UNIVERZITA KOMENSKÉHO V BRATISLAVE FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SimpleDesk

2019/20 EDUARD KRIVÁNEK

UNIVERZITA KOMENSKÉHO V BRATISLAVE FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SimpleDesk

Študijný program: Aplikovaná informatika
Študijný odbor: 2511, Aplikovaná informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Vedúci práce: RNDr. Peter Borovanský, PhD.

Bratislava 2019/20

EDUARD KRIVÁNEK

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Eduard Kriváne k
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: SimpleDesk
SimpleDesk

Anotácia: Webovská aplikácia informačného tiketovacieho systému je postavená na moderných technológiách, front-end Angular, back-end Spring Boot. Keďže ide o produkčnú verziu aplikácie, dôraz musí byť kladený na bezpečnú autentifikáciu užívateľov systému. Informačný systém má ambíciu byť nasadený do testovacej prevádzky, preto aj výsledky z nasadenia systému sa v závere práce očakávajú.

Cieľ: Clieľom práce je vytvorenie webovskej aplikácie firemného tiketovacieho systému (help desk). Základnou funkciou systému je správa požiadaviek (tiketov), ktoré majú stav, prioritu, autora, cieľových adresátov a riešiteľov. Životnosť a fázy riešenia požiadaviek monitoruje notifikačný systém, ktorý informuje cieľovú skupinu adresátov. Prehľad žiadostí budú mať len autorizovaní užívatelia s možnosťou zadávania komentárov pre vyskytnuté nejasnosti. Súčasťou systému je aj služba plánovacieho kalendára, personalizovaná každému užívateľovi. Systém umožní aj správu užívateľov a skupín. Štatistické prehľady o stave zadaných/vyriešených požiadavkách poskytnú stručný náhľad aktivity každému používateľovi.

Vedúci: RNDr. Peter Borovanský, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 17.09.2019

Dátum schválenia: 09.10.2019 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

---------- ----------
študent vedúci práce

ČESTNÉ VYHLÁSENIE

Vyhlasujem, že som bakalársku prácu vypracoval samostatne a uviedol som všetku použitú literatúru.

……

vlastnoručný podpis študenta

POĎAKOVANIE

Ďakujem môjmu školiteľovi za ochotu, cenné rady a usmernenie pri písaní a vývoji záverečnej práce.

Abstrakt

Cieľom práce je vytvoriť aplikáciu ako interné úložisko požiadaviek tvorené zamestnancami firmy cielené na jednotlivé oddelenia. Podľa vymedzených práv prihlásení užívatelia vypĺňajú formuláre s potrebnými informáciami na dané oddelenia a autorizovaní riešitelia na týchto problémoch pracujú.
Intuitívnosť, príjemný vzhľad a hlavne bezpečnosť sú podstatné črty pri vývoji projektu. Aplikácia je taktiež nasadená do produkčného prostredia a vzdialená správa nie je potrebná.

Kľúčové slová:
Architektúra, Redux, požiadavky, právomoc, autorizácia

Abstract

The aim of the bachelor thesis is to design and implement an internal repository for requests, created by emloyees to target a single department.
Authenticated users may fill forms with neccessary informations, under their defined rights and authorized people solve those problems. Intuition, pleasant look and especially security are the main factors in development. Application is deployed in production and no remote handling is necessary

Key words:
Architecture, Redux, requests, privilege, authorization

Obsah

1. Úvod

Pracovníci zamestnaní v rastúcej firme časom zaznamenávajú ťažšiu komunikáciu medzi vlastnými oddeleniami. Manažéri spadajú do situácie kedy strácajú prehľad na akej úlohe pracuje ich tím, v akom stave je, kto na nej pracuje a kto ju vlastne zadal.

Ak neexistuje interný systém na evidovanie požiadaviek, pre riešiteľa sa stáva až nemožné si všetko pamätať. Buď má zasypanú mailovú schránku požiadavkami od klientov, alebo interných zamestnancov, resp. on sám si musí uchovávať tieto zadania.
Neefektívna komunikácia môže viesť k zabudnutiu vyriešenia úloh, informácie požiadaviek môžu byť nekompletné, stráca sa prehľad stavu žiadosti a ťažko sa nám robia štatistiky koľko požiadaviek sa za jeden mesiac vytvorilo a koľko z nich sa vyriešilo.

Cieľom práce je navrhnúť a implementovať informačný systém na evidovanie rôznych druhov požiadaviek a sledovanie ich stavu tak, aby prístup k tejto aplikácie bol jednoduchý (webové rozhranie namiesto inštalácie desktopovej aplikácie), ale dostatočne zabezpečené na autentifikáciu a autorizáciu užívateľov.

Projekt vznikol spoluprácou so spoločnosťou COFIDIS SA, pobočka zahraničnej banky, kde bude nasadená, takže špecifikácia úlohy je prispôsobená pre klienta.

1.1. Požiadavky klienta

Hlavné funkcie systému: evidencia požiadaviek, notifikačný modul, kalendár udalostí, správa aplikácie, oprávnenie užívateľov.

Evidencia požiadaviek umožňuje vytvárať tikety, reporty a financie s prispôsobeným formulárom a povinnými políčkami, ktoré sa posielajú, každé na iné oddelenie. Autorizovaný užívatelia, iným slovom riešitelia môžu na nich pracovať.

Notifikačný modul bude informovať o zmenách. Ak užívateľ „A“ zmení prioritu, pridá komentár, pridelí riešiteľa a pod. , užívateľ „B“, ktorý vidí danú požiadavku, teda buď ju vytvoril, je naň pridelený, alebo mu len jednoducho systém pridelil práva, bude informovaný o počte zmien a o type zmeny formou logu na danej požiadavke, od poslednej kontroly.

Vlastné komentáre k požiadavkám sa dajú zmeniť so stavu verejné na stav privátny a zdieľať so skupinami, v ktorých je užívateľ členom. Žiaden externý človek skupiny nebude môcť vidieť komentár.

Dôraz sa musí klásť na bezpečnosť systému. V systéme musí byť autorizovaný člen, ktorý dokáže modifikovať práva užívateľov, aké typy požiadaviek môžu posielať, resp. aké môžu riešiť. Chceme sa vyhnúť situácie, kde si bežný zamestnanec vyžiada ročné vyúčtovanie firmy, alebo riešitelia oddelenia IT mali možnosť nazerať požiadavkám na financie.
Manažér musí mať možnosť monitorovať aktivitu vlastnej skupiny, resp. aby si na túto úlohu mohol on sám niekoho zvoliť. Taktiež chceme pridať do systému ďalších dvoch typov užívateľov. Admina, ktorý môže manipulovať s celou aplikáciou a „ducha“, typ užívateľa, ktorý vidí všetko, ale nemôže robiť nič.

1.2. Existujúce riešenia

1.2.1 Jira

Jira je aplikačné riešenie navrhnuté na pomoc tímom v organizovaní práce. Široký obsah funkcionalít ju posunulo na úroveň najpoužívanejších aplikácií v oblasti projektového manažmentu. Využívanie sa hlavne zaznamenáva v oblasti softvérového vývoja, kde tímy programátorov pracujú na rozdielnych projektov. Každý projekt môže byť na cenený, mať svojho vlastníka a čiastkové úlohy, nevyhnuté na dokončenie zadania. Na analyzovanie profitu zákaziek a náročnosti úloh sa strávené hodiny riešenia projektu priamo evidujú k čiastkovým úlohám. Reportovacím modulom môžeme pozorovať celkový výkon tímu, jednotlivca, či dĺžku etapy projektu. Na urýchlenú komunikáciu vieme notifikovať dotyčného použitím anotácii. Hosting aplikácie vyberáme z možností lokálneho udržiavania, alebo cloudového riešenia.

Spočiatku jira bola vytvorená na evidencia chýb a problémov, ale dnes ju používame na širokú škálu potrieb. Prináša užitočné funkcie ako vytváranie projektov s prispôsobenými povinnými poľami formuláru, zaznamenávanie chýb, tvorenie testovacích scenárov, notifikácie, reporty užívateľa, monitorovanie stavu projektov, vytváranie čiastkových úloh až po agilné programovanie metódou scrum. [1]

Scrum je spôsob vývoja aplikácií, kde ľudia sa rozdeľujú do troch rolí. Vlastník produktu zodpovedajúci za definovanie funkcií v projekte, scrum mastera, zodpovedajúceho za napredovanie tímu k spoločnému cieľu a samotného tímu, ktorí sa skladá z vývojárov, testerov a analytikov.
Z listu požiadaviek, iným slovom „product backlog“, sa vyberú zadania s najvyššou prioritou, ktoré by mohli ísť do ďalšieho vývoja produktu.
Následne nastáva plánovanie šprintu, kde sa vybrané požiadavky z „product backlog“ presunú do kategórie „sprint backlog“, ktoré sa za dobu 1 až 4 týždne majú dokončiť. [1]

1.2.2 Bugzilla

Bugzilla, webový zaznamenávací a testovací systém vyvinutý v roku 1998 licencovaný pod Mozila Public License, ktorý uchováva tikety užívateľov na jednom mieste. Ponúka možnosť vytvárania vlastných formulárov pre rôzne typy problémov, ich triedenie na správne oddelenie, prehadzovanie medzi oddeleniami a sledovanie histórie zmien.
Ďalej umožňuje uzamknutie úlohy len pre jedného riešiteľa, aby nedošlo k dvojitej odpovedi, podporuje generovanie reportov pre pohľad aktivity a výkonu helpdesku. Sprevádza inštaláciou, integráciu a migráciou. [2]

V porovnaní s Jirou, najväčšia výhoda Bugzilla je bezplatná licencia na používanie a menšia náročnosť záťaži servera, ale disponuje s nedostatkom funkcionalít ako priorizovať problémy ťahaním myšou, prispôsobovanie domovskej stránky a informovanie v reálnom čase.
Bugzilla je taktiež open source, jeho repozitár nájdeme na github, kde mnoho nezávislých programátorov môže napomáhať ku rozšíreniu projektu, písaním čistého kódu, ktorý, ale musí prejsť medzi 1 až 10 revíziami. Schválenie kódu je odmietnuté, ak je v konflikte s inými čakajúcimi zmenami, alebo nenasleduje štandardizovanú architektúru systému. Wiki časť Bugzilly tvrdí, že len 0,1% zmien kontribútorov prejde ku schvaľovaciemu procesu. [2]

2. Architektúra informačných systémov

Snaha definovať termín softvérová architektúra je nebezpečná aktivita. Teória sa často líši od praxe a za počet rokov vznikla široká škála akceptovateľných definícií v priemysle.
Najviac ho však vystihuje definícia od IEEE :
"Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." [3]

Správnou architektúrou sa snažíme rozumne rozdeliť aplikáciu na sadu vnútorno-prepojených komponentov, modulov, objektov, resp. podľa vlastných potrieb. Architektúra musí byť navrhnutá tak, aby vyhovovala špecifickým požiadavkám a obmedzeniam aplikácie.

Kľúčový bod takmer každej aplikácie je minimalizovať závislosť medzi komponentami, vytvorenie tzv. vysokej súdržnosti a nízkej spojitosti. Závislosť komponentov môže viesť k problému, kde zmena v jednom si vyžaduje zmenu v celom projekte. Extenzívne prepojenia tvoria zmeny v systéme komplikovaným, zväčšuje sa rozsah testovania aplikácie a sťažujú tímovú spoluprácu. Elimináciou nepotrebnej závislosti sú zmeny komponentov lokálne a nepropagujú sa na ďalšie.

Postupom času sa vyvinulo mnoho úspešných návrhových vzorov programovania, ktoré opisujú interakciu medzi kolekciu komponentov. Rozsiahle systémy sa často skladajú zo súboru niekoľkých návrhových vzorov, kombinovaných do celku, ktoré uspokoja architektonické požiadavky. Ak architektúra nasleduje presne stanovené princípy, stáva sa aj pre nových členov projektu zrozumiteľná. Pre abstraktnú reprezentáciu systému často používame diagramy ako use-case, uml, komponent diagram, ktoré slúžia ako podklad pre analyzovanie a skúmanie aplikácie.

Napriek tomu, že máme jasnú predstavu správania a poznámke rozdielne architektúry, ešte neznamená, že návrh je mechanický a triviálny. Návrh komplexného systému si vyžaduje dostatočnú kreativitu, znalosť, skúsenosti a disciplínu. Nesprávne zvolená architektúra môže viesť k prerobeniu pomerne veľkej časti aplikácie. [4]

2.1. Klient – server architektúra

Klient – server model je distribuovaná aplikačná štruktúra, ktorá presne definuje hranicu medzi poskytovateľom služby, alebo zdrojom nazývaným server, a žiadateľom služby nazývaným klientom. Centrálny server hostí, dodáva a spravuje väčšinu zdrojov a služby na požiadavku klienta doručenú cez sieť. Klientsky počítač sprostredkováva interfejs pre užívateľa o umožnenie požiadania služby na servery a následné zobrazenie údajov. Server čaká na žiadosť klienta, spracuje ho a odpovedá. Klient sa často nachádza na pracovnej stanici užívateľa a server na sieti pripojenom výkonnom počítači. Komunikácia prebieha cez protokol http a rozdeľujeme ho na tri vrstvy : rozhranie užívateľa, business logiku a perzistenciu dát.
Návrh modelu si vyžaduje skúsenosti v oblastiach databázového dizajnu, spracovanie transakcii, komunikačných protokolov a UX dizajnu.
Servery dnes už poskytujú služby nielen jednému, ale viacerým klientov, preto pri návrhu musíme uvažovať medzi tenkým a hrubým klientom. [5]

V modeli tenký klient – hrubý server, na strane klienta beží softvér menej náročný na požiadavky počítača. Väčšinu práce vykonáva server, preto zväčšujúcim sa množstvom užívateľom responzivita klesá a narastá počet relácií na strane servera (angl. server side sessions), ktoré môžu obsahovať veľké objektové štruktúry. Relácia medzi klientom a serverom sa udržiava fixný časový úsek, alebo je perzistentná až do odhlásenia užívateľa. Týmto sa zdroje servera rýchlo vyčerpajú a klesá výkonnosť.

Alternatívou tohto modelu je hrubý klient a tenký server. Narastajúcim sa výkonom počítačov vývojári mohli presunúť náročnejšie výpočty na stranu klienta a server ponechať len ako sprostredkovávateľa dát, ktorý vykonáva bežné CRUD (create, read, update, delete) operácie. Udržiavanie stavu aplikácie sa tiež presunulo na stranu klienta, kde väčšina údajov sa ukladá do úložiska prehliadača (coockies, localstorage, sessionstorage) a následne sa nevyhnutné informácie posielajú na server.

Najdôležitejšou úlohou implementovanie tejto architektúry je rozhodnúť aká časť kódu má bežať na strane klienta a aká na strane servera. Kvalitná implementáciu modelu klient – server má mať jednoduché komunikačné rozhranie. Strana klienta musí pokrývať najväčšie množstvo prezentačnej vrstvy aplikácie a interakciu s užívateľom. Naopak biznis logika, modifikácie dát a bezpečnosť by mala byť oddelená na stranu servera.

Návrh modulárneho, znovu použiteľného rozhrania servera sa nazýva architektúra orientovaná na služby (service oriented architecture - SOA). Komunikácia medzi serverom a klientom by mala byť založená len na posielaní dát. [5]

2.2. Architektúra servera

Pri zvolení správnej architektúry sa softvér ľahko škáluje a budúce integrácie nie sú zložité. Naopak, pri zlej architektúre sa nám môže stať, že veľkosť programu prerastie do výšky, kde už samotný vývojár stráca prehľad o funkcionalite aplikácie.
Softvérová architektúra je o štrukturálnych rozhodnutiach projektu, ktoré sú náročné na zmenu ak sú raz implementované. Taktiež pri návrhu architektúry je možné viaceré kombinovať dokopy.

2.2.1 Monolitická architektúra

Monolitická architektúra je považovaná za tradičný spôsob vývoja, kde výsledkom celého produktu je jedna veľká aplikácia, ktorá časom len rastie.
Ak monolitický prístup implementujeme na strane servera hovoríme o tom, že jedna veľká jednotka reaguje na http požiadavky, vykonáva biznis logiku a interaguje s databázou.
Primárny benefit tohto prístupu je jednoduchosť infraštruktúry, testovania a rýchle nasadenie softvéru, pretože máme len jeden spustiteľný súbor. Je to jednoduchý prístup ako začať stavať novú aplikáciu.

Monolitické aplikácie sa často navrhujú ako N – vrstvové (angl. N- tier) aplikácie, kde sa snažíme rozčleniť kód do vrstiev. Týmto prístupom sa snažíme vytvoriť samostatné vrstvy pre kontrolery, služby, entity a databázové prístupy. Každá vrstva sa môže samostatne zabezpečiť, resp. škálovať. Vrstvová architektúra sa ujala aj v praxi, kde každý tím môže pracovať na vlastnej vrstve softvéru.

