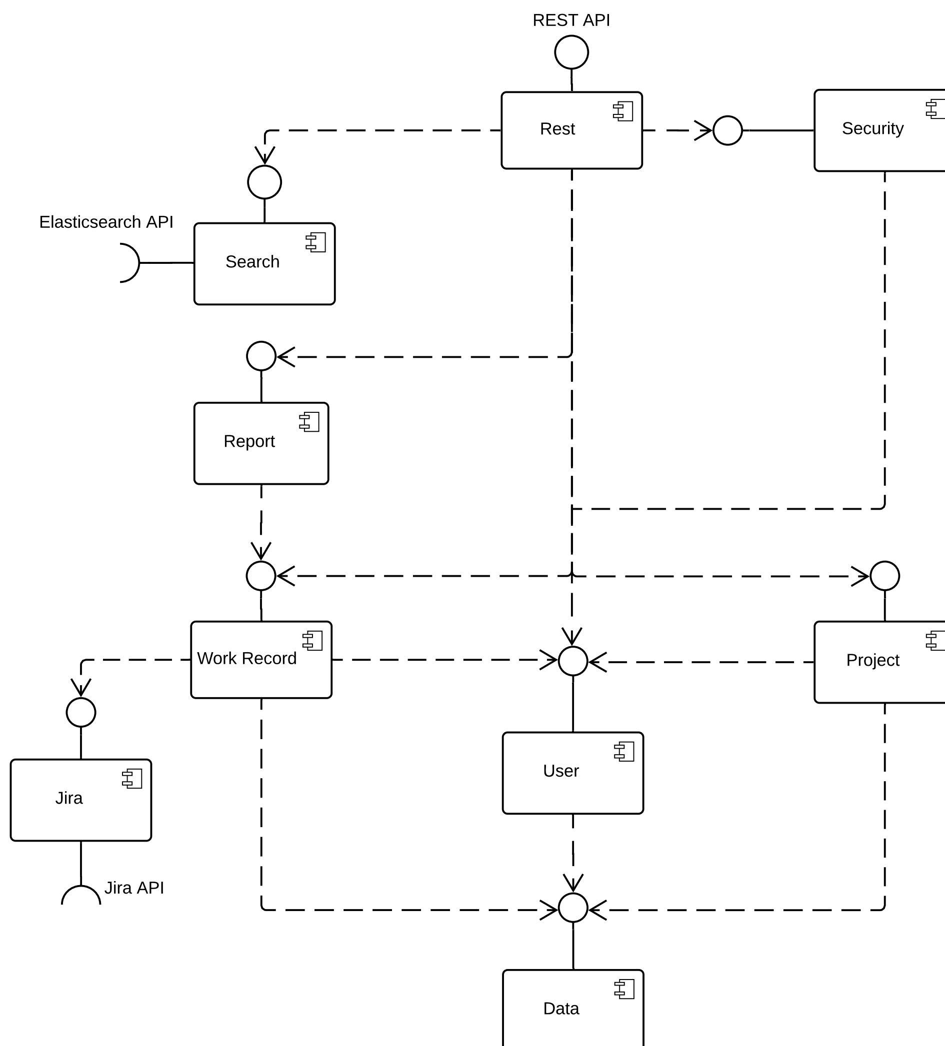
Ak by sme tieto moduly chceli zoskupiť, vznikli by nám abstrakné vrstvy reprezentujúce trojvrstvú architektútu – prezentačná, business a dátová vrstva [31, str. 19, 20].

Do prezentačnej vrstvy by sme zaradili modul Rest. Jej úlohou je zobraziť informácie uživateľovi a poskytnúť možnosť interagovať so systémom pomocou grafického uživateľského rozhrania či vystavením API.

Do business (servisnej) vrstvy by sme zaradili moduly User, Project, Work Record, Report a Security. V business vrstve je umiestnená všetka logika aplikácie a moduly v nej môžu, ak je to potrebné, interagovať s ostatnými modulmi.

Do dátovej vrstvy by sme zaradili moduly Data, Jira a Search. Úlohou dátovej vrstvy je komunikovať s inými systémami, kroré vykonávajú rôzne úlohy v mene aplikácie. Pre väčšinu podnikových aplikácií sa jedná o databázy, ktoré sú zodpovedné za ukládanie persistentých údajov. Okrem databáz to môže byť ľubovoľná iná aplikácia, EMS (messaging systems) apod. [31, str. 20]

Takto navrhnutú serverovú časť systému vieme v prípade potreby ľahko upraviť, rozšíriť o nové funkcie a nezávisle implementovať nové moduly.



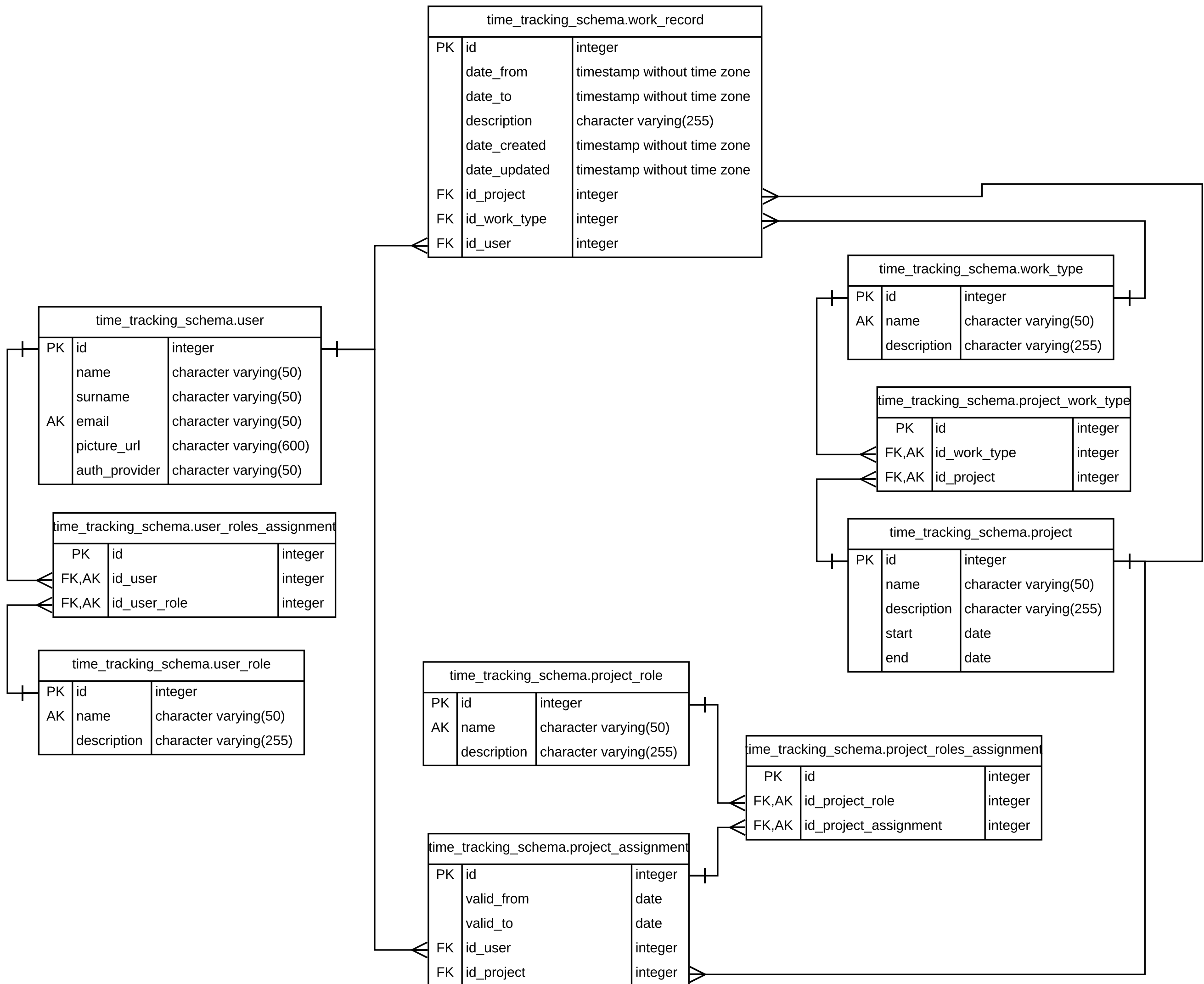
Obr. 2.11: Diagram komponent serverovej časti systému

2.8 Databázový model

Databázový model, znázornený na obrázku 2.12, vychádza z doménového modelu, ktorému sme sa venovali v sekcii 2.5. K jednotlivým atribútom v tabuľkách su priradené ich dátové typy a integritné obmedzenia.

Oproti doménovému modelu databázový model obsahuje naviac tabuľky pre dekompozíciu M:N väzieb a pribudla nová tabuľka v odlišnej schéme s

názvom `changelog`, znázornená na obrázku 2.13. V tejto tabuľke evidujeme vykonané operácie (INSERT, UPDATE, DELETE) nad určitými tabuľkami, čas, kedy táto operácia bola vykonaná a identifikátor záznamu, nad ktorým bola operácia vykonaná. Význam a dôvod vzniku tejto tabuľky si povieme v sekcii 3.2.



Obr. 2.12: Databázový model

time_tracking_changelog_schema.changelog		
PK	id	integer
	id_row_changed	integer
	table	character varying(50)
	action	character varying(50)
	stamp	timestamp with time zone

Obr. 2.13: Databázový model – changelog schéma

2.9 Indexy v Elasticsearch

V Elasticsearch budú existovať tri indexy, do ktorých sa budú ukladať dokumenty odpovedajúce databázovým entitám `project`, `user` a `work_record`. To znamená, že index s názvom `projects` bude obsahovať dokumenty len jedného typu, ktorým je `project`. To isté platí aj pre indexy `users` a `work_records`.

