EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/B/2019/36103158109192964

POROVNANIE REACTJS A ANGULAR FRAMEWORKOV

Bakalárska práca

2019 **Matúš Baxant**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

POROVNANIE REACTJS A ANGULAR FRAMEWORKOV

Bakalárska práca

Študijný program:	Hospodárska informatika
Študijný odbor:	Hospodárska informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Vedúci záverečnej práce:	Ing. Miroslav Kršjak, PhD.

Bratislava 2019 **Matúš Baxant**

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracoval samostatne a že som uviedol všetky použité zdroje.

Dátum: 30.3.2019 **.....**

Poďakovanie

Týmto by som rád vyjadril poďakovanie vedúcemu bakalárskej práce **Ing. Miroslavovi Kršjakovi, PhD.** za pomoc a poskytnutie odborných rád, konštruktívnu kritiku a pripomienky, ktoré boli pre mňa veľkým prínosom pri zhotovovaní tejto záverečnej práce.

ABSTRAKT

BAXANT, Matúš: *Porovnanie ReactJS a Angular frameworkov.* – Ekonomická univerzita v Bratislave. Fakulta Hospodárskej Informatiky; Katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing. Miroslav Kršjak, PhD. – Bratislava: FHI EU, 2019, 53s.

Cieľom záverečnej práce je porovnanie dvoch v súčasnosti najpoužívanejších frameworkov skriptovacieho jazyka JavaScript. Práca je rozdelená na tri kapitoly a obsahuje 3 grafy, 2 tabuľky a 25 obrázkov. Prvá kapitola je venovaná definíciám základnych pojmov, obsahuje krátky úvod do jazyka JavaScript a oboznámenie sa s fungovaním frameworkov. Druhá kapitola je zameraná na metodiku porovnania práce a tretia kapitola je samostatne venovaná frameworkom ReactJS a Angular. Opisuje možné prípady použitia a ich charakteristické vlastnosti. Zameraná je tiež na odlišnosti pri vývoji aplikácií a obsahuje aj ukážky kódu. Jej hlavným účelom je vyhodnotenie poznatkov a porovnaní z prechádzajúcich kapitol. Záver práce je venovaný výberu efektívnejšej voľby pre vývoj.

Kľúčové slová:
javascript, framework, angular, reactjs

ABSTRACT

BAXANT, Matúš: *Comparison between ReactJS and Angular frameworks* – Economic university in Bratislava. Faculty of Business Informatics; Department of Applied Informatics. – Consultant: Ing. Miroslav Kršjak, PhD. – Bratislava: FHI EU, 2019, 53p.

The thesis aims to compare two most commonly used frameworks for JavaScript scripting language. The thesis is divided into three chapters and contains 3 graphs, 2 tables and 25 images. First chapter is focused mainly on definitions of basic terms, it is brief introduction to language JavaScript, what frameworks are and how they work. Second chapter is involved in methodic of comparison and third chapter is more focused on both frameworks ReactJS and Angular. It describes possible uses and characteristic features. It also focuses on main differencies of application development and contains code snippets. Its main purpose is evaluation of findings and comparison based on previous chapters. The final part is involved in choice of the best solution for development.

Key words:
javascript, framework, angular, reactjs

OBSAH

Zoznam tabuliek a ilustrácií

Úvod

Moderný vývoj aplikácií a webu je v súčasnosti pevne prepojený s JavaScriptom na takmer neoddeliteľnej úrovni. V priebehu rokov tento proces vývoja prešiel rôznymi etapami a výraznými zmenami. Od statických stránok sme sa dostali až ku dynamickým a responzívnym dizajnom, ktoré používateľovi prinášajú úplne nový zážitok.

V poslednom čase čoraz viac vývojarov siaha vo svojich projektoch po frameworkoch založených na JavaScripte. Znásobujú efektivitu vývoja a umožňujú odbremeniť vývojára od mnohých základných úloh, na ktoré existujú dostupné už predpripravené riešenia. To zahŕňa aj samotnú optimalizáciu pre viacero druhov zariadení, akými sú napr. klasické desktopy, mobilné zariadenia alebo tablety.

Táto bakalárska práca si kladie za cieľ analyzovať jednotlivé aspekty vývoja aplikácií v dvoch najpoužívanejších frameworkoch, ktorými sú v súčasnosti React a Angular. Na základe získaných poznatkov a ich implementácii v praxi môžeme potom posúdiť zmysel použitia týchto technológií a porovnať ich jednotlivé vlastnosti. Výstupom tejto práce je finálne zhodnotenie a odporúčanie pre vývojára.

1 Súčasný stav riešenej problematiky

V priebehu posledných rokov JavaScript zaznamenal výrazný nárast popularity vo svete. Tento úspech bol založený prevažne na jeho spôsobe fungovania na strane klienta. V súčasnosti môžeme povedať, že sa stáva viacúčelovým nástrojom schopným fungovať na strane servera a jeho použitie je prítomné aj v desktopových aplikáciách. Webový ekosystém je jedinečným prostredím pre dynamické aplikácie, pre ktoré je ideálnou až neodmysliteľnou súčasťou práve JavaScript. Moderné enginy ako napr. Chrome V8, ktorý je súčasťou prehliadača Google Chrome, už nie sú iba jednoduchými interpretermi kódu. Ich výkon mnohonásobne vzrástol a zlepšili sa tak možnosti vývoja. Dokážu využívať databázy založené na JavaScripte a komunikovať s webovými službami, vymieňať si dáta prostredníctvom JSON – formátu, ktorý sa v posledných rokoch zaradil medzi jeden z najdôležitejších webových formátov.

Jedným z najvýraznejších prínosov v súčasnosti je možnosť univerzálneho využitia JavaScriptu na strane klienta aj servera, teda nepotrebujeme ďalší jazyk pre serverovú časť. Túto funkcionalitu je možné dosiahnuť pomocou interpretera NodeJS, ktorý zaisťuje rovnakú úlohu ako napr. jazyk PHP. Je založený na spomínanom Chrome V8 engine, takže rýchlosť spúšťania aplikácií a spracovania kódu je vysoká [1].

Keď sa prvý krát objavili JavaScript frameworky ako Angular alebo BackboneJS, boli považované za možno až príliš inovatívne riešenia pre väčšinu seriózných projektov. Postupom času ako sa frameworky vyvíjali, softvéroví architekti si začali uvedomovať ich vysoký potenciál. Aplikácie založené na JavaScript frameworkoch fungujúcich na strane klienta sú závislé priamo od hardvéru používateľa, pracujú na rovnakom spôsobe ako tzv. tuční klienti ktorí v sebe obsahujú prezentačnú aj aplikačnú vrstvu. Aplikácie písané za použitia frameworkov sú výrazne rýchlejšie než klasické webové aplikácie a vytvárajú oveľa lepší používateľský dojem najmä vďaka ich dynamickým vlastnostiam. V súčasnosti je použitie JavaScript frameworkov zamerané hlavne na vývoj mobilných HTML5 aplikácií, čo môžeme označiť ako prvotnú fázu. V budúcnosti majú totiž potenciál výrazne zmeniť spôsob vývoja moderných webových aplikácií [2]. Frameworkov založených na JavaScripte je aktuálne dostupných oveľa viac než v minulosti a ich nasadenie v reálných projektoch je vysoké. Frameworky ako Angular alebo React sú aktuálne využívané mnohými veľkými spoločnosťami, medzi ktoré patria napr. Google, Facebook alebo YouTube.

1.1 Stručný úvod do jazyka JavaScript

JavaScript je možné definovať ako udalosťami riadený skriptovací jazyk, ktorý obsahuje prvky objektovo orientovaného programovania. Jeho vývoj začal v roku 1995 firmou Netscape Communications a stál za ním Brendan Eich. Jeho prvý pracovný názov bol Mocha, neskôr LiveScript a až neskôr bola presadená myšlienka priblížiť sa Jave formou zápisu – tak vznikol konečný názov JavaScript. Syntax jazyka (zápis zdrojového textu) je zaraďovaná medzi jazyky ako C, C++ a Java, ale v princípe je jazyk od nich značne odlišný a s Javou akú poznáme nemá toho veľa spoločného. V roku 1996 bol štandardizovaný asociáciou ECMA (European Computer Manufacturers Association) – jeho štandardizovaná verzia má názov ECMAScript, z ktorej sú odvodené aj ďalšie implementácie ako napr. ActionScript [3].

Jedná sa o plnohodnotný programovací jazyk, ktorý funguje v prostredí webového prehliadača (na strane klienta). Dokáže deklarovať premenné, pracovať s poliami a objektami, dynamicky vytvárať elementy, umožňuje definovať vlastné triedy, funkcie a veľa iného. Webový prehliadač prvotne spracúva HTML kód webovej stránky, následne vytvára model DOM (Document Object Model) ktorý poskytuje „živý pohľad“ (live view) na stránku prostredníctvom kódu JavaScriptu, pomocou ktorého je možné obsah DOM dynamicky meniť a uplatniť zmeny v prehliadači.

Kód JavaScriptu je vykonávaný až po načítaní základnych častí stránky, ktoré tvorí HTML kód a kaskádové štýly CSS. Zároveň sa vykonáva na základe postupnosti volania funkcií a udalostí v dokumente. Samotný kód JavaScriptu alebo externý odkaz naň sa obvykle pripájajú v HTML časti dokumentu, konkrétne v hlavičke (tag HEAD). Ukážka obidvoch spôsobov pridania JavaScript kódu do dokumentu:

1. Definícia kódu v hlavičke dokumentu:

```
<script type="text/javascript"> ...zdrojový kód... </script>
```

2. Vloženie odkazu na externý súbor (s príponou.js) v hlavičke dokumentu:

```
<script type="text/javascript" src=".../cesta/subor.js">
</script>
```

Existujú aj iné spôsoby vkladania kódu JavaScriptu, napr. dynamicky za behu aplikácie. Niektoré frameworky sú orientované práve na takúto funkcionalitu [2].

1.2 Definícia a význam frameworkov

O vývoji webových stránok a aplikácií je možné uvažovať v reálnom príklade zo života – keď sa rozhodneme stavať dom, môžeme postupovať dvomi spôsobmi. V prvom prípade, by sme si všetky potrebné materiály vyrobili sami a začali stavať bez konkrétneho plánu stavby – takýto postup by bol pre nás výrazne časovo náročný, neefektívny a navyše by z praktického hľadiska nemal zmysel. Druhou a oveľa logickejšou variantou, je teda zaobstarať si už hotový materiál a následne z neho na základe určitého plánu postaviť dom, ktorý by spĺňal naše špecifické kritéria a požiadavky.

Tento praktický príklad sa dá veľmi jednoducho transformovať do oblasti programovania, pretože opisuje situáciu kedy sa rozhodujeme medzi tým písať aplikáciu tzv. „od podlahy“, alebo dáme prednosť možnosti postaviť ju na hotových základoch – využijeme **framework**.

Presnejšia definícia frameworku je tzv. aplikačný rámec“. Môžeme o ňom uvažovať aj ako o určitej forme aplikačnej šablóny – predprogramovanej open-source štruktúry, ktorá je zložená z viacerých komponentov a knižníc. Primárne použitie je väčšinou orientované na tvorbu SPA a UI aplikácií. Dôležitým bodom je stanoviť si rozdiel medzi knižnicou a frameworkom, ktoré sa líšia najmä v spôsobe ich fungovania. Knižnice pre JavaScript ako napríklad jQuery (jedna z najpoužívanejších) pracujú tak, že prikladáme ich prepojenie v už existujúcom kóde a následne ich funkcie voláme prakticky odkiaľkoľvek v rámci danej aplikácie. Ak by sme napríklad potrebovali vložiť jednoduchú animáciu pre určitý element, jednoducho zavoláme príslušnú funkciu z knižnice a zároveň jej odošleme informáciu o identifikátore konkrétneho elementu alebo o skupine viacerých elementov pre ktoré sa má vykonať.

Uvedený príklad, ktorý používa knižnicu by ale v prípade použitia frameworku fungoval odlišne. Framework neposkytuje riešenia na vzniknuté situácie iba na individuálnej báze, ale pracuje ako celok – je základnou kostrou webu/aplikácie. Organizuje a riadi jej jednotlivé súčasti ktorými sú komponenty, moduly a podobne. Jedná sa teda o vopred stanovenú formu, podľa ktorej je možné vytvárať vlastný obsah, ale zároveň je nutné nasledovať viaceré pravidlá a konvencie – tým sa odlišuje od knižníc, v ktorých nie sú pevne vymedzené pravidlá použitia. Princípy fungovania založené na frameworkoch môžu byť výhodou (efektivita kódu, znížená časová naročnosť, existujúce riešenia na bežné problémy), ale môže mať aj nevýhodu v podobe limitácie slobody vlastného kódu [5].

1.3 Základné pojmy a princípy fungovania frameworkov

1.3.1 SPA a UI

Prvým základným pojmom pri vývoji aplikácií je SPA (Single Page Application). Jedná sa o typ webových aplikácií, ktoré server využívajú iba ako zdroj a úložisko dát. Samotné dáta sú potom už vykreslované JavaScriptom, pomocou ktorého sa obsah dynamicky mení a tým poskytuje kvalitnejší zážitok pre používateľa. Ďalším úzko súvisiacim pojmom je UI. Reprezentuje používateľské prostredie (User-Interface) a teda vizuálnu časť s ktorou prichádza používateľ do kontaktu. Prostredníctvom tejto časti môže používateľ aplikáciu ovládať [6].

