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 úloha a kto vlastne zadal úlohu.

Ak neexistuje interný systém pre 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 zadania týchto požiadaviek.
Neefektívna komunikácia môže viesť k zabudnutiu vyriešenia úloh, informácie požiadaviek môžu byť nekompletné, stráca sa prehľad stavu úlohy 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, kam bude nasadená, takže špecifikácia úlohy je prispôsobená pre klienta.

2. Analýza problematiky

2.1. Požiadavky klienta

Systém musí v prvom rade vedieť evidovať rôzne druhy požiadavkou (tikety, reporty, financie, projekty, a pod.), ktoré sa posielajú, každé na iné oddelenie, kde autorizovaný užívatelia, iným slovom riešitelia týchto požiadaviek môžu na nich pracovať. Každý druh požiadavky bude mať vlastný prispôsobený formulár s povinnými políčkami.

Užívatelia si môžu požiadavky, ktoré majú právo otvárať, zadať do stavu sledovania. To znamená, že o každom komentári a každej malej zmene v požiadavke budú v systéme notifikovaný.
Vlastné komentáre užívateľa požiadavkám sa dajú zmeniť so stavu verejné na stav privátny a zdieľať so skupinami, v ktorých je užívateľ členom.
Taktiež chceme v logoch zaznamenať každú zmenu požiadavky, ako zmenu priorita, riešiteľa, uzatvorenie a pod. , ktoré sa zobrazia v detailoch požiadaviek. Taktiež generovanie štatistík koľko požiadaviek užívateľ vytvoril, resp. vyriešil.

Ďalej aplikácia bude slúžiť ako školiaci systém, kde autorizovaný užívateľ (väčšinou pracovníci z oddelenia ľudských zdrojov) budú môcť zdieľať dokumenty ako vnútorné predpisy vybraným užívateľom, ktorí (kliknutím na checkbox) súhlasia s obsahom dokumentu.

Každý užívateľ bude mať prispôsobený kalendár, kde sa mu zobrazia hraničné termíny, ale taktiež si vie vytvoriť vlastnú udalosť na určitý dátum a zdieľať ju s inými pracovníkmi. Účastníci udalosti s blížiacim termínom budú notifikovaný emailom.

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 skupiny musí mať možnosť monitorovať aktivitu skupiny, resp. aby si na túto úlohu mohol on sám niekoho zvoliť. Taktiež chceme do systému ďalšie dva typy užívateľov. Admina, ktorý môže manipulovať z celou aplikácie a užívateľmi v nej a „ducha“, typ užívateľa, ktorý vidí všetko, ale nemôže robiť nič.

2.2. Existujúce riešenia

2.2.1. Jira

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

Scrum je spôsob vývoja aplikácii, kde ľudia sa rozdeľujú do troch rolí. Vlastník produktu zodpovedajúci za definovanie funkcii 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 vyberie z najvyššou prioritou, ktorá by mohla 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 spraviť. [3]

2.2.2. OsTicket

Ďalším systémom, ktoré spĺňa požiadavky klienta je aplikácie OsTicket.
Je to open source projekt, ktorá uchováva všetky tikety užívateľov na jednom mieste.
OsTicket ponúka možnosť vytvárania vlastných formulárov pre rôzne typy problémov, ich triedenie na správne oddelenie, prehadzovanie medzi oddeleniami a sledovanie histórie zmien.

OsTicket taktiež umožňuje uzamknutie úlohy len pre jedného riešiteľa, aby sa nedošlo k dvojitej odpovedi a podporuje generovanie reportov pre pohľad aktivity a výkonu helpdesku. Sprevádza inštaláciou, integráciu a migráciou. [2]

3. Architektúra informačných systémov

3.1. Klient – server architektúra

Klient – server model je distribuovaná aplikačná štruktúra, ktorá presne definuje hranicu medzi poskytovateľom služby, alebo zdroja nazývaným server a žiadateľom služby nazývaným klientom.
Komunikácia klient – server prebieha cez protokol http, prevažne na rozdielnych zariadeniach a rozdeľujeme ju do troch kategórii : rozhranie užívateľa, business logika a perzistencia dát.
Mnohé servery dnes už poskytujú služby nielen jednému, ale viacerým klientov, preto pri navrhovaní modelu klient- server musíme uvažovať medzi tenkým a hrubým klientom. [1]
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 servera klesá a narastá počet relácii na strane servera (angl. server side sessions), ktoré môžu