Kapitola 3

Realizácia

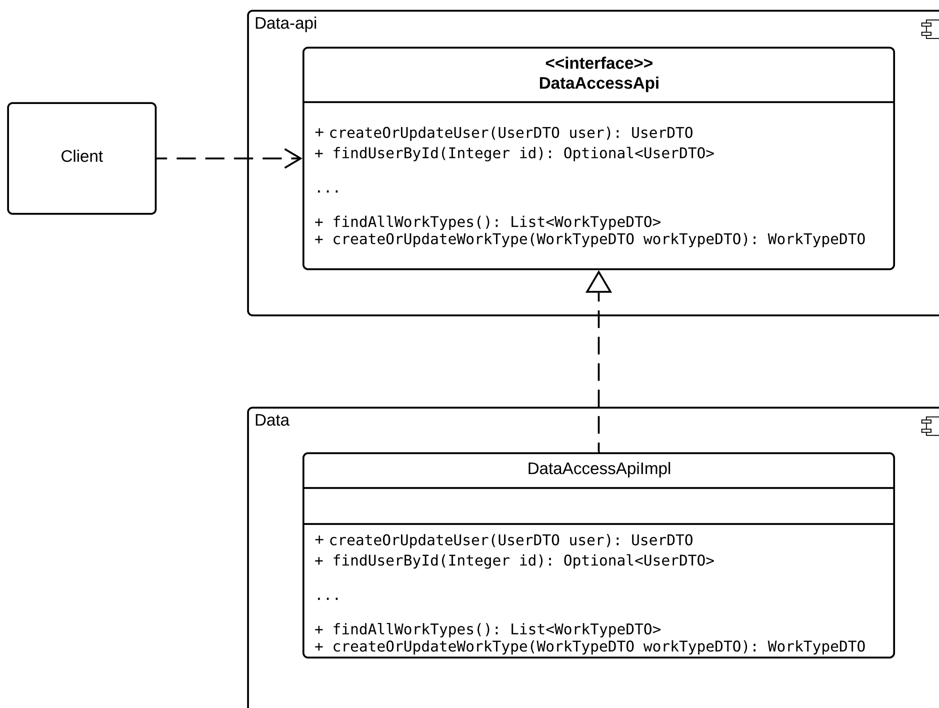
V tejto kapitole sa zameriame na realizačnú časť bakalárskej práce. Popíšeme spôsob implementácie serverovej časti systému a modulov, klientskej aplikácie a ukážeme si akým spôsobom sme nakonfigurovali Logstash tak, aby údaje medzi databázou a Elasticsearch boli synchronizované.

3.1 Serverová časť systému

Serverová časť systému je implementovaná pomocou frameworku Spring a nasledujúce sekcie obsahujú popis spôsobu implementácie jednotlivých modulov.

3.1.1 Data

Modul Data je implementovaný pomocou SPI vzoru [32], ktorý prináša výhodu v tom, že aplikácia, obecnejšie klient, využivajúci tento modul nie je priamo závislý na implementácii. V dobe kompilácie stačí, že klient pozná rozhranie API a samotná implementácia sa použije až v dobe behu aplikácie (runtime). Pomocou tohto vzoru abstrahujeme spôsob ukládania dát. V prípade potreby vieme ľahko zmeniť implementáciu spôsobu ukládania údajov (databázový systém, cloud, atď.) a to bez úpravy klienta, pretože klient nemá žiadnu informáciu o spôsobe ukládania údajov a ani ju nepotrebuje. Aby sme zabezpečili úplné oddelenie SPI od SP na úrovni kódu, SPI sme vložili do samostatného modulu s názvom `Data-api`. Class diagram, na ktorom je znázornená myšlienka SPI vzoru sa nachádza na obrázku 3.1. Deklarované metódy v SPI – `DataAccessApi` v diagrame sú zámerne, pre prehľadnosť, neúplné.



Obr. 3.1: Class diagram – moduly Data-api a Data

V implementácií tohto modulu využívame, pre prístup k PostgreSQL databáze, Spring Data JPA, časť projektu Spring Data, ktorému sme sa venovali v sekcii 2.6.1. Základ tvoria triedy entít a repozitárov.

3.1.1.1 Entity

Súčasťou Spring Data JPA je samotná JPA implementácia s názvom Hibernate [33], ktorá okrem iného umožňuje mapovať Java triedy (POJO) na tabuľky v databáze – Hibernate ORM [34]. Takéto triedy nazývame entity a instancie týchto tried reprezentujú jednotlivé záznamy v tabuľke, ktorú daná entita reprezentuje.

Príklad entity `User` môžeme vidieť v ukážke kódu 3.1. Všimnúť si môžeme anotáciu `@Entity` ktorá špecifukuje, že daná trieda je entitou. Anotácia `@Table` špecifikuje na ktorú tabuľku v databáze je daná trieda namapovaná. Následujú jednotlivé členské premenné, pri ktorých špecifikujeme pomocou anotácie `@Column` názvy stĺpcov tabuľky, na ktoré sú premenné namapované. Ďalej si môžeme všimnúť anotácie `@NotEmpty` resp. `@Email`, ktoré špecifikujú, že daná premenná nemôže byť prázdna resp. musí mať validný formát emailovej adresy. Hibernate sa o tieto premenné postará a hodnoty v nich validuje. Pri

nevalidnej hodnote zahlási chybu v podobe vyhodenia `ValidationException` apod.

Okrem mapovania členských premenných na stĺpce tabuľky, vieme mapovať aj jej relácie. Jednou z anotácií, umožňujúce mapovať relácie, je `@OneToMany`. Pri pohľade na databázový model si všimneme, že užívateľ môže mať niekoľko pracovných záznamov a pracovný záznam môže byť priradený práve k jednému užívateľovi. Jedná sa o väzbu 1:N. Presne tomu odpovedá aj táto anotácia a pri získavaní tejto entity máme možnosť získať všetkých N pracovných záznamov. Avšak využitie tejto možnosti sa neodporúča z dôvodu neefektivity a štandardné chovanie anotácie `@OneToMany` je, že N záznamov v relácii sa z databázy nezískava. Osvedčené postupy využívania anotácii, umožňujúce mapovať relácie, sú detailne popísané v článkoch od Vlada Mihalcea [35] a [36].

Pre prehľadnosť je ukážka definície entity neúplná a pre úplnú definíciu tejto entity je potrebné siahnuť do zdrojových kódov v priloženom USB. V zdrojových kódoch sa nachádadzajú všetky namapované entity odpovedajúce databázovému modelu.

```
@Entity
@Table(name = "user", schema = "time_tracking_schema")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;

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

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

    @NotEmpty
    @Email
    @Column(name = "email", unique = true)
    private String email;

    @OneToMany(mappedBy = "user")
    private Set<WorkRecord> workRecords = new HashSet<>();
}
```

Ukážka kódu 3.1: Príklad entity User

3.1.1.2 Repozitáre

„The goal of the Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.“ [37]

Význam vyššieho citátu a výhody, ktoré nám Spring Data prináša si ukážeme rovno na príklade 3.2. K jednej z týchto abstrakcií patrí rozhranie `JpaRepository`, ktoré ponúka okrem iného automatickú implementáciu CRUD operácií. Stáčí ak špecifikujeme generické typové parametre (napr. užívateľská rola – entita UserRole) a typ primárneho kľúča (Integer) a Spring Data sa o automatickú implementáciu postará.

V prípade, že by nám základné operácie nestačili, môžeme pomocou anotácie `@Query` špecifikovať SQL dotaz v JPQL alebo v natívnom SQL jazyku [38]. Pri jednoduchších dotazov nie je potrebné špecifikovať SQL dotaz, pretože Spring Data dokáže niektoré metódy rozpoznať a vytvoriť k ním odpovedajúce SQL dotazy [39]. Príklad si môžeme všimnúť na metóde `findAllByNameIn(List<UserRoleName> names)` – entita `UserRole` obsahuje členskú premennú `name`, Spring Data to rozpozná a vytvorí dotaz, ktorý vyhľadá všetky užívateľské role pomocou operátora IN. Parametre metódy predstavujú parametre SQL dotazu.

```
public interface UserRoleRepository extends JpaRepository<UserRole,
    Integer> {

    @Query("SELECT r FROM UserRole r LEFT OUTER JOIN r.users u WHERE
        u.id = :id")
    List<UserRole> findAllByUserId(@Param("id") Integer id);

    List<UserRole> findAllByNameIn(List<UserRoleName> names);
}
```

Ukážka kódu 3.2: Príklad repozitára pre entitu UserRole

3.1.2 Search

Modul Search integruje Elasticsearch pomocou Elasticsearch knižnice – Java High Level REST Client [40]. Základom našej implementácie je vytvorenie full-textového dotazu, pomocou ktorého sa dotazujeme Elasticsearch pre získanie projektov, užívateľov alebo pracovných záznamov.

