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

Moderní frontendové frameworky

Bakalářská práce

Autor: Adam Černohorský
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové Březen 2020

Prohlášení:

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

V Hradci Králové dne 14.4.2020 Adam Černohorský

Poděkování

Děkuji svému vedoucímu doc. Ing. Filipu Malému, Ph.D. za veškerou pomoc, rady a odborné vedení při psaní bakalářské práce. Dále bych rád poděkoval Ing. Pavlovi Krbálkovi Ph.D. za možnost podílet se na vývoji interního frameworku Generali České Pojišťovny a za jeho cenné rady a pomoc během mého působení v jeho týmu.

Anotace

Cílem této práce je seznámit čtenáře s tvorbou moderního front-endu webových aplikací za použití JavaScriptu. V první části je čtenář seznámen s historií a základy JavaScriptu a jeho derivátů. Po představení JavaScriptu pak dochází k představení datových struktur, používaných nejčastěji pro komunikaci mezi front-endem a beck-endem aplikace a bundlovacích nástrojů, které jsou nezbytné pro práci s dalšími javascriptovými knihovnami.

Po úvodu do front-endu webových aplikací bude čtenář seznámen s knihovnou React JS a frameworkem Angular se zaměřením na teorii a principy, jak tyto technologie fungují a jak by mělo docházet k jejich použití.

V závěru této práce pak aplikujeme znalosti, které jsme se dozvěděli v předchozích kapitolách a za použití knihovny React JS a frameworku Angular vytvoříme dvě totožné aplikace, na kterých si názorně ukážeme, jakým způsobem je možné vytvořit moderní javascriptový front-end.

Annotation

Title: Modern Frontend Frameworks

The goal of the bachelor thesis is to acquaint reader with the creation of modern front-end web applications using JavaScript. In the first part the reader is acquainted with the history and basics of JavaScript and its derivates. After the introduction of JavaScript, there are introduced data structures, which are mostly used for communication between application front-end and beck-end and bundling tools, which are necessary for working with others JavaScript libraries.

After the introduction to front-end of web applications, the reader will be introduced to the React JS library and Angular framework, focusing on the theory and principles of how these technologies work and how they should be used.

At the end of this work we apply the knowledge we learned in the previous chapters. Using React JS library and the Angular framework we will create two identical applications where we will show how to create a modern JavaScript front-end.

Obsah

1 Úvod

Již je to nějaký pátek, kdy za front-end webových aplikací zodpovídal server a dominovaly technologie jako Java Server Pages tzv. *SPA* a JavaScript sloužil pouze pro vizuální efekty. Postupem času se začala ukazovat síla JavaScriptu a jeho použití na straně klienta bylo čím dál více sofistikované. Roku 2006 pak vznikla knihovna jQuery, která manipulovala s Document Object Modelem tzv. *DOMem* a oddělovala tak logiku aplikace z Hypertext Markup Language tzv. *HTML* do JavaScriptu a na dlouhou dobu se stala dominantní knihovnou pro tvorbu front-endu webových aplikací (32). Roku 2010 pak přišel framework Angular JS, dále známý spíše jako Angular 1, který zavedl na front-endu architekturu Model View Controller tzv. *MVC*, který oddělovala aplikační logiku od zobrazovací, která pak byla pomocí speciálních značek vkládána do HTML (33). Angular JS stejně jako jQuery manipuloval s DOMem a postupem času se ukázalo, že manipulace s reálným DOMem při složitých operací může být velmi pomalá a díky tomu vznikla knihovna React JS a posléze i framework Angular 2, který již nemanipuluje s reálným DOMem, ale používá tzv. *shadow DOM*, který následně hledá nejrychlejší cestu, jak změny projevit v reálném DOMu.

V této práci si představíme moderní koncept javascriptového front-endu a ukážeme si, jak je možné tyto koncepty využít za pomocí technologií React JS a Angular 2. Cílem této práce pak je seznámit čtenářem s problematikou javascriptového fronte-endu, objasnit jakým způsobem fungují moderní webové aplikace, jakým způsobem dochází ke komunikaci mezi klientem a serverem a v neposlední řadě blíže představit výše zmíněné technologie a ukázat, jakým způsobem je možné za využití těchto technologií postavit moderní front-end webové aplikace. Závěrem získané informace z praktické části použijeme pro porovnání a doporučení, kdy a jakou technologii zvolit.

Postupy a metodologie použité v praktické části v sobě kombinují informace z teoretické části práce spolu se zkušenostmi z praxe získané během mého 2letého působení v Generali České Pojišťovně, kde jsem působil jako front-endový vývojář interního frameworku postaveném na technologii React JS, který měl za úkol sjednotit IT a byl použit pro přepis starých aplikací z Angularu a dále při vzniku všech dalších nových fronte-endových aplikací v Generali České Pojišťovně.

2 JavaScript

JavaScript je multiplatformní, objektově orientovaný, dynamicky typovaný, interpretovaný skriptovací jazyk, jehož syntaxe vychází z jazyka Java. (2)(3)

JavaScript sice připomíná Javu, ale nemá její statické typování ani silnou typovou kontrolu. JavaScript má místo běžného objektového modelu tvořeného třídami prototypový objektový model. Ten poskytuje dynamické dědění, což znamená, že zděděné vlastnosti se mohou napříč jednotlivými objekty lišit. (1)(2)

JavaScript je oproti Javě velmi uvolněný jazyk. Nemusíte deklarovat všechny proměnné, třídy a metody. Nemusíte se starat o to, jestli jsou public, private nebo protected a nemusíte implementovat žádná rozhraní. Proměnné, parametry a návratové typy funkcí nejsou explicitně typovány (2).

2.1 Historie

Roku 1995 společnost Netscape Communications Corporation chtěla vytvořit dostupnější programování, než jaké bylo možné v Javě a zároveň vytvořit podporu Javy v Netscape Navigatoru, která by byla otevřenější pro programátory, kteří neprogramují v Javě (1). Na tuto práci si společnost najala nového vývojáře jménem Brendan Eich, který rozhodl, že vytvoří skriptovací jazyk, který by byl rychlý a zároveň jednoduše použitelný (1)(3). Tento nový skriptovací jazyk byl nejprve pojmenován jako LiveScript (1), jehož název byl později změněn na JavaScript. Ačkoli byl JavaScript vyvinut společností Netscape, Sun Microsystems, Inc. měl na Javu ochranné známky, pod které spadá i jméno JavaScript (1).

Vznik nového Jazyka byl oznámen 4. prosince roku 1995 (1) a původně byl uveden jako doplněk k Javě a HTML, než jako jednoduchý a podpůrný jazyk pro Javu. Nicméně JavaScript vývojáři rychle přijali a navzdory různým problémům s ošetřením chyb a později i bezpečností, JavaScript slavil úspěch. Postupem času, kdy prohlížeče byly stále výkonnější a umožňovaly stále rychleji spouštět skripty, se z JavaScriptu stal silný životaschopný vývojářský nástroj.

Jakmile začal JavaScript získávat ohlas, společnost Microsoft, hlavní konkurent Netscape, ho již nemohla déle ignorovat a rozhodla se vytvořit vlastní verzi JavaScriptu, která byla nazvána jako JScript (1). První oficiální verze JScriptu

byla vydána 16. července 1996 v Internet Explorer 3.0 (1). Původní JScript byl pouze redukovanou verzí JavaScriptu 1.1 a rozdíly zbytečně mátly programátory.

Microsoft později vydal další verzi prohlížeče Internet Explorer 3.0, aby dodal svému JScriptu kompletní podporu JavaScriptu 1.1. Od této chvíle začaly problémy mezi prohlížeči a jejich verzemi, které komplikovaly život programátorům. Programátoři začali zjišťovat, že ačkoliv JScript vypadá stejně jako JavaScript, některé skripty fungující v Netscape nefungují v Internet Exploreru a naopak. Ve snaze zarazit chaos, Netscape a Sun začali hledat třetí stranu, která by standardizovala JavaScript a obrátili se na European Computer Manufacturers Association. (1)(3)

2.2 ECMAScript

Jak již bylo zmíněno v kapitole 2.1, JavaScript byl standardizován společností Ecma International, dříve známou pod jménem European Computer Manufacturers Association pod standardem ECMA-262 (4).

ECMAScript je založený na nám dobře známém JavaScriptu, který byl vyvinutý společností Netscape a již méně známém JScriptu, který vytvořil Microsoft v reakci na rostoucí popularitu JavaScriptu (1). Samotný vývoj specifikace začal v listopadu 1996 a první verze spatřila světlo světa v červenci 1997 (4).
ECMAScript specifikace nepopisuje *Document Object Model* tzv. *DOM*, který je standardizovaný organizací World Wide Web Consortium neboli *W3C* (2).

2.2.1 ECMAScript 5.1

V současné době prohlížeči nejvíce podporovaná verze ECMAScriptu je verze 5.1, která byla vydána v červnu roku 2011 jako oprava verze 5.0. Tato verze, tedy ES5.1 je zároveň poslední verzí s tímto stylem číslování, jelikož od dalších verzí došlo ke změně vydávání verzí a to tak, že nové verze vychází každý kalendářní rok. (4)

2.2.2 ECMAScript 2015

ECMAScript 2015, který je mezi vývojáři také známí pod názvem **ES6** vyšel 6 let po vydání poslední verze ES5. ES2015 přináší mnoho výrazných změn a jedná se tak o první velké vylepšení jazyka od verze ES5. (5)

Let

Let je nový druh pojmenování, který je na rozdíl od *var* viditelná pouze v rámci jednotlivého bloku kódu tzv. *scope* a jeho zanořených částí které obsahuje (5). Zároveň je vhodné upustit od používání pojmenování *var* a nahradit ji *let*. Definování *let* mimo funkci na rozdíl od *var* nevytváří globální proměnnou (5).

```
function variableUnderScope() {
   let scopeVar = 1;
}
```

Const

Pokud proměnnou definujeme jako *let* nebo *var* je možné později v programu jejich hodnotu změnit a znovu přidělit. Jakmile je c*onst* inicializován, jeho hodnota se již nezmění a takto definované proměnné nemůže být přidělena jiná hodnota. Zároveň stejně jako *let* i c*onst* je viditelná pouze v rámci konkrétního bloku kódu. *Const* neposkytuje imutabilitu, ale pouze zajišťuje, že reference nemůže být změněna. (5)

```
const a = 1;
a = 2; // Attempt to assign to const or readonly variable
```

Arrow funkce

Jakmile byly *arrow funkce* představeny, zásadně změnily, jak kód v JavaScriptu vypadá a funguje. *Arrow funkce* využívají zápisu *=>*, který je podobný lambdám v Javě 8. Přebírají *this* ze *scope*, kde je funkce definována, což se využívá například při definování *class komponent* v Reactu, jelikož s použitím *arrow funkce* není třeba již dále *bindovat* funkce uvnitř komponenty. (5)

Jako další funkcionalitu pak *arrow funkce* přináší implicitní *return*, což v praxi znamená, že již není třeba používat klíčové slovo *return*, ale je možné ho nahradit kulatými závorkami nebo žádnými závorkami v případě jednoduchého výrazu. Při použití složených závorek, je třeba *return* opět použít.

V případě práce s událostmi tzv. *eventy*, *DOM event listener* nastavuje *this* pro cílený element a v tomto případě je třeba použít normální funkci namísto *arrow funkce* (5).

```
function oldSyntax() {
   return 'hello world'
}

const arrowFunction = () => {
   return 'hello world';
};

const defaultReturn = () => (
   'hello world'
);

const defaultReturn2 = () => 'hello world';

const arrowWithParam = (a, b) => {
   const first = a + b;
   const second = a * b;
   return (
      `first value is: ${first}, second value is: ${second}`
   );
};
```

Template literals

Template literals jsou definovány pomocí zpětných uvozovek a přinášejí nové způsoby, jak pracovat s textovými řetězci, tzv. *stringy*. Jedním z využití je například definice více řádkového *stringu*, kdy oproti starému přístupu stačí pouze použít *enter* a dojde k vytvoření nové řádky. Jedinou drobnou nevýhodou pak může být fakt, že se do výsledku promítne každá mezera, takže je třeba si dát pozor na odřádkování. Další způsob využití *template literals* je vkládání proměnných do *stringu*, které je možné pomocí symbolu *${...}*. Uvnitř složených závorek pak může být proměnná nebo jakýkoliv výraz. (5)

```
const a = 'value';

const templateLiteral = `the value is ${a}`;
```

Třídy

JavaScript má takzvanou prototypální dědičnost, která na rozdíl od jazyků založených na třídách *(Java, C#, ...)* funguje na jiném principu a mnoho programátorů může mít problémy pochopit, jak s ní pracovat (5). ES6 proto přináší syntaktický cukr v podobě tříd, které se navenek chovají jako třídy nicméně uvnitř stále fungují na principu objektového prototypu.

```
class Person {
   constructor(name) {
      this.name = name;
   }

   hello() {
      return `Hello, my name is ${this.name}`;
   }
}
```

Promise

Příslib budoucí hodnoty neboli *promis* je jeden ze způsobů, jak se vypořádat s asynchronním kódem bez psaní velkého množství tzv. *callback* funkcí *(Callback = zpětné volání)*. Ačkoliv *promise* existují již déle, k jejich standardizaci došlo až v ES2015 a v ES2017 byly nahrazeny asynchronními funkcemi, které jsou, ovšem postaveny nad *promise* Application Programming Interface tzv. *API*. (5)

Promise nepředává funkci callback, ale objekt, který funguje jako příslib budoucí hodnoty, kterou funkce jednou vrátí. S tímto objektem lze pak dále pracovat, předávat ho a tím je nezávislý na čase. Výsledek *promise*, pak získáme pomocí *then()* (5). *Promise*, využívají moderní web *API* jako je například *Fetch* nebo *Service Workers*.

Moduly

Modul je javascriptový soubor, který exportuje jednu nebo více hodnot (objekty, funkce, proměnné) za pomocí klíčového slova ***export***. Tyto exportované hodnoty se pak stávají viditelné ostatním javascriptovým souborům, které si je mohou pomocí klíčového slova ***import*** importovat stejně jako například různé knihovny. (5)

Addition.js

```
export const addition = (a, b) => {
   return a + b;
};
```

Usage.js

```
import { addition } from './Addition.js';

console.log(addition(2, 4)); // console log => 6
```

2.2.3 ECMAScript 2016

ECMAScript 2016 byla první edice ECMAScriptu vydaná pod Ecma TC39's, což je nový způsob vydávání verzí ECMAScriptu každý rok. Dokumentace k ES2016 vychází z dokumentace ES2015 a slouží jako předloha pro vydávání dalších verzí. Mezi novinky pro ES2016 patří například nový exponenciální operátor nebo nová metoda *includes*, která se využívá při práci s poli. (6)

Exponenciální operátor

```
const squareTheParam = (a) => a**2;

console.log(squareTheParam(2)); // 4
```

Array.includes

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

console.log(array.includes(2)); // true
console.log(array.includes(-2)); // false
console.log(array.includes('2')); //false
```

2.2.4 ECMAScript 2017

Mezi vybrané funkcionality ES2017 patří:

Object.entries

Do objektu byla přidána nová metoda *entries*, která jako parametr bere objekt, který má klíče a hodnoty a jako výstup této metody je dvou dimenzionální pole, které vrací klíče a jejich hodnoty. (7)

```
const data = {
   'key1':'value1',
   'key2':'value2',
   'key3':'value3',
};

console.log(Object.entries(data)); // [ ['key1', 'value1'], ['key2',
'value2'], ['key3', 'value3']]
```

Object.values

Funguje na podobném principu jako metoda *entries*, s tím rozdílem, že na rozdíl od dvou dimenzionálního pole vrací pole s jednou dimenzí, které obsahuje pouze hodnoty. (7)

```
const data = {
   'key1':'value1',
   'key2':'value2',
   'key3':'value3',
};

console.log(Object.values(data)); // ['value1', 'value2', 'value3']
```

Async/Await

Nově je možné využít *async/await* k zpřehlednění kódu od zbytečných *callback* funkcí a vytvořit tak čistější asynchronní funkce. (7)

```
async function fetchJson(url) {
   try {
      const request = await fetch(url);
      const text = await request.text();
   } catch (e) { console.error(e) }
}
```

2.2.5 ECMAScript 2018

Mezi vybrané novinky ES2018 patří:

Object rest properties

V případě destrukturalizace objektu, *object rest properties* umožní sebrat zbývající atributy objektu a vytvořit z nich objekt nový. (8)

```
const input = {a: '1', b:'2', c:'3', d:'4'};
const {a, b, ...others} = input;

console.log(a); // 1
console.log(b); // 2
console.log(others); // {c: '3', d: '4'}
```

Promise finally

Promise.prototype.finally() ukončuje celou *promise* a umožňuje zaregistrovat *callback*, který bude zavolán, v případě dokončení *promise*, jak úspěchem, tak neúspěchem (8). Nejběžnější využití této nové funkcionality je například schování komponenty symbolizující dotahování dat po jejich dotažení.

