Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Implementace aplikace na plánování volného času

Bakalářská práce

Autor: Marek Laušman
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Michal Macinka

Hradec Králové duben 2020

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hradci Králové dne 30.4.2020 Marek Laušman

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Michalu Macinkovi za metodické vedení práce.

Anotace

Cílem této bakalářské práce je představit případnému čtenáři technologie použité pro vývoj single page aplikace na plánování volného času – PlanApp, které jsou předmětem tohoto projektu. Práce se tedy konkrétně zaměřuje na popis základních vlastností a funkcí frameworku ReactJS, popisem knihovny Redux, komunikace mezi Reactem a Reduxem, backendové technologie Spring Framework a na následný návrh a implementaci samotné aplikace. Práce je rozdělena na čtyři hlavní části. První část se věnuje obecně vývoji aplikací. Druhá část se zaobírá použitými frontendovými a backendovými technologiemi. Třetí část se týká praktické části, a to konkrétně návrhu databázové struktury. Ve čtvrté části se již řeší konkrétní problémy vzniklé při implementaci aplikace.

Annotation

Title: Implementation of Free-time Planning App

The goal of this thesis is to introduce the eventual reader to technologies used for bachelor's practical project development, which is a single page application. Specifically, it is a free-time planning application – PlanApp. The thesis's main focus is the description of some basics of the ReactJS Framework, Redux library, communication between React and Redux, backend technology Spring Framework, database design, and the implementation itself. The thesis is divided into four main parts. The first part is generally about web app development. The second one is a chapter about used frontend and backend technologies. The last but not least part describes database structure design and the last one is about specific implementation issue.

Obsah

1 Úvod ... 1

2 Cíl práce ... 2

3 Vývoj webových aplikací ... 3

3.1 Výhody a nevýhody webových aplikací ... 3

3.2 Single page aplikace ... 3

3.3 Multi page aplikace ... 5

3.4 Progresivní webové aplikace ... 6

4 Přehled platforem ... 8

4.1 ReactJS ... 8

4.1.1 Popis a srovnání Virtuálního DOM ... 8

4.1.2 JSX – syntaxe Reactu ... 9

4.1.3 Komponenty v ReactJS ... 9

4.1.4 Funkce ... 13

4.1.5 State a Props ... 14

4.1.6 Komunikace s API třetích stran ... 15

4.2 Architektura MVC ... 16

4.2.1 Výhody a nevýhody architektury ... 17

4.3 Spring Framework ... 19

4.3.1 Spring moduly ... 19

4.3.2 Inversion of Control (IoC) ... 20

4.3.3 Dependency Injection (DI) ... 21

4.3.4 Anotace ... 22

4.4 Redux ... 25

4.4.1 Hlavní principy 25
4.4.2 Store 26
4.4.3 Actions 26
4.4.4 Reducers 27
4.4.5 Komunikace React + Redux 28
5 Návrh aplikace 29
5.1 Návrh databázové struktury 29
5.2 Use-Case diagram 30
5.3 Funkční a nefunkční požadavky 31
5.4 Drátový model aplikace 32
6 Implementace aplikace 34
6.1 Architektura aplikace 34
6.2 Heroku 34
6.3 Práce s mapami 35
6.4 Finální vzhled aplikace 38
7 Závěr 40
8 Seznam použité literatury 41
9 Seznam obrázků 44
10 Seznam použitých ukázek kódu 45

1 Úvod

Webové aplikace jsou v dnešní době velikým trendem jak pro IT specialisty, tak pro lidstvo jako takové. Víceméně každý je dnes připojen k internetu, takže aplikaci lze pustit kdykoliv a kdekoliv. Tyto webové aplikace se používají hlavně pro přenos informací a práci s nimi. Aplikace se skládají z backendu, frontendu a databáze. Frontendových i backendových technologií existuje v dnešní době nespočet, nicméně tato práce se hlavně zabývá velmi populární technologií poslední doby, frameworkem ReactJS a backendovou technologií Spring Framework. Konkrétně se práce zabývá popisem, výhodami a nevýhodami v použití apod.

Důvodem vypracování této práce byly zkušenosti se zmíněnými technologiemi, popularita frameworků, a hlavně vývoj bakalářského praktického projektu, kterým je webová aplikace na plánování volného času – PlanApp, která využívá základních principů fungování Reactu a Springu. Mezi základní principy ReactJS patří například rozdělení komponent, popis state a props, princip VDOM nebo například komunikace s API třetích stran. Základními principy Spring Frameworku je zase myšlen základní popis vkládání závislostí (DI) a obráceného řízení (IoC), popis frameworku a anotace.

2 Cíl práce

Cílem bakalářské práce je navrhnout a implementovat webovou single page aplikaci na plánování volného času za využití frontendové technologie ReactJS, backendové technologie Spring Framework a databáze MySQL. Aby byl cíl splněn, je potřeba si vytvořit představu o databázové struktuře a základních funkčních a nefunkčních požadavcích. Pro tento návrh byl použit SW pracující s jazykem UML. V tomto případě se primárně jedná o diagram tříd a diagram případů užití. Následně je již samotná implementace aplikace a vložení aplikace na Heroku, aby byla aplikace volně dostupná a fungující bez jakéhokoliv omezení pro uživatele. Výsledkem je tedy webová aplikace plánující volný čas uživatele podle jeho lokace, kde se právě nachází.

3 Vývoj webových aplikací

Webová aplikace je jednoduše jakýkoliv software, který běží na internetu. Funguje na bázi klienta a serveru, kdy klient komunikuje se serverem za účelem získání či uložení dat. Webová aplikace nepotřebuje žádnou instalaci nebo .exe soubor. Je volně přístupná všem uživatelům, kteří mají přístup na internet. Pokud je aplikace responzivní, lze ji dokonce používat na jakémkoliv zařízení, které má možnost připojení k internetu.

3.1 Výhody a nevýhody webových aplikací

Výhody

Posledních pár let velmi stoupá popularita webových aplikací právě díky zásadním výhodám oproti desktopovým aplikacím. Mezi hlavní výhody webových aplikací patří [1]:

- Nemusí se generovat .exe soubor a díky tomu se nemusí instalovat.
- Uživatel sám nemusí aplikaci aktualizovat – aktualizace probíhá pouze na serveru.
- Data jsou přístupná odkudkoliv – jsou uložená na serveru.
- Na aplikaci se lze připojit skrze jakékoliv zařízení disponující připojením k internetu.

Nevýhody

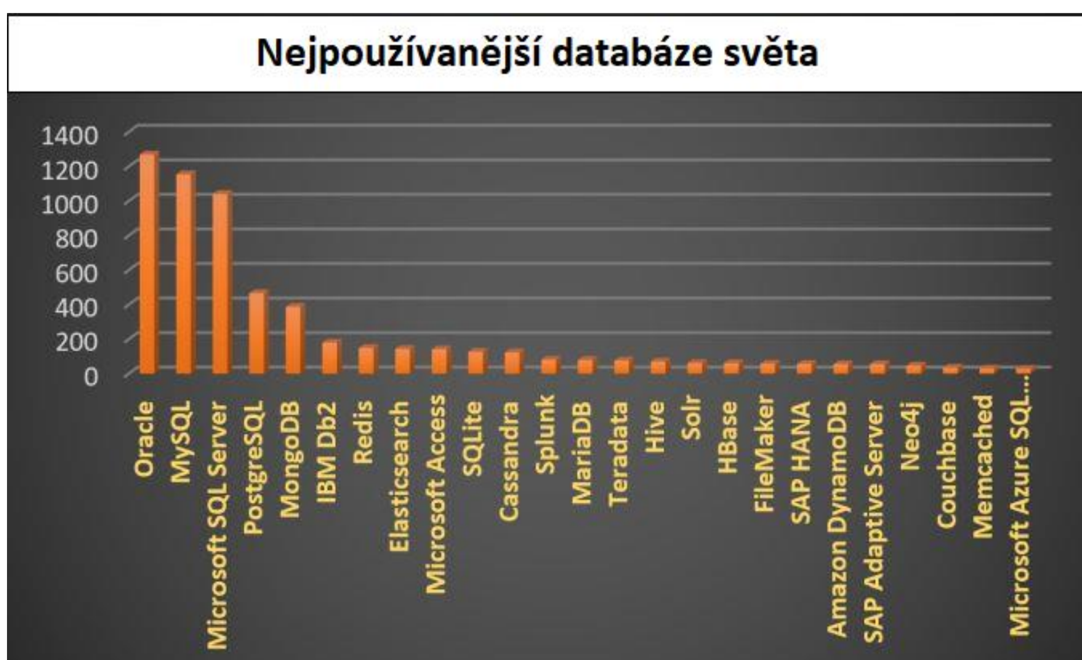
I webové aplikace mají některé nepřehlédnutelné nevýhody, které v rámci možností omezují použití. Hlavními nevýhodami jsou [1]:

- Aplikace vyžaduje připojení k internetu.
- V závislosti na připojení k internetu někdy aplikace nemusí pracovat správně anebo může pracovat velmi pomalu v omezeném režimu.
- Nejzávažnější nevýhodou je riziko zabezpečení. Je nutné mít kvalitního poskytovatele a kvalitní zabezpečení, jinak bude například hrozit únik citlivých dat z databáze.

3.2 Single page aplikace

V dnešní době rychlého internetu není potřeba po každé akci volat a vykreslovat novou stránku. Tento problém je řešitelný tzv. jednostránkovými aplikacemi (SPA). SPA jsou v dnešní době velkým trendem. Jsou to webové aplikace, ve kterých se nevykresluje každá

nová stránka, ale Javascript umí dynamicky měnit elementy v DOM (rozhraní umožňující přistupovat k jednotlivým prvkům HTML dokumentu) a tím měnit obsah webové aplikace. Výhodou je, že není nutné vždy měnit celou stránku, ale stačí jen aktualizovat požadované elementy [2]. Následkem tohoto způsobu je bezpochyby rychlejší a efektivnější chod celé aplikace. SPA je možné implementovat s frontendovou technologií, nicméně je možné použít i backendovou technologii pro práci s databází a zvolit vyhovující typ databáze k specifikovaným potřebám. Nejznámějšími technologiemi pro vývoj frontendu SPA je ReactJS, VueJS, AngluarJS nebo EmberJS. Pro backend je možné použít například Spring Framework nebo NodeJS. Pro databázi pak nejčastěji Oracle, MySQL nebo Microsoft SQL Server.



Obrázek 1 Graf nejpopulárnějších databází světa. Zdroj: [3]

Výhody

- Rychlá odezva – Jak již bylo zmíněno, při každé sebemenší změně není potřeba znovu načíst celou stránku, ale jen se aktualizuje konkrétní element. Rychlost je velmi důležitá, protože stránka by se neměla načítat déle než 200 milisekund.
- Jednoduchý debugging – Do prohlížeče Google Chrome lze jednoduše nainstalovat všelijaká rozšíření, která jsou velmi nápomocná při vývoji. Například React developer tools, za předpokladu, že se používá ReactJS.

- Uživatelská přívětivost a responzivní design – Jelikož je SPA webová aplikace, poběží bez starostí i v prohlížečích na mobilu [4].

Nevýhody

- Špatná SEO (optimalizace pro vyhledávače) optimalizace – Nelze cílit na jednotlivá klíčová slova, jelikož obsah se mění dynamicky podle dotazů klienta na server. Naštěstí Google vytvořil algoritmus, který indexuje i dynamické stránky**.**
- Poměrně složité na vývoj – Nutné znát fungování například NPM nebo YARN, Webpack, backendové a frontendové technologie, a mnoho dalších technologií.
- Slabá schopnost pro sdílení odkazů – Existuje pouze jeden vstupní bod sloužící pro vstup do aplikace [5].
- Nezbytné povolení Javascriptu v prohlížeči

3.3 Multi page aplikace

Multi page aplikace (MPA) neboli vícestránkové aplikace, fungují na principu nového vykreslování celé stránky pokaždé, když nastane změna. To znamená například zobrazení jiných dat nebo třeba odeslání dat z formuláře přímo na server. Základními technologiemi používanými pro vývoj MPA jsou HTML, CSS a Javascript nebo jQuery. MPA jsou mnohem komplexnější a větší než SPA. MPA je perfektní technologií pro společnosti, které nabízejí velmi široký sortiment produktů nebo služeb a potřebují například víceúrovňové menu nebo další speciální vlastnosti. Celkově se MPA hodí více na vývoj internetových obchodů či různých katalogů než SPA [6].

