VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

CLOUDOVÉ APLIKACE S JAVASCRIPTOVÝM KLIENTEM

CLOUD APPLICATIONS WITH JAVASCRIPT CLIENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **LUKÁŠ HUDEC**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19906/2016/xhudec26

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Hudec Lukáš**
Obor: Informační technologie
Téma: **Cloudové aplikace s JavaScriptovým klientem**
Cloud Applications with a JavaScript Client
Kategorie: Informační systémy

Pokyny:

1. Seznamte se dostupnými cloudovými databázemi. Zaměřte se na službu Amazon S3.
2. Prostudujte současné technologie pro tvorbu aplikací s tlustým klientem v jazyce JavaScript, jako např. frameworky React, Redux a další.
3. Po dohodě s vedoucím zvolte vhodnou kombinaci technologií a navrhněte aplikaci demonstrující využití těchto technologií v praxi.
4. Navrženou aplikaci implementujte na zvolené platformě.
5. Proveďte testování aplikace.
6. Zhodnoťe klady a zápory cloudové technologie a dosažené výsledky.

Literatura:

- Ondřej Žára: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015
- James Murty: Programming Amazon Web Services, O´REILLY, 2008

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Burget Radek, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L. S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práce zkoumá současné technologie pro tvorbu aplikací v cloudu s tlustým klientem v jazyce JavaScript. Teoretická část tyto technologie popisuje a porovnává. Praktická část se zabývá návrhem a implementací demonstrační aplikace, která využívá zvolené technologie.

Abstract

Thesis contains a research of current technologies for developing cloud applications with a JavaScript client. Its theoretical part describes and compares these technologies. Practical part is focused on design and implementation of a sample application, which is using the related technologies.

Klíčová slova

cloudové aplikace, Amazon Web Services, AWS, jednostránkové aplikace, SPA, JavaScript, ES6, node, react, redux, immutable, virtuální DOM, JSX, webové technologie

Keywords

cloud applications, Amazon Web Services, AWS, single page applications, SPA, JavaScript, ES6, node, react, redux, immutable, virtual DOM, JSX, web technologies

Citace

HUDEC, Lukáš. *Cloudové aplikace s JavaScriptovým klientem.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Burget Radek.

Cloudové aplikace s JavaScriptovým klientem

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta Ph.D. a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Hudec
15. května 2017

Poděkování

Zde bych chtěl poděkovat svému vedoucímu panu doktorovi Burgetovi za odborné vedení a cenné rady při tvorbě této práce. Také bych chtěl poděkovat panu inženýrovi Malému za cenné rady ohledně návrhu aplikace a lidem, kteří se podíleli na testování aplikace.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce je prostudovat oblast cloudových služeb a zaměřit se i na skupinu front-endových technologií pro vývoj moderních webových aplikací. Práce představí přední poskytovatele cloudových služeb a poskytne základní přehled o nejpoužívanějších knihovnách a frameworcích pro tvorbu webových uživatelských rozhraní. Na základě těchto zvolených technologií poté bude implementována aplikace, která ukáže, jak jednotlivé technologie do sebe zapadají.

Kapitola 2 vysvětlí pojmy *Cloud computing* (2.1), *single page application* (2.2) a seznámí nás s technologiemi cloudových aplikací, vyčlení typy cloudových služeb a představí technologie pro vývoj front-endů webových aplikací (2.3). Následně kapitola srovná představené technologie v sekci 2.4.

Kapitola 3 popíše zadání aplikace, rozebere návrh databázového modelu a stavového objektu klientské aplikace. Dále vysvětlí, jak bude probíhat komunikace klientské aplikace se serverovou aplikací uloženou v cloudu.

Kapitola 4 popíše nejdůležitější poznatky získané při vývoji serverové 4.1 i klientské 4.2 části. Sekce o serverové části důkladně popíše, jak správně vytvořit a nastavit jednotlivé instance služeb na Amazon Web Services pro databázi, úložiště a výpočetní sílu. Klientská část například popíše integraci se serverovou částí 4.2.3 a nasazování klientské produkční verze na AWS S3 4.2.4.

Kapitola 5 vysvětlí, jak probíhalo testování aplikace během vývoje a jak probíhalo koncové testování aplikace.

V závěru této práce je shrnutí studie o technologiích a implementace demonstrační aplikace. Závěr dále obsahuje popis dalších možných budoucích rozšíření aplikace.

Kapitola 2

Technologie

V dnešní době u velkého množství startup projektů žádná firma dopředu neví, v jakém měřítku se bude jejich aplikace používat. Tedy například neví, jaké hardwarové nároky bude muset splňovat server. Tyto problémy ohledně škálovatelnosti aplikace řeší cloudové služby, které mohou být jednoduše modifikovány a nemusí se řešit rozšiřování výpočetní síly serveru a následné překonfigurace a mnoho dalších věcí s tím spojených.

Firma díky tomu ušetří spoustu financí, protože k vybudování kvalitní infrastruktury je potřeba tým specialistů a mnoho komponent, realizujících samotnou vlastní architekturu. Místo toho se může zaměřit přímo na věci, kterým skutečně rozumí a řešení či údržbu systémové architektury nechá na cloudových službách.

Hlavními poskytovateli cloudových služeb na trhu jsou korporace Amazon, Google, Apple a Microsoft.

Tato kapitola přiblíží pojmy jako *cloud computing*, *single page application* a předá základní přehled o technologiích, které umožňují vyvinout moderní multiplatformní cloudovou webovou aplikaci.

2.1 Cloud Computing

Cloud computing charakterizuje poskytování služeb a programů servery přístupnými z internetu a platí, že uživatelé k nim mohou přistupovat vzdáleně.

Poskytované služby cloudu se dělí na 11 základních typů [10]:

- Storage as a Service
- Database as a Service
- Information as a Service
- Process as a Service
- Application as a Service nebo také Software as a Service
- Platform as a Service
- Integration as a Service
- Security as a Service

- Management / Governance as a Service
- Testing as a Service
- Infrastructure as a Service

Nejtypičtějšími poskytovanými službami jsou však přímo balíčky *Software as a Service*, *Platform as a Service* a *Infrastructure as a Service*, které již obsahují dílčí typy cloudových služeb [2].

Software as a Service (SaaS) software jako služba, tedy poskytování softwarové aplikace prostřednictvím internetu tak, že aplikace běží na serverech poskytovalete služby. Klient se nemusí zabývat instalací, správou ani údržbou této aplikace.

Platform as a Service (PaaS) platforma jako služba: poskytování výpočetní a softwarové infrastruktury formou služby. Součástí je nejen samotný hardware, ale také tzv. *Solution stack*, tedy software potřebný k provozu vlastních aplikací. Zpravidla zahrnuje operační systém a softwarový ekosystém dle potřeby - např. webový server, databázový server apod. Klient se nemusí zabývat provozem platformy, ale řeší pouze instalaci, provoz a údržbu své aplikace.

Infrastructure as a Service (IaaS) infrastruktura jako služba: poskytování výpočetní infrastruktury (typicky virtuálního stroje s odpovídajícím úložným prostrorem a síťovou konektivitou) formou služby. Klient se nemusí starat o údržbu a provoz hardwaru.

2.1.1 Amazon Web Services

Amazon nabízí širokou škálu cloudově založených produktů jako jsou výpočetní infrastruktura, úložiště, databáze, sítě, zabezpečení a mnoho dalších. Je zaměřena převážně na služby typu *Infrastructure as a Service*.

Každá tato služba má určité cenové úrovně závislé na mnoha faktorech. Jedním z těchto faktorů je například přidělený hardware virtuálního stroje.

AWS nabízí i tyto služby v testovacím režimu, tedy zadarmo. Zde se však nacházejí určitá omezení např. co se týče počtu dotazů, rychlosti zpracování a jsou zpravidla omezeny na 1 rok fungování. Pokud chceme některou službu zrychlit či rozšířit musíme si samozřejmě připlatit.

2.1.2 Google Cloud Platform

Firma Google je dalším známým poskytovatelem *cloud computing* 2.1 služeb. Poskytuje kupříkladu *SaaS* služby jako Gmail, Google Drive, Google Docs a mnoho dalších. Kupříkladu také nabízí stejně jako Amazon *IaaS* služby pro úložiště, databáze atd.

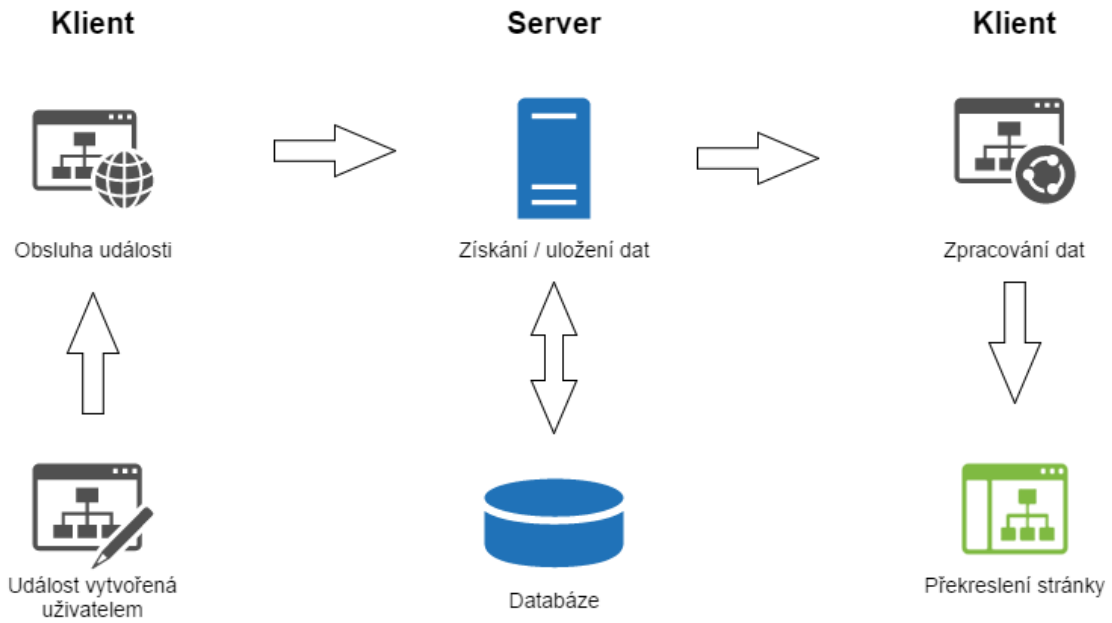
2.2 Single page application (SPA)

Webové aplikace vždy dodržovaly pravidla pro rozmístění funkcionality do mnoha stránek. Každá stránka měla svůj úkol a odpovídala určitému stavu či události. Výhoda tohoto přístupu je, že každá stránka měla zvlášť svou logiku a pevně danou adresu. Hlavní nevýhodou však byl zbytečný přenos velkého objemu dat, který se opakovaně posílal při přechodu mezi jednotlivými stránkami.

S nástupem *Asynchronous JavaScript and XML* (AJAX) a vývojem moderních webových prohlížečů se na scéně objevil nový model webových aplikací - jednostránkové aplikace, známe spíš pod pojmem *Single Page Application* či *One Page Application*. Tento model řeší právě problém posílání zbytečně velkých kvant dat a zrychlení interakce uživatele s aplikací. Z názvu tedy jasně plyne, že veškerá funkcionalita aplikace je umístěna na jedné stránce. S příchodem nových JavaScriptových API a frameworků, SPA nabývají stále větší popularity při tvorbě a návrhu webové aplikace.

Hlavní výhodou tohoto přístupu je rychlá odezva. Uživatele už zejména neznepokojuje tzv. *click-and-wait*, kdy při přechodu na jinou stránku, se musela kompletně celá stránka znovu vykreslit. Další výhodou je, že server je ve většině případů využit pouze jako zdroj a úložiště dat, není tolik vytížen, protože veškerá klientská logika se odehrává na jednostránkové aplikaci, viz obrázek 2.1. Nemusí se psát kód pro datovou logiku, zvlášť na serveru a zvlášť na klientovi. Všechna logika klienta je pokrytá JavaScriptem, potřebujeme-li například získat data z databáze, odešleme AJAXový dotaz na server. A po získání dat se překreslí pouze ta část stránky, kde nastala změna, což je velkým přínosem, co se týče odezvy.

Zásadní nevýhodou tohoto modelu je, že aplikace nemůže fungovat, pokud je v prohlížeči vypnutý JavaScript. SPA jsou také hůře zpracovatelné pro roboty vyhledávačů a analytické nástroje, protože vše běží pouze na jedné stránce. Musí se tedy vytvořit fiktivní URL adresy simulující stránky. Tímto se dá vyřešit i problém s historií a funkcionalitou tlačítka zpět v prohlížeči.



Obrázek 2.1: Komunikace SPA se serverem

2.3 Front-endové technologie

JavaScript, převážně známý jako front-endový jazyk pro webové aplikace v současnosti nabývá na popularitě. Dříve byl považován jako nástroj pro oživení webu díky animacím, ale stává se z něj plnohodnotný skriptovací jazyk a to nejen co se týče webů, kde konkuruje

PHP, ale i v oblasti back-endu (Node.js), kde konkuruje tradičním back-endovým jazykům jako je například Java.
Komunita okolo JavaScriptu neustále roste a s tím souvisí neustálý posun dopředu, vývoj nových technologií a přístupů. Navíc díky výkonnému *V8 JavaScript Enginu* využitý např. v Node.js či v Google Chrome jsou dnešní JavaScriptové aplikace skutečně rychlé.
Vývojář má tedy k dispozici početnou škálu frameworků, knihoven a nástrojů. V této práci se však zaměříme pouze na nejznámější či nejpoužívanější zástupce těchto technologií.

2.3.1 Angular

AngularJS známý jako Angular 1 byl vyvinut v roce 2010 firmou Google jako plnohodnotný *Model-View-Controller (MVC)* framework. Je tedy navržen tak, aby od sebe odstínil vrstvy vykreslovací a aplikační logiky. Navzdory velké oblíbenosti po roce 2010, dnes začíná upadat pod tlakem nových a modernějších frameworků a knihoven.
Jeho následovníkem je Angular 2, který byl vydán opět pod křídly Google v roce 2016. A je chápan jako reakce Google na React, kterým se velmi inspiroval. Přináší velké změny oproti jeho první verzi. Angular 1 a 2 jsou tedy nekompatibilní. Migrace z jednoho na druhý vesměs znamená přepsat téměř celou aplikaci.

2.3.2 React

React 2.5.3 opensourcovala firma Facebook v květnu 2013, která jej už řadu let před tím vyvíjela a užívala. Po vydání se však React dočkal velkého zklamání. Kritizovalo se hlavně míchání HTML a programování. Postupem času se naopak ukázalo, že došlo k nepochopení základního konceptu, kterým React přispěl. Dnes je jedním z nejoblíbenějších a nejaktivnějších repozitářů na GitHubu, má přes 62800 hvězd a přes 8300 commitů.
Zároveň s Reactem je vyvíjen i React Native, kde jak už název napovídá, se bude jednat o nativní aplikace. Účelem React Native je vyvíjet aplikace jednotným multiplatformním kódem. To přináší obrovské výhody:

- až 90% kódu aplikace je sdíleno mezi aplikacemi pro jednotlivé platformy
- téměř celá aplikace je napsaná v JavaScriptu, v týmu tedy nemusí být vývojáři Javy pro Android nebo vývojový tým zaměřený na Objective-C / Swift pro iOS

2.3.3 VueJS

VueJS byl zveřejněn v únoru roku 2014. Byl vyvinut Evanem You, bývalým core-developerem Meteoru (což je další JavaScriptový MVC framework). I v tomto případě se jedná o knihovnu, pro tvorbu webového rozhraní pomocí znovupoužitelných komponent.

