Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Analýza ekosystému React

David Pavel

© 2019 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

David Pavel

Informatika

Název práce

Analýza ekosystému React

Název anglicky

React environment analysis

Cíle práce

Bakalářská práce je tematicky zaměřena na ekosystém React. Cílem práce je analýza ekosystému React s následným vývojem reálné aplikace. Dílčí cíle jsou:
- vypracování přehledu ekosystémů pro vývoj front-end aplikací,
- vypracování přehledu webových technologií na straně klienta.

Metodika

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní práce spočívá v analýze ekosystému React v závislosti na modelových případech a následné realizaci front-endového řešení webové aplikace. Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry bakalářské práce.

Doporučený rozsah práce

40 – 50 stran textu.

Klíčová slova

front-end, web, ekosystém, Javascript, React, JSX, Angular, Vue.js

---

Doporučené zdroje informací

Angular Docs: Typescript. Angular [online]. Mountain View (CA): Google, 2016- [cit. 2017-04-16]. Dostupné z: https://angular.io/docs/ts/latest/

Design Principles. React Docs [online]. Menlo Park (CA): Facebook, 2004- [cit. 2017-04-16]. Dostupné z: https://facebook.github.io/react/contributing/design-principles.html

Front-end JavaScript frameworks. GitHub [online]. San Francisco (CA): GitHub, 2008- [cit. 2017-04-16]. Dostupné z: https://github.com/showcases/front-end-javascript-frameworks

Hacking with React. Hacking with React [online]. Bath, UK: Hudson, 2016- [cit. 2017-04-16]. Dostupné z: http://www.hackingwithreact.com/

Rangle's Angular Training Book. Rangle.io [online]. Toronto, ON: Rangle.io, 2015- [cit. 2017-04-16]. Dostupné z: https://angular-2-training-book.rangle.io/

React Enlightenment. React Enlightenment [online]. Meridian (ID): Lindley, 2016- [cit. 2017-04-16]. Dostupné z: https://www.reactenlightenment.com/

Thinking in React. React Docs [online]. Menlo Park (CA): Facebook, 2004- [cit. 2017-04-16]. Dostupné z: https://facebook.github.io/react/docs/thinking-in-react.html

Vue.js Guide. Vue.js [online]. New York: You, 2014- [cit. 2017-04-16]. Dostupné z: http://vuejs.org/v2/guide/

Předběžný termín obhajoby

2018/19 LS – PEF

Vedoucí práce

Ing. Jan Masner, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 11. 9. 2018

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 18. 10. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 15. 03. 2019

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Analýza ekosystému React" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15. 3. 2019 ______________________________

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce Ing. Janu Masnerovi, Ph.D. za cenné rady a odborné vedení při zpracovávání této práce.

Analýza ekosystému React

Abstrakt

Aplikace se postupně přesouvají z osobních počítačů na web, který se vznikem nových webových technologií umožňuje stále komplikovanější a mocnější aplikace. Pro vývoj těchto aplikací vzniká mnoho různých knihoven a nástrojů, které se uskupují do digitálních ekosystémů. Kvůli jejich množství je ale o to náročnější pro vyvíjenou aplikaci vybrat nejvhodnější kandidáty.

Cílem práce je analyzovat nejpopulárnější z těchto ekosystémů, a webové technologie s nimi souvisejícími, a poté získané znalosti uplatnit při vývoji reálné aplikace.

V teoretické části se nachází porovnání dnešních populárních frameworků a knihoven, a podrobné rozebrání nejpopulárnější z nich, Reactu, včetně některých knihoven určených pro jeho rozšíření. Praktická část pak popisuje vývoj aplikace pomocí analyzovaných knihoven, a ukazuje, jakým způsobem mohou ovlivnit implementací různých částí aplikace.

Porovnání ekosystému ukázalo vhodnost a nevhodnost určitých vlastností frameworků pro vývoj velkých aplikací, a způsoby kterými se tímto novější řešení vyrovnávají. Vývoj zas předvedl výhody velkého ekosystému, hlavně v dostupnosti hotových integrovatelných řešení a množství informací pro řešení problémů.