Výhody

MPA mají i v dnešní urychlené době řadu výhod. Mezi výhody konkrétně patří [7]:

- SEO (optimalizace pro vyhledávače) – Celá architektura MPA je nativní pro vyhledávače. MPA má jednoduše větší šanci na zobrazování na více různých klíčových slov, protože každá stránka může být orientována na jiné klíčové slovo.
- Neomezená škálovatelnost – MPA umožňuje vytvořit nový obsah a s tím vytvořit novou stránku přímo pro ten obsah. Jak bylo již zmíněno, toto je výhoda zejména pro společnosti, které potřebují stále přidávat nové a nové produkty.

- Google Analytics – Z MPA lze získat řadu důležitých informací pro internetový marketing. Například jaké speciální vlastnosti aplikace obsahuje, jaké ne. SPA v tomto ohledu poměrně zaostává a lze měřit pouze kdo aplikaci navštěvuje a jak dlouho návštěvníci v aplikaci zůstali.

Nevýhody

V poslední době popularita MPA z pohledu vývoje mírně zaostává. Disponuje nevýhodami, které například SPA řeší [7].

- Pomalý chod a nízký výkon – Jak již bylo zmíněno, s každou interakcí je nutné znovu načíst dané stránky (tzn. načíst celé HTML, CSS a případně skripty). Tím je způsobena pomalá rychlost a práce aplikace.
- Čas na vývoj aplikace – Ve srovnání se SPA je potřeba mnohem více času na vývoj, což je v dnešní době velká nevýhoda.
- Správa – Například z pohledu zabezpečení je nutné každou jednotlivou stránku individuálně zabezpečit. Podobných problémů nastává mnohem více. A čím je aplikace větší, tím hůře se bude spravovat a aktualizovat její obsah.

3.4 Progresivní webové aplikace

Technologie progresivní webové aplikace (PWA) je prozatím ještě v počátcích, nicméně její popularita každým okamžikem stále více a více roste. PWA je jednoduše řečeno webová aplikace chovající se jako aplikace mobilní, nicméně kombinuje vlastnosti jak z pohledu webové aplikace, tak z pohledu mobilní aplikace. Aplikace jsou programovány primárně za použití HTML a CSS, Javascriptu, Reactu, nebo Angularu. Nabízí výbornou uživatelskou přívětivost a není potřeba instalovat či stahovat programy. Přes PWA lze posílat takzvané "push notifications", což jsou vyskakovací notifikace ve webovém prostředí. Dále poskytují uživatelům přístup i v offline verzi, kde bude fungovat vše, s čím se pracovalo do doby, než byl odpojen internet. PWA běží na jakémkoliv zařízení a lze ji dokonce nainstalovat na plochu zařízení jako odkaz na danou stránku [8] [9].

Výhody

- Dosah – Jako SPA i MPA běží na internetu, nicméně jak již bylo zmíněno výše, PWA může fungovat i bez internetu, to znamená, že se velmi hodí například pro místa/země se špatným dosahem internetového připojení.
- Rychlost a menší velikost – Stáhnutí z prohlížeče na plochu je velmi jednoduché a rychlé, a nevyžaduje tolik volného místa na úložišti jako nativní aplikace stažené z Google Play (obchod pro Android) nebo App Store (obchod pro Apple).
- Offline režim – Při vypadnutí připojení k internetu lze s aplikací nadále pracovat. Dokonce i když spadne celý server, pro uživatele se nic nemění. To je zásadní benefit pro dokončení konverzí zákazníků.
- Neexistují žádná omezení – Na Google Play nebo Apple Store existují různá omezení, která když daná aplikace nesplňuje, nebude umístěna na již zmíněné obchody s aplikacemi. Ve webovém prohlížeči žádná taková omezení neexistují.
- Notifikace – Jak již bylo zmíněno výše, PWA nabízejí vyskakovací notifikace přímo ve webovém prohlížeči, což je velký benefit zejména pro online marketing, kde lze zákazníky "popostrčit" v akci, či jim připomenout cokoliv ohledně jejich nákupu.

Nevýhody

- Limitovaná podpora prohlížečů – PWA nefungují na starších zařízeních, která mají starší verze prohlížečů.
- Špatná kompatibilita s iOS – Dříve nebylo možné fungování PWA na zařízeních se systémem iOS, nicméně od verze 11.3 je možné PWA bez problému používat. Starší verze systémů ale bohužel PWA nepodporují a nejspíš ani podporovat nebudou.
- Javascript – Protože jsou PWA psány v jazyku Javascript, který je jednovláknový, chod nativních aplikací psaných například ve Swiftu bude rychlejší. Některé důležité funkce jsou stále nepodporované. Patří mezi ně například Bluetooth a různé snímače v zařízení [9].

4 Přehled platforem

V dnešní době existuje nespočet frameworků, knihoven a backendových technologií pro efektivní vytváření jednostránkových aplikací. Mezi nejznámější frontendové technologie patří právě ReactJS, který je rozebrán v následující kapitole. Backendových technologií existuje také mnoho, nicméně tato kapitola se bude hlavně zabývat popisem technologií ReactJS, Spring Framework, Redux a MVC architektury.

4.1 ReactJS

React je populární Javascriptová knihovna používaná pro vytváření uživatelských rozhraní [10]. Principiálně je React založen na skládání malých částí kódu, takzvaných komponent, do větších, komplexnějších uživatelských rozhraní. Komponenty jsou reaktivní, což jednoduše znamená, že jsou responzivní vzhledem k měnícímu se stavu. Technologie je dnes velmi populární zejména pro tvorbu single page aplikací, nebo mobilních aplikací. Hlavním důvodem popularity frameworku je jeho vysoký výkon, který je způsoben využitím virtuálního DOM, a především jeho reaktivnost.
React byl uveden na trh v roce 2013 zaměstnancem Facebooku, Jordanem Walkem. Dodnes je stále vyvíjen a spravován společností Facebook. Dnes v produkčním prostředí používá technologii například technologický gigant Facebook, Netflix, Airbnb či Twitter [11].

4.1.1 Popis a srovnání Virtuálního DOM

DOM (Document Object Model – objektový model dokumentu) je rozhraní umožňující přistupovat k jednotlivým prvkům HTML dokumentu. Prvky jsou myšleny elementy, tagy, atributy apod. DOM umožňuje přidat jakýkoliv obsah na jakékoliv místo v HTML pomocí Javascriptu. DOM má stromovou strukturu obsahující jednotlivé uzly. Při vývoji DOM nebyl brán zřetel na používání v dynamickém uživatelském rozhraní. Bylo by časově velmi náročné, a ne příliš efektivní spravovat DOM s několika sty či tisíci uzly.
VDOM (Virtual Document Object Model – Virtuální Objektový Model Dokumentu) je programovací koncept založený na uložení virtuální reprezentace uživatelského rozhraní do paměti a poté synchronizace se skutečným DOM – konkrétně knihovnou ReactDOM [10]. Byl navržen jako vylepšený, rychlejší a efektivnější DOM. Je to nadstavba nad skutečným DOM. Virtuální DOM lze měnit jakýmkoliv způsobem. Po změně se VDOM

a DOM porovnávají a hledají odlišnosti, které nastaly během práce. Pokud odlišnost najdou, změní se pouze ta část DOM, která má být změněna. Nemusí se tedy zbytečně překreslovat celý DOM při každé sebemenší změně. Překreslení celého DOM by byl velmi složitý a časové náročný proces, což je v dnešní urychlené době nepřípustné. Virtuální DOM je strom uzlů, který pracuje s elementy a jejich atributy jako s objekty a vlastnostmi. Metoda *render* vytvoří uzlový strom ze všech existujících komponent a aktualizuje ho pokaždé, když dojde ke změně v datovém modelu [12]. Jinými slovy, je nutné v ReactJS specifikovat v jakém stavu se mají zobrazovat vytvořené komponenty a virtuální DOM se o to už postará.

4.1.2 JSX – syntaxe Reactu

JSX je rozšíření Reactu, které nemusí být používáno, nicméně sám React doporučuje psát jednotlivé komponenty s touto nadstavbou. Syntaxe JSX transformuje HTML text na standardní Javascript objekty. Psaní kódu v JSX je podobné jako psaní kódu v XML. Díky tomu je možné jednoduše definovat stromovou strukturu. Je možné vkládat elementy přímo do proměnných. Níže uvedený příklad demonstruje ukázku kódu za použití JSX přímo v praxi.

```
Const name = "jmeno";
<div> <h1> Hello, {name}</h1> </div>
```

Ukázka kódu 1 Příklad použití JSX. Zdroj: [autor]

Kód bude transformován následovně:

```
var nav = React.createElement
      ("ul", {id: "nav",},
      React.createElement("li", null, …)
)
```

Ukázka kódu 2 Tansformovaný kód ukázky 1. Zdroj: [autor]

Lze tedy konstatovat, že používání JSX ulehčí mnoho práce a kód je čistější a v neposlední řadě i lépe formátovaný.

4.1.3 Komponenty v ReactJS

Jednoduše řečeno jednotlivé komponenty jsou menší, znovupoužitelné Javascriptové funkce. V ReactJS existuje více typů komponent. V této kapitole budou rozebrány

funkcionální komponenty, komponenty typu třídy, řiditelné a neřiditelné komponenty a kompozice komponent.

Funkcionální komponenty

Výstupem těchto funkcí jsou prvky, které se mají zobrazit na obrazovce. Komunikují spolu pomocí takzvaných *vlastností (props),* skrze které si komponenty předávají informace. Přijímají číselný vstup, textový vstup a mnoho dalších. Přes props je možné přistupovat k jednotlivým informacím. Funkcionální komponentu lze napsat dvěma styly. Níže uvedený kód zobrazuje obě varianty [10].

```
function Welcome(props) {
      return <h1>Hello, {props.name} </h1>
}
const Welcome = (props) => {
      return <h1>Hello, {props.name} </h1>
}
```

Ukázka kódu 3 Typy zapsání funkcionální komponenty. Zdroj: [autor]

Komponenta typu třídy

Třídy jsou prakticky to samé jako funkcionální komponenty, jen mají jinou syntaxi. Dříve byl mezi těmito komponentami velký rozdíl. Pouze v komponentě typu třídy se daly používat metody životního cyklu, konstruktor a lokální stav, nicméně od příchodu takzvaných hooků to již není pravda. V této komponentě lze specifikovat metodu konstruktor, která slouží k prvotní inicializaci dat dané třídy, například stavu. Třídu lze definovat dvěma způsoby. Prvním způsobem je deklarovaná třída, kterou popisuje níže uvedený kód [13].

```
class Person extends React.Component{
      constructor (props) {
            name: props.name
      }
}
```

Ukázka kódu 4 Deklarovaná třída. Zdroj: [autor]

Druhým způsobem je výrazová třída, která vypadá takto:

```
let person = class NewPerson{
      constructor(props) {
            name: props.name
      }
}
```

Ukázka kódu 5 Výrazová třída. Zdroj: [autor]

Aby bylo v jednotlivých komponentách možné využít takzvané metody životního cyklu (life-cycle methods), je zapotřebí třídu dědit z *React.Component,* nebo React.PureComponent. Dají se použít kdykoliv během života dané komponenty. Mezi metody životního cyklu patří [10]:

- **constructor()** – metoda, která je zavolána předtím, než je daná komponenta vložena do uzlového stromu DOM. Je používána pro počáteční naplnění stavu a „bindování" metod. Když nebude funkce svázaná (bind), tak se k ní nedá dostat skrze *this.* React doporučuje zavolat metodu *super(props)* pro získání *props*.
- **componentDidMount()** – metoda, která je provedena hned, jakmile je komponenta vložena do uzlového stromu DOM. Používá se například při načtení (fetch) dat z API.
- **componentWillUnmount()** – metoda, která je provedena předtím, než je komponenta odstraněna z DOM, nebo zničena. Není vhodné například volat metodu *setState ()*, protože komponenta bude zničena, tudíž nebude už znovu vykreslena.
- **componentDidUpdate()** – metoda, která je vykonána ihned po provedení aktualizace DOM. Je možné zavolat metodu *setState ()*, nicméně je nutné stanovit podmínku, protože jinak se začne vykonávat nekonečný cyklus.
- **shouldComponentUpdate()** – použitím této metody se React dozví, zda byla zasažena komponenta změnou ve *state* nebo *v props*. Metoda je tedy zavolána poté, co dostane *state* nebo *props* a před samotným vykreslením komponenty.