1.3.2 DOM

DOM (Document Object Model) je objektovou reprezentáciou HTML kódu. Definuje logickú štruktúru dokumentov a spôsob ako sa má k nim pristupovať. Môžeme si ju predstaviť ako stromovú štruktúru zloženú z jednotlivých uzlov. Manipulácia s DOM je možná pomocou JavaScriptu, navzdory tomu že nebol pôvodne optimalizovaný pre použitie s dynamickým používateľským rozhraním. Takže rozsiahlejší DOM obsahujúci tisíce uzlov je náročný na spravovanie a časová efektivita pri komplexnejších operáciách je nízka [7].

1.3.3 Virtuálny DOM

Virutálny DOM je abstrakciou HTML DOM, ktorá nie je závislá od spôsobu implementácie v prehliadači. Môžeme ho chápať ako koncept virtuálnej reprezentácie, ktorý je synchronizovaný so „skutočným“ DOM. Manipulácia s ním je výrazne menej náročná na výpočetný výkon, pretože sa nejedná o priamu manipuláciu s DOM. Umožňuje dynamicky uplatniť zmeny iba tých častí aplikácie, v ktorých sa majú prejaviť (tzn. nie je potrebné obnovovať celý dokument). [8].

1.3.4 TypeScript

TypeScript je open-source jazyk ktorého autorom je Anders Hejlsberg zo spoločnosti Microsoft (navrhol aj C#). Je nadstavbou ktorá rozširuje pôvodný jazyk o prvky ako statické typovanie, rozhrania a ďalšie atribúty typicky známe z objektovo orientovaného programovania, ktorými sú triedy, dedičnosť, moduly a pododobne. Každý JavaScript kód je aj priamo kódom TypeScriptu, nie je potrebné používať žiadne wrappery pre kompatibilitu. Otypovanie premennej vložením jedného z valídnych typov za názov premennej bude vyzerať nasledovne.

```
let a: number
```

Jednoduché otypovanie funkcie v TypeScripte potom vyzerá nasledovne [9].

```
function greet(greeting: string): void {
     console.log(greeting)
}
```

1.3.5 JSX

JSX (JavaScript eXtension) je rozšírením syntaxe JavaScriptu, ktoré umožňuje koexistenciu kódu JavaScriptu a HTML. Slúži na viazanie dát a zobrazovaných HTML elementov, ktoré tieto dáta využívajú.

```
const element = <h1>Hello, world!</h1>
```

Na tieto elementy je možné naviazať dáta JavaScriptu formou špeciálnej syntaxe v zložených zátvorkách [10].

```
const message = "Hello World!"
const element = <h1>{message}</h1>
```

Porovnanie klasickej syntaxe a JSX si môžeme ukázať v rámci frameworku React, kde si uvedieme dva ekvivalentné príklady zápisu syntaxe kódu za účelom vytvorenia popisu elementu ktorý React používa na vytvorenie zodpovedajúceho DOM [10].

```
const element = (
   <h1 className="greeting">
     Hello world
   </h1>
);
const element = React.createElement(
   'h1',
   {className: 'greeting'},
   'Hello world!'
);
```

1.3.6 NodeJS

Node je interpreter na strane servera rovnako ako napr. PHP, ktorý funguje asynchrónne a je riadený udalosťami. Je určený pre aplikácie písané v JavaScripte a umožňuje vývojárom vytvárať vysoko škálovateľné aplikácie. Dokáže spracovávať tisíce súčasných pripojení na jeden fyzický stroj. V súvislosti s Node je vhodné spomenúť aj robustný balíčkovací systém NPM, ktorý umožňuje sťahovať a používať v aplikáciach knižnice (balíčky) tretích strán. Samotný Node toho v základe veľa neponúka, preto tieto balíčky obsahujú riešenia na väčšinu bežných problémov a bez nich by bol vývojár odkázaný si množstvo funkcií sám odznovu naprogramovať. Jeho použitie je teda do istej miery závislé od týchto balíčkov [11].

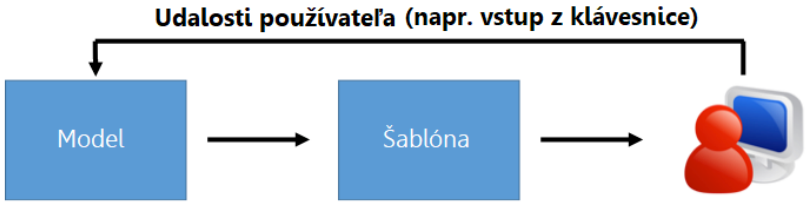
1.4 Väzba dát a architektonické vzory

1.4.1 Jednosmerná a dvojsmerná väzba dát

Väzbou dát rozumieme vlastnosť, ktorá dovoľuje synchronizáciu dát medzi stavom aplikácie (model) a obrazovkou (view). Rozlišujeme dva typy – jednosmernú a dvojsmernú. V prvom prípade každá zmena automaticky aktualizuje obrazovku, v druhom väzba spája vlastnosti a udalosti v rámci jedného objektu.

Jednosmerná väzba dát

Jednosmerná väzba dát (One-way data binding) je princíp jednosmerného toku dát, ktorý vytvára stále spojenie medzi modelom a UI. Zmeny v modeli sa potom prejavia v UI prostredníctvom procesov, ktoré sú riadené frameworkom alebo knižnicou. Tieto zmeny nie je možné vykonávať naopak, preto nie je možné uplatniť zmeny v UI na model.



Obr. 1: Jednosmerná väzba dát (One-Way data binding) [12]

Dvojsmerná väzba dát

Dvojsmerná väzba dát (Two-way data binding) je princíp dvojsmerného toku dát, ktorý rozširuje jednosmernú verziu o možnosť uplatniť zmeny v UI na model. Bez ohľadu na to, či knižnica alebo framework podporuje obojsmerný tok dát, jedinou možnosťou získavania dát z UI je prostredníctvom udalostí od používateľa.

Obr. 2: Dvojsmerná väzba dát (Two-way data binding) [12].

Princíp fungovania frameworkov je postavený na architektonických vzoroch. Ich podstatou je rozdeliť aplikáciu do troch rôznych vrstiev – model, obrazovka a radič. U väčšiny frameworkov a knižníc sa môžeme stretnúť s typom architektúry MV ktorá súhrne označuje vzory: **MVC, MVP** a **MVVM**. Prípadne s niektorou z ich špecifických implementácií ako napr. **Flux**, ktorý používa framework React [12].

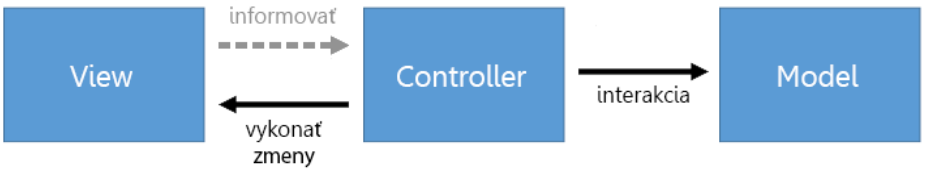
1.4.2 Architektonický vzor, základne typy a grafická interpretácia

Základné časti architektonického vzoru:

- **Model** – obsahuje dáta a stavy danej aplikácie, môžeme povedať že je jej „mozgom“. Nie je priamo viazaný na obrazovku a radič, takže môže byť využíteľný v kontexte rôznych potrieb.
- **View (obrazovka)** – je reprezentáciou modelu. Jej účelom je vykresľovanie (renderovanie) UI a komunikácia s radičom počas interakcie používateľa s aplikáciou. Táto časť nevie sama rozhodovať a riadi sa len tým, čo potrebuje poznať v rámci svojho fungovania. Je to najmä z dôvodu lepšej flexibility pri vykonávaní zmien v modeli.
- **Controller (radič)** – v MVC vzore je základným spojovacím článkom, ktorý „drží aplikáciu pokope“. Na základe komunikácie s obrazovkou (napr. kliknutie používateľa na tlačítko) rozhoduje o následnej interakcii s modelom. Riadi uplatňovanie zmien a stavov a teda je hlavným vykonávacím prvkom.
- **Presenter (predkladateľ)** – v MVP vzore zastupuje úlohu radiča, ktorý nie je viazaný na obrazovku ale iba na UI. Do istej miery tak preberá funkcionalitu zo vzoru MVC.
- **ViewModel (obrazovka-model)** – v MVVM vzore je určený na zviazanie modelu s dátami ktoré sa pripravujú a následne sú volané obrazovkou. Poskytuje zároveň funkcie na prenášanie udalosti do modelu ale s obrazovkou priamo viazaný nie je [13].

Popis a grafické znázornenie jednotlivých vzorov:

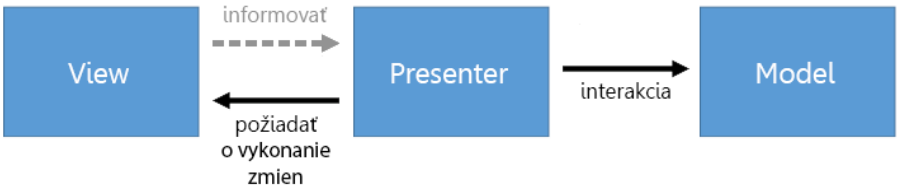
MVC (Model-View-Controller):



Obr. 3: Znázornenie architektonického vzoru MVC [13].

Vzor MVC je jedným z prvých architektonických vzorov, ktoré boli vyvinuté pre vývoj webových aplikácií. Umožňuje rozdeliť aplikáciu na jednotlivé časti a tým výrazne znižuje náročnosť na udržatelnosť, škálovateľnosť a testovanie. V tradičnom vývoji aplikácií tvorilo obrazovku aplikačné okno s používateľskými prvkami a celý kód, ktorý zaisťoval aplikačnú logiku, bol súčasťou jej triedy. Toto riešenie zvyšovalo veľkosť tejto triedy a vytváralo veľmi silné závislosti na UI a spôsobe väzby dát. MVC bolo navrhnuté na zredukovanie kódu na úrovni UI a malo zaistiť aby bol kód prehľadnejší a dlhodobo udržovateľný [15].

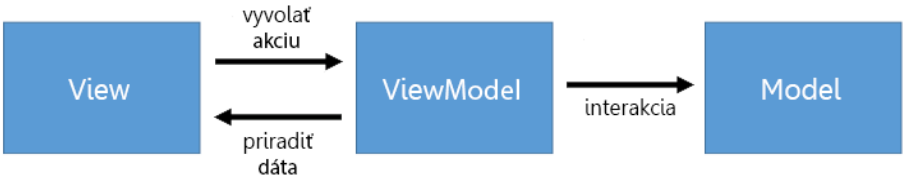
MVP (Model-View-Presenter):



Obr. 4: Znázornenie architektonického vzoru MVP [13].

Vzor MVP je veľmi podobne založený ako MVC. Jediná zmena spočíva v tom, že radič je nahradený predkladateľom (Presenter). Predkladateľ je zodpovedný za adresáciu všetkých UI udalostí v rámci obrazovky. Prostredníctvom obrazovky získava od užívateľa vstup a dáta posiela ďalej do modelu, ktorý výsledky vracia naspäť obrazovke. Tento druh architektúry sa sústredi na väzbu dát a je komplexnejším riešením najmä pre náročnejšie projekty, ale obecne je vhodným a efektívnym riešením. Bežne sa využíva v aplikáciach ako ASP.NET Web Forms a Windows Forms [14].

MVVM (Model-View-ViewModel):



Obr. 5: Znázornenie architektonického vzoru MVVM [13].

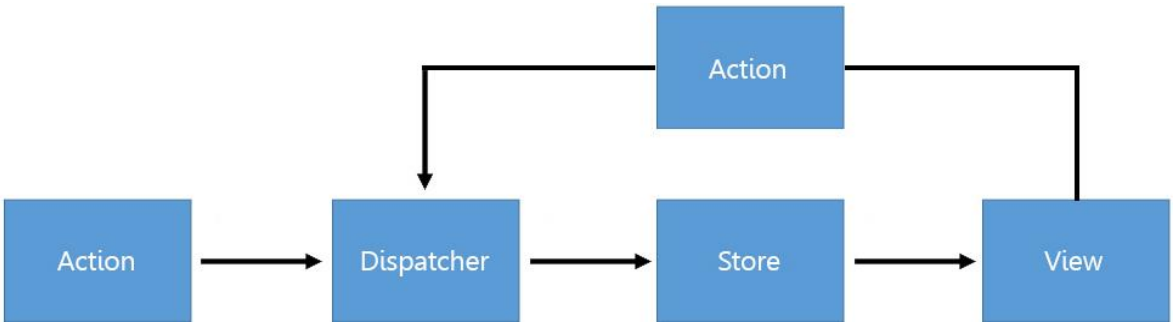
Vzor MVVM je priamo odvodený z MVC, podporuje dvojsmernú väzbu dát medzi obrazovkou a obrazovka-model. To umožňuje automatické uplatnenie zmien v zobrazení stavov z obrazovka-model do obrazovky [14].

Pri niektorých frameworkoch sa môžeme stretnúť so špecifickým vymedzením architektonického vzoru. Angular môže byť konfigurovaný ako MVC alebo MVVM, ale často sa používa aj označenie MVW (Model-View-Whatever), kde „Whatever“ symbolizuje „Whatever works for you“, teda doslova – čokoľvek vám vyhovuje. V prípade Reactu je takáto definícia ešte o niečo špecifickejšia, pretože jeho architektúra využíva jednosmernú väzbu dát a tým vlastne odporuje princípom uvedených vzorov. React využíva vlastný architektonický vzor s názvom **Flux**.