Klíčová slova: front-end, web, ekosystém, Javascript, React, JSX, Angular, Vue.js

React environment analysis

Abstract

Applications are gradually moving from personal computers to the Web, which allows for ever more complicated and powerful applications, thanks to rise of new web technologies. There are many software libraries and tools being created to help develop these applications, and they are grouping together into digital ecosystems. However, because of their numbers, choosing the right candidate for the job is harder than ever.

The objective of this work is to analyse most popular of these ecosystems, the web technologies related to them, and then use knowledge so gained during the development of a real application.

The theoretical part of the work contains comparison of today's popular frameworks and libraries, and an in-depth look at the most popular of them, React, including some of the libraries designed to extend it. The practical part then describes the development of an application with before analysed libraries, and shows how they affect implementation of various parts of the application.

The comparison of ecosystems showed the suitability, or lack thereof, of certain aspects of frameworks for the development of large applications, and ways through which newer solutions solve this. Development then revealed the advantages of a large ecosystem, especially in availability of ready integrable solutions and amount of information for solving problems.

Keywords: front-end, web, environment, Javascript, React, JSX, Angular, Vue.js

Obsah

1 Úvod ... 12
2 Cíl práce a metodika ... 13
2.1 Cíl práce ... 13
2.2 Metodika ... 13
3 Současný stav poznání řešené problematiky ... 14
3.1 Modely webových aplikací ... 14
3.1.1 Klasický web ... 14
3.1.2 Jednostránkové webové aplikace ... 15
3.1.3 Webová aplikace na serveru ... 16
3.2 Webové technologie ... 17
3.2.1 Prezentační část ... 17
3.2.1.1 Sémantika ... 18
3.2.1.2 Interaktivita ... 19
3.2.2 Aplikační část ... 19
3.2.3 Databázová část ... 21
3.3 Ekosystémy pro webové aplikace ... 21
3.3.1 Architektury MV* ... 22
3.3.1.1 MVC ... 22
3.3.1.2 MVVM ... 24
3.3.1.3 Jiné ... 24
3.3.2 Angular ... 24
3.3.2.1 AngularJS ... 25
3.3.2.2 Angular 2+ ... 25
3.3.3 Vue.js ... 25
3.3.4 React ... 26
3.3.4.1 Vykreslování ... 26
3.3.4.2 Struktura ... 28
3.3.4.3 Životní cyklus komponent ... 29
3.3.4.4 Stav aplikace ... 32
3.3.4.5 Flux ... 32
3.3.4.6 Redux ... 33
3.3.4.7 Routování, rozdělení aplikace ... 33
3.3.4.8 React Native ... 34
3.3.4.9 Licence ... 34
4 Vlastní práce ... 36
4.1 Téma projektu ... 36
4.1.1 WordPress ... 36
4.1.2 WordPress REST API ... 36

4.2 Požadavky ... 36
4.2.1 Části aplikace ... 37
4.2.1.1 Příspěvky ... 37
4.2.1.2 Kategorie ... 37
4.2.1.3 Značky ... 37
4.2.1.4 Statické stránky ... 38
4.2.1.5 Uživatelé ... 38
4.2.1.6 Komentáře ... 38
4.2.2 Stránky aplikace ... 38
4.3 Použité technologie ... 39
4.3.1 react-create-app ... 39
4.3.2 redux ... 40
4.3.2.1 redux-saga ... 40
4.3.3 react-router-dom ... 40
4.3.4 SuperAgent ... 40
4.3.5 styled-components ... 41
4.3.6 prop-types ... 41
4.3.7 moment ... 41
4.4 Implementace ... 41
4.4.1 Moduly ... 41
4.4.1.1 api.js ... 42
4.4.1.2 constants.js ... 42
4.4.1.3 DetailContainer.jsx a ListContainer.jsx ... 42
4.4.1.4 index.js ... 42
4.4.1.5 Page.jsx ... 43
4.4.1.6 reducer.js ... 43
4.4.1.7 sagas.js ... 44
4.4.1.8 Ostatní moduly ... 44
4.4.2 Zastřešující kód ... 44
4.4.3 Adresář _global ... 45
4.4.4 Soubory mimo src ... 46
4.5 Spuštění aplikace ... 46
4.5.1 Proměnné ... 46
4.5.2 Lokálně ... 47
5 Zhodnocení a doporučení ... 48
5.1 Budoucí cíle ... 48
6 Závěr ... 49
7 Seznam použitých zdrojů ... 50