```
fetch(url)
   .then(result => {
      // do something
   })
   .catch((err) => {
      // do something while error
   })
   .finally(() => {
      // e.g. close spinner
   });
```

2.3 Využití JavaScriptu

Ačkoli byl JavaScript v minulosti využíván především k rozblikání stránky (1), postupem času si vývojáři uvědomili jeho potenciál a v současné době se JavaScript využívá napříč všemi platformami. (3)

2.3.1 Web

Už dávno neplatí, že JavaScript slouží pouze k přidání efektů na webové stránce, ale s pomocí JavaScriptu lze přenést část výpočtů ze serveru na klienta a ulevit tak zátěži serveru. (3) Tohoto přístupu se v současné době nejvíce využívá u single-page aplikací tzv. *SPA*, kde klient pomocí *API* zažádá o data a klientská část se postará o jejich zobrazení uživateli. Mimo pouhé zobrazování dat, klientská část aplikace také může využívat základních operací s daty včetně použití validací na straně klienta, což vede k odchycení některých chyb předtím, než se data dostanou na serverovou část aplikace a díky tomu ušetří zatížení serveru.

2.3.2 Server

Jak bylo již zmíněno v kapitole 2.1, JavaScript byl dílem společnosti Netscape a měl sloužit především pro jejich prohlížeč Netscape Navigator, nicméně Netscape měla zároveň svůj obchodní plán pro webové servery, což zahrnovalo prostředí zvané Netscape LiveWire. Bohužel pro Netscape, LiveWire nebyl úspěšný tah a k popularitě a velkému rozšíření JavaScriptu na serveru došlo až s nástupem Node.js. (9)

Node.js je veřejně dostupné tzv. *open source*, cross-platform javascriptové prostředí postavené na *Chrome V8* JavaScript enginu od společnosti Google a událostní smyčce tzv. *event loop*, jehož primární účel je tvorba serverové části webové aplikace (10). Node.js aplikace běží v jednom procesu a nevytváří nová vlákna pro každý nový dotaz. Node.js poskytuje sadu asynchronních I/O primitiv, které brání javascriptovému kódu před zablokováním. Když Node.js potřebuje provést I/O operaci jako je čtení ze sítě, nebo přístup k databázi, tak namísto zablokování vlákna, Node.js obnoví operace, jakmile se vrátí odpověď (9). Díky tomuto přístupu, Node.js může obsloužit tisíce souběžných připojení jediným

serverem bez nutnosti spravovat přidělování vláken v případě více souběžných připojení.

Node.js má velikou výhodu, jelikož miliony front-end vývojářů po celém světě, kteří píší javascriptový kód pro prohlížeče teď mohou zároveň psát serverovou stranu webové aplikace bez nutnosti učit se dalšímu jazyku.

2.3.3 Desktop

O JavaScriptu se říká, že lze použít všude a jinak tomu není ani u desktopových aplikací, které mají tu výhodu, že díky JavaScriptu jsou podporovány všemi platformami. Budování javascriptové desktopové aplikace je založeno, stejně jako webové aplikace, na *HTML* a Cascading Style Sheets tzv. *CSS* (11). Pro vytvoření javascriptové desktopové aplikace slouží frameworky jako jsou například AppJS, OS.js nebo Electron, který patří k nejznámějším a nejpoužívanějším. Electron kombinuje Chromium a Node.js do jednoho runtime, což umožňuje spouštět HTML, CSS a JavaScript jako desktopovou aplikaci (11).

2.3.4 Mobilní zařízení

Mobilní aplikace pro různé platformy využívají rozdílných programovacích jazyků a díky tomuto přístup, pokud chceme napsat stejnou aplikaci pro rozdílné platformy, typicky iOS a Android, musíme aplikace napsat pro každou platformu zvlášť. Tento přístup je poměrně časově i zdrojově náročný, nicméně JavaScript nám přináší řešení tohoto problému, kdy napíšeme jednu aplikaci, která bude platformě nezávislá.

Mezi populární nástroje pro budování nativních cross-platform aplikací pak patří například Tabris.js, nebo React Native od Facebooku, který je velice podobný samotnému Reactu, nicméně jako stavební kameny používá namísto webových komponent komponenty nativní. React Native Facebook aktivně využívá k vývoji aplikací jako je Facebook, Instagram či Pinterest. (12)

2.4 Typování proměnných v JavaScriptu

JavaScript patří do jazyků s dynamickou typovou kontrolou, což znamená, že kontrola typu proměnných probíhá až za běhu programu a není tedy požadováno deklarovat datové typy u proměnných, které díky tomu mohou odkazovat na hodnotu jakéhokoliv typu (3). Dynamické typování může být pro vývojáře pohodlnější a méně časově náročné, nicméně na druhou stranu díky kontrole typů až za run-time není způsob, jak odchytit případné chyby díky špatnému datovému typu. Tyto chyby se pak projeví až za běhu programu a většinou jsou hůře dohledatelné než chyby odhalené již při kompilaci. Abychom minimalizovali runtime chyby, existují způsoby, jak provádět kontrolu typů proměnných, a díky tomu pak snadněji identifikovat místo problému. Toto přidané typování proměnných je pak především užitečné v případě větších aplikací.

2.4.1 TypeScript

TypeScript je *open-source* programovací jazyk vytvořený a spravovaný společností Microsoft, který byl navržen především pro vývoj velkých aplikací. Jedná se o nadstavbu nad jazykem JavaScript, která přidává možnost striktního typování proměnných, definování tříd, rozhraních či generické datové typy. Díky těmto přidaným vlastnostem se z TypeScriptu stává oblíbený jazyk pro budování aplikací s velkým množstvím javascriptového kódu, a to především mezi vývojáři, kteří jsou zvyklý na objektově orientované programovací jazyky, jako je například Java.

TypeScript se kompiluje do JavaScriptu a jelikož je nadstavbou nad JavaScriptem, tak zároveň platí, že každý javascriptový program je validním typescriptový programem. TypeScript může být využitý k vývoji javascriptových aplikací jak na straně serveru, tak na straně klienta. (13)

2.4.2 Proptypes

Proptypes jsou podpůrnou knihovnou vytvořenou společností Facebook, která byla původně součástí jádra knihovny React, ale později došlo k jejímu vyčlenění. Proptypes slouží především pro runtime kontrolu typu *props* v React komponentách. Proptypes zkontrolují *props*, které přichází do komponent, kde je

porovná s jejich definicí a v případě, že přicházející *props* nesedí s definovaným typem, tak o této skutečnosti varuje vývojáře v podobě varování v konzoli.

Proptypes tedy nezabraní na rozdíl od TypeScriptu v kompilaci kódu, ale pouze uživatele upozorní, že dochází k použití špatného typu, než jaký uživatel definoval. (14)

2.4.3 Flow

Flow na rozdíl od konkurenčního TypeScriptu není sám o sobě přímo programovacím jazykem, ale jedná se o statický typový kontrolér tzv. *Static Type Checker* pro JavaScript, což znamená, že Flow je nástroj, který je možné stáhnout a nainstalovat do vývojového prostředí, kde bude analyzovat a generovat informace o kódu. (15)

JavaScript na rozdíl od staticky typovaných jazyků automaticky nekontroluje správnost typu proměnných a v případě, že narazí na nekonzistenci typů, tak namísto vyhození chyby může dojít k tomu, že interpret je schopný v případě potřeby provedení určité operace automaticky převést číslo uložené jako *string* na *number.* Toto chování se může jevit jako výhoda, nicméně na druhou stranu může dojít k tomu, že se program chová jiným způsobem, než programátor zamýšlel a jelikož JavaScript na takovou chybu neupozorní, tak její dohledání může být mnohdy složité a časové náročné. (15)

Cílem Flow je odhalit potenciální část kódu, která by se mohla chovat neočekávaným způsobem díky nekonzistenci datových typů a poradit programátorovi, jak tento problém vyřešit. Mimo samotnou kontrolu datových typů Flow zároveň poskytuje standardní podporu pro IDE jako je zvýrazňování chyb, automatické doplňování či automatické refaktorování kódu. (15)

3 JSON

JSON neboli *JavaScript Object Notation* je druh syntaxe textového formátu, který pomáhá usnadnit výměnu strukturovaných dat, jako je například komunikace pomoci *AJAX* mezi serverovou a klientskou částí aplikace (18). Tento formát je díky svým pevně daným pravidlům do jisté míry snadný pro čtení a zápis člověkem a zároveň velice snadno čitelný a generovatelný strojově.

Syntaxe JSON je založena na definování klíčů a jejich hodnot, které se definují pomocí složených závorek, hranatých závorek, dvojteček, uvozovek a čárek. Jako hodnoty této datové struktury pak mohou být ukládány všechny primitivní datové typy, jako jsou například čísla, desetinná čísla, textové řetězce, boolean hodnoty či *null*. Mimo tyto jednoduché hodnoty je také možné ukládat objekty a pole. (17)

```
{
  "firstName" : "Adam",
  "lastName" : "Černohorský",
  "birthNumber" : "950626/4256",
  "address" : {
    "country" : "Czech Republic",
    "city" : "Hradec Králové",
    "street" : "Labská Kotlina",
    "zipCode" : 50002
  },
  "married" : false,
  "documents" : ["document1", "document2"],
  "other" : null
}
```

3.1 Struktura JSON

Objekt

Objekt v JSON není plnohodnotný objekt, který můžeme znát z JavaScriptu, ale jedná se o kontejner, který obsahuje pouze serializovatelá data. Každá hodnota objektu má svůj vlastní klíč. Do objektu je možné vkládat další objekty nebo pole, díky čemuž můžeme snadno vytvářet složitější struktury. (17)

Pole

Pole je seřazenou kolekcí hodnot, které na rozdíl od objektu, neobsahuje klíče, ale pouze 0 až N hodnot, kde samotné hodnoty dále mohou být objektem nebo dalším polem. (18)

Řetězec

Řetězec musí být vložený do dvojitých uvozovek a může obsahovat všechny znaky Unicode. V případě problémů s diakritikou, je možné znaky vložit ve tvaru „uXXXX“, kde „XXXX“ značí kód znaku z Unicode tabulky zapsaný v šestnáctkové soustavě. *Char* je reprezentovaný jako řetězec s jediným znakem. (18)

3.2 XML

eXtensible Markup Language, zkráceně XML je značkovací jazyk, který definuje sadu pravidel pro kódování dokumentů ve formátu, který je čitelný pro člověka i stroj. Formát XML definovalo konsorcium W3C jako formát pro přenos obecných dokumentů a dat. Jedná se o textový formát dat se silnou podporou od Unicode po jakékoliv lidské jazyky. Ačkoliv se XML specializuje na dokumenty, tak se zároveň velmi často používá pro reprezentaci různých datových struktur, například těch, které se používají ve webových službách. XML se svou strukturou velice podobá HTML, nicméně na rozdíl od HTML neobsahuje žádné předem definované tagy, a jejich definici pak nechává plně na jejich uživateli, což napomáhá k lepšímu pochopení dané struktury. Mezi další možnosti XML je například možnost definování výsledného vzhledu tisků či generování souborů do PDF (19).

Deklarace XML

XML dokument by měl začínat deklarací XML, která o sobě podává některé informace, jako je verze či kódování. Příkladem deklarace pak může být *<xml version=“1.0“ ecoding=“UTF-8“?>* (19)

Validní znaky

Dokument XML je řetězec znaků a jako znak se může objevit téměř každý platný znak Unicode. (19)

Procesor a aplikace

Procesor analyzuje značení a předává strukturované informace do aplikace. Specifikace ukládá požadavky na to, co musí procesor XML dělat. Procesor je často označován jako XML parser. Mezi používané parsery pak patří *DOM parser*, který vezme XML dokument a vytvoří z něj obraz v paměti, nebo *SAX parser*, který postupně prochází XML dokument a vyvolává události, které pak programátor zpracovává. (19)

Značky a obsah

Znaky tvořící XML dokument se dělí na značky a obsah, což lze odlišit použitím jednoduchých syntaktických pravidel. Řetězec, který začíná znakem „<“ a končí znakem „>“ nebo začíná znakem „*&*” a končí znakem „;“ je značkou a ostatní řetězec je pak obsahem. (19)

Tag

Tag je značkovací konstrukt, který začíná znakem „<“ a končí znakem „>“. Tagy se dají rozdělit na start-tag *(<something>),* end-tag *(</something>)* a empty-element tag *(</something>)* (19)

Element

Element je složka logického dokumentu, která buď začíná start-tagem a končí odpovídajícím end-tagem, nebo se skládá pouze z empty-element-tagu. Znaky mezi start-tagem a end-tagem jsou obsahem elementu a mohou obsahovat další vnořené elementy, které se nazývají child elementy. Příklade XML elementu může být: *<element> obsah elementu </element>* (19)

Atribut

Atribut je konstrukt skládající se z dvojice název-hodnota, která existuje uvnitř start-tagu nebo empty-element tagu. Příkladem může být *<author name=“John“ age=“25“> content </author>.* XML atribut může mít pouze jednu hodnotu a jeden atribut se může objevit v rámci elementu pouze jednou. (19)

4 Babel a Webpack

Kompilátor převádí zdrojový kód jazyka z vyšší úrovně abstrakce na úroveň nižší, nejčastěji pak do strojového kódu. Transpilátor na rozdíl od kompilátoru pak převádí zdrojový kód jednoho programovacího jazyka na stejný zdrojový kód jiného programovacího jazyka v rámci stejné úrovně abstrakce. (20)

4.1 Babel

Babel je javascriptový transpilátor, který je schopný převést prohlížeči nepodporované nové konstrukty JavaScriptu do starého všemi prohlížeči podporovaného ES5. Díky tomu mohou programátoři psát kód dle posledních konvencí a nemusí řešit, jestli je daná verze podporovaná prohlížečem. (20)

Ukázka kódu

```
const numbers = [ 5, 10, 15];
console.log(numbers.map(number => number + 5));
```

Babel převede do:

```
var numbers = [ 5, 10, 15];
console.log(numbers.map(function (number) {
   return number + 5;
}));
```

4.1.1 Babel 7

Babel 7 je posledním major updatem Babelu a přichází s několika novinkami, jako jsou například: (21)

Babel upgrade

Je nový nástroj, který automaticky upgraduje změny v souborech *package.json* a *.babelrc*. Pro použití upgrade nástroje je možné použít buď příkaz *npx babel-upgrade* nebo *npm i babel-upgrade –g.* (21)

JavaScript configuration files

Babel 7 také představil nový soubor *babel.config.js*, který slouží podobně jako *.babelrc* nicméně jeho konfigurace je za použití JavaScriptu namísto JSONu. Tento soubor se dá použít například při konfiguraci projektu či kompilaci *node_modules*. Přidáním tohoto souboru do projektu, Babel snadněji vyřeší konfiguraci namísto vyhledávání z každého souboru, dokud nenajde konfigurační kód. Díky tomuto způsobu konfigurace je také snazší provádět overrides. (21)

Babel.config.js