Nevýhodou monolitického prístupu je škálovateľnosť v priebehu času. Nastáva vysoká spojitosť a nízka súdržnosť (angl. tight coupling low cohesion) medzi vrstvami a strácame prehľad o závislosti vrstiev, preto sa časti kódu ťažko mieňa. Ďalej máme jednotný bod zlyhania aplikácie (angl. single point of failure). Ak nám pribudne ďalší človek do tímu, musí porozumieť častiam kódu, ktoré sa ho ani netýkajú [5]

2.2.2 Architektúra orientovaná na služby (SOA)

Architektúra orientovaná na služby (angl. Service oriented architecture – SOA) je štýl dizajnu aplikácie, ktorá je určená na rozuzlenie veľkej monolitickej aplikácie na menšie služby, ktoré sú orientované na naplnenie jedného účelu.
Výhodou SOA architektúry je ten, že aj keď ostáva z istej časti monolitická, každá služba je nezávislá od tej druhej. Vývojári nemusia rozumieť celej aplikácii, stačí ak vedia ako fungujú služby, na ktorých pracujú. [6].
Je to skôr abstraktné rozčlenenie kódu, kde súvislé súbory vytvárajú moduly, ako napr. modul user, než by sme ich rozbili na samostatné fungujúce jednotky, iným slovom mikro-servisy.

Podľa IBM knowledge center [6] má služba tieto charakteristiky:

- Spracováva biznis logiku ako výpočet mzdy, posielanie emailov, riadi technické úlohy, ako prístup k databáze, alebo poskytuje obchodné údaje.
- Povolená je komunikácia s inou službou
- Je nezávislá od žiadateľa požiadavky. Zmeny vo vnútornej logike si vyžadujú minimálne, alebo žiadne zmeny v žiadateľovi. Vzniká nízka spojitosť medzi jednotlivými službami.
- Na komunikáciu v http odpovediach používa REST v JSON formáte, alebo SOAP v XML formáte.

2.3. Architektúra Klienta

Pri navrhovaní webového klienta máme na výber z dvoch architektúr. Tradičný webový prístupu (angl. multi page application, skr. MPA) , ktorá vykonáva väčšinu logiky na strane servera, alebo jednostránková aplikácia (angl. single page applicaiton, skr. SPA), ktorá vykonáva logiku používateľského rozhrania na strane klienta a so serverom zväčša komunikuje cez REST API rozhranie.
Podľa microsoft docs [7] preferencie pre MPA sú ak:

- Webová aplikácia sa navrhuje iba na čítanie, napr. blog . Tento druh aplikácie si nemusí pamätať stav užívateľa v relácii so serverom a pri prístupe na článok sa nám logika renderovanie HTML-ka vykoná na serveri, ktorý klient len zobrazí.
- Webová aplikácia musí fungovať bez javascriptu.
- Vývojový tím je neoboznámený s javascriptom

Tradičný webový prístup má aj svoje nedostatky. Je pomalý kvôli renderovaniu celej HTML stránky na severi, čím sa zväčšuje šírka pásma pri prenose informácií cez sieť priamo úmerne s časom zobrazenie obsahu.

2.3.1 Jednostránková aplikácia

Jednostránková aplikácia (ďalej len SPA), je aplikácia, ktorá beží priamo v prehliadači. Za svoju životnosť sa načíta len raz a nepotrebuje sa obnovovať počas behu, čím prináša vysokú reaktivitu a lepší UX (angl. user experience). Prispôsobenie na mobilné zariadenie sa stáva jednoduchším, pretože server posiela už len údaje, nie celé HTML stránky. Predstavuje hlavne komponent architektúru, kde každý komponent je znovu použiteľný a má a presne definovanú funkcionalitu, pričom interakciou užívateľa meníme ich zobrazenie. Gmail, Facebook, Github a mnoho ďalších aplikácií používa architektúru SPA. [7]

Prehliadač inicializačnou požiadavkou na server obsiahne index.html súbor a to je všetko. Je to prvý a poslednýkrát, kedy sa html súbor prenáša cez sieť. V ňom sú prilinkované javascript súbory, ktoré prevezmú kontrolu nad rederovaním. Samotná stránka sa už nikdy neobnovuje, len komponenty sa zobrazujú a skrývajú. Tradičné webové aplikácie oproti SPA na každé volanie servera získajú vykreslenú stránku a spustia automatické obnovenie.

Je jednoznačné, že prvou výhodou SPA je vyhnutie sa posielaniu html súborov cez sieť, čím sa znižuje čas, šírka pásma a responzivita servera. Html súbory sú kvôli tagovaniu veľké a pri viacnásobnom navštívení rovnakej podstránky vzniká duplicitné zasielanie dát. Ďalšou výhodou SPA je oddelený klient od servera, nasledovaním dizajnu hrubý klient, tenký server. Komunikácia so serverom prebieha pomocou web servisov posielaním údajov. [8]

Argumenty voči SPA sú nasledovné. Veľké množstvo logiky na strane klienta môže finálny javascript súbor zväčšovať čím narastá čas sťahovania a sparsovanie zväzku. Väčšina rámcov na SPA sa s touto problematikou vysporiadavajú dostatočným optimalizovaním veľkosti výsledného javascriptu. Napriek automatizovanej rámcovej modifikácii, správny programátor musí vedieť ako najlepšie optimalizovať importy externých knižných a načítavanie komponentov na dopyt.
Nevýhodou SPA je taktiež slabý SEO (angl. search engine optimalization). Google má problém spracovať javascript obsah stránok, preto indexovanie SPA aplikácie môže byť nižší ako MPA.
V skratke to znamená, že google nesprávne vyhodnotí obsah nášho článku a nezaradí nás medzi popredné voľby vyhľadávania.
Taktiež SPA vyžadujú povoliť javascript v prehliadačoch, bez ktorého nedokáže fungovať.
Únik pamäte v javascripte môže spôsobovať spomalenie systému a sprístupnením väčšieho zdrojového kódu na strane klienta otvárame cestu tzv. XSS útokom (Cross-Site scripting). [8]

2.4. Web servisy

Web servis je komunikácia medzi dvoma elektronickými zariadeniami cez sieť. V praxi web servis zvyčajne sprostredkováva aplikačný programovateľný interfejs, ďalej len API, pozostávajúci z niekoľkých verejne vystavených koncových bodov pre definovanie komunikácie na základe požiadavky a odpovede skrz HTTP protokol.
Dôležitým aspektom na interakciu s API serverom sú koncové body, pretože špecifikujú, kde sa zdroje nachádzajú, ktoré môžu byť prístupné pre účastníkov tretích strán prevažne cez URI, pomocou HTTP požiadavky. Koncové body musia byť statické, inak správna komunikácia nemusí byť zaručená. Zmenou umiestnenia zdroja, alebo modifikovaním jeho koncového bodu môžeme znefunkčniť celú aplikáciu. API taktiež potrebuje používať štandardizovaný protokol na komunikáciu. Najčastejšie používané metódy sú SOAP a REST.

2.4.1 SOAP

SOAP (angl. Simple Object Access Protocol), je komunikačný protokol na výmenu dát v decentralizačnom a distribuovanom prostredí pracujúci s protokolmi na aplikačnej úrovni TCP/IP ako s HTTP, SMTP, TCP a UDP. Bezpečnosť, autorizácia a spracovanie chýb sú vnútorne zabudované funkcionality. Dáta sú vo formáte XML a nasledujú formálny a štandardizovaný prístup šifrovania. SOAP bol navrhnutý spoločnosťou Microsoft v roku 1998 ako náhrada zastaralých technológií, ktoré boli nedostatočné na príchod internetu a spoliehali sa na binárny prenos údajov, protokoly ako DCOM (Distributed Component Object Model) a CORBA (Common Object Request Broker Architecture). SOAP správa má dva najpodstatnejšie časti, obálku (angl. envelope), reprezentuje začiatočný a konečný tag v správe, a telo (angl. body), ktoré obsahuje xml údaje, generované serverom prijímateľovi.

Nevýhodou SOAP je nevyužívanie HTTP status kódov. Ťažko sa hovorí klientskej strane, kedy sa má užívateľ presmerovať, kedy nebol autorizovaný, či došlo ku chybe na strane klienta, alebo servera. Všetky správy smerované na server sa taktiež posielajú ako POST žiadosti, čo nie je zakázané v REST architektúre, len nepoužívané, pretože nevyužijeme všetku podporu prinášajúcu z architektúry REST.

Napriek nevýhodám, SOAP bol predchodca REST, pomerne zaužívaný protokol a v súčasnosti stále v obehu. Staršie systému dodnes podporujú túto formu prenášania dát. [9]

2.4.2 REST

REST je architektúra na rozdiel od SOAP, slúži na posielanie reprezentačného stavu objektu. Je to odľahčená komunikačná alternatíva na zníženie práce vývojárom. SOAP v javascripte si vyžaduje veľa kódu, musí byť vytvorená presná XML štruktúra nasledujúca predpisy protokolu aj na vykonanie jednoduchej úlohy. Aktuálne REST je najviac používaný spôsob vytvárania webových API, kde server reaguje na CRUD požiadavky klienta pomocou HTTP protokolu. Použitím REST rozhrania získame rýchlejší prenos dát medzi klientom a serverom, pretože prenášame iba reprezentáciu objektov vo viac zaužívanom json formáte.
Json (angl. JavaScript Object Notation) je ľahko parsovateľný a čitateľný formát dát. Jeho syntax je podmnožinou štandardu ECMA-262 3rd Edition. Kolekcia dvojíc kľuč-hodnota, ktorá je univerzálna dátová štruktúra pre väčšinu programovacích jazykov napomohla k nezávislosti výberu programovacieho jazyka.

REST okrem json podporuje aj ďalšie formáty údajov ako XML, HTML, Blob, txt atď. Výhodou posielania objektov je ich jednoduché ukladanie na strane klienta. Pri načítaní stránky sa vykoná jedna požiadavka na server, obdržia sa údaje a uchovávajú sa v pamäti prehliadača, čím sa šetrí čas zobrazovania a nezaťažuje sa sieť.
Taktiež z veľkej časti využívame HTTP status kódy. Pri kóde 40X hovoríme, že na strane užívateľa vznikla chyba, kódom 50X, že na strane servera a kódom 20X oznamujeme o úspešnom priebehu udalostí. Implementovaním tejto architektúry vytvárame tzv. bez stavový server.

Spomenutím nevýhod môže byť chýbajúci štandard. Ako sme vyššie spomenuli, REST je architektúra, nedefinuje žiadnu striktnú schému či formát písania URI požiadaviek, čo môže často viesť rôznym spôsobom vývoja v závislosti od skúseností programátora. [9]

2.4.3 Stavové alebo Bez stavové API

Stavové (angl. statefull) API ukladajú stav aplikácie na serveri. Po prvom úspešnom autentifikovaní užívateľa na serveri vznikne prepojenie (angl. session) medzi klientom a severom. Server vygeneruje unikátny kľúč, ktorý pošle na stranu klienta. Tento kľúč následne klient posiela každou požiadavkou na server. Odpovede servera sú rýchle, pretože nevzniká opakované databázové vyhľadávania žiadateľa. Server si informácie stráži v pamäti a pomocou unikátneho kľúča obdržaného zo strany klienta vie identifikovať žiadateľa. Dizajn stavových API sa hodí pri malých projektov. Musíme si ale položiť otázku ako dlho má byť naviazané spojenie s klientom a odkiaľ viem, či ho už klient neuzavrel ? Zároveň pri väčších aplikáciách môžeme naraziť na niekoľko problémov.

Povedzme, že máme dva inštancie servera a potrebujeme pridať load balancer na vyrovnanie záťaže. Očakávali by sme smerovanie prvej požiadavka na server „A“ a druhej na server „B“. To sa ale v tomto scenári nestane. Keďže server „A“ drží stav prihláseného užívateľa ako objekt, každá jeho žiadosť bude smerovaná na server „A“. Ak by jeho požiadavka bola presmerovaná na server „B“, muselo by aj tam vzniknú nové prepojenie (session) medzi klientom a severom. Takže by sme mali dva servery, ktoré by si držali stav rovnakého prihláseného užívateľa, čím míňajú svojimi zdrojmi, ako napríklad pamäťou. Nemáme možnosť horizontálne škálovať našu aplikáciu, len vertikálne pridám viacej CPU, pamäte a pod.

Riešením tohto problému sú bez-stavové (angl. stateless) API. Ako názov značí, server si neuchováva žiadne informácia v pamäti, čím šetrí so svojimi zdrojmi a umožní nám horizontálne škálovanie aplikáciu load balancerom. Otázka je, ak server nedrží žiaden stav, ako bude vedieť identifikovať prihláseného užívateľa ?

Autentifikáciou užívateľa do systému sa vygeneruje token, nazývaným ako Json Web token (JWT). Je to štandard RFC 7519, ktorý definuje bezpečný prenos informácií medzi dvoma stranami ako zašifrovaný json objekt. Privátnym kľúčom definovaným na strane servera a algoritmom HMAC, či RSA sú informácie užívateľa a taktiež jeho práva (angl. claims) enkryptované do hash-u, ktorý je zaslaný na stranu klienta. Klient každou požiadavkou na server pripojí do HTTP hlavičky tento token s prefixom "bearer ", čo je len konvencia pomenovania.

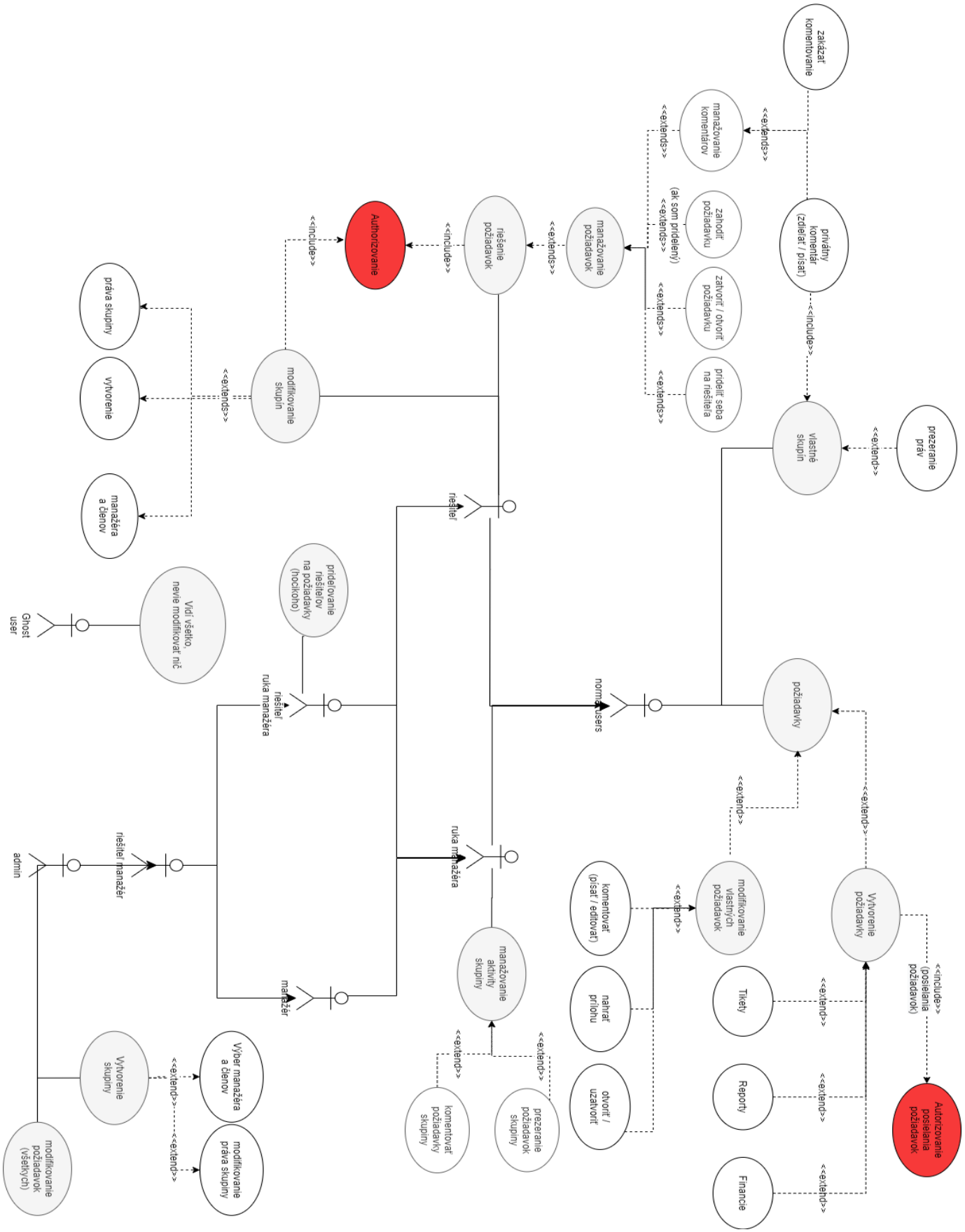
Z bezpečnostného hľadiska by sme mohli zvažovať, že ak sa token pošle na stranu klienta, tak to značí, že ho užívateľ môže prepísať a zmeniť si prístupové práva do systému. To je pravda, ale ľubovoľnou malou zmenou tokenu sa zmení aj jeho hash a opätovným zaslaním tokenu na server sa nebude dať rozkódovať privátnym kľúčom, čiže požiadavka klienta bude zamietnutá. Napriek tomu, že server každou HTTP požiadavkou musí vykonať vyhľadávanie žiadateľa v databáze, aby si mohol načítať aktuálne autorizovaného užívateľa, tento prístup je jednoduchý pri horizontálnom škálovaní a kritický pri navrhovaní moderných aplikácií. [9]

3. Návrh

Vo fáze plánovania sa dôraz musí klásť na návrh systému. Softvéroví inžinieri postupne vyvinuli univerzálny jazyk, a to diagramy, pre rýchle pochopenie systému s minimálnymi komentármi. V nasledujúcej kapitole návrhu si predstavíme use case diagram aplikácie, ktorý prezentuje typy užívateľov a ich možnosť interakcie s aplikáciou, komponent diagram, pre zjednodušený uhoľ pohľadu funkcionality, štruktúru databázy a v neposlednom rade návrh používateľského rozhrania.