Následující metody s příchodem Reactu verze 17 dostaly před název slovo UNSAFE. Tyto metody jsou již označeny jako *deprecated* a není tedy doporučeno je používat [10]:

- **UNSAFE_ComponentWillReceiveProps()** – metoda, která je volána před získáním nových *props*. Pokud ale rodičovská komponenta způsobí znovu-vykreslení potomka, metoda se provede i bez obdržení *props*.
- **UNSAFE_ComponentWillUpdate()** – metoda je volána před vykreslením komponenty, ale až po obdržení nového *state* nebo *props*.

- **UNSAFE_ComponentWillMount() –** metoda, která je zavolána ještě před samotnou funkcí *render*. React doporučuje používat metodu *constructor* pro počáteční
inicializaci stavu.

Řiditelné komponenty (Controlled components)

Řiditelnou komponentou je komponenta, která si spravuje svůj vlastní stav. To znamená, že si řídí aktivitu ve svém vykresleném prvku, například ve formu. Stav je nutné inicializovat již v konstruktoru. Stav může být aktualizován v komponentě pouze pomocí metody *setState()* [14].

```
constructor(){
      this.state=({name: “jmeno”})
}
```

Ukázka kódu 6 Inicializace stavu. Zdroj: [autor]

Neřiditelné komponenty (Uncontrolled components)

V neřiditelné komponentě jsou hodnoty z jednotlivých vstupních elementů uloženy přímo v DOM, ne v komponentě, kde se dané vstupní elementy nacházejí. Připomínají tedy spíše klasické HTML vstupní elementy. Je potřeba v konstruktoru vytvořit *ref (odkaz)* na daný vstup [14] [15].

```
constructor(){
      this.input = React.createRef()
}
```

Ukázka kódu 7 Neřiditelná komponenta za použití Ref. Zdroj:

A poté se odkázat na *this.input* v atributu *ref*:

```
<input type="“email“" ref={this.input} />
```

Ukázka kódu 8 Vstup za použití Ref. Zdroj: [autor]

Kompozice komponent

Kompozice je jeden ze vzorů modelu komponent. Komponenty mohou odkazovat na jiné komponenty. To znamená, že přes jednotlivé komponenty lze přistupovat k dalším komponentám. Díky tomu je v Reactu možné používat stejnou abstrakci jednotlivých komponent pro různé úrovně detailu. V Reactu se pracuje i s prvky jako je *form, dialog nebo button,* jako s jednotlivými komponentami. Tento model komponent umožňuje vývojářům poměrně jednoduše psát znovupoužitelné komponenty [16].

4.1.4 Funkce

Arrow funkce neboli šipkovitá funkce je jeden ze zápisů funkcí v ES6 syntaxi. Značí se znakem "=>". Hlavním rozdílem oproti klasickému zápisu funkce v syntaxi ES5 je, že v šipkovité funkci nelze pracovat s příkazem `this` (příkaz, kterým se lze například dotazovat na instanci nějaké proměnné v dané třídě, kde se `this` použije). Hodnota `this` dané funkce pochází přímo od rodiče. Hodnota `this` v klasických funkcích se ale mění podle toho, kdy, kde a jak je funkce volána. Šipkovité funkce používají hodnotu `this` pouze ve svém lexikálním rozsahu. To způsobuje odlišné chování obou funkcí. React doporučuje používat tento typ zápisu funkce všude, kromě konstruktoru nebo kde je nevyhnutelná práce s příkazem `this` [17].

```
//klasická funkce
      let writeSomething = function(something) {
      console.log(something)
      }
//šipkovitá funkce
      let writeSomething = (something)=> {console.log(something)}
```

Ukázka kódu 9 Klasická a šipkovitá funkce. Zdroj: [autor]

Asynchronní funkce

Asynchronní funkce v Reactu je prakticky to samé, co *Promise* (Promise reprezentuje dokončení nebo selhání asynchronní funkce a její výslednou hodnotu). V některých situacích se z jednoduchého *promise* může stát dlouhý, složitý kód plný *then* bloků. To v podstatě řeší zápis asynchronní funkce. Výhodou asynchronních funkcí je čistota, čitelnost a jednoduchost kódu. Práce s chybami v asynchronní funkci je velmi jednoduchá. Stačí odchytávat jednotlivé chyby pomocí *try/catch* bloků. Pro použití asynchronní funkce a tím pádem příkazu *await,* je nutné označit funkci jako asynchronní použitím příkazu *async*. Syntaktický rozdíl mezi *promise* a asynchronní funkcí uveden níže:

```
      //Promise
fetch("http://localhost:8081/api/user")
.then(res => res.json())
.then(json => this.setState({ data: json }))
```

Ukázka kódu 10 Použití promise/then. Zdroj: [autor]

```
      //Asynchronní funkce
export const fetchApi = () => async => {
      const response = await
fetch("http://localhost:8081/api/user")
      const json = await response.json()
      this.setState({ data: json })
}
```

Ukázka kódu 11 Použití async/await. Zdroj: [autor]

4.1.5 State a Props

Stav komponenty

State neboli stav vyjadřuje stav komponenty. Komponenta si uchovává informace o sobě ve svém stavu. Stav je nutné inicializovat již na počátku vytvoření komponenty v tzv. konstruktoru.

```
Class Person extends React.Component {
      constructor () {
      super ();
      this.state = ({age: 30})
      }
}
```

Ukázka kódu 12 Vytvoření a inicializace stavu. Zdroj: [autor]

V uvedeném příkladu je stav inicializován napevno. Nicméně konstruktor může získat *props*, díky čemuž je kód mnohem flexibilnější a daná komponenta může pracovat s přiřazenými daty více dynamičtěji. Stav je možné v životním cyklu komponenty jakkoliv měnit pomocí metody *this.setState()*. K informacím uloženým ve stavu se přistupuje přes příkaz *this.state.something*. To znamená, že například zobrazení věku u osoby vyobrazené výše vypadá přibližně takto:

```
<h1> {this.state.age} </h1>
```

Ukázka kódu 13 Použití hodnoty ve stavu. Zdroj: [autor]

Props

React *properties* neboli *props,* jsou základní stavební jednotkou pro psaní znovupoužitelného kódu. Myšlenkou je, aby se daly psát komponenty použitelné na více místech aplikace. Komponenty dané vlastnosti získají pouze přes svoje rodičovské komponenty a jednotliví potomci potom mohou s *props* libovolně pracovat podle svých potřeb. Hlavním rozdílem oproti *state* je jejich neměnnost. Není tedy možné používané *props* za pochodu měnit [18].

```
function Hello(props) {
      return <h1> Hey, {props.firstname} {props.lastname} </h1>
}
```

```
<div> <Hello firstname ="Pepa" lastname="Dvořák" /> </div>
```

Ukázka kódu 14 Použití props. Zdroj: [autor]

4.1.6 Komunikace s API třetích stran

V Reactu se pro komunikaci s API nejvíce používá knihovna AJAX a AXIOS. Pro dotazování se na API, za účelem získání dat, při otevření stránky, by se měla používat metoda životního cyklu komponenty componentDidMount(). V této metodě lze rovnou jednoduše nastavit například *state* (příkazem *setState*) z dat příchozích z dané API a s daty je tak možné ihned pracovat.

AJAX

Pro získání dat z API je doporučeno používat asynchronní funkci *async/await*, nebo *promise/then*, která čeká, až požadovaná data budou z dané API získána. Bez použití asynchronní funkce by se mohlo stát, že se objeví chyba, jelikož komponenta se již načetla ale data z API zatím ne, takže by komponenta pracovala s *null* daty. Níže je uveden příklad, jak může vypadat metoda za využití AJAXu a `promise/then` [19]:

```
componentDidMount() {
      fetch("http://localhost:8081/api/user")
      .then(res => res.json())
      .then(json => this.setState({data: json}))
}
```

Ukázka kódu 15 Volání API za pomocí AJAXu a promise/then. Zdroj: [autor]

AXIOS

AXIOS disponuje několika základními metodami, mezi které patří například metody GET, POST nebo DELETE. Příklad volání metody GET je zobrazen na příkladu uvedeném výše. AXIOS také obsahuje mnoho využitelných parametrů, jako je například baseURL nebo responseType. Zde je uveden příklad volání API za použití knihovny AXIOS [19]:

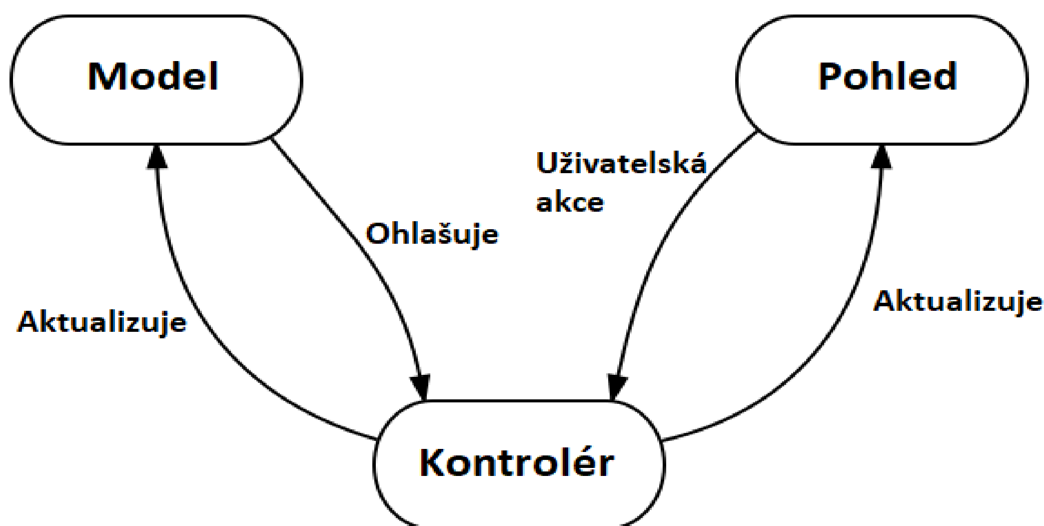
```
componentDidMount () {
const res = axios.get("http://localhost:8081/api/place/all")
}
```

Ukázka kódu 16 Volání API za použití AXIOSu. Zdroj: [autor]

4.2 Architektura MVC

Principem MVC architektury je vyřešit problém "špagetového kódu", kdy jedna třída obsahuje jak vykreslování výstupu, tak i celou logiku fungování aplikace. Je tedy nutné rozdělit aplikace na tři části – Model (Model), Pohled (View) a Kontrolér (Controller). Celé fungování je založené na posílání dotazů *(request)* z frontendu na backend. Zjednodušeně je celé fungování popsáno v krokovaném seznamu níže [20]:

1. V prohlížeči klient (frontend) zašle dotaz na požadovanou URL adresu (end-point) do kontroléru.
2. Kontrolér zašle dotaz na Model, který daný dotaz zpracuje a výsledek pošle zpátky do kontroléru.
3. Kontrolér data pošle zpět do Pohledu.
4. Pohled zobrazí výsledná data na obrazovce.



Obrázek 2 Architektura MVC. Zdroj: [21]

Model

Model je zjednodušeně řečeno veškerá business logika celé aplikace. Obsahuje všechno přes různé výpočty, validace vstupů, odchytávání chyb, dotazování se na databázi a tak dále. Model se absolutně nestará o nic jiného než o svá data, se kterými pracuje. To znamená, že vůbec o Pohledu či Kontroléru neví [20].

Pohled

Pohled se stará o vykreslování dat na obrazovku. Většinou to bývá HTML šablona. Obsahuje minimální množství logiky, kterou potřebuje pro správné zobrazování požadovaných dat. Pohled také může obsahovat základní validaci, aby se zbytečně data neposílala do backendu a server ihned nevrátil chybu. Pro efektivní a moderní implementaci Pohledu jsou v dnešní době velmi populárními frontendovými technologiemi ReactJS, VueJS anebo AngularJS [20].

Kontrolér

Kontrolér se stará o veškerý datový tok probíhající mezi vrstvami Model a Pohled, a aktualizuje Pohled pokaždé, když se data na backendu, ve vrstvě Model, změní. Pokud klient pošle dotaz na backend, Kontrolér je zodpovědný za vrácení požadované hodnoty do zaslaného dotazu. Kontrolér získá input od uživatele skrze vrstvu Pohledu. Poté ve spolupráci s vrstvou Model provede všechny potřebné operace a vrátí výsledek zpátky do vrstvy Pohledu [22].

4.2.1 Výhody a nevýhody architektury

Jako každá architektura má MVC výhody a nevýhody, nicméně v tomto případě převažují výhody nad nevýhodami [21].