2.3.4 Redux

Autorem této knihovny je Dan Abramov, současný React core-developer Facebooku. Konkrétně mluvíme o predikovatelném stavovém kontejneru pro JavaScriptové aplikace. Využívá prvky funkcionálního programování jako je kompozice, *high order functions*, *pure functions*, *immutable*, *currying*, parciální aplikace aj. díky nimž se aplikace chová deterministicky a je jednoduše testovatelná.

2.3.5 MobX

MobX je další velmi používanou knihovnou pro správu stavu aplikace. Filozofie MobXu se dá shrnout do věty: „Vše co může být odvozeno ze stavu aplikace, by mělo být automaticky odvozeno.“ [1] K těmto účelům aplikuje *funkcionální reaktivní programování (FRP)* a návrhový vzor *Observer*.

2.3.6 The Reactive Extensions for JavaScript (RxJS)

RxJS je považován za soubor knihoven pro asynchronní a událostmi řízené aplikace využívající observable sekvence. Rozšiřuje návrhový vzor *Observer* o podporu sekvencí dat a událostí a přidává operátory, které umožňují deklarativní kompozici těchto sekvencí dohromady, kdy zároveň abstrahujeme problémy jako nízkoúrovňová vlákna, synchronizace, konkurentní datové struktury a neblokující vstupy a výstupy.

2.4 Srovnání

Tato část kapitoly je zaměřena na porovnání jednotlivých technologií, které si konkurují. Proto tedy důkladněji vysvětluje rozdíly mezi poskytovateli cloudových služeb, konkrétně Amazon a jeho Amazon Web Services a Google Cloud, zaměří se jak na technologické kapacity a možnosti, tak na finanční politiku těchto poskytovatelů. Dále srovná knihovny či frameworky pro tvorbu uživatelských rozhraní a knihovny pro správu stavu aplikace a dat.

2.4.1 Amazon Web Services vs Google Cloud

Nejdříve je potřeba zmínit, že co se týče poskytování cloudových služeb, tak Amazon byl v této oblasti jedním z prvních průkopníků této technologie. A proto má v mnoha věcech oproti konkurenci navrch.
Donedávna platilo, že největším rozdílem mezi Amazon Web Services a Google Cloud bylo množství poskytovaných služeb. To už dnes úplně neplatí. Google velmi zapracoval na nabídce služeb a i jeho Google Cloud dnes nabízí širokou škálu možností.
Dnes je hlavním rozdílem pokrytí. Zde je Amazon jasným vítězem, jeho datová centra se vyskytují téměř na všech kontinentech. Google má oproti AWS pokrytou hlavně Severní Ameriku.
Dalším důležitým rozdílem je cena. Google je výrazně levnější. Například rozdíl v cenách pro computing engine. Oba jsou placeny *pay-per-use*, tedy za každé užití, ovšem s podstatným rozdílem:

- *Elastic Compute Cloud (EC2)* od Amazonu zaokrouhluje veškeré použití na minimálně jednu hodinu. Použije-li se virtuální stroj pro výpočet například na 15 minut, musíme zaplatit za celou hodinu této služby.
- *Google Compute Engine (GCE)* je cenově daleko vstřícnější a nastavuje minimální sazbu 10 minut, s tím že každá další minuta je naceňována samostatně.

Google ale převyšuje Amazon v oblasti *Big Data*. Zde není divu, aby firma, která je zaměřená na práci s velkými kvanty dat, nezužitkovala své dlouhodobě nabyté zkušenosti. Jeho služba *BigQuery* dokáže zanalyzovat obrovské množství dat za velmi krátký čas.

[1] MobX dokumentace https://github.com/mobxjs/mobx#introduction

Jak Amazon tak i Google nabízí slevy na určité balíčky služeb. Zde ovšem záleží, jaký typ uživatele jsme. Pokud máme aplikaci, která bude konstantně využívat služby *cloud computing* 2.1, ať už na Google Cloud či Amazon Web Services, rozhodně musíme počítat s tím, že si připlatíme.
Zvolit tedy, který z těchto kandidátů je lepší nebo horší, je velmi těžké. Obě technologie disponují výbornou infrastrukturou a širokým množstvím služeb, které lze jednoduše integrovat mezi sebou. Záleží tedy, k čemu tyto služby budeme využívat a jaké máme zdroje. [4]

2.4.2 Angular vs React vs VueJS

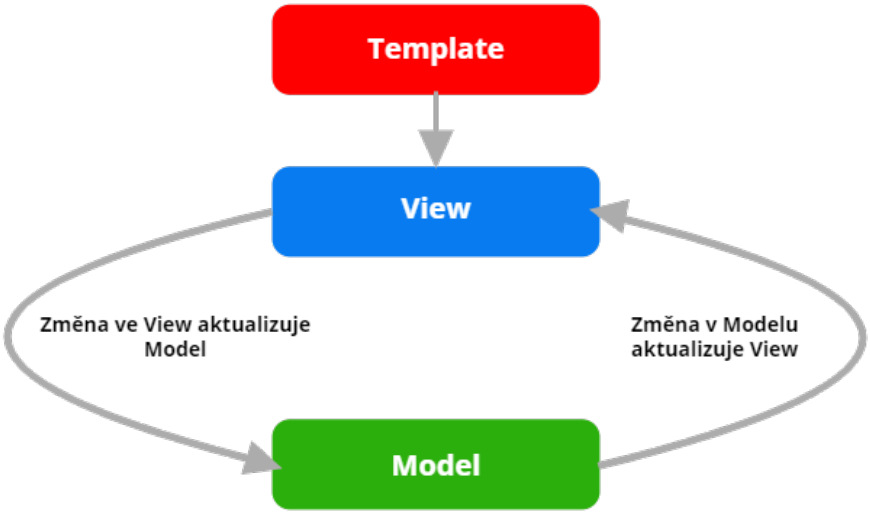
Nyní se zaměříme na knihovny a frameworky, které jsou orientovány na front-end. Konkrétně na tvorbu uživatelských rozhraní v JavaScriptu. Porovnáme tedy obě verze Angularu, React a VueJS.

AngularJS

Je jedním z nejpoužívanějších plnohodnotných MVC frameworků pro JavaScript. Obohacuje HTML značky o další vlastnosti s cílem tvořit dynamické prvky. Hlavními přednostmi tohoto frameworku jsou:

- *Two way data-binding*, což je koncept, který řeší synchronizaci mezi Modelem a vrstvou View. Jde tedy o deklarativní přístup, kdy programátor nemusí imperativně řešit synchronizace. Místo toho se pošlou data do Modelu a Angular je synchronizuje s View viz obrázek 2.2.
- *Dependency injection*. Obecně se jedná o návrhový vzor, který řeší závislosti mezi jednotlivými moduly v programu. Angular obsahuje subsystém, který umožňuje vždy přesně specifikovat, které jiné komponenty jsou uvnitř konkrétní komponenty používány.
- Testovatelnost je jedna z oblastí, na kterou je framework zaměřen. Přispívá k tomu i implementace Dependency injection. Což velmi usnadňuje unit testing, díky tomu, že nemusíme nahrazovat či obcházet spousty dalších závislostí.
- Jako první představuje koncept vlastních komponent.

Dnešní nevýhodou tohoto frameworku je, že se už považuje za zastaralý, a používá se spíše pro údržbu již vyvinutých aplikací. Nové aplikace se již ve většině vyvíjejí v modernějších frameworcích. Další nevýhodou může být rozsáhlost tohoto frameworku a s tím související délka učící křivky. Pakliže jej programátor dobře ovládá, má v rukou mocný nástroj pro vývoj.



Obrázek 2.2: Two way data-binding

React

Mnohdy se mylně pokládá za plnohodnotný framework, jedná se spíše o knihovnu pro tvorbu uživatelských rozhraní. Často se také uvádí, že v pomyslném MVC modelu představuje vrstvu View. Což také úplně není pravda. Komponenty obsahují i jistou řídící logiku, tedy mohou představovat i vrstvu Controller v kontextu MVC modelu.
React je založen hlavně na zapouzdřených a znovupoužitelných komponentách, které si řídí svůj vlastní stav. Tyto jednoduché komponenty můžeme poté dále skládat do dalších a složitějších celků.
Komponenty jsou uspořádány ve stromové struktuře. Dojde-li ke změně dat v některé z komponent, změna se aplikuje pouze na daný prvek či podstrom. Z toho vyplývá, že se zbytečně nepřekresluje celá stránka.
Tento koncept rozhodně převyšuje AngularJS, pokud naše aplikace pracuje s velkým množstvím dat, která následně vykresluje. Jakákoliv změna ovlivní pouze tu komponentu, ve které ke změně došlo.
Oproti Angularu je tedy hlavně zaměřen na vykreslování. Používá tzv. virtuální DOM (2.5.3) pro vnitřní zpracování a vyhodnocování, kterou komponentu je nutno překreslit. Programátor je tedy v podstatě úplně odstíněn od DOMu, jehož implementace se může na mnoha prohlížečích lišit.
Stejně jako u Angularu píšeme z podstatně větší části deklarativní kód.
Další výhodou je, pokud umíme React, tak z velké části umíme i React Native, což je knihovna pro tvorbu uživatelských rozhraní pro různé mobilní platformy (Android, iOS).
Je doporučeno jej používat s ECMAScript 2015 (ES6) (2.5.2), který obohacuje JavaScript o další užitečnou funkcionalitu.

Angular 2

Angular 2 je inspirovaný Reactem, co se týče rychlosti vykreslování. Opravil chyby jeho první verze jako je např. tzv. *dirty-checking*, která způsobovala, že AngularJS byl pomalý.

U *dirty checking* se totiž pokaždé kontrolují hodnoty scope u komponenty, zda se nezměnila data, což značně zpomaluje běh aplikace.
Velkou změnou je, že je určen pro *TypeScript*, což je rozšíření JavaScriptu o statické typování a prvky objektově orientovaného programování. Cílem by mělo být méně chyb způsobených prací se špatnými datovými typy. Statické typování tomu ale úplně nezabrání. Výsledkem je, že kód obsahuje příliš tzv. *boilerplate kódu*, tedy mnoho vyžadovaného kódu pro implementaci malé funkcionality. Statická kontrola navíc výrazně zpomaluje dobu pro zkompilování, což pocítíme hlavně u větších aplikací.

VueJS

Je dalším MVC frameworkem, který se primárně soustředí na zobrazování a je založený na komponentách. VueJS vznikla inspirací z mnoha existujících frameworků či knihoven a aplikovala jejich nejlepší praktiky. Principy shodné s Reactem:

- Stejně jako React poskytuje reaktivní komponenty, které můžeme skládat do složitějších celků.
- Využívá virtuální DOM.
- Zaměřuje se hlavně na vykreslování. Směrování a management stavu nechává na jiných knihovnách.

Principy odvozené od Angularu:

- Velmi podobná syntaxe (v-if, ng-if)
- Užívá direktivy, ale má čistější oddělení od komponent. Direktivy jsou zaměřeny jen a pouze na manipulaci DOMu. V Angularu je okolo direktiv a komponent spousta nejasností.
- Na rozdíl od Angularu, který má *Two way data-binding*, VueJS vás striktně nutí používat jednosměrný proud dat (stejně jako React).

Největší výhodou VueJS, kterou převyšuje výše zmíněné knihovny, je rychlost.

2.4.3 Redux vs MobX vs RxJS

Stejně jako v předchozí sekci se zaměříme na JavaScriptové knihovny. Tentokrát se podrobněji podíváme na ty, které jsou orientované na management dat a stavu aplikace.
Jak Redux, tak MobX jsou knihovny, které mohou být použity s AngularJS, React či VueJS, ale nejvíce zapadají do filozofie Reactu.
Redux je podmnožinou *Flux* architektury. Narozdíl od ní, používá pouze jeden *Store* (2.8) místo mnoha k uchování stavu aplikace. Opírá se o principy funkcionálního programování. Užívá *pure funkcí*, což jsou funkce, které pro daný vstup vrátí vždy ten samý výstup a neovlivňují nijak okolí - nezpůsobují vedlejší účinky [6].
Stav celé aplikace je *immutable*. Pokud dojde ke změně stavu, vytvoří se kompletně nový objekt reprezentující stav a nenastane mutace žádné položky. Vždy se vytvoří nový stav, což otevírá možnosti jako tzv. *time-travel debugging* (4.3.12).
Oproti Reduxu, MobX vychází spíše z objektově orientovaného programování, ale má i prvky reaktivního programování. Obaluje stav aplikace do observable entit. Máme tedy veškerou

sílu *Observable* ve stavu naší aplikace. Stav aplikace je *mutable*, tedy při změnách dochází k mutaci stavu a ne k vytváření vždy nového stavu jako to bylo u Reduxu. Dále velkým rozdílem oproti Reduxu je fakt, že můžeme mít více *Store* objektů. Výhodou oproti Reduxu je, že nepíšeme tolik boilerplate kódu.

Obrázek 2.3: MobX Flow

RxJS funguje zcela jinak než Redux či MobX. Redux a MobX jsou zaměřeny na hodnoty, tedy na data. RxJS se zabývá událostmi. V *Reactive Extensions* uvažujeme o událostech jako o poli, nad kterým můžeme provádět různé operace jako je mapování, filtrování nebo dokonce můžeme jednotlivé události opožďovat [9]. Společné s MobX mají pouze to, že vycházejí z reaktivního programování a používají návrhový vzor *Observable*.

2.4.4 Verdikt

Co se týče cloudových služeb, zvolil jsem pro vývoj Amazon Web Services a to hlavně z důvodu praktických zkušeností s jeho používáním.
Pro účely vývoje demonstrační aplikace jsem se rozhodl použít knihovnu React na tvorbu uživatelských rozhraní webové aplikace. Management stavu aplikace jsem se rozhodl řešit Reduxem. V obou případech na základě již nabytých zkušeností na komerčních projektech.

2.5 Popis zvolených technologií

Tato část podrobně popíše technologie, které jsem se rozhodl použít pro vývoj aplikace.

2.5.1 Amazon Web Services (AWS)

Z pohledu serverové části budeme potřebovat hlavně služby pro poskytnutí výpočetní infrastruktury, databáze, úložiště a základní zabezpečení.
Nabídka služeb pro zprostředkování výpočetní infrastruktury je velmi pestrá. Amazon např. nabízí virtuální stroj *Elastic Compute Cloud (EC2)* či službu *Lambda* pro událostmi řízené, bezestavové aplikace, vyžadující rychlé odezvy.
Ohledně databázových platforem máme na výběr mezi *Relational Database Service (RDS)*, *Aurora* a *DynamoDB*.
RDS a Aurora patří do skupiny relačních databází. DynamoDB je typem tzv. *NoSQL* databáze, což jsou rychlé, nerelační databáze, využívající širší škálu modelů jako jsou dokumenty, grafy aj.
Co se týče služeb pro úložiště, Amazon také nabízí více možností:

- *Elastic Block Storage (EBS)* je lokálním trvalým úložištěm pro EC2, relační i nerelační databáze, datové skladování, *enterprise-level* aplikace, práci s velkými daty nebo pro zálohy.
- *Elastic File System (EFS)* je rozhraním pro souborové systémy, systémy pro zpřístupnění dat napříč více instancemi (kupříkladu EC2), správu obsahu, *enterprise-level* aplikace a práci s velkými daty nebo pro zálohy.
- *Simple Storage Service (S3)* je škálovatelnou platformou pro zpřístupnění dat napříč Internetem. Je hlavně určen pro uživatelem vytvářený obsah, aktivní archivy, bezserverové zpracovávání dat a práci s velkými daty nebo pro zálohy.