3.1. Use case diagram

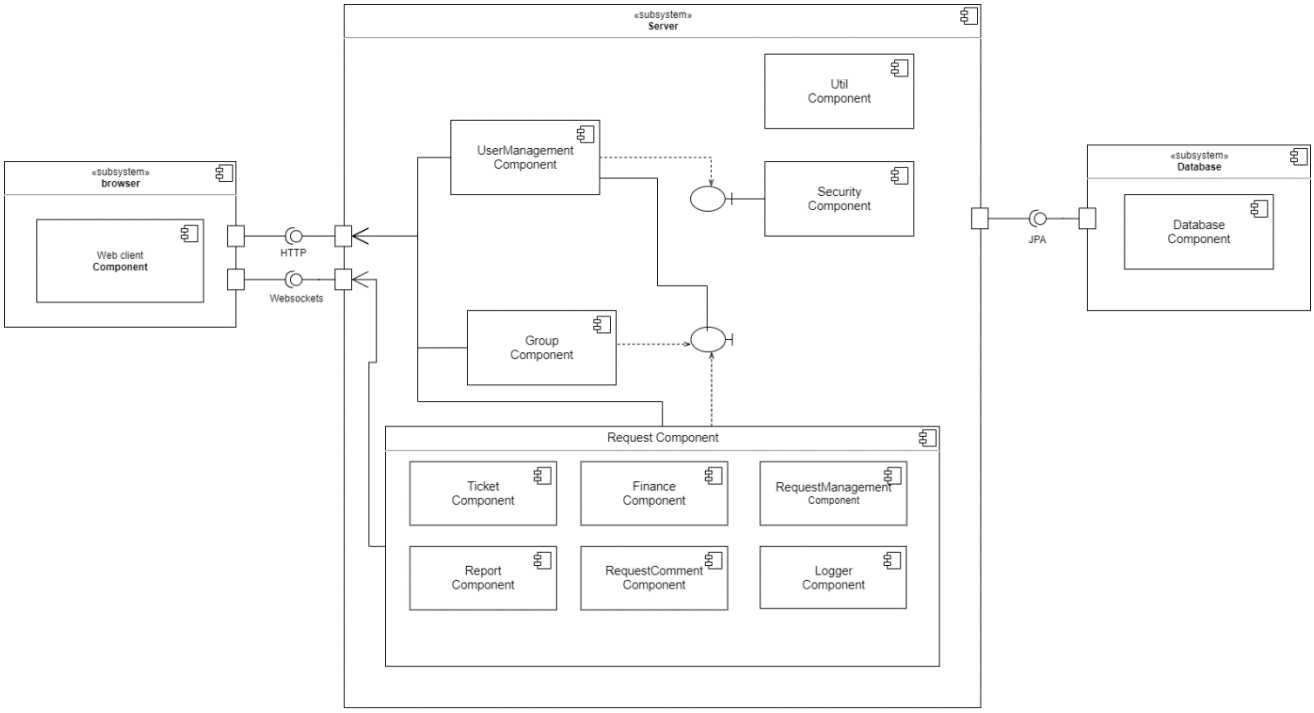
Kvôli bezpečnosti budeme mať viacero typov užívateľov. Novo vytvorený užívateľ nebude môcť nijak zaujímavo interagovať s aplikáciou. Aby mohol posielať, alebo riešiť požiadavky musí byť priradený do skupiny. Práva využívania jednotlivých modulov sa priradzujú skupine, užívateľ ich následne dedí priradením do nich. Byť členom, manažérom, alebo sledovateľnom viacerých skupín je taktiež povolené. Role typu : riešiteľ, manažér sa automaticky priradzujú v závislosti postavenia užívateľa v skupine.



Obrázok 1 Use case diagram aplikácie

3.2. Komponent diagram

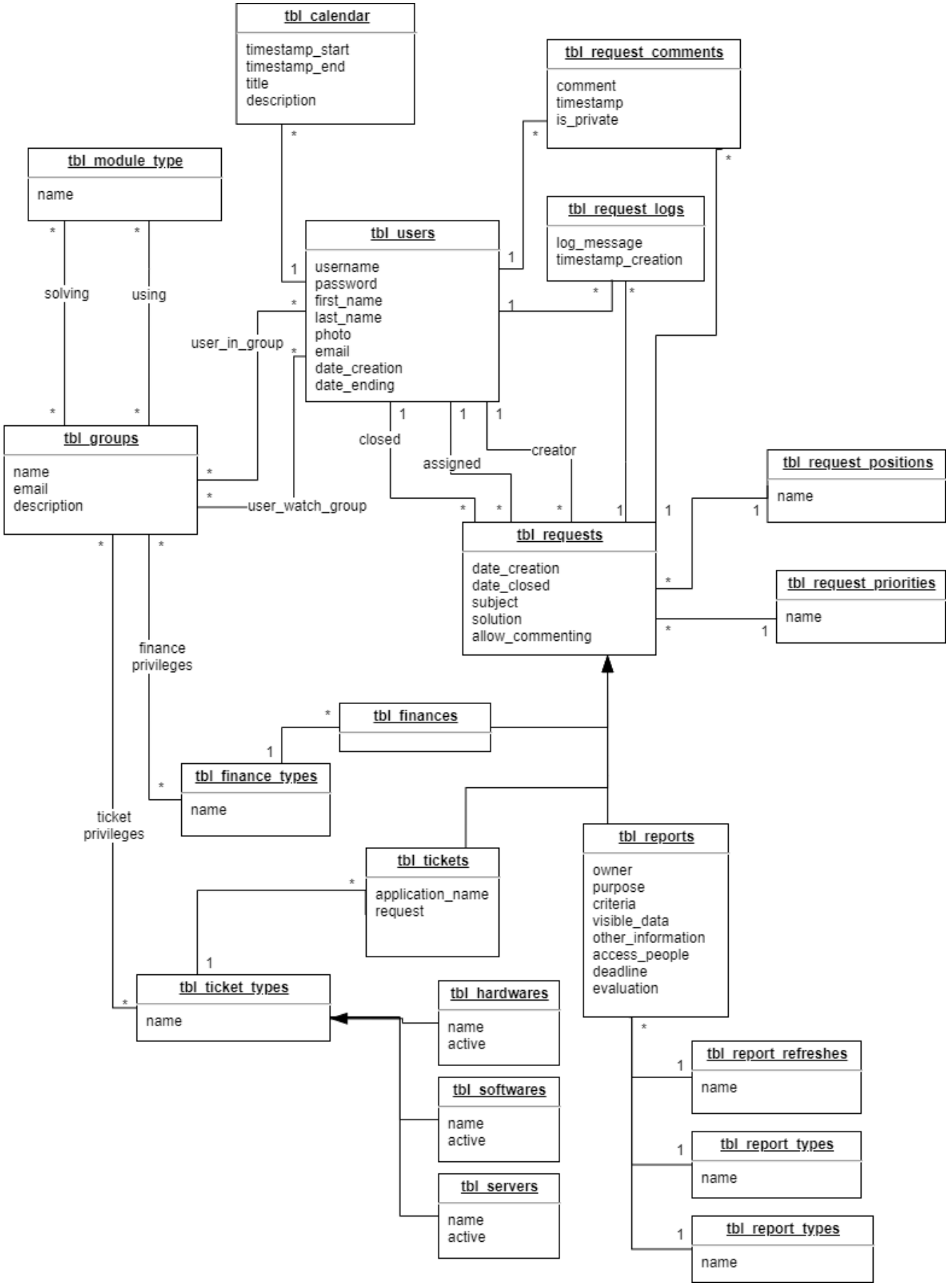
Komponent diagram rozčlení systém na rozdielne funkcionality z jednoduchého uhľa pohľadu. Každý komponent je zodpovedný za jeden jasný cieľ v aplikácii, je možné ho nahradiť iným a interaguje s ostatnými v prípade nevyhnutnosti.
Na obrázku č. 2 vidíme, že na strane servera je najviac používaný komponent *UserManangement*. Je to z dôvodu, lebo v bez stavovom API neustále vyťahujeme odosielateľa z databázy, ktorý vytvoril požiadavku na server. Tento komponent zároveň intenzívne komunikuje so *Security* komponentom, kvôli prístupovým právam ku zdroju servera.



Obrázok 2 Komponent diagram

3.3. Štruktúra databázy

Entitno relačným modelom abstraktne opíšeme používateľskú aplikáciu na konceptuálnej úrovni, kde modelujeme entity a ich relácie medzi nimi.
Hlavný účel systému je uchovávať požiadavky v tabuľke *tbl_requests*. Prefix *tbl_* je len zaužívaná konvencia pomenovania. Zmysel tejto tabuľky je zoskupiť spoločné atribúty pre podmnožiny požiadaviek ako tikety, reporty, financie.
Každý registrovaný člen sa nachádza v *tbl_users*. Ako sme spomenuli, človek bez pridania do skupiny nemá žiadnu právomoc vykonávania činností v systéme. Preto skupiny registrované v *tbl_groups* tvoria M:N reláciu s *tbl_module_type*, *tbl_finance_types* a *tbl_ticket_types*, aby sa im vymedzili prístupy.



Obrázok 3 Entitno-relačný model databázy

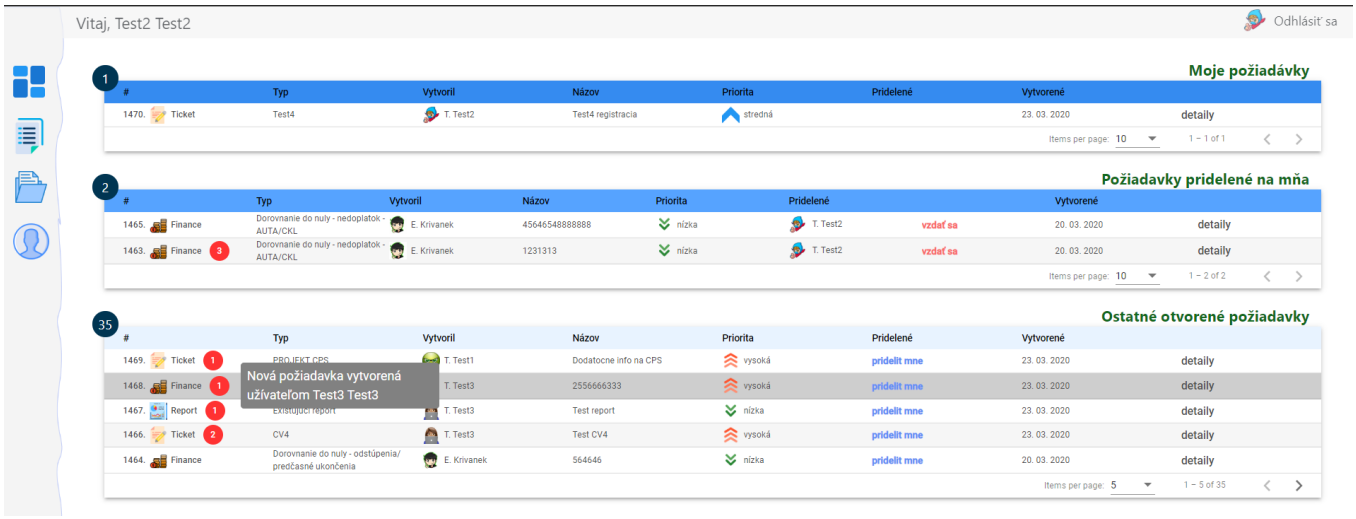
3.4. Návrh používateľského rozhrania

Návrh užívateľského rozhrania, ďalej už len UX, si vyžaduje cit dizajnéra ku farbám, štýlu písma, tvaru a umiestnenia komponentov. Krásny obaľ môže zakryť nedostatky, resp. nedotiahnutý dizajn odradiť návštevníka.

Kniha The Non-Designer's Design Book [10] opisuje jednoduché pravidlá návrhu UX:

- Kontrast, najdôležitejší pôvab stránky, ktorý okamžite zachytáva pozornosť diváka. Ak využívané prvky nie sú rovnaké (farby, tvar, veľkosť) , potom nech sa úplne odlišujú.
- Opakované využitie tvaru prvkov na rozdielnych miestach pre ľahkú a intuitívnu navigáciu
- Vyhnutie sa svojvoľnému umiestneniu prvok. Každý partia stránky musí na seba vizuálne nadväzovať
- Zoskupením malých útvarov do jednotného celku dáva čitateľovi čistejšiu vizuálnu štruktúru

Snahou dodržať spomenuté zásady sa podstatné informácie užívateľovi vypíšu na hlavnej stránke, zvanou dashboard, ktorá bude tvorená stručnými detailmi požiadaviek rozdelených do troch kategorizovaných tabuliek zobrazených podľa typu prehláseného užívateľa.
Bežným užívateľom sa zobrazí len tabuľka „*Moje požiadavky*“ s informáciami požiadaviek, ktoré oni sami vytvorili.
Riešiteľovi chceme pridať ďalšiu oddelenú tabuľku „*Požiadavky priradené na mňa*“ so separovanými požiadavky, ktoré sú pridelené len na neho.
V neposlednom rade všetky typy požiadaviek zaslané do systému, ktoré korešpondujú s právami užívateľa na riešenie sa zobrazia v tabuľke „*ostatné otvorené požiadavky*“. V tejto tabuľke sa aj manažérom zobrazia otvorené požiadavky členov ich skupín.



Obrázok 4 Domovská obrazovka

Kliknutím na detaily sa zobrazia podrobné informácie žiadosti. Užívateľ bude mať možnosť nahrať dokumenty so zaznamenaním času a taktiež písať verejné, či súkromné komentáre, ktoré môže zdieľať s členmi svojej skupiny. V detailoch sa taktiež zobrazí panel na modifikovanie stavu žiadosti. Riešiteľ bude mať možnosť predeliť žiadosť na iného registrovaného člena systému, bez ohľadne na jeho oprávnenia, zmeniť prioritu, alebo ju uzatvoriť, ktorá je taktiež prístupná pre tvorcu.
Prístupové práva a detaily prihlásených jednotlivcov sa budú nachádzať na obrazovke profil. Zobrazuje informácie skupín užívateľovi, ktorých je manažér, ktoré sleduje, resp. je len členom.

Kliknutím na skupinu, v sekcii profil, sa načítajú jej detaily, čo zahŕňa dodatočných členov a práva danej skupiny. Nakoniec v jednom odseku budú vlastné celkové práva užívateľa zdedené zo skupín v ktorých vystupuje ako člen.
Správa aplikácie, bude sekcia stránky vyžadujúca si autentifikáciu, ktorá sprístupní registrovanie, či zobrazenie detailov každého vytvoreného užívateľa, alebo skupiny, s možnosťou modifikácie oprávnenia, pridávanie, odoberanie členov a menenie manažérov skupinám.

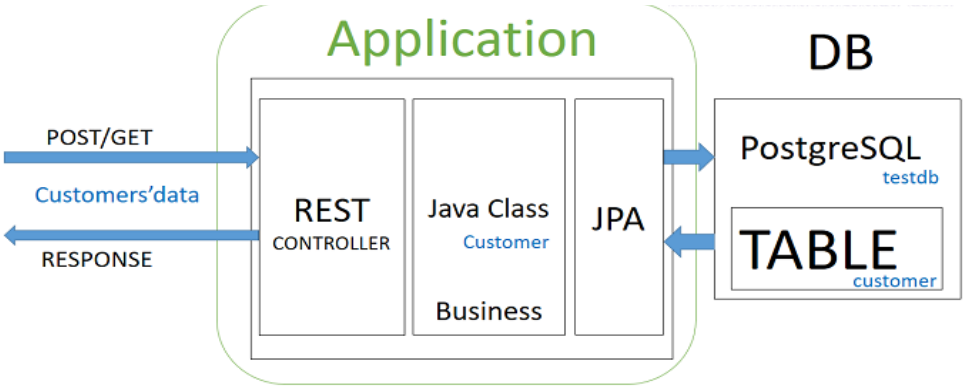
4. Implementácia

4.1. Implementácia servera

Aplikácia bude nasledovať architektúru hrubý klient – tenký server, kvôli lepšiemu škálovaniu softvéru a potenciálneho nárastu užívateľov. Výberom tohto prístupu sa nakláňame k REST rozhraniu, využitím json formátu údajov, kvôli zníženiu záťaže siete. Taktiež rátame s možnosťou, že klient môže byť napojený na viacero REST rozhraní, odkiaľ získava údaje na zobrazenie.