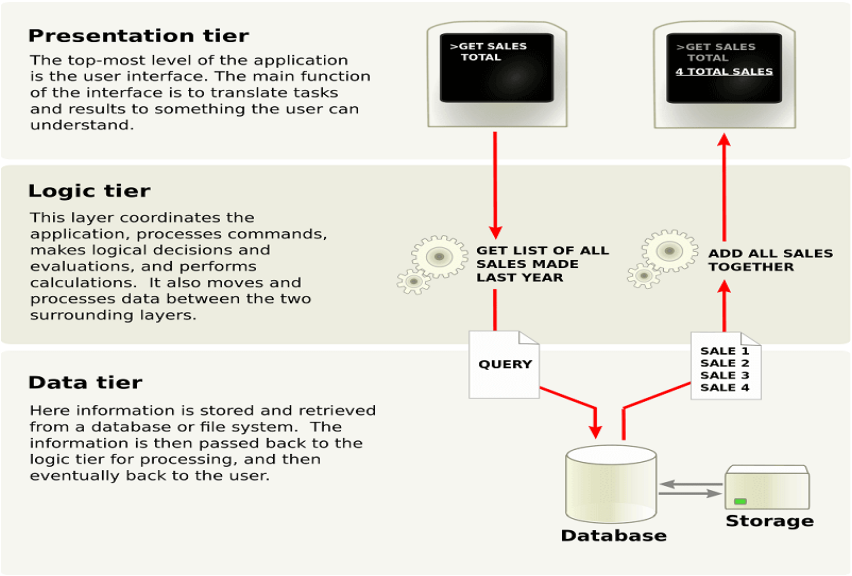
obsahovať veľké objektové štruktúry. Relácie medzi klientom a serverom sa udržiava fixný časový úsek, alebo je perzistentná až do odhlásenia užívateľa. Týmto sa zdroje servera rýchlo vyčerpajú a klesá výkonnosť.
Alternatívou tohto modelu je hrubý klient a tenký server. Narastajúcim sa výkonom počítačov 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 užívateľov sa presunulo na stranu klienta, kde väčšina informácii sa ukladá do browsera (coockies, localstorage, sessionstorage) a následne sa nevyhnutné informácie klienta 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 business logika, modifikácie dát a bezpečnosť by mala byť oddelená na stranu servera.
Návrh modulárneho, znovu používateľného 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. [2]

3.2. Architektúra servera

Pri zvolení správnej architektúry softvér sa 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 aplikácie, ktoré sú náročné na zmenu ak sú raz implementované. Taktiež pri návrhu architektúry je možné viaceré kombinovať dokopy.

3.2.1. Monolitická architektúra

Monolitická architektúra (angl. Monolithic architecture) je považovaná za tradičný spôsob tvorby aplikácii, kde výsledkom celého produktu je jedna veľká aplikácie, ktorá časom len rastie.
Ak monolitický prístup implementujeme na stranu servera hovoríme o tom, že jedna veľká jednotka reaguje na http požiadavky, vykonáva business 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. [3]
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 kontroléry, business logiku, entity a databázovým prístupom. Každá vrstva sa môže samostatne zabezpečiť, resp. škálovať. Vrstvová architektúra sa ujala aj v praxi, kde každý tím môže pracovať na vlastnej vrstve softvéru. [4].



Obrázok 1 troj vrstvová monolitická architektúra [3]

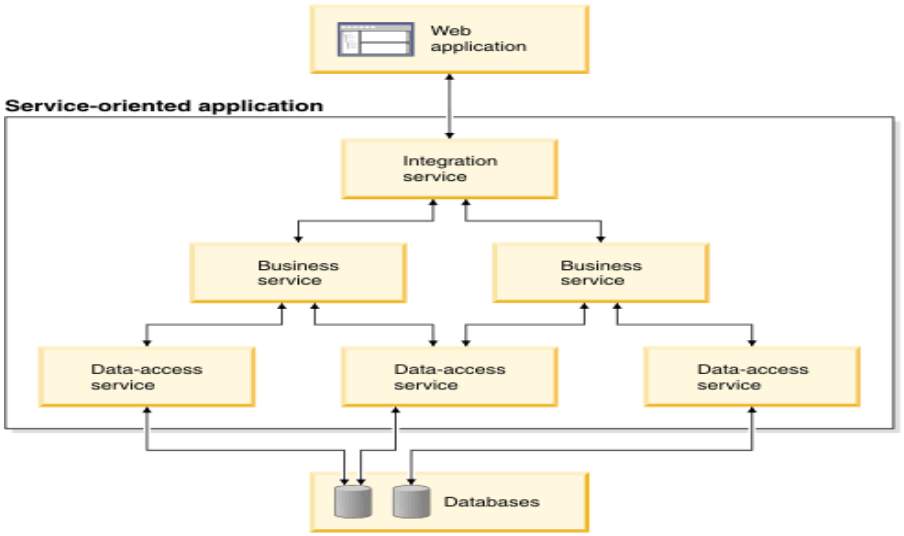
Nevýhodou monolitického prístupu je škálovateľnosť v priebehu času. Nastáva vysoká spojitosť a nízka súdržnosť (angl. tight coupling low cohesion) medzi vrstvami a strácame prehľad o závislosti vrstiev, preto sa časti kódu ťažko mieňa. [4]

3.2.2. 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 jednej biznis logiky, ako samostatná jednotka.
Výhodou SOA architektúry je ten, že aj keď to ostáva z istej časti monolitickým, každá služba je samostatne fungujúca jednotka. Vývojári nemusia rozumieť celej aplikácie, stačí ak vedia ako fungujú služby na ktorých pracujú. [5]. Príklad architektúry je obr.2 .