AWS také nabízí celou škálu služeb, poskytující zabezpečení aplikací:

- *Certificate Manager* - pro správu *Secure Sockets Layer (SSL)* nebo *Transport Layer Security (TLS)*
- *Identity and Access Management (IAM)* - pro správu rolí a přístupů do aplikací běžících na AWS.
- *Shield* - ochrana proti DDoS útokům
- *Web Application Firewall (WAF)* pro ochranu proti běžným útokům na webové aplikace.

K účelům bakalářské práce z velké nabídky služeb AWS využijeme pouze následující služby:

- Relational Database Service (2.5.1)
- Identity and Access Management (2.5.1)
- Simple Storage Service (2.5.1)
- Lambda (2.5.1)
- API Gateway (2.4)

Relational Database Service (RDS)

Už z názvu této služby vyplývá, že se bude jednat o službu poskytující platformu pro relační databáze. Relational Database Service nabízí jednoduché vytvoření, správu a škálovatelnost databáze. Nabízí podporu pro 6 databázových enginů:

- Amazon Aurora
- MySQL
- MariaDB
- Oracle
- Microsoft SQL Server
- PostgreSQL

Používáme-li například jeden z nabízených enginů, pak je velmi jednoduché přenést databázi ze svého serveru na RDS od Amazonu.
Pro účely naší aplikace využiji jeden z nejrozšířenějších open-source databázových enginů pro relační databáze - MySQL. Na výběr máme několik jeho verzí. Naše aplikace bude využívat jeho nejnovější verzi 5.7, která přináší spoustu novinek oproti starším verzím. A to hlavně nový datový typ JSON, dynamické indexování sloupců a všeobecné optimalizace a zrychlení.
Pro testovací účely v rámci bakalářské práce budou veškerá nastavení (přidělená úložná kapacita, procesory, paměti a podobně) nastavena na minimum. Např. úložná kapacita bude pouze 5 GB.

Identity and Access Management (IAM)

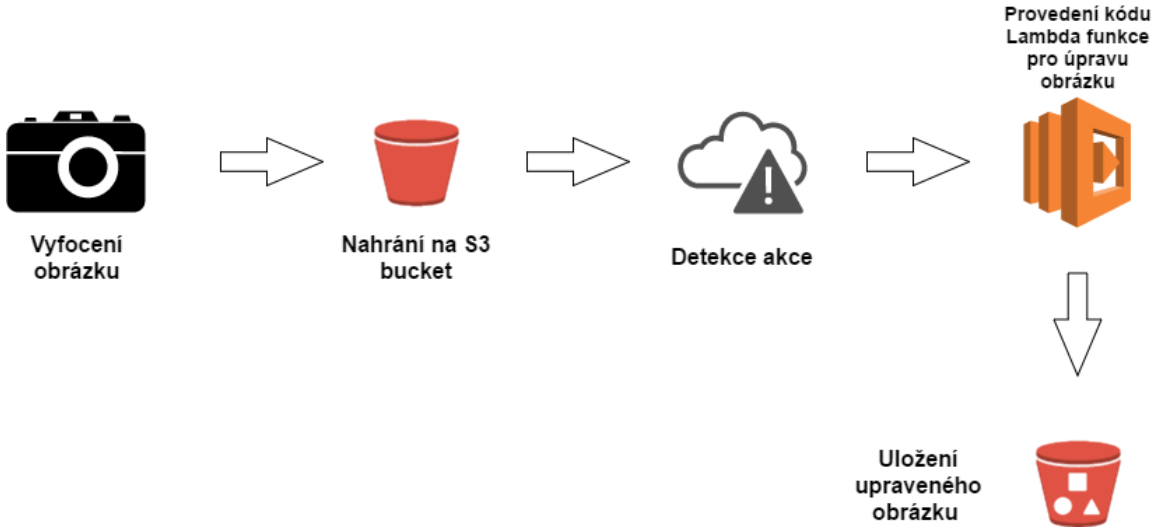
IAM je službou patřící do kategorie služeb zaměřených na zabezpečení aplikace. Tato služba umožňuje správu uživatelů Amazon Web Services, kde můžeme spravovat přístupy uživatelů k různým poskytovaným službám AWS. Tato služba není přímo placená.

Simple Storage Service (S3)

Úložištěm pro naší aplikaci bude služba Simple Storage Service. Mezi další její typické use-case patří i hosting pro statické stránky nebo v našem případě i jednostránkové aplikace.
S3 ukládá data jako jednotlivé objekty uložené ve zdrojích, které jsou označovány pojmem *buckets*. Nad těmito objekty můžeme používat operace jako *read*, *write* a *delete*. Objekty mohou dosahovat kapacity až 5 TB. Jednotlivým objektům můžeme zadávat různá přístupová práva v daném bucketu. Dále si můžeme vybrat, ve kterém Amazon Web Services regionu bude náš *bucket* uložen. V našem případě zvolíme region Frankfurt, který je nejblíže a nabídne tedy nejrychlejší odezvu.
Pro správu dat S3 nabízí jednoduchou webovou management konzoli, mobilní aplikaci či REST API a SDK pro snadnou integraci s technologiemi třetích stran.
Samozřejmostí je možnost integrace do AWS ekosystému s dalšími službami jako RDS, Route 53, Lambda a podobně.

Lambda

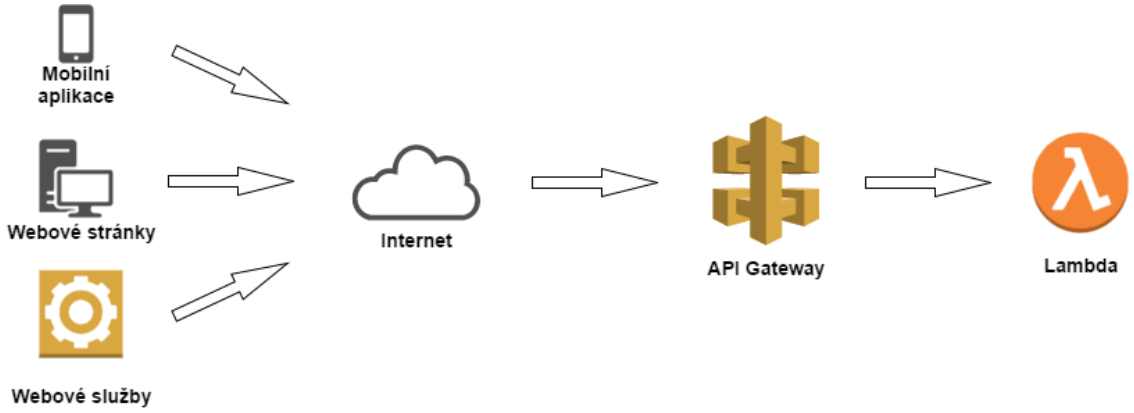
Služba AWS Lambda umožňuje běh kódu naší aplikace, aniž bychom museli spravovat jakýkoliv server. U této služby platíme pouze za čas, který náš kód spotřebuje a neplatí se nic, pokud kód není využíván.
Využívání Lambdy je hlavně spojeno se zpracováváním dat nebo zpracovávání souborů v reálném čase.
Jednotlivé funkce Lambda můžeme nastavit, aby se spouštěly automaticky nebo událostmi, způsobené interakcí klientské aplikace s uživatelem.
V našem případě budeme využívat služby Lambda hlavně pro práci s databází, kde si napíšeme kód v Node.js, který poté nasadíme na AWS Lambda, kde službu propojíme se službou API Gateway (viz obrázek 2.5).



Obrázek 2.4: Zpracování obrázku využitím služby Lambda

API Gateway

Amazon API Gateway je další službou typu *pay-as-you-go*. Umožňuje vývojářům jednoduchou správu, monitoring, udržování a zabezpečení API.
API Gateway nabízí i generaci SDK v libovolném jazyce (např. JavaScript, PHP, Java, Swing a mnoho dalších), které poté můžeme integrovat do klientské aplikace.



Obrázek 2.5: Propojení API Gateway s Lambda

2.5.2 ECMAScript 2015 (ES6)

Je specifikací skriptovacího jazyka od *European Computed Manufactures Association (ECMA)*. Byl vytvořen za účelem standardizace např. JavaScriptu. Vychází z něj i další skriptovací jazyky jako JScript či ActionScript. Jedná se o rozdílné skriptovací jazyky, ale všechny tyto jazyky mají společné jádro - ECMAScript.
V projektu bude využívána poslední standardizovaná edice ECMAScriptu, známá hlavně pod zkratkou ES6. Existuje už i verze ES7, která je ovšem ve fázi draftu a obohacuje JavaScript například o dekorátory.

const, let

Do verze ES6 se používalo pro deklaraci proměnných klíčové slovo `var`. Docházelo tady k tzv. *hoistingu*, což je známý jev pro JavaScript. Hoisting způsobuje, že automaticky přesune veškeré deklarace na začátek funkce. Tedy můžeme pracovat s proměnnou předtím než dojde k její deklaraci. Tento jev se děje protože `var` je *function-scoped* [5]. To znamená, že přestože deklarujeme proměnnou v určitém bloku, tak k dané proměnné bude mít přístup celé tělo funkce. Což může být matoucí pro programátory, kteří jsou zvyklí z jazyků jako C, C++, Java a jiných.
ECMAScript představil nová klíčová slova `const` a `let` pro deklaraci proměnných, která jsou *block-scoped*, tzn. jsou definovaná pouze v daném bloku a nepodléhají hoistingu [14]. Přístup k této proměnné mimo tento blok způsobí chybu. Demonstrující příklady kódu:
ES5:

```
function foo(x) {
  y = 0;

  if (x > 0) {
    var y = 2;
  }

  return x * y; // 10
}
```

ES6:

```
const bar = x => {
  y = 0; // ReferenceError

  if (x > 0) {
    let y = 2;
  }

  return x * y;
}
```

Třídy

Dlouho očekávanou vlastností JavaScriptu byla možnost vytváření tříd. Do ECMAScript 6 se objekty určitého typu vytvářely pomocí prototypů. ECMAScript 2015 přináší možnost vytvářet objekty pomocí tříd. Nejedná se o klasické třídy, jak je známe z třídních objektově orientovaných jazyků jako jsou C++ či Java. Jde pouze o implementaci tzv. továrních funkcí [13].

Arrow funkce

Jedná se o spíše tzv. *syntactic sugar*. Odstraňuje zbytečný a opakující se boilerplate kód, potřebný pro definici funkce. Před ES6 definice Redux middleware (2.8) vypadala například takto:

```
function exampleMiddleware(store) {
  return function (next) {
    return function (action) {
      return next(action);
    };
  };
```

```
}
```

Stejná implementace pomocí arrow funkcí:

```
const exampleMiddleware = store => next => action => next(action);
```

Výchozí hodnoty argumentů funkce

ES6 přináší možnost zadávat výchozí hodnoty parametrům funkce, pokud nebyly specifikovány při volání funkce.

```
const reducer = (state = {}, action = { type = 'DEFAULT' }) => { ... };
```

Spread a rest operátory

Slouží ke zjednodušení práce s poli či objekty. *Spread* operátor představuje výraz pro expanzi na místech, kde se očekává více argumentů, elementů nebo proměnných (volání funkce, literály pole).
Volání funkce

```
const add = (x, y, z) => x + y + z;

const args = [ 5, 6, 7 ];

add(...args);
```

Slučování polí

```
const array1 = [ 1, 2, 3 ];
const array2 = [ 4, 5 ];

const mergedArray = [ ...array1, ...array2 ]; // [ 1, 2, 3, 4, 5 ]
```

Rest operátor slouží hlavně k destrukturalizaci polí a objektů.
Destrukturalize pole:

```
const array = [ 1, 2, 3, 4, 5 ];

const [ one, two, ...rest ] = array;
// one = 1
// two = 2
// rest = [ 3, 4, 5 ]
```

... a objektu:

```
const person = {
  firstName: 'Jack',
  surname: 'Brien',
  email: 'jack.brien@gmail.com',
  phone: '(454) 123-4567',
};

const { firstName, surname, ...contact } = person;
// firstName = 'Jack'
// surname = 'Brien'
// contact = { email: 'jack.brien@gmail.com', phone: '(454) 123-4567' }
```

Promise

Promise je nový typ objektu používaný při asynchronních výpočtech. Reprezentuje hodnotu, která může být přístupná teď, v budoucnosti nebo nikdy. Promise může být v jednom ze tří stavů [7]:

- *pending* - počáteční stav
- *fulfilled* - stav, který značí, že operace proběhla úspěšně
- *rejected* - stav, který značí, že operace selhala

Příklad práce s Promise:

```
const fetchData = () => {
  return fetch('url')
    .then(response => JSON.parse(response))
    .then(json => someHandler(json))
    .catch(error => errorHandler(error));
};
```

Template literals

Template literály představují prostředek pro tvorbu dynamických a parametrizovatelných řetězců.

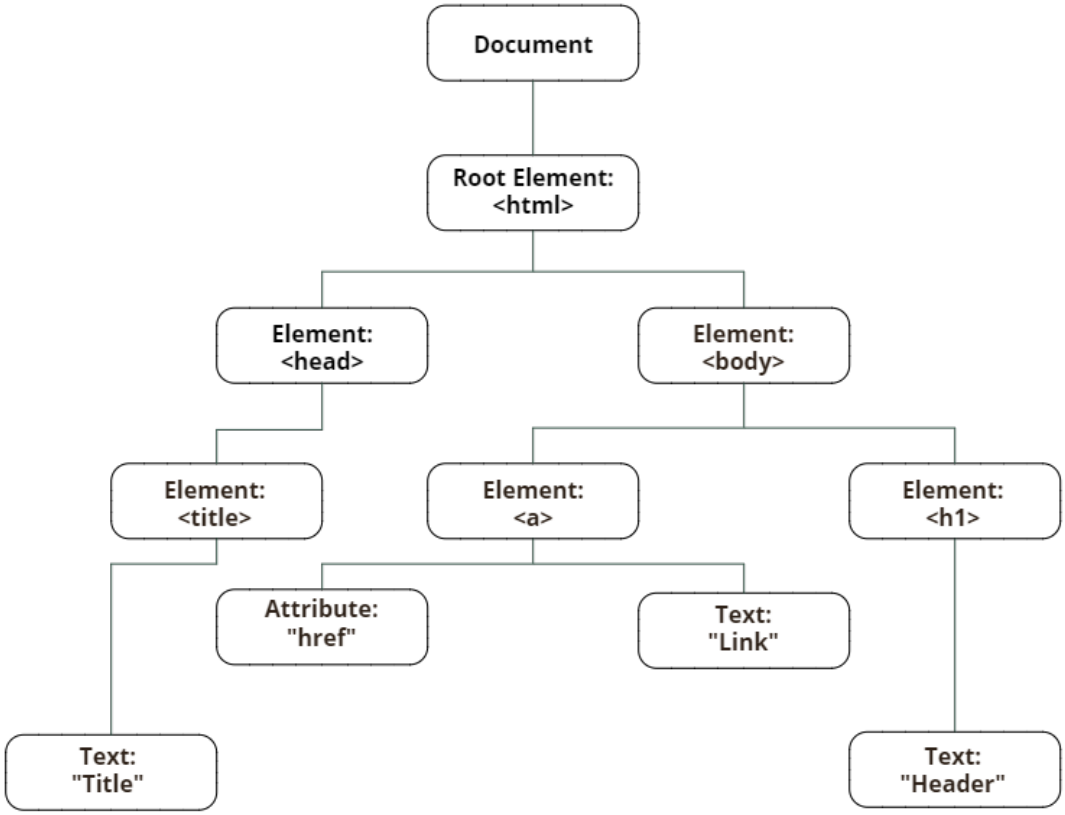
```
// ES5
var name = 'Jack';
var text = 'Hello ' + name;

// ES6
const name = 'Jack';
const text = `Hello ${name}`;
```

2.5.3 React

Předchozí sekce stručně popsaly a představily knihovnu React. Tato část podrobněji popíše Virtual DOM, jak funguje překreslování komponent, když nastane změna. Vysvětlí životní cyklus komponenty a její vnitřní stav a vyčlení základní typy komponent.