4.1.1 Spring boot

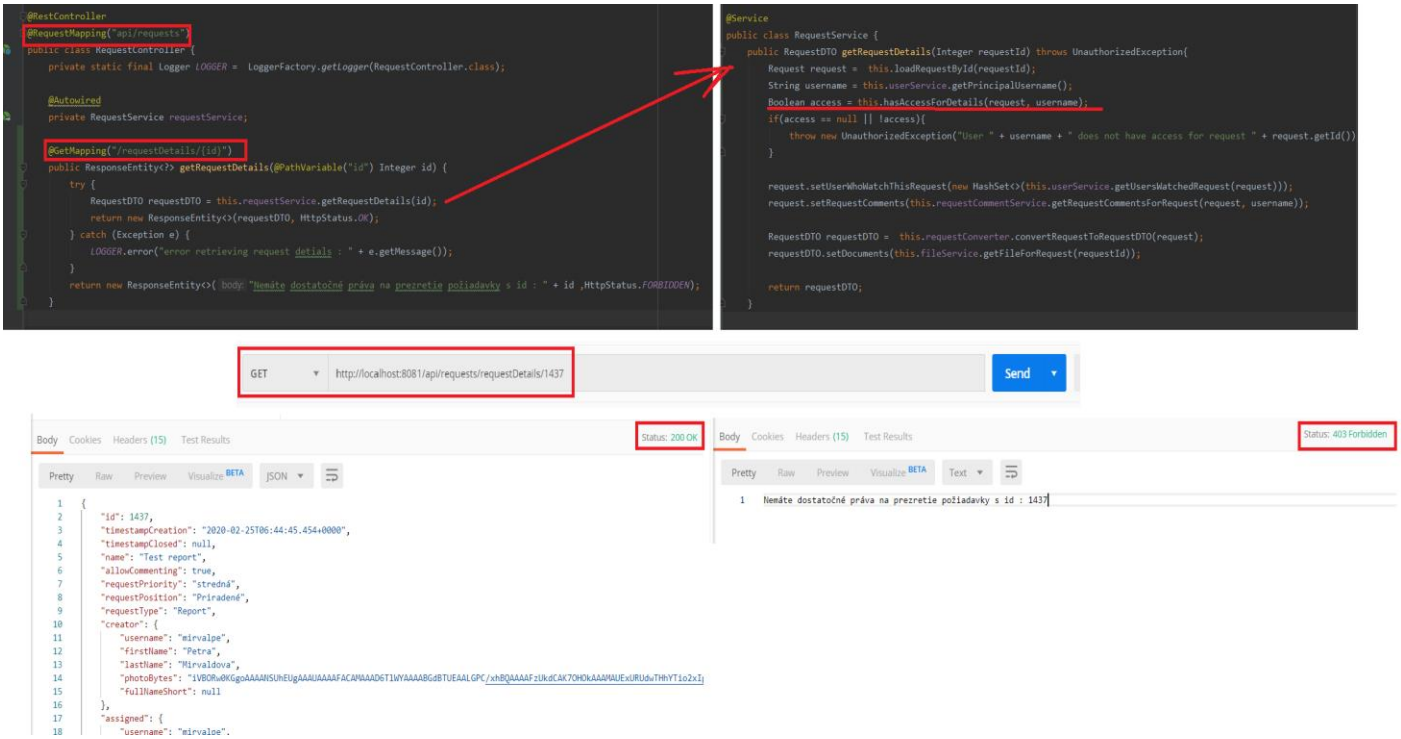
K dosiahnutiu nášho cieľa nám pomôže jazyk java a rámec (angl. framework) spring boot.
Architektúra rámca spring boot nasleduje monolitickú architektúru rozdelenú do vrstiev, odsek 2.2.1. Vrstvovú architektúru projektu spring boot skombinujeme s architektúrou orientovanú na služby, odsek 2.2.2. Kvôli dosiahnutiu nízkej spojitosti sa snažíme aplikáciu rozobrať na oddelené služby, tvoriacu jednu biznis logiku (pozri komponent diagram 3.2).
Najvrchnejšiu úroveň tvoria kontroléry, označením *@RestController* , ktoré prijímajú požiadavky klienta cez URI. Hlavný kontrolér nazývame DispatcherServlet, na ktorý narazia všetky požiadavky klienta. DispatcherServlet pri inicializácii načíta svoj aplikačný kontext a deleguje prichádzajúce požiadavky klienta na správne koncové body.
Strednú vrstvu tvorí biznis logika aplikácie označením *@Service*. Slúži na zložitejšie modifikovanie údajov, resp. vykonávanie transakcií. Vyznačujú sa vlastnosťou singletonu a pomocou injekcie závislosti (angl. dependency injection) cez označenie *@Autowired* sa posielajú medzi komponentami označenými *@service*, alebo *@controller*. Nakoniec spodnú úroveň tvoria repozitáre, označením *@Repository* na načítanie entít z databázy. [11]



Obrázok 5 Spring boot architektúra

Integráciu viacerých rámcov do jedného systému riadi projektový manažér Maven. Využíva koncept POM (angl. project object model), kde pridaním projektových závislostí, automaticky stiahne a pridá dodatočné .jar súbory do cesty projektu. [12]

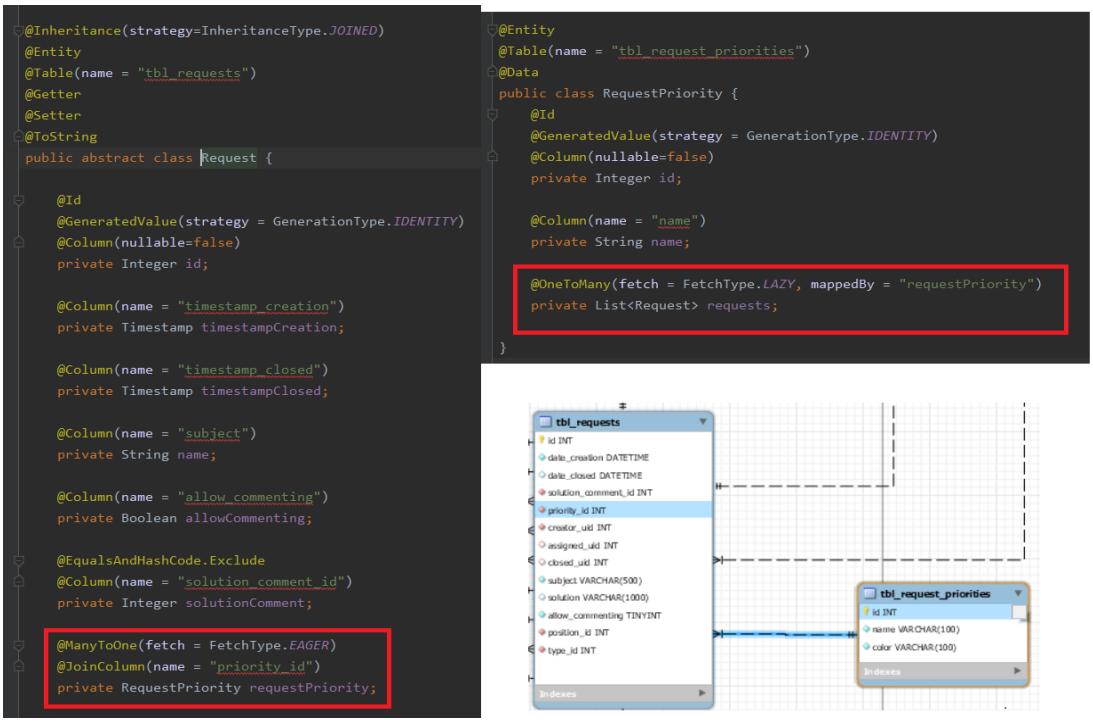
Pozrime sa na príklad načítania detailov jednej požiadavky, ktorú reprezentuje obrázok č. 6. V prvom rade klient musí zaslať http GET žiadosť na server v tvare :
http://localhost:8082/api/requests/requestDetails/X , kde X reprezentuje id požiadavky.
Na strane servera počúva REST kontrolér presne na túto adresu a reaguje len na metódu GET. Každý kontrolér má z bezpečnostného dôvodu zapuzdrenú svoju logiku do try-catch bloku, aby nám v prvom rade systém nepadol a v druhom rade, aby sme mohli užívateľa informovať, ak sa naskytla chyba na servery. Odozva kontroléra je zapuzdrená v objekte *ResponseEntity*, kde klientovi odosielame nielen údaje, ale taktiež aj http status kódy.
Ako bolo vyššie spomenuté, logika aplikácie sa deje v triedach označených anotáciou *@Service*.
V našom prípade metóda *getRequestDetails* v *RequestService,* ktorá načíta požiadavku podľa ID uvedeného v parametri URI. Pomocná metóda *hasAccessForDetails* preverí aktuálne autorizovaného užívateľa, ktorý vykonáva žiadosť na server, či má dostatočné práva na zobrazenie detailov požiadavky.
Ak systém vyhodnotí neoprávnený prístup, vracia status kód 403, čo značí užívateľovi, že žiadosť na API bola legálna, ale server odmietol odpoveď. Logika neoprávneného prístupu s informovaním žiadateľa sa vykonáva na strane klienta chybovou hláškou zobrazenou v malom okienku. V opačnom prípade, ak užívateľ bude autorizovaný, obdrží odpoveď zo servera so status kódom 200, čo je štandardná odpoveď na úspešnú http žiadosť s detailmi požiadavky v json formáte.



Obrázok 6 Logika servera

4.1.2 Hibernate & JPA

Java je OOP jazyk, skladá sa z enkapsulácií, dedenia, polymorfizmu a všetky údaje sa uchovávajú v objektoch. Oproti tomu relačné databázy, ako postgres, ktorý budeme v našom projekt používať, ukladajú údaje do tabuliek. Dizajn objektov sa líši od dizajnu tabuliek, hlavne pri reprezentácii vzťahov vzniká implementačný nesúlad. [13]
Zvážme vyššie spomenutý príklad zobrazenia detailov. Každá požiadavka, tiket, report a pod. sa ukladá do tabuľky *tbl_requests*. Užívateľ taktiež definuje prioritu požiadavky z tabuľky *tbl_request_priorities*. Týmto pádom vzniká vzťah OneToMany medzi *tbl_request_priorities* a *tbl_requests*. Reprezentovanie tabuľky javovským objektom sa volá Object Relational Mapping (ORM).
Najpopulárnejší rámec na ORM mapovanie je Hibernate. Hibernate môže definovať tri typy vzťahov medzi objektami ako *OneToOne*, *OneToMany* a *ManyToMany*. Taktiež nám ponúka možnosť lenivého načítania (angl. lazy loading) prislúchajúceho objektu ku načítavanému.
Lenivé načítania využívame, ak chceme získať entitu bez kolekcie potomka. Na príklade obrázku č. 7 vidíme, že pri zobrazení požiadavky, objekt *Request*, sa nám hibernate, kvôli nastavenému *FetchType.EAGER,* postará o to, aby sa s ňou načítala aj priorita, pretože požiadavka bez priority v našom systéme nemôže existovať.
Na druhej strane, ak chceme načítať len samostatnú prioritu, je nám zbytočné načítať k nej všetky existujúce požiadavky, preto sme zvolili *FetchType.Lazy* v triede *RequestPriority*. Ak by sme zmenili *FetchType* na kolekciu *requests* z *LAZY* na *EAGER*, s čím by sme povedali hibernatu, že pri každom načítaní priority chcem k nim aj kolekciu požiadaviek, vznikla by nekonečne do seba vnorená json štruktúra a došlo by ku chybe.



Obrázok 7 Hibernate OneToMany vzťah medzi prioritou a požiadavkou

JPA (angl. java persistance API) je nadstavba na zjednodušenie práce s Hibernate. Je to implementovateľné rozhranie, ktoré sa stará o persistenciu dát v relačnej databáze. Aby sme povolili JPA, musíme rozšíriť naše interfejsy označenými anotáciou *@Repository* o *crudrepository*, alebo *jparepository*.

4.1.3 Obojsmerné prepojenie

Dlho zaužívané XHR žiadosti na získavanie údajov zo servera tvorili aplikáciu zbytočne komplexnú. XHR je v podstate asynchrónne HTTP, alebo s viac poznaným názvom AJAX volania. Ak by sme s týmto prístupom chceli vytvoriť komunikáciu v reálnom čase medzi klientom a serverom, museli by sme prestúpiť na metódu zvanú polling, kde sa klient pýta servera na nové informácie. Rozlišujeme rozdiel medzi krátkym a dlhým pollingom. V krátkom pollingu sa klient v pevne stanovenom časovom intervale neustále vypytuje servera na nové informácie aj keď žiadne nie sú. Server musí reagovať na každý pokus o pripojenie, spracovať žiadosť a odpovedať. Ak máme priveľa klientov používaných túto techniku, nastavenú na krátky frekventovaný časový interval, tak nevedomky vytvárame DDOS útok na náš vlastný systém.
Ďalší spôsob je dlhý polling, ktorý odpovedá okamžite, čím je efektívnejší z časového hľadiska, ale míňa viacero zdrojov servera. V krátkosti ide a vyslanie GET žiadosť, ktorú si server drží pokiaľ sa údaje očakávané klientom nestanú dostupné. Klient získa dáta a okamžite nadviaže nové prepojenie so serverom. Implementácia tohto spôsobu má hlavne dva nevýhody. Je náročná na míňanie zdrojov servera a za druhé, nie je to štandard, len technika, ktorá môže byť implementovaná na každom serveri inak. [20]
Aké máme teda riešenia na prenos údajov v reálnom čase?

Websoket je komunikačná architektúra, s označením RFC 6455, vytvorená v roku 2011, povoľujúca webovým aplikáciám narábať s full-duplex komunikáciou medzi klientom a serverom cez jedno TCP prepojenie, štandardizovaný spoločnosťou IETF (Internet Engineering Task Force), ktorá vyvíja a propaguje internetové štandardy. Websockety sú navrhnuté na pracovanie cez HTTP port 80 a 443, pričom upgradnú štandardnú http hlavičku na zmenu s http protokolu ku websocket protokolom. Oba HTTP a websockety sa nachádzajú na aplikačnej vrstve OSI modelu a sú závislé na transportnej vrstve. Na rozdiel od bežných XHR, websockety poskytujú obojsmernú komunikáciu v reálnom čase. Akonáhle prebehne prvé nadviazanie so servom cez http, prepojenie sa zmení na novo vytvorené TCP/IP spojenie pomocou websocketov a ďalej sa už žiadne hlavičky správ nepripájajú pri posielaní dát medzi klientom a severom, čím zľahčujú šírku pásma a intenzitu nárokov zdrojov servera v porovnaní s dlhým pollingom. [20]

Websocket je architektúra, definujúca tok bytov, skladané z textov či binárnych súborov. Správy obdržané touto komunikáciou neobsahujú žiadne dodatočné informácie ako ich spracovať, preto komplexné aplikácie si často vyžadujú mnoho pridaného kódu. Našťastie, špecifikácia websocketov sprístupňuje použitie čiastkových protokolov, ktoré prinášajú dodatočné sémantika a pôsobia na vyššej úrovni aplikačnej vrstvy. Jeden z nich je STOMP podporovaný spring bootom a taktiež angularom. [21]

Na spojazdnenie STOMP protokolu v spring boote budeme potrebovať závislosť na knižnicu *spring-boot-starter-websocket*, ktorú si stiahneme len pridaním cez maven. Vytvoríme si triedu *WebSocketConfig*, ktorú označíme anotáciami *@Configuration*, čím oznámime, že trieda deklaruje *@Bean* metódy, ktoré budú spracované Spring kontajnerom. Bean-y sú objekty inštanciované a spracované springovským procesom zvaným Inversion of Control (IoC), v ktorom objekty definujú svoje závislosti bez ich vytvorenia. Triedu taktiež označíme anotáciou *@EnableWebSocketMessageBroker*, ktorou len povolíme spracovanie websocket správ.