Podľa IBM knowledge center [5] má služba tieto charakteristiky:

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



Obrázok 2 Architektúra orientovaná na služby

3.3. 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 [6] naklonenie k MPA je výhodné ak:

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

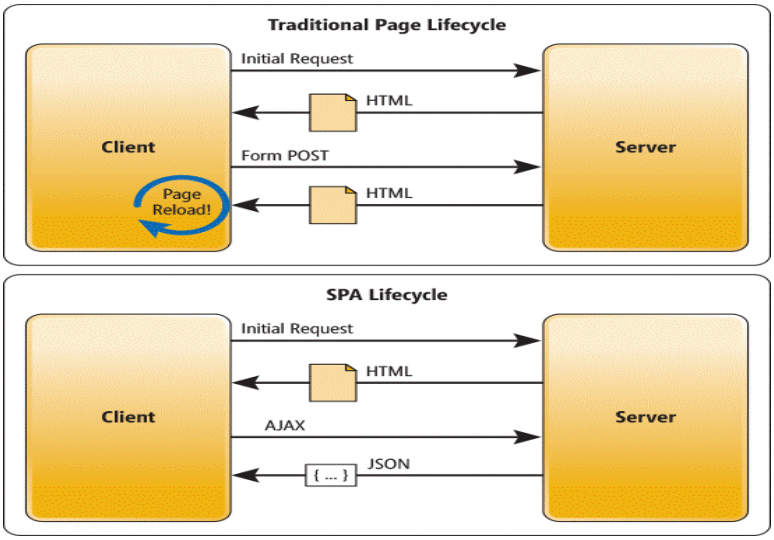
Tradičný webový prístup má aj svoje nedostatky. Je pomalý kvôli renderovanie celej HTML stránky na severy, čím sa zväčšuje šírka pásma pri prenose informácii cez sieť a znižuje sa spokojnosť užívateľa.

3.3.1. 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 prinášajú vysokú reaktivitu a lepší UX (angl. user experience). Prispôsobenie na mobilné zariadenie nastáva jednoduchšie, pretože server posiela už len údaje, nie celé stránky. Vystihuje sa hlavne komponent architektúrou, kde každý komponent je znovu použiteľný a má a presne definovanú, enkasulovanú funkcionalitu, kde interakciou užívateľa meníme ich zobrazenie. Gmail, Facebook, Github a mnoho ďalších aplikácii používa architektúru SPA. [7]

Ďalšou výhodou SPA je oddelený klient od servera, nasledovaním dizajnu hrubý klient, tenký server. Komunikácia so serverom prebieha pomocou REST rozhrania, posielaním si reprezentačného stavu objektu, ktorý reaguje na GET, PUT, POST, DELETE požiadavky od klienta, cez http protokol.

Použitím REST rozhrania získame rýchlejší prenos dát medzi klientom a serverom, pretože už neprenášame celé stránky, ale iba reprezentáciu dát v JSON či XML formáte. [8]
REST rozhraním získame možnosť posielanie http status kódov, pri ktorých môžeme správanie klienta prispôsobiť danému kódu. Viac v časti o bezpečnosti.



Obrázok 3 Rozdiel medzi MPA a SPA

Nevýhodou SPA je slabý SEO (angl. search engine optimalization). Google má problém spracovať javascript obsah stránok, preto indexovanie SPA aplikácie môže byť nižší ako MPA. [7]
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. XXS útokom (Cross-Site scripting). Viac v sekcii o bezpečnosti.