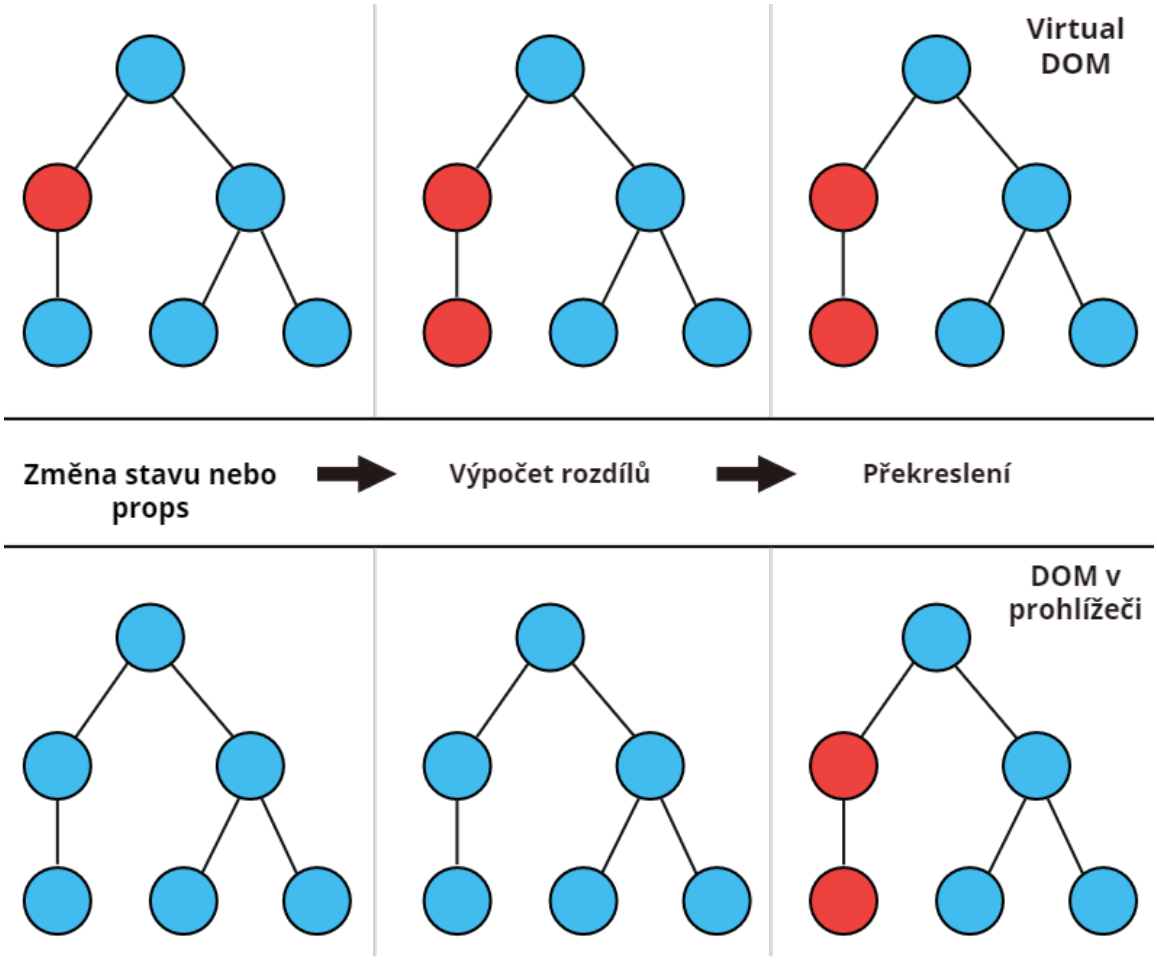
Virtuální DOM

Document Object Model (DOM) (obrázek 2.6) je abstrakcí strukturovaného textu. Tento text reprezentuje HTML kód. Jednotlivé HTML elementy představují uzly DOMu. DOM je vždy stromové struktury, což je dáno strukturou HTML dokumentu. To přináší poměrně jednoduché vyhledávání ve stromě, ale bohužel není to rychlé. V dnešní éře dynamických single page aplikací, kde pracujeme s velkým DOM stromem, ve kterém dochází k častým změnám, přesto potřebujeme, aby aplikace na tyto změny rychle reagovala.



Obrázek 2.6: Document Object Model

Virtuální DOM představuje abstrakci nad klasickým DOMem, tedy abstrakci nad abstrakcí. Prakticky vývojáře úplně odstíní od práce s DOMem v prohlížeči. V komponentě pouze deklarativně nadefinujeme strukturu (HTML) kompozicí JavaScriptových funkcí. To znamená, že popíšeme, jak má výsledná stránka vypadat na základě přijatých dat. React následně z těchto komponent poskládá virtuální Document Object Model, porovná jej s DOM v prohlížeči pomocí rychlých, efektivních algoritmů a aktualizuje pouze komponenty, ve kterých nastala změna [11].



Obrázek 2.7: Posloupnost kroků pro překreslení ve virtuálním DOM

Životní cyklus komponenty

React komponenta reaguje na 3 události:

- *mounting*, kdy dochází ke vložení komponenty do DOMu
- *updating*, kdy dochází ke změně stavu nebo props
- *unmounting*, kdy dochází k odstraňování komponenty z DOMu

Pro každou z těchto událostí React komponenta nabízí soubor metod, které můžeme přetěžovat. Posloupnost funkcí, které se volají při vkládání komponenty do DOMu:

1. `constructor()`
2. `componentWillMount()`
3. `render()`
4. `componentDidMount()`

Pořadí volaných metod při aktualizaci komponenty:

1. `componentWillReceiveProps()`
2. `shouldComponentUpdate()`
3. `componentWillUpdate()`
4. `render()`
5. `componentDidUpdate()`

A při odstraňování z DOMu:

1. `componentWillUnmount()`

Stav komponenty

Pokud nepoužíváme k Reactu žádnou knihovnu pro management stavu aplikace, můžeme použít pro práci s daty stav komponenty. Stav React komponenty je immutable objekt. Pokud budeme upravovat objekt reprezentující stav jinak než pomocí metod, které má React pro práci se stavem, může dojít k neočekávaným chybám.
Pro nastavení stavu se používá hlavně metoda `setState()`. Tato metoda se snaží porovnat zadaný objekt s aktuálním stavem a sloučit tyto objekty. Zde se však hůře pracuje se stavem, který má složité a hlouběji zanořené struktury.
Nevýhodou tohoto principu práce se stavem je horší organizace a přehlednost. Pokud vyvíjíme rozsáhlejší aplikaci, je lepší mít stav aplikace uchovaný mimo komponenty - např. v Redux *Store* (2.8).

Předávání dat

React je založen hlavně na předávaní dat jedním směrem (tzv. *uni-directional data flow*). Pro předávaní dat mezi komponentami se používají tzv. *props* a data se předávají z rodičovské komponenty směrem dolů, tedy potomkům. Pokud je to nezbytně nutné, React nabízí možnost, jak může potomek ovládat data rodičovské komponenty přes *refs*. Pokud je potřeba využívat tohoto postupu, často to znamená, že hierarchie komponent je špatně navržená.
Data se předávají potomkům následovně:

```
1 <ChildComponent data={this.state.dataForChild} />
```

Potomek poté přistoupí k přeposlaným datům takto: `this.props.data`.

Kontrola props

React nabízí i způsob kontroly přijatých props pomocí `propTypes`, můžeme tedy zadat, že komponenta `ChildComponent` očekává řetězec data v `props`. Pokud komponentě pošleme jiný datový typ než očekává, React nás upozorní, že jsme poslali jiná data, než daná komponenta očekává. Také se dá nastavit i výchozí hodnota `props`, pokud komponentě není nic posláno, pomocí `defaultProps`.
Příklad:

```
function ChildComponent({ data }) {
  return (
    <div className="data-container">
      {data}
    </div>
  );
}

ChildComponent.propTypes = {
  data: React.PropTypes.string,
};

ChildComponent.defaultProps = {
  data: '',
};
```

JSX

U předchozí ukázky kódu si můžeme všimnout části uvnitř bloku návratové hodnoty funkce.

```
return (
  <div className="data-container">
    {data}
  </div>
);
```

Kód velmi připomíná HTML, ale ve skutečnosti je to JavaScript, proto je například HTML atribut `class` nahrazen za `className`, protože `class` je v JavaScriptu klíčové slovo. Další odlišnou částí oproti klasickému HTML je část ve složených závorkách `{data}`. Mezi složenými závorkami můžeme zadávat JavaScriptový kód. V tomto případě tedy předáváme data pro vykreslení.
Jedná se o tzv. JSX syntaxi, což je *syntactic sugar* pro metodu `React.createElement(component, props, ...children)`.
Předchozí JSX kód se překompiluje do:

```
React.createElement(
  'div',
  { className: 'data-container' },
  data
);
```

Programátor není nucen používat JSX pro vývoj, může používat přímo metodu `React.createElement()`. Nicméně jak je vidět, kód v JSX je daleko přehlednější.

Typy komponent

V Reactu můžeme vytvářet komponenty více způsoby. Buď zastaralejší metodou přes `React.createClass()` nebo novějším způsobem využívajícím ES6 třídy:

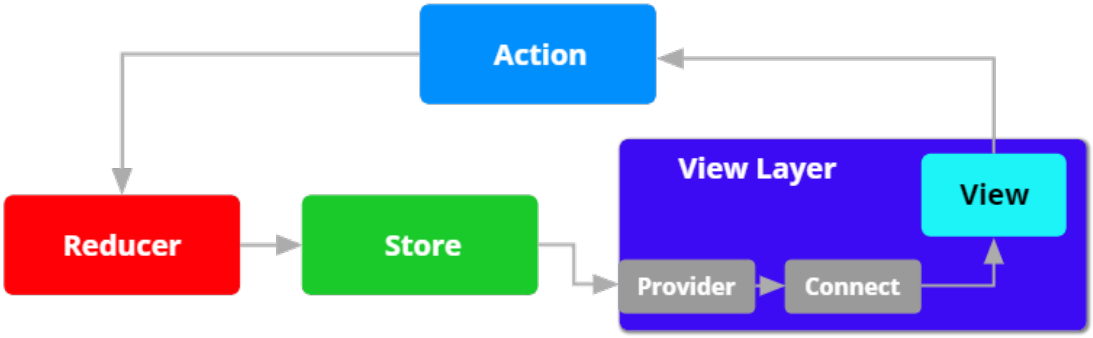
```
class MyComponent extends React.Component { ... }
```

Těmito způsoby vytváříme komponenty, které mohou mít stav a mají přístup k lifecycle metodám. Naproti tomu je možno vytvářet i tzv. funkcionální komponenty. Tedy komponenty, které nemají lifecycle metody ani stav. Jsou jednodušší a tedy nejrychlejší na vykreslování. Většina komponent v aplikaci by měly být právě funkcionální komponenty.
Jedním z dalších dělení komponent je rozdělení na *stateful* a *stateless* komponenty. Dle názvu stateful komponenta je komponenta, která má stav a naopak stateless stav nemá.

Funkcionální komponenty stav nemají, proto jsou také často označovány pojmem *functional stateless component*.
Dalším dobrým zvykem je rozdělit komponenty na prezentační a kontejnerové. Prezentační nebo-li „hloupé“ komponenty nemají žádnou logiku, slouží pouze pro prezentaci dat [1]. Ve většině případů jsou prezentační komponenty právě bezstavovými funkcionálními komponentami.
Naopak kontejnerové komponenty obsahují veškerou logiku pro řízení toku a zpracování dat. Neměly by obsahovat žádné stylování, od toho máme právě prezentační komponenty. Plní funkci *wrappera*, obaluje nějakou prezentační komponentu a předá jí data a funkcionalitu pro zpracování událostí vzniklých při interakci s uživatelem [1].

2.5.4 Redux

Redux je podmnožinou architektury Flux, což je spíše návrhový vzor než samostatná knihovna. Striktně určuje jednosměrný tok dat (viz obrázek 2.8).
Rozdíl mezi Fluxem a Reduxem je, že Redux má pouze jeden Store a nemá Dispatchera.



Obrázek 2.8: Jednosměrný tok dat v Reduxu

Redux se opírá o 3 fundamentální principy:

- „*Store* je jediný zdroj pravdy“ - veškerý stav aplikace je uložen v jediném stromovitém objektu. Výhodou tohoto principu, je univerzálnost. Stav může být jednoduše serializován a hydratován ze serveru bez nějakého složitějšího zpracovávání. Jeden zdroj informací také přináší lepší přehled o datech v aplikaci a rychlejší ladění.
- Stav aplikace je pouze čitelný - jediný způsob, jak změnit stav aplikace je vyslat *akci* (2.8), tedy objekt, který nese informace o tom, co se stalo. Výhodou je menší pravděpodobnost vedlejších účinků - vše je předvídatelné. Všechny změny jsou centralizované a dějí se v přesném pořadí, nemůže tedy např. dojít k *race conditioning*.
- Změny jsou zpracovávány funkcemi bez vedlejších účinků - tzv. *reducery* (2.8) určují, jak daná *akce* změní stav aplikace. Dodržíme-li zásadu, že reducery budou *pure funkce* [6], získáme opět jasné a prediktivní chování aplikace.

Akce

Akce jsou objekty s informacemi, které jsou odeslány z aplikace do *Store* (2.8). Jsou jediným možným zdrojem informací pro *Store*. Jedinou podmínkou pro objekt akce je, že

musí obsahovat atribut `type`, který definuje tuto akci pro další zpracovávání. V komunitě okolo knihovny Redux existuje více různých pravidel, která se týkají akcí. Jedním z nich je například *Flux Action Standard*, který definuje, jak má vypadat struktura akce.

```
{
  type: 'ADD_TODO',
  payload: {
    id: 1,
    text: 'Make something.',
  },
}
```

Dobrým zvykem pro vytváření akcí je mít tzv. *action creator* funkce, kterým pošleme data a funkce z nich vytvoří objekt *akce*.

```
const addTodo = (id, text) => ({
  type: 'ADD_TODO',
  payload: {
    id,
    text,
  },
});
```

Reducer

Název *reducer* je odvozen z JavaScriptové funkce `Array.prototype.reduce`, která umožňuje ve funkcionálním programování procházet jednotlivými položkami pole a aplikovat na ně námi zvolenou funkci pro netriviální výpočty [8].
Úkolem reducerů je reagovat na vyslané akce a na jejich základně měnit stav aplikace. Reducer pracuje se dvěma parametry a to s aktuálním stavem a akcí, na základě informací z akce provede změny stavu a vrátí nový objekt, reprezentující stav aplikace.
Jak už bylo mnohokrát zmíněno, reducery musí být *pure function* [6]. V reducerech tedy nikdy nesmíme:

1. Měnit nebo mutovat argumenty.
2. Vytvářet vedlejší efekty používáním API nebo směrováním, tyto věci musí být obslouženy mimo reducery.
3. Volat funkce, které nejsou *pure function*. Tedy například: `Date.now()` nebo `Math.random()`.