Výhody

- rozdělení aplikace na tři části – Model, Pohled a Kontrolér,
- aplikace je lépe spravovatelná a mnohem přehlednější, s čímž souvisí i rychlejší vývoj samotné aplikace,
- nabízí možnost více Pohledů,
- podporuje asynchronní operace,
- každá změna změní pouze danou vrstvu, například změna v Pohledu nenaruší fungování Modelu či Kontroléru,
- architektura je takzvaně „SEO friendly“ (optimalizace webových stránek pro vyhledávače), což ve zkratce znamená, že lze velmi jednoduše vytvořit a optimalizovat webové aplikace z pohledu SEO

Nevýhody

- velmi komplexní architektura,
- je vhodnější pro rozsáhlejší webové aplikace,
- rozdělení vývoje na tři části občas může vést ke zpoždění vývoje v jednotlivých částech

4.3 Spring Framework

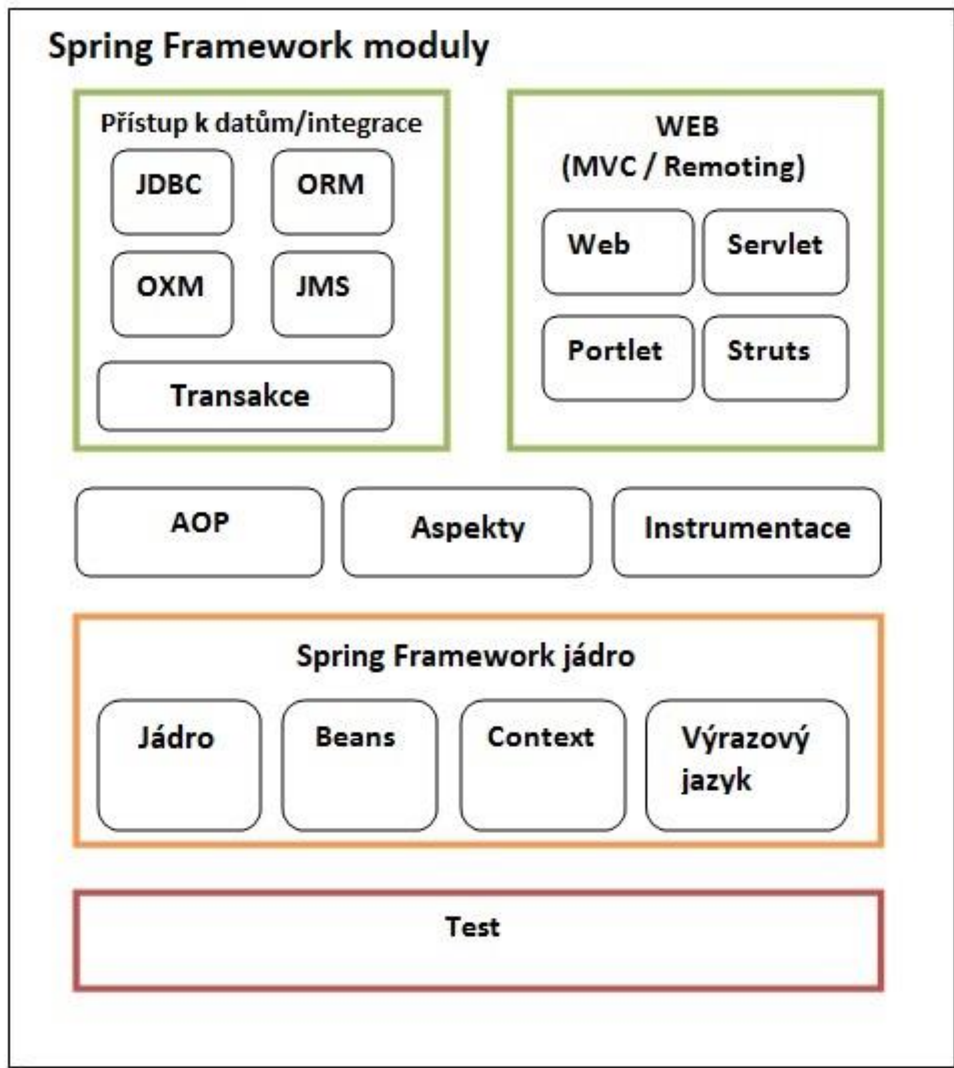
Spring Framework je open-source (lze upravovat podle svých představ) framework postavený na principu vkládání závislostí (Dependency Injection) a obráceného řízení (Inversion of Control). Spring byl zpočátku vyvinut a fungoval několik let pouze jako IoC kontejner, který obsahoval několik dalších rozšíření, mezi které patří například aspektově orientované programování, autorizace, nebo architektura Model-View-Controller (MVC). Prvotní verze Springu vyšla v roce 2002 a byla vytvořena Rodem Johnsonem společně s jeho knihou. Nicméně první ucelená verze Springu byla vydána až v roce 2003 pod licencí Apache. Poté vyšlo spousty nových verzí Springu, nicméně až v roce 2012 vyšla verze 3.2, která představila konfiguraci pro Javu, Hibernate 4 nebo Servlet 3.0. Spring Boot byl představen v roce 2014 [23].

4.3.1 Spring moduly

Spring Framework sestává z několika základních modulů, mezi které patří například [24]:

- AOP (Aspektově Orientované Programování),
- JDBC,
- Servlet,
- Web,
- Aspects,
- Core, a další

Jednotlivé moduly jsou dále seskupeny do větších skupin, což je vyobrazené na Obrázek 3.



Obrázek 3 Spring Framework moduly. Zdroj: [24]

4.3.2 Inversion of Control (IoC)

Inversion of Control, česky obrácené řízení, je způsob programování, kde jednotlivé části programu jsou převzaty do kontejnerů, nebo do frameworků. Převzetím částí programu IoC umožní frameworku (či kontejneru) převzít kontrolu nad programem a jeho životním cyklem a volat metody z vlastního kódu. To je podstatný rozdíl oproti klasickému principu programování, kde náš vlastní kód volá metody z knihoven. Pro fungování IoC architektury framework využívá abstrakcí se zabudovaným předdefinovaným chováním. Pro přidání vlastního chování je potřeba rozšířit třídy frameworku, nebo doplnit své vlastní třídy. Svoje využití IoC nachází hlavně v Objektově Orientovaném Programování (OOP).

Mezi výhody IoC architektury patří [25]:

- lepší modularita programu,
- usnadnění přepínání mezi implementacemi,
- snadnější testování programu skrze izolaci komponenty.

4.3.3 Dependency Injection (DI)

Dependency Injection, česky vkládání závislostí, je způsob, jakým lze jednoduše implementovat výše zmíněnou Inversion of Control architekturu. Je to vzor fungující na vkládání závislostí mezi komponentami aplikace tak, aby si jednotlivé komponenty mohly navzájem využívat svoje služby poskytující okolí, bez vložení referencí na dané komponenty. Je to způsob, kdy si o svých závislostech nerozhoduje sám objekt, ale kdy daný objekt přijde k závislostem externě, příkladem je použití konstruktoru. Tímto způsobem objekt počítá pouze s tím, že jednotlivé závislosti budou implementovat rozhraní. V této fázi už objekt nezajímá, které konkrétní instance těch závislostí to budou. Ukázkovým příkladem DI by mohl být například člověk a auto, kdy objektu člověk přijde objekt auto a člověka už nezajímá, jaké auto to je. To znamená, že člověk není závislý na konkrétní instanci daného auta [25].

Existují tři přístupy řešení Dependency Injection. Prvním způsobem je vkládání konstruktorem (constructor injection), který je popsán v textu výše. Druhým, pro Spring známějším způsobem je vkládání `setterem` (setter injection). Tento způsob funguje tak, že ke každé jednotlivé závislosti je vytvořen setter a ostatní objekty tyto settery volají ještě před prvním použitím objektu. Velkou nevýhodou tohoto přístupu je výskyt chyby při běhu programu. Typickou chybou, která může nastat je `NullPointerException`, která se objeví v okamžiku, dojde-li k postrádání potřebné závislosti [26]. Třetím a posledním přístupem k DI je tzv. vkládání pomocí rozhraní (interface injection). Klient musí implementovat rozhraní, které volně vystavuje metodu set, která přijímá danou závislost [27].

Zde je uveden příklad, jak by vypadala závislost objektu v klasickém přístupu:

```
public class Clovek {
      private Car car1;
      public Clovek () {
            car1 = new Car ();
      }
}
```

Ukázka kódu 17 Závislost objektu v klasickém přístupu. Zdroj: [autor]

Zde je příklad za použití Dependency Injection:

```
public class Clovek {
      private Auto auto;
      public Clovek () {
                  this.auto = auto; }
      }
}
```

Ukázka kódu 18 Použití Dependency Injection. Zdroj: [autor]

4.3.4 Anotace

Java umožnila podporu používání anotací již ve verzi 5.0. Spring framework začal anotace používat od verze 2.5. Před příchodem anotací bylo chování Springu řízeno XML konfiguracemi. Dnes většinou definujeme chování Springu přes anotace. Nicméně lze použít i přístup XML. Anotace se vkládají na celou třídu, na metodu, nebo na atribut [28].

Stereotypní anotace

- @Component – Anotace používaná na jednotlivé třídy za cílem označení dané třídy jako Spring komponenta.
- @Controller – Anotace používaná na označení, že daná třída je Controller. Zastupuje Controller v MVC architektuře.
- @Service – Anotace označující třídu, která poskytuje služby. Například všechny výpočetní úlohy nebo volání API.
- @Repository – Anotace používaná u té třídy, která má za úkol přímo pracovat s daty uložené v databázi.

Kontrolér

V kontroléru se obsluhují dotazy od klienta. Je nutné definovat třídy, které budou používané v daném kontroléru. Anotace `@Autowired` na třídě zajišťuje, že se bude pracovat pouze s tou jednou a tou samou instancí třídy v celé aplikaci. Následně je nutné vytvořit metodu s anotací `@RequestMapping` nebo jinými alternativami, kde lze specifikovat typ dotazu (GET, POST, DELETE, PUT), který na kontrolér přijde ze strany klienta. Níže je uvedený příklad, který demonstruje dotaz od klienta na vrácení všech míst uložených ve specifickém výletu, podle identifikátoru výletu.

```
@GetMapping(”/all/{tripIdentifier”}
Public Iterable<Place> getAllPlaces(@PathVariable String
tripIdentifier, Principal principal){
      return placeService.findAllPlaces(tripIdentifier,
      principal.getName());
}
```

Ukázka kódu 19 Metoda v kontroléru vracející místa podle ID výletu. Zdroj:

Služba

Ve službě (Service) se provádějí veškeré výpočty a operace potřebné pro splnění klientova dotazu. Je nutné brát v potaz i ošetření všech možných chyb, které mohou nastat v rámci práce s daným dotazem. Nesmí se také zapomenout na anotaci `@Service`, která označí danou třídu jako obsluhu dat. Níže uvedený kód popisuje nalezení konkrétního výletu podle e-mailu (username) uživatele a následně nalezení všech míst obsažených v tomto nalezeném výletu.

```
public Iterable<Place> findAllPlaces(String username, String
tripIdentifier){
      Trip trip = tripService.findTripByTripIdentifier(username,
tripIdentifier);
      return placeRepository.findAllByTrip(trip)
}
```

Ukázka kódu 20 Metoda ve službě, která najde všechna místa. Zdroj: [autor]

Repozitář

Skrze repozitář lze pracovat přímo s daty v databázi. Stejně jako u obsluhy dat a u kontroléru je nutné repozitář označit anotací `@Repository`, tím se aplikace dozví, že v tomto místě se bude pracovat s databází. Repozitáře mohou dědit z řady externích

repozitářů, například JpaRepository, nebo CrudRepository. Níže uvedený příklad popisuje, jak může repozitář vypadat.

```
@Repository
Public interface placeRepository extends CrudRepository<Place, Long>{
      Iterable<Place> findAllByTrip(Trip trip);
}
```

Ukázka kódu 21 Metoda v repozitáři, která hledá daná místa v databázi. Zdroj: [autor]

Mapování dotazů

Anotace `@RequestMapping`, která se používá k mapování požadavků z klientské strany na konkrétní třídy či metody řešící HTTP požadavky. Pokud se použije na celou třídu, vytvoří URL, pro kterou bude daný kontrolér používán. To znamená, že všechny požadavky směřované na danou URL bude obstarávat definovaný kontrolér. Pokud se ale anotace použije pouze na metodu, vytvoří URL pouze pro tu danou metodu a při HTTP požadavku se provede jen tato metoda. Je nutné specifikovat, jaký typ metody se má provést. Existují čtyři základní typy metod: GET, POST, DELETE, PUT.

```
@RequestMapping ("/user", method = RequestMethod.GET)
```