Príklad dotazu pre vyhľadávanie užívateľov, ktorý sme zostavili pomocou Query DSL [41] sa nachádza v následujúcej ukážke 3.3. V ňom definujeme zložený dotaz `bool`, ktorý nám umožňuje kombinovať viaceré poddotazy a vyhodnocuje ich na základe Booleovej algebry. V klauzulách `should` sa ďalej nachádzajú už konkrétne fulltextové dotazy a to `multi_match` a `query_string`. Pre zahrnutie dát vo výsledku vyhľadávania stačí, ak je jeden

z týchto dotazov vyhodnotený správne. V dotaze `multi_match` špecifikujeme polia, podľa ktorých sa má fulltextovo vyhľadávať a parameter `fuzziness`, pomocou ktorého sa prepočítava editačná vzdialenosť (Levenshtein distance [42]). Na základe editačnej vzdialenosti vie Elasticsearch zahrnúť také výsledky, ktoré sa presne nezhodujú s vyhľadávaným textom a vo vyhľadávanom texte môžu existovať preklepy. V dotaze `query_string` taktiež špecifikujeme polia, podľa ktorých sa má vyhľadávať, ale na základe prefixu alebo sufixu. Prefix alebo sufix definujeme pomocou znaku `*`.

```
1  {
2    "query": {
3      "bool" :{
4        "should" :[
5          {
6            "multi_match" :{
7              "query" :"rastisla zlacky",
8              "fields" :[
9                "name",
10               "surname"
11             ],
12             "fuzziness" :"AUTO"
13           }
14         },
15         {
16           "query_string" :{
17             "query" :"*rastisla* *zlacky*",
18             "fields" :[
19               "name",
20               "surname"
21             ]
22           }
23         }
24       ]
25     }
26   }
27 }
```

Ukážka kódu 3.3: Elasticsearch dotaz pre vyhľadávanie užívateľov

Dotazy Elasticsearch vytvoríme v jazyku Java pomocou vyššie spomínanej knižnice a ako to vyzerá v kóde spolu s metódou pre vyhľadávanie užívateľov sa nachádza v ukážke 3.4. Pre zostavovanie dotazov sa využíva návrhový vzor Builder, čo výrazne túto činnosť uľahčuje. Podobným spôsobom boli vytvorené metódy pre vyhľadávanie projektov a pracovných záznamov.

```java
@Override
public List<UserDocument> searchUsers(String keyword) {
    QueryBuilder queryBuilder = QueryBuilders.boolQuery()

          .should(
           new MultiMatchQueryBuilder(keyword)
          .fuzziness(Fuzziness.AUTO))
          .fields(getUserFieldsForSearchByKeyword())

          .should(
           new QueryStringQueryBuilder(
               StringUtils.wrapWordsInAsterisks(keyword))
          .fields(getUserFieldsForSearchByKeyword()));

    List<UserDocument> users =
      elasticsearchService.search(usersIndexName, queryBuilder,
      searchHit ->
          elasticsearchDocumentsMapper.mapHitToUser(searchHit));
    return users;
}

private Map<String, Float> getUserFieldsForSearchByKeyword() {
    Map<String, Float> userFields = new HashMap<>();
    userFields.put(NAME, NAME_BOOST);
    userFields.put(SURNAME, SURNAME_BOOST);
    return userFields;
}
```

Ukážka kódu 3.4: Vyhľadávanie uživateľov pomocou kľúčového slova

3.1.3 Jira

Modul Jira integruje software Jira pomocou knižnice Jira Java API [43]. Základné rozhranie, ktoré modul Jira vystavuje sa nachádza v ukážke 3.5. Pseudokód implementácie tohto rozhrania je v podstate jednoduchý:

1. Zistím, či uživateľ existuje v Jira s danou emailovou adresou. Ak nie, vyhodím výnimku.
2. Pomocou dotazu JQL získam všetky úlohy (Issues) v danom časovom období, na ktorých daný uživateľ vykázal prácu. Formát takéhoto dotazu je `worklogAuthor = %s AND worklogDate >= %s AND worklogDate < %s`.
3. Zo získaných úloh vyextraktujem všetky pracovné záznamy, vložím ich do mapy, v ktorej kľučom je daná úloha (Issue) a výsledok vrátim.

Aby sme boli schopní využívať Jira API je potrebné sa najprv autorizovať. Jira ponúka niekoľko spôsobov autentizácie (OAuth, Basic, Cookie-based).

Pre naše účely sme zvolili Basic autentizáciu kôli tomu, že náš systém nemení údaje v Jire, údaje len číta a nie je potrebné aby pri dotazovaní Jira API vystupoval vždy iný uživateľ, ktorý by bol inak autentizovaný napr. pomocou protokolu OAuth. Vytvorili sme preto nového uživateľa priamo v Jire, ktorému sme dali len právo čítanie. S týmto uživateľom sa náš systém vždy autentizuje a výsledná implementácia je jednoduchšia. Serverová časť systému priamo nevystavuje uživateľovi API pre interakciu s týmto rozhraním, prístupy uživateľa sú riešené na úrovni modulu Rest a Security a Jira API sa volá vždy interne. Takže využitie tohto spôsobu autentizácie je v našom prípade bezpečné.

```
public interface JiraWorklogService {
    Map<Issue, List<Worklog>> findWorklogsByUserEmail(String email);
    Map<Issue, List<Worklog>> findWorklogsByUserEmail(String email,
        LocalDate fromInclusive);
    Map<Issue, List<Worklog>> findWorklogsByUserEmail(String email,
        LocalDate fromInclusive, LocalDate toExclusive);
}
```

Ukážka kódu 3.5: Základné rozhranie modulu Jira

3.1.4 User, Project, Work Record

Moduly User, Project, Work Record implementujú základnú logiku pre vytváranie, aktualizovanie, mazanie a získavanie údajov (CRUD operácie) a kontrolujú obmedzenia systému. K takýmto obmedzeniam patrí napríklad to, že uživateľ nemôže byť priradený na ten istý projekt v rôznych časoch, ktoré by sa prekrývali. Hlavná časť metódy, v ktorej je táto logika implementovaná, s návratovym typom `boolean` vracia hodnotu `true` v prípade vzniku konfliktu sa nachádza v ukážke 3.6.

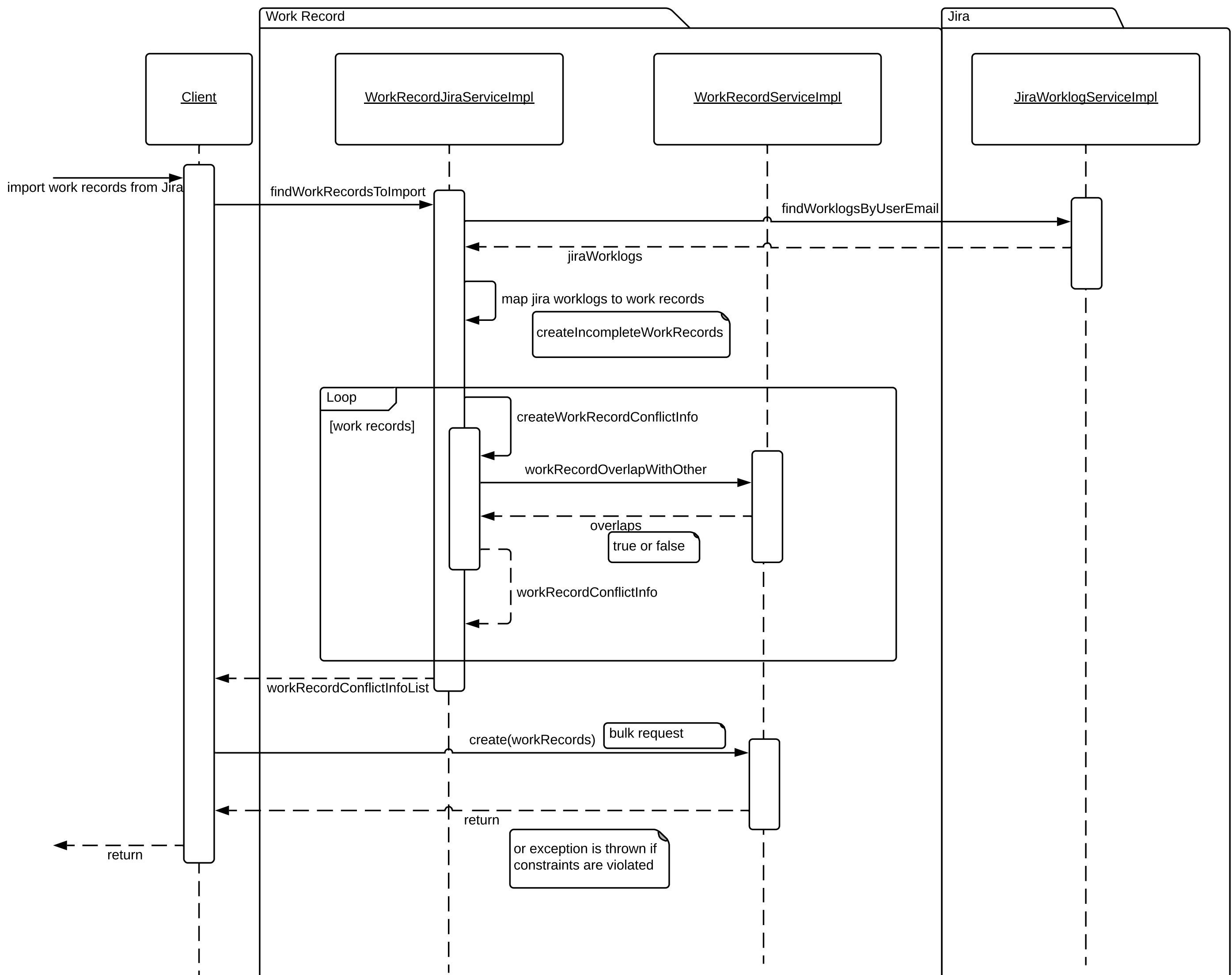
```
for (ProjectAssignmentDTO projectAssignment : projectAssignments) {
    if (projectAssignment.getId().equals(projectAssignmentId)) {
        continue;
    }

    if (projectAssignment.getValidTo() == null) {
        if (validTo == null ||
            !validTo.isBefore(projectAssignment.getValidFrom())) {
            return true;
        }
    } else if (!(validFrom.isAfter(projectAssignment.getValidTo()) ||
        (validTo != null &&
        validTo.isBefore(projectAssignment.getValidFrom())))) {
        return true;
    }
}
return false;
```

Ukážka kódu 3.6: Kontrola pri priradzovaní uživateľa na projekt

Podobne kontrolujeme aj logiku vytvárania pracovných záznamov. Chceme zabezpečiť aby nedošlo k vytvoreniu pracovného záznamu na projekt, na ktorom uživateľ skončil resp. doba validity priradenia projektu vypršala. Ďalej kontroluje aby sa pracovné záznamy neprekrývali a typ práce bol vždy v súlade s vybraným projektom. Táto logika sa nachádza v triede `WorkRecordServiceImpl` v metóde `checkWorkRecordConstraintsOrThrow`.