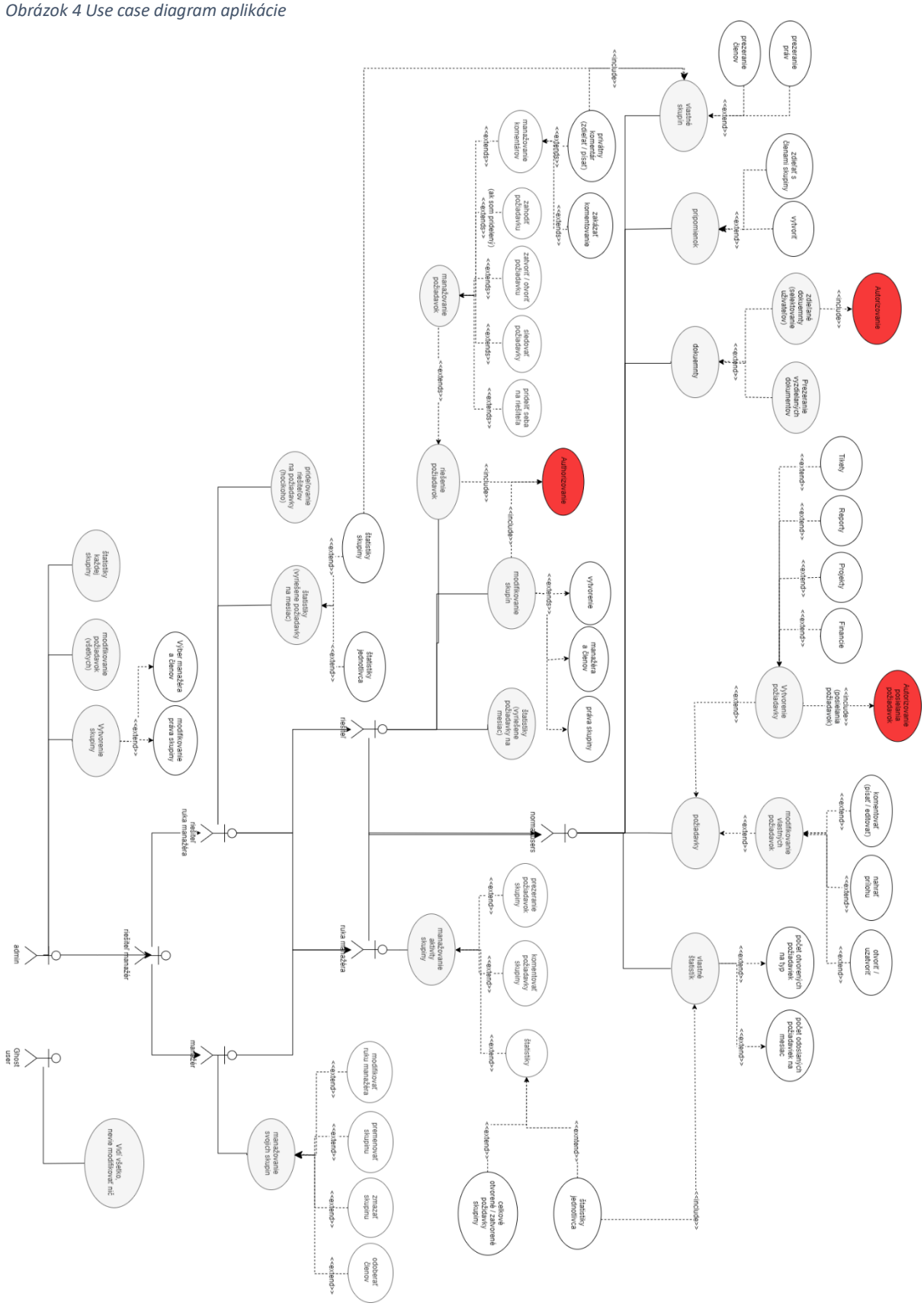
4. Návrh

V tento kapitole si predstavíme use case diagram aplikácie, ktorý zobrazuje všetky podstatné funkcionality systému a typy užívateľov, ktorí majú k týmto funkcionalitám prístup. Z predošlých kapitol si spravíme architektúru klienta a servera, zoznámime sa s technológiami, ktoré nám najlepšie poslúžia pre vybudovanie aplikácie, komponent diagramom znázorníme oddelené aspekty aplikácie a závislosť medzi nimi a v neposlednom rade si povieme o pravidlách návrhu užívateľského rozhrania

4.1. Use case diagram

Kvôli bezpečnosti aplikácie budeme mať viacero typov užívateľov, každý z nich pri autentifikácii obsiahne práva a role k použitiu istej časti aplikácie. Tieto informácie sa pri prvom prihlásení zahešujú do tokenu, nazývaným json web tokene (skr. JWT), ktorý sa pošle na stranu klienta. Klient si musí uchovať tento token na svojej strane a pri každej žiadosti o nový zdroj ho musí poslať, inak jeho žiadosť bude serverom odmietnutá. Token má ale svoju životnosť a musí sa žiadať o jeho obnovenie z bezpečnostných rizík o ktorých si povieme viac v časti bezpečnosť aplikácie.