```
module.exports = function () {
   const presets = [ ... ];
   const plugins = [ ... ];

   return {
      presets,
      plugins
   };
};
```

Overrides

Overrides v Babelu umožní specifikovat různé konfigurace, což je užitečné v případě, že projekt potřebuje různé konfigurace kompilace pro testovací soubory, kód klienta nebo serveru. Namísto definování souboru *.babelrc* pro každou konfiguraci je možné veškeré konfigurace definovat v rámci souboru *babel.config.js* (21)

Macros

Babel začal jako transpilátor z ES6 do ES5, ale dnes je to již mnohem více. Existují stovky pluginů, které lze použít pro určité knihovny a případy užití pro zlepšení celkového výkonu aplikace. Přidávání těchto pluginů do aplikace bohužel není tak jednoduché, jak by se na první pohled mohlo zdát. Například pokud pro vytvoření aplikace použijeme *create-react-app*, pak již nebude možné tyto pluginy používat. Řešením může být instalace balíčku *babel-plugin-macros*. Tento balíček se postará nejen o konfiguraci kódu, aby byl kompatibilní s pluginy, ale také usnadňuje psaní vlastní transformace pro unikátní scénáře aplikace. (21)

4.2 Webpack

Webpack je statický module bundler pro moderní javascriptové aplikace. Když webpack zpracovává aplikaci, interně vytváří závislostní graf, který mapuje každý modul, který projekt potřebuje a generuje jeden, nebo více balíčků. (22)

4.2.1 Moduly

V modulárním programování (programovací paradigma), vývojáři rozdělují program na malé části funkcionalit zvané moduly.

Každý modul zajišťuje pouze určitou funkcionalitu celého programu, což výrazně usnadňuje případné hledání chyb a pokrytí funkcionality modulu testy. Dobře napsané moduly poskytují pevnou abstrakci a zapouzdření, takže každý modul má ucelený design a jasný účel v rámci celé aplikace.

Node.js podporuje modulární programování téměř od svého počátku, nicméně na webu podpora modulů dorazila pomaleji. Existuje několik nástrojů, které podporují modulární programování JavaScriptu na webu s celou řadou výhod a omezení. Webpack staví na zkušenostech získaných právě z těchto nástrojů a aplikuje koncept modulů na jakýkoliv soubor v projektu. (22)

4.2.2 Graf závislostí

Jakmile jeden soubor závisí na dalším, webpack to považuje za závislost, což umožňuje webpacku přijímat také nekódové soubory, jako jsou obrázky nebo fonty písma a zařadit je do závislostí aplikace. Když webpack zpracovává vaši aplikaci, začíná ze seznamu modulů definovaných v příkazovém řádku nebo konfiguračním souboru. Počínaje těmito vstupními body, webpack rekurzivně vytváří závislostní graf, který zahrnuje každý modul, který vaše aplikace potřebuje a poté všechny tyto moduly spojí do malého počtu balíčků – nejčastěji jednoho, který si načte prohlížeč. (22)

4.2.3 Základní principy

Entry

Vstup neboli *entry point* označuje modul, který by měl webpack použít k zahájení vytváření interního grafu závislostí. Webpack zjistí na kterých dalších modulech a

knihovnách *entry point* přímo či nepřímo závisí. Ve výchozím nastavení je *entry pointem ./src/index.js,* ale je možné určit vlastní *entry point* (či *entry pointy*) konfigurací vlastnosti entry ve webpack konfiguraci, jak je ukázáno v kódu níže: (22)

webpack.config.js

```
module.exports = {
   entry: './path/to/entryPoint.js'
};
```

Output

Výstup neboli *output* říká webpacku, kde má vytvořit výstupní soubory a jak tyto soubory pojmenovat. Výchozí nastavení je *./dist/main.js* pro hlavní výstupní soubor a do složky *./dist* pro jakýkoliv další vygenerovaný soubor. Konfigurace vlastnosti *output* říká webpacku, jak zapsat kompilované soubory na disk. Na rozdíl od *entry point*, kterých může být více, existuje pouze jeden *output*. (22)

webpack.config.js

```
module.exports = {
   filename: 'bundle.js'
};
```

Loaders

Webpack v základu rozumí pouze souborům typu JavaScript a JSON. Loadery dovolují webpacku zpracovávat i další typy souborů a převádět je do validních modulů, které mohou být zpracovány a přidány do grafu závislostí. Loadery nejčastěji používají dvě základní vlastnosti, a to vlastnost *test*, která říká, který soubor nebo soubory mají být transformovány a vlastnost *use*, která říká, který loader by měl být použit k provedení transformace. Použití těchto vlastností je ukázáno v následující části kódu. (22)

```
Module.exports = {
   module: {
      rules: [
         { test: /\.css$/, use: 'css-loader' },
         { test: /\.ts$/, use: 'ts-loader' }
      ]
   }
};
```

Plugins

Zatímco se Loadery používají k transformaci určitých typů modulů, pluginy mohou být využívány k provádění širšího spektra úkolů, jako jsou například bundle optimalizace, assets management nebo injekce proměnných prostředí. Pokud chceme použít plugin, je třeba použít *require()* a přidat ho do pole pluginů. Většina pluginů je customizovatelná skrze nastavení. Vzhledem k tomu, že je možné jeden plugin použít pro více účelů, je třeba vytvořit jeho instanci zavoláním pluginu pomocí klíčového slova *new*. (22)

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed
via npm
const webpack = require('webpack'); //to access built-in plugins

module.exports = {
   module: {
      rules: [
         { test: /\.txt$/, use: 'raw-loader' }
      ]
   },
   plugins: [
      new HtmlWebpackPlugin({template: './src/index.html'})
   ]
};
```

Mode

Nastavením parametru mode na *development*, *production* nebo *none*, můžete povolit vestavěné optimalizace, které odpovídají jednotlivým prostředím. Výchozí hodnotou je *production mode*. (22)

5 React JS

React je deklarativní, efektivní a flexibilní javascriptová knihovna vytvořená společností Facebook, která byla zveřejněna 29. května 2013 (23). React JS může být použit jako základní stavební kámen *single-page* nebo mobilních aplikací, kde slouží především pro vytváření uživatelských rozhraní, která se skládají pomocí malých, znovupoužitelných, izolovaných částí kódu známých jako komponenty. Jelikož se React JS používá především pro tvoru uživatelského rozhraní, tak v případě realizace složitějších a komplexnějších webových aplikací se spolu s touto knihovnou dále často používají knihovny jako jsou například Redux pro state management, React-Router pro směrování či Axios pro interakci s *API*.

V rámci celé této kapitoly bylo převážně čerpáno ze zdroje (23). V případě, že došlo k čerpání z jiných zdrojů, bude tak náležitě specifikováno.

5.1 ReactDOM

Knihovna ReactDOM je nedílnou součástí knihovny React, od které se ve verzi 0.14 oddělila. ReactDOM poskytuje metody specifické pro práci s *DOMem*, které lze použít na nejvyšší úrovni. ReactDOM obsahuje následující metody. (24)

Render

Tato metoda vykresluje neboli *renderuje* React elementy do *DOMu* do cílového kontejneru a vrací referenci na komponentu, nebo *null* pro bez stavové tzv. *stateless* komponenty. Pokud byl React element již dříve vykreslen do kontejneru, provede se na daném elementu update a DOM se aktualizuje pouze v případě potřeby. Dále je možné v rámci této metody definovat *callback* funkci, které se zavolá pokaždé, když dojde k vykreslení nebo updatu komponenty.

Definice metody:

```
ReactDOM.render(element, container[, callback])
```

Příklad užití:

```
import React from 'react';
import { render } from 'react-dom';
import App from './containers/App';

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

Hydrate

Tato metoda je téměř stejná jako metoda *render()*, nicméně liší se tím, že se používá k hydrataci kontejneru jehož *HTML* obsah byl vykreslen za pomocí *ReactDOMServer*.

Definice metody:

```
ReactDOM.hydrate(element, container[, callback])
```

UnmountComponentAtNode

Odebere připojenou React komponentu z *DOMu* a pročistí její funkce obsluhy událostí tzv. *event handlery* a stav tzv. *state*. Pokud v kontejneru nebyla připojena žádná komponenta, zavolání této funkce nic neudělá. Metoda vrací *true* v případě, že došlo k odebrání komponenty z *DOMu* a *false*, pokud nebyla žádná komponenta k odebrání z daného kontejneru nalezena.

Definice metody:

```
ReactDOM.unmountComponentAtNode(container)
```

Příklad užití:

```
ReactDOM.unmountComponentAtNode('root');
```

5.2 React komponenty

Komponenty jsou základními stavebními kameny React JS aplikací a umožňují rozdělit uživatelské rozhraní tzv. *User Interface* neboli *UI* na logicky nezávislé a

znovu použitelné části, ze kterých se následně skládají složitější komponenty, nebo jednotlivé stránky. Komponenty jsou v podstatě javascriptové funkce, které mohou přijímat vstupní parametry tzv. *props* a vracejí React JS elementy, které popisují, co by se mělo objevit na obrazovce. React považuje komponenty začínající malým písmenem jako značky *DOMu* např. * * a proto by měli všechny vlastní komponenty začínat velkým písmenem.

5.2.1 JSX

JSX je druh syntaxe, která se podobá značkovacímu jazyku XML/HTML a slouží pro definování React JS komponent. Jedinou výjimku oproti zápisu HTML je možnost zapsání komponenty pomocí tzv. *self-closing tags*, což se používá především v případě, že nechceme do komponenty zanořovat další tzv. *children* komponenty. Ačkoli JSX působí jako značkovací jazyk, ve skutečnosti se jedná o javascriptovou funkci *React.createElement*, která vytváří definovaný element. Výhodou JSX syntaxe je, že namísto rozdělování definice uživatelského rozhraní ve značkách tzv. *markup* a vlastní logiky do samostatných souborů můžeme definovat uživatelské rozhraní včetně jeho logiky v rámci jedné komponenty. Jak bylo již zmíněno výše, JSX je pouze tzv. *syntactic sugar* a pro jeho správné fungování v prohlížeči je třeba použít transpilátor, jako je například Babel, který převede stejně jako je tomu u ECMAScriptu tento druh syntaxe do podporované verze JavaScriptu. (25)

Zápis komponenty bez JSX

Na ukázce kódu níže je vidět zápis React JS komponenty pomocí syntaxe JSX a dále zápis stejné komponenty bez použití JSX. V tomto konkrétním případě tak vidíme, že definice navigačního menu pomocí JSX je výrazně snazší a ušetří mnoho řádků kódu. Zároveň je zápis komponenty bez JSX výsledkem překladu JSX komponenty pomocí Babelu.

Zápis komponenty pomocí JSX:

```
const nav = (
    <ul id="nav">
        <li><a href="#">Home</a></li>
    </ul>
);
```

Zápis stejné komponenty bez JSX:

```
var nav = React.createElement('ul', {
        id: 'nav',
    }, React.createElement('li', null, React.createElement('a', {
        href: '#',
    }, 'Home')),
);
```

Přístup k proměnným

Jelikož je JSX javascriptová funkce, je možná v rámci JSX také přistupovat k proměnným, a to pomocí složených závorek.

```
const anyVariable = 'hodnota';

const AnyComponent = () => {
    return <h1>"The values is" + { anyVariable }</h1>;
};
```

Conditional rendering

Podmíněné vykreslování neboli *conditional rendering* slouží pro vykreslování komponent na základě hodnot *true* a *false*. V případě, že chceme nějakou komponentu vykreslit pouze, pokud je splněn nějaký předpoklad můžeme použít výraz *if* nebo operátor *&&.* Operátor *&&* funguje tak, že v případě, že je kontrolovaná hodnota *true*, tak vykreslí následující obsah. V případě, že chceme rozhodovat o vykreslení specifické komponenty pro *true* a specifické pro *false*, je vhodné použít tzv. *ternální* operátor.

```
const FirstComponent = () => <div>First Component</div>;
const SecondComponent = () => <div>Second Component</div>;
```

Použití podmínky If

```
const Wrapper = () => {
    if (anyBooleanValue) {
        return <FirstComponent/>;
    } else {
        return <SecondComponent/>;
    }
};
```

Použití ternálního operátoru

```
const Wrapper = () => (
    <div>
        {anyBooleanValue ? <FirstComponent/> : <SecondComponent/>}
    </div>
);
```

Použití operátoru &&

```
const Wrapper = () => {anyBooleanValue && <FirstComponent/>};
```

5.2.2 Funkcionální komponenty

Funkcionální komponenta je obyčejná javascriptová funkce, která přijímá vlastnosti tzv. *props* jako svůj argument a vrací JSX. Funkcionální komponenty jsou nejjednodušší způsob, jak vytvořit React JS komponentu. Jejich výhoda oproti tzv. *class-based* komponentám spočívá ve snadnějším porozumění a testování komponenty a obecně v menším počtu řádků kódu, který je potřeba pro pokrytí funkcionalit komponenty.

Definice pomocí klíčového slova „function“

Vytváření komponenty pomocí klíčového slova *function* se v současné době považuje za zastaralý přístup a mělo by se od něj upustit a přejít k použití *arrow funkcí*.

```
function Example() {
    return (
        <div>This is deprecated approach</div>
    );
}
```

Definice pomocí arrow funkce

Použití *arrow funkce* je nejlepší způsob, jakým definovat funkcionální komponentu. Jak je v příkladu níže naznačeno, tento způsob se hodí především, když v rámci komponenty potřebujeme řešit dílčí operace jako jsou práce s *props* komponenty nebo dotahování tzv. *fetchování* dat.

```
const Example = ({ number }) => {
    const plusOne = number + 1;

    return (
        <div>New values is {plusOne}</div>
    );
};
```

Definice pomocí arrow funkce a „default return“

V případě, že nechceme v rámci komponenty řešit dílčí výpočty a chceme pouze vrátit JSX, je možné vynechat klíčové slovo return a složené závorky.

```
const Example = () => <div>Defaul return example!</div>

const Example = () => (
    <div>Defaul return example!</div>
);
```

Props

Jedná se o mechanismus, pomocí kterého je možné předávat informace React JS komponentám. Jako *props* můžeme zvolit jakýkoliv datový typ včetně funkcí, objektů, polí, ale také v případě potřeby i komponentu. V ukázce kódu níže si ukážeme, jakým způsobem je možné *props* funkcionální komponentě předat a jakým způsobem je zpřístupnit.

Předání props komponentě

```
<Example value={"This is the Value!"}/>
```

Zpřístupnění props v komponentě

```
const Example = (props) => <div>Prop value is: + {props.value}</div>;

const Example = ({value}) => <div>Prop value is: + {value}</div>;
```

5.2.3 Class komponenty

Class komponenty jsou stejně jako funkcionální komponenty javascriptovými funkcemi s tím rozdílem, že class komponenty pro svoji definici využívají syntaxe ES6 tříd, které dědí své vlastnosti z rozhraní *React.Component*. Do příchodu háků tzv. *hooks*, viz. následující kapitola, React class komponenty se lišili oproti funkcionálním především v tom, že umožňovali držet vlastní stav, mohli používat funkce životního cyklu komponenty a umožňovali definovat funkce uvnitř class komponenty, které mohli modifikovat stav komponenty.

V současné době, tedy po příchodu *hooků*, se doporučuje upouštět od používání class based komponent, jelikož je tento koncept pro některé vývojáře matoucí a obecně class based komponenty jsou často velice komplexní, což brání především při testování jednotlivých funkcionalit komponent. Class based komponenty jsou každopádně zpětně kompatibilní a není potřeba tak tyto komponenty přepisovat na funkcionální komponenty s použitím *hooků*.

Definování class komponenty

Na příkladu níže vidíme, jakým způsobem je možné definovat class komponentu, a to včetně definování vlastního stavu komponenty, metody životního cyklu *componentDidMount* a obslužné funkce, která modifikuje definovaný stav pomocí metody *this.setState*.