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

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/request", "/queue");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("").setAllowedOrigins("*").withSockJS();
    }

}
```

Obrázok 8 Konfigurácia websocketov cez spring boot

Na obrázku č. 8 pozornosť treba upriamiť na metódu *configureMessageBroker*, ktorá vytvára v pamäti message broker s poskytnutými prefixami destinácií na reagovanie správ. V nej registrujeme koncový bod */requests* , cez ktorú sa budú posielať nové, alebo modifikované žiadosti oprávneným užívateľom. V aplikácii treba dbať na bezpečnosť, teda nechceme zmenu jednej požiadavky poslať každému. Našťastie sme registrovali len prefix destinácie, to znamená, že ho môžeme rozšíriť. K vyriešeniu tohto problému sa postavíme nasledovným prístupom. Napíšeme si vlastnú sql funkciu s názvom *get_users_to_send_request_change(searching_request_id integer)* , ktorá dostane ako vstup ID požiadavky a vráti list užívateľských mien, ktorí majú právo byť notifikovaní o zmene. Následne pomocou injekcie závislosti si v springu vytvoríme inštanciu triedy *SimpMessageSendingOperations*, ktorá poskytujúcou metódou *convertAndSend* bude posielať klientom na destináciu */request/${username}* zmenený objekt požiadavky, kde *${username}* reprezentuje užívateľské meno klienta.

Ďalej metóda *withSockJS()* v beane *registerStompEndpoints* zaručuje funkčnosť websocketov, aj keď nie sú podporované prehliadačom. V neposlednom rade metódou *setAllowedOrigins()* povoľujeme spojenie s externým originom, kvôli rozdielnym doménam klienta a servera. Viac ale o tomto v sekcii bezpečnosť - CORS (5.2).

Na strane klienta si pomocou npm pridáme knižnicu *ng2-stompjs*. Vytvoríme si globálnu službu v ktorej budeme počúvať na destináciu *ws://localhost:8082/websocket/${username}* a potom už na základe biznis logiku spracujeme prijaté dáta.
Implementovanie websocketov ešte neznamená, že sme sa úplne vzdali bežných REST operácií a teraz všetku komunikáciu presunieme na websockety. Naopak, tieto prístupy môžeme skombinovať a rozhodnúť sa kedy má väčší zmysel preferovať jedno nad druhou.
Obrázok č. 9 zobrazuje logy úspešného prepojenie klienta so serverom cez STOMP protokol v konzole prehliadača.

Obrázok 9 inicializačné pripojenie na websockety pri prihlásení do systému

4.2. Implementácia Klienta

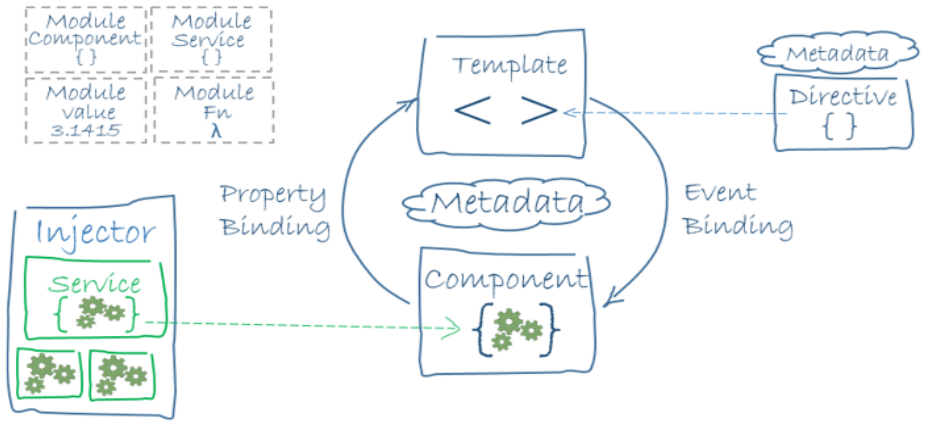
Implementácia klienta bude vytvorená ako jednostránková aplikácia k čomu nám napomôže rámec angular. Výber angularu spočíval z dôvodu nasledovania znovu použiteľných komponentov, jazyku typescript, ktorý je príjemnejší pre OOP programátorov a veľkej komunite vývojárov.

4.2.1 Angular

Štruktúra angularu sa skladá z komponentov, označenými dekorátorom *@Component()*, kde každý z nich obsahuje tvoriacu prezentačnú časť stránky (html), definíciu štýlov (css) a logiku reagujúcu na interakciu užívateľa (typescript). Nasledujú jednoduchý návrh kvôli znovu použiteľnosti v iných častiach aplikácie. Najzakalenejšou stavebnou jednotkou angularu je NgModule, ktorá sprostredkováva kompilačný kontext pre definované komponenty. Každá angular aplikácia má aspoň jeden root modul nazývaným AppModule, ktorá spúšťa aplikáciu a dovoľuje importovanie ďalších NgModulov. Navigáciu užívateľa zachytáva služba s názvom Router, ktorá skrýva a zobrazuje komponenty. [8]

Komunikáciu toku dát medzi nezávislými jednotkami nám napomáhajú služby označením *@Injectable()*, ktoré sa pridávajú pomocou injekcie závislosti do konštruktora komponentu. Služba je širokouhlé označenie funkcionality, typicky triedy, ktorá má presne určený cieľ. Angular rozdeľuje komponenty od služieb pre zvýšenie modularity. Komponenty môžu delegovať úlohy pre služby, ako načítanie údajov zo servera, validácia vstupu formuláru, alebo len slúžiť ako centrálne úložisko údajov. Definovaním spomenutých procesov v separátnej injektovateľnej službe môžeme sprístupniť tieto funkcionality hocijakému komponentu. Keď angular objaví komponent, ktorý je závislí na službe, prv sa pozrie či už v danom moduly existuje inštancia. Pokiaľ inštancia služby neexistuje, injektor jeden vytvorí. V scenári, keď sa nám aplikácia skladá z viacerých modulov a chceme vytvoriť len jednu inštanciu služby v celej aplikácii, tzv. singleton, musíme mu definovať akého modulu má byť súčasťou. Ak nám ide za účel vytvorenia singletonu, definovaný modul musí byť root. V prípade, že definujeme iný modul, angular nám nemusí vrátiť existujúcu inštanciu, ale môže vytvoriť novú. [8]

Npm, manažér balíka node (angl. node package manager) nám uľahčí prácu integrovaním externých knižníc na strane klienta [14].



Obrázok 10 Angular architecture

4.2.2 Stavové a bez stavové komponenty

Komponenty rozdeľujeme na dve skupiny a to stavové a bez stavové, alebo viacej používaným názvom múdre a prezentačné. Kategorizovaním podľa tejto logiky získame flexibilitu a znovu použiteľnosť kódu na viacerých miestach.
Stavové komponenty implementujú služby na vykonávanie biznis logiky, načítavanie údajov zo servera a menenie stavu informácií. Sú to hlavne kostry stránok, ktoré následne udržujú v sebe viacero bez stavových komponentov a odovzdávajú im informácie pomocou tzv. property binding-u.

Prezentačné komponenty sú pripojené ako deti múdreho komponentu (kostry stránky), nemajú prístup k žiadnym službám, sú izolované od aktuálneho stavu aplikácie a slúžia len na zobrazenie informácií. Údaje na zobrazenie im odovzdáva rodič do atribútov označeným anotáciu *@Input()*. Udalosť vyvoľujúca zmenu obsahu zobrazených informácii, ktoré sú prezentované v bez stavovom komponente, sa priamo v nich nemodifikujú. Ako sme spomenuli, hlúpe komponenty slúžia len na prezentáciu údajov. Premennými označeným anotáciou *@Output()* dokážu deti notifikovať rodiča, ktorý reaguje a zmení stav dát na základe vyvolanej akcie. [8]

```
<app-navigation></app-navigation>

<div id='contentContainer'>
    <app-request-table
        #myOpenRequests
        headerColor="#358BF0"
        [numberOfRequests] = 'myOpenRequests.dataSource.data.length'
        [displayedColumns]= "viewTable"
        tableTitle="Moje požiadávky"
        (moveToDetails) = 'moveToDetials($event)'
        [hidden] = " (isGhost$ | async) || (isAdmin$ | async) ">
    </app-request-table>

    <app-request-table
        #meAssignedRequests
        [hidden] = '!(isSolver$ | async) &&  meAssignedRequests.dataSource.data.length === 0'
        [numberOfRequests] = 'meAssignedRequests.dataSource.data.length'
        headerColor="#56A3FF"
        [displayedColumns]= "modifyTable"
        tableTitle="Požiadavky pridelené na mňa"
        (moveToDetails) = 'moveToDetials($event)'
        (removeFromMeEmitter) = "removeFromMe($event)">
    </app-request-table>

    <app-request-table
        #otherOpenRequests
        [hidden] = '!((isSolver$ | async)  || (isGhost$ | async)  || (isAdmin$ | async) ||  otherOpenRe
        [numberOfRequests] = 'otherOpenRequests.dataSource.data.length'
        headerColor="#E8F3FF"
        [displayedColumns]= "viewTable"
        [displayAssignToMe] ='(isSolver$ | async)'
        tableTitle="Ostatné otvorené požiadavky"
        (moveToDetails) = 'moveToDetials($event)'
        (assignOnMeEmitter) = "assignOnMe($event)">
    </app-request-table>
</div>
```

```
@Component({
  selector: 'app-request-table',
  templateUrl: './request-table.component.html',
  styleUrls: ['./request-table.component.scss']
})
export class RequestTableComponent implements OnInit, AfterViewInit {
  @Input() displayedColumns = []; // table columns to diplay
  @Input() headerColor: string; // table header color
  @Input() displayAssignToMe = false;
  @Input() displayDownloadExcel = false;
  @Input() tableTitle: string;
  @Input() numberOfRequests: number;

  @Output() assignOnMeEmitter: EventEmitter<RequestTable> = new EventEmitter<RequestTable>();
  @Output() removeFromMeEmitter: EventEmitter<RequestTable> = new EventEmitter<RequestTable>();
  @Output() moveToDetails: EventEmitter<number> = new EventEmitter<number>();

  @ViewChild(MatPaginator) paginator: MatPaginator;

  dataSource: MatTableDataSource<RequestTable> = new MatTableDataSource<RequestTable>();


  dateFrom: string;
  dateTo: string;
```

Obrázok 11 Dashboard komponent

V odseku *„návrh používateľského rozhrania (3.3)“* sme sa rozprávali o tom, že stránka dashboard bude zobrazovať maximálne tri tabuľky pre užívateľa. Všetky tri tabuľky sú pomerne rovnaké, takže sa z nich spravil jeden prezentačný komponent s čím sme získali moc jeho znovu použitia .
Obrázku č. 11 reprezentuje html a typescript štruktúru stránky dashboard. Vidíme, že je v ňom trikrát implementovaný selektor *app-request-table.*
App-request-table je prezentačný komponent na tabuľku žiadostí, ktorý nedrží žiaden stav. Pomocou data binding-u sa atribútom označeným anotáciou *@Input()* sa priradia statické hodnoty, ktoré sú odlišné pre každú tabuľku, ako titulok, farba a samozrejme údaje načítané so servera a skrz atribúty označenými anotáciou *@Output()* notifikujú rodiča o vykonaní akcie užívateľom, ako priradenie, alebo zmenenie riešiteľa.

4.2.3 Observable & RxJS

Informácie prichádzajúce z asynchrónnych volaní externého zariadenia za určitý časový interval, ktorých úspešný, alebo neúspešný príchod treba spracovať, nazývame tok (angl. stream) údajov. Nemáme žiadne informácie pri vykonávaní žiadosti na API server v akom stave a kedy nám príde odpoveď. *Observable* môžeme chápať ako funkciu obaľujúcu tento asynchrónny tok, ktorý potrebuje odoberateľa na vykonanie účelu a odovzdanie odpovede. Odoberateľ každou novou hodnotou sa notifikuje a reaguje na zmeny. [8]
Moderné reaktívne programovanie dosiahneme knižnicou Rxjs (angl. Reactive extension for Javascript), ktorá sa stavia k všetkému ako k toku dát a implementuje princíp pozorovateľov. Ponúka nám funkcionalitu vytvárania, či modifikovanie údajov, resp. vykonanie vedľajších akcii (ako logovanie do konzoly). Vo výpočtovej vede, reaktívne programovanie je deklaratívne programovanie, ktoré propaguje zmeny. V RxJs zmena vykonaná jedným komponentom bude propagovať notifikáciu všetkým poslucháčom pozorovateľa. [8]

Príklad metódy *login* na obrázku č. 12, ktorá pri aspoň jednom odoberateľovi vykoná http POST žiadosť na server s údajmi poskytnutého užívateľom pri pokuse o prihlásenie do systému. *Pipe* je zjednotenie RxJs operátorov, ktoré sa vykonávajú synchrónne. V uvedenom príklade operátor *tap* vytvára vedľajší efekt a to ukladanie tokenu do interného úložiska prehliadača localstorage, následne sa odpoveď mapujeme na true, aby sme vedeli odoberateľa informovať o správnom vykonaní akcie. V prípade zlyhania požiadavky, alebo poskytnutým nesprávnymi údajmi server vráti chybu. RxJs operátor *catchError* zobrazí chybovú hlášku užívateľovi a metóda vracajúcou odpoveďou false zamietne žiadosť o vstup do systému.

```
login(username: string, password: string): Observable<boolean> {
  const headers = new HttpHeaders().set('Content-Type', 'application/json');

  return this.http.post<Response>( url: environment.loginUrl + "login",
    body: {username: username.toLowerCase(), password},
    options: {headers, observe: 'response'}
  ).pipe(
    tap( next: ressponse => {
      this.setAccessToken(ressponse.headers.get("authorization"));
      // this.setRefreshToken(tokens.refresh_token);
    }),
    mapTo( value: true),
    catchError( selector: error => {
      this.handleAuthenticationError(error);
      return of( args: false);
    }));
}
```

Obrázok 12 Angular POST žiadosť na prihlásenie

Skúsený javascript programátor sa môže zamyslieť a položiť si otázku, prečo *observable* namiesto *promise* ? Charakteristika *promisu* je uvoľnenie jednej hodnoty, nie toku dát. V jednoduchých http žiadostiach na server *promise* dokážu spracovať odpoveď a želaný výsledok je rovnaký. *Promise* taktiež používa dvojicu príkazov *async/await*, ktoré v skratke z asynchrónnych volaní vytvoria synchrónne bez zamrznutia UI. *Promise* sú výborné na reagovanie jednej odpovedi z API, ale ak sa rozprávame o toku údajov, teda viacerých hodnotách prichádzajúci za rôzny časový úsek, *promise* by reagoval na prvý úspešný príchod, ale na zvyšok nie, preto sa musíme sa prikloniť k *Observable*. *Observable* nám taktiež ponúkanie mnoho výhod ako zreťazenie operátor a jeden veľmi užitočný druh *observable* s názvom *behaviourSubject*. Význačná funkcionalita *behaviourSubject* je udržanie si hodnoty a zdieľanie ho s každým odoberateľom. Je to užitočné, ak v aplikácii chceme mať len jednu premennú, tzv. singleton a chceme notifikovať každý komponent ak došlo k zmene. [8]
V našej aplikácii *behaviourSubject* má význam na udržanie údajov prihláseného užívateľa. Komponenty napojené na toto úložisko budú okamžite notifikované, ak sa zmení atribút užívateľa, napr. zmena fotky, alebo ak sa niekto iný prihlási do systému.

4.2.4 Spracovanie údajov

Najväčšia výzva pri budovaní aplikácie je spracovanie údajov. Angular nám ponúka mnoho šablón na odovzdanie údajov medzi komponentami, ktoré môžeme nasledovať, ako dekorátory s označením *@Input()* a *@Output(),* alebo RxJs pozorovateľa názvom *behaviourSubject*, ako sme vyššie spomínali, ktorý reaguje na zmenu hodnoty. Problém, ktorý sa aktuálne snažíme vyriešiť je, že rastúcou komplexitou aplikácie, mnoho od seba nezávislých komponentov modifikuje rovnaké údaje. V našom prípade je to pole požiadaviek natiahnuté zo servera. Vytváranie, uzatvorenie, zmena priority či riešiteľov sa deje na rozličných miestach a zároveň nám server skrz websockety posiela každú zmenu požiadaviek, vyvolané akciou iného užívateľa, na ktoré musíme reagovať. Naša aplikácia sa stáva nepredvídateľnou, debugovanie, testovanie a budúce rozvoje sa ťažšie implementujú.

Riešením je zabudovanie architektúry Redux. Princíp Reduxu je centrálne úložisko, takzvaná databáza na strane klienta, ktorá uchováva stav celej aplikácie ako jeden objekt. Následne múdre komponenty pristupujú k tomuto úložisku, vyhnutím sa potreby posielanie údajov cez dekorátory.

Akcie vyjadrujú unikátne udalosti vykonávané po celej aplikácii, napr. interakciou užívateľa, je to jediný spôsob ako poslať dáta do centrálneho úložiska. Reduktory, čisté funkcie, ktoré produkujú rovnaký výstup na ten istý vstup, obstarávajú prechod celého úložiska z jedného stavu do druhého určením typu vykonanej akcie, synchrónne a bez vedľajších účinkov. Na základe vykonanej akcie rozhodnú či majú vracať nový modifikovaný, alebo originálny stav objektu.

Výhod nasledovania tohto vzoru je hneď mnoho. Stav angularovského projektu je zvyčajne rozčlenený do služieb, kde súmerne s narastajúcou komplexitou, rastie aj zložitosť debugovania a testovania. Ukladanie dát do jednotného objektu získame v celom projekte jeden zdroj pravdy. Údaje v úložisku sa nikdy priamo nemenia, namiesto toho sa vyvolá akcia, na ktorú reaguje reduktor, ktorý vezme zmeny a globálny objekt a vytvorí nový. Vyvolanie rovnakej akcie vracia rovnakú odpoveď, čím je projekt predvídateľný a ľahšie testovateľný. V neposlednom rade Redux nám ponúka plugin do prehliadača s názvom Redux dev tools, v ktorom vieme sledovať všetky vystavené akcie a stav aplikácie v čase s čím uľahčuje debugovanie.

NgRx je open source knižnica, inšpirovaná Reduxom, ponúka reaktívnu správu stavu aplikácie ako jeden zdroj pravdy. Namiesto injekcie závislostí viacerých služieb a komunikáciu medzi nimi, máme len jednu službu a rozdielne komponenty si pýtajú rozdielnu časť úložiska. Predtým, než si vysvetlíme zabudovanie NgRx do nášho projektu si musíme vysvetliť ďalšie dva pojmy, a to selektory a efekty. Uchovaním všetkých údajov na jednom mieste môže objekt prerásť do extrémnych veľkostí a nemá zmysel pozorovateľom tohto objektu odovzdať všetky údaje, kde je len časť úložiska potrebná. Taktiež niekedy musíme vykonať biznis logiku pred odovzdaním údajov z úložiska na čo nám slúžia selektory. Sú to čisté funkcie, ktoré nám umožňujú vybrať potrebné dáta, vykonať zmeny a vrátiť hodnotu pozorovateľom.
V angularovských projektoch bez NgRx sú komponenty zodpovedné na interakciu s externými zdrojmi pomocou služieb. Efekty sú vedľajšie udalosti vykonané na základe typu vyvolanej akcie, je to izolovanie biznis logiky od komponentov určené na komunikáciou s API pomocou http, alebo websocketov a vyvolanie novej akcie s prijatými dátami. [22]

Nasledujúci príklad demonštruje implementovanie NgRx knižnice do našej aplikácie. Presnejšie sa pozrieme na to ako hlavný dashboard komponent, spomenutý v odseku 4.2.2 získava a uchováva otvorené žiadosti pre prihláseného užívateľa.