Obrázok 4 Use case diagram aplikácie

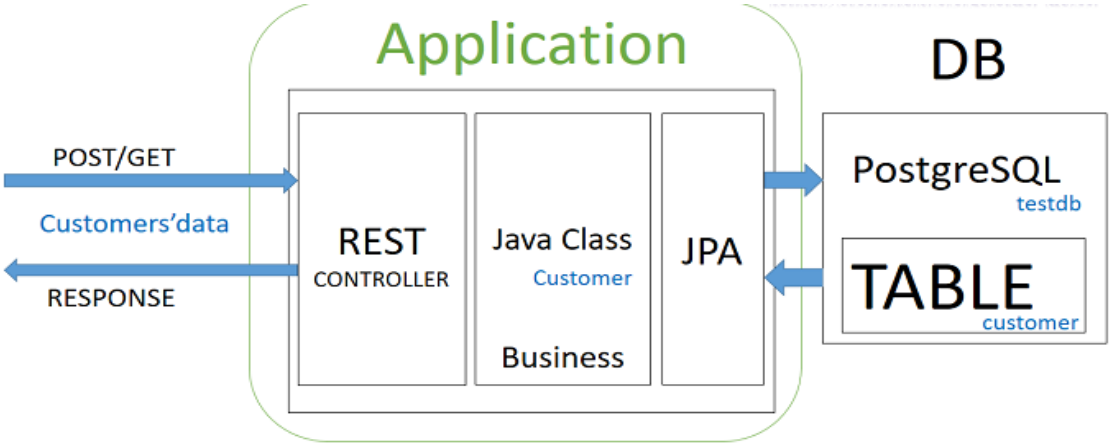


4.2. Návrh servera

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.

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 3.2.1. Na najvrchnejšej úrovni sa nachádzajú kontroléry, označením @RestController , ktoré počúvajú na CRUD požiadavky klienta. Všetky prichádzajúce žiadosti najprv narazia na hlavný kontrolér, nazývaným DispatcherServlet, ktorý má za úlohu delegovať prichádzajúce http požiadavky a procesy iným kontrolérom.

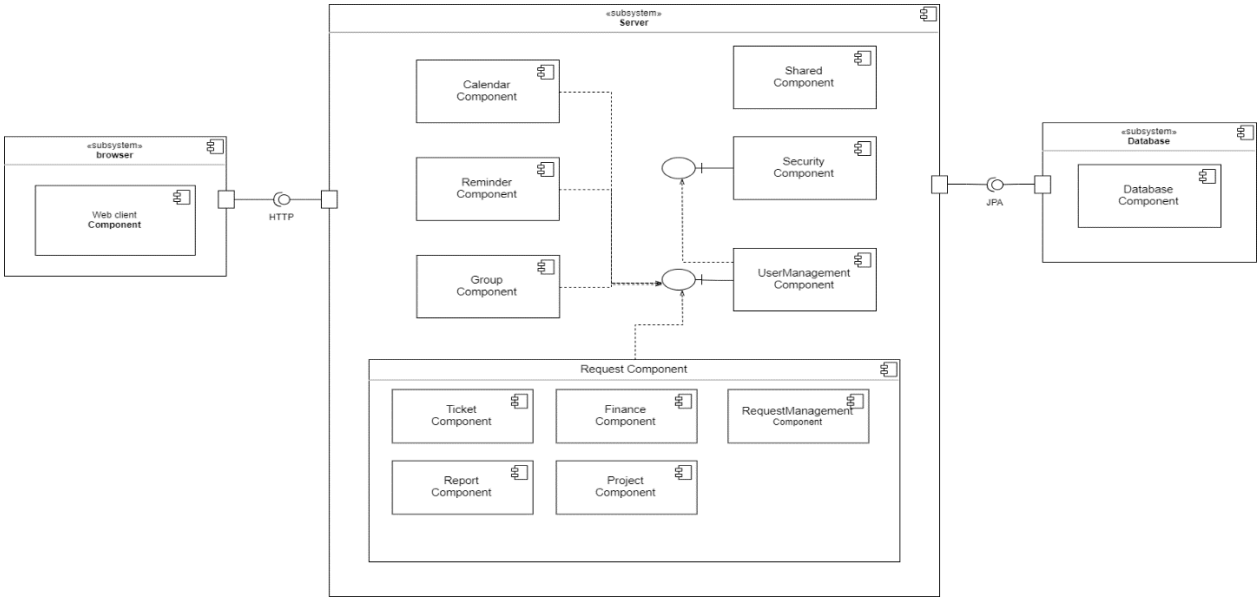
Strednú vrstvu tvorí biznis logika aplikácie označením @Service. Slúži na zložitejšie modifikovanie údajov, resp. vykonávanie transakcii. Vyznačujú sa vlastnosťou singletonu a pomocou injekcie závislosti (angl. dependency injection) cez označenie @Autowired sa posielajú medzi komponentami označenými @service, alebo @controller.
Nakoniec spodnú úroveň tvoria repozitáre, označením @Repository na získanie entít z databázy. [9]