Modul Work Record využíva modul Jira a vystavuje rozhranie pre proces importovania pracovných výkazov zo softwaru Jira. Pre rýchlejšie pochopenie tohto procesu je na obrázku 3.2 nakreslený sekvenčný diagram. Vec, ktorá môže byť v diagrame nejasná je objekt `workRecordConflictInfo`. Nie je to nič iné ako samotný pracovný záznam s informáciou o tom, či je pracovný záznam v konfikte s už existujúcim pracovným záznamom alebo nie (buď sa časy prekrývajú alebo nie). Vďaka tejto informácii vie klient tieto pracovné záznamy odfiltrovať, poprípade ponúknuť uživateľovi možnosť úpravy daného pracovného záznamu. Následne si uživateľ vyberie záznamy a k nim odpovedajúce projekty ktoré chce importovať a klient zavolá metódu pre hromadné vytvorenie pracovných záznamov. Automatické importovanie pracovných záznamov nebolo zvolené kôli tomu, že nie je zaručená konzistentnosť projektov medzi softwarom Jira a našou serverovou časťou systému, na ktorých je práca vykazovaná. Riešením by bolo vytvoriť spôsob konfigurácie, pomocou ktorej by sme potom vedeli určiť k akému projektu sa pracovný záznam z Jiry vzťahuje. Avšak túto konfiguráciu by musel vytvárať sám uživateľ a vzdialili by sme sa od dodržiavania princípu KISS [44].



Obr. 3.2: Sekvenčný diagram importovania pracovných záznamov z Jira

3.1.5 Report

V rozhraní, ktoré vystavuje modul Report definujeme päť základných typov metód nachádzajúcich sa v ukážke 3.7. Tieto metódy sú preťažené s argumentami, pomocou ktorých zmenšujeme veľkosť množiny výsledkov napr. identifikátor užívateľa alebo projektu. Pre prehľadnosť, preťažené metódy nie sú zahrnuté v následujúcej ukážke.

```
public interface ReportService {
    List<DayReportItem> createDailyReports(
                            LocalDate from,
                            LocalDate to);

    List<MonthReportItem> createMonthlyReports(
                            LocalDate from,
                            LocalDate to);

    List<YearReportItem> createYearlyReports(
                            LocalDate from,
                            LocalDate to);

    List<UserReportItem> createUsersReports(
                            LocalDate from,
                            LocalDate to);

    List<ProjectReportItem> createProjectsReports(
                            LocalDate from,
                            LocalDate to);

    ...
}
```

Ukážka kódu 3.7: Rozhranie modulu Report

Základnou jednotkou návratových typov týchto metód je trieda `WorkReportItem` 3.8. Predstavuje vykonanú prácu, bez ohľadu na projekt, na ktorom bola práca vykázaná či uživateľa. Chceme vedieť len typ práce a čas trvania. Túto triedu tak môžeme používať v špecifickejších triedach ako je napr. `ProjectReportItem` – špecifikuje projekt, na ktorom sa pracovalo. Triedu `ProjectReportItem` taktiež využívame v triedach, ktoré špecifikujú časovú periódu a to buď denne, mesačne alebo ročne, resp. ich odpovedajúce triedy `DayReportItem`,`MonthReportItem` a `YearReportItem`. Implementovaná logika využíva modul Work Record, pomocou ktorého získava pracovné záznamy. Ďalej zabezpečuje, aby sa pracovné záznamy správne zoskupili podľa danej periódy v určitom období, daného projektu či uživateľa. Klientovi využívajúcemu rozhranie modulu Report tak poskytujeme všetky potrebné údaje, ktoré môže zobraziť napr. v rôznych grafoch v GUI alebo ďalej spracovávať. V našom prípade ho využíva modul Rest, ktorý tieto údaje len deleguje a sprístupňuje pomocou vystavených endpointov – vstupných bodov REST služby.

```
public class WorkReportItem {

    private WorkType workType;
    private Integer minutesSpent;

}
```

Ukážka kódu 3.8: Trieda WorkReportItem

3.1.6 Rest, Security

Modul Rest vystavuje spolu 45 REST API metód pomocou tried (kontrolerov) označených anotáciou `@RestController` z knižnice Spring Web MVC. Tieto metódy sme rozdelili do 7 kontrolérov nachádzajúcich sa v balíčku `cz.cvut.fit.timetracking.rest.controller`, ku ktorým odpovedajú ich URI. V tabuľke 3.1 sú zachytené len základné REST API metódy týchto kontrolerov. Tieto metódy predstavujú korene URI, ktoré sú v ďalších metódach rozšírené.

URI	HTTP request metódy	Trieda
/work-records	`GET, POST, PUT, DELETE`	WorkRecordController
/projects	`GET, POST, PUT, DELETE`	ProjectController
/users	`GET, PUT, DELETE`	UserController
/project-assignments	`GET, POST, PUT, DELETE`	ProjectAssignmentController
/search	`GET`	SearchController
/work-types	`GET, POST, PUT, DELETE`	WorkTypeController
/reports	`GET`	ReportController

Tabuľka 3.1: Základné REST API metódy

Modul Rest je úzko spojený s modulom Security, ktorý využíva knižnicu Spring Security a obsahuje logiku pre riadenie prístupu k jednotlivým REST API metódam. Rozhranie triedy, ktorá túto logiku implementuje sa nachádza v ukážke 3.9. Túto logiku potrebujeme najmä preto, lebo potrebujeme overiť validitu prihláseného uživateľa v rámci jeho projektových rolí a overiť, či prihlásený uživateľ je vlastníkom pristupovaného zdroja (resource) alebo nie. Takto umožníme napr. projektovému manažérovi upraviť pracovný záznam inému uživateľovi v rámci projektu. Zdrojový kód tohto príkladu sa nachádza v ukážke 3.10. Jedná sa o deklaráciu REST metódy v triede `WorkRecordController`, ktorá umožňuje upraviť pracovný záznam. Všimnúť si môžeme anotáciu `@PutMapping`, ktorá hovorí, že sa jedná o PUT HTTP metódu. Najpodstatnejšia je anotácia `@PreAuthorize`. V nej využívame triedu `SecurityAccessService` a špecifikujeme podmienky, ktoré pri úspešnom vyhodnotení povolia prístup k metóde a daná metóda sa realizuje. V opačnom prípade je uživateľovi vrátený

HTTP status kód 403 – Forbidden (zakázaný). Tieto podmienky sa vyhodnocujú vždy pred volaním danej metódy a špecifikujeme ich pomocou SpEL – Spring Expression Language [45]. O volanie `@PreAuthorize` podmienok, vždy pred volaním danej metódy, sa stará Spring AOP (Aspect Oriented Programming) [46]. Takýmto spôsobom riadime prístup ku všetkým REST API metódam, ktoré modul Rest vystavuje.

```
public interface SecurityAccessService {
    boolean itIsMe(Integer userId);
    boolean hasAnyProjectRole(Integer projectId, String... roles);
    boolean hasProjectRole(Integer projectId, String role);
    boolean itIsMeOrNull(Integer userId);
    boolean workRecordIsMineOrHasProjectRole(Integer workRecordId,
        String role);
}
```

Ukážka kódu 3.9: Trieda SecurityAccessService

```
@PutMapping("/{id}")
@PreAuthorize("hasAuthority('ADMIN') or
    @securityAccessServiceImpl.workRecordIsMineOrHasProjectRole(#id,
    'PROJECT_MANAGER')")
public ResponseEntity<WorkRecordDTO> update(@PathVariable("id")
    Integer id, @Valid @RequestBody CreateOrUpdateWorkRecordRequest
    request) {
    WorkRecord workRecord = workRecordService.update(id,
                            request.getDateFrom(),
                            request.getDateTo(),
                            request.getDescription(),
                            request.getProjectId(),
                            request.getWorkTypeId());
    return ResponseEntity.ok(map(workRecord));
}
```

Ukážka kódu 3.10: REST metóda pre úpravu pracovného záznamu

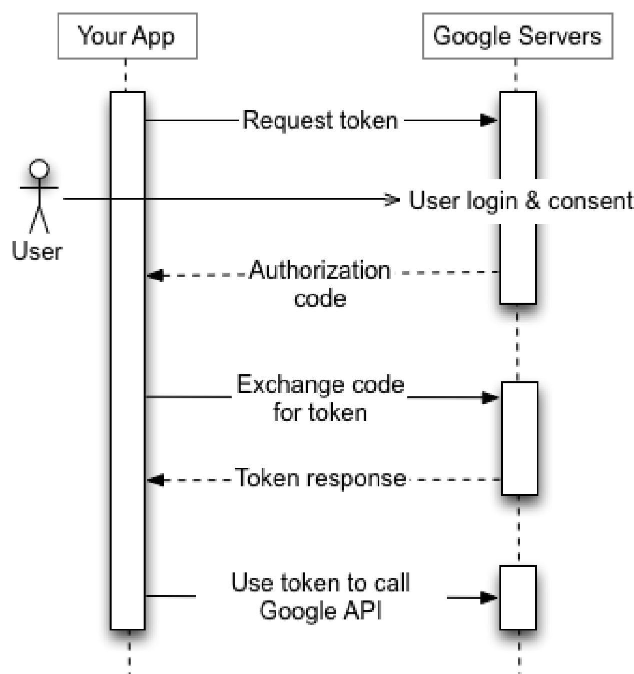
Samotné prihlásenie užívateľa prebieha s účtom Google pomocou protokolu OAuth 2.0. Z veľkej časti nám v tomto procese pomáha knižnica Spring Security, pomocou ktorej vystavujeme dva endpointy `/oauth2/authorize` a `/oauth2/callback` a špecifikujeme konfiguráciu 3.11, v ktorej sú prístupove údaje aplikácie vytvorenej pre testovacie účely v Google konzoli pre vývojárov [47]. Ukážka tejto konfigurácie sa nachádza na obrázku 3.3. Pri volaní autorizačného endpointu `/oauth2/authorize`, Spring Security sám zabezpečí presmerovanie na prihlasovaciu stránku Google s požiadavkom o získanie tokenu, užívateľ sa prihlási a následne nás Google presmeruje a zavolá sa endpoint `/oauth2/callback`. V tomto endpointe sa po úspešnom prihlásení spracuje odpoveď a zavolá Google API pre získanie údajov užívateľa (meno, priezvisko,

email, URL fotografie), viď. sekvenčný diagram 3.4. Tieto údaje uživateľa si potom uložíme v databáze a predstavuje to spôsob registrácie.

```
spring:
 security:
  oauth2:
   client:
    registration:
     google:
      clientId:
          185475724985-bc71ltf88i5rvfrcnvvp51n2l5u2lfen.apps.googleusercontent.com
      clientSecret: UfLq1xByllR6SVnJ8dFvJ5XE
      redirectUriTemplate:
          "http://localhost:${server.port}/oauth2/callback/{registrationId}"
      scope:
       - email
       - profile
```