```
class Example extends React.Component {
    state = {
        count: 0,
        test: () => console.log('test'),
    };

    componentDidMount() {
        // Do something when component is mounting
        this.state.test();
    }

    handleCount = () => {
        this.setState({
            count: this.state.count + 1,
        }, () => console.log("callback!"));
    };

    render() {
        return (
            <div>
                <h1>Count is: {this.state.count}</h1>
                <button onClick={this.handleCount}>Update
count</button>
            </div>
        );
    }
}
```

State

Jedná se o immutable objekt, který uchovává vnitřní stav class komponenty. Do *state* neboli stavu komponenty můžeme uložit prakticky všechny typy dat, nicméně obecně bychom se měli vyhnout ukládání funkcí. Nejčastěji se do proměnných uvnitř stavu ukládají objekty, pole, čísla nebo textové řetězce. V případě, že budeme chtít k proměnným stavu class komponenty chtít přistoupit, budeme muset využít mechanismu *this.state.názevProměnné*. V případě, že budeme chtít hodnotu stavu komponenty změnit, budeme muset použít metodu *this.setState()*, která přijímá dva argumenty. První argument je samotný objekt, ve kterém definujeme, jaká hodnota se má změnit a její novou hodnotu a argument druhý, kterým je *callback*, který se vykoná, jakmile dojde k úspěšnému updatu stavu komponenty. Ukázku kódu si můžeme prohlédnout v příkladu definice class komponenty.

Metody životního cyklu

Metody životního cyklu neboli *lifecycle methods*, jsou funkce, které poskytuje *React.Component* a slouží pro definování událostí, které se mají stát v určitých fází stavu komponenty.

1. ***render()***

   Metoda render je nejvíce používaná metoda životního cyklu, a to především díky tomu, že je jako jediná pro class based komponenty vyžadována. Tato metoda zajišťuje renderování JSX do *DOMu* a je spouštěna během každého připojení komponenty do *DOMu* tzv. *mountu* a updatu komponenty.

2. ***componentDidMount()***

   Tato metoda životního cyklu je volána, jakmile se dokončí metoda *render* a komponenta je připravena a připojena v DOMu. Oproti metodě *render* se ještě liší tím, že umožňuje ve svém těle použít metodu pro změnu stavu *setState*, která se nejčastěji používá v kombinaci s *API* voláním pro uložení nově získaných dat do stavu komponenty.

3. ***componentDidUpdate(prevProps, prevState)***

   K jejímu spuštění dochází, jakmile dojde ke změně „update" komponenty vyjma počátečního vykreslení. Nejčastěji se používá jako reakce na změny *props* nebo *state* komponenty. Pro správné použití metody je třeba zároveň definovat podmínky, které budou porovnávat předchozí a novou hodnotu *props* nebo *state* class komponenty.

```
componentDidUpdate(prevProps, prevState) {
    // Do not forgot state comparision
    if (this.state.count !== prevState.count) {
        // Do something
    }
}
```

4. ***componentWillUnmnout()***

   Tato metoda, jak již její název napovídá, je spouštěna tehdy, když dochází k odebrání komponenty z DOMu a jejího následného zničení. Nejčastěji se používá tehdy, pokud potřebujeme vykonat nějaké čistící akce, jako může být například odebrání posluchačů tzv. *listenerů*. V těle této metody není možné použít metodu pro úpravu stavu komponenty *setState*.

5. ***shouldComponentUpdate(nextProps, nextState)***

   Jedná se o metodu, které se používá nejčastěji v případě optimalizace komponenty. Jejím úkolem je zamezit *re-renderování* komponenty ačkoli dojde ke změně určitého stavu nebo komponenta přijme určité *props*. Tyto výjimky, kdy má být zabráněno *re-renderování* jsou definovány v podobě výrazu, který vrací *true* nebo *false* v těle *return* této *lifecycle* metody. V případě, že dojde k vrácení hodnoty *false*, komponenta nebude *re-renderována*.

```
shouldComponentUpdate(nextProps, nextState) {
    return this.state.count === nextState.count;
}
```

5.2.4 Hooks

Do vydání verze React 16.8 byly funkcionální komponenty bez stavové tzv. *stateless*, což znamenalo, že jediná data, která do nich mohla vstupovat byla pomocí mechanismu *props* a tyto funkcionální komponenty si nemohly držet vlastní stav – na rozdíl od komponent založených na třídách.

Háky neboli *hooks* jsou funkce, které umožňují funkcionálním komponentám používat *state* a funkce životního cyklu komponenty. Ačkoli jsou *hooky* pouze javascriptové funkce je třeba se řídit ještě dalšími dvěma pravidly. *Hooky* je možné volat pouze z nejvyšší úrovně *scope*. Není vhodné volat *hooky* uvnitř cyklů, podmínek a zanořených funkcí. *Hooky* se volají pouze z funkcionálních komponent a není vhodné je volat v obyčejných javascriptových funkcích kromě vyčleněných vlastních *hooků*.

State hook

Funkce *useState* je *hook*, který umožní přidat *state* do funkcionální komponenty. *UseState* přijímá jediný argument výchozí stav tzv. *initial state* a vrací dvojici hodnot, a to aktuální stav a funkci pro update aktuálního stavu.

```
const Example = () => {
    const [count, setCount] = useState(0);

    return (
        <>
            <div>Count value is: { count }</div>
            <button onClick={() => setCount(count + 1)}>
                Update count!
            </button>
        </>
    );
};
```

Ref hook

Hook useRef umožňuje přístup k *DOM* elementům a vrací objekt s parametrem *current*, přes který je pak možná například získat hodnotu inputu ve formuláři.

```
const Example = () => {
    const inputEl = React.useRef(null);
    const onButtonClick = () => console.log(inputEl.current.value);

    return (
        <div>
            <input ref={inputEl} type="text" />
            <button onClick={onButtonClick}>Get Input Value</button>
        </div>
    );
};
```

Effect hook

Při použití *hooku useEffect* bude funkcionální komponenta, ve které je *hook* použit, reagovat na změny životního cyklu komponenty. Jedná se o zavedení komponenty do *DOMu*, přerenderování komponenty a její odebrání z *DOMu*. *Hook useEffect* se zavolá při zavedení komponenty do *DOMu* i při jejím překreslení. V případě, že budeme chtít, aby se *hook* nevolal při každém překreslení, ale pouze na základě změny nějakého parametru, je možné tyto parametry kontrolovat v poli a v případě, že se jejich hodnota nezmění, tak se *hook* nebude volat. Pokud chceme použít efekt

při odebrání komponenty z *DOMu*, je třeba specifikovat danou akci do těla funkce *return*.

```
const Example = () => {
    const [count, setCount] = useState(0);

    useEffect(() => {
        // Volá se při zavedení a přerenderování komponenty

        return () => {
            // Volá se při odebrání komponenty z DOMu
        };

    }, [count]); // Kontrolovaný parametr

    return (
        <>
            <div>Count value is: { count }</div>

            <button onClick={() => setCount(count + 1)}>
                Update count!
            </button>
        </>
    );
};
```

6 Angular

Angular je typescriptový *open-source* webový framework vytvořený společností Google sloužící pro budování *single-page* aplikací (26). Angular si během své historie prošel zásadní změnou, a to v podobě přepsání původní verze *AngularJS*, známé také jako *Angular 1* do verze současné, která je formálně pojmenována pouze jako *Angular* a mezi vývojáři je také známa jako *Angular 2+.*

V rámci celé této kapitoly bylo převážně čerpáno ze zdroje (26). V případě, že došlo k čerpání z jiných zdrojů, bude tak náležitě specifikováno.

6.1 Moduly

Aplikace Angularu jsou modulární a Angular má vlastní systém modulů tzv. *NgModules*. NgModuly jsou logicky izolované bloky kódu, které mohou importovat a exportovat vybrané funkcionality z a do ostatních NgModulů. Každá aplikace musí mít minimálně jeden kořenový NgModul, který je dle konvence pojmenován *AppModule* a je umístěn v souboru s názvem *app.module.ts*. Ačkoli malé aplikace mohou mít pouze jeden NgModul, většina Angular aplikací se skládá z většího počtu *NgModulů*.

6.1.1 NgModule

NgModul je definován pomocí dekorátoru *@NgModule()*, což je funkce, která jako svůj jediný argument přijímá objekt metadat *(Metadata = strukturovaná data, popisující další data a informace),* která popisují modul. Níže si představíme nejdůležitější metadata.

Declarations

Obsahuje všechny komponenty, direktiva a pipes, které patří k danému NgModulu.

Exports

Definuje vybranou množinu deklarací, která bude viditelná a použitelná v šablonách komponent dalších NgModulů.

Import

Další moduly, jejichž exportované třídy jsou potřebné pro šablony komponent deklarovaných v tomto modulu.

Providers

Množina služeb tzv. *service*, kterými tento NgModul přispívá do celkové množiny služeb celé aplikace. Tyto definované služby se tak stanou přístupné ve všech částech aplikace.

Bootstrap

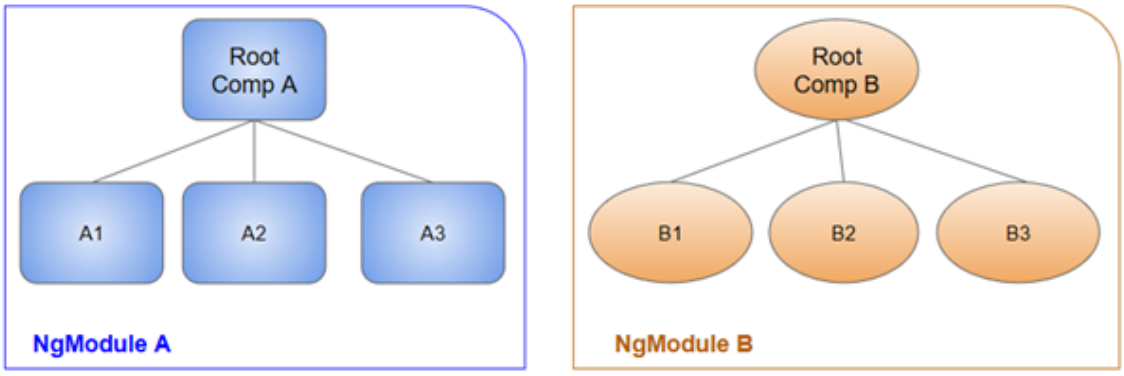
Jedná se o hlavní pohled tzv. *view* aplikace, který se nazývá *root* komponenta, která obsahuje všechny dílčí pohledy. Pouze kořenový NgModul by měl používat *bootstrap*.

src/app/app.module.ts

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

@NgModule({
  imports: [BrowserModule],
  providers: [Logger],
  declarations: [AppComponent],
  exports: [AppComponent],
  bootstrap: [AppComponent]
})

export class AppModule {}
```

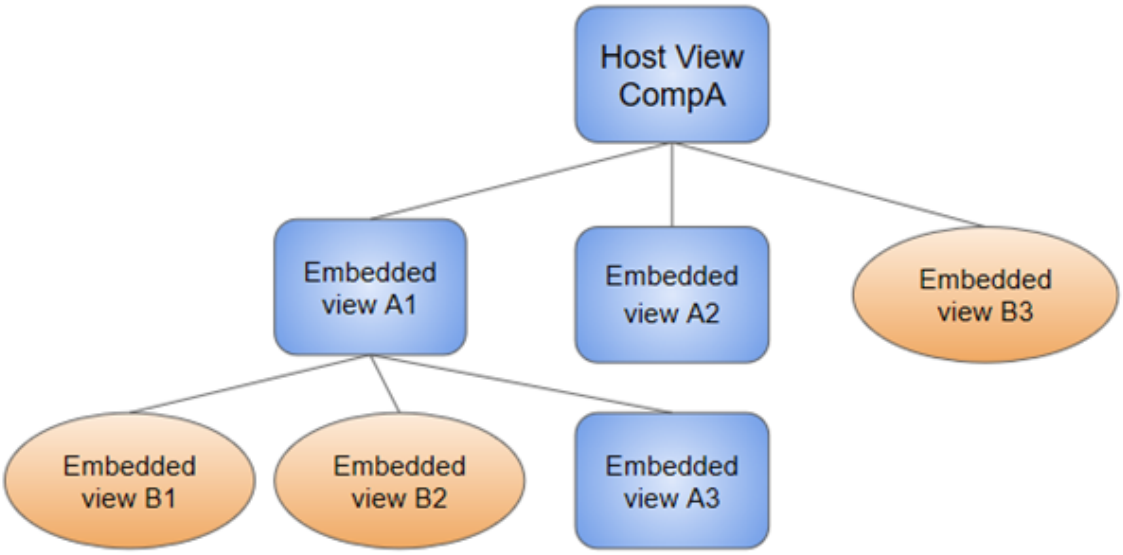
6.1.2 NgModules a komponenty

NgModuly poskytují kompilační kontext tzv. *compilation cont*ext pro své komponenty. Kořenový NgModul má vždy svoji kořenovou komponentu, která je vytvořena během *bootstrapu*. Jakýkoliv NgModul může obsahovat libovolný počet dalších komponent, které lze načíst přes *router*, nebo vytvořit pomocí šablony. Komponenty, které patří do NgModulu sdílí kompilační kontext.



Obr. 1 NgModuly a jejich kompilační kontext (26)

Komponenta a její šablona, tzv. *template* společně definují pohled, tzv. *view*. Komponenta může obsahovat hierarchii pohledů, což umožňuje definovat libovolně složité části obrazovky, které mohou být vytvořeny, upraveny a zničeny jako celek. Hierarchie pohledů může kombinovat pohledy definované v komponentách, které patří do různých NgModulů.

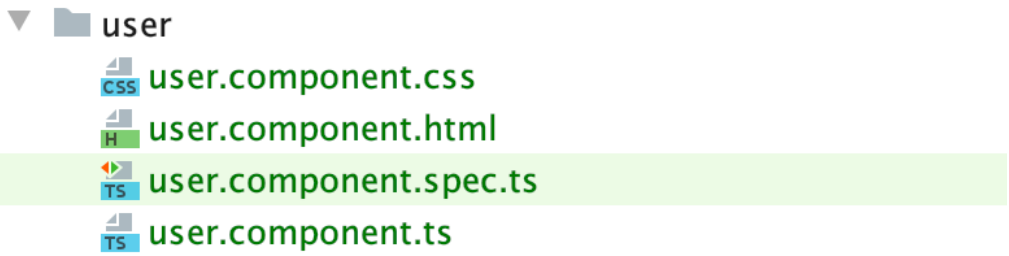


Obr. 2 Hierarchie pohledu komponent (26)

Když vytváříme komponentu, je přímo spojena s jediným pohledem zvaným *host view*. Tento pohled může být kořenový pohled v hierarchii pohledů, která může obsahovat dílčí zanořené pohledy, které mohou být *host view* dalších komponent. Tyto komponenty mohou být ve stejném NgModulu, nebo mohou být importovány z dalších NgModulů.

6.2 Komponenty

Komponenty slouží pro rozdělení obrazovky na malé logicky rozdělené bloky, které je možné skládat dohromady a vytvářet tak složitější obrazovky. Tento přístup umožňuje snazší orientaci v kódu, přispívá k modularitě celé aplikace a usnadňuje tak testovaná. Angular komponenty se na rozdíl od Reactu dělí na šablonu tzv. *template*, což je soubor s *HTML* a typescriptový soubor, který v sobě obsahuje logiku komponenty, která komunikuje právě s touto šablonou. V Angularu je komponenta dle konvence pojmenovaná složka, která v sobě obsahuje *template* – tedy *HTML* soubor, typescriptový soubor, *CSS* soubor a v případě potřeby také soubor určený pro jednotkové testy komponenty, jak je ukázáno na obrázku níže.