Za každou cenu musí reducer pro stejné vstupy vrátit ten samý výstup.

```
function todoReducer(state = {}, action) {
  switch(action.type) {
    case 'ADD_TODO':
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            id: action.payload.id,
            text: action.payload.text,
          }
        ]
      });
    default:
      return state;
  }
}
```

Příklad ukazuje, že nemutujeme stav. Vytvoříme vždy nový objekt funkcí `Object.assign()`. Důležité je i mít obsluhu pro *akce*, které daný *reducer* nezpracovává. Akce totiž procházejí všemi reducery a pokud bychom neměli obsluhu pro neznámé akce, přišli bychom o data, které má daný reducer na starost.
Reducery se dají komponovat. Je lepší mít více menších reducerů, které jsou zodpovědné za menší celky stavu aplikace.

Store

Store slouží jako úložiště, je v něm uložen objekt, reprezentující stav aplikace. Dohromady spojuje logiku *akcí* a *reducerů*. *Store* je zodpovědný za:

1. Udržování stavu aplikace.
2. Poskytnutí přístupu ke stavu aplikace skrze funkci `getState()`
3. Zpřístupnění funkce `dispatch(action)`, která posílá akce, díky kterým měníme stav aplikace.

Middleware

Middleware rozšiřují funkcionalitu akcí. Poskytují rozšíření v bodě mezi dispatchem akce a momentem, kdy se akce dostane ke zpracování v reduceru. Nejčastějším využitím middleware je logování, crash reporting, asynchronní komunikace s API, směrování apod.
Příklad middlewaru pro logování akcí a následné změny stavu:

```
const loggerMiddleware = store => next => action => {
  console.log(`Dispatched action: ${action}`);

  const nextState = next(action);

  console.log(`Next state: ${nextState}`);

  return nextState;
};
```

Listing 2.1: Logger middleware

Propojení s Reactem

Aby mohly být komponenty napojeny na Redux a mít tak přístup ke stavu aplikace, je nutno mít knihovnu *react-redux*, která poskytuje komponentu `Provider` a funkci `connect()`. `Provider` nejčastěji bývá kořenová komponenta aplikace, a očekává *Store* jako vstupní `props`. Jeho úlohou je zpřístupnit data komponentám.
Výchozí bod aplikace tedy zpravidla vypadá následovně:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import { rootReducer } from './reducers';
import App from './App';

const store = createStore(rootReducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Chceme-li, aby měla komponenta přístup ke stavu aplikace, musíme ji obalit o rozšířenou funkcionalitu a přístup ke *Store*, pomocí funkce `connect()`, která obalí naši komponentu komponentou vyššího řádu. Funkce `connect()`, očekává dva parametry:

- `mapStateToProps()` slouží k namapování stavu do komponenty. Nejčastěji se zde používají tzv. *selectory*, které vrací pouze určitou část stavu aplikace.
- `mapDispatchToProps()` slouží k namapování dispatch funkce pro různé handlery.

A předá data či handlery komponentě skrze `props`. Demonstrující příklad:

```
import React from 'react';
import { connect } from 'react-redux';
// import addTodo, TodoListComponent, ...

function TodoList ({ todos, handleAddTodo }) {
  return <TodoListComponent todos={todos} addTodo={handleAddTodo} />;
}

const mapStateToProps = state => ({
  todos: state.todos,
});

const mapDispatchToProps = dispatch => ({
  handleAddTodo: (id, text) => dispatch(addTodo(id, text)),
});

const connectedTodoList = connect(
  mapStateToProps,
  mapDispatchToProps,
)(TodoList);
```

2.5.5 ImmutableJS

ImmutableJS je knihovna, která disponuje *immutable* strukturami nebo-li datovými strukturami, které nemohou mutovat. Knihovna ImmutableJS výborně zapadá do filozofie Reduxu (2.5.4) a proto se snadno integruje. Ideálním výsledkem je, že celá aplikace pracuje s těmito *immutable* strukturami. Zaručí se tím udržovatelnost stavu aplikace a přehled o toku dat.

Kapitola 3

Návrh aplikace

Tato kapitola se zaměří na popis návrhu aplikace, která bude sloužit pro demonstrační účely, jak se dají technologie vzájemně využít a integrovat.
Kapitola nejprve popíše, jakou aplikaci budeme vyvíjet (3.1) a srovná ji se současnými, existujícími aplikacemi (3.2). Pro aplikaci navrhneme relační model databáze (3.3.1) a *Application Programming Interface (API)*, se kterou bude klientská aplikace komunikovat.
Dále kapitola popíše návrh grafického uživatelského rozhraní webové aplikace (3.4.2). Zaměří se i na popis struktury stavu (3.4.1).

3.1 Zadání aplikace

Bude se jednat o systém na míru pro správu interních věcí firmy. Aplikace by měla umožňovat snadnou a intuitivní správu klientů, jejich projektů a zaměstnanců, kteří budou na daném projektu pracovat. Aplikace by měla vycházet z Material UI design.
Aplikace bude mít dva typy uživatelů - manažery a zaměstnance.

3.1.1 Manažer

- Bude mít přístup ke všem datům.
- Možnost vytváření a editace klientů, úkolů a činností
- Možnost vytvářet a editovat zaměstnance

3.1.2 Zaměstnanec

- Přístup pouze k úkolům, ke kterým je přidělen
- Editace svých činností, editace svého profilu

3.1.3 Správa klientů

Systém bude evidovat klienty. Ke klientovi můžeme mít přiřazených více úkolů. O klientovi si evidujeme základní informace jako jméno klienta nebo název firmy, adresu, registrační či identifikační číslo firmy a daňové identifikační číslo. Dále si u klienta uchováváme řešené projekty.

3.1.4 Správa úkolů

Dalším požadavkem na systém je evidence tzv. úkolů, které se vztahují k danému projektu. U jednotlivých úkolů evidujeme, název úkolu, název projektu, typ úkolu, kategorie, a pak na základě typu úkolu budeme rozlišovat výpočet ceny dle strávených hodin či jednorázovou částkou. Dále u úkolu je vyžadován WYSIWYG editor pro psaní poznámek a mít možnost přidávat různé přílohy. Dále evidence, kdy byl úkol dokončen a kdy zaplacen klientem.

3.1.5 Správa činností

Systém by měl také spravovat jednotlivé činnosti, které budou vázané na úkoly. U těchto činností si budeme uchovávat zaměstnance, který činnost prováděl, popis, poznámky. Od systému se očekává, že zaměstnanec si bude moct hlídat čas strávený nad daným úkolem.

3.1.6 Správa zaměstnanců

Posledním požadavkem na systém je jednoduché spravování zaměstnanců. U jednotlivých zaměstnanců si systém bude uchovávat jméno, příjmení, login, heslo a roli. Jednotliví zaměstnanci budou mít možnost si editovat pouze svůj profil. Manažeři budou moci editovat vše.

3.2 Podobné aplikace

Pro správu a management zaměstnanců na projektu existuje mnoho aplikací. Neexistuje však žádná, která splňuje přímo dané zadání. Pro splnění by se daly využít kombinace různých aplikací, ale zde už by se firma zbavila pohodlí mít veškeré potřebné věci na jednom místě.

3.2.1 Toggl

Toggl je aplikací zaměřenou na měření času. Uživatel zde může mít více workspace, kde si může měřit časy strávené na různých projektech. Aplikace dále nabízí přehledné vizualizace naměřených dat po dnech, týdnech, měsících atd. Toggl je i mobilní aplikací. A umožňuje tedy synchronizaci mezi více zařízeními.

3.2.2 Asana

Asana je dalším existujícím systémem pro kooperaci a řízení týmů prostřednictvím úkolů. Opět nabízí i mobilní aplikaci a umožňuje automatickou synchronizaci.

3.2.3 Meistertask

Další možností pro správu úkolů se sdílením mezi více uživateli je Meistertask. Opět hlavním prvkem je úkol, který můžeme přiřazovat dalším lidem, a různě jednotlivé úkoly značkovat, kategorizovat a podobně. I v tomto případě existuje mobilní aplikace. Pro větší týmy, kde je potřeba sdílení informací a úkolů mezi více lidmi je potřeba vlastnit placenou verzi. Dále může být integrován s dalšími službami jako například GitHub či Dropbox.

3.2.4 Trello

Trello je asi nejznámější aplikací pro týmový management. Nabízí stejnou funkcionalitu jako Meistertask, ale je dražší.
Existuje mnoho dalších aplikací, řešících problematiku řízení projektů, ale žádná nenabízí, přesně to, co je v zadání.

3.3 Serverová část

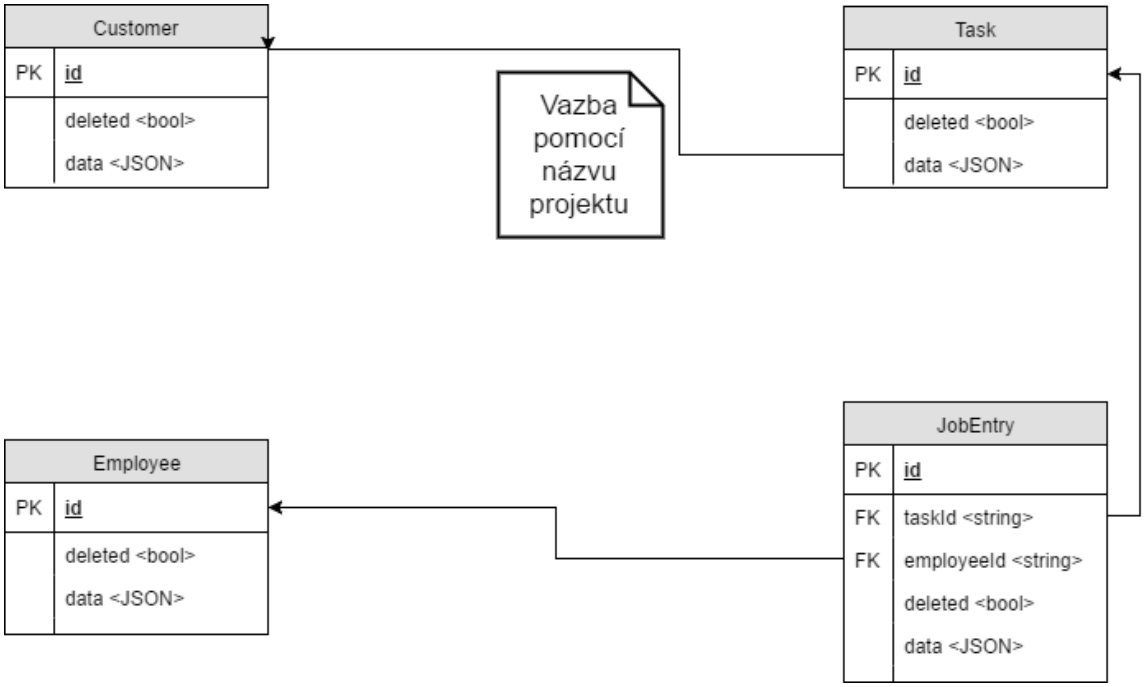
Tato část popíše, jak budeme ukládat jednotlivé entity v databázi a popíše vazby mezi nimi. Dále provede rozbor a návrh potřebného API, se kterým bude webová aplikace komunikovat. A na závěr popíše návrh komunikačního protokolu mezi serverem a klientem.

3.3.1 Databázový model

Ze zadání plyne, že je potřeba mít následující databázové tabulky:

- Customer
- Task
- JobEntry
- Employee

Entita tabulky Customer může mít více vazeb na tabulku Task. Dále entita tabulky Task může mít více vazeb na tabulku JobEntry, která bude ještě spojena vazbou s tabulkou Employee. A v databázi si budeme udržovat i entity, které budou vymazány. Tuto funkcionalitu vyřešíme tzv. lehkým smazáním (angl. *soft delete*), tedy daná entita bude mít atribut nesoucí informaci o smazání. *Soft delete* se používá z důvodu, že chceme uchovávat i vymazaná data ze systému a udržovat databázi konzistentní.
Jak již bylo zmíněno v sekci popisující Amazon Web Services - konkrétně o službě Relational Database Service (2.5.1). Pro aplikaci použijeme MySQL verze 5.7, která umožňuje používat datový typ JSON, který značně ulehčí posílání dat skrze REST API [3].
Výsledný relační model databáze na základě požadavků vychází následovně (obrázek 3.1):



Obrázek 3.1: Relační model databáze

3.3.2 REST API

Klientská aplikace bude single page aplikací, proto nepotřebujeme žádnou pokročilou logiku či zpracovávání na straně serveru. Z tohoto důvodu, pro veškerou komunikaci mezi klientem, bude stačit implemetace tzv. *Representational State Tranfer Application Programming Interface*, známé spíš pod zkratkou REST API.
REST API nabízí jednoduchou cestu k získávání, editování či mazání dat. Hlavním konceptem této architektury je design pro distribuované systémy. Tedy systémy, které běží na více serverech, což zapadá do funkcionality cloudových aplikací.
Omezení REST API:

- Aplikace je architektury klient-server.
- Komunikace mezi klientem a serverem je bezstavová, nepřenášejí se na server kontextová data klienta. Pouze data, potřebná pro práci s databází.
- Dotazy se mohou cachovat, jednotlivé dotazy tedy musí nést informaci, zda se bude cachovat či nikoliv. Správně navržená strategie cachování dotazů umožní lepší rychlost komunikace a může snížit počet dotazů.
- Vrstvený systém - klient nemůže přesně vědět, jestli je napojen na koncový server či na nějakou mezivrstvu.

Každý end-point RESTful API musí mít definovaný:

- Přesnou *Uniform Resource Locator* (URL), která jednoznačně určuje koncovou adresu zdroje na internetu.

- Typ internetového média, který definuje elementy stavového přenosu dat (Atom, application/json, ...)
- Standardní HTTP metodu (GET, PUT, POST, DELETE, ...)

Pro účely demonstrační aplikace navíc budou dostačující základní *Create-Read-Update-Delete* (CRUD) metody, což dobře zapadá do filozofie REST API.