Ukážka kódu 3.11: Konfigurácia Google OAuth 2.0 – server

Obr. 3.3: Konfigurácia aplikácie Google OAuth 2.0 v Google konzole



Zdroj: `https://developers.google.com/identity/protocols/OAuth2`

Obr. 3.4: Sekvenčný diagram autorizácie Google OAuth 2.0

REST je stateless – bezstavová webová služba. To znamená, že server si neukladá žiadny stav o relácii klienta na strane servera [48]. Po úspešnej autorizácii užívateľa s Google nastáva otázka, ako si túto reláciu udržíme medzi klientom a našou serverovou časťou. Pre tento účel využijeme JWT – JSON Web Token [49]. Zjednodušený princíp fungovania JWT je, že klientovi v našom prípade pošleme identifikačný údaj užívateľa a dobu expirácie relácie. Tieto údaje následne digitálne podpíšeme (vytvoríme hash) pomocou tajného kľúča (secret key) napr. algoritmom HMAC [50]. Údaje sú vo formáte JSON, ktoré sa následne kódujú pomocou Base64 a to je výsledný token. Identifikačný údaj užívateľa a dobu expirácie nemusíme šifrovať. Hash je ďalším parametrom tokenu. Klient tento token odosiela v každom požiadavku pri následných volaniach REST API. Serverová časť, pri prijatí požiadavky, dekóduje token z Base64, vytvorí hash z identifikačného údaju užívateľa a doby expirácie a následne porovná s hash z tokenu. Takto vieme overiť integritu údajov a pri nezhode vrátime užívateľovi HTTP status kód 401 – Unauthorized. V zdrojovom kóde za tento proces zodpovedajú triedy `TokenServiceImpl`, `OAuth2AuthenticationSuccessHandler`, a `TokenAuthenticationFilter`.

3.2 Logstash

Synchronizáciu údajov medzi databázou a Elasticsearch zabezpečuje Logstash. Vytvorili sme celkovo šesť konfigurácii Logstash pipeline, ktoré využívajú JDBC plugin pre čítanie údajov z databázy a Elasticsearch plugin pre zápis údajov do Elasticsearch. Tieto Logstash pipeline sa pravidelne a automaticky, v našom prípade každú minútu, spúšťajú a dotazujú sa na údaje z tabuliek `project`, `work_record` a `user`.

Aby nedochádzalo k zbytočnému prelievaniu všetkých údajov z databázy do Elasticsearch každú minútu, bolo potrebné vymyslieť spôsob, ktorý zabezpečí, aby sa do Elasticsearch premietli len nové dátové zmeny t.j. vytvorenie, aktualizovanie alebo zmazanie záznamu. Vytvorili sme preto pomocnú tabuľku v databáze s názvom `changelog`, ktorú sme opísali v sekcii 2.8. Údaje do tejto tabuľky sa vkladajú automaticky, pomocou databázového triggeru 3.12, ktorý je aplikovaný na tabuľky `project`, `work_record` a `user`. Logstash pipeline sa nad touto tabuľku dotazuje a ak pribudli nové záznamy, tak podľa stĺpca `action` zistí akú operáciu má vykonať v Elasticsearch.

V ukážkach 3.13 a 3.14 sa nachádzajú zdrojové kódy konfigurácie Logstash pipeline, ktoré zodpovedajú za správne vytvorenie, úpravu a zmazanie projektov v Elasticsearch (rovnakým spôsobom sú vytvorené konfigurácie pre pracovné výkazy a uživateľov). Všimnúť si môžeme parameter `statement`, v ktorom špecifikujeme SQL dotaz pre získanie údajov, ktoré chceme spracovať a poslať do Elasticsearch. Parameter, ktorý v SQL dotaze zohráva veľkú rolu je `sql_last_value`. Tento parameter obsahuje dátum a čas posledného spustenia SQL dotazu. Ukládanie hodnoty do tohto parametra má na starosti Logstash JDBC vstupný (input) plugin. Vďaka tomúto patametru vieme získať vždy len novo vzniknuté záznamy. V konfigurácii výstupného (output) pluginu elasticsearch si ďalej môžeme všimnúť parameter `action`, ktorý špecifikuje akciu, ktorá sa má vykonať v Elasticsearch. Využívame akcie `update` a `delete`. Akcia `update` aktualizuje dokument v Elasticsearch ak existuje. V opačnom prípade overí parameter `doc_as_upsert` a ak je hodnotoa `true` tak dokument vytvorí. Akcia `delete` dokument odstráni.

```
CREATE OR REPLACE FUNCTION process_changelog() RETURNS TRIGGER
BEGIN
   IF (TG_OP = 'DELETE') THEN
      INSERT INTO
          time_tracking_changelog_schema.changelog(id_row_changed,
          "table", "action", stamp) VALUES (OLD.id, TG_ARGV[0],
          'delete', now());
      RETURN OLD;
      ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO
            time_tracking_changelog_schema.changelog(id_row_changed,
            "table", "action", stamp) VALUES (OLD.id, TG_ARGV[0],
            'update', now());
```

```
      RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
      INSERT INTO
          time_tracking_changelog_schema.changelog(id_row_changed,
          "table", "action", stamp) VALUES (NEW.id, TG_ARGV[0],
          'create', now());
      RETURN NEW;
    END IF;
    RETURN NULL;
END;
```

Ukážka kódu 3.12: Databázový trigger, plniaci tabuľku changelog

```
input {
  jdbc {
     id => "input_projects_upsert"
     jdbc_driver_library => "${JDBC_DRIVER_LIBRARY}"
     jdbc_driver_class => "${JDBC_DRIVER_CLASS}"
     jdbc_connection_string => "${JDBC_CONNECTION_STRING}"
     jdbc_user => "${JDBC_USER}"
     jdbc_password => "${JDBC_PASSWORD}"
     schedule => "* * * * *"
     statement =>
       "SELECT p.* FROM time_tracking_schema.project p
        WHERE p.id IN
          (SELECT c.id_row_changed
           FROM time_tracking_changelog_schema.changelog c
           WHERE c.action != 'delete' AND
                 c.table = 'project' AND
                 c.stamp > :sql_last_value)"
  }
}

filter {
}

output {
  elasticsearch {
     id => "output_projects_upsert"
     hosts => ["${ELASTICSEARCH_HOST}"]
     action => update
     doc_as_upsert => true
     document_id => "%{id}"
     index => "${ELASTICSEARCH_PROJECTS_INDEX_NAME}"
  }
}
```

Ukážka kódu 3.13: Konfigurácia Logstash pipeline pre projekty - vytvorenie, aktualizácia

```
input {
  jdbc {
     id => "input_projects_delete"
     jdbc_driver_library => "${JDBC_DRIVER_LIBRARY}"
     jdbc_driver_class => "${JDBC_DRIVER_CLASS}"
     jdbc_connection_string => "${JDBC_CONNECTION_STRING}"
     jdbc_user => "${JDBC_USER}"
     jdbc_password => "${JDBC_PASSWORD}"
     schedule => "* * * * *"
     statement => "SELECT c.id_row_changed AS id
                  FROM time_tracking_changelog_schema.changelog c
                  WHERE c.action = 'delete' AND
                        c.table = 'project' AND
                        c.stamp > :sql_last_value"
     }
}

filter {
}

output {
  elasticsearch {
     id => "output_projects_delete"
     hosts => ["${ELASTICSEARCH_HOST}"]
     action => delete
     document_id => "%{id}"
     index => "${ELASTICSEARCH_PROJECTS_INDEX_NAME}"
  }
}
```

Ukážka kódu 3.14: Konfigurácia Logstash pipeline pre projekty - odstraňovanie

3.3 Klientska aplikácia

Klientska aplikácia, demonštrujúca proces autorizácie pomocou Google, je vyvíjaná pomocou frameworku React v jazyku JavaScript. Využívame taktiež pomocné knižnice `react-router-dom` pre smerovanie len na strane klienta (client-side routing) a `react-s-alert` pre zobrazovanie upozornení. V nasledujúcich sekciách si popíšeme základné komponenty aplikácie.

3.3.1 index.js

Základným prístupovým bodom aplikácie je `index.js` 3.15. Renderuje komponentu `App` v DOM elemente, ktorá je obalená v smerovacej komponente `Router` zodpovedajúcej za smerovanie na strane klienta.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './app/App';
import * as serviceWorker from './serviceWorker';
import { BrowserRouter as Router } from 'react-router-dom';

ReactDOM.render(
    <Router>
        <App />
    </Router>,
    document.getElementById('root')
);

serviceWorker.unregister();
```

Ukážka kódu 3.15: Zdrojový kód index.js

3.3.2 App

`App` komponenta, nachádzajúca sa v súbore `src/app/App.js`, je najhlavnejšou komponentou a definuje základné rozdelenenie stránky a smerovania (routes) aplikácie. Ďalej obsahuje logiku, ktorá načíta detaily aktuálne overeného užívateľa zo serverovej časti systému a získané údaje odovzdáva do detských komponent. V ukážke 3.16 sa nachádza časť kódu zo súboru `App.js`, v ktorom definujeme rozdelenie a smerovania aplikácie.