Seznam obrázků

Obrázek 1, Sémantická hlavička 18
Obrázek 2, Hlavička definovaná vzhledem 19
Obrázek 3, Popularita vybraných ekosystémů, počet stažení za týden 22
Obrázek 4, Komunikace v architektuře MVC 23
Obrázek 5, Konverze JSX do React.createElement() 27
Obrázek 6, Vlastnosti Javascriptu v JSX 28
Obrázek 7, Metody životního cyklu 30
Obrázek 8, Tok dat v architektuře Flux 32
Obrázek 9, Importy 43

Seznam tabulek

Tabulka 1, Stránky aplikace 39

Seznam použitých zkratek

API	Application Programming Interface, rozhraní pro programování aplikací
CSS	Cascading Style Sheets, kaskádové styly
CSV	Comma-separated values, hodnoty oddělené čárkami
DOM	Document Object Model – objektový model dokumentu
ES2015	ECMAScript 2015 (také 6), nový standard pro implementaci JavaScriptu
ES5	ECMAScript 5, předchozí verze ES2015
HTML	HyperText Markup Language, značkovací jazyk pro webové stránky
JS	JavaScript, interpretovaný programovací jazyk
JSON	JavaScript Object Notation, datový formát založený na JavaScriptu
JSONP	JSON with Padding, varianta JSONu pro komunikaci mezi doménami
JSX	Syntax pro zápis HTML elementů uvnitř JavaScriptového kódu
MV*	Model-View-cokoliv, označení pro skupinu architektur jako MVC a MVVM
MVC	Model-View-Controller, softwarová architektura
MVP	Model-View-Presenter, softwarová architektura vycházející z MVC
MVVM	Model-View-ViewModel, softwarová architektura vycházející z MVC
SPA	Single Page Application, jednostránková webová aplikace
TS	TypeScript, nadstavba nad JavaScriptem
URL	Uniform Resource Locator, jednotné umístění zdroje, např. webová adresa

1 Úvod

Během několika posledních let se čím dál tím víc aplikací přesouvá na web, a tyto aplikace jsou čím dál tím komplikovanější. Zároveň se tyto aplikace častěji používají z mobilních telefonů a tabletů, a musí tedy plnohodnotně podporovat jak je, tak i klasické počítače. Vývoj aplikací se tudíž velmi zkomplikoval, a napsat plnohodnotnou aplikaci která splňuje všechny požadavky od úplného počátku je prakticky nemožné.

Ve snaze vyřešit tento problém jsou vytvářeny různé frameworky a knihovny, a s příchodem nových webových technologií neustále vznikají nové, ve většině ohledů lepší, a staré zanikají. S každým nástrojem se však pracuje jinak, a proto je nutné se značnou část vývoje neustále učit od znovu.

Z tohoto pohledu je velmi zajímavým nástrojem knihovna React od Facebooku, který byl vytvořen původně pouze pro jejich vlastní použití, ale později uvolněn pro veřejnost. Momentálně je základem všech jejich webových, a díky Reactu Native, i mobilních aplikaci. Díky tomu je krajně nepravděpodobné, že bude v blízké budoucnosti nahrazen.

React se těší enormní popularitě, jak v porovnání se staršími, tak i novějšími řešeními, a vznikl okolo něj obrovský digitální ekosystém – tedy knihovny určené pro spolupráci s ním, nástroje, informační zdroje, a další. Naučení se práce s Reactem je tedy díky tomu nejen snazší, ale získané znalosti také pravděpodobně zůstanou déle aktuální a použitelné.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této práce je React a jeho digitální ekosystém analyzovat. To znamená nejprve se seznámit s různými architekturami používanými pro webové aplikace, a skrz ně poté React porovnat s ostatními dnes populárními ekosystémy, včetně toho před vznikem Reactu nejpopulárnějšího, což je Angular.js. Dále podrobněji prozkoumat jeho fungování a používání, a to nejen Reactu samotného, ale i několika populárních částí jeho ekosystému.