```
24  export class DashboardComponent implements OnInit, OnDestroy {
25    otherOpenRequests$: Observable<Request[]>;
26    meAssignedRequests$: Observable<Request[]>;
27    myCreatedRequests$: Observable<Request[]>;
28
29    constructor(private userStoreService: UserStoreService,
30                private store: Store<RequestState>) {}
31
32    ngOnInit() {
33      const userName = this.userStoreService.user.username;
34      this.myCreatedRequests$ = this.store.pipe(select(getMyCreatedRequests(userName)));
35      this.meAssignedRequests$ = this.store.pipe(select(getMeAssignedRequests(userName)));
36      this.otherOpenRequests$ = this.store.pipe(select(getOtherRequests(userName)));
37
38      this.store.dispatch(RequestAction.getOpenRequests());
39    }
```

V prvom rade si musíme vytvoriť tri typy akcii. Základná akcia, ktorú komponent odosiela je *getOpenRequests,* riadok 38 na obrázku č. 13. a ďalšie dva na úspešný a neúspešný príchod informácií.

Obrázok 13 NgRx Dashboard

Ďalej vidíme, že máme troch pozorovateľov, riadky 34-36, ktoré z globálneho úložiska, pomocou selektorov filtrujú potrebné informácie pre komponent, presnejšie otvorené požiadavky rozdelené do trocha kategórii a to vytvorené užívateľom, pridelené naňho a zvyšné otvorené. Na začiatku sú spomenutí pozorovatelia bez dát, pretože naše úložisko je prázdne. Údaje sa nachádzajú v databáze a prístup k nim máme cez API volanie. Na ich získanie vyvoláme akciu *getOpenRequests.*

```
25  getOpenRequests$: Observable<Action> = createEffect( source: () => this.actions$.pipe(
26    ofType(RequestAction.getOpenRequests),
27    withLatestFrom(this.store.select(isDashboardLoaded)),
28    filter( predicate: ([_, loaded]) => !loaded),
29    mergeMap(
30      project: () => this.requestHttpService.getRequestOnDashboard()
31        .pipe(
32          map( project: requests => RequestAction.getOpenRequestsSuccess( props: {requests})),
33          catchError( selector: error => of(RequestAction.getOpenRequestsError( props: {error})))
34        )
35    )));
```

Obrázok 14 NgRx effect

Obrázok č. 14 zobrazuje vedľajšiu udalosť, efekt, ktorý automaticky reaguje na typ akcie, riadok 26, ktorú sme v dashboard komponente vystavili a vytvorí nám žiadosť na API pre dáta, riadok 30. Po zaznamenaní úspešného príchodu údajov vyvoláme akciu *getOpenRequetsSuccess,* riadok 32, na ktorý reaguje reduktor a zmení stav úložiska z prázného na naplnení a následne spomenutí pozorovateľia na obrázku č. 13 selektormi získajú údaje a naplnia prezentačné komponenty.

Ak by sa uživateľ navigáciou prepínal medzi komponentami a viacnásobne navštívil dashboard, tak ngOnInit neustále odosiela akciu *getOpenRequests,* na ktorý náš efekt reaguje a stále by sme vytvorili žiadosť pre dáta na server, čím server zaťažujeme. Obdržaním údajov a implementovaním websocketov, nie je potreba viac ako raz vykonať žiadosť na server pre tieto informácie. Problém vyriešia riadky 27-28, na obrázku č.14, ktoré pomocou boolean premennej, *IsDashboardLoaded,* kontrolujú obražanie údaje a zabraňujú viacnásobnému volaniu.

5. Bezpečnosť

Bezpečnosť, oblasť, ktorej dôležitosť sa neustále spomína, ale častokrát sa na ňu zabúda pri implementácií. Vznikom internetu a integrovanie na online obchod sa spoločnosti taktiež otvorili riziku webových útokov, ktoré môžu prísť v rôznych typoch a komplexitách, preto my, programátori musíme byť oboznámení s najnovšími technológiami na zabezpečenie nášho webového produktu. Keďže dáta sú nové zlato v 21. storočí, útoky na vládu každým rokom stúpajú, čím priamoúmerne stúpajú aj výdavky na bezpečnosť. Podľa analytikov z Positive Technologies, výdavky na bezpečnosť v roku 2018 presiahli 120 miliárd celosvetovo. Vystopovanie útočníka je často náročný proces, skrývajú si IP adresy, presmerujú svoje správy cez viacero rútrov, alebo sa utiahnu za proxy server. Najzaužívanejšie útoky sú SQL injection, Cross-site request forgery (CSRF) a v neposlednom rade Cross-site scripting (XSS), na ktorú sa v nasledujúcej kapitole pozrieme hlbšie, a ktorá len sama predstavuje 31% webových útokov. [15]

5.1. Cross-site scripting (XSS)

XSS je zraniteľnosť, ktorá povoľuje útočníkovi vložiť javascript na strane klienta za cieľom pristúpiť ku dôležitým informáciám druhej osoby. Najčastejšie sa tento druh útoku prejaví nedostatočnou kontrolou vstupu užívateľa do formulára pred odoslaním na server. Následne klient na opačnej strane si môže stiahnuť skript vsunutý do formulára útočníkom a kliknutím na tlačidlo, či len samotným načítaním stránky ho spustiť, pretože javascript sa parsuje a vykonáva automaticky prehliadačom. Nižšie si opíšeme ako takýto XSS útok môžeme vykonať úplne jednoducho.

```
<!-- custom input -->
<div class="pretty">
    <mat-form-field>
      <input matInput placeholder="zadajte nieco" [(ngModel)]="inputByUser">
    </mat-form-field>

  <button mat-raised-button color="primary" (click)="display()"> Odoslať</button>
</div>

<!-- print text on the screen -->
<div id="divTest"></div>
```

```
export class RequestFinanceFormComponent implements OnInit {
  inputByUser: any;

  constructor() {
  }

  ngOnInit() {
    document.getElementById( elementId: 'divTest').innerHTML =
      new URL(window.location.href).searchParams.get('query');
  }

  display() {
    // create new URI with custom parameter 'query'
    const newUrl = window.location.protocol + "//" +
      window.location.host +
      window.location.pathname +
      '?query=' + this.inputByUser;

    window.history.pushState( data: {path: newUrl}, title: '', newUrl);

    // display information from parameter 'query' into DIV
    document.getElementById( elementId: 'divTest').innerHTML =
      new URL(window.location.href).searchParams.get('query');
  }
```

Obrázok 15 komponent podliehajúci XSS útoku

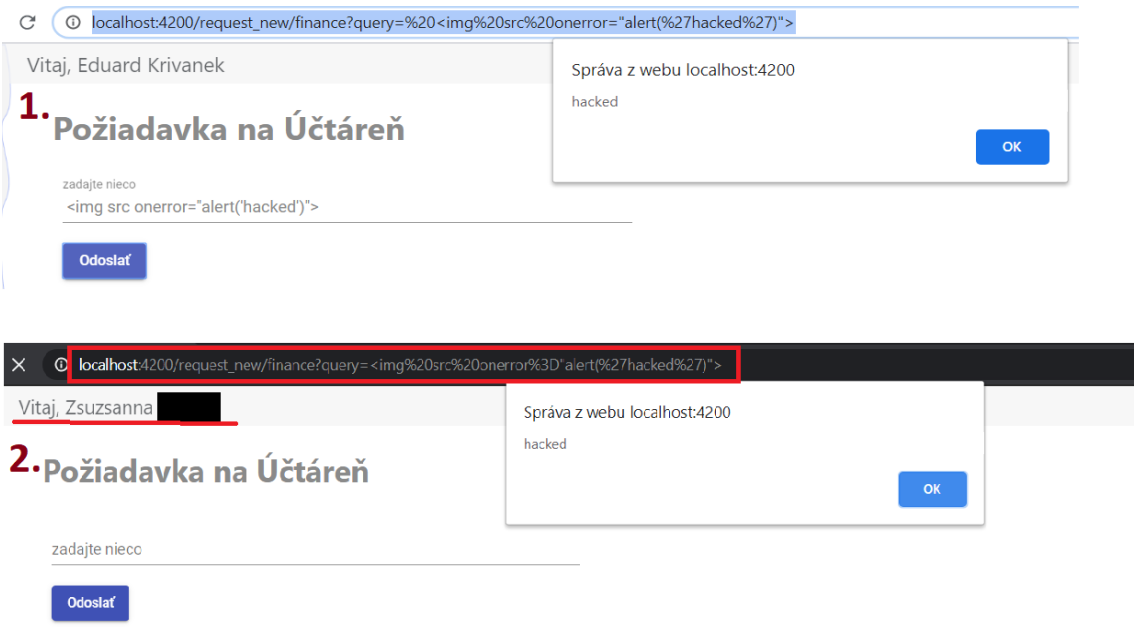
Majme nasledujúci príklad prezentovaný na obrázku č. 15.

Od užívateľa očakávame vstup do formulára označeným *input* a následne kliknutím na tlačidlo (*button*) sa vykoná metóda *display()*, ktorá uložený vstup z formulára v premennej *this.inputByUser* z hocijakých neznámych dôvodov pripojí do URI ako parameter *query*.

Ďalej pokračuje metóda *display()* a z aktuálne pripojeného parametra *query* do URI, vezme informáciu a zobrazí ho v div-e ako *innerHTML*.
Implementovaním interfejsu OnInit docielime to, že ak už existuje v URI adrese parameter s názvom *query*, tak to okamžite vypíšeme do div-u s id *divTest*.

V skratke, vezmeme vstup užívateľa, pridáme do URI s parametrom *query* a vypíšeme ho na HTML bez kontroly.

V tomto príklade je jedna obrovská chyba a to parsovanie parametra *query* do div pomocou *innerHTML*. *InnerHTML* je len bežný javascript príkaz na zobrazenie obsahu do DOM-u. Pokiaľ zadávame bežné vstupy, nič sa nedeje. Zaujímavosť prichádza, ak sa pokúsime pridať javascript.



Obrázok 16 Simulácie XSS útoku

Na obrázku č. 16 vidíme, že pod užívateľom Eduard sme vpísali *img* tag do vstupu, čo indikuje pridanie obrázku. *Img* potrebuje cestu k zdroju, tzv. *src*, aby sa obrázok objavil. Ak ho neudávame, čo sme v našom prípade aj spravili, automaticky sa vykonáva javascript funkcia *onError*, na oznámenie chyby. My ale funkciu *onError* prepíšeme a namiesto zalogovaniu chyby hovoríme javascriptu, aby užívateľovi vyskočilo okienko (angl. alert) s textom „hacked“. Namiesto tohto okienka si predstavme, že náš javascript zozbiera všetky uložené heslá, coockies a sessionID v prehliadači užívateľa a posiela na FTP server.
Otázka je či nás to má vlastne zaujímať, veď predsa sme sami sebe vsunuli javascript a tým pádom sme získali len lokálne informácie. To je pravda, ale ako sme sa vyššie spomínali, naša aplikácia každý vstup sparsovala ako parameter *query* do URI a pri načítaní stránky ak existuje parameter, tak ho vykoná. Čo môžeme teraz spraviť je, že skopírujeme celé URI a pošleme ho inému užívateľovi. Akonáhle si to ten druhý človek otvorí, automaticky mu prehliadač zrealizuje vsunutý javascript z URI parametra, kvôli implementovaniu interfejsu OnInit. Na obrázku č. 15 to môžeme vidieť, ako si užívateľka Zuzana otvorila skopírovanú adresu a automaticky sa jej vykonal vsunutý javascript.

Na krátkej ukážke a troche fantázie vidíme, že XSS útoky môžu byť naozaj škodlivé. Keďže stranu klienta vytvárame pomocou angularu, ako teda využijeme jeho silu proti takýmto útokom ? Sanitizácia, vnútorne zabudovaná funkcionalita angularu, je kontrola voči nedôveryhodným hodnotám skonvertovaním na podobu, ktoré sú bezpečné na vsunutie do DOM-u, alebo využitím Content Security Policy (CSP), ktorá pridáva bezpečnostnú vrstvu nad našu aplikáciu a povoľuje vykonanie javascript súborom len z destinácii, ktoré sa nachádzajú vo white-liste.

```html
<!-- custom input -->
<div class="pretty">
    <mat-form-field>
      <input matInput placeholder="zadajte nieco" [(ngModel)]="inputByUser">
    </mat-form-field>

  <button mat-raised-button color="primary" (click)="display()"> Odoslať</button>
</div>

<!-- print text on the screen -->
<div id="divTest"> {{ inputByUser }}</div>
```

```typescript
export class RequestFinanceFormComponent implements OnInit {
  inputByUser: any;

  constructor() {
  }

  ngOnInit() {
    this.inputByUser = new URL(window.location.href).searchParams.get('query');
  }

  display() {
    // create new URI with custom parameter 'query'
    const newUrl = window.location.protocol + "//" +
      window.location.host +
      window.location.pathname +
      '?query=' + this.inputByUser;

    window.history.pushState( data: {path: newUrl}, title: '', newUrl);

    // display information from parameter 'query' into DIV
    this.inputByUser = new URL(window.location.href).searchParams.get('query');