3.3.3 Návrh komunikačního protokolu

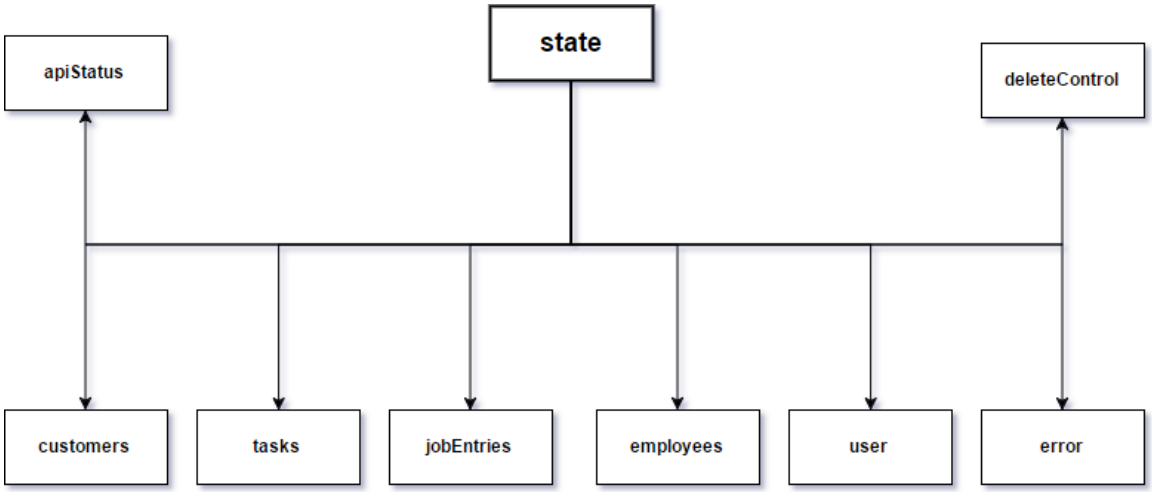
Komunikační protokol bude závislý na implementaci Lambda funkcí, které budou pracovat s databází. Jedinou společnou věcí, která se bude odesílat s každým dotazem na server, bude hlavička *Api-Version*, která jasně určuje, s jakou API komunikuje klientská aplikace. Toto opatření bude implementováno z důvodu možných budoucích změn v databázi či API.

3.4 Klientská část

Klientská část podrobněji popíše proces návrhu stavového objektu aplikace, kde rozebere návrh struktury dat pro jednotlivé databázové entity, celý tento proces bude brán v kontextu uvažování Reduxu. Tedy s možností jednoduché a smysluplné kompozice reducerů, které budou mít odpovědnost pouze nad určitým celkem dat.
Další část stručně popíše návrh grafického uživatelského rozhraní pro jednotlivé sekce.

3.4.1 Návrh stavového objektu aplikace

Datové struktury objektů, reprezentující databázové entity budou mít v klientovi, budou mít v klientovi složitější strukturu. A to hlavně z důvodu, že veškerá data pošleme na server, kde budou uložena jako JavaScript Object Notation (JSON), tedy nemusíme v databázi téměř vůbec řešit, co bude JSON obsahovat [3]. Veškerá tato logika spadá do klientské aplikace. Entity však nebudou jedinými objekty ve stavu aplikace, budou tam i další pomocné podstromy pro kontrolu stavu API a podobně.
Celý stav aplikace se dá shrnout do obrázku 3.2:



Obrázek 3.2: Návrh stavu aplikace

Jednotlivé atributy stavu na obrázku představují podstromy. Pro každý tento podstrom bude existovat reducer (2.8), který nad ním bude mít kontrolu. Reducery pro *customers*, *tasks*, *jobEntries* a *employees* budou mít ještě pomocné reducery, které budou pracovat přímo s jedním objektem na základě jeho ID. A to hlavně z důvodu, že nemusíme v reduceru pracovat s celým polem, ale právě pouze s jedním objektem, což usnadňuje práci a kód je přehlednější.
Podstromy *customers*, *tasks*, *jobEntries* a *employees* budou pole obsahující objekty jednotlivých entit. Každý z těchto objektů bude nést potřebné informace, dle zadání.
Např. struktura objektu pro entitu task bude určena dle následující ukázky:

```
tasks = [
  {
    id: "",
    data: {
      taskName: "",
      project: "",
      type: "",
      category: "",
      price: "",
      priceHours: "",
      perHour: "",
      description: {
        // objekt pro stav WYSIWYG editoru
      },
      attachment: [
        // pole objektu s informacemi o priloze
      ],
      finished: "",
      finishedDate: "",
      billed: "",
      billedDate: "",
      billNumber: "",
    },
  },
];
```

3.4.2 Návrh grafického uživatelského rozhraní

Hlavním požadavkem na grafické uživatelské rozhraní klientské aplikace je, aby vycházel z Material Design. I když je to spíše vzor pro responzivní weby, demonstrační aplikace nebude optimalizována pro mobily, nebude tedy responzivní.
Dalším požadavkem je, aby aplikace nabízela přehledy pro každou z entit. Přehledy budou jednoduché datagridy obsahující důležitá data s tlačítkem pro rychlý přestup na detailní obraz dané entity.

Material Design

Material Design je designovým jazykem pro tvorbu uživatelských rozhraní. Publikovala ho v roce 2014 firma Google. Je na něm postavená většina aplikací pro Android.
Zabývá se hlavně animacemi, pohyby, odstaveními mezi jednotlivými elementy, hloubkovými efekty, jako jsou stíny a světla, aby se vše co nejvíce přiblížilo realitě. Určuje například pravidla pro animace, aby byly co nejplynulejšími a nejpřirozenějšími pro uživatele:

- Responsivita - animace by měly trvat minimálně 150 ms a maximálně 450 ms. Ideálně by však měly trvat 300 ms. A to hlavně, aby netrvaly moc dlouho a uživatele nezdržovaly či zbytečně neupoutávaly pozornost. Zároveň by neměly být příliš krátké, aby si jich uživatel všiml a nebyl zmatený.
- Přirozenost - animace by se měly chovat přirozeně, tedy měly by být co nejblíže reálnému světu. Animace tedy nemá konstantní rychlost po celou dobu, ale na začátku zrychluje a ke konci zpomaluje.
- Choreografie, návaznost - existují určité vazby mezi jednotlivými elementy aplikace. Proto by animace měla vycházet z těchto vazeb a neporušovat je.
- Intuitivnost - týká se hlavně rozbalování a sbalování elementů do větších komponent. Animace rozbalování by teda dle přirozenosti měla začít tam, kde se nacházel původní element a ten postupně zvětšovat.

Material Design definuje celou škálu dalších pravidel ohledně layoutů, elementů, stylů a podobně.
Existuje i spousta knihoven implementujících jednotlivé komponenty Material UI přímo pro React a proto pro vývoj aplikace jednu z nich použijeme.

Popis návrhu GUI pro jednotlivé sekce

Aplikace by měla uživateli umožňovat rychlý přechod mezi jednotlivými sekcemi. Proto tedy aplikace bude obsahovat vysunovací menu s nabídkou odkazů. Jednotlivé položky vysunovacího menu budou sloužit jako navigace aplikace a budou tedy odkazovat do jednotlivých sekcí.
Aplikace také bude vyžadovat přihlášení uživatele. Stránka pro přihlašování bude obsahovat jednoduchý formulář pro vyplnění loginu a hesla.
Po úspěšném přihlášení se uživatel dostane na hlavní stránku, která obsahovat základní informace a poslední novinky.

Správa klientů

Sekce pro správu klientů by měla nabízet 2 typy pohledu. Jeden, kde budeme mít datagrid všech klientů firmy nesoucí základní informace o klientech. Nebude chybět ani tlačítko pro vytváření nových klientů.

- Název či jméno klienta
- Adresa
- Registrační číslo firmy
- Daňové identifikační číslo firmy
- Názvy projektů
- Odkaz na detail klienta

Druhý pohled bude detail klienta, který bude obsahovat vyplnitelný formulář pro zadávání informací o klientovi. Dále bude obsahovat datagrid všech úkolů, které jsou vázané na daného klienta.
Tabulka úkolů bude nést tyto informace:

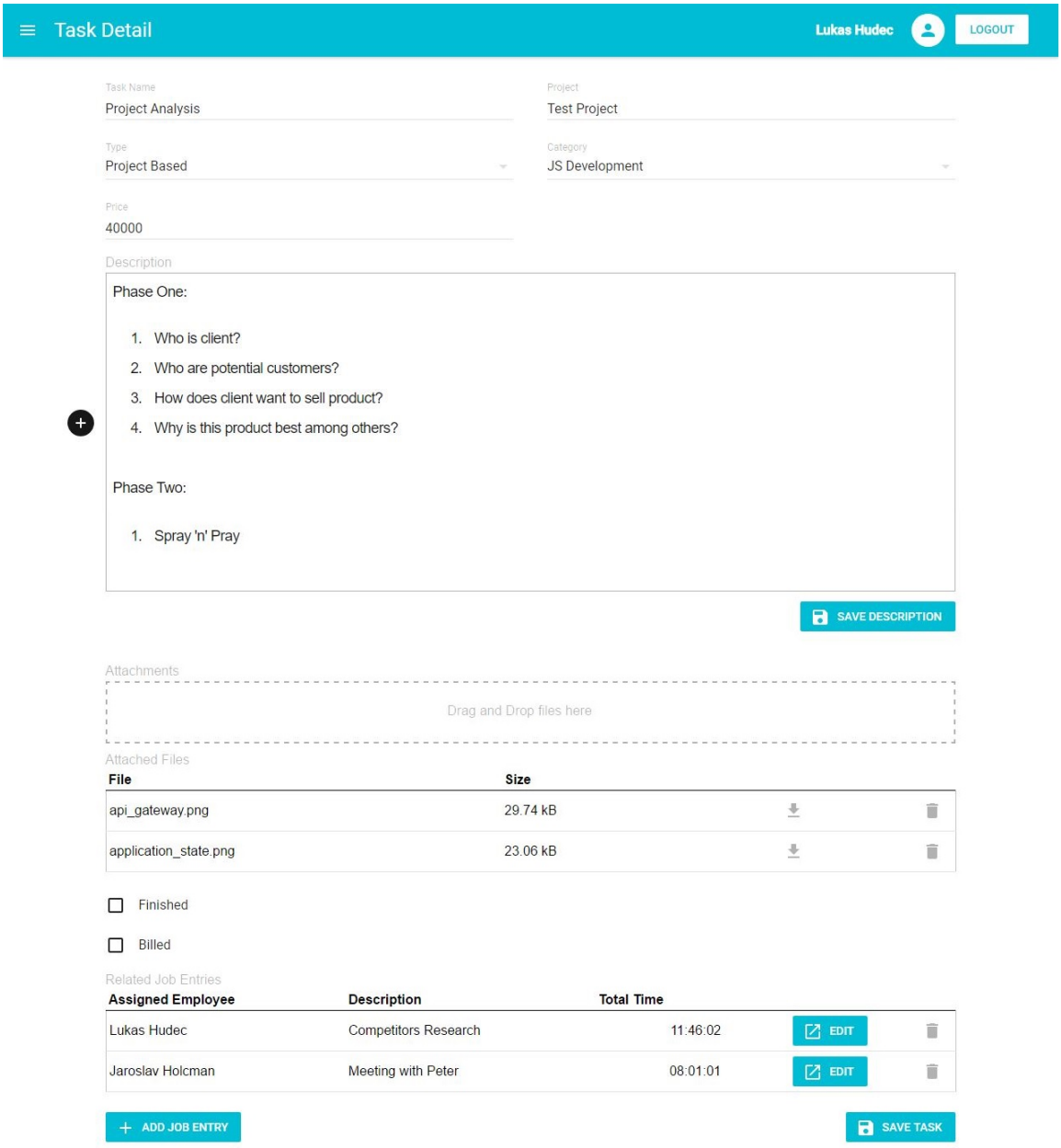
- Název úkolu
- Název projektu
- Typ úkolu
- Kategorie
- Cena
- Odkaz na detail úkolu

Správa úkolů

Správa úkolů bude obsahovat také 2 pohledy. Stejně jako u klientů i zde budeme mít samostatnou stránku pro seznam všech úkolů a detail.
Detail bude mít vyplňovací pole pro zadávání informací, dále bude mít jednoduchý WYSIWYG editor pro tvorbu formátovaného popisu úkolu. Bude nabízet vysunovací pole pro výběr kategorie a typu projektu. Typ úkolu určuje, jak se bude počítat výsledná částka.

- Hodinový
- Projektový
- Oprava chyby
- Nákladový

Na základě typu úkolu budou nabízeny odlišné vyplňovací kolonky. Např. pro hodinový typ budou celkově tři. Počet hodin, cena za hodinu a následně vypočítaná hodnota. U projektového a nákladového to bude jednorázová částka. Jestliže bude vybrat typ oprava chyby, cena se nebude udávat žádná.
Detail úkolu by měl také obsahovat sekci pro zadávání a nahrávání příloh s podporou *drag and drop*, u které bude jednoduchá tabulka obsahující základní informace o přílohách s ikonkou ke stažení.
Pod touto tabulkou budou dvě zaškrtávací políčka, včetně volby data. Jedno bude značit, zda je úkol dokončen a druhé, zda byl zaplacen.
Posledním požadavkem na detail úkolu je výpis činností, které jsou vztažené k danému úkolu. Tento výpis bude opět jednoduchou tabulkou obsahující jméno zaměstnance, který je k činnosti přidělen, popis činnosti, čas strávený nad danou činností a tlačítko pro přechod na detail činnosti.
Stránka pro detail úkolu je nejsložitější, proto si návrh ukážeme na obrázku 3.3.



Obrázek 3.3: Stránka pro detail úkolu

Správa činností

Stejně jako u předchozích případů i správa činností bude mít dva možné pohledy. Jeden jako jednoduchý seznam a druhý jako detail. Detail činnosti bude mít i ovládání časovačů, kterými si zaměstnanec bude moci měřit čas strávený nad prováděnou činností.

Správa zaměstnanců

Systém pro management projektu musí mít i jednoduchou správu uživatelů, tedy zaměstnanců a manažerů. Manažeři budou mít přístup k seznamu všech zaměstnanců firmy. A budou moci editovat jejich profil kromě hesla.
Zaměstnanci budou mít pouze přístup ke svému profilu.

Kapitola 4

Implementace řešení

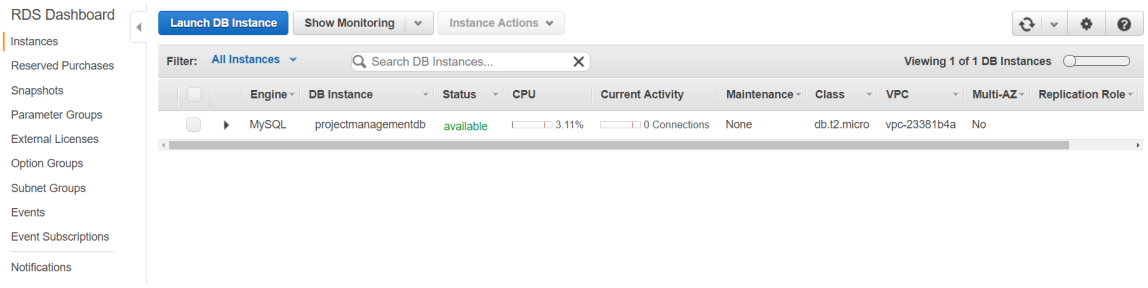
V této kapitole se opět podíváme na serverovou (4.1) a klientskou část (4.2), nyní však z pohledu implementace.
Serverová část demonstruje a vysvětlí nejdůležitější části. Zaměří se tedy hlavně na to, jak se vytváří jednotlivé instance Amazon Web Services služeb a jak je správně nastavit. Dále popíše, co všechno je potřeba nastavit v API Gateway, aby naše klientská aplikace měla přístup k jednotlivým end-pointům.
Klientská část mimo jiné popíše, jak integrovat vygenerovaný kód z AWS. Vysvětlí například použití startovacího balíčku (angl. *starter kit*) React Boilerplate pro tvorbu React aplikací a jak nasadit produkční verzi na Amazon S3.

4.1 Server

Zde si důkladně popíšeme, jak vytvořit a nastavit instanci Relational Database Service (RDS) a jak uvést do provozu S3 bucket. Sekce o AWS Lambda například vysvětlí, jak automatizovaně nasazovat jednotlivé Lambda funkce na AWS. A na závěr, jak všechno propojit skrze API Gateway.