Flux funguje na princípe jednosmerného toku dát v UI (teda opak dvojsmerného) a vychádza z predpokladu, že výsledné UI je reprezentovateľné pomocou dát, ktoré do neho

prúdia z úložisk (Store). Interakcie užívateľov sú reprezentované pomocou akcií. Akcie spracováva Dispatcher, ktorý následne notifikuje o zmenách jednotlivé úložiská.

Flux (Action-Dispatcher-Store-View):



Obr. 6: Znázornenie architektonického vzoru Flux [15].

V rámci architektúry Flux, môžeme pri pohľade na niektoré konkrétne implementácie naraziť na pár problémov. Komponenty poznajú svoje úložiská a majú prístup k Dispatcheru. Taktiež dáta (a stavy aplikácie) sú distribuované cez niekoľko úložisiek a Dispatcher musí často koordinovať zmeny na niekoľkých miestach a to v praxi nemusí byť výhodou. Niektoré z problémov sa dajú obísť, napríklad môžeme úložisko extrahovať do komponentu typu Container. V ňom vložený komponent bude zásobovať dátami z úložiska pomocou props a tak sa dá dosiahnuť znovupoužiteľnosť a oddelenie od znalosti konkrétneho úložiska [15].

Záverom tejto kapitoly môžeme povedať, že výber vhodného typu architektúry pre aplikáciu je primárne závislý od povahy projektu a nemá jednoznačnú voľbu. V prípade jednoduchších aplikácií, ktoré obsahujú iba malý počet obrazoviek (jednu alebo dve) je vhodnejšie zvoliť variantu MVC. Pri náročnejších riešeniach sú v oblasti modularity rozdelenia aplikácie vzory MVP a MVVM obecne efektívnejšie než vzor MVC. Ich použitie znamená aj zvýšenie náročnosti a komplexnosti aplikácie. Najviac atraktívnym riešením pre vývojarov sa môže zdať MVVM z hľadiska redukcie kódu. Na konci je Flux, ktorý úplne nezapadá do vyššie spomenutých špecifikácií, ale pre aplikácie, ktoré sú riadené udalosťami je najvhodnejšou alternatívou a naopak pri zameraní na väzbu dát je vhodnejšia varianta MVC [13].

2 Cieľ práce, metodika práce a metódy skúmania

Cieľom záverečnej práce, je oboznámiť používateľa s princípami fungovania frameworkov založených na JavaScripte a porovnať jednotlivé odlišnosti vo vývoji aplikácií pri dvoch vybraných frameworkoch – Angular a React. Výsledkom záverečnej práce je zhodnotenie, na základe ktorého je rozhodnutie o voľbe efektívnejšieho nástroja pre vývoj.

Metodika záverečnej práce odráža celkovú koncepciu smerovania záverečnej práce podľa stanovenej postupnosti. Od prvotného výberu a oboznámenia sa s témou a základnými definíciami z jej oblasti, cez výber správnych metód porovnania vzhľadom na charakter práce a požadovaný cieľ, spracovanie a porovnanie informácií, až po konečné uplatnenie osnovy záverečnej práce a vyhodnotenie výsledkov výskumu.

Pri analýze metodológií sme čerpali z odborných publikácií a internetových stránok, z ktorých sme získali potrebné informácie o problematike, ktorou sa táto práca zaoberá. Informácie boli čerpané z rôznych, prevažne zahraničných zdrojov, z dôvodu väčšieho rozsahu dostupných materiálov a informácií. Na základe získaných metodologických analýz a poznatkov, následne pomocou komparatívnej metódy porovnávame vlastnosti v architektúre a spôsoby zápisu zdrojového kódu pri porovnávaných frameworkoch.

Komparatívna metóda je základnou metódou používanou vo vyskúme a je zároveň metódou ktorá sa používa v tejto práci. Pri jej použití je potrebné:

- stanovenie cielov,
- definovať objekt skúmania,
- stanoviť komparatívne kritéria,
- vykonať komparatívnu analýzu,
- vyhodnotiť výsledky výskumu.

3 Výsledky práce

3.1 Predstavenie frameworkov

Angular – prvá verzia (AngularJS) vydaná v roku 2010 spoločnosťou Google: jedno z najpoužívanejších open-source riešení, určené pre vývoj dynamických aplikácií typu SPA a tvorbu UI. Angular využíva abstraktné oddelenie zobrazovacej a aplikačnej logiky, funguje na klasickom princípe architektúry obrazoviek, modelov a radičov vzájomne previazaných s DOM. Tieto prvky sú previazané so šablónami, pričom pre každú z nich je definovaný vlastný rámec ktorý sa označuje ako `$scope`. Jeho prostredníctvom sa udržujú referencie na všetky aktuálne premenné a časti DOM, s ktorými sa práve pracuje a v rámci šablóny je ich možné volať alebo s nimi akokoľvek manipulovať. Zároveň sa sledujú zmeny definovaných premenných pre daný rámec, zisťuje sa či sú hodnoty rovnaké alebo sa zmenili. Na základe toho Angular vyhodnocuje potrebu uplatniť zmeny v DOM a aktualizovať šablónu do najnovšieho stavu. Podporuje funkcionalitu Dependency Injection (DI) – podporu vkladania závislostí v rámci frameworku. DI je návrhový vzor, ktorého podstatou je externé vkladanie závislostí objektom, teda bez toho aby si objekty závislosť obstarali sami.

Používaný spoločnosťami: YouTube, Paypal, Nike, Google, iStockPhoto, Weather a i.

React – vydaný v roku 2013 spoločnosťou Facebook: určený pre vývoj SPA a UI aplikácií. Vznikol v spoločnosti Facebook ako interný projekt, ktorý slúžil primárne na správu noviniek v rámci aplikácie Facebook Ads. Neskôr bol vydaný ako open-source framework pre vývojárov. Je vhodný pre využitie pri náročných projektoch s veľkým počtom užívateľských prístupov a jeho použitie je založené na komponentoch, ktoré je možné medzi sebou navzájom kombinovať. Hlavnými výhodami sú integrácia podpory SEO, rozšírenia JSX a vykresľovania na strane servera (SSR). Pri použití SSR je odozvou servera pre prehliadač iba HTML, ktoré sa má vykresliť, takže prehliadač nemusí čakať na načítanie a vykonanie celého kódu JavaScriptu. Architektonický vzor Reactu sa líši podľa povahy a požiadaviek projektu, môže fungovať na viacerých princípoch rovnako ako Angular, ale obecne sa jeho architektúra označuje ako Flux. Je riadený dátami a udalosťami, zároveň využíva jednosmernú väzbu dát a tým nenapĺňa funkcionalitu štandardných vzorov.

Používaný spoločnosťami: Facebook, Instagram, Netflix, Yahoo, New York Times, WhatsApp, Dropbox, Microsoft a i. [16]

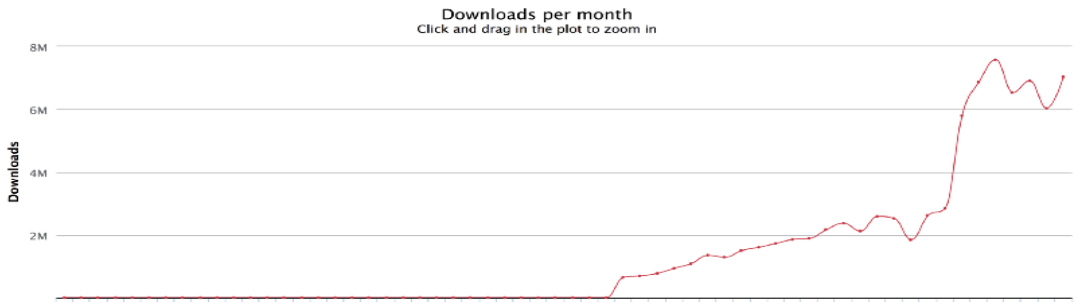
Súhrnná tabuľka pre porovnanie vlastností frameworkov Angular a React:

Angular	Porovnávací ukazovateľ	React
2010 (Angular JS), 2016 (Angular 2+)	Dátum vydania	2013
Google	Spoločnosť	Facebook
TypeScript	Programovací jazyk	JavaScript
NativeScript (Web, iOS, Android)	Prenositeľnosť	ReactNative (Web, iOS, Android)
Reálny	DOM	Virtuálny
Založená na komponentoch	Architektúra	Založená na komponentoch
MVC / MVVM	Architektonický vzor	Flux
HTML+TypeScript	Šablóna	JSX+JavaSscript (ES5 a i.)
Dvojsmerná	Väzba dát	Jednosmerná
Vysoká	Úroveň abstrakcie	Stredná
Na strane klienta/servera	Vykresľovanie UI	Na strane klienta/servera
Relatívne nízka	Veľkosť aplikácie	Relatívne nízka
Vysoký	Výkon	Vysoký
Angular 7.2.4	Posledná stabilná verzia	React 16.8.5
Vysoká	Krivka učenia (náročnosť)	Priemerná
59 tis.	Počet hviezd na GitHube	124 tis.
Open-source	Licencia	Open-source
Rozsiahla	Podpora komunity	Rozsiahla

Tab. 1: Súhrnné porovnanie vlastností frameworkov Angular a React [17, 18, 19, 20].

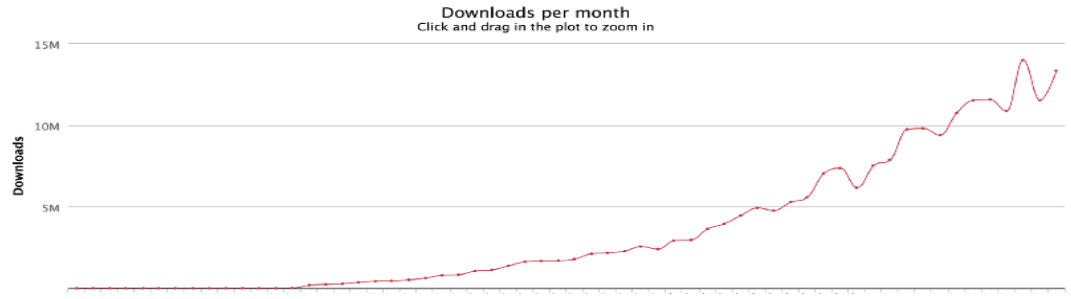
Podľa posledných prieskumov za rok 2018, v ktorých sú zahrnuté štatistiky počtu stiahnutí, je spomedzi JavaScript frameworkov aktuálne vedúcou trojicou Angular, React a Vue. V januári získal prvenstvo React, za ním sa umiestnil Angular a tretie Vue.

Štatistika počtu stiahnutí Angularu:



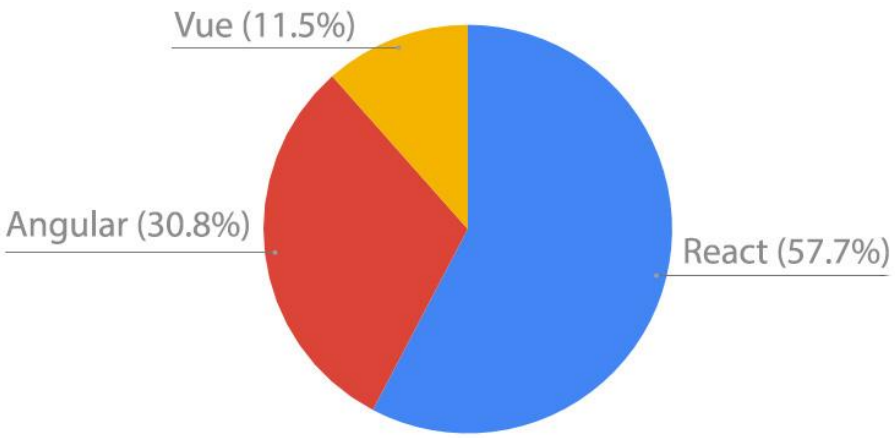
Obr. 7: Graf počtu stiahnutí Angularu za jednotlivé mesiace v rokoch 2014-2018 [21].

Štatistika počtu stiahnutí Reactu:



Obr. 8: Graf počtu stiahnutí Reactu za jednotlivé mesiace v rokoch 2014-2018 [21].

Prostredníctvom kruhového diagramu môžeme vidieť celkový podiel najpoužívanejších frameworkov na trhu. Na prvom mieste je React (57,7%), za ním s výraznejším odstupom Angular (30,8%), tretie miesto patrí Vue (11,5%) [21]:



Obr. 9: Percentuálny podiel najpoužívanejších frameworkov na trhu [21].

3.1.1 Výkonnostné porovnanie

Porovnanie výkonu pre frameworky Angular a React je realizované pomocou voľne dostupnej aplikácie s názvom „Movies of 2018“, ktorú je možné nájsť na stránke GitHub. Je písaná vo viacerých verziách osobitne pre každý z testovaných frameworkov. Aplikácia volá kolekciu funkcií API (Application programming interface), pomocou ktorej má za úlohu nájsť všetky filmy, ktoré majú v názve slovo „Love“. Zároveň, umožňuje aj tieto filmy zoraďovať na základe názvu a roku [22].

Lighthouse – open-source nástroj vyvinutý spoločnosťou Google, ktorý umožňuje komplexnú analýzu webových stránok a aplikácií v podobe rozličných testov (auditov) a meraní. Umožňuje testovanie rýchlosti vykreslenia DOM, kontrolu štandardov, zahŕňa aj SEO a veľa ďalších. Je dostupný iba pre prehliadač Google chrome: [23]:

- Chrome Developer Tools (nástroje pre vývojárov v prehliadači Google Chrome),
- príkazová konzola (cez Node CLI),
- rozšírenie v prehliadači Google Chrome.