    /*document.getElementById('divTest').innerHTML =
      new URL(window.location.href).searchParams.get('query');*/
  }
```

Obrázok 17 Zabránenie XSS útoku interpoláciou

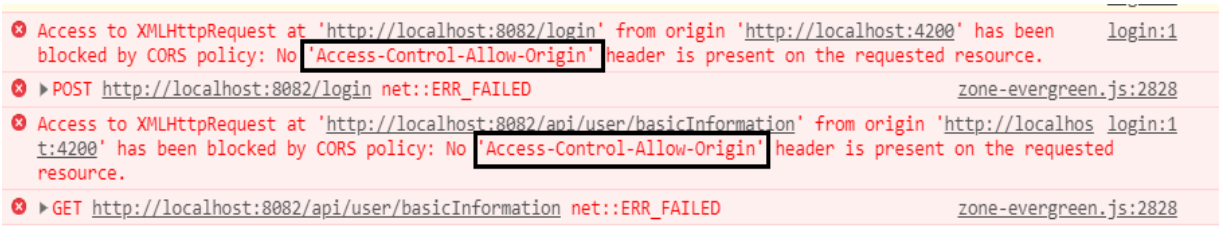
Vyššie uvedenú ukážku, parsovanie parametra *query* do div-u, by sme pomocou angularu vyriešili nasledovne. Časť metódy *display()*, ktorá pripájala obsah do divu pomocou *innerHTML* úplne vymažeme a namiesto toho informáciu uloženú v atribúte *inputByUser* pomocou interpolácie, v zložených zátvorkách, vypíšeme na želanom mieste šablóny HTML. Vďaka tejto malej zmene, prezentovanú na obrázku č. 17, sme sa vyhli vyššie spomenutému XXS útoku.

5.2. Cross-origin resource sharing (CORS)

Žijeme vo svete dát, servery zdieľajú informácie cez verejné koncové body, a aj keď sa to na prvý pohľad nezdá, navštívením jednej stránky, klient cez internet sťahuje zdroje z mnohých API serverov, ako fonty, obrázky, javascript, či dáta. Ak obsah klientovi prichádzaj bez kontroly, vystavuje sa riziku napadnutia škodlivými súbormi, ktoré bežia v pozadí počítača bez nášho vedomia. Ako výsledok, mnoho moderných prehliadačov nasleduje bezpečnostnú politiku na zníženie rizík útokov.

CORS je mechanizmus, ktorá pripája dodatočnú HTTP hlavičku do packetu hovoriac prehliadaču, ktorá beží na jednom origine, aby povolila stiahnutie zdrojov servera, nachádzajúceho sa na inom origine. Origin je kombinácia protokolu, hostiteľa (IP adresy, alebo DNS názvu) a portu. WHATWG (Web Hypertext Application Technology Working Group) je komunita ľudí s účelom rozvíjať HTML a príbuzné technológie. V štandarde "Fetch Living Standard" sú zahrnuté špecifikácie CORS. [16]

Aj keď chyba je viditeľná na strane klienta, je to server, ktorý musí implementovať špecifikáciu CORS, tj. kto má mať prístup k akým zdrojom. Je to nevyhnutné, pretože môžeme povoliť či zakázať vybraným originom modifikovanie perzistenciu dát. Ak server neimplementuje CORS môžeme naraziť do nasledujúcej chybovej hlášky.



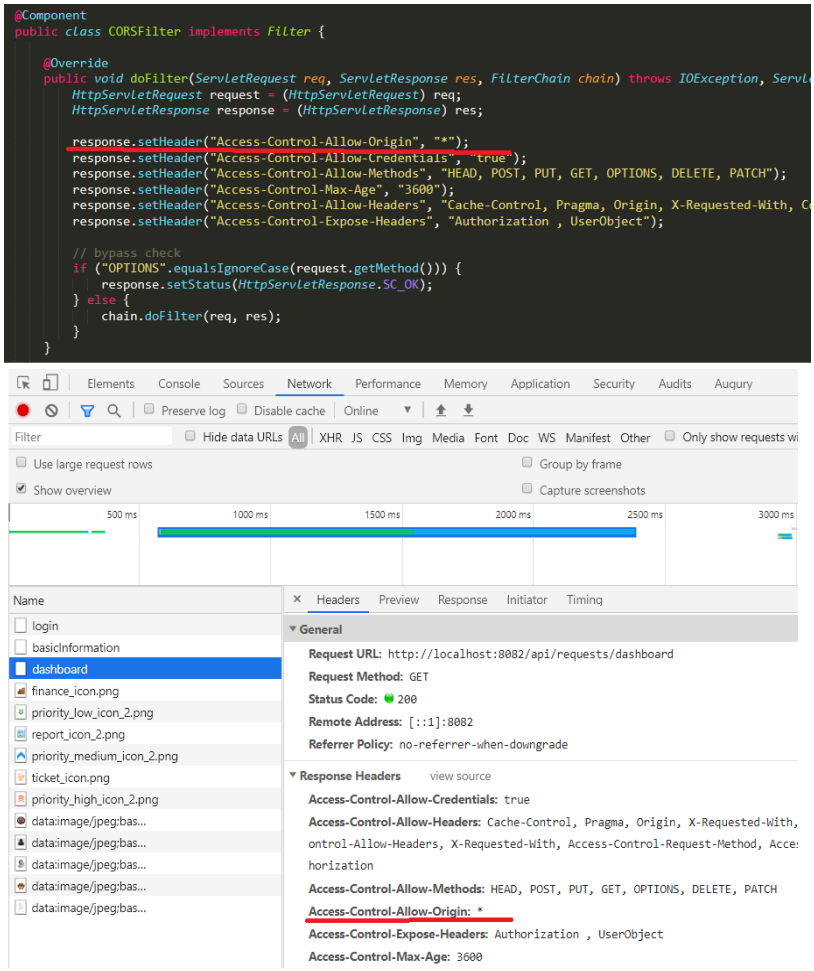
Obrázok 18 CORS chybová hláška

V tomto príklade sa nachádzame na úvodnej prihlasovacej stránke aplikácie a chceme užívateľa autentifikovať. Pri zadaní mena a hesla, klient odošle XHR (XMLHttpRequest) požiadavku na sever, čo je len bežná AJAX komunikácia bez obnovenia stránky, ale dostane chybu zobrazujúcu sa na obrázku č. 18.

Chyba vznikla z obranného mechanizmu nazývaným same-origin policy. Je to implementácia, ktorá bojuje proti kybernetickému útoku s názvom Cross-site request forgery. Pri používaní relácii na klient-server komunikáciu (kapitola 2.4.3), každou HTTP žiadosťou na server prehliadač automaticky pripojí HTTP coockie príbuznú s doménou. Je to užitočné na autentifikáciu. Staršie servery navrhnuté cez relačnú komunikáciu si načítali do pamäte detaily autentifikovaného užívateľa a odoslali ID relácie na jednoduché rozpoznanie, čo bolo automaticky pripojené na každú http požiadavkou smerovanú na daný server, aby sa vyhlo novému načítavaniu prihláseného užívateľa z databázy, implementovali architektúru stavového API. Užívateľ sa následne nemusel prihlasovať ak znovu navštívil rovnakú stránku, pretože bol rozoznaný zaslaným relačným IDčkom. [17]

Z dôvodu neustáleho pripájania relevantného coockie špecifickú na danú doménu prehliadačom, ako relačné ID autentifikácie, aj keď požiadavka je zaslaná z domény útočníka na napr. doménu našej banky, tak bez obranného mechanizmu same-origin policy, útočník získa autorizačné právo obete. Účet obete bolo úspešne hacknuté útokom cross-site request forgery.
Same-origin policy tomuto útoku zabráni, ale je to veľmi limitovaný obranný mechanizmus. Pod touto politikou aplikácie komunikujú medzi sebou len v tom prípade ak sa nachádzajú na rovnakom serveri s rovnakým originom. Čiže ak javascript aplikácia si praje vytvoriť AJAX volanie na API existujúcej na inej doméne, žiadosť bude zamietnutá kvôli same-origin policy. [17]

Chybu zobrazujúcu na obrázku č. 18 sme dostali, pretože klient, budovaný v angulary, beží pod serverom nodeJs na porte 4200 a server pomocou tomcatu na porte 8082. Každá žiadosť na náš server vytvára cross-origin request, kvôli rozdielnym portom. Chyba ja zaznamenaná vo vývojovom prostredí, ale nevieme ako aplikácia bude nasadená, či všetko bude na jednom origine, alebo na úplne dvoch rozdielnych serveroch. Kľúčové slovo "*Access-Control-Allow-Origin*" v chybovej hláške je hlavička v packete, vytvorená na strane servera, ktorá opisuje ako zdroje servera môžu byť prístupné externými doménami. Chyba nám hovorí, že hlavička neexistuje, klient si tým pádom myslí, že nemá právo vykonávať žiadosť na serveri, čím odpoveď zamietne.



Spring boot nám nastavenie CORS-u uľahčí na minimum, viď obrázok č. 19. Naša trieda *CORSFilter* implementuje triedu *Filter*. Znamená to, že každou odpoveďou klientovi sa metóda *doFilter* musí vykonať, ktorá pridá dodatočné informácie do http hlavičky, aké REST metódy sa môžu vykonať a kto má povolenie prístupu na API.

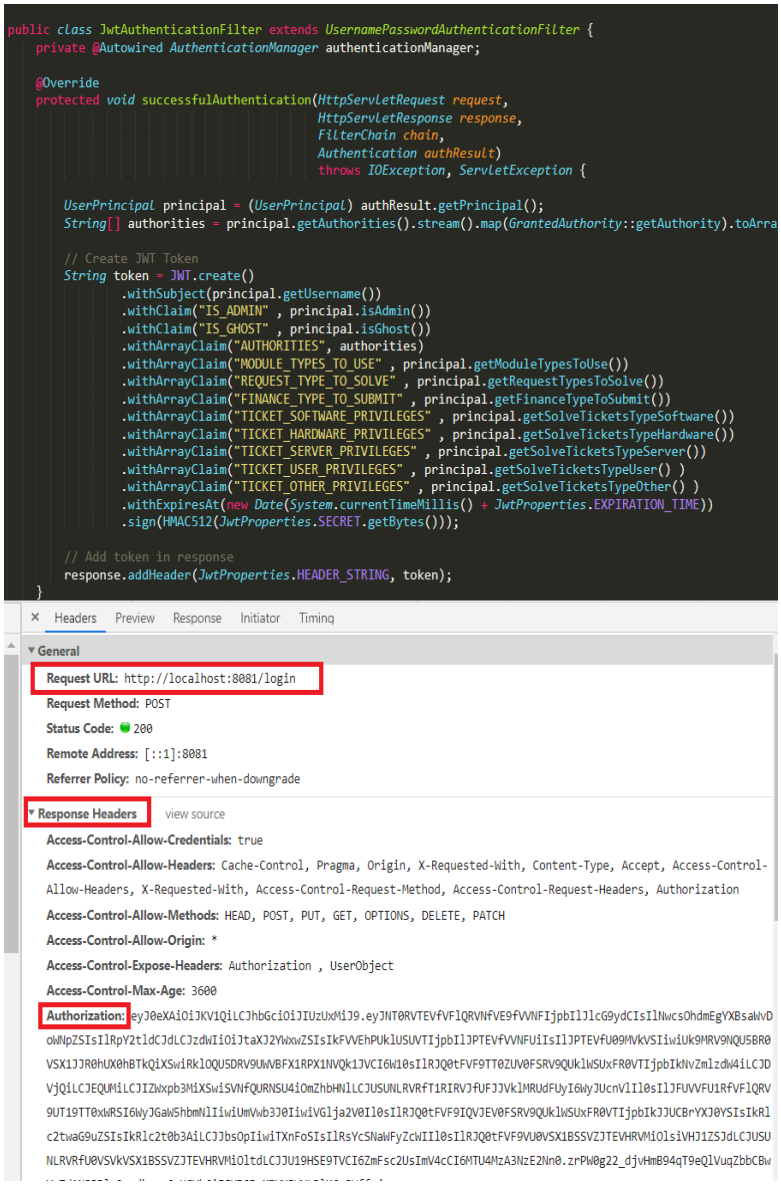
Najpodstatnejšia informácia je *Access-Control-Allow-Origin*, pridaním hviezdičky hovoríme, že všetky žiadosti z hocijakého originu sú povolené. Hviezdička je dostačujúca na vývojové účely, ale v produkčnom prostredí by sa mala definovať škála povolených IP adries.

Implementovaním tohto riešenia sme sa chybovej hlášky v našej aplikácii zbavili.

Obrázok 19 Konfigurovanie CORS-u na servery

5.3. Json Web Token (JWT)

V kapitole 2.4.3, stavové a bez stavové API, sme otvorili tému komunikácie pomocou tokenov. JWT je štandard na vytvorenie prístupového tokenu poskytovateľom zdroja informácií. Funguje to nasledovne. Pri autorizácii, server načíta oprávnenia užívateľa z databázy, následne ich zakóduje algoritmom HMAC 512 do tokenu, zahesluje privátnym kľúčom a odošle klientovi.
Pripájaním tokenu do každej http žiadosti na API, prevažne pod názvom *Authorization,* sa totožnosť klienta ľahko overí, pretože token zahŕňa prihlasovacie meno žiadateľa a právomoc k pristupujúcim zdrojom, čím vieme eliminovať databázové vyhľadávanie na overenie žiadateľa. Ak token nie je uvedený, požiadavka bude automaticky zamietnutá serverom.
Znie to podobne ako vyššie spomenutý CORS, ktorý taktiež posielal dodatočné informácie v hlavičke packetu, len v prípade JWT, dodatočnú informáciu do hlavičky paketu musí pripojiť klient, nie server. Táto architektúra sa preukázala byť veľmi efektívna na moderné webové aplikácie pri vykonávaní žiadostí na REST či GraphQL API. [18]
Poďme si to podrobne rozobrať a uviesť príklad ako to funguje v našej aplikácii.



Obrázok 20 Konfigurácia generovania JWT

Pomocou spring bootu rozšírením triedy *UsernamePasswordAuthenticationFilter* vieme prepísať metódu *successfulAuthentication*, ktorá sa vykoná len v prípade úspešnej autentifikácie.
Principal je aktuálne autorizovaný užívateľ, čiže človek, ktorý sa snaží prihlásiť do systému. V tomto kroku už prebehlo overenie prihlasovacieho mena a hesla v databázovom systéme a generujeme token.
Vytvorenie tokenu nám na serveri pomáha knižnica *java-jwt*, ktorú sme pridali do projektu pomocou maven-u. *Claim* znamenajú práva užívateľa, čo môže a nemôže robiť v aplikácii, zakódované v tokene.
Token má expiračnú dobu. Po jeho vypršaní systém automaticky odhlási užívateľa pre novú autorizáciu. Je to z dôvodu bezpečnosti proti krádeže tokenu. Nakoniec sa token podpíše privátnym kľúčom a pošle na stranu klienta.
Obrázok č. 20 prezentuje nastavenie vytvárania tokenu na serveri s úspešným odoslaním klientovi na POST žiadosť v hlavičke pod kategóriou Authorization .

Manuálne modifikovanie hlavičky každého AJAX dopytu na sever s potrebou pripojenia tokenu by bol náročný proces a zmeny by vyžadovali opravu na mnohých miestach. Interceptory sú služby vo svete angularu s jediným účelom, spracovať http dopyt pred zaslaním, alebo reagovať na každú odpoveď servera. Interceptormi chceme vytvoriť injekčnú triedu, teda singleton, ktorá implementuje rozhranie *HttpInterceptor*. Trieda by následne mala definovať metódu *intercept* s dvoma argumentami, prvá typu *HttpRequest*, kde vieme pridať extra informácie do hlavičku požiadavky a *HttpHandler*, ktorá len posiela požiadavku ďalej na API. [8]
V našom projekte interceptory implementujeme na pridanie autorizačného tokenu pri každej žiadosti na server a na informovanie užívateľa okienkom, ak došlo ku chybe pri XHR žiadosti.
V JWT sme vygenerovali všetky práva užívateľa. Využitím najbežnejšej bezpečnostnej techniky angularu, nasadíme tzv. strážcov (*guards*), ktorí vyhodnocujú prístup užívateľa na jednotlivé podstránky, kontrolou existencie určitých oprávnení v JWT tokene. [8]

6. Najčastejšie chyby pri vývoji

6.1. Angular únik pamäte

V Laymanovej terminológii, únik pamäte vzniká ak aplikácia zlyhá v zbavení sa nepoužívaných zdrojov s čím sa znižuje jej výkonnosť. Únik pamäti je jeden z najčastejších a najhorších typov problému s ktorým sa programátor musí vysporiadať. Náročne sa hľadá, debuguje, či rieši. *Observable* v angulari sú úžasné kvôli toku dát a reaktívnemu programovaniu, ale tieto benefity prinášajú aj seriózny problém s únikom pamäte. Neustále pri vytváraní *subscription* na *observable*, vytvárame referenciu v pamäti. Ak nie sme opatrní, alebo zabúdame odstrániť túto referenciu, bude si ďalej žiť, aj keď komponent, v ktorej bol *observable* vytvorený už dávno neexistuje. Keďže v angulari komponenty majú životnosť pokiaľ sú zobrazené, takpovediac prepínaním sa v navigačnej lište neustále vznikajú a umierajú, lenže zabudnutím odstránenia referencie *observable* sa nám zdroje pamäti iba alokujú, ale nikdy sa neuvoľnia, čo vedie k serióznemu spomaleniu aplikácie v závislosti od jej veľkosti. Najpoužívanejšie spôsoby na anulovanie pozorovateľov je implementovanie interfejsu *OnDestroy*, s čím sa nám automaticky pri zrušení komponentu zavolá metóda *ngOnDestroy()* v ktorej môžeme aplikovať metódu *unsubscribe()* na každú vytvorenú referenciu pozorovateľov v komponente. Ďalšie spôsoby mazania pozorovateľov z pamäte je využitie RxJs operátora *takeUntil(),* alebo *async* pipe, ktorá automaticky vykonáva metódu *unsubscribe()* pri zrušení komponentu. [8]

6.2. Angular detekcia pohybu

Angular dokáže detekovať zmenu dát a automaticky re-renderovať prezentačnú časť komponentu reflektovaním zmien pomocou vnútorne zabudovanej knižnice Zone.js. Detekcia zmien môže byť vyvolaná manuálne, alebo skrz asynchrónny event, ako XHR požiadavka. Keď angular vytvára DOM nody na renderovanie obsahu šablóny, potrebuje miesto na držanie referencii DOM nod-ov. Pre tento účel, existuje vnútorne zabudovaná dátová štruktúra zvaná View. Zahŕňa referencie na komponenty a ich predošlé stavy. Zobrazenie medzi komponentom a View je bijektívne. Kompilátor analyzovaním šablóny identifikuje úseky DOM elementu, ktoré musia byť aktualizované a vytvára viazanie medzi aktualizovateľnou hodnotou a angularovskou štruktúrou na získanie nových hodnôt. Následne angular skontroluje všetky viazania generované pre daný View a vyhodnocuje aktuálnu hodnotu s hodnotami uloženými vo vlastnom poli predošlých dát. Ak sa zdetekuje zmena, úseky DOM elementu sa aktualizujú relevantne k naviazanej hodnote a uložia sa do poľa hodnôt vnútornej štruktúry angularu.[19]

```
<div id='headerBackground' *ngIf="user$ | async as user">
    <div  id='headerContainer'>
        <div id="headerTitle">
            Vitaj, {{ getFullName() }}
        </div>