Obr. 3 Struktura adresáře Angular komponenty (vlastní zdroj)

6.2.1 Definice komponenty

V rámci této podkapitoly se zaměříme na typescriptový soubor komponenty. Aby se tento typescriptový soubor stal komponentou, je třeba definovat ES6 třídu, a to včetně dekorátoru *@Component*, který stejně jako dekorátor *@NgModul* bere jako svůj jediný argument objekt metadat. V ukázce kódu níže můžeme vidět, jak může vypadat definice komponenty *user*.

user.component.ts

```
@Component({
  selector: 'app-user',
  templateUrl: './user.component.html',
  styleUrls: ['./user.component.css']
})
export class UserComponent implements OnInit {
  constructor() { }
  ngOnInit(): void {}
}
```

Metadata komponenty říkají Angularu, kde získat hlavní stavební bloky, které potřebuje k vytvoření a prezentaci komponenty včetně jejího pohledu. Zejména pak slouží pro propojení s šablonou komponenty. Kromě toho, že obsahuje nebo odkazuje na template, *@Component* metadata také mohou popisovat, jak může být komponenta odkazována v template a jaké služby vyžaduje. Níže si představíme nejpoužívanější metadata.

Selector

Jedná se o *CSS* selektor, který říká Angularu, aby vytvořil a vložil instanci této komponenty kdekoliv, kde najde odpovídající značku v *HTML* template. Pokud například template *HTML* aplikace obsahuje *<app-user> </app-user>*, Angular mezi tyto značky vloží instanci komponenty *UserComponent*.

TemplateUrl

Relativní cesta k *HTML* šabloně komponenty, která definuje *host view* komponenty. V případě potřeby je možné tuto vlastnost nahradit pomocí vlastnosti *template*, která umožňuje namísto cesty ke komponentě použít přímo in-line *HTML* kód.

StyleUrls

Odkazuje na pole relativních cest k *CSS* stylům komponenty. V případě potřeby je možné nahradit pomocí vlastnosti *styles*, což podobně jako u *template*, umožní definovat *CSS* komponenty pomocí in-line kódu.

Providers

Definuje pole providerů pro služby, které komponenta potřebuje.

6.2.2 Template

Šablona neboli *template* vypadá jako obyčejné *HTML* kromě toho, že zároveň obsahuje *template Angular syntaxi*, která mění *HTML* na základě aplikační logiky, stavu aplikace a dat *DOMu*. Template může použít *data binding* ke koordinaci dat aplikace a *DOMu*, *pipes* k transformování dat před jejich zobrazením a *direktiva* pro aplikování logiky aplikace na to, co se má zobrazit. Na ukázce kódu níže vidíme

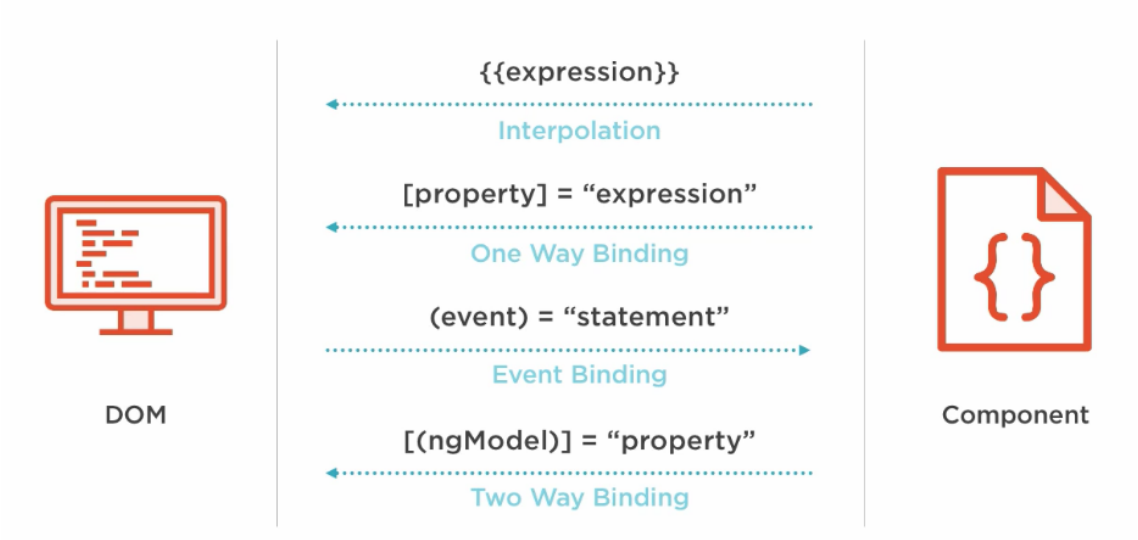
známé *HTML* elementy jako jsou *<h2>* nebo *<ul>*, ale také *Angular template syntaxi* jako je **ngFor*, *{{ user.name }}* nebo *<app-footer>*. (27)

```
<h2>User Template</h2>

<ul *ngFor="let user from users">
  <li (click)="showUser(user)">
    {{ user.name }}
  </li>
</ul>

<app-footer></app-footer>
```

6.2.3 Data binding

Vázání dat neboli *data binding* je proces, který umožňuje aplikacím zobrazovat hodnoty dat uživateli a reagovat na akce uživatele, jako jsou například kliknutí myši či stisknutí klávesy. V rámci vázání dat je třeba definovat vztah mezi *HTML* a zdrojem dat a Angular dále vše vyřeší sám a není tak třeba manuálně tlačit data do *HTML*, definovat *event listenery* či zajišťovat aktualizaci dat na obrazovce. (27)

Angular podporuje obousměrné vázání dat tzv. *two-way data binding*, což je mechanismus pro koordinaci částí *template* s částmi komponenty. Následující diagram ukazuje, jakým způsobem dochází ke komunikaci komponenty s *DOMem*. (27)



Obr. 4 Možnosti vázání dat v Angularu (27)

Interpolation

Interpolace umožňuje začlenit vypočítaný *string* do textu mezi *HTML* značky a do přiřazených atributů. Interpolace pro svoji definici používá dvě dvojice složených závorek, jak je ukázáno na příkladu níže.

```
{{ user.name }}
```

Property binding

Vázání vlastností neboli *property binding* zajišťuje tok dat pouze v jednom směru, a to z vlastnosti komponenty do cílové vlastnosti elementu. *Property binding* nelze použít ke čtení nebo vytažení hodnot či volání metod z cílových elementů. Mezi nejběžnější příklady užití pak patří nastavení hodnoty vlastnosti elementu nebo pokud potřebujeme deaktivovat element na základě *boolean* hodnoty. Ačkoli je *interpolace* i *property binding* velmi podobný mechanismus, tak *property binding* by se měl používat především ke změně stavu elementu a *interpolace* pak jako způsob zobrazení *string* hodnoty.

```
<img [src]="urlValue">

<button [disabled]="isDisabled">Disabled Button</button>
```

Event binding

Vázání událostí neboli *event binding* umožňuje poslouchat určité události, jako jsou například stisk klávesy nebo pohyb či kliknutí myši. Pro použití *event bindingu* je třeba v *template* specifikovat událost, na kterou chceme reagovat a v rámci komponenty pak implementovat příslušnou metodu. V případě, že chceme jako parametr funkce použít událost tzv. *event*, je třeba jej vložit spolu se znakem dolaru *$event*. V ukázce kódu níže můžeme vidět, jakým způsobem lze reagovat na kliknutí myši.

Component.html

```
<button (click)="onClick($event)">Click me</button>
```

Component.ts

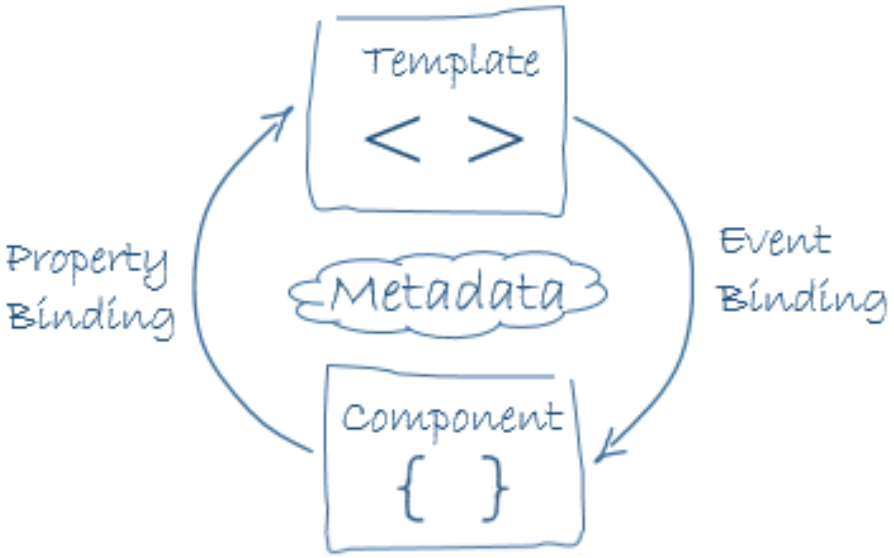
```
onClick(event: MouseEvent) {
  console.log(event);
}
```

Two-way data binding

Obousměrné vázání dat neboli *two-way data binding* kombinuje *property* a *event binding* v rámci jediné notace a jeho syntaxe tak zároveň kombinuje hranaté i složené závorky ve tvaru tzv. *banana in a box*. *Two-way data binding* se nejčastěji používá v rámci formulářů za použití direktivy *ngModel*, jak je ukázáno na kódu níže.

```
<input type="text" [(ngModel)]="value">
```

Při použití *two-way data binding* v rámci vstupu tzv. *inputu* formuláře jeho hodnota teče z komponenty do *template*, stejně jako *u property binding*, ale zároveň veškeré změny v *inputu* také tečou zpět z *template* do komponenty a updatují vázanou hodnotu, stejně jako u *event binding*. *Two-way data binding* hraje důležitou roli v komunikaci mezi *template* a jeho komponentou, ale zároveň i při komunikaci mezi *parent* a *child* komponenty.



Obr. 5 Two-way data binging (26)

6.2.4 Lifecycle hooks

Obdobně jako je tomu u Reactu, tak i Angular má vlastní metody životního cyklu, což jak již víme, jsou metody, které se spouští v různých fázích životního cyklu komponenty a Angularu také navíc u *direktiv*. Níže si představíme základní metody životního cyklu v pořadí, jakém jsou volány.

1. **ngOnChanges()**

   Volá se, jakmile Angular nastaví nebo znovu nastaví vstupní vlastnosti. Metoda přijímá *SimpleChanges* objekt aktuálních a předchozích hodnot vlastností. Tato metoda se volá před *ngOnInit,* anebo kdykoliv dojde ke změně jedné nebo více vstupních vlastností. (28)

2. **ngOnInit()**

   Inicializuje *direktivum* nebo komponentu poté co Angular poprvé zobrazí vlastnosti a nastaví vstupní vlastnosti komponenty nebo *direktiva*. Metoda je volána pouze jednou a volá se po dokončení metody *ngOnChanges*. (28)

3. **ngOnDestroy()**

   Jedná se o čistící metodu, která je volána pouze v případě, že Angular ničí komponentu nebo *direktivum*. V rámci této metody se typicky ruší odebírání pozorovatelů či obsluhy událostí, aby nedocházelo k zbytečnému úniku paměti. (28)

6.2.5 Pipes

Potrubí neboli *pipes* umožňuje deklarovat transformace v rámci *HTML templatu*. Třída, která je označena dekorátorem *@Pipe* definuje funkci, která transformuje vstupní hodnoty na výstupní hodnoty. Angular sám o sobě obsahuje již připravené *pipes* jako je například *date pipe*, nicméně není problém definovat vlastní. Pro specifikaci *pipe* transformace uvnitř *template* je třeba použít *pipe* operátor *(transformed value | pipe name)*.

Component.html

```
<p>Today is {{transformedDate |  date:'fullDate'}}</p>
```

Component.ts

```
transformedDate: any = new Date();
```

Pipes je možné řetězit za sebe a použít tak výstup jedné *pipe* transformace jako vstup do další *pipe* funkce. *Pipe* také mohou přijímat argumenty, které ovlivňují výslednou transformaci.

6.2.6 Directives

Direktivum je třída s dekorátorem *@Directive()* která má za úkol předávat instrukce během vykreslování *HTML templatu* do *DOMu*, které říkají, jakým způsobem má dojít k této transformaci. Komponenta je sama o sobě také *direktivem*, nicméně komponenta je pro celou Angular aplikaci tak klíčová, že dostala vlastní dekorátor *@Component()*, který rozšiřuje dekorátor *@Directive()*.

Stejně jako u komponent, metadata *direktivy* spojují dekorovanou třídu s jejím selektorem, který se používá pro vložení do *HTML*. V *template* se *direktivy* nejčastěji vyskytují uvnitř značek elementu jako jeho atributy. Angular definuje celou řadu již připravených *direktiv*, nicméně je možné vytvářet i vlastní. Kromě komponent existují další dva typy *direktiv* – strukturální a atributové.

Direktiva struktury

Direktiva struktury neboli *structural directives* mění layout přidáním nebo odebráním elementu *DOMu*. *Structural directives* používají aplikační logiku, která ovlivňuje výsledné vykreslení. Mezi direktivy struktury pak patří například ****ngFor***, který vykreslí tolik elementů, kolik jich je v daném poli, nebo direktivum ****ngIf***, které slouží pro podmíněné vykreslování na základě *boolean* hodnoty nebo výrazu, který vrací *boolean* hodnotu. Použití obou *direktiv* můžeme vidět v příkladu níže.

```
<div *ngFor="let item of array">Item number: {{item}}</div>
<div *ngIf="1 + 1 === 0">This element is not rendered</div>
<div *ngIf="1 + 1 === 2">This element is rendered</div>
```

Direktiva atributů

Direktivy atributů neboli *attribute directives*, jak jejich název napovídá, vypadají v *template* jako obyčejné atributy *HTML* elementů a ovlivňují vzhled nebo chování existujících prvků. Jako typický zástupce *atribute directives* je například *ngModel*, který jak jsme si řekli výše, zajišťuje *two-way data binding*.

```
<input type="text" [(ngModel)]="value">
```

6.3 Service

Služba neboli *service* je třída, která zahrnuje funkcionality, které na rozdíl od komponenty přímo nesouvisí se samotným vykreslováním uživatelského rozhraní, ale řeší nějakou obecnou logiku, která může být znovupoužita v rámci celé aplikace.

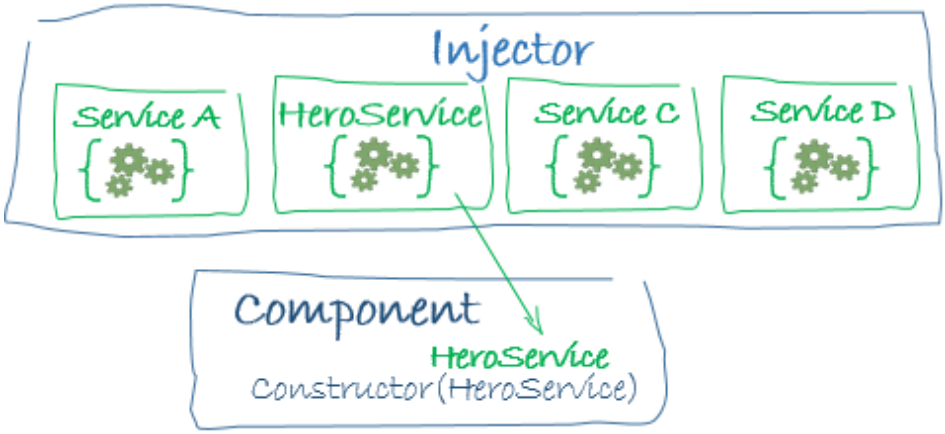
Angular rozlišuje komponenty a služby, aby zvýšil modularitu a znovu použitelnost kódu. Komponenta tak může delegovat některé úkoly, jako jsou například načítání dat ze serveru nebo ověření vstupních dat uživatele, na služby, které umí tyto dílčí úlohy vyřešit.