Obrázok 5 Spring boot architektúra , zdroj : https://dzone.com/articles/spring-boot-with-embedded-postgresql-for-dao-integ

Integráciu viacerých rámcov do jedného systému nám spracováva projektový manažér Maven. Využíva koncept POM (angl. project object model), kde pridaním projektových závislostí, automaticky stiahne a pridá dodatočné jar súbory do cesty projektu. [10]
Objektovo relačné mapovanie a mapovanie vzťahov medzi databázovou entitou a objektovou entitou v jave nám poslúži rámec Hibernate, ktorý vieme ľahko integrovať do projektu pomocou maven-u. [11]
JPA (angl. java persistance API) je nadstavba na zjednodušenia práce s hibernatom. Je to implementovateľné rozhranie, ktoré sa stará o persistenciu dát v relačnej databáze. [12]

Vrstvovú architektúru projektu spring boot skombinujeme s architektúrou orientovanú na služby, odsek 3.2.2. Snažíme sa aplikáciu rozobrať na oddelené služby, inými slovami komponenty, tvoriacu jednu biznis logiku a mať nízku spojitosť medzi nimi.
Obrázok č. 6 predstavuje komponent diagram aplikácie. Na strane servera každý jeden komponent vnútorne nasleduje vrstvovú architektúru.

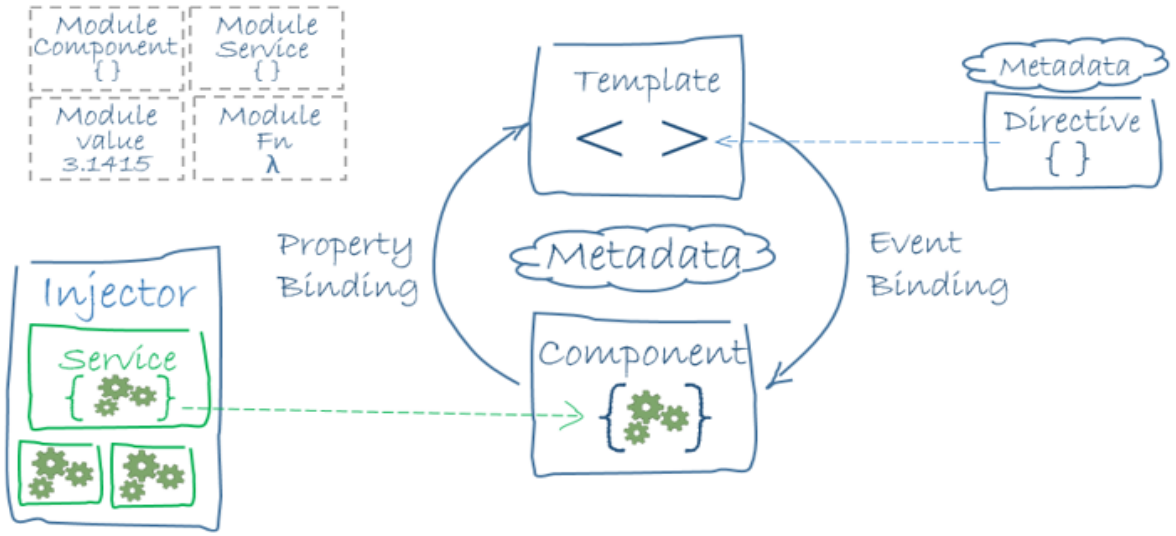


Obrázok 6 Komponent diagram aplikácie

4.3. Návrh Klienta

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

Štruktúra angularu sa skladá z komponentov, označenými dekorátorom @Component(). Každý z týchto komponentov obsahuje tvoriacu prezentačnú časť stránky (html), definíciu štýlov (css) a logiku komponentu reagujúcu na interakciu užívateľa (typescript).
Návrh komponentov by mal byť jednoduchý, aby sme ich vedeli viackrát použiť v iných častiach aplikácie.
Komunikáciu toku dát medzi komponentami nám napomáhajú služby označením @Injectable(), ktoré sa pridávajú ako závislosť injekciou do konštruktora komponentu.
Najzakalenejšou stavajúcou 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 importovať ďalšie NgModuly.
Navigáciu užívateľa zachytáva služba s názvom Router, ktorá skrýva a zobrazuje komponenty. [14]



Obrázok 7 Angular architecture [8]

Npm, Manažér balíka node (angl. node package manager) nám uľahčí prácu integrovaním externých knižníc na strane klienta [13].

Moderné reaktívne programovanie dosiahneme knižnicou Rxjs (angl. Reactive extension for Javascript), ktorá sa stavia k všetkému ako k toku údajov. Využíva myšlienku pozorovateľov (angl. observable), na ktorý sa môžeme zavesiť (angl. subscribe) a využiť koncepciu asynchrónneho programovania. [14]
Viac v časti optimalizácia aplikácie.