Ukázka kódu 22 Mapování dotazu pomocí metody GET. Zdroj: [autor]

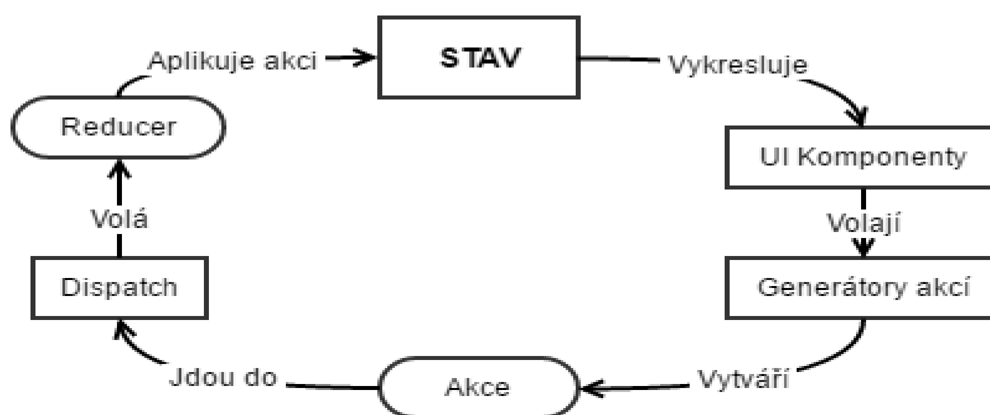
`@RequestMapping` lze zapsat přehlednější zkratkou v kontextu zmiňovaných metod:

```
@GetMapping("/user")
```

Ukázka kódu 23 Mapování dotazu pomocí @GetMapping anotace. Zdroj: [autor]

4.4 Redux

Redux je open-source knihovna pro správu stavu používaná pro aplikace napsané v programovacím jazyce Javascript. Slouží ke spravování stavu aplikace. Smyslem Reduxu je, že stav aplikace je udržován a spravován pouze na jednom místě aplikace. Redux je velmi populární a oblíbený v kombinaci s použitím frontendové technologie React. Nicméně dá se použít s jakoukoliv jinou knihovnou zastávající architekturu View v Model View Controller architektuře. Redux nabízí spoustu zajímavých a pomocných doplňků, čímž se stává mezi programátory velmi oblíbenou technologií. Aby byl Redux funkční a efektivní, využívá takzvaného reduxového úložiště (redux store), akcí (actions) a reduktorů (reducers). Redux řeší uchovávání stavu do jednoho objektu, který ovšem nemá žádné settery. To znamená, že stav nemůžeme klasicky měnit pomocí setterů, ale pouze pomocí definovaných akcí.



Obrázek 4 Architektura technologie Redux. Zdroj: [29]

4.4.1 Hlavní principy

Fungování Reduxu je možné rozdělit na tři hlavní principy, na kterých je Redux založen [30]:

1. Jedno úložiště – Stav celé aplikace je uložen v objektu ve stromové hierarchii v jednom jediném úložišti nazývaném Redux `Store`. Hlavní předností tohoto jednoho úložiště je, že všechna data jsou uložena pouze na jednom místě, v jednom objektu. Tento objekt má stromovou strukturu, což znamená, že k datům se lze jednoduše dostat a pracovat s nimi.

2. Stav sloužící pouze ke čtení – Stav může být změněn pouze zavoláním takzvané akce, což je jednoduchý objekt popisující, co se má stát se stavem. Tím pádem není možné zapisovat do stavu například pomocí view, nebo takzvaných `callback` funkcí.
3. Změny jsou prováděny pouze jednoduchými funkcemi – Těmito jednoduchými funkcemi jsou myšleny takzvané `reducers`, což jsou tedy funkce, které dostanou předchozí stav aplikace a vyvolanou akci, a vrátí stav další. Stavem dalším je myšleno, v jakém stavu se nachází `state` po provedení určité akce.

4.4.2 Store

Store neboli úložiště je pouze objekt, ve kterém je uložen stav celé aplikace. V Reduxu je důležité si uvědomit a pochopit, že zde existuje pouze jedno úložiště. Pokud se bude aplikace zvětšovat a bude potřeba pracovat s více odlišnými daty, nevytváří se další úložiště, ale je nutné rozdělit základní `reducer`, `rootReducer`, na menší jednotlivé `reducers` starající se o jednotlivé části stavu zvlášť. K tomu slouží metoda `combineReducers`, ve které lze kombinovat všechny `reducers` do jednoho `reduceru`, se kterým se poté dá pracovat při vytváření úložiště v metodě `createStore`. Výsledný `reducer` poté volá ty jednotlivé `reducers` a výsledná data ukládá do jednoho stavu. Úložiště disponuje třemi základními metodami, kterými jsou [31]:

- `getState()` – metoda vracející momentální stav celé aplikace
- `subscribe(listener)` – metoda, která se zavolá pokaždé, když se provede `dispatch` akce a změní se stav
- `dispatch(action)` – vyšle akci měnící stav

4.4.3 Actions

`Actions` neboli akce jsou jednoduché Javascriptové objekty, které při provedení posílají data z aplikace do vytvořeného úložiště. Každá akce musí obsahovat vyspecifikovaný typ. Typy se vytváří jako klasické konstanty nejčastěji v externím souboru, anebo pokud je to dostačující, tak přímo v dané akci například jako `string`. Celé funkci, která vytváří jednotlivé akce se říká tvůrce akce `action creator`. Aby došlo k zaslání dat z aplikace

do úložiště, je nutné zavolat funkci Odeslat `dispatch`. V akci, konkrétně ve funkci Odeslat, lze specifikovat ještě argumenty chyba `error` a výsledek `payload`. `Error` standardně vrací chyby a `payload` je jakýkoliv výsledek jisté provedené akce. Akce popisují pouze co se má stát, ne jak se to má stát [32].

Akce mohou být buď plně synchronní, anebo zcela asynchronní. Synchronní funkce je taková funkce, ve které když se zavolá funkce Odeslání na akci, stav se okamžitě změní bez ohledu na to, zda například byla získána data z API apod. Tento problém řeší funkce asynchronní, kde celá aplikace čeká, až se provede například zavolání API. Příklad níže popisuje tvůrce akce, který zavolá akci s typem `GET_TRIP` a do `payload` se uloží výsledek z volání API. Celá akce je asynchronní.

```
export const getTrip = () => async dispatch => {
      const result = await axios.get (API)
      dispatch ({
            type: GET_TRIP,
            payload: result.data});
}
```

Ukázka kódu 24 Příklad akce, která volá API a vrací výsledek. Zdroj: [autor]

4.4.4 Reducers

`Reducers` neboli reduktory určují, jak se mění stav celé aplikace v závislosti na vyvolaných akcích. Jsou to jednoduché funkce, které pro svoje fungování potřebují parametry předchozí stav `previousState` a akci, a vrací hodnotu následujících `props` `nextState`. Reduktor patří do skupiny čistých `pure` funkcí, což znamená, že by pouze měly dostat argumenty, změnit stav podle provedených akcí a vrátit stav. V reduktoru by se neměly například měnit argumenty daného reduktoru anebo volat API. Doporučuje se vytvořit a pracovat s více reduktory, pokud je aplikace obsáhlejší a je nutné data separovat. Pracovat s reduktory je možné až po vytvoření jednoho celkového reduktoru, který všechny vytvořené reduktory kombinuje dohromady pomocí metody kombinaceReduktorů `combineReducers` [33].

4.4.5 Komunikace React + Redux

Jak už bylo zmíněno výše. V dnešní době je velmi populární kombinací pro vývoj webových aplikací React a Redux. Obě tyto technologie jsou samostatné knihovny, to znamená, že mohou fungovat zcela samostatně.

Balíček react-redux

O celou komunikace mezi Reactem a Reduxem se stará balíček nazvaný react-redux. Myšlenkou fungování balíčku jsou dva typy komponent: Prezentační komponenty (Presentational components) a Kontejnerové komponenty (Container components). Prezenčními komponentami se rozumí klasické funkcionální komponenty nebo komponenty typu třídy v Reactu, které se starají o vykreslování HTML elementů na obrazovku. Pokud není nutné používat metody životního cyklu, je lepší psát komponenty jako funkcionální. Jsou přehlednější a jednodušší z pohledu Reduxu. Kontejnerové komponenty jsou klasické React komponenty využívající metody `subscribe()`, kterou nabízí `store`, která slouží ke čtení stavu aplikace a dodává `props` do prezenčních komponent, které jsou vykresleny. Kontejnerovou komponentu vytvoříme přes metodu `connect()` [34]. Metoda `connect()` potřebuje pro své fungování funkci `mapStateToProps`, která definuje, jak transformovat aktuální stav Reduxu do `props`, se kterými se bude pracovat v prezentační komponentě, která se nachází ve vytvořené kontejnerové komponentě. Kontejnerové komponenty mohou také vyvolávat akce. K tomu je zapotřebí vytvořit ještě další funkci `mapDispatchToProps`, která na vstupu dostane `dispatch()` a vrátí `callback props`, které se vloží do prezentační komponenty [34].

Aby komunikace mezi `store` a kontejnerovými komponentami fungovala, musí mít všechny kontejnerové komponenty přístup do `store` Reduxu. To se řeší přes komponentu nazvanou `Provider`, která zpřístupní úložiště všem kontejnerovým komponentám. Je nutné ji definovat co nejvýše v kořenu `(root)` aplikace. Například v App.js, nebo ve vykreslování hlavní komponenty [34].

5 Návrh aplikace

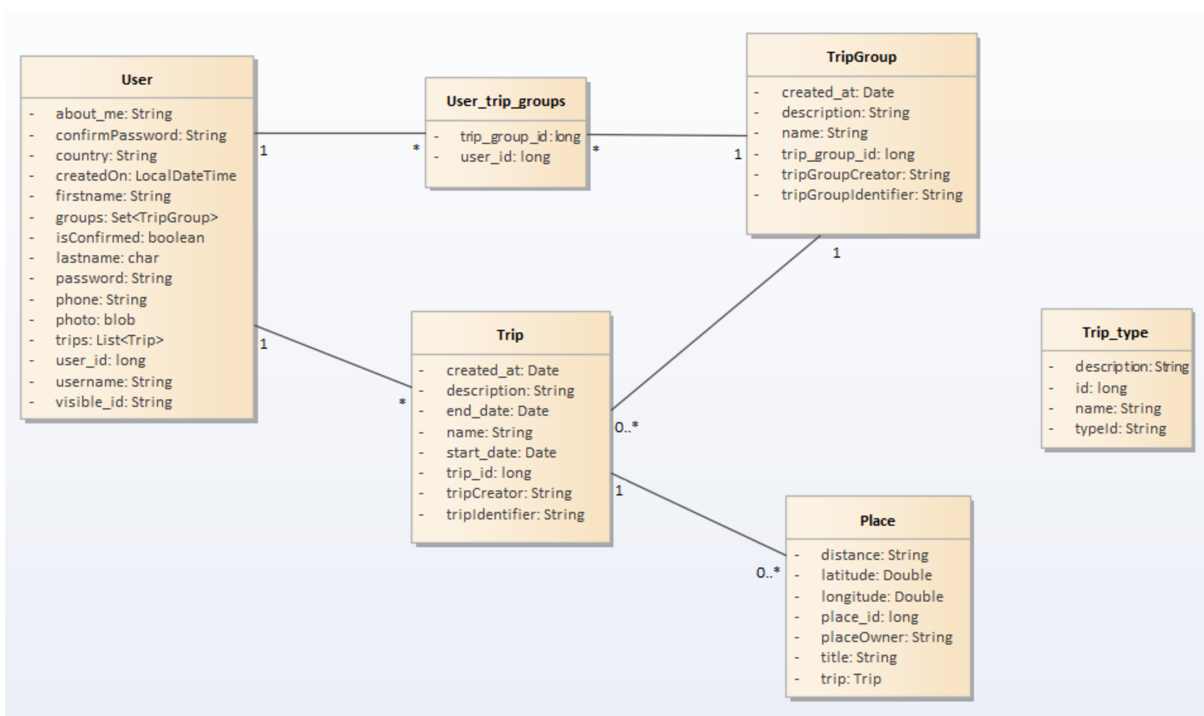
Cílem návrhu aplikace bylo vybrat vhodné technologie, programovací jazyky a návrh databázové struktury pro vývoj webové aplikace pro plánování volného času – PlanApp. Pro vývoj praktické části byl tedy použit na frontendu ReactJS, na backendu Spring Framwork a databáze MySQL. Dalším krokem bylo detailnější vymyšlení funkcionality aplikace. Bylo zapotřebí vyřešit vytváření uživatelů, výletů a skupin, a přidávání výletů a uživatelů do skupin jinými uživateli. Následně byly vymyšleny vztahy mezi jednotlivými doménami. Výstupem bylo navrhnutí databázové struktury (Obrázek 5) aby aplikace fungovala tak, jak byla zpočátku zamýšlena. Konečnou fází návrhu byl počáteční design aplikace, kde je hlavní strana s přehledem vytvořených výletů, formuláře pro vytváření výletů a skupin a v neposlední řadě design profilu uživatele.