6.3.1 Dependency Injection

Vstřikování závislostí neboli *dependency injection* se v Angular frameworku používá všude, kde je třeba poskytnou komponentám např. služby, které komponenty potřebují. Pro definování třídy jako služby v Angularu je třeba použít dekorátor *@Injectable()*, který poskytuje metadata, která umožní Angularu vložit službu do komponenty jako závislost.

Pro jakoukoliv závislost, kterou potřebujeme v aplikaci, je třeba zaregistrovat poskytovatele tzv. *provider*, aby vstřikovač tzv. *injektor* mohl pomocí *provideru* vytvářet nové instance. Jakmile Angular vytváří novou instanci třídy, tak za pomocí konstruktoru dané třídy definuje, které služby nebo další závislosti potřebuje.

Pokud Angular zjistí, že komponenta závisí na nějaké službě, tak nejprve zjistí, jestli *injektor* má existující instanci dané služby. Pokud daná instance ještě neexistuje, *injektor* ji vytvoří za pomocí *provideru* a přidá ji do *injektoru* předtím, než službu vrátí Angularu.



Obr. 6 Dependency injection (26)

Injector

Vstřikovač neboli *injector* je hlavní mechanismus *dependency injection*. Angular vytvoří *injektor* pro celou aplikaci během bootstrap procesu a následně i další *injektory* v případě potřeby. *Injektor* vytváří závislosti a udržuje kontejner závislých instancí, které v případě potřeby znovu používá. *Provider* je pak objekt, který říká *injektoru*, jak získat nebo vytvořit závislost.

6.4 Routing

Angular Router umí reagovat na změny URL v prohlížeči a tyto změny následně využít jako pokyny k navigaci mezi *client-generated* pohledy. Router také může předávat parametry, které slouží pro rozhodování, jaký obsah má být ve výsledku zobrazen. Router je možné použít spolu s odkazy tzv. „*links*“, kdy po kliknutí dojde ke změně výsledného pohledu. Kromě odkazů je možné router provázat například s tlačítky nebo jako reakci na jakoukoliv akci. Router zároveň zaznamenává historii prohlížení, takže fungují i tlačítka vpřed a zpět.

Angular Router je sám o sobě službou, která zobrazuje konkrétní pohled na základě vybrané URL. Angular Router není sám o sobě součástí knihovny *@angular/core*, ale je obsažen v rámci své knihovny *@angular/router*.

6.4.1 Nastavení Routeru

Aplikace, které využívá Angular Router obsahuje pouze jedinou instanci služby *Router*. Jakmile dojde ke změně URL, router vyhledá odpovídající cestu tzv. *route*, která definuje, jaká komponenta má být vykreslena. Router neobsahuje žádné *routes*, dokud nedojde k jejich definování. Na příkladu níže si ukážeme, jakým způsobem je možné konfigurovat Angular Router v rámci aplikace.

```
const appRoutes: Routes = [
  {path: 'home', component: HomeComponent},
  {path: 'products', component: ProductsComponent},
  {path: 'products/:id', component: ProductDetailComponent},
  {path: '**', component: PageNotFoundComponent}
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      {enableTracing: true} // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule {}
```

Konstanta *appRoutes* nám definuje seznam všech cest, na které je možné v naší aplikaci navigovat. Tento seznam cest pak pomocí metody *forRoot()* vložíme do *RouterModule*. Každá *route* se pak skládá z cesty tzv. *path* a komponenty, která má být na této cestě vykreslena. V případě, že chceme používat proměnnou část URL, jako je tomu například u detailu produktu, použijeme symbol dvojtečky, který říká, že na tomto místě může být zobrazeno cokoliv. V případě, že parametr *path* zůstane prázdný, Router bude tuto cestu považovat za výchozí pro danou aplikaci. V případě, že budeme potřebovat stránku, kterou zobrazíme uživateli, pokud zadá *path*, která neexistuje, můžeme nastavit hodnotu *path* na ****, což Routeru říká, že tato cesta se zobrazí v případě, že žádná jiná neodpovídá.

6.5 HttpClient

Většina moderních front-endových aplikací komunikuje s beck-endovými službami pomocí Hypertext Transfer Protokolu tzv. *HTTP*. *HttpClient* slouží právě pro vytváření dotazů na beck-endové služby a je stejně jako Angular Router vyčleněn do vlastního modulu *@angular/common/http*. Tento klient poskytuje zjednodušené *HTTP API*, a kromě jeho snadného použití zároveň podporuje testování, typované dotazy a odpovědi nebo efektivní zpracování chyb tzv. *error handling*.

6.5.1 Použití HttpClient

Před použitím *HttpClient* je třeba importovat *HttpClientModule*, nejběžněji pak přímo v *AppModule*.

app.module.ts

```
import { NgModule }          from '@angular/core';
import { BrowserModule }     from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Po importování modulu je pak již možné injektovat *HttpClient* do tříd aplikace. Nejčastěji se *HttpClient* používá uvnitř služeb, které jak bylo zmíněno výše, odstiňují logiku, která se netýká přímo vykreslení uživatelského rozhraní do samostatných tříd.

test.service.ts

```
@Injectable()
export class TestService {
  requestedURL = 'http://10.20.30.40/api';
  response: any;
  constructor(private http: HttpClient) {}
  getData() {
    return this.http.get(this.requestedURL)
      .subscribe((data) => this.response = data);
  }
  postData(data: any) {
    return this.http.post(this.requestedURL, data);
  }
}
```

7 Praktická část

V rámci praktické části této práce si ukážeme, jakým způsobem je možné postavit React a Angular aplikaci. Během implementace jednotlivých části si ukážeme nejlepší možná řešení pro jednotlivé případy. Aplikace, kterou budeme v rámci této kapitoly vytvářet bude reflektovat základní funkcionality webových aplikací jako jsou navigace mezi stránkami, komunikace s beck-endem a principy práce s komponenty.

Pro vytvoření aplikace nebude použita žádná knihovna komponent či *CSS* framework jako je např. *Bootstrap*, ale dojde k vlastní implementaci komponent včetně stylování v *SCSS*. Tento postup byl zvolen především z edukativních důvodů, abychom si ukázali, vytvoření aplikace na tzv. „zelené louce“. Pro vytváření aplikací nicméně silně doporučuji použít již hotové komponenty, jelikož to výrazně ovlivní čas dodání výsledné aplikace.

7.1 Aplikace v React JS

Ještě před tím, než začneme psát aplikaci, je třeba, abychom v našem počítači měli nainstalovaný *node.js* ve verzi *4.X* nebo vyšší a správce balíčků *npm*.

7.1.1 Založení nového projektu

Nejjednodušším způsobem, jak založit novou React aplikaci je za pomocí nástroje ***create-react-app***, který pro nás vytvořili a neustále vylepšují vývojáři Facebooku. Tento projekt slouží pro rychlé a efektivní založení nového projektu v React JS. *Create-react-app* přináší již vyřešenou konfiguraci Babelu, Webpacku, základní kostru projektu a nainstalované základní závislosti. Po spuštění pouhých třech příkazů, které dotáhnou veškeré závislosti a nastaví celý projekt je možné okamžitě začít vyvíjet.

Příkazový řádek

```
cd anyFolder // Vybereme složku, kde chceme projekt vytvořit
npx create-react-app app-name // Vytvoří projekt se jménem app-name
cd app-name // Vybereme vytvořený projekt
```

Po vytvoření nového projektu pomocí *create-react-app* jsou kromě již připravených konfigurací k dispozici následující čtyři příkazy, které výrazně ulehčí ovládání celé aplikace. Dobré je si uvědomit, že ačkoliv nám *create-react-app* již přináší hotové řešení, je možné pomocí příkazu *eject* toto řešení opustit což nám v případě potřeby umožní vlastní konfigurování Webpacku nebo Babelu dle naší libosti. Tento přístup se může hodit především u složitějších projektů, kde z nějakých důvodu je třeba nějaké věci řešit vlastním specifickým způsobem. Na ukázce kódu níže vidíme příkazy, které nám po svém nainstalování *create-react-app* přináší.

```
cd app-name

npm start // Nastartuje aplikaci

npm build // Vytvoří produkční build

npm test // Spustí jednotkové testy aplikace

npm eject // Opustí create-react-app a umožní vlastní konfigurace
```

7.1.2 Dev-stack

Jak již bylo zmíněno v kapitole věnující se Reactu, tak víme, že React je knihovnou, a proto pro vytvoření celé aplikace je třeba často použít i další knihovny. V rámci implementace této aplikace byly kromě *react* a *react-dom* použity následující knihovny:

1. **Axios**
   Axios je knihovna, pomocí které je možné řešit komunikaci s beck-endem. Tato knihovna byla silně inspirována *http service* z frameworku Angular a vznikla, aby poskytla služby *http service* mimo tento framework (29).

2. **Prop-types**
   Knihovna, která slouží pro kontrolu datových typů proměnných. Tato knihovna je blíže popsána v kapitole 2.4.

3. **Ramda**

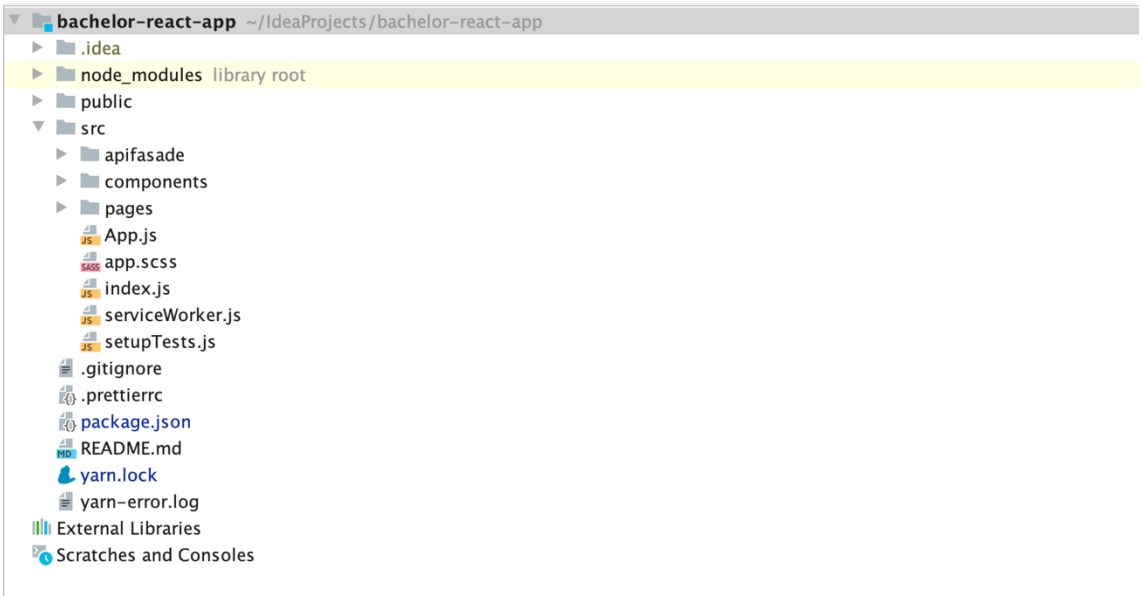
   Jedná se o knihovnu podporující paradigma funkcionálního programování. Tato knihovna obsahuje celou řadu užitečných funkcí, které například velice ulehčí práci s kolekcemi.

4. **React-router a react-router-dom**

   React router je mechanismus, který zajišťuje navigaci v rámci *single-page* aplikace. Díky *react-routeru* můžeme reagovat na změny URL v prohlížeči a simulovat tak chování přechodu mezi jednotlivými stránkami. Obdobně jako je tomu u knihovny React, tak doplňující knihovna *react-router-dom* byla vyčleněna a zajišťuje navigaci v rámci prohlížečích a knihovna *react-router* je pak jádrem celé knihovny (30). Kromě *react-router-dom* existuje i například *react-router-native* nebo *connected-react-router*, který umí pracovat s knihovnou Redux.

7.1.3 Struktura aplikace

Celá aplikace se skládá z logicky oddělených adresářů a konfiguračních souborů. Na obrázku níže můžeme vidět strukturu aplikace *bachelor-react-app*, která vznikla vytvořením projektu pomocí spuštění skriptu *create-react-app*, kromě struktury uvnitř adresáře *src*, který byla vytvořena na základě zvyklostí a konvencí strukturování React JS aplikace.



Obr. 7 Struktura aplikace bachelor-react-app (vlastní zdroj)

1. **.idea**

   Adresář obsahuje konfiguraci vývojového prostředí *IntelliJ IDEA*. Jelikož každý vývojář může používat rozdílná vývojová prostředí nebo rozdílné konfigurace, jedná se tak o adresář, který patří do souboru *.gitignore*

2. **Node_modules**

   Adresář, který obsahuje veškeré další závislosti, které jsou potřeba pro chod aplikace. Node_modules v projektu vznikají jakmile dojde k nainstalování závislostí definovaných v *package.json*. Stejně jako soubor *.idea*, tak tento adresář patří do souboru *.gitignore.*

3. **Public**

   Adresář obsahující jediný *index.html* soubor.

4. **.gitignore**

   V případě, že projekt využívá verzování kódu *git*, tak tento soubor definuje všechny adresáře a soubory, které nemají být komitovány.

5. **.prettierrc**

   Jedná se o soubor, který definuje styl kódu projektu, jako je například odsazení nebo uvozovky.

6. **Package.json**

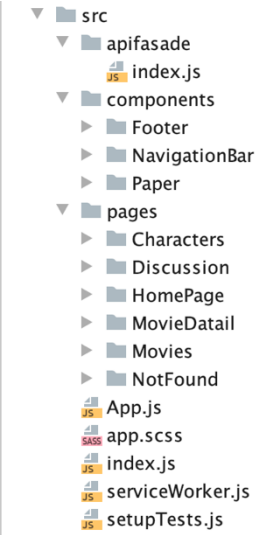
   Obsahuje seznam všech závislostí projektu a seznam spustitelných skriptů jako jsou skript pro spuštění celé aplikace či jednotkových testů. Jedná se o jeden z nejdůležitějších souborů, jelikož díky definici závislostí uvnitř je možné za pomocí příkazu *npm install* nebo *yarn install* závislosti instalovat a po jejich instalaci vzniká adresář *node_modules*.

7. **Yarn.lock**

   Soubor, který udržuje seznam všech nainstalovaných závislostí, a to včetně jejich verze a zdroje odkud byly dotaženy.

8. **Src**

   Jedná se o nejdůležitější adresář celé aplikace, který v sobě ukrývá veškerý javascriptový kód. Adresář *src* je dále členěn na logicky vymezené podadresáře. Strukturu adresářů projektu pak můžeme vidět na obrázku níže.



Obr. 8 Struktura adresáře src (vlastní zdroj)

Dále si podrobněji představíme jednotlivé adresáře a soubory, které jsou přímo v těle adresáře *src* a nespadají tak do žádného dílčího adresáře. Důležité je podotknout, že většina adresářů obsahuje javascriptový soubor *index.js*, který do sebe importuje danou komponentu. Tento přístup se využívá z toho důvodu, že jakmile dochází k importování je možné v cestě ušetřit specifikaci souboru v rámci adresáře, jelikož *index.js* se bere jako automatický export číslo jedna z adresáře.

a. **Apifasade**

   Obsahuje funkci, která obaluje funkci *get()* z knihovny *Axios* a přidává k ní autorizační hlavičku, která je vyžadována při volání na beck-endem vystavené *API*. Tato funkce byla vyčleněna z důvodu redundantního přidávání autorizace pro každé volání beck-endu.

b. **Components**

   Obsahuje všechny obecné komponenty aplikace, které jsou většinou vykreslovány na každé stránce. Jedná se opět o vyčlenění komponent

z důvodu zmenšení výsledného počtu řádků, omezení redundantního kódu a zvýšení logické celistvosti aplikace.

c. **Pages**

V rámci tohoto adresáře jsou obsaženy jednotlivé stránky aplikace. Ačkoli se dle definice Reactu jedná také o komponenty jsou vyčleněni do vlastního adresáře z důvodu, že se obvykle skládají z komponent umístěných v adresáři *components,* a navíc korespondují s nastavením *routeru* a jsou tedy vykreslovány na základě změn URL.

d. **App.js**

Jedná se o komponentu, která je vykreslována pomocí funkce *render()* z knihovny *react-dom* přímo do *HTML* souboru. Komponenta *App.js* v sobě sdružuje veškeré další komponenty celé aplikace, a to nejčastěji v podobě *routeru*.

e. **Index.js**

Jedná se o jediný soubor s názvem *index.js* v těle struktury adresáře *src*. Tento soubor je jako jediný spouštěn a musí tedy obsahovat metodu *render()* z knihovny *react-dom*, která zajistí vykreslování obsahu do *HTML* souboru aplikace. Dle konvence je v rámci tohoto souboru nejčastěji renderování komponenty *App.js*, která v sobě obsahuje veškeré další komponenty aplikace.

Zbylé dva javascriptové soubory *serviceWorker.js* a *setupTests.js* jsou generovány pomocí skriptu *create-react-app* a slouží ke konfiguraci *service workers*, kteří slouží pro progresivní webové aplikace a nastavení jednotkových testů.

7.1.4 App.js a nastavení react-routeru

React na rozdíl od Angularu neposkytuje v rámci své knihovny žádný mechanismus, který umí řešit navigaci v rámci single-page aplikace. Mezi nejvíce používanou knihovnu, která nám umožní routování pak patří *react-router*, popř. jeho derivát

připravený na použití s Reduxem – *connected-react-router*. V rámci praktické části projektu jsem se pak rozhodl pro použití knihovny *react-router*, kterou si na ukázce kódu níže představíme a následně rozebereme, jakým způsobem *react-router* funguje.