4.1.1 Vytvoření a nastavení instance databáze

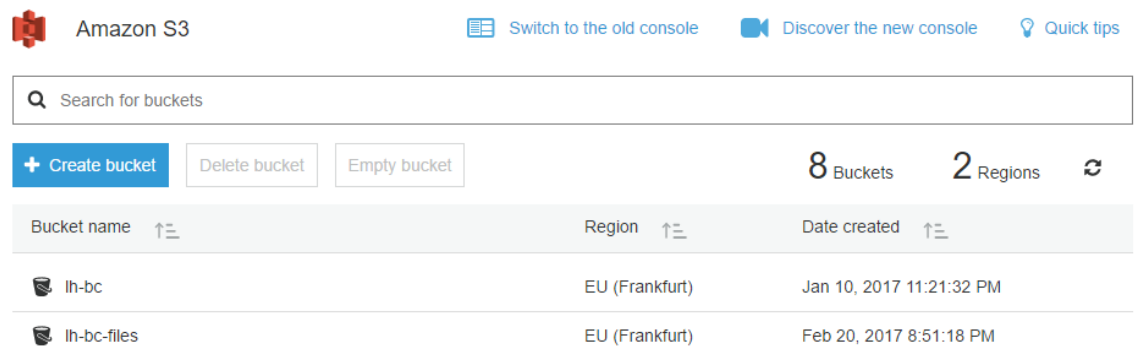
Po úspěšném přihlášení do webové konzole Amazon Web Services vybereme RDS odkaz z nabídky všech AWS služeb. Ten nás přesune do správy databázových instancí. Zde klikneme na Launch a DB instance v sekci Create Instance. A nyní jsme u prvního kroku vytváření databázové instance. V tomto kroku si volíme databázový engine. Po zvolení MySQL driveru budeme mít možnost zvolit si produkční verzi nebo development verzi (která je v rámci Free Tier).
Dalším krokem je specifikace naší databázové instance. Zde už si volíme, jakou verzi MySQL budeme používat, o jakou třídu se bude jednat (pro účely bakalářské práce zvolíme ty nejnižší, abychom zůstali v rámci Free Tier) a vyplníme název a přístupové údaje do databáze. V posledním kroku zkontrolujeme dosavadní nastavení a nastavíme pokročilejší věci jako zálohování a podobně. Pak už stačí jen kliknout na Launch DB Instance a máme hotovo.
Zda databázová instance běží můžeme zkontrolovat v monitorovací sekci (obrázek 4.1), do které se dostaneme kliknutím na DB Instances.



Obrázek 4.1: Ukázka monitorovací sekce pro AWS RDS

4.1.2 Vytvoření a nastavení S3 bucketu

Opět se nejprve musíme dostat do přehledu AWS služeb, zde tentokrát vybereme možnost S3. Po kliknutí budeme přesunutí do přehledu S3 bucketů. Klikneme na tlačítko Create bucket a objeví se wizard dialog se 4 kroky.
V prvním kroku nastavíme jméno bucketu, vybereme region, kde bude náš bucket umístěn (v našem případě Frankfurt, protože je nejblíže). Je zde i nepovinná možnost zkopírovat nastavení z již existujících bucketů.
V dalším kroku si můžeme nastavit, zda chceme objekty verzovat, logovat či značkovat.
V předposledním kroku nastavujeme přístupy AWS uživatelů, tedy jestli mají přístup do bucketu, jestli mohou z něj číst data nebo i nahrávat.
V posledním kroce máme opět přehled dosavadního nastavení, takže máme možnost zkontrolovat, zda jsme vše nastavili správně a pro vytvoření bucketu už jen stačí kliknout na tlačítko Create bucket.
Po vytvoření by se měl v přehledu bucketů ukázat nově vytvořený bucket.



Obrázek 4.2: Přehled S3 bucketů v AWS konzoli

Z obrázku 4.2 je patrné, že zde máme celkem 2 buckety. Jeden bude sloužit jako úložiště příloh a souborů (lh-bc-files), druhý jako přístupový bod pro klientskou aplikaci. Po rozkliknutí některého z bucketů se objevíme na přehledu konkrétního bucketu, obsahujícím nahrané soubory se základními informacemi jako jsou nahrané soubory a podobně.
Jak už bylo zmíněno, bucket *lh-bc* bude sloužit pro hosting naší webové aplikace. Tuto funkcionalitu nastavíme po najetí do detailu bucketu, kliknutím na sekci Properties v horním menu a rozkliknutí karty Static Website Hosting. Zde zaškrtneme, že chceme bucket využívat pro hostování webové stránky a nastavíme výchozí index dokument - tedy index.html.

Poté pokud nahrajeme aplikaci na bucket a napíšeme do prohlížeče adresu našeho endpointu pro bucket, měli bychom vidět obsah souboru index.html.

4.1.3 Vytvoření AWS Lambda funkce

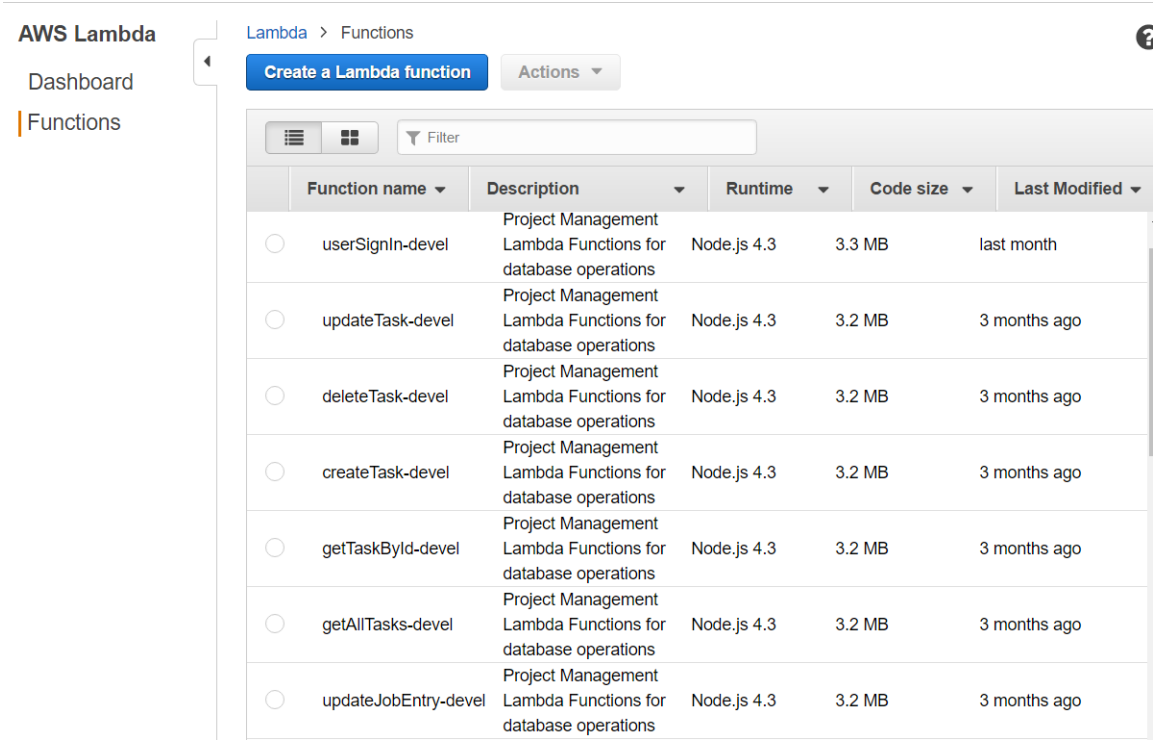
Stejně jako v předchozích případech, i Lambda má svou dedikovanou sekci, do které se dostaneme kliknutím na odkaz Lambda v sekci Compute.
Odkaz nás přesune na stránku obsahující přehled existujících Lambda funkcí. Pro vytvoření Lambda funkce klikneme na tlačítko Create a Lambda function. Zobrazí se nám nabídka různých šablon. Například po kliknutí na *Blank Function* se ocitneme v konfiguraci funkce. Kde máme možnost zadat název funkce, její popis a jazyk, na kterém poběží. Dále AWS přímo na stránce nabízí editor pro implementaci dané Lambda funkce.
V našem případě tento postup je nedostačující, protože budeme pracovat s dalšími knihovnami jako aws-sdk, mysql a object-assign a ve webovém editoru není možnost importu těchto knihoven.
Řešením je si napsat vlastní kód lokálně u sebe. A pro nasazení na AWS Lambda použít CLI nástroj *node-lambda*, který vzdáleně nahraje zkompilovaný kód na AWS Lambda. Než budeme moct posílat naše balíčky na AWS Lambda, musí se vytvořit konfigurační soubor *.env*, který bude obsahovat AWS Credentials klíče a další konfigurační informace.

```
1  AWS_ENVIRONMENT=devel
2  AWS_ACCESS_KEY_ID=      # vas AWS_ACCESS_KEY_ID
3  AWS_SECRET_ACCESS_KEY=  # vas AWS_SECRET_ACCESS_KEY
4  AWS_SESSION_TOKEN=
5  AWS_ROLE_ARN=arn:aws:iam::661593375096:role/lambda_basic_execution
6  AWS_REGION=eu-central-1 # AWS Region (Frankfurt)
7  AWS_FUNCTION_NAME=
8  AWS_HANDLER=index.handler
9  AWS_MEMORY_SIZE=128     # pridelena pamet
10 AWS_TIMEOUT=20
11 AWS_DESCRIPTION=
12 AWS_RUNTIME=nodejs4.3   # pouzity jazyk
13 AWS_VPC_SUBNETS=
14 AWS_VPC_SECURITY_GROUPS=
15 EXCLUDE_GLOBS="event.json"
16 PACKAGE_DIRECTORY=build
```

Příklad užití:

```
1  node-lambda deploy -n getAllCustomers --handler modules/customers.
       getAllCustomers
```

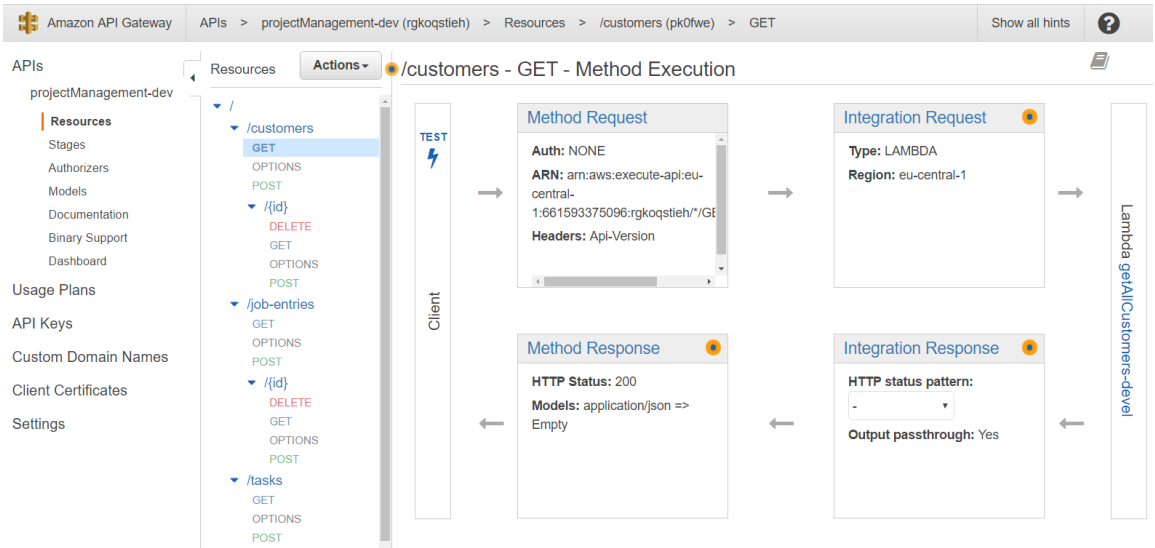
Po zadání výše zmíněného příkazu se vytvoří balíček s kódem a veškerými jeho závislostmi a v AWS Lambda se vytvoří nová funkce.



Obrázek 4.3: Přehled Lambda funkcí

4.1.4 Nastavení API Gateway

Poté, co máme připravené Lambda funkce pro práci s databází, je potřeba vytvořit API, které bude volat tyto funkce. K tomu slouží služba API Gateway.



Obrázek 4.4: API Gateway detail

Jednotlivé zdroje vytvoříme stisknutím tlačítka Actions, kde vybereme Create Resource. Vyplníme název zdroje a další věci. Poté u daného zdroje můžeme přidávat metody, tentokrát přes Create Method.

Po vytvoření zdroje a jeho metod musíme správně nastavit všechny 4 fáze komunikace:

1. *Method Request*
2. *Integration Request*
3. *Integration Response*
4. *Method Response*

V případě *Method Request* nemusíme nastavovat nic, tam by už všechno mělo být nastavené. U *Integration Request* musíme zvolit typ integrace - v našem případě Lambda Function, pak nastavit region, na kterém je naše Lambda funkce umístěna a vybrat, která Lambda funkce bude volána. V tomto kroku je nejdůležitější správně namapovat tělo requestu. My budeme používat *Content-Type: application/json*. A do editoru vložíme kód pro mapování. Dalším důležitým krokem je nastavit *Integration Response*. V tomto kroku nastavíme HTTP response kódy, poté je potřeba správně nastavit hlavičky a namapovat data do těla odpovědi. A nakonec je potřeba nastavit *Method Response*. Zde opět nastavujeme hlavičky odpovědi jako například *Access-Control-Allow-Origin* a tělo odpovědi, které bude application/json.
Jakmile máme tato nastavení hotová, musíme povolit CORS. To provedeme opět přes tlačítko Actions stisknutím Enable CORS.
Poté, co máme nastavené všechny zdroje a jejich metody, musíme provést Deploy API, které je opět obsaženo v tlačítku Actions. Po stisknutí můžeme vybrat fázi nasazení a poznámku.
Pro vygenerování SDK, které bude využívat klientská aplikace, je potřeba rozkliknout API v levém seznamu. Poté kliknout na Stages a rozkliknout SDK Generation. Zde si vybereme platformu, ve které chceme SDK mít (v našem případě JavaScript) a stiskneme Generate SDK. A nyní máme vygenerované API, které můžeme integrovat do klientské aplikace.

4.2 Klient

Tato podkapitola popíše nejdůležitější části z implementace klientské aplikace. Vysvětlí, proč jsme využili právě React Boilerplate jako startovací balíček (angl. *starter kit*) a v čem spočívají výhody těchto balíčků a jak dokáží ušetřit čas.
Vysvětlí pojem *Hot Module Replacement* (4.2.2), který je nesmírně silným nástrojem při vývoji.
Na závěr demonstruje, jak integrovat vygenerované AWS SDK pro komunikaci se serverem a hlavně, jakým způsobem se vytváří produkční verze, její rozdíl oproti vývojové a nasazení aplikace na AWS S3.

4.2.1 React Boilerplate

Existuje mnoho startovacích balíčků pro aplikace v Reactu. Jejich cílem je usnadnit konfiguraci aplikace. Hlavně tedy například, aby Babel správně převedl ES6 kód do ES5, kterému prohlížeče rozumí. Dále, aby správně webpack (4.3.3) vytvářel propojené balíčky a mnoho dalších věcí.
React Boilerplate už přímo obsahuje knihovny immutable (2.5.5), redux (2.5.4), react (2.5.3), react-router (4.3.4) a mnoho dalších, které už jsou nakonfigurované. Vývojář tedy nemusí řešit poměrně složitou konfiguraci a nastavování knihoven, aby správně fungovaly. Místo toho může v podstatě rovnou začít vyvíjet.