Následně získané znalosti o Reactu a jeho ekosystému aplikovat v praktické části, která bude popisovat vývoj webové aplikace, od definování požadavků na funkcionalitu, až po vysvětlení výsledné implementace.

2.2 Metodika

Metodikou použitou pro dosažení cíle práce bude zejména analýza odborných informačních zdrojů, které však v tomto případě nebudou úplně standardní. Vzhledem k častým a závažným změnám v oblasti vývoje front-endových aplikací zpravidla nebývá příliš času pro vznik mnoha knižních zdrojů, a neustálý posun dopředu je rychle zastarává.

Hlavními zdroji informací budou tedy zejména oficiální dokumentace zkoumaných knihoven a frameworků, a články od vývojářů používající tyto ekosystémy.

3 Současný stav poznání řešené problematiky

Tato část práce má za cíl popsat strukturu webových aplikací, způsoby kterými je lze implementovat, a pro to relevantní technologie.

3.1 Modely webových aplikací

3.1.1 Klasický web

Typická webová aplikace využívá tříúrovňový model: u uživatele se nachází prohlížeč (prezentační část), který zobrazuje stránky posílané mu webovým serverem (aplikační část), který se zpravidla nachází v datovém centru. Ten vytváří webové stránky, a naplňuje je daty získanými z databáze (databázová část), která se často nachází ve stejné lokaci, někdy i na stejném počítači. Velmi jednoduché weby s neměnnými daty, které pouze zobrazují statické informace a nemají databázi, samozřejmě tuto třetí úroveň nemají.

Jednotlivé stránky posílané serverem do prohlížeče jsou na sobě nezávislé, a každá obsahuje všechny informace nutné pro její zobrazení, ať již přímo (text), či formou odkazu (např. obrázky). Tyto informace se nachází ve formátu, kterému rozumí prohlížeč, a není tedy nijak specifický pro danou aplikaci. Díky tomu lze webové stránky zobrazovat na všech zařízeních, od stolních počítačů, přes telefony, až po čtečky pro nevidomé. Stačí aby prohlížeč i webový server implementovaly stejné webové standardy.

Přenášení celé stránky v univerzálním formátu má ovšem i nevýhody. Formát jednotný pro všechny webové stránky, a srozumitelný pro všechna zařízení, je „upovídanější" než formát specifický pro specifickou aplikaci, a tudíž náročnější na přenos.

Další nevýhodou je, že části stránky které se opakují, například hlavička a patička, a data která již prohlížeč dostal v minulosti, se posílají znovu při každém načtení další stránky, což samozřejmě znamená větší tok dat a pomalejší interakci. Informace poskytované formou odkazu znovu posílají pouze onen odkaz, a samotný například obrázek je tedy možné stáhnout pouze jednou a poté držet lokálně v paměti, ale pro informace vložené přímo do stránky toto není možné.

Tato nevýhoda se projevuje zpravidla pro dynamický obsah, který se mění málo a často – diskuzní fórum či sociální síť, kde se na stránce nachází desítky zpráv či příspěvků, a nový přibyde každých pár minut. Stránka je sice ve většině případů z 99% stejná, ale stahuje se vždy celá znovu.

3.1.2 Jednostránkové webové aplikace

Alternativou je část aplikační části přesunout k uživateli. Tato pak komunikuje s databází v datovém centru, a po síti se data přenáší ve formátu specifickém pro tuto jednu aplikaci. Jedná se do určité míry o návrat k dřívějšímu modelu klient/server, ale na základě webových technologií a standardů.

Místo toho aby uživatel stahoval, instaloval, a spouštěl samostatnou aplikaci, stále používá univerzální webový prohlížeč. Při otevření webové stránky aplikace dostane prohlížeč víceméně prázdnou stránku s referencí na část aplikační logiky, která zprostředkovává komunikaci se vzdálenou aplikační logikou a skrz ní s databází, a zobrazování získaných dat uživateli. Tato stahovaná část aplikační logiky je vždy stejná, a server který jí posílá tedy nemusí být „chytrý" – lze ji tedy mít uloženou v mezipaměti, a to jak serveru, tak po prvním stažení i u uživatele.