Porovnávací test je zameraný na dva vybrané ukazatele:

- **Veľkosť JavaScript súborov** – rozdiely medzi celkovou veľkosťou skompilovaných JS súborov pri jednotlivých frameworkoch v rámci rovnakej aplikácie.
- **FCP (First Contentful Paint)** – súčasť Lighthouse, meria čas odozvy od začiatku vykresľovania prvého bitu obsahu definovaného v DOM. Načítanie aplikácie sa totiž nedá vymedziť iba ako jeden samostatný moment, ale ide o súhrn viacerých momentov prebiehajúcich v čase, ktoré určujú výsledný pocit používateľa – to či bude vnímať tento proces ako pomalý alebo rýchly.

1. Veľkosť skompilovaných JS súborov (testovaná aplikácia Movies of 2018):

Angular	React
65.5 KB	36.3 KB

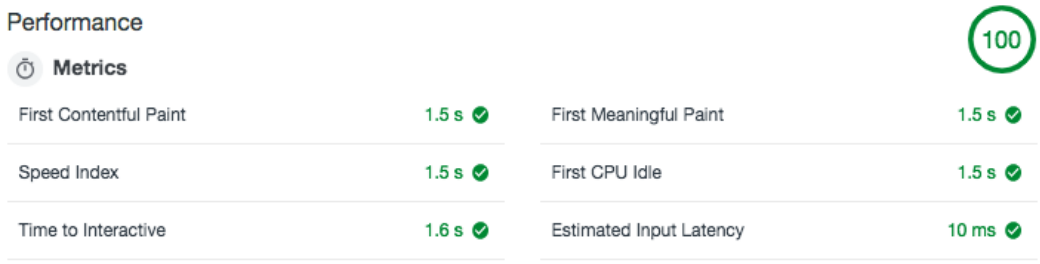
Tab. 2: Porovnanie veľkosti skompilovaných JS súborov [22]

V prípade Angularu je teda celková veľkosť oproti Reactu až dvojnásobná, takže v tomto porovnaní vychádza horšie.

2. Lighthouse FCP audit (testovaná aplikácia Movies of 2018):

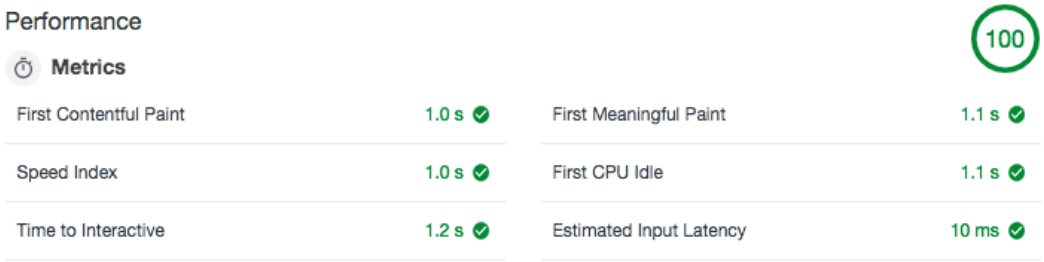
Nastavenia pre CPU a Network throttling settings: Fast 3G, 4x CPU slowdown.

FCP test pre Angular:



Obr. 10: Angular Lighthouse FCP audit [23].

FCP test pre React:



Obr. 11: React Lighthouse FCP audit [23].

Z dosiahnutých meraní môžeme vidieť, že React dosiahol v teste vykresľovania pomerne dobrý čas – 1.0 sekundy. Angular v tomto porovnaní opäť dosiahol horší výsledok – 1.5 sekundy [23].

3.1.2 Vývoj aplikácií vo frameworkoch Angular a React

V rámci predchádzajúcich kapitol sme si vysvetlili základné princípy fungovania JavaScriptu a frameworkov, ktoré sú na ňom založené. Zároveň sme si ukázali aj základné ukážky zdrojového kódu. Nasledujúca časť práce je zameraná na praktický opis vývoja aplikácií v obidvoch frameworkoch – Angular a React. Jej účelom bude praktické aplikovanie poznatkov a vedomostí, ktoré boli doteraz prezentované hlavne na teoretickej báze. Popíšeme si podrobný postup vývoja aplikácie v jednotlivých krokoch, tzn. od prvého spustenia až po vytvorenie základnej aplikácie v obidvoch frameworkoch. Tieto porovnania budú sprevádzané ukážkami kódu a obrázkami z výstupov. Naplnením účelu práce bude rozhodnutie o najvhodnejšej alternatíve pre vývojara.

3.2 Úvod do Angularu

V prvom rade je potrebné si utriediť pojmy Angular a AngularJS, ktoré spolu úzko súvisia. Prvá verejná verzia sa označovala ako AngularJS alebo Angular 2, jej nasledujúce verzie sú známe už iba ako Angular. Dnes už existuje niekoľko jeho vyšších verzií (aktuálne 7) a univerzálna skratka je iba Angular. Tieto dva frameworky sú navzájom **nekompatibilné,** ich fungovanie je založené na odlišných princípoch a **odlišnom jazyku**. Angular je postavený na komponentoch a princípoch objektovo orientovaného programovania (OOP). Základom každého komponentu je trieda (*class*), k tejto triede následne Angular naväzuje HTML šablónu a kaskádové štýly CSS pomocou dírektív. O samotnú logiku aplikácie sa potom starajú služby. Tie sú určené na komunikáciu a distribúciu dát medzi klientskou aplikáciou Angularu a serverovým API. Jednotlivé komponenty potom môžu využívať tieto služby pre svoju potrebu, pričom ich získavajú pomocou Dependency Injection.

3.2.1 Direktívy

Direktívy sú najzákladnejším (a teda aj najdôležitejším) elementom aplikácie Angularu. Komponenty sú tiež v podstate direktívy (vyššieho rádu), ktoré majú šablónu a slúžia ako základné stavebné prvky. V praxi existujú tri typy direktív:

- Komponenty
- Štrukturálne direktívy
- Atribútové direktívy

Komponenty – špecifický typ direktívy, ktorý umožňuje využívať encapsulovanú funkcionalitu webových komponentov a opakovane ich používať v celej aplikácií. Jedná sa o hlavný spôsob riadenia logiky na stránke, cez vlastné elementy a atribúty dokážeme pridať funkcionalitu do už existujúcich komponentov.

Štrukturálné direktívy – sú tzv. „DOM-unfriendly“ kvôli princípu na akom vytvárajú, ničia alebo znovu inicializujú DOM elementy (na základe rôznych podmienok). V porovnaní s atribútovou direktívou sú určité odlišnosti, zatiaľ čo napr. štrukturálna direktíva ***nglf** element zničí, atribútová direktíva **hidden** ho iba skryje pred používateľom.

```
<div *ngIf="showMe">
Ak showMe nadobudne FALSE *ngIf zničí tento element a
vymaže ho z DOM
</div>
```

Atribútové direktívy – aplikujú sa do elementov ako atribúty. Sú používané na manipuláciu s DOM všetkými spôsobmi okrem ich vytvárania a ničením. Môžeme ich teda nazvať tzv. „DOM-friendly“ direktívy [24].

```
<p [hidden]="hideMe">
Direktíva zobrazí alebo schová element
</p>
```

3.2.2 Prvé spustenie a CLI

Na začiatok práce s frameworkami ako Angular alebo React potrebujeme nainštalovať základný nástroj, ktorý už bol spomenutý v predchádzajúcich kapitolách – NodeJS, ktorého súčasťou je aj NPM. Stiahnuť ho je možné na stránke **http://nodejs.org** (na verzii prakticky nezáleží). Po nainštalovaní je ešte nutné si overiť či inštalácia prebehla v poriadku. Urobiť tak môžeme prostredníctvom terminálu v našom operačnom systéme (napr. cmd v prípade Windowsu). Zadáme nasledovný príkaz:

npm -v

a mala by sa nám zobraziť aktuálna nainštalovaná verzia (v mojom prípade 6.4.1).

Angular CLI

Angular CLI (Command Line Interface) je súčasť rozhrania pre príkazový riadok, ktoré nám umožňuje vytváranie a správu aplikácií v Angulari. Pomocou jednoduchých príkazov môžeme vytvárať komponenty, direktívy, služby a pod. Nainsťalujeme ho príkazom:

npm install -g @angular/cli

Základné konzolové príkazy (CLI)

ng new <nazov> – príkaz pre vytvorenie novej Angular aplikacie. Angular CLI za nás vytvorí potrebnú projektovú štruktúru, predpripraví konfiguračné súbory a tiež vytvorí základný modul a komponent.

ng generate <typ> <nazov> – príkaz pre vytvorenie komponentov, služieb a iných základných súčastí Angularu (to čo chceme vytvoriť špecifikujeme ako druh, napríklad pre vytvorenie komponentu: ng generate component my_component). Jedná sa o veľmi praktický príkaz, pretože tvorba nového komponentu zahŕňa vytvorenie viacerých súborov na viacerých miestach a viacnásobné deklarácie [25].

ng serve – príkaz pre spustenie serveru s aplikáciou, ktorá je po spustení prístupná na linku http://*localhost:4200* v našom počítači. Ostáva bežať aj po spustení aplikácie pričom stráži zmeny a ak nejaké zaznamená, tak aplikáciu znova načíta (bez nutnosti obnovenia stránky). Je možné použiť aj klasický príkaz **npm start**, ale ten vo výsledku aj tak zavolá ng serve. Hlavným rozdielom medzi nimi je, že príkaz **ng serve** je natívny príkaz Angular CLI rozhrania, zatiaľ čo NPM spúšťa to, čo je definované pre *scripts* v adresári aplikácie v súbore package.json.

Vytvorenie prvej aplikácie

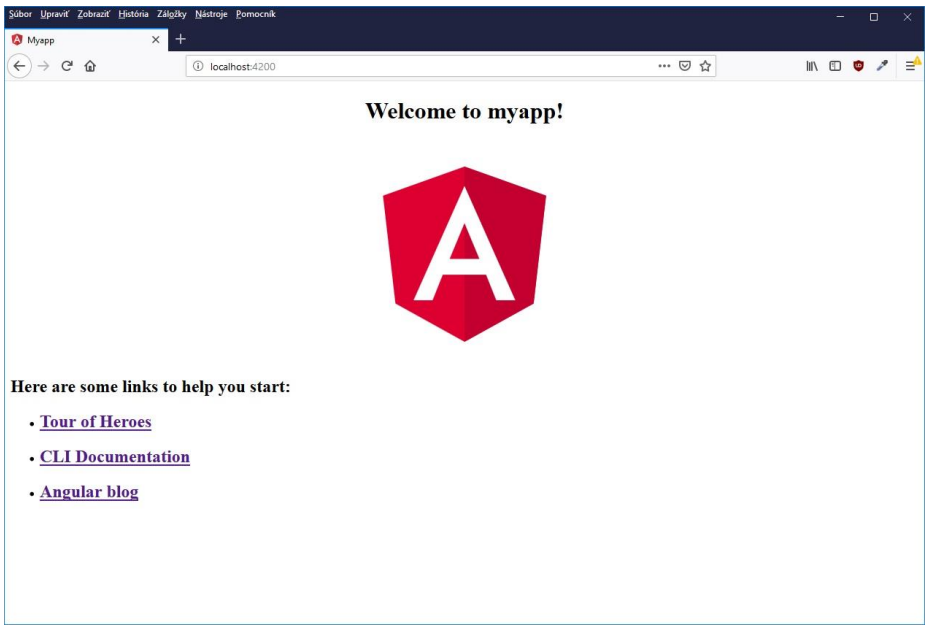
V termináli nastavíme cestu na adresár v ktorom chceme aplikáciu vytvoriť (v mojom prípade C:\Angular). Potom zadáme príkaz na vytvorenie aplikácie a zvolíme jej názov:

Príkaz na vytvorenie (v mojom prípade): C:\Angular\ > **ng new myapp**

Proces inštalácie sa automaticky opýta, či chceme v našom projekte použiť knižnicu Angular Routing a aký druh štýlov chceme použiť (v mojom prípade CSS). Vo zvolenom adresári sa nám automaticky vygenerovala celá projektová štruktúra našej aplikácie. Na spustenie aplikácie sa musíme znova nastaviť na zvolený adresár, v ktorom sme aplikáciu vytvorili (mimo neho nebude spustenie fungovať).

Príkaz na spustenie (v mojom prípade): C:\Angular\myapp\ > **ng serve**

Po zadaní adresy **http://localhost:4200** sa nám zobrazí základná automaticky vygenerovaná aplikácia Angularu s naším zvoleným názvom (v mojom prípade myapp):



Obr. 12: Zobrazenie základnej aplikácie Angularu

Aplikácia, ktorú Angular automaticky vytvoril obsahuje logo frameworku, názov našej aplikácie, odkazy na dokumentáciu a odkaz na modelovú aplikáciu „Tour of Heroes“.