```
<div className="app">
  <div className="app-top-box">
    <AppHeader authenticated={this.state.authenticated}
        onLogout={this.handleLogout} />
  </div>
  <div className="app-body">
    <Switch>
      <Route exact path="/" component={Home}></Route>

      <PrivateRoute path="/profile"
          authenticated={this.state.authenticated}
          currentUser={this.state.currentUser} component={Profile}>
      </PrivateRoute>

      <Route path="/login" render={(props) =>
         <Login authenticated={this.state.authenticated} {...props}
            />}>
      </Route>

      <Route path="/oauth2/redirect"
          component={OAuth2RedirectHandler}>
```

```
    </Route>

    <Route component={NotFound}></Route>
   </Switch>
  </div>
  <Alert stack={{limit: 3}}
   timeout = {3000}
   position='top-right' effect='slide' offset={65} />
</div>
```

Ukážka kódu 3.16: Rozdelenie a smerovania aplikácie

3.3.3 Login

Komponenta `Login` umožňuje používateľovi prihlásiť sa pomocou poskytovateľa Google použitím OAuth 2.0 protokolu. Po prechode na túto komponentu sa zobrazí prihlasovacie Google tlačítko a po kliknutí sa zavolá autorizačný endpoint serveru `http://localhost:8080/oauth2/authorize/google?redirect_uri=http://localhost:3000/oauth2/redirect`, ktorý následne presmeruje používateľa na Google prihlasovací formulár.

3.3.4 OAuth2RedirectHandler

Komponenta `OAuth2RedirectHandler` je načítaná v momente, keď užívateľ dokončí proces autorizácie so serverom. Ak sa užívateľ úspešne autorizoval, server ho následne presmeruje na komponentu `Profile` s prístupovým tokenom. V opačnom prípade je užívateľ presmerovaný opäť na komponentu `Login`. Ukážka kódu tejto logiky si môžeme všimnú v následujúcej ukážke 3.17.

```
const token = this.getUrlParameter('token');
const error = this.getUrlParameter('error');

if(token) {
  localStorage.setItem(ACCESS_TOKEN, token);
  return <Redirect to={{
    pathname: "/profile",
    state: { from: this.props.location }
  }}/>;
} else {
  return <Redirect to={{
    pathname: "/login",
    state: {
      from: this.props.location,
      error: error
    }
  }}/>;
```

```
}
```

Ukážka kódu 3.17: Presmerovanie uživateľa po autorizácii

3.3.5 Profile

Úlohou komponenty `Profile` je zobraziť údaje úspešne autorizovaného uživateľa.

3.4 Nasadenie a spustenie

Pre nasadenie a spustenie aplikácie a všetkých využívaných technológií stačí mať nainštalované technológie `Docker` a `Docker Compose`. Aktuálna dokumentácia inštalácie technológie `Docker` je dostupná na adrese `https://docs.docker.com/install` a technológie `Docker Compose` na adrese `https://docs.docker.com/compose/install`. Po inštalácii stačí otvoriť koreňovú zložku projektu, t.j. zložka, v ktorej sa nachádza konfiguračný `Docker Compose` súbor s názvom `docker-compose.yml` a spustiť shell príkaz `docker-compose up`. O všetko ostatné sa postará `Docker` spolu s `Docker Compose`, ktorý vytvorí kontajner pre každú aplikáciu/technológiu. Ďalej vytvorí spoločnú sieť, ktorá umožňuje aby kontajnery mohli medzi sebou komunikovať. Po určitom čase, keď budú stiahnuté všetky závislosti a zostavené kontajnery, bude klientska aplikácia dostupná na adrese `http://localhost:3000`, serverové REST API na adrese `http://localhost:8081`, Jira na adrese `http://localhost:8080` atď.

Definícia konfigurácie `Docker Compose` a všetkých jej služieb sa nachádza v ukážke 3.18. Pri každej službe (aplikácii/technológii), v parametri `build`, špecifikujeme zložku s odpovedajúcim `Dockerfile` súborom. Dôležitý parameter je parameter `ports`, ktorý určuje na aký port hostiteľského počítača sa má namapovať port danej bežiacej služby. Preto je potrebné pred spustením overiť, či tieto porty na hostiteľskom počítači nie sú obsadené a prípadne ich uvoľniť.

Každá zložka, nachádzajúca sa v koreňovej zložke projektu, predstavuje určitú technológiu/aplikáciu a obsahuje vlastný konfiguračný súbor `Docker` kontajneru s názvom `Dockerfile`. Príklad `Dockerfile` súboru, pomocou ktorého konfigurujeme kontajner serverovej časti sa nachádza v ukážke 3.19. V ňom špecifikujeme inštrukcie, ktoré `Docker` interpretuje a vytvorí obraz – Docker image. Špecifikujeme inštrukciu `FROM`, ktorá určuje z akého obrazu (Docker image) budeme vychádzať. Serverová časť bola vyvíjana v jazyku Java s verziou 11, takže hodnotu inštrukcie volíme `openjdk:11`. Následuje stiahnutie aplikácie `dockerize`, pomocou ktorej v inštrukcii `CMD` prikážeme, aby serverová aplikácia počkala na spustenie databázy a softwaru Jira pred samotným spustením. V ďalších inštrukciách vytvárame zložku, do ktorej sa prekopíruje zdrojový kód serveru, aplikácia sa zostaví pomocou `Gradle` a následne zosta-

vený súbor vo formáte JAR čaká na spustenie v inštrukcii `CMD`. Ak hodnota v premennej prostredia `RUN_TESTS` je `true`, tak aplikácia sa spustí aj s testami.

```
version: '3.1'
services:
  db:
    build: ./postgres
    ports:
      - "5432:5432"
  time-tracking-backend:
    build:
      context: ./time-tracking-backend
      args:
        - RUN_TESTS=false
    ports:
      - "8081:8081"
    depends_on:
      - "db"
      - "elasticsearch"
      - "jira"
  google-login-poc-app:
    build: ./google-login-poc-app
    ports:
      - "3000:3000"
  jira:
    build: ./jira
    ports:
      - "8080:8080"
  elasticsearch:
    build: ./elasticsearch
    environment:
      - cluster.name=time-tracking-cluster
      - discovery.type=single-node
    ports:
      - "9200:9200"
      - "9300:9300"
  logstash:
    build: ./logstash
    depends_on:
      - "elasticsearch"
      - "db"
  sonarqube:
    build: ./sonarqube
    ports:
      - "9000:9000"
```

Ukážka kódu 3.18: Konfigurácia Docker Compose

```
FROM    openjdk:11
```

```
ENV DOCKERIZE_VERSION v0.6.0
RUN wget https://github.com/jwilder/dockerize/
        releases/download/$DOCKERIZE_VERSION/
        dockerize-linux-amd64-$DOCKERIZE_VERSION.tar.gz \
    && tar -C /usr/local/bin -xzvf
        dockerize-linux-amd64-$DOCKERIZE_VERSION.tar.gz \
    && rm dockerize-linux-amd64-$DOCKERIZE_VERSION.tar.gz

RUN     mkdir /home/time-tracking
WORKDIR /home/time-tracking/
COPY    . ./
RUN     ./gradlew clean build -x test

EXPOSE 8081
CMD if $RUN_TESTS ; then
  dockerize
   -wait tcp://db:5432 -timeout 60s
   -wait http://jira:8080 -timeout 320s
   ./gradlew test --info &&
  java -jar
      /home/time-tracking/app/build/libs/time-tracking-app-1.0-SNAPSHOT.jar
  else
   dockerize -wait tcp://db:5432 -timeout 60s
   java -jar
       /home/time-tracking/app/build/libs/time-tracking-app-1.0-SNAPSHOT.jar
  fi
```

Ukážka kódu 3.19: Dockerfile serverovej časti systému

3.4.1 Spustenie Jira

Keďže Jira nie je nástroj zdarma, je potrebné mať licenciu. Počas vývoja a testovania som použil 30 dennú licenciu, ktorá pri prvotnom spustení softwaru Jira v kontajneri nie je dostupná a je potrebné si ju vygenerovať. Okrem toho je potrebné, pri prvotnom spustení, samotný software Jira nakonfigurovať. Základná konfigurácia Jira, ktorá nám pre spustenie vystačí je jednoduchá a v podstate stačí len jedno kliknutie na tlačítko. Po spustení kontajnerov stačí prejsť na adresu `http://localhost:8080`, na ktorej sa zobrazí návod.

Ďalej je potrebné vytvoriť užívateľa, ktorého užívateľské údaje (ak budú rôzne od stávajúcich) sa musia upraviť v konfigurácii našej serverovej časti a to v súbore `time-tracking-backend/app/src/main/resources/application.yml`, viď. 3.20. To isté platí aj pre testy, t.j. súbory `time-tracking-backend/rest/src/test/resources/application.yml` a `time-tracking-backend/jira/src/test/resources/application.yml`.

```
jira:
```

```
restClient:
  serverUri: http://jira:8080
  authentication:
    username: time.tracking
    password: Q557hKbtsRT9yf6
```

Ukážka kódu 3.20: Jira konfigurácia

Kapitola 4

Testovanie

V tejto kapitole sa zameriame na testovanie serverovej časti systému a klientskej aplikácie.

4.1 Server

Pre testovanie serverovej časti systému sme využili knižnicu JUnit 4.12, ktorá spolu s frameworkom Spring ponúka nástroje uľahčujúce testovanie. Celkovo bolo vytvorených 123 testov, ktoré sú umiestnené a rozdelené v každom module. Napríklad v module `Search` testujeme implementáciu a integráciu technológie `Elasticsearch`, v module `Data` testujeme repozitáre a napojenie na databázu `PostgreSQL` apod.

Testy zamerané na otestovanie funkčnosti REST API sú intergračné testy a nachádzajú sa v module `Rest`. Pri každom teste je odoslaná HTTP požiadavka na REST API, pričom sa kontroluje, či pre daný vstup vráti očakávané dáta. Odosielanie HTTP požiadaviek a kontrolovanie výsledkov nám umožňuje trieda `MockMvc` z frameworku Spring. Príklad sa nachádza v ukážke 4.1. Všimnúť si môžeme anotácie `@Sql`, ktorá špecifikuje SQL skript spúšťací sa vždy pred alebo po testovacej metóde. Týmto spôsobom pripravujeme testovacie dáta. Ďalej pomocou anotácie `@WithMockOAuth2AuthenticationToken` vytvárame dočasný mock objekt, ktorý predstavuje autorizovaného uživateľa. Vďaka tomu vieme otestovať role uživateľov a ich prístupy k určitým zdrojom. Táto anotácia spolu s jej Factory triedou sa nachádza v balíčku:
`cz.cvut.fit.timetracking.rest.context`.