```
const App = () => (
    <Router>
        <NavigationBar/>
        <Switch>
            <Route exact path={'/'} component={HomePage}/>
            <Route exact path={'/movie'} component={Movies}/>
            <Route exact path={'/movie/:id'} component={MovieDetail}/>
            <Route exact path={'/characters'} component={Characters}/>
            <Route exact path={'/discussion'} component={Discussion}/>
            <Route component={NotFound}/>
        </Switch>
        <Footer/>
    </Router>
);
```

Komponenty react-router

Knihovna *react-router* nám kromě samotné funkcionality přináší následující komponenty, které usnadní práci s navigací v rámci naší aplikace. Níže si představíme základní komponenty, které byly použity v rámci práce. (30)

1. **Router**

   Komponenta *<Router>* je hlavní komponentou této knihovny. Veškeré další komponenty pak musí být uvnitř této komponenty, a to buď přímo, nebo uvnitř zanořených komponent. Vzhledem k tomuto faktu je důležité, aby Router byl definovaný na co možná nejvyšší úrovni aplikace a obsahoval ideálně všechny další komponenty aplikace.

2. **Route**

   Tato komponenta může být použita pouze uvnitř komponenty *<Router>* a její hlavním úkolem je definovat seznam cest a odpovídajících komponent, které mají na dané URL být zobrazeny. Pro definování URL používá *props path* a pro definování odpovídající komponenty pak *props component*, kam se vkládá reference na komponentu.

3. **Link**

   Jedná se o komponentu, která slouží pro navigaci mezi jednotlivými cestami v rámci aplikace. *Link* je nejčastěji stylován do podoby tlačítka, nebo navigačního baru. Použití *linku* si ukážeme v následující ukázce kódu.

```
const NavigationItem = ({ path, label }) => {

const isActive = useHistory().location.pathname === path;

return (
    <div className={'navigation-item'}>
        <Link
            className={`navigation-item__link ${isActive&&'active'}`}
            to={path}>{label}
        </Link>
    </div>
    );
};
```

Algoritmus hledání shody

React-router sekvenčně prohledává seznam našich cest tzv. *Rout* od shora dolů a hledá shodu mezi parametrem *path* v *routě* a aktuální URL v prohlížeči. Router vezme hledanou URL a začne ji znak po znaku porovnávat z leva doprava s hodnotou parametru *path* v *routě*. Jakmile dojde ke shodě a celý hledaný řetězec URL je obsažen v parametru *path*, router vrátí odpovídající komponentu nebo komponenty. Díky tomu se může stát, že pokud je hodnota v url pouze „/“ a všechny *routy* začínají právě znakem „/“ tak router vyhodnotí shodu u všech *rout* a vrátí tak i všechny komponenty. (30)

V případě, že chceme zajistit, aby došlo k vrácení pouze jediné komponenty, *react-router* nám nabízí možnost využití komponenty ***<Switch>***, která má za úkol vrátit pouze první shodu. Nicméně v případě, že budeme mít *routy* seřazené v našem routeru v pořadí „/“ a „*/second*“, a URL prohlížeče bude odpovídat druhé *routě*, tedy „*/second*“, tak k našemu zjištění dojde vrácení komponenty na prvním místě. (30)

Toto se děje z toho důvodu, jelikož jak bylo zmíněno výše, router porovnává URL znak po znaku, ovšem *Switch* vrátí první úplnou shodu s parametrem *path* a pokud tento parametr obsahuje pouze „/“ dojde k vrácení této komponenty i přes to, že hodnota URL v prohlížeči je „*/second*“. Způsoby, kterými je možné tento problém vyřešit jsou dva. První způsob je řadit *routy* v routeru od těch nejdelších po

nejkratší, tedy pokud prohodíme pořadí a na první místo přijde *routa „/second“* a na druhé „/“, tak router bude fungovat dle očekávání. Druhý způsob, jakým lze problém vyřešit je použití další *props* v rámci komponenty <Route> a to *props* ***exact***. Tato *props* říká routeru, že tato konkrétní *routa* může být vrácena pouze pokud URL obsahuje přesnou hodnotu, která je definována uvnitř *props path*. (30)

Hook useHistory()

Tento *hook* je možné použít pouze v těle funkcionálních komponent a slouží k zpřístupnění objektu *historie (30)*. Tento objekt nám pak například umožní manipulovat s historií a pomocí funkce *push()* měnit URL v prohlížeči. Díky tomu tak zajistí přecházení mezi námi definovanými *routami*, a to bez použití komponenty *Link*. Funkce *push* se nejčastěji používá například v *callbacku* funkcí nebo pokud potřebujeme změnit stránku po kliknutí na konkrétní element, jako je například přechod z nějakého listu obsahujícího množinu položek na detail vybrané položky.

```
import { useHistory } from 'react-router-dom';

const Example = () => {

    const history = useHistory();

    return (
        <div onClick={() => history.push('/')}/>
    );
};
```

7.2 Aplikace v Angularu

Ještě před tím, než začneme psát aplikaci, tak je stejně jako u Reactu třeba, abychom měli v našem počítači nainstalovaný *node.js* a správce balíčků *npm* nebo *yarn* v nejvyšší verzi.

7.2.1 Založení nového projektu

Nový projekt ve frameworku Angular je možné založit pomocí Angular CLI, které poskytuje sadu příkazů, které zajistí vygenerování nového projektu a na rozdíl od *create-react-app* zároveň zajišťuje i další ovládání projektu. Níže můžeme vidět přehled příkazů, které je třeba spustit pro vytvoření a spuštění nového projektu.

Příkazový řádek

```
npm install -g @angular/cli // Globání nainstalování Angularu
cd anyFolder // Vybereme složku, kde chceme projekt vytvořit
ng new project-name // Vygenerování nového projektu
cd project-name // Vybereme vytvořený projekt
ng serve // Spustí vytvořený projekt
```

Po vytvoření nového projektu pomocí příkazu *ng new* dojde obdobně jako u *create-react-app* k vygenerování projektu a jeho struktury. Projekt je již plně funkční a po jeho nastartování pomocí příkazu *ng serve* dojde k jeho spuštění. Základní nainstalovaný projekt obsahuje obdobně jako u *create-react-app* pouze testovací komponenty a pokud chceme pokračovat ve vývoji je třeba tyto testovací komponenty přepsat.

Kromě příkazů pro vygenerování a spuštění projektu nám Angular CLI poskytuje celou řadu dalších příkazů, které nám umožní ovládat vytvořený projekt.

Ng generate component

Jedná se o jeden z nejdůležitějších příkazů, které během vytváření Angular aplikace použijeme. Tento příkaz slouží pro vygenerování nové komponenty do zvoleného adresáře a na rozdíl od manuálního vytvoření komponenty zajistí vytvoření celého adresáře komponenty najednou a zároveň přidá závislost do *app.module.ts*. Na ukázce kódu níže si ukážeme základní případy užití příkazu *ng generate component*.

cd project-name

```
ng generate component component-name // Vytvoří komponentu do adresáře
src/app
ng g c component-name // Zkrácená alternativa příkazu výše
ng g c component-name/sub-component // Vytvoří komponentu s názvem
sub-component uvnitř komponenty component-name
```

Přehled dalších základních příkazů můžeme vidět na následující ukázce kódu.

```
cd project-name

ng help // Zobrazí přehled všech dostupných příkazů

ng build // Vytvoří produkční build aplikace

ng test // Spustí jednotkové testy

ng lint // Spustí lintovací nástroj pro formátování kódu
```

7.2.2 Dev-stack

Angular je na rozdíl od Reactu frameworkem, který nám poskytuje celou sadu funkcionalit, přesněji modulů, potřebných pro vytvoření front-endové aplikace. V rámci této aplikace došlo k použití jediné knihovny mimo Angular, a to knihovna Ramda, která podporuje funkcionální programování. Níže si blíže představíme jednotlivé moduly, které byly použity pro tvorbu naší aplikace. Knihovnu Ramda si v této části již nepředstavíme, jelikož se jedná o stejnou knihovnu, která byla použita při tvorbě aplikace v React JS.

1. **Browser Module**

   Jedná se pravděpodobně o nejdůležitější modul celé aplikace. Jak název tohoto modulu napovídá, modul slouží k tomu, aby aplikace mohla běžet v prohlížeči. V případě, že je aplikace vytvořena pomocí příkazu *ng new*, tak dochází automaticky k přidání a importování tohoto modulu v rámci souboru *app.module.ts*. Na ukázce kódu níže pak můžeme vidět způsob importování modulu.

   ```
   import {BrowserModule} from '@angular/platform-browser';
   ```

2. **AppRouting Module**

   Tento modul slouží pro konfiguraci routování aplikace. V kapitole 6.4 jsme si ukázali, jakým způsobem je možné konfigurovat cesty přímo v soubory *app.module.ts*. Na ukázce kódu níže si ukážeme, konfiguraci *routování* v souboru

app-routing.module.ts, který slouží pro vyčlenění *routovací* logiky. Tento soubor následně importujeme jako modul do *app.module.ts*.

app-routing.module.ts

```
import {NgModule} from '@angular/core';
import {Routes, RouterModule} from '@angular/router';
…

const routes: Routes = [
    {path: '', pathMatch: 'full', component: HomePageComponent},
     …
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule {}
```

3. **HttpClient Module**

   Jak bylo již popsáno v kapitole 6.5, tento modul zajišťuje HTTP komunikaci. V rámci této aplikace byl tento modul použit uvnitř služby *api.service.ts*, která jak si můžeme ukázat níže, slouží pro zavolání *get* požadavku na určenou url. V naší aplikaci byla metoda *get* použita spolu s přidanými parametry *headers*, které vkládají autorizační token do hlavičky každého dotazu.

api.service.ts

```
@Injectable()
export class ApiService {
    API_KEY = 'Bearer RgLfLp0ts07PcXAiTwlM';

    BASE_URL = 'https://the-one-api.herokuapp.com/v1';

    constructor(private http: HttpClient) {
    }

    getWithAuthorization(url): any {
        return this.http.get(`${this.BASE_URL}${url}`, {
            headers: new HttpHeaders(
                {
                    Authorization: this.API_KEY
                }
            )
        });
    }
}
```

4. **Forms Module**

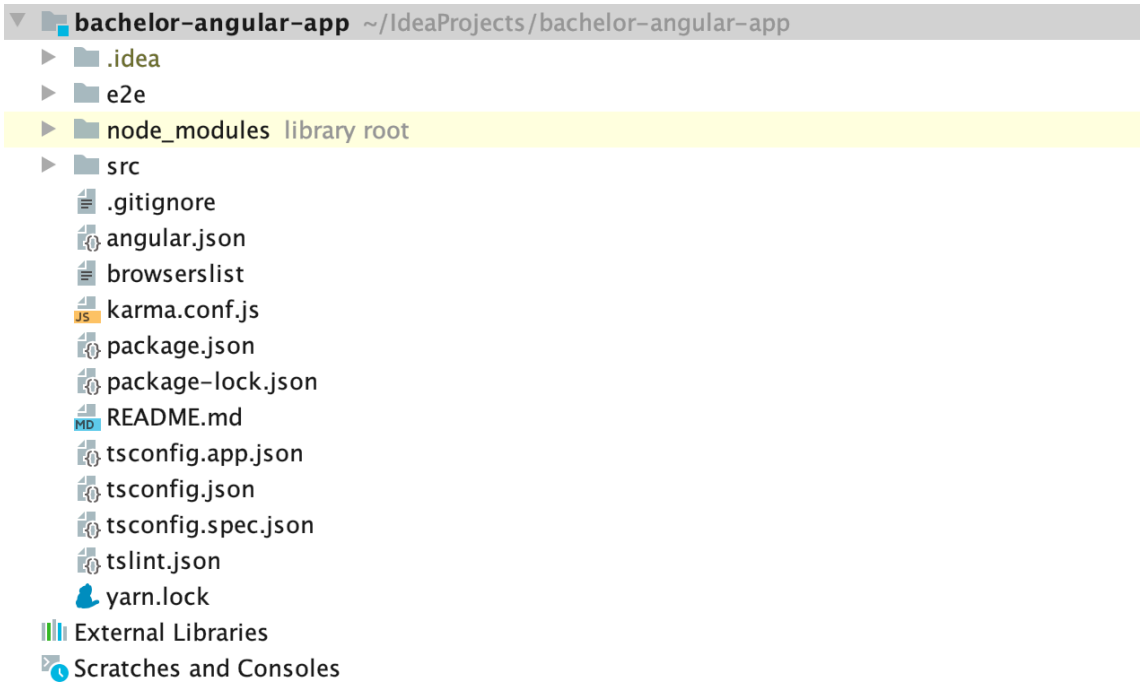
   Forms modul poskytuje sadu *direktiv* a *providerů* pro práci s formuláři, které komunikují s elementy nativního *DOMu* a snáze tak umožní zachytit uživatelské vstupy. V rámci naší aplikace došlo k použití tohoto modulu, konkrétně pak *direktivy ngModel* v případě potřeby zachycení vstupu od uživatele, jak můžeme vidět na následujícím příkladu:

characters.component.html