3.2.3 Projektová štruktúra

Štruktúra ktorá nám vznikla v projektovom adresári aplikácie obsahuje viacero rôznych súborov, pre pochopenie ako celku si popíšeme tie najhlavnejšie časti:

- e2e/ (automatické testy aplikácie)
- node_modules/ (nainštalované moduly aplikácie, ktoré si ukladá npm)
- **src/** (aplikačný adresár, obsahuje súbory zdrojového kódu)
    - **app/** (hlavný modul aplikácie, defaultne obsahuje iba jeden komponent – AppComponent)
    - assets/ (adresár pre umiestnenie ďalších zdrojov, napr. obrázkov)
    - environments (nastavenie prostredia pre vývoj)
    - **index.html** (vstupný súbor webovej aplikácie, obsahuje HTML kostru)
    - **main.ts** (hlavný typescript súbor aplikácie)
    - **style.css** (globálne kaskádové štýly CSS pre celú aplikáciu)
- package.json (konfiguračný súbor, obsahuje meta dáta o aplikácii)
- tsconfig.json (jeho prítomnosť označuje, že sa nachádzame v koreňovom adresári projektu, obsahuje detaily a nastavenia kompilácie)

Súbor package.json (vytvorený NPM) obsahuje informácie (meta dáta) o aplikácii:

```
"name": "myapp",
"version": "0.0.0",
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  ...  },
```

Teda základné údaje ako názov aplikácie, verziu a v *scripts* môžeme vidieť definované CLI príkazy. Ďalej tento súbor obsahuje tzv. *dependenies* (závislosti), tie označujú rôzne súčasti ako podporné knižnice a ďalšie rozšírenia funkcionality.

Štruktúra samotného adresára **app** (v prípade mnou vytvorenej aplikácie **myapp**) [26]**:**

- **app.component.html** (vstupný súbor pre AppComponent, obsahuje HTML kostru)
- **app.component.css** (kaskádové štýly CSS pre AppComponent)
- app.component.specs.ts (unit testy pre koreňový komponent AppComponent)
- **app.component.ts** (definuje aplikačnú logiku pre koreňový komponent AppComponent, zároveň určuje hierarchiu komponentov a služieb použitých v aplikácii)
- app.module.ts (koreňový modul AppModule, defaultne načítava iba základný komponent AppComponent, pričom všetky ďalšie pridávané komponenty musia byť deklarované tu)
- app-routing.module.ts (štruktúra ciest aplikácie, spravuje jednotlivé linky a spája ich s komponentami s ktorými sú previazané)

3.2.4 Angular Router

Základná aplikácia, ktorú vytvára Angular automaticky neobsahuje žiadne zložitejšie prvky a preto v tejto časti jej funkcionalitu rozšírime. Angular Router je jednou zo základných funkcií, ktorá umožňuje navigáciu (prepínanie sa) medzi jednotlivými obrazovkami aplikácie. Ukážku aplikujeme na vytvorenú aplikáciu **myapp**. Vytvoríme jednoduché menu o dvoch položkách (stranka1 a stranka2), ktoré bude slúžiť na zobrazenie obsahu pre jednotlivé stránky.

1. Vytvorenie nových komponentov

Ako prvý krok vytvoríme v adresári aplikácie pomocou CLI dva nové komponenty:

ng generate component stranka1

ng generate component stranka2

Ako sme už spomenuli, výhodou Angularu je že po zavolaní týchto príkazov automaticky vygeneruje samostatné adresáre komponentov (stranka1 a stranka2) s predpripravenou štruktúrou súborov (analogicky rovnaká štruktúra aj pre druhý komponent):

- stranka1.component.html
- stranka1.component.css
- stranka1.component.spec.ts
- stranka1.component.ts

Súbor kaskádových štýlov CSS je defaultne prázdny a zdrojový kód HTML súboru oboch komponentov obsahuje iba tento jednoduchý kód:

```
<p>stranka1 works!</p>
```

Ten nás prakticky informuje o funkčnosti komponentu, pretože sa jedná o obsah, ktorý sa nám zobrazí po kliknutí na odkaz v ďalšej časti. V súbore *app.module.ts* sa nám automaticky obidva komponenty importovali. Aby sme ich mohli použiť, je nutné uviesť ich názvy aj v časti deklaracie. Ďalším krokom je import súčastí RouterModule a Routes ktoré zaistia potrebnú funkcionalitu. Následne vytvoríme konštantu *appRoutes*, v ktorej budú definované všetky cesty (odkazy), ktoré bude aplikácia využívať. Je dôležité uvádzať v časti *component* presné názvy importovaných komponentov, potom *path* predstavuje názov v adrese URL. Tretí riadok označený ako **, určuje domovský odkaz, ktorým bude defaultne komponent *stranka1*. Ak by sme ho neuviedli, automaticky by sa nezobrazil žiadny obsah (až po kliknutí na zvolený link).

```
import { BrowserModule } from '@angular/platform-
browser';
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { Stranka1Component } from
'./stranka1/stranka1.component';
import { Stranka2Component } from
'./stranka2/stranka2.component';

export const appRoutes: Routes = [
  { path: 'stranka1', component: Stranka1Component },
  { path: 'stranka2',component: Stranka2Component },
  { path: '**', redirectTo: 'stranka1', pathMatch:
'full' }];

@NgModule({
  declarations: [
    AppComponent,
    Stranka1Component,
    Stranka2Component
  ],
  imports: [
    BrowserModule, RouterModule.forRoot(appRoutes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Teraz potrebujeme upraviť hlavný súbor HTML aplikácie *app.component.html*:

```
<div style="text-align:center">
  <h1>Ukážka: Angular Router</h1>
  <ul>
   <li><a [routerLink]="['/stranka1']"
[routerLinkActive]="['active-link']">Stránka
č.1</a></li>
   <li><a [routerLink]="['/stranka2']"
[routerLinkActive]="['active-link']">Stránka
č.2</a></li>  </ul>
  <div class="obsah_sekcie">
    <router-outlet></router-outlet>
  </div>
</div>
```

V rámci tagu *a* vkladáme dva atribúty – routerLink a routerLinkActive. Prvý z nich nám určuje samotný názov adresy, ktorý uvidíme v URL, druhý nám dovoľuje priradiť triedu štýlov pre aktívnu položku v menu. Atribúty su označené hranatými zátvorkami z dôvodu, aby bolo možné vložiť sekvenciu (pole) viacerých výrazov, napr. viacero štýlov pre jeden element. Zobrazenie nami volaných komponentov má potom na starosti vkladaný element <*router-outlet*>. Nakoniec v globálnom súbore štýlov *app.component.css* pridáme jednoduché štýlovanie pre menu ktoré bude platiť v rámci celej aplikácie.

```
ul {
  list-style-type: none;
  margin: 0;
  padding: 0;
  overflow: hidden;
  background-color: #333;
}
li {
  float: left;
}
li a {
  display: block;
  color: white;
  text-align: center;
  padding: 14px 16px;
  text-decoration: none;
}
li a.active-link {
  border-bottom: 4px solid #fff;
}
.obsah_sekcie {
  text-align: left;
  padding: 20px;
  font-size: 25px;
  background: #F5F5F5;
}
```

Rozdielne štýlovanie si zároveň môžeme ukázať na samotných komponentoch. V adresári komponentu *stranka1* upravíme súbor *stranka1.components.css*. Analogicky potom upravíme aj súbor *stranka2.component.css* (líšia sa len farbou textu sekcie) [27]:

```
.komponent { color: red; }     /* color: blue; pre
stranka2 *}
```

Výstupom všetkých týchto úprav bude po spustení upravenej aplikácie táto obrazovka:

Obr. 13: Ukážka navigácie Angular Router

3.2.5 Väzba dát

Už na samom začiatku predstavenia frameworkov sme si uviedli základné charakteristiky samotnej väzby dát a zaradili Angular medzi frameworky využívajúce princíp dvojsmernej dátovej väzby. V prípade Angularu ale môžeme obecne hovoriť o štyroch základných druhoch dátovej väzby:

- Interpolácia reťazcov
- Väzba vlastností
- Väzba udalostí
- Dvojsmerná väzba dát

Interpolácia reťazcov – typ jednosmernej dátovej väzby, pri ktorej sa text umiestňuje medzi dvojicu dvojitých zložených zátvoriek, pričom tento text predstavuje názov konkrétneho komponentu. Angular potom takýto text nahradzuje reťazcom zodpovedajúcim danému komponentu. Tento príklad sa dá najlepšie demonštrovať na situácii, kedy napríklad

nemôžeme napevno uvádzať nadpis jednotlivých sekcií (komponentov) a rovnako tak ani samotný nadpis aplikácie (hlavný komponent). Do súboru *app.component.ts* pridáme:

```
export class AppComponent {
  title: string = "Ukážka: Data binding (interpolácia
reťazcov)";
}
```

Analogicky pridáme nadpisy aj do súborov *stranka1.component.ts* a *stranka2.component.ts*:

```
export class Stranka1Component implements OnInit {
  constructor() { }
  title: string = "Angular";  /* title = „React“; */
                              /* pre stranka2    */
  ngOnInit() { }
}
```

Do súborov *app.component.html*, *stranka1.component.html* a *stranka2.component.html* pridáme výraz umiestnený medzi dvojicou dvojitých zložených zátvoriek:

```
<h1> {{ title }} </h1>
```

Výstup z aplikácie bude vyzerať takto:

Obr. 14: Ukážka interpolácie reťazcov v Angulari

V zdrojovom kóde komponentov môžeme v rámci triedy vidieť defaultne zadefinovanú metódu **konštruktoru**, ktorá je volaná vo chvíli vytvorenia inštancie triedy. Ďalšou je

metóda **ngOnInit**, ktorá je volaná po inicializácii konštruktora. Táto metóda dáva Angularu vedieť, že inicializácia samotného komponentu je vykonaná a jednotlivé vytvorené väzby dát sú k dispozícii. Rovnakým spôsobom môžeme priradiť napr. aj cesty k obrázkom:

```
export class Stranka1Component implements OnInit {
  constructor() { }
  title: string = "Angular";
  imagePath: string = "assets/Angular.png";
  ngOnInit() { }
}
```

Zápis v HTML kóde:

```
<img src={{ imagePath }} class="obrazok"/>
```

Výstupom bude táto obrazovka:

Obr. 15: Ukážka interpolácie reťazcov v Angulari (adresy obrázkov)

Väzba vlastností – umožňuje zviazanie obrazovky (view) s určitým výrazom, ktorý je definovaný pre šablónu. V skratke môžeme povedať, že sa jedná o možnosť meniť hodnotu určitej premennej komponentu (model) a následne jej zobrazenie na obrazovke (prezentačná vrstva). Jedná sa o jednosmerný spôsob, ktorý nám umožňuje kedykoľvek túto hodnotu zmeniť, ale iba na úrovni daného komponentu. Oproti interpolácii reťazcov sa na prvý pohľad nemusí tento spôsob javiť ako odlišný. Vyššie uvedený kód pre zobrazenie cesty obrázka by v tomto prípade vyzeral takto:

```
<img [src]="imagePath" class="obrazok"/>
```

Výstup na obrazovku by bol totožný s obrázkom č. 3.6, pretože sa jedná iba o dosadenie reťazca do atribútu. Prvé rozdiely nastanú keď budeme chcieť pracovať napr. s logickými hodnotami. Ukážeme si to na názornom príklade. Ako prvé upravíme obsah HTML súboru komponentu č.1 (*stranka1.component.html*):

```
<div>
  <h2>{{title}}</h2>
  <div class="sekcia">
     <h4>Interpolácia reťazcov</h4>
     <img src={imagePath}} class="image-adjustment"/>
     <button disabled={{currentValue}}>KLIKNI</button>
  </div>
  <div class="sekcia">
     <h4>Väzba vlastností</h4>
     <img [src]="imagePath" class="image-adjustment"/>
     <button [disabled]="currentValue"
(click)="onClick()">KLIKNI</button>
  </div>
  <div style="clear:both;"></div>
</div>
```

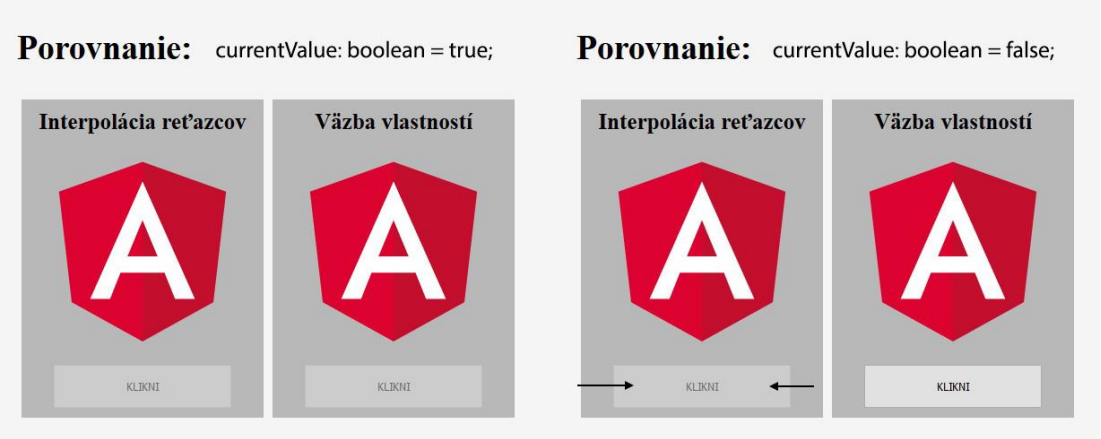
Rozdelili sme obsah na dva elementy, v každom z nich je umiestnené rovnaké tlačítko ale zapísané odlišným spôsobom. Vytvoríme aj jednoduché štýly (*stranka1.component.css*):

```
h4 {
     padding: 0;
     margin: 0;
}

button {
     width: 200px;
     height: 50px;
}

.sekcia {
     width: 250px;
     background: #B8B8B8;
     text-align: center;
     padding: 10px;
     margin: 5px;
     float: left;
}
```

A nakoniec definujeme v súbore *stranka1.component.ts* novú premennú logického typu boolean s názvom *currentValue*:

```
export class Stranka1Component implements OnInit {

  constructor() { }

  title = 'Porovnanie:';
  imagePath: string = "assets/Angular.png";
  currentValue: boolean = true;

  ngOnInit() {}
}
```

Výstup na obrazovku bude nasledujúci (defaultný stav vľavo):



Obr. 16: Ukážka rozdielov medzi intepoláciou reťazcov a väzbou vlastností

V prípade pravdivej hodnoty premennej (true) budú obidve tlačítka zablokované a nebude možné na nich kliknúť. Zmena nastane, keď hodnotu premennej zmeníme na nepravdivú (false). Vtedy tlačítko, ktorého stav spravuje interpolácia reťazcov ostane zablokované aj navzdory tomu, že už by nemalo byť zablokované (variant vpravo na obrázku). Tlačítko spravované väzbou vlastností sa po správnosti odblokuje. V prípade použitia reťazcov je vhodný prvý spôsob a v prípade výrazov, ktoré nie sú reťazcami je optimálne použiť druhý spôsob.

Väzba udalostí – definuje možnosť úpravy a poslania určitej hodnoty alebo informácie uloženej v premennej a to smerom z prezentačnej vrstvy (view) do komponentu (model). Praktický príklad si môžeme ukázať znovu na použití tlačítka. Po kliknutí na tlačítko s textom „KLIKNI“ sa tento text zmení na „UZ SOM KLIKOL“.

V Angulari sa udalosť kliknutia nazýva ako *(click)*, navzdory inému názvu sa stále jedná o udalosť JavaScriptu *onClick()*, ktorú uvedený príkaz Angularu zavolá.

Upravíme zdrojový kód súboru *stranka1.component.html*:

```
<div>
  <h2> {{ title }} </h2>
  <div class="sekcia">
  <h4>Väzba vlastností</h4>
  <img [src]="imagePath" class="image-adjustment"/>
  <button (click)="changeMyTitle()">{{title_button}}</button>
  </div>
  <div style="clear:both;"></div>
</div>
```

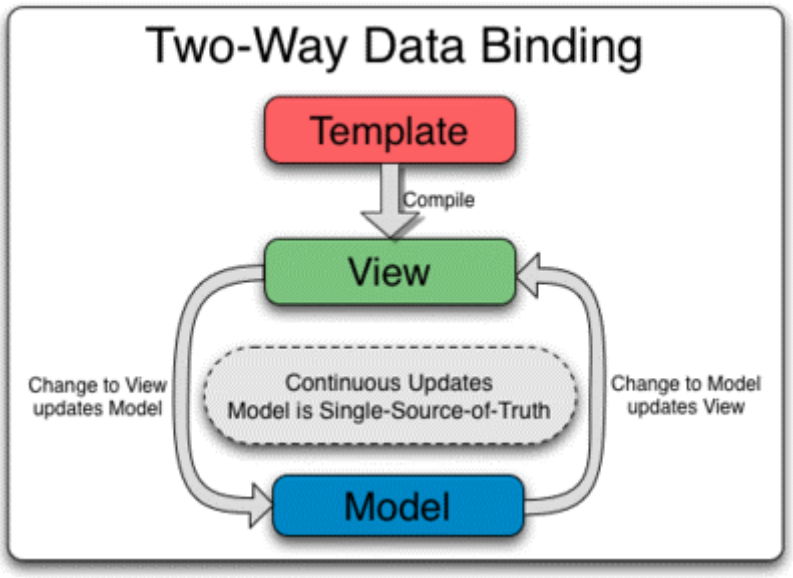
Titulok tlačítka „KLIKNI“ už nie je definovaný napevno, ale využili sme interpoláciu reťazca. Teraz definujeme novú premennú *title_button* aj v súbore *stranka1.component.ts*:

```
export class Stranka1Component implements OnInit {

  constructor() { }
  title = 'Porovnanie:';
  title_button = 'KLIKNI';
  imagePath: string = "assets/Angular.png";

  ngOnInit() {}
  changeMyTitle() {
     this.title_button = 'UZ SOM KLIKOL';
  }
}
```

Po kliknutí na tlačítko sa aktivuje udalosť JavaScriptu onClick a zavolá sa nami definovaná funkcia *changeMyTitle()* ktorá prepíše obsah naviazanej premennej na nový text.
Výstup na obrazovku vyzerá nasledovne.

Obr. 17: Ukážka väzby vlastností v Angulari

Dvojsmerná väzba dát – je kombináciou väzby vlastností a udalostí. Jedná sa o synchronizáciu dát z obrazovky (view) do komponentu (model) a naopak. Zmeny dát vykonané na úrovni komponentu by mali byť synchronizované s obrazovkou a mali by okamžite aktualizovať model do najnovšieho stavu.



Obr. 18: Schéma dvojsmernej väzby dát, Zdroj [28]

Tento spôsob väzby dát je primárne využívaný pri rôznych druhoch formulárov kde používateľ zadáva dáta. Vytvára tak zmeny na obrazovke, ktoré sa premietnu do modelu a zase naopak. Angular na implementáciu dvojsmernej dátovej väzby využíva kombináciu vlastností a udalostí za pomoci direktívy *ngModel*.

Dôležité je poznamenať, že *ngModel* nie je natívnou súčasťou knižnice Angularu. Je dostupný v rámci modulu formulárov s názvom *FormsModule* a je potrebné ho naimportovať v súbore aplikácie *app.module.ts* takto:

```
import { FormsModule } from "@angular/forms";
```

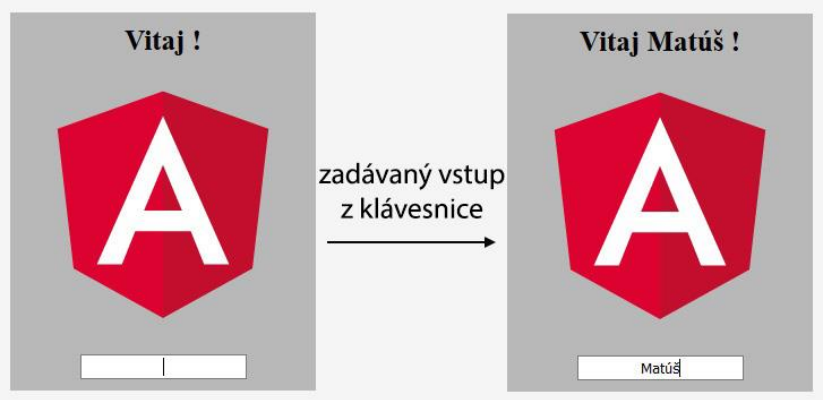
Zároveň ho potom musíme uviesť aj v časti imports:

```
imports: [
  BrowserModule,
  RouterModule.forRoot(appRoutes),
  FormsModule
],
...
```

Jeho použitie si ukážeme na textovom poli, ktoré bude reagovať na vstup používateľa. Ako prvý krok upravíme obsah HTML súboru *stranka1.component.html*:

```
<div>
  <div class="sekcia">
  <h4>Vitaj {{userName}} !</h4>
  <img [src]="imagePath" class="image-adjustment"/>
  <input type="text" [(ngModel)]="userName"/>
  </div>
  <div style="clear:both;"></div>
</div>
```

Výstup na obrazovku bude:



Obr. 19: Zmena textového pola v Angulari

Pri úvodnom stave aplikácie máme hodnotu (*userName)* umiestnenú v nadpise štvrtej úrovne prázdnu. Je to z dôvodu jej naviazanosti na prázdne textové pole pod obrázkom. Pri písaní do tohto poľa sa dynamicky začne táto hodnota meniť na text ktorý zadávame [29].

3.3 Úvod do Reactu

Na úvod je vhodné poznamenať, že React (alebo aj ReactJS) je oficiálne zaraďovaný medzi **knižnice** pre tvorbu UI. Na druhú stranu napĺňa aj vlastnosti, ktoré sú charakteristické pre frameworky, z toho dôvodu je k nemu tak pristupované aj v tejto práci.

React nepôsobí tak komplexne a robustne ako Angular, jeho hlavnými vlastnosťami sú jednoduchosť, deklaratívnosť a dostupnosť. Je určený na vývoj rozsialych aplikácii v ktorých sa dáta menia dynamicky. Hlavnou výhodou oproti Angularu je použitie virtuálneho DOM a rozšírenia JSX, ktoré už boli popísané v predchádzajúcich kapitolách. V tejto kapitole si v Reacte postupne popíšeme základné funkcie a prejdeme obdobnú funkcionalitu toho, čo sme už videli pri Angulari vrátane ukážok kódu.

3.3.1 Prvé spustenie a CLI

Ako už bolo spomenuté, rovnako ako pri Angulari tak aj pri Reacte je nutné mať nainštalovaný interpreter NodeJS ktorého súčasťou je balíčkovací systém NPM.

Základné konzolové príkazy (CLI)

npm install – jeho zavolanie vo zvolenom adresári aplikácie nainštaluje všetky potrebné súčasti v adresári *node_modules* (knižnice, rozšírenia atď).

npm start – príkaz pre spustenie serveru s aplikáciou, ktorá je po spustení prístupná na linku http://*localhost:3000* v našom počítači. Rovnako ako v prípade Angularu, po zavolaní ostáva server spustený a uplatňuje nami vykonané zmeny bez nutnosti obnovenia stránky.

create-react-app <nazov> – príkaz pre vytvorenie novej React aplikacie. Rovnako ako pri Angulari, CLI za nás vytvorí potrebnú projektovú štruktúru a predpripraví konfiguračné súbory a súčasti. Defaultne sa inštalujú súčasti ako rozšírenie JSX, React DOM, podpora TypeScriptu a ďalšie.

Vytvorenie prvej aplikácie

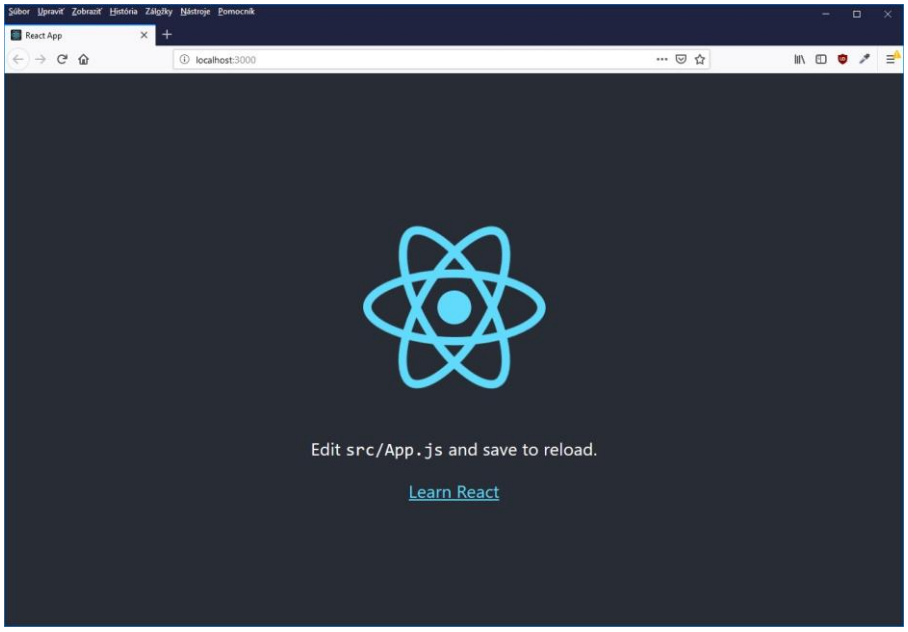
V termináli nastavíme cestu k adresáru, v ktorom chceme aplikáciu vytvoriť (v mojom prípade C:\React). Zadáme príkaz na vytvorenie novej aplikácie a zvolíme jej názov (v mojom prípade myapp):

C:\React\> create-react-app myapp

Vo zvolenom adresári sa nám automaticky vytvorila projektová štruktúra našej aplikácie so zvoleným názvom **myapp**. Pre spustenie aplikácie je potrebné v konzole nastaviť cestu k tomuto adresáru (mimo neho nebude spustenie fungovať). Oproti Angularu trvá vytváranie aplikácie o niečo dlhšie. Aj v tomto prípade React predinštaluje základné súčasti aplikácie. Pre spustenie potom zadáme príkaz:

C:\React\myapp\> npm start

Po zadaní tohto príkazu sa spustí server s našou aplikáciou, ktorá je prístupná na adrese **http://localhost:3000**. Aplikácia vytvorená Reactom obsahuje iba rotujúce logo frameworku a jednoduchú inštrukciu, ktorá nám hovorí, že môžeme začať editáciou hlavného aplikačného súboru *App.js*. Zároveň obsahuje aj odkaz na stránky Reactu a teda aj prístup k podrobnejšej dokumentácii [30].



Obr. 20: Zobrazenie základnej aplikácie Reactu

3.3.2 Projektová štruktúra

Štruktúra, ktorá nám vznikla v projektovom adresári aplikácie obsahuje viacero rôznych súborov a je výrazne jednoduchšia od tej v Angulari. Pre jej pochopenie ako celku si popíšeme tie najhlavnejšie časti:

- node_modules/ (nainštalované moduly aplikácie, ktoré si ukladá npm)
- public (obsahuje súbor šablóny)
    - **index.html** (HTML kód aplikácie)
- **src/** (aplikačný adresár, obsahuje súbory zdrojového kódu)
    - **App.js** (hlavný komponent aplikácie App)
    - **index.js** (súbor Javascriptu prepojený so súborom index.html)
    - **App.css** (kaskádové štýly aplikácie pre komponent App)
    - **index.css** (kaskádové štýly pre index.js)
- package.json (konfiguračný súbor, obsahuje meta dáta o aplikácii)

3.3.3 React Router

V tejto časti budeme vytvárať rovnaké sekvencie ukážok z malých aplikácií, tak ako v prípade Angularu. Bude zameraná na základnú funkcionalitu navigácie, tzn. prepínanie sa medzi jednotlivými obrazovkami aplikácie. Ukážku aplikujeme na vytvorenú aplikáciu

myapp. Vytvoríme jednoduché menu o dvoch položkách (stranka1 a stranka2), ktoré bude slúžiť na zobrazenie obsahu pre jednotlivé stránky.

1. Vytvorenie nových komponentov

Narozdiel od Angularu, vytváranie komponentov v Reacte väčšinou pozostáva zo skopírovania a vloženia súborov už existujúceho komponentu s následným premenovaním. Vytvorenie komponentu podobnou formou ako sme videli pri Angulari je samozrejme možné, ale na to už je potrebné doinštalovanie rozšírenia. Obecne zaužívanou praktikou je vytvorenie adresára *components* v hlavnom adresári aplikácie *src*.

Vo vyššie uvedenom adresári si potom vytvoríme tieto súbory:

- stranka1.js
- stranka2.js
- stranka.css

Štruktúra tak môže byť výrazne voľnejšia než v prípade Angularu. Tam má každý komponent samostatný adresár a v ňom hneď štyri súbory. V tomto prípade si pre obidva komponenty môžeme spraviť jeden súbor kaskádových štýlov, ktorý môže obsahovať spoločný kód. Pri Angulari to až tak jednoduché nie je, keďže každý komponent má svoj vlastný súbor štýlov.

Zdrojový kód súboru *stranka1.js*:

```
import React, { Component } from 'react';
class stranka1 extends Component {
  render() {
    return ( <p>stranka1 works!</p> );
    /* analogicky pre stranka2:
    /*<p>stranka2 works!</p> */
  }
}
export default stranka1;
```

Už v tejto časti môžeme vidieť značné odlišnosti v zápise kódu pri oboch frameworkoch. React vracia výsledný HTML kód priamo z funkcie *render*, takže je možné bez problémov kombinovať kód JavaScriptu a HTML bez toho, aby sme boli obmedzení syntaxou. Takto vytvorené komponenty je pre použitie samozrejme nutné naimportovať (podobným spôsobom ako v Angulari).

Zdrojový kód súboru *App.js*:

```
import React, { Component } from 'react';
import Stranka1 from './components/stranka1.js';
import Stranka2 from './components/stranka2.js';
import './App.css';
class App extends Component {
  render() {
    return (
      <div className="App">
        <Stranka1/>
        <Stranka2/>
      </div>
    );
  }
}
export default App;
```

Zdrojový kód *index.js*:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

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

Výstup funkcie *return* musí byť vždy „obalený“ v hlavnom HTML elemente (napr. div), inak sa aplikácia nespustí a vypíše varovné hlásenie. Ďalším dôležitým pravidlom je to, že **je potrebné importovať názvy komponentu s veľkým začiatočným písmenom** (v mojom prípade Stranka1). Pri samotnom názve súboru je to jedno, ale ak by sme pri importe v tomto prípade napísali *import stranka1*, React by síce nevypísal žiadne chybové hlásenie ale ani by sa nám nič nezobrazilo. Ďalším pravidlom je zápis volania komponentov s veľkým začiatočným písmenom. Komponenty sa od klasických HTML elementov odlišujú práve veľkým začiatočným písmenom. Ak by sme naimportovali názov *stranka1* a následne ho volali v kóde ako *<stranka1/>*, došlo by k zámene s HTML elementom a nezobrazilo by sa nám nič. Vhodné je poznamenať aj to, že v prípade syntaxe Reactu nepoužívame pre HTML elementy atribút *class*. Je to z dôvodu, že slovo *class* je vyhradené jazykom JavaScript a keďže React je jeho nadstavbou preto používame *className*.

V tejto časti využijeme štýly a HTML kód z predchádzajúcej kapitoly o Angulare a na ich vytvoríme tak v Reacte rovnakú ukážku navigácie. Keďže React natívne neobsahuje modul pre routing (narozdiel od Angularu ktorý sa naň hneď pri inštalácii pýta), je potrebné cez príkazový riadok doinštalovať do adresára našej aplikácie rozšírenie *react-router-dom*:

npm install --save react-router-dom

Následne komponent importujeme globálne v rámci celej aplikácie. Na to využijeme súbor index.js ktorý bude vo výsledku slúžiť ako vykresľovací medzičlánok. Do vrchnej časti potom pridáme import modulu *reactDOM*:

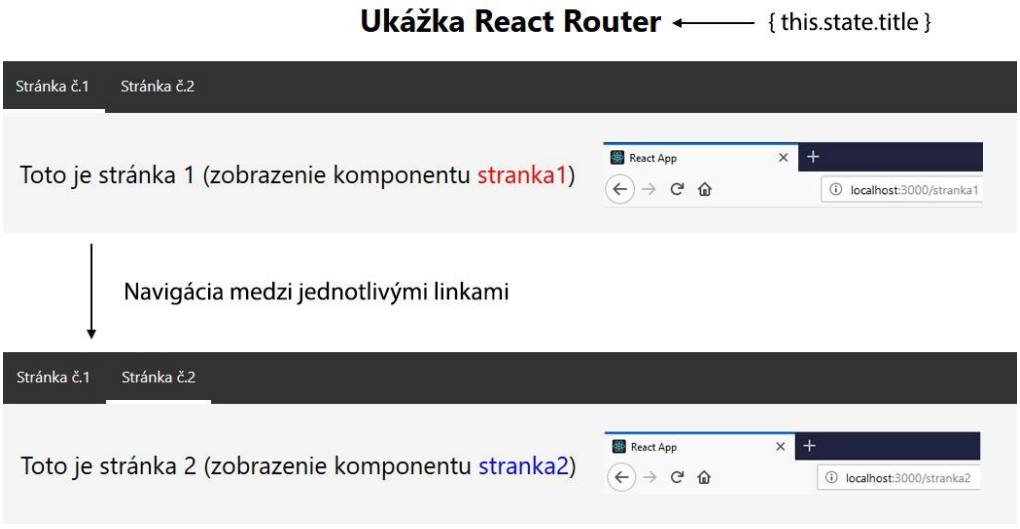
```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render((
  <BrowserRouter>
    <App />
  </BrowserRouter>
), document.getElementById('root'));
serviceWorker.unregister();
```

V dolnej časti sme koreňový element *App* „zabalili“ do importovaného modulu, tým pádom bude celá aplikácia mať prístup k jednotlivým cestám a linkom. Hlavný aplikačný súbor *App.js* upravíme takto:

```
import React, { Component } from 'react';
import Stranka1 from './components/stranka1.js';
import Stranka2 from './components/stranka2.js';
import { Switch, Route, NavLink } from 'react-router-
dom'
import './App.css';

class App extends Component {

 constructor(props) {
    super(props);
    this.state = { title: "Ukážka React Router" };
 }

render() {
  return (
  <div style={{textAlign: 'center'}}>
    <h1>{ this.state.title }</h1>
```

```
    <ul>
      <li><Link exact to='/' activeClassName='active-
link'>Stránka č.1</Link></li>
      <li><Link to='/stranka2' activeClassName='active-
link'>Stránka č.2</Link></li>
    </ul>
   <div class="obsah_sekcie">
  <Switch>
    <Route exact path="/" component={Stranka1}/>
    <Route path='/stranka1' component={Stranka1}/>
    <Route path='/stranka2' component={Stranka2}/>
  </Switch>
  </div>
  </div>);
  }
}
export default App;
```

Rovnako ako v prípade Angularu vieme prenášať hodnotu reťazcov cez interpoláciu, ale v tomto prípade sa výraz umiestneňuje iba do dvoch zložených zátvoriek. React zároveň pristupuje k premenným a stavom aplikácie odlišne, využíva *props* a *state*. Na začiatku sme si definovali hodnotu title v samotnom konštruktore triedy a následne sme v kóde k nemu pristúpili pomocou *this.state.title*. V porovnaní samotnej navigácie, má Angular pre routing definovaný vlastný modul v samostatnom súbore. V HTML časti je potom vkladaný element <*router-outlet*> kvôli prepojeniu (aby mala aplikácia prístup k údajom o jednotlivých cestách). React na to automaticky modul vyhradený nie je, ale nie je problém si ho vytvoriť a použiť ho rovnakým spôsobom ako pri Angulari. V Reacte koreňový element aplikácie vkladáme do elementu <*BrowserRouter*> [31].



Obr. 21: Ukážka navigácie React Router

3.3.4 Väzba dát

Interpolácia reťazcov

Interpolácia reťazcov je v Reacte realizovaná rovnakým spôsobom ako v Angulari s tým rozdielom, že výraz sa uvádza iba v jednom páre zložených zátvoriek (vyššie uvedený príklad s pristúpením k premennej cez *this.state.title*).

Props a State

Props a State sú základným stavebným kameňom fungovania aplikačnej logiky v Reacte. Keďže komponenty Reactu je možné do seba vkladať, je potrebné aby existoval prostriedok pre predávanie dát medzi sebou. Týmto prostriedkom sú *props*, zároveň môžeme povedať, že prakticky tvoria hlavnú kostru React aplikácií. Predávať môžeme funkcie aj objekty, pričom za všetkými operáciami stojí čistý JavaScript. Props fungujú iba jednosmerne smerom zhora nadol, takže komponent nemôže zmeniť *props* iného komponentu. Je možné od rodičovského komponentu cez *props* obdržať callback a ten zavolať. Callback môžeme jednoducho definovať ako funkciu ktorá je volaná inou funkciou, pričom táto druhá funkcia berie prvú funkciu ako parameter. Tento jednosmerný prístup výrazne redukuje komplexnosť React aplikácií. Pri zmenách v *props* a *state* sa volá metóda *render()* a s týmto volaním sa zároveň znova renderuje celý podstrom potomkov. To znamená, že ak by sme vykonali zmeny v *props* alebo *state* v rámci rodičovského komponentu, zavolajú sa automaticky aj *render* metódy u jeho potomka.

Rovnako ako na príklade Angularu si u obidvoch vytvorených komponentov definujeme *state* pre obrázky umiestnené v adresári *assets*. Súbory *stranka1*.js a *stranka2*.js potom upravíme takto:

```
import React, { Component } from 'react';

class stranka1 extends Component {

 constructor(props) {
    super(props);
    this.state = {
      imagePath: require( "../assets/React.png" )
   /* analogicky pre stranka2.js "../assets/Angular.png"
*/
    };
  }
  render() {
    return (
```

```
        <div>
           <p>Toto je stránka 1 (zobrazenie komponentu
           <span className="kmp1">stranka1</span>)</p>
           <img src={this.state.imagePath} />
        </div>
        );
      }
}
export default stranka1;
```

Obrázky môžeme v Reacte vkladať dvoma spôsobmi. Prvým z nich je import, rovnako ako v prípade komponentov. Zápis tejto varianty vyzerá takto:

```
import Obrazok from "../assets/React.png";
```

Ďalším spôsobom je uloženie adresy obrázku do *state* za použitia funkcie *require*.

Obr. 22: Ukážka state a props v Reacte

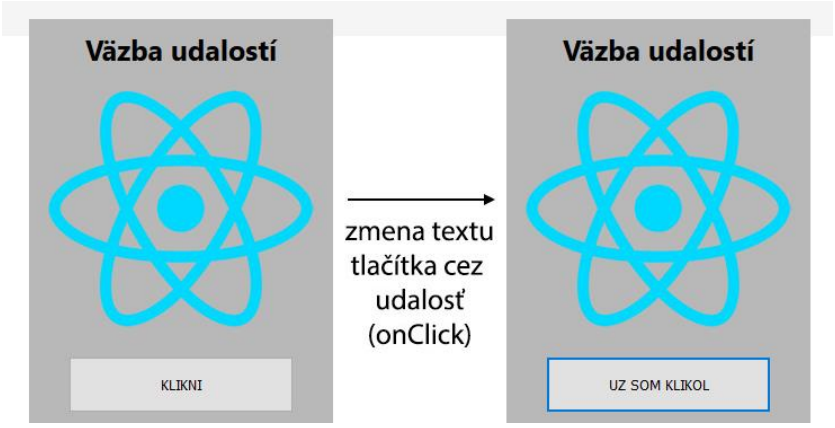
Väzba udalostí

Riadenie udalostí v Reacte si ukážeme na rovnakom príklade ako pri Angulari. Po kliknutí na tlačítko s textom „KLIKNI“ sa tento text zmení na „UZ SOM KLIKOL“. Znovu si definujeme začiatočný *state* pre text tlačítko a potom na samotné tlačítko naviažeme udalosť *onClick* ktorá zavolá funkciu *changeMyTitle()*. Zmeny vykonáme v súbore komponentu *stranka1.js*:

```
import React, { Component } from 'react';

class stranka1 extends Component {

 constructor(props) {
    super(props);
    this.state = {
      imagePath: require( "../assets/React.png" ),
```