```
@Sql(scripts =
    "/sql_initialization_test_scripts/insert_for_integration_tests.sql",
    executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD)
@Sql(scripts =
    "/sql_initialization_test_scripts/delete_for_integration_tests.sql",
    executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD)
public class ProjectControllerTests
```

```
        extends RestApiTestsConfiguration {

    private static final String PATH = "/projects";

    @Autowired
    private MockMvc mockMvc;

    @Test
    @WithMockOAuth2AuthenticationToken(authorities = {"USER",
        "ADMIN"})
    public void whenGetAllWithAdmin_shouldReturnProjects() throws
        Exception {
        mockMvc.perform(
          get(PATH)
          .contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().isOk())
          .andExpect(jsonPath("$.projects", hasSize(3)));
    }

    @Test
    @WithMockOAuth2AuthenticationToken(authorities = "USER")
    public void whenGetAllWithUser_forbidden() throws Exception {
        mockMvc.perform(
          get(PATH).contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().isForbidden());
    }

    @Test
    @WithMockOAuth2AuthenticationToken(authorities = {"USER",
        "ADMIN"})
    public void whenGetById_shouldReturnProject() throws Exception {
        mockMvc.perform(
          get(PATH + "/-1")
          .contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().isOk())
          .andExpect(jsonPath("$.name", is("test google project")))
          .andExpect(jsonPath("$.id", is(-1)))
          .andExpect(jsonPath("$.workTypes[1].name", is("vyvoj")))
          .andExpect(jsonPath("$.workTypes[0].name", is("analyza")));

    ...
}
```

Ukážka kódu 4.1: Príklad testovania API

Vo väčšine prípadov existujú testy, závislé na niektorú z tried iného modulu. Z tejto triedy môžu získavať dáta, pričom spôsob akým tieto dáta získavajú je buď komplexný alebo závislý na inej technológii akou môže byť databáza. Takáto závislosť je testovaná v integračných testoch a pri testovaní

určitej logiky nie je potrebná. Na to využijeme anotáciu `@MockBean`, ktorá indikuje, že sa jedná o Mock objekt – objekt, ktorý simuluje správanie komplexných, reálnych objektov. Tento objekt si jednoduchým spôsobom nakonfigurujeme tak, aby na daný vstup vrátil požadované dáta a samotná logika môže byť tak testovaná. Príklad týchto testov sa nachádza napr. v module `Report`, v ktorom nie je potrebná závislosť na databázu a získavanie pracovných záznamov. Stačí nám nasimulovať získanie pracovných záznamov nasimulovať a potrebnú logiku otestovať. Príklad testu a konfigurácie Mock objektu sa nachádza v ukážke 4.2 v metóde `init()`.

```
public class ReportServiceTests extends ReportTestsConfiguration {

    @MockBean
    private WorkRecordService workRecordService;

    @Autowired
    private ReportService reportService;

    private final LocalDate from = LocalDate.parse("2019-02-02");
    private final LocalDate to = LocalDate.parse("2019-02-27");

    @Before
    public void init() {
        given(workRecordService.findAllBetween(from.atStartOfDay(),
            to.atStartOfDay()))
        .willReturn(ReportServiceTestsHelper.createWorkRecordsTestCase());
        ...
    }