4.3.1. Návrh používateľského rozhrania

Návrh užívateľského rozhrania, ďalej už len UX, je často mnohonásobne komplikovanejší ako sa na prvý pohľad zdá. Krásny obaľ môže zakryť nedostatky, resp. nedotiahnutý dizajn odradiť návštevníka.

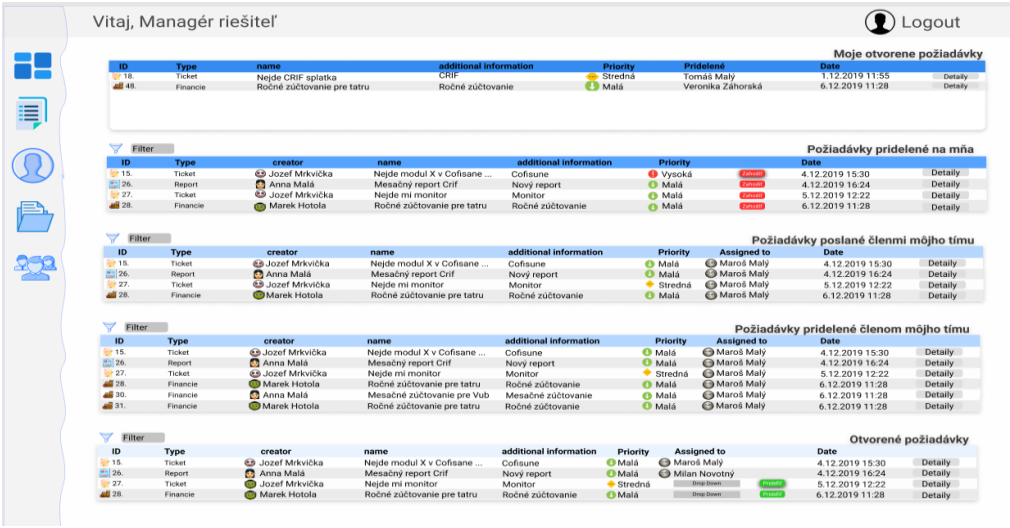
Kniha The Non-Designer's Design Book [15] opisuje jednoduché pravidlá návrhu UX:

- 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 tieto zásady sme sa rozhodli podstatné informácie užívateľovi vyvesiť na hlavnú stránku (angl. dashboard), ktorá bude tvorená hlavne stručnými detailmi požiadaviek rozdelených do kategórii tabuliek podľa typu prehláseného užívateľa.

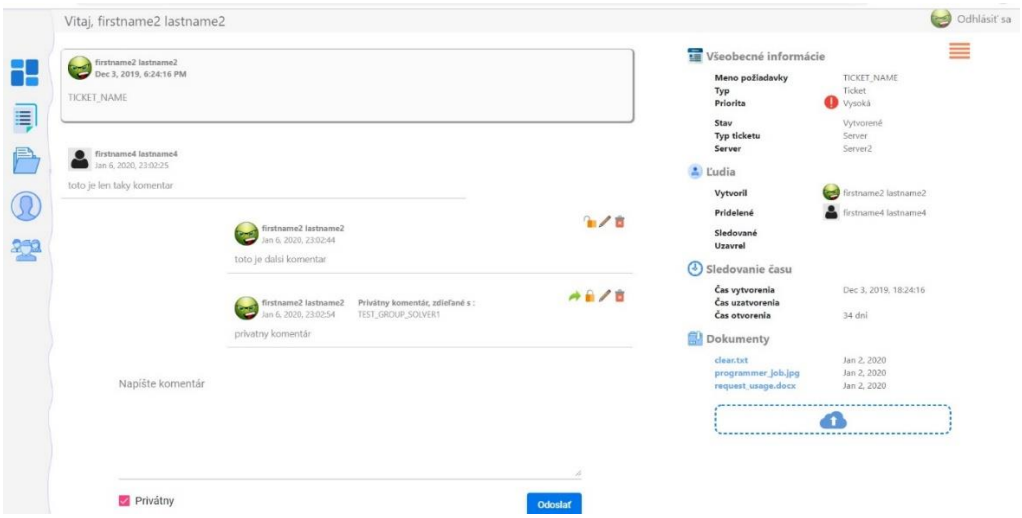
Bežným užívateľom sa zobrazí len jedna tabuľky s informáciami požiadaviek, ktoré oni sami vytvorili. V prípade prihlásenia manažéra skupiny chceme rozdelene zobraziť požiadavky, ktoré jednak boli odoslané členmi jeho tímu a, ktoré sú pridelené na nich.
Riešiteľovi chceme pridať ďalšiu oddelenú tabuľku so separovanými požiadavky, ktoré sú pridelené len na neho.