Nevýhodou těchto startovacích balíčků je, že pokud vývojář potřebuje upravit předem nastavenou konfiguraci, může do jisté míry narazit.
Např. máme-li modul třetí strany, který není nainstalován přes *npm* (4.3.1). Nakonfigurovaný *Babel* v React Boilerplate se bude snažit převést tuto knihovnu z ES6 do ES5, která je napsaná už v ES5, což bude vyvolávat mnoho chyb.
Zatím jsem bohužel nenašel způsob, jak tento problém efektivně vyřešit a musel jsem problém obejít nástrojem Gulp (4.3.2).

4.2.2 Hot Module Replacement (HMR)

Hot Module Replacement je jedna z vlastností Webpacku (4.3.3). Pokud jej správně nastavíme, tak se nám při každém uložení souboru se zdrojovým kódem automaticky přetranspiluje a importuje se do bundlu (balíčku), se kterým pracuje aplikace v prohlížeči. Prakticky to znamená, že při změně souboru se aktualizují zdrojové soubory a webpack informuje prohlížeč, že došlo ke změně a prohlížeč překreslí věci, které byly postiženy touto změnou. A při tom všem si aplikace zachovává stav.
Motivačním příkladem z praxe, který demonstruje úsporu času - bez Hot Module Replacement vývojář musí provést tyto kroky:

- 1 - 2x Alt + Tab, abychom se dostali z editoru na prohlížeč
- F5 nebo Ctrl + R pro znovunačtení stránky
- přibližně 1 sekunda než se načte stránka
- uvedení stránky do stavu, který chceme (může být mnoho sekund, stav aplikace se totiž zahodí)
- Alt + Tab zpět na editor

Na příkladě to vypadá jako pár sekund, ale když předchozí kroky vývojář provede stokrát až tisíckrát denně, ztráta času je velká. Při používání HMR se v prohlížeči upravené soubory nahradí novou verzí a automaticky se aktualizují, což značně urychlí vývoj aplikace.
Většina startovacích balíčků už má předem nakonfigurovanou podporu pro HMR, včetně i React Boilerplate, který používáme v naší aplikaci.

4.2.3 Integrace vygenerovaného AWS SDK

Jednoduše zkopírujeme zdrojové soubory a přesuneme do projektu. Bohužel, jak bylo již zmíněno v sekci o React Boilerplate (4.2.1) - nenašel jsem způsob, jak nastavit webpack, aby se nesnažil transpilovat SDK.
Pro vyřešení tohoto problému jsem využil nástroj Gulp (4.3.2) s dalšími jeho moduly pro minifikaci a uglifikaci těchto zdrojovým souborů a následné nahrání na S3 bucket, ze kterého si tento jeden soubor nahraváme do aplikace. SDK nám zpřístupní tovární funkci pro třídu `ApiClient`, která obsahuje vygenerované metody, které jsme vytvořili v API Gateway.
V entry pointu naší aplikace si tedy vytvoříme instanci třídy:

```
1 (() => {
2     window.apiClient = window.apigClientFactory.newClient({
3         region: config.AWS.region,
4     });
5 })();
```

A následně si můžeme implementovat např. funkci, pro získání všech zákazníků z databáze:

```
export const fetchCustomers = () => {
    return dispatch => {
        apiClient.customersGet({
            "Api-Version": config.apiVersion,
        }, {}).then(response => {
            response.data.data.forEach(customer => {
                dispatch(createCustomer({
                    id: customer.id,
                    name: customer.name,
                    data: JSON.parse(customer.data),
                }));
            });
        }).catch(error => {
            console.log('Error msg: ', error);
        });
    };
};
```

4.2.4 Nasazování produkční verze

React Boilerplate (4.2.1) přímo nabízí npm skripty pro vytvoření minifikované produkční verze, která je ochuzená o veškeré nástroje pro vývoj a je tedy daleko rychlejší a optimalizovanější. Tyto vygenerované soubory jednoduše pomocí Gulp tasku nahrajeme na S3 bucket.
Implemetace Gulp tasku pro nahrávání na S3 bucket může vypadat následovně:

```
gulp.task('deploy', function () {
    var headers = { 'Cache-Control': 'max-age=86000, must-revalidate, public' };

    var awsConfig = {
        accessKeyId: config.AWS.accessKeyId,
        secretAccessKey: config.AWS.secretAccessKey,
        params: {
            Bucket: config.AWS.params.s3bucket,
        },
        region: config.AWS.region,
    };

    var publisher = awsPublish.create(awsConfig);

    return gulp.src('./build/**')
        .pipe(publisher.publish(headers))
        .pipe(publisher.sync())
        .pipe(awsPublish.reporter());
});
```

Nejprve musíme správně zadat přístupové údaje pro AWS a o zbytek se postará modul `awsPublish` pro Gulp, který vymaže staré soubory v S3 bucketu a nahradí je novými.

4.3 Použité nástroje a knihovny

Zde si zmíníme část nástrojů, které jsem používal pro vývoj a ladění aplikace. Jsou zde i nástroje pro jednoduchou automatizaci, které značně urychlují proces vývoje a efektivitu práce programátora.

4.3.1 Node Package Manager (npm)

Jak už název napovídá, jedná o systém pro správu balíčků v platformě Node. Skrze něj můžeme jednoduše přidávat nebo odebírat knihovny do našeho projektu.

4.3.2 Gulp

Jedná se o task manager, kde si naprogramujeme věci, které budeme při vývoji opakovaně využívat jako spuštění klientské aplikace, vytvoření minifikované a uglifikované produkční verze aplikace, nasazovaní na hosting a podobně. Jeho defaultní funkcionalitu můžeme rozšiřovat mnoha dalšími moduly.

4.3.3 Webpack

Webpack je nástroj pro tvorbu balíčků u moderních JavaScriptových aplikací. Zabaluje jednotlivé CommonJS/AMD moduly do balíčků pro prohlížeče. Umožňuje rozdělení aplikační codebase do více balíčků, které mohou být poté postupně načítány.

4.3.4 react-router

Směrování mezi stránkami je řešeno knihovnou react-router, která disponuje komponentami s jednoduchým API pro snadné a deklarativní směrování v aplikaci.

4.3.5 styled-components

Styled-components je poměrně novou knihovnou pro stylování komponent v Reactu. Využívá ES6 template literals (2.5.2), díky kterým můžeme psát CSS kód přímo pro komponenty, které píšeme v JavaScriptu. Podporuje zanořování jako například LESS, SASS či Stylus.
Největším přínosem je dynamičnost - v CSS kódu máme přístup ke `props`, na jejichž základě můžeme přiřazovat různé styly. Mimo jiné podporuje i media queries a další věci, které jsou v CSS a nemohou být nahrazeny JavaScriptovými inline-styly.

4.3.6 GitHub

Pro zálohování a verzování projektů jsem využil GitHub. GitHub nabízí pro studenty mít mnoho privátních repozitářů. Já konkrétně využívám pro bakalářskou práci dva - jeden pro klientskou aplikaci a druhý pro kódy Lambda funkcí.

4.3.7 ESLint

ESLint je nástrojem, kde si nastavíme pravidla pro jednotný styl kódu a jeho kvalitu. V našem případě budeme vycházet z JavaScript Style Guide od Airbnb, který je využíván mnoha firmami a programátory na světě.

4.3.8 JSDoc

JSDoc je jazyk pro tvorbu dokumentace kódu pomocí anotací. V praxi pokud se používá například s WebStorm IDE nebo i dalšími vývojovými prostředími, tak vám IDE přímo na základě JSDoc anotací napovídá při programování.

4.3.9 WebStorm IDE

WebStorm je silným IDE vytvořeným přímo vývoj aplikací v JavaScriptu, který má i podporu pro React JSX.

4.3.10 Chrome Developer Tools

Chrome Developer Tools je souborem nástrojů pro ladění webových aplikací, které jsou integrovány v prohlížeči Chrome. Nabízí například přehled HTML struktury HTML dokumentu, přehled síťové komunikace a mnoho dalších užitečných věcí.

4.3.11 React Dev Tools

Je rozšířením do prohlížeče Chrome. Umožňuje napojení se na React aplikaci a můžeme tedy přes konzoli získat stav aplikace, pokud jsme ve vývojovém režimu nebo také sledovat strukturu našich komponent, jejich stav, props a podobně. Při vývoji React aplikací je to velmi užitečný nástroj pro ladění.

4.3.12 Redux Dev Tools

Také se jedná o rozšíření do Chrome. Tento nástroj je však určen pro Redux, kde můžeme přehledně sledovat stav aplikace v reálném čase, můžeme aplikovat i tzv. *time-travel debugging*, tzn. můžeme se vracet v čase v kontextu stavu aplikace. Můžeme také přes něj přehledně sledovat jaké akce byly spuštěny a jaké nesou informace. Opět velmi užitečný nástroj, pokud používáme knihovnu Redux pro správu stavu aplikace.

Kapitola 5

Testování

Testování je nezbytnou součástí vývoje softwarových aplikací. Cílem testování je odhalování chyb a ověřování správné funkcionality. V této kapitole je popsán postup, jakým byla výsledná aplikace testována a jaké metody či technologie byly k tomuto účelu využity. [12]

5.1 AWS Lambda

Pro testování správné funkcionality jednotlivých AWS Lambda funkcí nabízí služba API Gateway testovací rozhraní, kde máme možnost vyplnit hlavičky i těla dotazů, které poté odešleme na AWS. A testovací konzole API Gateway nám ihned vypíše podrobné údaje o dotazu, jeho zpracování a výsledku.

5.2 Klient

Balíček React Boilerplate 4.2.1 poskytuje knihovny pro jednotkové a integrační testování jako jsou *mocha*, *chai*, *karma* a další. Pro účely mého testování jsem využíval nástrojů React Dev Tools 4.3.12 a Redux Dev Tools 4.3.12, které jsou pro vývoj ukázkové aplikace dostačující.

5.3 Uživatelské (end-to-end) testování

Celkové testování systému probíhalo s koncovými uživateli. Tito uživatelé dostali úkoly, které mají v dané aplikaci udělat. Během testování jsem jednotlivé uživatele sledoval, jak rychle jsou schopni splnit dané úkoly a zda se uživatelé během práce nezasekli na některé z částí. Na závěr uživatelského testování vždy proběhla konzultace s uživatelem, který odpovídal na otázky, jaký měl pocit z užívání aplikace a zda-li vše bylo jednoduché na pochopení a intuitivní.

Kapitola 6

Závěr

V této práci je zpracována studie cloudových služeb včetně možností jejich využívání v praxi. Studie představuje současné přední poskytovale cloudových služeb a porovnává je mezi sebou.
Práce mimo jiné představila a porovnala nejpoužívanější knihovny a frameworky pro tvorbu klientských webových aplikací. A následně poskytla fundamentální informace pro používání ECMAScript 2015, React a Redux s ImmutableJS.
Bakalářská práce poskytuje náhled do procesu návrhu a implementace klientské aplikace v ReactJS pomocí jednoduchých znovupoužitelných komponent, které shlukujeme do složitějších celků s rozdílnou funkcionalitou a vysvětluje základní typy komponent.
U ukázkové aplikace byly úspěšně propojeny zmíněné cloudové služby od Amazon Web Services jako AWS Lambda s API Gateway, se kterou komunikuje klientská aplikace. Jednotlivé metody API Gateway byly integrovány do klientské aplikace pomocí vygenerovaného SDK, čímž vznikl fungující celek, který demonstruje, jak se technologie vzájemně doplňují.
Z hlediska dalšího vývoje aplikace se nabízí mnoho možností. Jednou z nich je vytvořit tuto aplikaci i pro mobilní zařízení využitím React Native a implementovat jejich synchronizaci se serverovou částí. Aby uživatelé či zaměstnanci firmy měli možnost si například hlídat a spouštět časovače na mobilu. S tím by mohl souviset i vývoj responzivní verze webové aplikace, aby vypadala strukturovaně a uceleně na jakémkoli zařízení.

Literatura

[1] Abramov, D.: *Presentational and Container Components*. Medium, Březen 2015, [Online; navštíveno 30.04.2017].
URL
https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

[2] Akiwatkar, R.: *Cloud Computing: Let's Keep it Simple*. Medium, Březen 2016, [Online; navštíveno 30.04.2017].
URL https://medium.com/@RohitAkiwatkar/cloud-computing-lets-keep-it-simple-5402c246be66

[3] Bassett, L.: *Introduction to JavaScript Object Notation*. O'Reilly Media, 2015, ISBN 978-1-491-92948-3.

[4] Colangelo, A.: *Google Cloud vs AWS: a comparison*. Cloud Academy, Říjen 2014, [Online; navštíveno 30.04.2017].
URL http://cloudacademy.com/blog/google-cloud-vs-aws-a-comparison/

[5] Crockford, D.: *JavaScript: The Good Parts*. O'Reilly Media, 2008, ISBN 978-0-596-51774-8.

[6] Elliot, E.: *Master the JavaScript Interview: What is a Pure Function?* Medium, Březen 2016, [Online; navštíveno 30.04.2017].
URL https://medium.com/javascript-scene/reduce-composing-software-fe22f0c39a1d

[7] Elliot, E.: *Master the JavaScript Interview: What is a Promise*. Medium, Leden 2017, [Online; navštíveno 30.04.2017].
URL https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-promise-27fc71e77261

[8] Elliot, E.: *Reduce (Composing Software)*. Medium, Březen 2017, [Online; navštíveno 30.04.2017].
URL https://medium.com/javascript-scene/reduce-composing-software-fe22f0c39a1d

[9] Khourshid, D.: *An Animated Intro to RxJS*. CSS Tricks, Únor 2017, [Online; navštíveno 30.04.2017].
URL https://css-tricks.com/animated-intro-rxjs/

[10] Linthicum, D. S.: *Cloud Computing and SOA Convergence: Where We Are, How We Got Here, and How to Fix It*. InformIT, Srpen 2009, [Online; navštíveno 30.04.2017].
URL http://www.informit.com/articles/article.aspx?p=1398772&seqNum=6

[11] M, V. A.; Sonpatki, P.: *ReactJS by Example - Building Modern Web Applications with React*. Packt Publishing, 2016, ISBN 978-1-78528-964-4.

[12] Miroslav Bureš, M. D., Miroslav Renda; aj.: *Efektivní testování softwaru*. Grada Publishing, 2016, ISBN 978-80-247-5594-6.

[13] Prusty, N.: *Learning ECMAScript 6*. Packt Publishing, 2015, ISBN 978-1-78588-444-3.

[14] Zakas, N. C.: *Understanding ECMAScript 6*. Leanpub, Březen 2017, [Online; navštíveno 30.04.2017].
URL https://leanpub.com/understandinges6/read/

Přílohy

Příloha A

Obsah DVD

- readme.txt soubor s pokyny ke spuštění aplikace, url adresou, kde běží produkční verze aplikace. Soubor dále obsahuje seznam použitých knihoven a jsou zde uvedeny chybějící části v aplikaci.
- /src adresář se zdrojovými soubory jak klientské tak i serverové aplikace
- /thesis_pdf adresář s písemnou zprávou ve formátu *Portable Document Format*
- /thesis_src adresář se zdrojovými kódy písemé práce psané v LaTeX