5.1 Návrh databázové struktury

Databázová struktura určuje chování aplikace ve smyslu ukládání dat a práce s nimi. Konečná efektivita aplikace je také závislá na správně navrhnuté databázi. Níže je uveden popis jednotlivých tabulek vyvíjené aplikace.

- První tabulkou je tabulka Uživatel (User) obsahuje všechny informace potřebné pro zamýšlené fungování uživatele. Jedinečným viditelným identifikátorem Uživatele je jeho e-mail (username), se kterým se pracuje skrze celou aplikaci.
- Výlet (Trip) je tabulkou zastupující vytvořené výlety uživatelů. Výlet lze tedy vytvořit, editovat, smazat, anebo přidat do skupiny výletů (tabulka TripGroup). Atribut tripIdentifier je zamýšlen jako viditelné unikátní ID každého vytvořeného výletu. Tím je způsobeno, že lze vytvořit výlet s daným ID pouze jednou, skrze celou aplikaci. Po smazání výletu s daným ID je to ID opět volné pro další výlety. Dalším atributem je tvůrce výletu (tripCreator), který znázorňuje tvůrce daného výletu pro identifikaci, komu konkrétní výlet patří.
- Skupina Výletů (TripGroup) je skupina, do které se dají přidávat výlety a uživatelé. Uživatelé poté vidí danou skupinu a výlety v ní.
- Místo (Place) je tabulka pro ukládání dat o jednotlivých místech získávaných z externí API. Ukládané informace jsou poté používány při zobrazování informací o daných místech uživatelovi.

- Poslední tabulkou je Typ Výletu (TripType), do kterého se ukládá zatím manuálně hned při vytvoření databáze druhy výletů. Jednotlivé druhy jsou použity v URL adrese při volání externí API pro získání míst z daného typu míst.



Obrázek 5 Model tříd vyvíjené aplikace. Zdroj: [Autor]

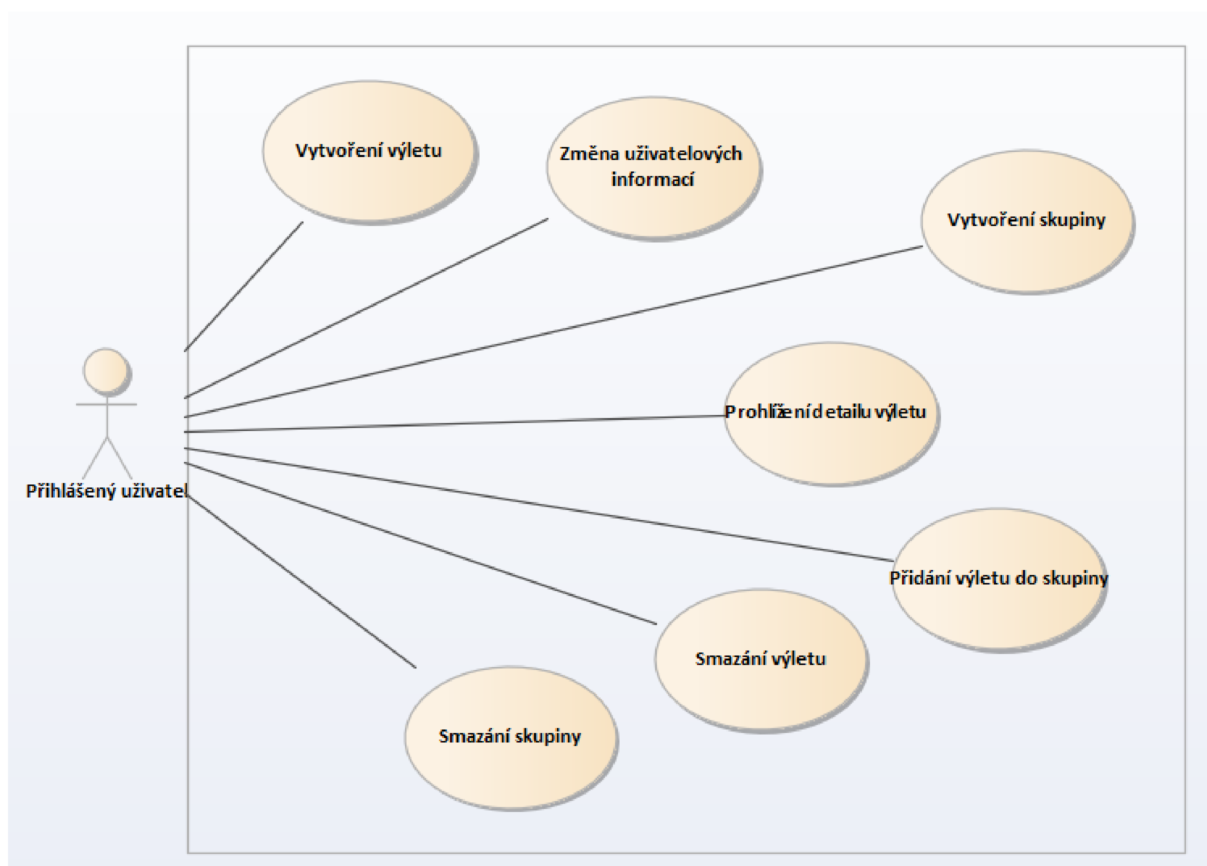
Vztahy mezi tabulkami

Tabulka Uživatel má vazbu na tabulku výletová skupina (TripGroup) ve vztahu Many-To-Many. To proto, že tabulka User může mít N výletových skupin a výletová skupina může mít N uživatelů. Pro vztah 1:N mezi tabulkami User a Výlet (Trip) platí, že jeden User může vytvořit N výletů, ale jeden výlet může být výletem právě jednoho uživatele. Logika vztahu mezi Výletem a Místem je 1:N, kdy jeden výlet bude obsahovat nula až několik jednotlivých míst. Poslední tabulkou je Typ Výletu (TripType), do které se ukládají zatím manuálně hned při vytvoření databáze druhy výletů. Jednotlivé druhy jsou použity v URL adrese při volání externí API pro získání míst z daného typu míst.

5.2 Use-Case diagram

Níže uvedený Use-Case diagram zobrazuje základní akce, které mohou uživatelé provádět. Přihlášený uživatel může plně využívat celou aplikaci. Může například vytvořit výlet,

editovat, smazat výlet, vytvořit skupinu, do které lze přidat jak výlet, tak dalšího uživatele, aby mohlo více uživatelů sdílet stejné výlety, editovat svůj uživatelský profil, a tak dále.



Obrázek 6 Diagram případů užití. Zdroj: [Autor]

5.3 Funkční a nefunkční požadavky

Funkční a nefunkční požadavky jsou takové požadavky, které je nutné stanovit ještě před začátkem vývoje a následně jsou očekávané při finální verzi vyvíjené aplikace. Funkční požadavky definují požadavky na funkčnost aplikace, tzn. veškerá funkčnost aplikace, co by aplikace měla umět, a co ne. Nefunkční požadavky jsou všechny požadavky, které nemají s funkčností nic společného (například design, rychlost apod.)

Funkční požadavky

- registrace uživatele,
- přihlášení uživatele,
- editace profilu,
- vytvoření "výletu" (ve skutečnosti se jedná pouze o složku, která v sobě obsahuje název, identifikátor, popis, zvolená místa a trasu mezi nimi),

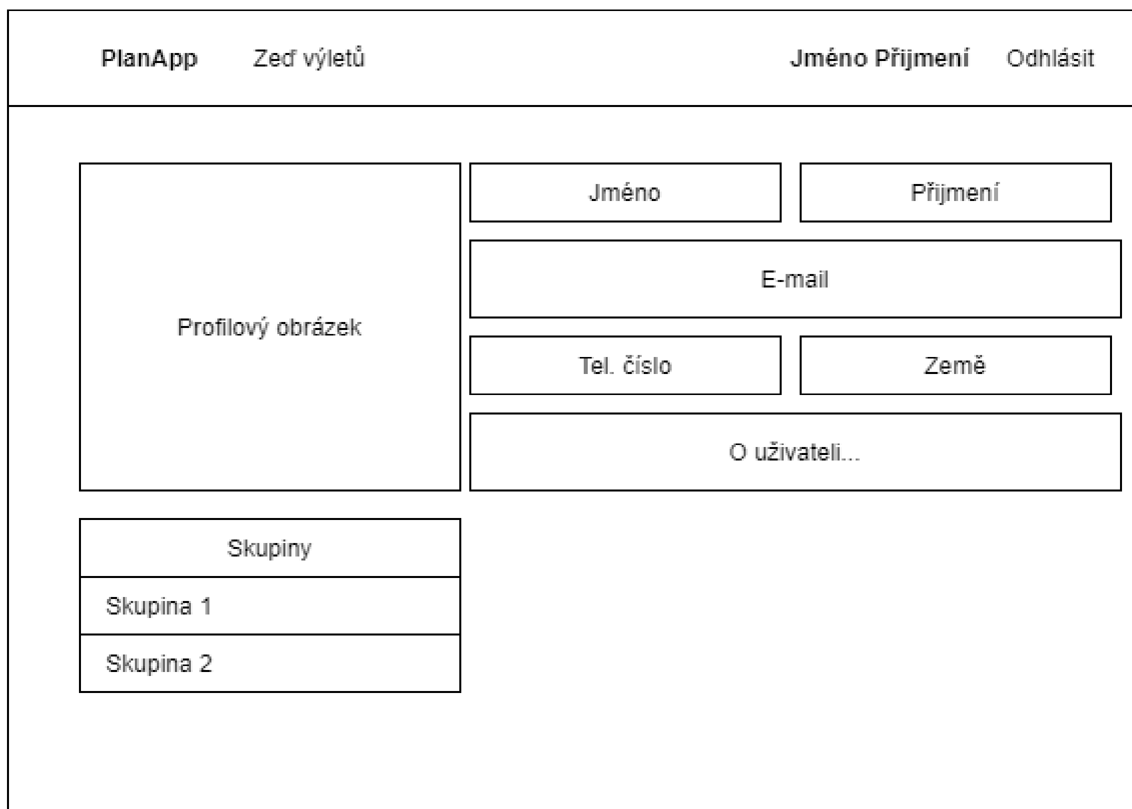
- vytvoření skupiny, kam bude uživatel moci přidávat jednotlivé výlety a uživatele,
- možnost vybrání nejbližších míst k uživatelovi,
- uložení míst do výletu,
- vygenerování trasy mezi zvolenými místy,
- editace výletů a skupin,
- smazání výletů a skupin

Nefunkční požadavky

- jednoduchý intuitivní design bez zbytečných složitostí,
- rychlá odezva (omezeno z důvodu hostingu na Heroku)

5.4 Drátový model aplikace

Drátový model slouží k prvotnímu navrhnutí designu celé vyvíjené aplikace. Níže jsou uvedeny obrázky znázorňující drátové modely hlavních stran. První obrázek představuje původní myšlenku profilu uživatele, kde by se nacházely i jednotlivé skupiny, nicméně to úplně nesplňuje intuitivnost aplikace.



Obrázek 7 Drátový model profilu aplikace. Zdroj: [Autor]

Druhý obrázek zobrazuje drátový model hlavní zdi (Dashboard) a hlavičky aplikace, která slouží jako menu. Zeď by měla obsahovat tlačítko na vytvoření nového výletu a jednotlivé vytvořené výlety.