    @Test
    public void testDailyReports() {
       List<DayReportItem> dayReportItems =
           reportService.createDailyReports(from, to);
       assertThat(dayReportItems).hasSize(3);
       ...
    }
```

Ukážka kódu 4.2: Príklad využitia MockBean

4.1.1 Testy Jira

Ako sme uvádzali v sekcii 3.4.1, pri prvotnom spustení je potrebné nakonfigurovať software Jira. Jedná sa o jednoduchý proces, avšak pár integračných Jira testov počítajú už s existujúcimi testovacími údajmi v Jire. Z tohto dôvodu tieto testy nemusia úspešne prechádzať. Pre úspešné splnenie všetkých testov je potrebné vytvoriť testovacie údaje, ktoré sa však budú zhodovať s očakávanými údajmi v testoch modulu `Jira` a `Rest`. Aby neúspech týchto

testov nespôsoboval ukončenie behu ostatných testov, pridali sme do súboru `build.gradle` príkaz, pomocou ktorého bude ignorovaný neúspešný výsledok testov v moduloch `Jira` a `Rest`, viď. 4.3.

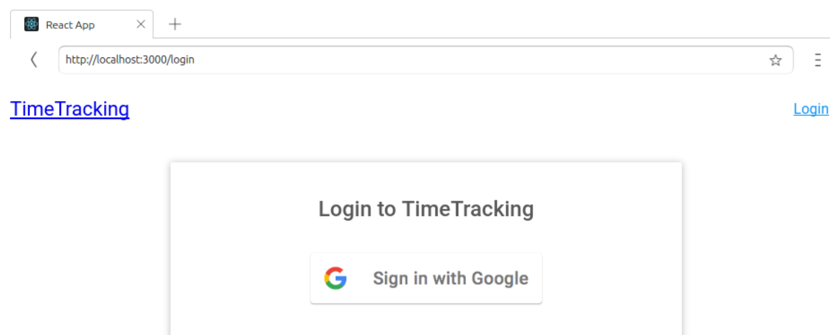
```
1 test {
2     ignoreFailures = true
3 }
```

Ukážka kódu 4.3: Gradle príkaz pre ignorovanie neúspešných testov

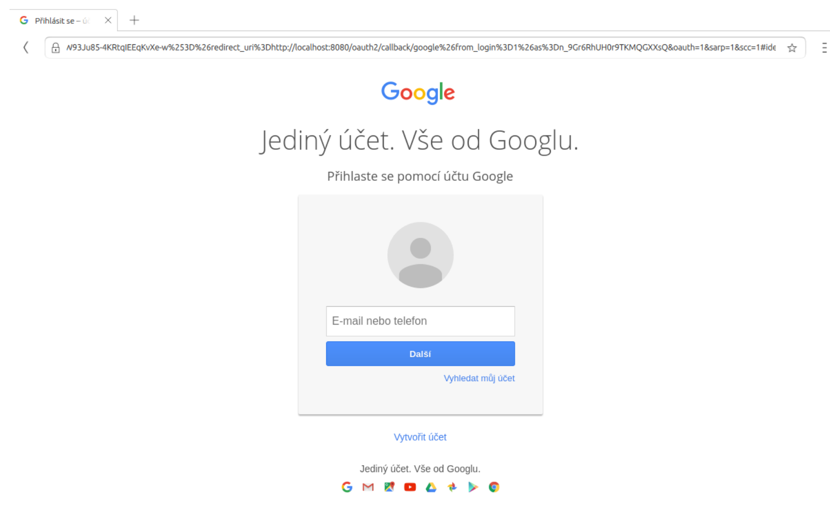
4.2 Klientska aplikácia

Proces prihlasovania pomocou Google účtu cez serverovú časť nášho systému bol otestovaný tzv. Smoke testom. Jedná sa o krátky test, ktorý slúži pre kontrolu chodu systému a k overeniu kľúčových funkčností.

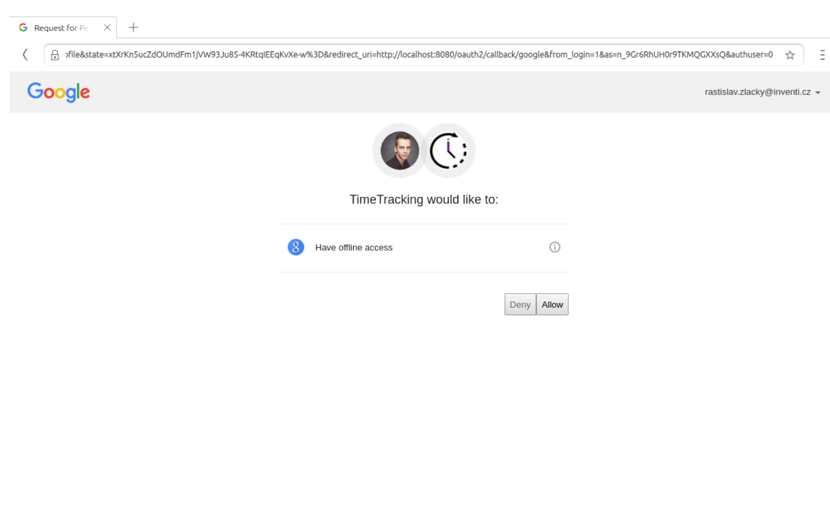
Test začína v momente prechodu na adresu `http://localhost:3000/login`. Užívateľovi sa zobrazí stránka ako na obr. 4.1. Po kliknutí na prihlasovacie tlačítko je užívateľ presmerovaný na prihlasovací formulár stránky Google, viď obr. 4.2. Užívateľ zadá prihlasovacie údaje a následne je požiadaný, aby povolil prístup aplikácii 4.3. Po dokončení je presmerovaný na profil, na adresu `http://localhost:3000/profile` 4.4.



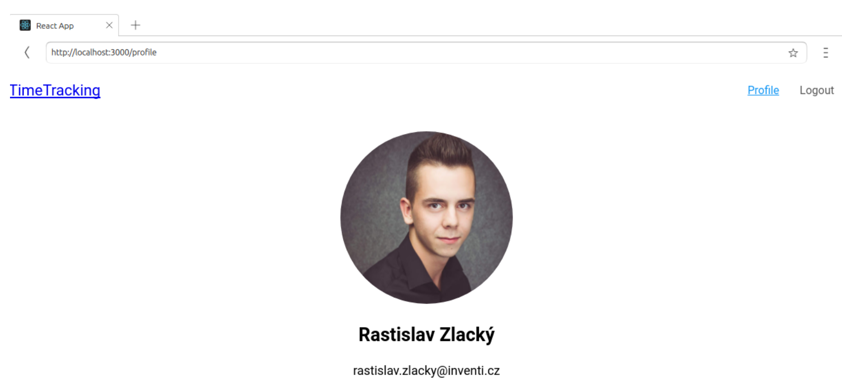
Obr. 4.1: 1. Prihlasovacia stránka



Obr. 4.2: 2. Prihlasovacia stránka Google



Obr. 4.3: 3. Žiadosť o povolenia



Obr. 4.4: 4. Profil užívateľa

Záver

Cieľom bakalárskej práce bola analýza, návrh a implementácia serverovej časti systému pre vykazovanie pracovného času. V rámci analýzy sme preskúmali existujúce riešenia problému, ktoré nám pomohli špecifikovať požiadavky pre naše riešenie a z ktorých sme usúdili, že jednotlivé funkcie sú v princípe rovnaké vo všetkých aplikáciach zameraných na túto problematiku. Výhodou nášho riešenia je v podstate len prístup k zdrojovým kódom a možnosť rozšírenia o ľubovoľnú funkciu. Ďalej sme vymodelovali prípady použitia a navrhli doménový model. Systém sme implementovali, integrovali so systémom Jira pre importovanie a jednotné ukládanie pracovných výkazov a vytvorili koncept klientskej aplikácie, ktorá demonštruje prihlasovanie pomocou účtu poskytovateľa Google použitím OAuth 2.0 protokolu.

Pre prípadné nadväzujúce práce je tu priestor pre vytvorenie plne funkčnej klientskej aplikácie s uživateľským rozhraním. Môže sa jednať o webovú aplikáciu, aplikáciu pre operačný systém Android či iOS. Z pohľadu serverovej časti systému, dá sa systém rozšíriť napr. o business intelligence nástroje, fakturáciu, kontrolu financií apod.

Literatúra

[1] 10 Metrics Every Growing Business Must Keep An Eye On. [online], 2011, [cit. 2019-04-22]. Dostupné z: https://www.forbes.com/sites/martinzwilling/2011/09/28/10-metrics-every-growing-business-must-keep-an-eye-on

[2] Jira. [software], [cit. 2019-04-22]. Dostupné z: https://cs.atlassian.com/software/jira

[3] Proof of Concept (POC), Techopedia. [online], [cit. 2019-04-22]. Dostupné z: https://www.techopedia.com/definition/4066/proof-of-concept-poc

[4] Clockify. [online], 2019, [cit. 2019-04-22]. Dostupné z: https://clockify.me

[5] DeskTime. [online], 2019, [cit. 2019-04-22]. Dostupné z: https://desktime.com

[6] Harvest. [online], 2019, [cit. 2019-04-22]. Dostupné z: https://www.getharvest.com

[7] Stripe, The new standard in online payments. [online], 2019, [cit. 2019-04-22]. Dostupné z: https://stripe.com

[8] Paypal. [online], 2019, [cit. 2019-04-22]. Dostupné z: https://www.paypal.com

[9] Arlow, J.; Neustadt, I.: *UML2 a unifikovaný proces vývoje aplikací.* Computer Press, 2007, ISBN 9788025115039.

[10] Hardt, D.: The OAuth 2.0 authorization framework. [online], 2012, [cit. 2019-05-06]. Dostupné z: http://www.rfc-editor.org/info/rfc6749

[11] Spring Framework Overview. [online], 2019, [cit. 2019-05-02]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html

[12] Core Technologies. [online], 2019, [cit. 2019-05-02]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html

[13] Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern. [online], 2004, [cit. 2019-05-02]. Dostupné z: https://martinfowler.com/articles/injection.html

[14] Spring Projects. [online], 2019, [cit. 2019-05-02]. Dostupné z: https://spring.io/projects

[15] Servlet API Documentation. [online], 2012, [cit. 2019-05-02]. Dostupné z: https://tomcat.apache.org/tomcat-5.5-doc/servletapi

[16] Apache Tomcat. [online], 2019, [cit. 2019-05-02]. Dostupné z: http://tomcat.apache.org/

[17] What is PostgreSQL? [online], 2019, [cit. 2019-05-02]. Dostupné z: https://www.postgresql.org/about/

[18] Welcome to Apache Lucene. [online], 2019, [cit. 2019-05-02]. Dostupné z: https://lucene.apache.org/

[19] Elasticsearch, The Heart of the Elastic Stack. [online], 2019, [cit. 2019-05-02]. Dostupné z: https://www.elastic.co/products/elasticsearch

[20] DB-Engines Ranking of Search Engines. [online], 2019, [cit. 2019-05-02]. Dostupné z: https://db-engines.com/en/ranking/search+engine

[21] Logstash. [online], 2019, [cit. 2019-05-06]. Dostupné z: https://www.elastic.co/products/logstash

[22] Logstash, Input plugins. [online], 2019, [cit. 2019-05-06]. Dostupné z: https://www.elastic.co/guide/en/logstash/current/input-plugins.html

[23] Logstash, Output plugins. [online], 2019, [cit. 2019-05-06]. Dostupné z: https://www.elastic.co/guide/en/logstash/current/output-plugins.html

[24] React. [online], 2019, [cit. 2019-05-06]. Dostupné z: https://reactjs.org/

[25] Docker. [online], 2019, [cit. 2019-05-06]. Dostupné z: https://docs.docker.com/get-started/

[26] What is a Container? [online], 2019, [cit. 2019-05-06]. Dostupné z: https://www.docker.com/resources/what-container

[27] Co jsou nástroje business intelligence (BI)? [online], [cit. 2019-05-07]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/what-are-business-intelligence-tools/

[28] Elasticsearch Resiliency Status. [online], 2019, [cit. 2019-05-07]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/resiliency/current/index.html

[29] Elasticsearch as a primary database. [online], 2017, [cit. 2019-05-07]. Dostupné z: https://discuss.elastic.co/t/elasticsearch-as-a-primary-database/85733

[30] Masse, M.: *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

[31] Fowler, M.: *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[32] Service Provider Interface: Creating Extensible Java Applications. [online], 2009, [cit. 2019-05-15]. Dostupné z: https://www.developer.com/java/article.php/3848881/Service-Provider-Interface-Creating-Extensible-Java-Applications.htm

[33] Hibernate. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://hibernate.org/

[34] Hibernate ORM. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://hibernate.org/orm/

[35] Mihalcea, V.: The best way to map a @OneToMany relationship with JPA and Hibernate. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/

[36] Mihalcea, V.: The best way to use the @ManyToMany annotation with JPA and Hibernate. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://vladmihalcea.com/the-best-way-to-use-the-manytomany-annotation-with-jpa-and-hibernate/

[37] Working with Spring Data Repositories. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories

[38] Paraschiv, E.: Spring Data JPA @Query. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://www.baeldung.com/spring-data-jpa-query

[39] Core concepts. Working with Spring Data Repositories. [online], 2019, [cit. 2019-05-15]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.core-concepts

[40] Java High Level REST Client. [online], 2019, [cit. 2019-05-17]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/client/java-rest/master/java-rest-high.html#java-rest-high

[41] Query DSL. [online], 2019, [cit. 2019-05-17]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html

[42] Gilleland, M.: Levenshtein Distance, in Three Flavors. [online], [cit. 2019-05-17]. Dostupné z: https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm

[43] Jira Java APIs. [online], 2019, [cit. 2019-05-17]. Dostupné z: https://developer.atlassian.com/server/jira/platform/java-apis/

[44] KISS (Keep it Simple, Stupid) - A Design Principle. [online], 2019, [cit. 2019-05-20]. Dostupné z: https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle

[45] Spring Expression Language (SpEL). [online], 2019, [cit. 2019-05-28]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions

[46] Aspect Oriented Programming with Spring. [online], 2019, [cit. 2019-05-28]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop

[47] Google APIs. [online], 2019, [cit. 2019-05-28]. Dostupné z: https://console.developers.google.com

[48] REST API Statelessness. [online], 2019, [cit. 2019-05-28]. Dostupné z: https://restfulapi.net/statelessness/

[49] Introduction to JSON Web Tokens. [online], 2019, [cit. 2019-05-28]. Dostupné z: https://jwt.io/introduction/

[50] Computer Network — HMAC Algorithm. [online], 2019, [cit. 2019-05-28]. Dostupné z: https://www.geeksforgeeks.org/computer-network-hmac-algorithm/

Dodatok A

Zoznam použitých skratiek

URL Uniform Resource Locator

RESTful Representational State Transfer

JDBC Java Database Connectivity

MVC Model–View–Controller

API Application Programming Interface

EMS Enterprise Messaging System

SPI Service Provider Interface

SP Service Provider

JPA Java Persistence API

ORM Object-relational mapping

POJO Plain Old Java Object

CRUD Create Read Update Delete

JPQL Java Persistence Query Language

SQL Structured Query Language

JQL JIRA Query Language

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

URI Uniform Resource Identifier

JSON JavaScript Object Notation

JWT JSON Web Token

DOM Document Object Model

JAR Java Archive

Dodatok B

Obsah priloženého USB

```
src
├── time-tracking ........................ zdrojové kódy implementácie
└── thesis ....................... zdrojová forma práce vo formáte LaTeX
text ........................................................ text práce
└── thesis.pdf ............................. text práce vo formáte PDF
```