```
<input type="text" [(ngModel)]="searchedExpression" />
```

7.2.3 Struktura aplikace

Obdobně jako je tomu u Reactu, tak i aplikace ve frameworku Angular se skládá z logicky oddělených adresářů a konfiguračních souborů, kterých je oproti Reactu podstatně více. Na obrázku níže můžeme vidět základní strukturu aplikace, která vznikne po spuštění příkazu *ng new*, kterou si dále podrobněji projdeme a zaměříme se tak především na odlišnosti oproti struktuře z *create-react-app*.



Obr. 9 Struktura aplikace bachelor-angular-app (vlastní zdroj)

1. **E2e**

   Jedná se o složku, která obsahuje soubory pro *end-to-end* testování ve frameworku Angular za použití *protractor* a *jasmine*.

2. **Angular.json**

   Angular.json poskytuje výchozí konfigurační hodnoty pro vývojové nástroje poskytované Angular CLI na úrovni projektu. Hodnoty cest uvnitř tohoto souboru jsou relativní vůči kořenovému adresáři pracovního prostoru.

3. **Karma.conf.js**

   Konfigurační soubor, který je částečným konfiguračním soborem pro Karma. CLI vytváří úplnou konfiguraci běhového modulu v paměti na základě struktury aplikace uvedené v souboru *angular.js*on doplněném právě výše zmíněným *karma.conf.js.*

4. **Tsconfig files**

   Jedná se o konfigurační soubory TypeScriptu, který Angular používá.

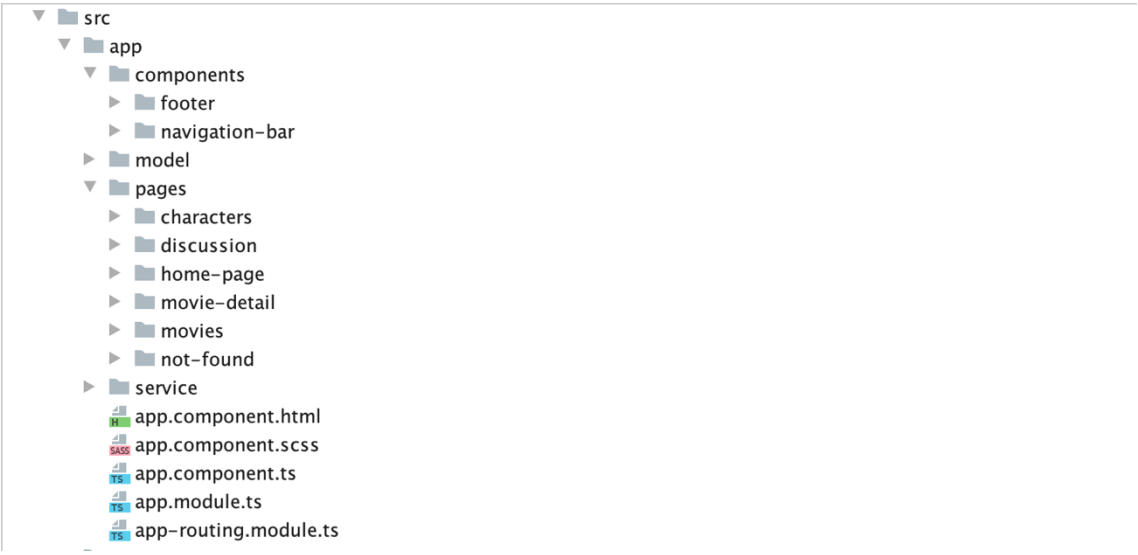
5. **Src**

   Obdobně jako u aplikace v Reactu, jedná se o klíčový adresář, který obsahuje veškerý kód naší aplikace. V rámci adresáře *src* pak mám obsažený zároveň *index.html* soubor, soubor obsahující *CSS*, či další konfigurační soubory. V rámci naší aplikace je nejdůležitějším adresářem adresář *src/app*, který při porovnání s Reactem přebírá úlohu adresáře *src*, tedy uvnitř adresáře *src/app* máme obsaženy komponenty, služby a další soubory, které již obsahují kód naší aplikace. Adresář *src* tak v Angularu má spíše obalovací charakter nad adresářem *src/app*.

7.2.4 Adresář src/app

Jak již bylo zmíněno výše, adresář src/app obsahuje kód samotné aplikace. Na obrázku níže pak můžeme vidět rozdělení adresáře naší aplikace, který se do jisté míry snaží kopírovat strukturu aplikace v React JS, nicméně jelikož Angular používá

jinou architekturu než React, jsou zde navíc adresáře jako je *model* či *service*. Kromě samotných adresářů pak tato struktura obsahuje i další soubory. Pokud se blíže podíváme na tyto soubory zjistíme, že všechny tyto soubory patří komponentě *app* a tím pádem zjistím, že samotný adresář je celý jedna komponenta.



Obr. 10 Struktura adresáře src/app (vlastní zdroj)

Prvky adresáře app/src

1. **Components**

   Jedná se o adresář, který v sobě obsahuje obecné komponenty, které jsou použity i v rámci ostatních komponent, které jsou umístěny v adresáři *pages*. Samotná komponenta se pak skládá z *HTML*, *CSS* a TypeScript soborů. V případě složitější komponenty v sobě pak může ukrývat další dílčí komponenty, které v sobě izolují logiku nadřazené komponenty. Tento koncept rozdělení na sub-komponenty se klasicky používá například pokud máme nějaký seznam a položky seznamu.

2. **Pages**

   Stejně jako u předchozího adresáře, adresář *pages* obsahuje komponenty. Důvodem, proč jsou rozděleny do samostatného adresáře je ten, že v adresáři pages jsou obsaženy pouze komponenty, které jsou vykreslovány v routeru – tedy, každá komponenta z tohoto adresáře patří k jedné definované cestě

v routeru. Komponenty v sobě mohou obsahovat další komponenty, které ovšem logicky souvisí s komponentou nadřazenou.

3. **Model**

   Adresář, který v sobě obsahuje modely datových struktur, které nejčastěji vrací beck-end po vykonání požadavku. Modely se používají ve spolupráci s TypeScriptem a pomáhají tak kontrolovat datové typy například polí, do kterých právě ukládáme získaná data. Na ukázce kódu níže si představíme model a jeho použití.

movie.model.ts

```
export class Movie {
    constructor(
        public _id: any,
        public name: any,
        public runtimeInMinutes: any,
        public budgetInMillions: any,
    ) {}
}
```

movies.component.ts

```
movies: Movie[]; // Očekává pole Movie
```

4. **Service**

   Adresář *service* v sobě ukrývá služby, které nejčastěji zajišťují komunikaci s beck-endem. Na ukázce níže můžeme vidět implementaci služby *ApiService*.

```
@Injectable()
export class ApiService {
    API_KEY = 'Bearer RgLfLp0ts07PcXAiTwlM';
    BASE_URL = 'https://the-one-api.herokuapp.com/v1';

    constructor(private http: HttpClient) {}

    getWithAuthorization(url): any {
        return this.http.get(`${this.BASE_URL}${url}`, {
            headers: new HttpHeaders({Authorization: this.API_KEY})
        });
    }
}
```

5. **App.component**

   Jedná se o tři soubory, které definují komponentu *app*. Tyto soubory spolu s následujícími dvěma jsou vygenerovány pomocí *ng new*. Za zmínku stojí soubor *app.component.html*, který v sobě obsahuje celé *view* aplikace, a to včetně routeru, který je zde použit jako komponenta *<router-outlet>*, která vrací příslušné komponenty na základě URL.

app.component.html

```
<app-navigation-bar></app-navigation-bar>
<router-outlet></router-outlet>
<app-footer></app-footer>
```

6. **App.module.ts**

   Klíčový soubor celé aplikace, který v sobě definuje závislosti na další moduly a deklaruje komponenty použité v rámci aplikace.

7. **App-routing.module.ts**

   Tento soubor, jak bylo již zmíněno v kapitole 7.2.2, definuje routovací mechanismus aplikace a je dále importován v rámci app.module.ts jako *AppRoutingModule*.

8 Shrnutí výsledků

Čtenář byl nejprve seznámen s obecnými principy JavaScriptu, na které následně navazovaly informace o datových strukturách a nástrojích úzce spojených s vývojem javascriptové aplikace v technologii React JS či Angular. Následně došlo k bližšímu seznámení s těmito technologiemi a implementaci aplikace v každé ze zmíněných technologií. Na základě takto nabitých poznatků bych rád nejprve zhodnotil knihovnu React JS a následně framework Angular.

Knihovna React JS je velice povedená a populární knihovna pro tvorbu javascriptového front-endu webové aplikace, která se bez pochyby hodí jak na malé jednoduché aplikace, ale zároveň je na ní možné postavit komplexní a velké systémy, jako jsou například sjednavače pojištění. Mezi její výhoda pak patří především křivka učení, která je díky tomu, že se jedná pouze o knihovnu podstatně menší než u Angularu, velice intuitivní syntaxe JSX a způsoby, jak sdílet data mezi komponentami a možnost si cokoliv a jakkoliv upravit dle vlastních požadavků. Z druhé výhody ovšem plyne i jedna zásadní nevýhoda a to ta, že neexistuje žádná ustálená metodologie, jakým způsobem psát aplikace v React JS, a proto především pro vývojáře může být obtížnější přechod mezi různými projekty, jelikož každý zaměstnavatel může zvolit jinou metodologii včetně dev-stacku. Za další nevýhodu pak může být považována téměř nutnost používat další podpůrné knihovny, které s Reactem spolupracují a které se mohou opět projekt od projektu lišit. V případě, že se rozhodneme na knihovně React JS postavit velkou a komplexní aplikaci u které chceme zajistit maximální životnost spolu se snadnou údržbou a možnou modularitou, je třeba velice pečlivě zvážit architekturu a knihovny, které budou spolu s Reactem použity.

Framework Angular a konkrétně pak jeho nová verze tzv. Angular 2 se poučil z chyb svého předchůdce a do jisté míry okopíroval to nejužitečnější s Reactu a vytvořil tak jednotný způsob, jak je možné tvořit front-end webových aplikací. Mezi hlavní výhody Angularu patří právě jednotnost, která je skrze všechny projekty stejná a fakt, že spolu s Angularem není téměř potřeba používat dalších knihoven, jelikož téměř vše je v rámci frameworku obsaženo. Díky tomuto přístupu je aklimatizace nových vývojářů, kteří již Angular znají jednodušší a jejich začlenění do

týmu kratší. Angular je bezpochyby možné použít jak na malé i velké aplikace, nicméně díky TypeScriptu je implementace stejné části kódu podstatně pomalejší než u Reactu, čímž může vzrůsti konečná cena projektu. Jako další nevýhodou pak je, že pokud se začneme učit Angular od začátku, křivka učení je výrazně delší než u Reactu. Tento fakt je způsoben především nutností dodržování Angular postupů a naučení se prací s velkým množství modulů. Na druhou stranu, pokud se zaměříme na dlouhodobou životnost projektu a máme dostatečné zdroje, tak kombinace jednotnosti řešení a silné typovosti TypeScriptu může být Angular velkým přínosem.

9 Závěr

Na základě výše zmíněných výsledků tak nelze jednoznačně říct, že jedna z technologiích je lepší než druhá, jelikož velice záleží na konkrétním případu užití. Obecně bych však React JS oproti Angularu doporučil spíše pro menší projekty, jelikož použití Angularu, by protáhlo a tím pádem i zdražilo výslednou aplikaci. Co se velkých a komplexních aplikací týče, tak v případě správně zvolené metodologie a dev-stacku je stejně dobré použít React i Angular, kde jako důkazem může být interní framework Generali České Pojišťovny GEF, který je právě postaven na technologii React JS a je úspěšně využíván na všech nově vzniklých a vznikajících projektech.

10 Seznam použité literatury

1. HOLZNER, Steven. JavaScript profesionálně: [kompletní referenční příručka]. Praha: Mobil Media, c2003. iDnes internet knihy. ISBN 80-86593-40-1.

2. About JavaScript. MDN web docs. [online]. Dostupné z URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript, 4. 6. 2019

3. SATERNOS, Casimir, St. Laurent, Simon ST. LAURENT, SIMON, Allyson MACDONALD a Rebecca DEMAREST. Client-server web apps with JavaScript and Java. Sebastopol, CA: O'Reilly Media, 2014. ISBN 1449369332.

4. Ecma International. Ecma-262, 5.1th edition, June 2011 [online]. Dostupné z URL: https://www.ecma-international.org/ecma-262/5.1/, 4. 6. 2019.

5. Ecma International. Ecma-262, 6th edition, June 2015 [online]. Dostupné z URL: https://www.ecma-international.org/ecma-262/6.0/, 4. 6. 2019.

6. Ecma International. Ecma-262, 7th edition, June 2016 [online]. Dostupné z URL: https://www.ecma-international.org/ecma-262/7.0/, 4. 6. 2019.

7. Ecma International. Ecma-262, 8th edition, June 2017 [online]. Dostupné z URL: https://www.ecma-international.org/ecma-262/8.0/, 4. 6. 2019.

8. Ecma International. Ecma-262, 9th edition, June 2018 [online]. Dostupné z URL: https://www.ecma-international.org/ecma-262/9.0/, 4. 6. 2019.

9. Nodejs [online]. Dostupné z URL: https://nodejs.dev/

10. V8 [online]. Dostupné z URL: https://v8.dev/

11. Electronjs [online]. Dostupné z URL: https://electronjs.org/

12. React-native [online]. Dostupné z URL: https://facebook.github.io/react-native/

13. TypeScript [online]. Dostupné z URL: https://www.typescriptlang.org/

14. Typechecking with proptypes [online]. Dostupné z URL: https://reactjs.org/docs/typechecking-with-proptypes.html

15. Proptypes [online]. Dostupné z URL: https://github.com/facebook/prop-types

16. Flow [online]. Dostupné z URL: https://flow.org/en/docs/lang/

17. Ecma International. Ecma-404, 2th edition, December 2017. [online]. Dostupné z URL:http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

18. Json [online]. Dostupné z URL: https://json.org/

19. XML [online]. Dostupné z URL: https://www.w3.org/XML/

20. Babel [online]. Dostupné z URL: https://babeljs.io/

21. Babel 7 [online]. Dostupné z URL: https://blog.bitsrc.io/so-whats-new-in-babel-7-ea97cb984ef0

22. Webpack [online]. Dostupné z URL: https://webpack.js.org/concepts/

23. React JS [online]. Dostupné z URL: https://reactjs.org/

24. ReactDOM [online]. Dostupné z URL: https://reactjs.org/blog/2015/10/07/react-v0.14.html

25. JSX [online]. Dostupné z URL: https://www.reactenlightenment.com/

26. Angular [online]. Dostupné z URL: https://angular.io/

27. Angular 2 [online]. Dostupné z URL: https://hahoangv.wordpress.com/category/angular-2/

28. Angular 2 Essentials [online]. Dostupné z URL: https://hahoangv.wordpress.com/category/angular-2-essentials/

29. Axios [online]. Dostupné z URL: https://github.com/axios/axios

30. React-router [online]. Dostupné z URL: https://reacttraining.com/react-router/

31. Angular CLI [online]. Dostupné z URL: https://cli.angular.io/

32. JQuery [online]. Dostupné z URL: https://jquery.com/

33. Angular JS [online]. Dostupné z URL: https://angularjs.org/

11 Seznam obrázků

12 Seznam příloh

1. CD 1 (bachelor-react-app)
2. CD 2 (bachelor-angular-app)

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Černohorský Adam	Kollárova 112, Chlumec nad Cidlinou - Chlumec nad Cidlinou III	I1600519

TÉMA ČESKY:

Moderní frontendové frameworky

TÉMA ANGLICKY:

Modern frontend frameworks

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl práce:
Cílem práce je porovnání frontendových technologií Reactu a Angularu

Osnova:
1. Úvod
2. JSON
3. React a Angular
4. Implementace
5. Závěr

SEZNAM DOPORUČENÉ LITERATURY:

SIMPSON, Kyle. You Don't Know JS: ES6 & Beyond. Kalifornie, USA: O'Reilly Media, 2015. 278 s. ISBN 978-1491904244

BANKS, Alex; PORCELLO, Eve. Learning React: Functional Web Development with React and Redux. Kalifornie, USA: O'Reilly Media, 2017. 350 s. ISBN 978-1491954621

Podpis studenta: ..... **Datum:** 15.10.2018

Podpis vedoucího práce: ..... **Datum:** 15.10.2018