Obrázek 8 Drátový model hlavní zdi aplikace. Zdroj: [Autor]

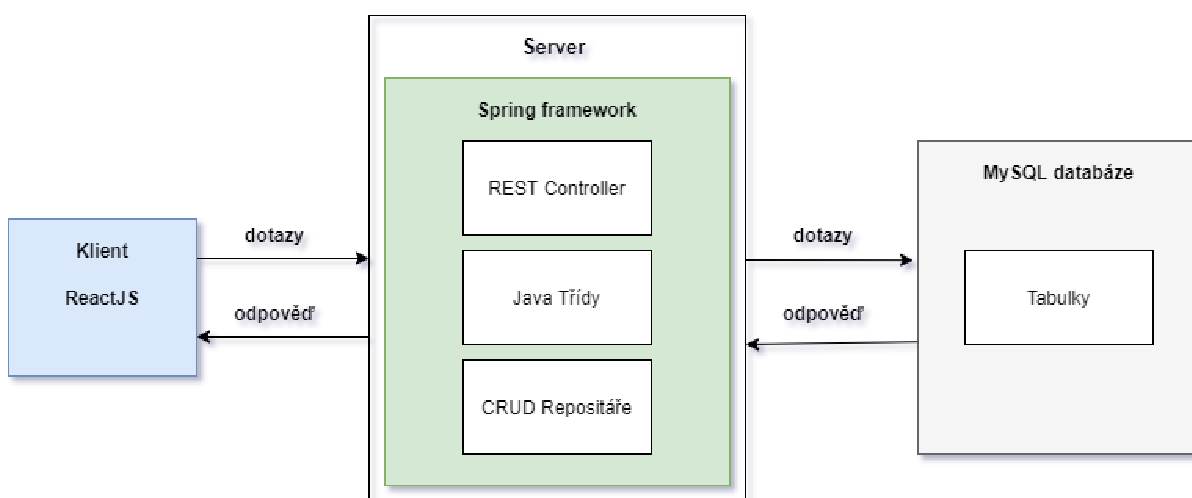
6 Implementace aplikace

Po důkladném návrhu aplikace z pohledu databázové struktury, funkčních a nefunkčních požadavků a vytvoření diagramů, byla dalším krokem samotná implementace aplikace, při které vznikalo spoustu zajímavých problémů.

6.1 Architektura aplikace

Obrázek a seznam níže popisuje architekturu aplikace postavené na ReactJS pohledu, Spring framework backendu a MySQL databáze. Jak Pohled, tak i celý server a databáze jsou hostované na serveru Heroku, o kterém jsou základní informace uvedené v následující kapitole.

1. klient neboli pohled, zašle dotaz na server (Spring backend),
2. server provede veškeré nutné operace dle dotazu od klienta,
3. server provede změny či dotazy na úrovni databáze,
4. z databáze se vezme výsledek a zpracuje se na úrovni serveru,
5. výsledek se vrátí pohledu,
6. pohled zobrazí uživatelovi data.



Obrázek 9 Struktura aplikace ReactJs, Spring Framework, MySQL. Zdroj: [Autor]

6.2 Heroku

U každé webové aplikace je nutné, aby běžela na produkčním serveru z důvodu dostupnosti a jednoduchosti navštívení dané aplikace. Pro tyto účely bylo tedy zvoleno

Heroku. Je to platforma, která umožňuje vývojářům vytvářet projekty a spustit a provozovat aplikace přímo v cloudu [35].

Nasazení backendu a databáze

Nasadit backend bylo složitější než frontend. Prvním krokem bylo vytvořit obyčejný soubor s názvem Procfile, podle kterého Heroku poznává, jaký příkaz má být vykonán, aby se aplikace nastartovala. Soubor Procfile má v sobě pouze jeden řádek informací a vypadá přibližně takto:

```
web: java -jar $JAVA_OPTS target/planapp-1.0-SNAPSHOT.jar -
Dserver.port=$PORT $JAR_OPTS
```

Web v příkazu je velmi důležitý, protože Heroku podle toho určuje, že tento typ procesu bude připojen ke směrovacímu zásobníku HTTP. Dalším krokem je vložení zdrojového kódu na github.com, kde je nutné si vytvořit klasický repositář. Alternativou pro github.com, je vytvoření repozitáře přímo na webových stránkách Heroku, do kterého se vloží zdrojový kód. Dalším krokem je zvolení databáze. Nejjednodušší je vybrání a nastavení PostgresSQL databáze, nicméně pro bakalářský projekt byla vybrána ClearDB, což je MySQL databáze. Po napojení na github.com, nebo jiný repozitář, a vytvoření databáze lze aplikaci skrze Heroku spustit.

Nasazení frontendu

Nasazení frontendu na Heroku je výrazně jednodušší. Je také nutné vytvořit další repositář na github.com, kam se pošle zdrojový kód z frontendu a vytvořit nový projekt na heroku.com. Poté již stačí propojit projekt na Heroku a github, a frontend je připraven na spuštění.

6.3 Práce s mapami

V aplikaci bylo zapotřebí vyřešit problém získávání nejbližších míst dle uživatelovi polohy.

Získání uživatelovi polohy

Získání uživatelovi polohy je jednoduše řešitelný problém například přes příkaz `navigator.geolocation.getCurrentPosition()`, který se zeptá uživatele, zda chce v prohlížeči povolit polohu. V případě že ano, příkaz vrátí jednotlivé uživatelovi souřadnice. Funkce určuje polohu uživatele s nízkou přesností, což rozhodně není ideální v případě rozsáhlejších či složitějších aplikací.

Získání nejbližších míst

K práci s nejbližšími místy v okolí byla zvolena Developer Here API. Je to jednoduchá API, která potřebuje pro své správné fungování několik parametrů. Pro účely bakalářské praktické práce stačily souřadnice, radius (ve kterém chce uživatel nacházet místa) a identifikátor míst, dle kterého lze vyspecifikovat, jaký typ míst chce uživatel zobrazit. Příklad volání API za pomocí Reduxu by mohl vypadat přibližně takto:

```
export const getPlacesFromAPI = (id, latitude, longitude, radius,
history) => async dispatch => {
      return await fetch(
      `https://places.sit.ls.hereapi.com/places/v1/discover/explor
      e?apiKey=API-
      KEY=${latitude},${longitude};r=${radius}&cat=${id}`)
      .then(res => res.json())
      .then(json => {
            dispatch({
            type: GET_PLACES,
            payload: json
            });
            return json;
      })
}
```

Ukázka kódu 25 Metoda získání míst z API. Zdroj: [autor]

Odpověď ze zavolaného API přijde ve formátu JSON a vrací všechny důležité informace:


```
{
“results”: {
“next”: “https://places.demo.api.here.com/places/v1/discover/here;...”
“items”: [
{
      “position”: [
      37,79429,
      -122,40698
      ],
      “distance”: 10,
      “title”: “Dung Fong Trading Company”,
      “averageRating”: 0,
      “category”: {
            “id”:“shop”,
            “title”: “Store”,
            “href”: “https://places.demo.api.here.com/places/v1/...”
      “type”: “urn:nlp-types:category”,
```

```
        “system”: “places”
        },
        “icon”: “https://download.vcdn.cit.data.here.com/p/...”,
        “vicinity”: “101 Waverly Place<br/> San Francisco, CA 94108”,
        “having”: [],
        „type“: „urn:nlp-types:place“,
        „href“: “https://places.demo.api.here.com/places/v1/...“,
        „id“: „8409q1sa-5d6jsjdm5fj8ad6959dwfoi62pz7tvf“
        },
        ...
        }]
}
```

Ukázka kódu 26 Výstup ze zavolané API. Zdroj: [36]

Mapa

K práci s mapou bylo využito API od Googlu – konkrétně Google Maps API. Je zapotřebí získat API key. To v podstatě obnáší pouze registraci a vložení detailů debetní karty na vytvořený účet. Posledním krokem je vygenerování klíče, což je velmi jednoduché a intuitivní. V ReactJS je velmi jednoduché pracovat s mapami od Googlu, jelikož existuje knihovna react-google-maps, ze které lze libovolně využívat nabízené komponenty jako jsou: Ukazatel (Marker), Informační okénko (InfoWindow) a poté již samotnou Google mapu (Google Map). Mapa byla jednoduše vytvořena v render metodě tímto způsobem:

```
<GoogleMap
      defaultZoom={14}
      defaultCenter={{lat: 50.782, lng: 14.548}}>
      {places.map((place, index) => (
            <Marker key={index}
            position={{lat: place.lat, lng: place.lng}}/>
            ))
      }
      {this.state.selectedPlace && (
            <InfoWindow position={{
                  lat:this.state.selectedPlace.lat,
                  lng: this.state.selectedPlace.lng
            }} >
            <div>
                  <div>{this.state.selectedPlace.title}<div>
                  <div>{this.state.selectedPlace.vicinity}<div>
                  <div>{this.state.selectedPlace.distance}m<div>
            </div>
```

```
                </InfoWindow>
            )}
        </GoogleMap>
```

Ukázka kódu 27 Nastavení Google mapy s ukazateli a info okénky. Zdroj: [autor]

V projektu byla vytvořena Google mapa společně s ukazateli jednotlivých míst a s informačními okny zmíněných ukazatelů se základními informacemi o místech (tzn. název, adresa a vzdálenost).

Nejkratší trasy mezi místy

Dalším zajímavým problémem bylo vyřešit vykreslování tras mezi jednotlivými místy na mapě. Jak Developer Here tak i Google nabízí generování tras mezi místy. Nicméně pro tento projekt bylo zvoleno generování tras pomocí Google API zejména pro jednodušší a efektivnější použití. Stejně jak tomu bylo u ukazatelů či informačních okének, i v tomto případě stačilo pouze použít komponentu z již zmiňované knihovny react-google-maps – `DirectionsRenderer`. Tato komponenta potřebuje ke svému fungování pouze jeden parametr, čímž je parametr `directions`, který potřebuje objekt skládající se z počáteční destinace, cílové destinace, typu trasy a zastávkové body. Počáteční a cílová destinace jsou pouze souřadnice daných míst. Typ trasy lze zvolit automobilem, pěšky a na kole. Zastávkové body jsou pole jednotlivých objektů obsahujících souřadnice. V projektu v ukázce kódu uvedeného níže, je použit další parametr komponenty nazvaný `options`. Obsah parametru `options` slouží k tomu, aby automaticky generované ukazatele právě komponentou `DirectionsRenderer`, nebyly vidět.

```
        <DirectionsRenderer directions={this.state.directions}
            options={{suppressMarkers: true}} />
```

Ukázka kódu 28 Použití DirectionsRenderer uvnitř mapy. Zdroj: [autor]

6.4 Finální vzhled aplikace

Vzhled aplikace se lehce pozměnil oproti popsaným drátovým modelům v kapitole návrhu. Obrázek níže vyobrazuje téměř stejný finální design jako tomu bylo u drátového modelu.

PlanApp - Trip planner Dashboard Group Dashboard Marek Laušman Logout

Your Trip Dashboard

Create a Trip

TEST

Testovací výlet

Tento výlet slouží jako testovací...

This trip belongs to group with ID:

Trip Detail

Choose places

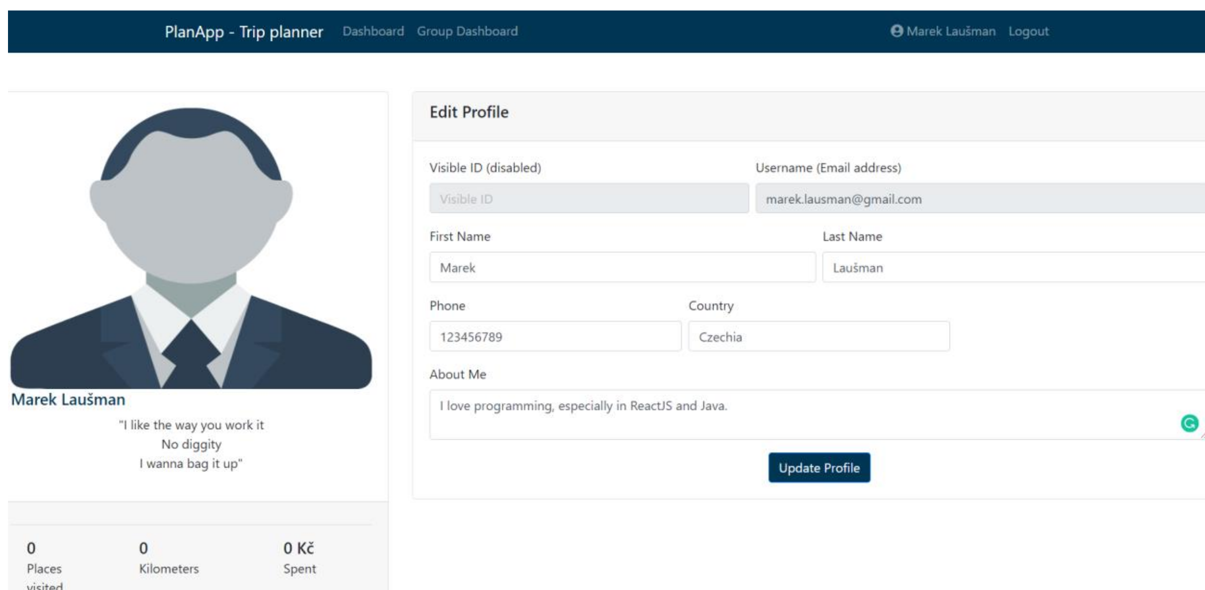
Edit Trip Info

Delete Trip

Add Trip to Group

Obrázek 10 Ukázka finálního designu zdi aplikace. Zdroj: [Autor]

Následující obrázek vyobrazuje profil uživatele. Byly odstraněny skupiny z důvodu snížení intuitivnosti ovládání aplikace. Pro skupiny byla vytvořena samostatná zeď (Group Dashboard), která vypadá obdobně jako zeď výletů.