    </div>
</div>
```

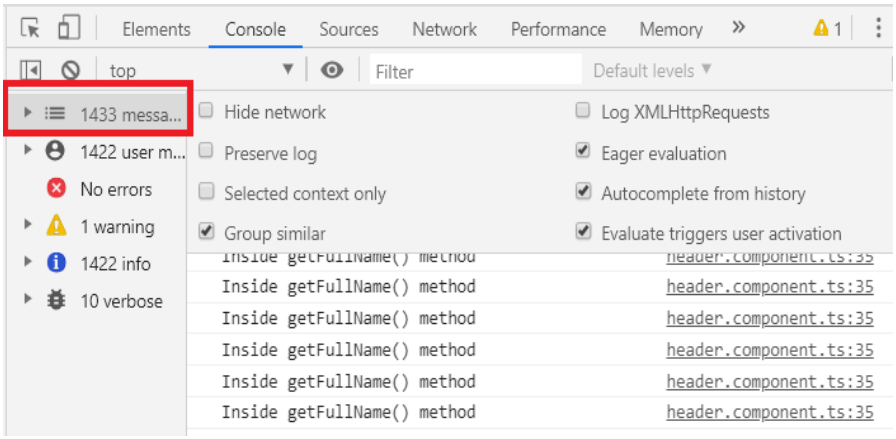
```
export class HeaderComponent implements OnInit {
 user$: Observable<User>;

 getFullName() {
   console.log("Inside getFullName() method");
   return this.userStoreService.user.fullName;
 }
```

Obrázok 21 Angular header komponent

Vezmime si príklad z našej aplikácie na demonštráciu detekcie zmien a vážnosť problému.
Obrázok č. 21 prezentuje hornú lištu stránky, kde sa nám zobrazuje meno aktuálne prihláseného užívateľa. Aby sme vypísali jeho meno voláme metódu *getFullName()* v ktorej pomocou injekčnej služby vraciame meno užívateľa. Testovací výpis do konzoly bude dôležitý neskôr pri debugovaní. Mysleli by sme si, že všetko je v poriadku, lenže sa tu vyskytla obrovská chyba.

Po každej detekcie cyklu zmeny, angular synchrónne spustí ďalší obeh na preverenie rovnakej hodnoty v šablóne ako v predošlej kontrole. Jadro funkcionality angularu hovorí, že pri každej zmene stavu DOM objektu, všetky sekcie HTML štruktúry, ktoré volajú metódy musia byť prehodnotené. Výsledky metód sa neuschovávajú vo vnútornej pamäti, takže neustále sa musia vykonať a vrátiť výsledok. Vrátenie nového výsledku je nekonzistencia s aktuálne zobrazenou hodnotou v šablóne, takže angular vykoná ďalšiu detekciu zmien na obnovenie stavu UI. Zmenil sa nám stav DOM elementu vrátením hodnoty z metódy a ako sme vyššie spomenuli, pri každej zmene DOM-u sa výsledky metód musia prehodnotiť, čím sa vytvoril nekonečný cyklus. [19]



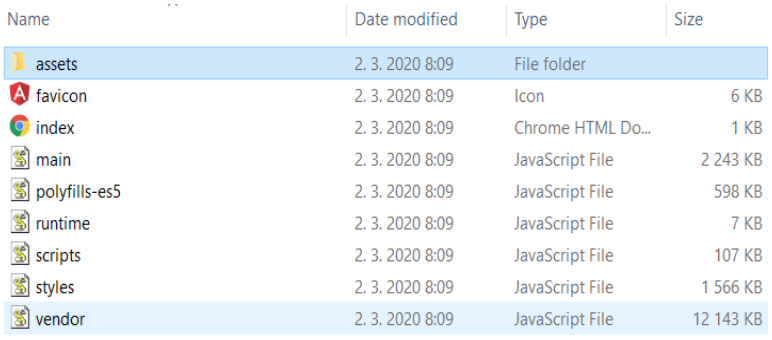
Logovaním reťazca do konzoly môžeme sledovať ako za pár sekúnd nám detekcia zmeny spôsobila nekonečný cyklus vyhodnocovania metódy *getFullName()* a rapídne spomalila responzivitu aplikácie.

Obrázok 22 Angular logovanie detekcie pohybu

Našťastie oprava uvedeného problému len jednoduchá. Prvá možnosť je modifikovanie konfigurácie detekcie zmien komponentu. *ChangeDetectionStrategy.OnPush* je funkcionalita, ktorá vynecháva kontrolu komponentu a reaguje len na prípady zmeny atribútov označením anotáciu *@Input()*, alebo príchod novej hodnoty skrz *observable* použitím *async* pipy. Stratégia *OnPush* je veľmi preferovaná na prezentačné komponenty, keďže sa všetky hodnoty zobrazené v šablóne získavajú pomocou *@Input()* a informujú rodiča cez *@Output()*. Vyriešenie spomalenia aplikácie na základe detekcie je triviálna, náročná časť práce je zistenie tejto chyby, pretože vývojár nie je nijak informovaný o výskyte problému.

6.3. Angular optimalizácia zväzku súborov

Dostávame sa do poslednej fázy, naša aplikácia je hotová a chceme ju nasadiť. Využijeme pomoc angular CLI a príkazom *ng build* skompilujeme projekt.



Na obrázku č. 23, vytvorenie zväzku, sú najpodstatnejšie dva súbory a to vendor.js, ktorá zahŕňa všetky naimportované externé knižnice a main.js, v ktorej sa nachádza náš kód. Na prvý pohľad sa zdá, že všetko je v poriadku.

Obrázok 23 Neoptimalizovaný skompilovaný projekt do zväzku

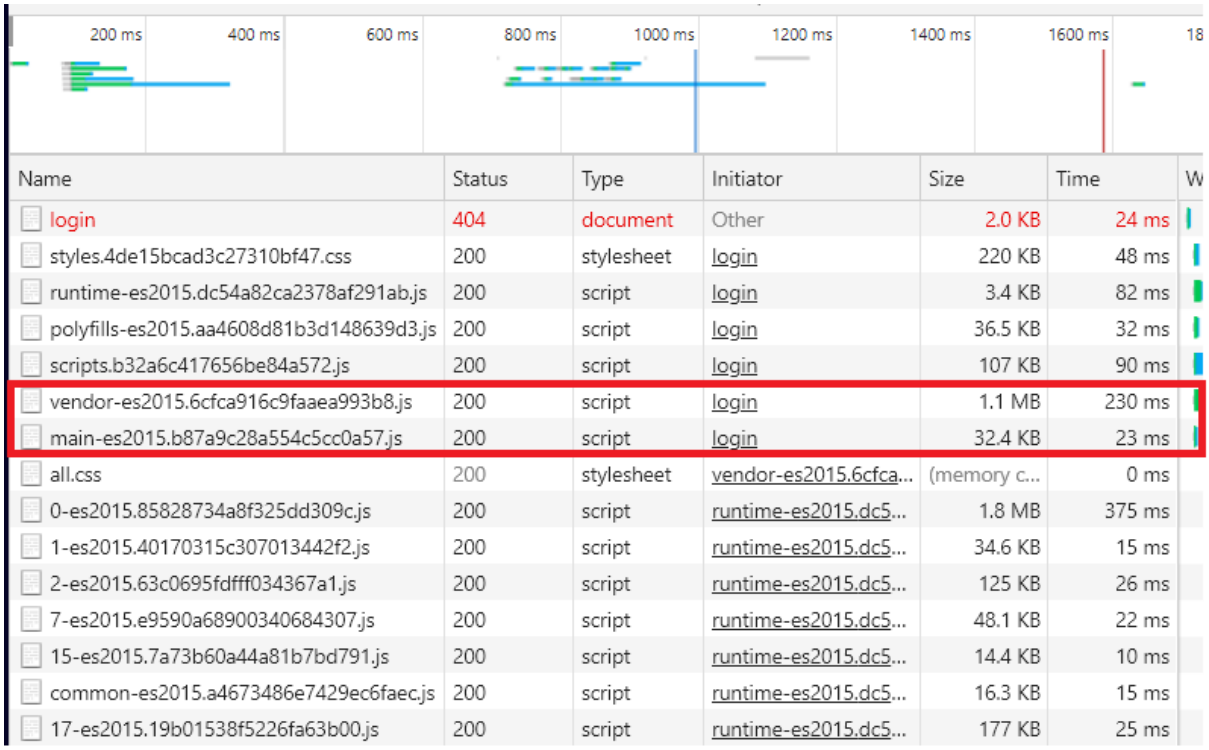
Najdiskutovanejšiu témou pri nasadzovaní je zníženie veľkosti súborov na minimum a to hlavne vyššie spomenutých dvoch. Akonáhle nám main.js začína rásť, výkon načítania aplikácie ide exponenciálne dole, pretože každým pridaným extra KB na strane klienta musí prebehnúť sťahovanie, parsovanie a vykonanie javascriptu. Poďme sa trocha pozrieť na optimalizačné techniky angularu.

Predvolebne NgModules je nastavené na skoré načítanie, čo znamená, že akonáhle otvoríme našu aplikáciu, na inicializujeme všetky komponenty a moduly pripojené v hlavnom NgModule. V skratke sa nám stiahnu a sparsujú všetky stránky aplikácie, ešte pred zobrazením úvodnej prihlasovacej obrazovky. Tento prístup je neefektívny, pretože jednak musíme dlho čakať a nemá zmysel sťahovať užívateľovi sekcie, ku ktorým má zakázaný prístup.

Na vyriešenie tohto problému použijeme techniku *lazy* routing so stratégiou *PreloadAllModules*. Odstránime všetky komponenty, okrem úvodného prihlasovania v hlavnom NgModule. Prihlasovanie je jediná stránka, ktorú si každý užívateľ môže otvoriť, preto sa musí zobraziť okamžite pri spustení aplikácie. Zvyšné úseky budú v samostatných moduloch, čo stránka, to vlastný ngModule, v ktorom sa budú nachádzať len komponenty využité daným modulom. Komponenty zdieľané po celej aplikácii, ako tabuľky, tlačidla a pod. sa oddelili do vlastného shared modulu. Pri navigácii na nenavštívené úseky aplikácie, užívateľ bude zaznamenať po prvýkrát pomalšie zobrazenie. Z dôvodu *lazy* routingu, naša aplikácia je vedomá len *login* komponentu a navštívenie zvyšných podstránok sa komponenty musia najprv stiahnuť a sparsovať. Vyhnutiu sa tohto problému a spríjemnenie dojmu používateľovi nám angular ponúka stratégiu *PreloadAllModules*, ktorá hneď po načítaní aplikácie (súboru main.js) nám v pozadí posťahuje zvyšné moduly a nezaznamená sa žiadne oneskorenie zobrazovania pri navigácii.

Využitím angularu verzie 9. nám pribudol kompilátor Ivy. Ivy bol vyvinutý na odstránenie nepoužitých častí angularu cez tree-shaking, generovanie menšieho kódu, optimalizovanie zväzkov, testovanie a pod. Ivy má taktiež prednastavené ahead-of-time (AOT) kompiláciu. Pretože komponenty a šablóny poskytované angularom sú nepochopiteľné priamo pre prehliadače, angular si vyžaduje kompiláciu pred spustením. AOT skompiluje angularovskú HTML a typescript časť na efektívny javascript ešte pred priamym nasadzovaním na produkčné prostredie, čím prehliadače sa o túto činnosť nemusia starať, len si spustia kód. [8]
Nakoniec namiesto jednoduchého *ng build* použijeme príkaz *ng build --prod --aot --vendor-chunk --common-chunk --delete-output-path --buildOptimizer*, ktorý pozapína dodatočné optimalizačné techniky.

Na obrázku č. 24, ktorý zobrazuje načítanie modulov v prehliadači cez network kartu, môžeme vidieť že z vyššie spomenutými optimalizačnými technikami sme main.js z 2,2MB znížili na 32KB a vendor.js z 12MB na 1.1MB. Zároveň vieme spozorovať (tie modré fľaky pri 800ms), že po inicializovaní týchto dvoch súborov sa nám po malom časovom úseku začali sťahovať zvyšné stránky aplikácie kvôli stratégii *PreloadAllModules*.



Name	Status	Type	Initiator	Size	Time	W
login	404	document	Other	2.0 KB	24 ms	
styles.4de15bcad3c27310bf47.css	200	stylesheet	login	220 KB	48 ms	
runtime-es2015.dc54a82ca2378af291ab.js	200	script	login	3.4 KB	82 ms	
polyfills-es2015.aa4608d81b3d148639d3.js	200	script	login	36.5 KB	32 ms	
scripts.b32a6c417656be84a572.js	200	script	login	107 KB	90 ms	
vendor-es2015.6cfca916c9faaea993b8.js	200	script	login	1.1 MB	230 ms	
main-es2015.b87a9c28a554c5cc0a57.js	200	script	login	32.4 KB	23 ms	
all.css	200	stylesheet	vendor-es2015.6cfca...	(memory c...	0 ms	
0-es2015.85828734a8f325dd309c.js	200	script	runtime-es2015.dc5...	1.8 MB	375 ms	
1-es2015.40170315c307013442f2.js	200	script	runtime-es2015.dc5...	34.6 KB	15 ms	
2-es2015.63c0695fdfff034367a1.js	200	script	runtime-es2015.dc5...	125 KB	26 ms	
7-es2015.e9590a68900340684307.js	200	script	runtime-es2015.dc5...	48.1 KB	22 ms	
15-es2015.7a73b60a44a81b7bd791.js	200	script	runtime-es2015.dc5...	14.4 KB	10 ms	
common-es2015.a4673486e7429ec6faec.js	200	script	runtime-es2015.dc5...	16.3 KB	15 ms	
17-es2015.19b01538f5226fa63b00.js	200	script	runtime-es2015.dc5...	177 KB	25 ms	

Obrázok 24 Optimalizovaný skompilovaný projekt do zväzku

7. Záver

Projekt bol úspešne nasadený do testovacieho prostredia dátumom 7.2.2020, kde sa ešte vyskytnuté chyby opravili a do produkčného prostredia dátumom 21.2.2020 s odovzdaním manuálom použitia. Aktuálne sa v ňom nachádzajú tri moduly na adresovanie požiadaviek rozdielnym oddeleniam firmy a modul na správu celej aplikácie, ktorú majú pod kontrolou interní zamestnanci, takže vzdialené modifikovanie aplikácie nie je potrebné.

Plány s vývojom v tejto fáze ale nekončia. Uvažuje sa o dodatočných moduloch, ako modul na interné špecifikácie. Autorizovaní užívatelia by mohli vytvoriť vnútornú špecifikáciu, ktorá musí spĺňať štandardné kroky stanovené firmou, musí prejsť schválením manažérov a napokon byť zásadne preštudovaný ostatnými zamestnancami.

Z osobného uhľa pohľadu, náročnosť projektu mi prinieslo porozumenie rôznym architektúram systému, pre ktoré sa aplikácia musela viackrát refaktorovať a rozbiť na menšie celky, kvôli dodržaniu zásad čistého kódu. Projekt mi taktiež zlepšil analýzu potreby implementovania princípu Redux na strane klienta, ktorá uľahčuje spracovanie a predvídateľnosť zmeny údajov, ale prináša ďalšiu vrstvu komplexity, má náročnejšiu fázu učenia a nie je nevyhnutná na vytváranie systémov v angulari oproti reactu.

Zdroje

1. Jira https://www.atlassian.com/software/jira [z dňa 3.4.2020]
2. Bugzilla www.bugzilla.org [z dňa 3.4.2020]
3. ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems http://cabibbo.dia.uniroma3.it/ids/altrui/ieee1471.pdf [z dňa 3.4.2020]
4. Ian Gorton: Essential of Software Architecture (2. ed.), 2011
5. Mark Richards & Neal Ford: Fundamentals of Software Architecture: An Engineering Approach, 2020
6. Ibm introduction to SOA https://www.ibm.com/support/knowledgecenter/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html [z dňa 3.4.2020]
7. MPA vs SPA https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps [z dňa 3.4.2020]
8. Angular https://angular.io/ [z dňa 3.4.2020]
9. Leonard Richardson & Sam Ruby: RestFul web Services, 2007
10. Robin Williams, The Non-Designer's Design Book , 1994
11. Spring boot https://spring.io/projects/spring-boot [z dňa 3.4.2020]
12. What is maven? https://maven.apache.org/what-is-maven.html [z dňa 3.4.2020]
13. Hibernate https://www.tutorialspoint.com/hibernate/index.htm [z dňa 3.4.2020]
14. What is npm https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/ [z dňa 3.4.2020]
15. Positive Technologies https://www.ptsecurity.com/ww-en/analytics/web-application-attack-statistics-2017/ [z dňa 3.4.2020]
16. Fetch standard https://fetch.spec.whatwg.org/ [z dňa 3.4.2020]
17. What is CORS https://www.codecademy.com/articles/what-is-cors [z dňa 3.4.2020]
18. Json web token https://jwt.io/ [z dňa 3.4.2020]
19. Angular change detection https://indepth.dev/a-gentle-introduction-into-change-detection-in-angular/ [z dňa 3.4.2020]
20. Communicating and displaying real-time data with websockets https://www.it.iitb.ac.in/frg/wiki/images/b/bf/113050009_websockets.pdf [z dňa 3.4.2020]
21. Spring boot STOMP https://docs.spring.io/spring-framework/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html#websocket-stomp-overview [z dňa 3.4.2020]
22. NgRx https://ngrx.io/ [z dňa 3.4.2020]

ZOZNAM OBRÁZKOV