```
        title_button: "KLIKNI"
      };
    }

    changeMyTitle() {
      this.setState({ title_button: "UZ SOM KLIKOL" });
    }
    render() {
      return (
       <div>
          <div class="sekcia">
            <h4>Väzba udalostí</h4>
            <img src={this.state.imagePath} />
            <button
onClick={this.changeMyTitle.bind(this)}>{
this.state.title_button }</button>
            </div>
      </div>
      );
    }
}
export default stranka1;
```

V tomto prípade sme vytvorili bezparametrickú funkciu *changeMyTitle()*, ktorej úlohou je zavolať príkaz *setState*. Ten nám umožňuje kedykoľvek meniť nami definovaný počiatočný stav pre tlačítko. Dôležitou časťou pri naväzovaní udalostí je metóda *bind*. Táto metóda vytvára novú funkciu, ktorej volanie nastaví kľúčové slovo *this* na stanovenú hodnotu [32].

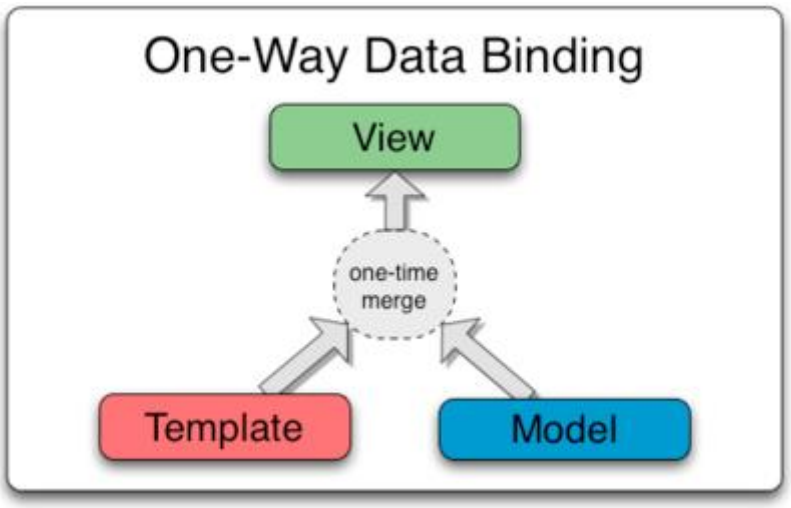


Obr. 23: Ukážka väzby vlastností v Reacte

Jednosmerná väzba dát

Aplikácie písané v Reacte sú organizované ako série návzajom závislých komponentov. Ako už bolo spomenuté vyššie, tieto komponenty získavajú dáta cez argumenty (*props*) a zároveň

predávajú informácie cez návratové hodnoty funkcie *return()*. Tento prístup sa nazýva aj jednosmerným tokom dát (unidirectional data flow). Obecne zaužívaným pravidlom vývoja je, že nami písaný kód by sa nemal snažiť priamo čítať alebo modifikovať DOM. Písanie kódu na úrovni selektorov za účelom konkrétneho prístupu k elementom na stránke je nevhodnou praktikou.



Obr. 24: Schéma jednosmernej väzby dát, Zdroj [33]

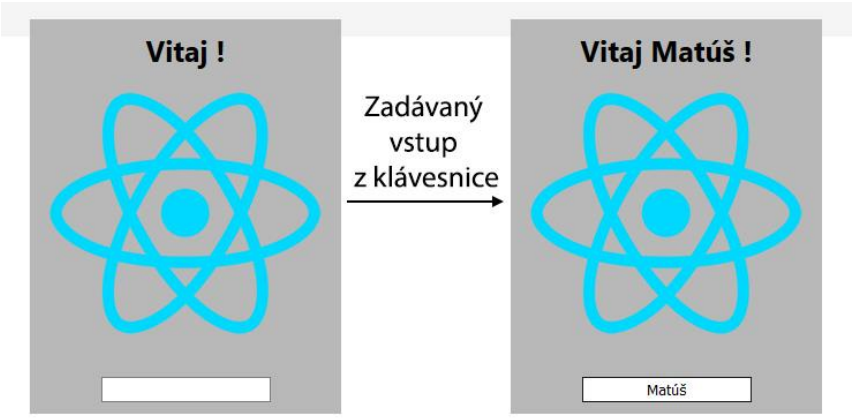
V prípade Angularu sme pri textovom poli predávali hodnotu pomocou špeciálneho atribútu *ngModel*, ktorý súčasne reprezentuje fungovanie dvojsmernej komunikácie. V prípade Reactu, ktorý funguje iba na jednosmernom princípe je implementácia zložitejšia. Uvedieme si rovnaký príklad zmeny textového pola pri interakcii používateľom v Reacte.

Upravíme komponent *stranka1.js*:

```
import React, { Component } from 'react';

class stranka1 extends Component {

 constructor(props) {
    super(props);
    this.state = {
      imagePath: require( "../assets/React.png" ),
       userName: ""
    };
     this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    this.setState({userName: event.target.value});
  }
```

```
  render() {
    return (
     <div>
        <div class="sekcia">
          <h4>Vitaj {this.state.userName} !</h4>
          <img src={this.state.imagePath} />
          <input type="text"
onChange={this.handleChange} value={this.state.userName}
/>
          </div>
    </div>
    );
  }
}
export default stranka1;
```

Výstup na obrazovku:



Obr. 25: Zmena textového pola v Reacte

3.4 Finálne zhodnotenie a odporúčanie

V tomto záverečnom zhrnutí frameworkov si zrekapitulujeme jednotlivé vlastnosti a poznatky o týchto technológiách.

Z pohľadu výkonu a objemnosti aplikácií získava navrch jednoznačne React. V testoch sa ukázal ako výkonnejšia alternatíva, ktorá zaberá menej miesta. Následne z pohľadu vývoja to môže byť mierne diskutabilné, pretože rovnako ako React tak aj Angular majú svoje nesporné výhody. Z môjho subjektívneho pohľadu sa mi lepšie pracovalo s Reactom, jeho štruktúra je voľnejšia a jednoduchšia na pochopenie. V prípade Angularu bol úvod relatívne ťažký, dokumentácia je písaná pomerne zložito a pre úplneho začiatočníka to môže byť veľmi náročné.

Na druhú stranu zdielanie dát medzi komponentami mi príde jednoduchšie v Angulari, práve vďaka dvojsmernej dátovej väzbe. V prípade Reactu je nutné písanie callback funkcií, aplikácia musí vedieť že ma vykonať zmeny v rodičovskom komponente a znovu ho vykresliť na základe zmien v jeho potomkoch. React zároveň obsahuje stovky veľmi nápomocných rozšírení, ktoré ocení nejeden vývojar, najmä pre ich jednoduchú integráciu v aplikácii. Z môjho osobného pohľadu je React vhodnejší najmä pre malé až stredné veľké aplikácie. Angular je robustnejšia alternatíva ktorá je skôr určená pre väčšie projekty a tomu je aj prispôsobená jeho štruktúra.

Adaptácia na obidva tieto frameworky je náročná, keďže obidva frameworky fungujú veľmi odlišne. Počas písania jednotlivých porovnaní a ukážok medzi nimi bolo nutné sa prepnúť na „iný spôsob myslenia“ pre daný framework. Z celého porovnania by som označil za jednoduchšiu alternatívu React, pričom Angular mi prišiel miestami až prehnane zložitý a to najmä vo svojej štruktúre modulov a ich fungovania. Klasický React ale tiež naráža na určité problémy, v mojom prípade to bolo prenášanie dát z rodičovského komponentu do potomkov, ktoré sa mi zdalo miestami až príliš komplikované. Väčšinu z týchto vecí ale rieši alternatíva Redux.

Záverom tejto kapitoly by som dodal, že z celého tohto porovnania mi bol sympatickejšou alternatívou React a viem si predstaviť jeho využitie vo svojich projektoch skôr než konkurenčný Angular.

Záver

Cieľom tejto bakalárskej práce bolo analyzovať dva najpoužívanejšie JavaScript frameworky súčasnosti, ktorými sú Angular a React. V logickej postupnosti štruktúry práce sme v jednotlivých krokoch analyzovali najskôr princípy fungovania frameworkov, vysvetlili sme si samotnú architektúru a popísali konkrétne charakteristické vlastnosti.

Úvodná kapitola popisuje základné pojmy a definície z tejto oblasti, ktoré je nutné poznať pre lepšie pochopenie problematiky. Poskytuje detailnejší náhľad do samotnej filozofie a princípov, na ktorých sú založené technológie frameworkov a zároveň aj na praktickom príklade porovnáva vývoj bez a s využitím frameworku.

Nasledujúca kapitola obsahuje krátky popis metodiky skúmania práce, vysvetľuje postupy komparatívnej analýzy a postupnosť logického štruktúrovania práce v presne stanovených krokoch.

Nosnou časťou práce je tretia kapitola. Obsahuje dve podkapitoly, ktoré sú osobitne venované úvodu do Angularu a Reactu. Tieto podkapitoly sú už priamo venované opisu vlastností zvolených frameworkov. Úvod do obidvoch frameworkov je písaný od absolútneho začiatku procesu vývoja, teda od inštalácie potrebných nástrojov a základných konzolových príkazov až po fungujúce ukážky z reálnej aplikácie. Doplnený je o ukážky zdrojového kódu a obrázky zo samotných aplikácií, na ktorých sú opisované vlastnosti prezentované. Celá kapitola je zameraná tak, aby odrážala implementáciu rovnakých princípov vývoja v dvoch rozličných frameworkoch a umožnila určiť vhodné ukazovatele pre súhrnné porovnanie.

V súčasnosti je použitie frameworkov na vzostupe. Všetky veľké spoločnosti ako napr. Facebook alebo YouTube používajú priamo konkrétny framework alebo jeho určitú implementáciu. V otázke použitia pre malých a stredných vývojárov je to skôr otázka preferencií, ale podľa môjho názoru sa túto možnosť jednoznačne oplatí vyskúšať.

Zoznam použitej literatúry

Knižné zdroje:

[1] RAUSCHMAYER Axel. 2012. The Past, Present, and Future of JavaScript O'Reilly Media, 2012, 1. vydanie, 56 s. ISBN: 978-1-44-934354-5

[2] WILLIAMSON Ken. 2015. Learning AngularJS: A Guide to AngularJS Development O'Reilly Media, 2015, 1. vydanie, 212 s. ISBN-13: 978-1-49-191675-9

Internetové zdroje:

[3] What is JavaScript [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.makeuseof.com/tag/what-is-javascript/

[4] How to add JavaScript to HTML [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-add-javascript-to-html

[5] What is JavaScript framework [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://skillcrush.com/2018/07/23/what-is-a-javascript-framework/

[6] Single page application [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: http://jecas.cz/spa

[7] Whats is the DOM? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.w3.org/TR/1998/WD-DOM-19980720/introduction.html

[8] Whats is the Virtual DOM? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://reactjs.org/docs/faq-internals.html

[9] Whats is the TypeScript? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.tutorialspoint.com/typescript/typescript_overview.htm

[10] Introducing JSX [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html

[11] Čo je NodeJS a kedy ho použiť [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://linuxos.sk/spravy/detail/co-je-to-nodejs-a-kedy-ho-pouzit/

[12] Two-Way Data Binding [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.accelebrate.com/blog/two-way-data-binding-angular-2-and-react/

[13] MVC vs MVP vs MVVM [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

[14] Architecture patterns [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.codeguru.com/csharp/.net/net_general/differences-among-mvc-mvp-and-mvvm-design-patterns.html

[15] Co je to flux? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.zdrojak.cz/clanky/potrebujeme-flux/

[16] Best JavaScript frameworks [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://geekflare.com/best-javascript-frameworks/

[17] React vs Angular [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://appinventiv.com/blog/react-vs-angular

[18] React vs Angular Compared: Which One Suits Your Project Better? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.altexsoft.com/blog/engineering/react-vs-angular-compared-which-one-suits-your-project-better/

[19] Github: React [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://github.com/facebook/react

[20] Github: Angular [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://github.com/angular/angular.js

[21] React vs Angular vs Vue.js – What to choose in 2019? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d

[22] Benchmarking Angular, React and Vue [cit. 2019-04-22]. Dostupné z: https://blog.bitsrc.io/benchmarking-angular-react-and-vue-for-small-web-applications-e3cbd62d6565

[23] Introduction to Chrome Lighthouse [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://medium.com/backticks-tildes/introduction-to-chrome-lighthouse-50ac623734cb

[24] Čo sú to direktívy? [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://angular.sk/kurz/9-strukturalne-a-atributove-direktivy

[25] Úvod do Angular [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://angular.sk/kurz/1-uvod-do-angular-2

[26] Angular: Základy [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.itnetwork.cz/javascript/angular/zaklady/uvod-do-angular-frameworku

[27] Route configuration [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://codecraft.tv/courses/angular/routing/route-configuration/

[28] Obrázok (upravený). Dostupné z: http://marcinkowalczyk.pl/warsztaty/uph/angular/user/pages/02.materialy-warsztaty/04.podstawy/Two_Way_Data_Binding.png

[29] Data binding in Angular [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://dzone.com/articles/data-binding-in-angular

[30] Create apps with no configuration [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://reactjs.org/blog/2016/07/22/create-apps-with-no-configuration.html

[31] Simple React Router tutorial [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://blog.pshrmn.com/entry/simple-react-router-v4-tutorial/

[32] React: props vs state [online]. Bratislava, 2019 [cit. 2019-04-22]. Dostupné z: https://www.dzejes.cz/react-props-vs-state.html

[33] Obrázok (upravený). Dostupné z: http://marcinkowalczyk.pl/warsztaty/uph/angular/user/pages/02.materialy-warsztaty/04.podstawy/One_Way_Data_Binding.png