V neposlednom rade všetky typy požiadaviek zaslané do systému, ktoré korešpondujúci s právami užívateľa na riešenie sa zobrazia v tabuľke „otvorené požiadavky“.
Požiadavky nachádzajúce sa v prvých štyroch tabuľkách (obr. 8 Dashboard) budú automaticky v stave sledované, to znamená, že každá malá zmena nasleduje notifikáciu užívateľa.



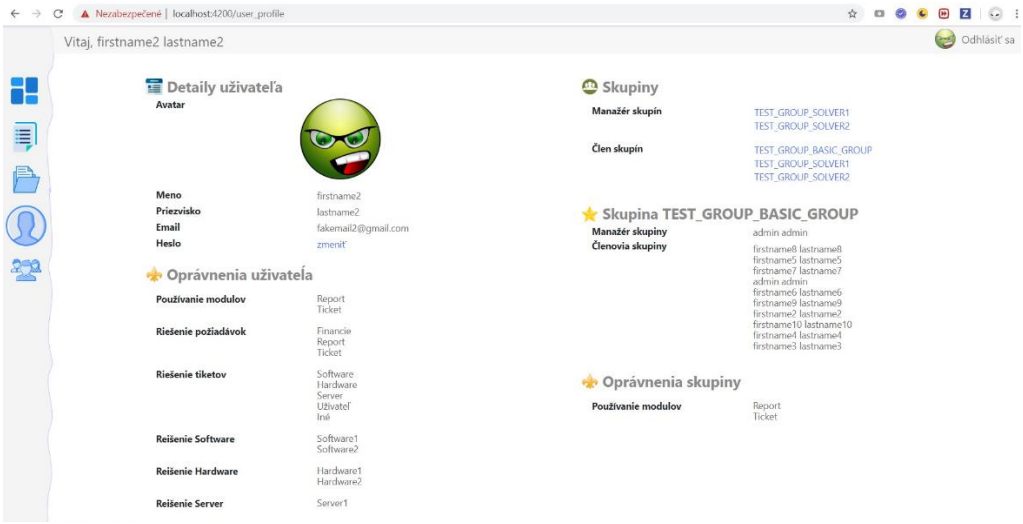
Obrázok 8 Dashboard

Kliknutím na detaily sa zobrazia podrobné informácie problému. Dodatočné informácie sa zobrazia na základe druhu požiadavky. Užívateľ bude mať možnosť nahrať dokument so zaznamenaním času a taktiež písať verejné, či súkromné komentáre, ktoré môže zdieľať s členmi svojej skupiny (obr. 9 , Detaily požiadavky).



Obrázok 9 Detaily požiadavky

Prístupové práva a informácie užívateľa sa nachádzajú na obrazovke profil (obr. 10). Oprávnenia v aplikácii sa určujú iba skupine, nie samotnému užívateľovi. Užívateľ nabodáva právomoc začlenením do skupiny. Profil zobrazujú zoskupenie oprávnení užívateľa zo skupín, skupiny, do ktorých je užívateľ priradení, členov danej skupiny a taktiež práva jednotlivých skupín.



Obrázok 10 Profil užívateľa

Angular nám svojou komponentnou štruktúru uľahčí prácu. Stránka dashboard môže byť jeden veľký komponent, ktorý udržuje v sebe viaceré malé komponenty, napr. tabuľku na zobrazenie detailov požiadaviek. Nemusíme kopírovať ten istý HTML kód na viacero miest, ale vytvoríme si jeden komponent, ktorý viackrát používame. Rovnakú stratégiu využijeme pri písaní komentárov. Držíme sa prístupu DRY (don't repeat yourself).

Zdroje

1. Jira https://www.atlassian.com/software/jira https://www.sitepen.com/blog/clientserver-model-on-the-web/
2. OsTicket https://osticket.com/
3. Monolithic application https://docs.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications
4. Ibm introduction to SOA https://www.ibm.com/support/knowledgecenter/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html
5. MPA vs SPA https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps
6. Single page application https://javatutorial.net/how-to-build-single-page-application-with-java-ee-and-angular
7. Angular university https://blog.angular-university.io/
8. Spring boot https://spring.io/projects/spring-boot
9. What is maven? https://maven.apache.org/what-is-maven.html
10. Hibernate https://www.tutorialspoint.com/hibernate/index.htm
11. Java Persistance wikibooks https://upload.wikimedia.org/wikipedia/commons/8/81/Java_Persistence.pdf
12. What is npm https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/
13. Angular official site https://angular.io/
14. Robin Williams, The Non-Designer's Design Book https://diegopiovesan.files.wordpress.com/2010/07/livro_-_the_non-designers_desi.pdf