Obrázek 11 Ukázka finálního designu profilu uživatele. Zdroj: [Autor]

7 Závěr

Cílem této bakalářské práce bylo vytvoření webové single page aplikace na plánování volného času s využitím frontendové technologie ReactJS, backendové technologie Spring Framework a MySQL databáze.

V práci byly představeny tři nejznámější typy přístupů k vývoji webových aplikací – SPA, MPA a PWA aplikace. Dále byla představena frontendová technologie ReactJS, konkrétně popis jejích základních vlastností, výhod a nevýhod a příkladů použití. Roli serveru zastával Spring Framework. Taktéž byly popsány základní vlastnosti a výhody a nevýhody této technologie v použití. Spolu s představením Spring Frameworku byla popsána architektura MVC, na které si vyvíjená aplikace zakládá. Konkrétně byly rozebrány jednotlivé části MVC architektury a již zmíněné výhody a nevýhody v použití. V teoretické částí se také dále mluví o popsání a použití knihovny Redux a o komunikaci mezi ReactJS a Redux.

V praktické části byl detailně popsán návrh a implementace aplikace, kdy v návrhu byly popsány konkrétní funkční a nefunkční požadavky, databázová struktura a případy užití vyvíjené aplikace. V implementaci poté byly popsány konkrétní zajímavé problémy, které bylo nutné řešit v průběhu vývoje. Aplikace počítá s odchylkou určení uživatelovi lokace.

Aplikace uvažuje několik budoucích rozšíření, mezi které patří například sledování uživatelova pohybu a na základě toho počítání trasy v rámci jednotlivých výletů. Systém bodování, ve kterém by za určité body uživatel dostával různé odměny. A v neposlední řadě například zobrazení oblíbených výletů jiných uživatelů na hlavní stránce.

8 Seznam použité literatury

[1] *Diplomova_prace_Dalibor_MOC.pdf* [online]. [vid. 2020-04-28]. Dostupné z: https://is.ambis.cz/th/dfcat/Diplomova_prace_Dalibor_MOC.pdf

[2] *What Is a Single-Page Application? - DZone Web Dev* [online]. [vid. 2020-03-31]. Dostupné z: https://dzone.com/articles/what-is-a-single-page-application

[3] *Most Popular Databases In The World* [online]. [vid. 2020-04-29]. Dostupné z: https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/

[4] Advantages & Disadvantages of Single Page Application. *Skenix Infotech* [online]. 4. červenec 2019 [vid. 2020-04-18]. Dostupné z: https://www.skenix.com/advantages-disadvantages-of-single-page-application/

[5] *Pros and Cons of Building Single Page Applications in 2019 - Gearheart* [online]. [vid. 2020-04-29]. Dostupné z: https://gearheart.io/blog/pros-and-cons-building-single-page-applications-2019/

[6] NEOTERIC. Single-page application vs. multiple-page application. *Medium* [online]. 30. březen 2018 [vid. 2020-04-28]. Dostupné z: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

[7] *What's the Difference Between Single-Page and Multi-Page Apps* [online]. [vid. 2020-04-18]. Dostupné z: https://rubygarage.org/blog/single-page-app-vs-multi-page-app

[8] *15 Top web development trends in 2020* [online]. [vid. 2020-04-28]. Dostupné z: https://lanars.com/blog/top-web-development-trends

[9] What Are The Advantages And Disadvantages Of Progressive Web Apps? *Blog Brainhub.eu* [online]. [vid. 2020-04-19]. Dostupné z: https://brainhub.eu/blog/advantages-disadvantages-progressive-web-apps/

[10] *Tutorial: Intro to React – React* [online]. [vid. 2020-03-22]. Dostupné z: https://reactjs.org/tutorial/tutorial.html

[11] MARTINEK, Vlastimil. Srovnání moderních javascriptových frameworků Vue.js a React.js. 2018, 49.

[12] *The Real Benefits of the Virtual DOM in React.js - Accelebrate* [online]. [vid. 2020-03-22]. Dostupné z: https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/

[13] Classes. *MDN Web Docs* [online]. [vid. 2020-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

[14] RITZCOVAN, Alex. Controlled vs Uncontrolled Components in React. *Medium* [online]. 4. únor 2020 [vid. 2020-03-21]. Dostupné z: https://itnext.io/controlled-vs-uncontrolled-components-in-react-5cd13b2075f9

[15] *Uncontrolled Components – React* [online]. [vid. 2020-03-21]. Dostupné z: https://reactjs.org/docs/uncontrolled-components.html

[16] *Thinking in React: Component Composition - DEV Community 👩‍💻👨‍💻* [online]. [vid. 2020-03-22]. Dostupné z: https://dev.to/bouhm/thinking-in-react-component-composition-fp5

[17] FOGARTY, Tim. Advantages and Pitfalls of Arrow Functions. *Medium* [online]. 29. září 2017 [vid. 2020-03-22]. Dostupné z: https://medium.com/tfogo/advantages-and-pitfalls-of-arrow-functions-a16f0835799e

[18] *Components and Props – React* [online]. [vid. 2020-04-28]. Dostupné z: https://reactjs.org/docs/components-and-props.html

[19] *AJAX and APIs – React* [online]. [vid. 2020-03-21]. Dostupné z: https://reactjs.org/docs/faq-ajax.html

[20] ČÁPKA, David. *MVC architektura* [online]. [vid. 2020-04-28]. Dostupné z: https://www.itnetwork.cz/mvc-architektura-navrhovy-vzor

[21] *What is MVC? Advantages and Disadvantages of MVC - Interserver Tips* [online]. [vid. 2020-03-31]. Dostupné z: https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/

[22] *MVC Framework - Controllers - Tutorialspoint* [online]. [vid. 2020-04-28]. Dostupné z: https://www.tutorialspoint.com/mvc_framework/mvc_framework_controllers.htm

[23] Introduction and History of the Spring Framework. *JAVAJEE.COM* [online]. 2. červen 2015 [vid. 2020-03-23]. Dostupné z: https://javajee.com/introduction-and-history-of-the-spring-framework

[24] Spring Modules Tutorial - javatpoint. *www.javatpoint.com* [online]. [vid. 2020-04-28]. Dostupné z: https://www.javatpoint.com/spring-modules

[25] CRUSOVEANU, Loredana. Inversion of Control and Dependency Injection with Spring. *Baeldung* [online]. 28. prosinec 2016 [vid. 2020-03-31]. Dostupné z: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

[26] *Vkládání závislostí (dependency injection) - Vojtěch Hordějčuk* [online]. [vid. 2020-03-31]. Dostupné z: http://voho.eu/wiki/vkladani-zavislosti/

[27] *A quick intro to Dependency Injection: what it is, and when to use it* [online]. [vid. 2020-04-28]. Dostupné z: https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

[28] SEPTEMBER 20 a 2017. Spring Framework Annotations. *Spring Framework Guru* [online]. 20. září 2017 [vid. 2020-04-28]. Dostupné z: https://springframework.guru/spring-framework-annotations/

[29] *Redux definuje tolik potřebná omezení na velkém projektu | Vsaď Na Javu.cz* [online]. [vid. 2020-04-28]. Dostupné z: https://vsadnajavu.cz/2016-08/odborne/javascript/jak-redux-definuje-tolik-potrebna-omezeni-na-velkem-projektu/

[30] *Redux* [online]. [vid. 2020-04-28]. Dostupné z: https://redux.js.org/introduction/three-principles

[31] *Redux* [online]. [vid. 2020-04-28]. Dostupné z: https://redux.js.org/basics/store

[32] *Redux* [online]. [vid. 2020-03-31]. Dostupné z: https://redux.js.org/basics/actions

[33] *Redux* [online]. [vid. 2020-04-28]. Dostupné z: https://redux.js.org/basics/reducers

[34] *Redux* [online]. [vid. 2020-03-27]. Dostupné z: https://redux.js.org/basics/usage-with-react

[35] *Getting Started on Heroku with Java | Heroku Dev Center* [online]. [vid. 2020-04-28]. Dostupné z: https://devcenter.heroku.com/articles/getting-started-with-java

[36] *Build apps with HERE Maps API and SDK Platform Access | HERE Developer* [online]. [vid. 2020-04-30]. Dostupné z: https://developer.here.com

9 Seznam obrázků

[1] Graf nejpopulárnějších databází světa. Zdroj: [3] 4
[2] Architektura MVC. Zdroj: [20] 16
[3] Spring Framework moduly. Zdroj: [23] 20
[4] Architektura technologie Redux. Zdroj: [28] 25
[5] Model tříd vyvíjené aplikace. Zdroj: [Autor] 30
[6] Diagram případů užití. Zdroj: [Autor] 31
[7] Drátový model profilu aplikace. Zdroj: [Autor] 32
[8] Drátový model hlavní zdi aplikace. Zdroj: [Autor] 33
[9] Struktura aplikace ReactJs, Spring Framework, MySQL. Zdroj: [Autor] 34
[10] Ukázka finálního designu zdi aplikace. Zdroj: [Autor] 39
[11] Ukázka finálního designu profilu uživatele. Zdroj: [Autor] 39

10 Seznam použitých ukázek kódu

[1] Příklad použití JSX. Zdroj: [autor] ... 9
[2] Tansformovaný kód ukázky 1. Zdroj: [autor] ... 9
[3] Typy zapsání funkcionální komponenty. Zdroj: [autor] ... 10
[4] Deklarovaná třída. Zdroj: [autor] ... 10
[5] Výrazová třída. Zdroj: [autor] ... 10
[6] Inicializace stavu. Zdroj: [autor] ... 12
[7] Neřiditelná komponenta za použití Ref. Zdroj: ... 12
[8] Vstup za použití Ref. Zdroj: [autor] ... 12
[9] Klasická a šipkovitá funkce. Zdroj: [autor] ... 13
[10] Použití promise/then. Zdroj: [autor] ... 13
[11] Použití async/await. Zdroj: [autor] ... 14
[12] Vytvoření a inicializace stavu. Zdroj: [autor] ... 14
[13] Použití hodnoty ve stavu. Zdroj: [autor] ... 14
[14] Použití props. Zdroj: [autor] ... 15
[15] Volání API za pomocí AJAXu a promise/then. Zdroj: [autor] ... 15
[16] Volání API za použití AXIOSu. Zdroj: [autor] ... 15
[17] Závislost objektu v klasickém přístupu. Zdroj: [autor] ... 22
[18] Použití Dependency Injection. Zdroj: [autor] ... 22
[19] Metoda v kontroléru vracející místa podle ID výletu. Zdroj: ... 23
[20] Metoda ve službě, která najde všechna místa. Zdroj: [autor] ... 23
[21] Metoda v repozitáři, která hledá daná místa v databázi. Zdroj: [autor] ... 24
[22] Mapování dotazu pomocí metody GET. Zdroj: [autor] ... 24
[23] Mapování dotazu pomocí @GetMapping anotace. Zdroj: [autor] ... 24
[24] Příklad akce, která volá API a vrací výsledek. Zdroj: [autor] ... 27
[25] Metoda získání míst z API. Zdroj: [autor] ... 36
[26] Výstup ze zavolané API. Zdroj: [36] ... 37
[27] Nastavení Google mapy s ukazateli a info okénky. Zdroj: [autor] ... 38
[28] Použití DirectionsRenderer uvnitř mapy. Zdroj: [autor] ... 38

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Marek Laušman**
Studium:	I1700103
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Implementace aplikace na plánování volného času**
Název bakalářské práce AJ:	Implementation of Free-time Planning App

Cíl, metody, literatura, předpoklady:

Cílem bakalářské práce je návrh a implementace aplikace pro plánování volného času s využitím získaných dat o uživateli.

Osnova: 1. Úvod, 2. Vývoj aplikací, 3. Přehled platforem, 4. Návrh aplikace, 5. Implementace aplikace, 6. Závěr

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Michal Macinka
Oponent:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	14.1.2018