Očividnou výhodou je, že aplikace sama sobě a svým datům rozumí. Když se uživatel přesune do jiné části aplikace, což by v klasickém modelu byla jiná stránka, tak se nestahuje vše znova, ale pouze obsah této specifické části. Pro změnu dynamického obsahu se stahuje pouze rozdíl.

Aplikace ví co je příspěvek, chápe jeho datovou strukturu, a umí ho do již existující stránky vložit. Po prvotním načtení celé diskuze tedy dokáže serveru říct „poslední příspěvek mám číslo 37", a server neposílá celou stránku, ani všechny příspěvky, ale pouze ty nové (nebo případně odpoví že nic nového není). Aplikace je také schopná s již získanými daty manipulovat, například je řadit či filtrovat, bez další komunikace se serverem.

Tento přístup má ovšem svoje nevýhody. Jednou z nich je přidaná komplexita aplikace pro vývojáře – aplikační logika se nyní nutně dělí na uživatelskou část (front-end) a serverovou část (back-end).

Dalšími nevýhodami je nutnost podpory více technologií, a vyšší výpočtová náročnost, na zařízení uživatele. Toto může být problémem nejen pro méně rozšířená zařízení jako čtečky pro nevidomé, ale také pro archivaci a vyhledávače – pokud služba či zařízení nepodporují spouštění kódu, jediné co se archivuje či zobrazí bude zpravidla oznámení o nekompatibilitě.

Ačkoliv například Google při indexování webů již několik let nejen čte samotné odpovědi od serverů jako text, ale také interpretuje a spouští aplikační logiku (1), používá k tomu starou verzi svého prohlížeče Chrome (41, z března 2015), a to s některou

funkcionalitou vypnutou (například perzistence skrz Cookies a LocalStorage) (2). Aplikace která svůj obsah načítá postupně také nemusí být správně indexována, vzhledem k omezení času a výkonu.

Nevýhoda existuje i pro uživatele jejichž zařízení všechny technologie podporuje – nutnost prvního stažení aplikační logiky než lze aplikaci používat. Na pomalém připojení se může každá stránka klasického webu stahovat několik vteřin, zatímco ve webové aplikaci jsou sice některé akce provedeny ihned (řazení již stažených dat) a ostatní jsou několikrát rychlejší, ale prvotní načtení aplikace může trvat desítky vteřin, které uživatel nemusí být ochotný čekat, či si dokonce může myslet že stránka nefunguje a opustit jí před jejích načtením.

3.1.3 **Webová aplikace na serveru**

Řešením mnoha nevýhod je kombinace obou metod, tedy používat jednostránkovou aplikaci na zařízeních které jí podporují, a statické webové stránky pro služby a zařízení která ne. Způsoby jak tohoto dosáhnout je Server-side Rendering (vykreslování na serveru).

Při prvním dotazu je na uživatelovo zařízení poslána plná webová stránka, obsahující vše nutné k jejímu zobrazení. Navíc k této klasické stránce je přidána reference na aplikační logiku. Pokud zařízení tuto logiku odmítne či nedokáže spustit, při interakci se stránkou (odkazy, formuláře) se ze serveru posílá celá kompletní stránka, a chování je stejné jako u klasického webu. (3)

Pokud se však aplikační logika úspěšně spustí, tak stránku „oživí“ a stane se jednostránkovou webovou aplikací, interakce je řešena lokálně, a ze serveru se stahují pouze nutná data či změny od poslední komunikace. Toto oživování je samozřejmě nutné v aplikační logice implementovat, ale jedná se víceméně pouze o inicializaci aplikace – první vykreslení není do prázdné stránky a bez dat, ale do již existující stránky a s daty. Veškeré další chování aplikace po oživení je stejné. (4)

Rozdílem oproti samotné jednostránkové webové aplikaci na zařízení které ji podporuje je, že data pro požadovanou stránku jsou již stažena. Místo toho aby se nejprve stahovala a interpretovala aplikační logika, vždy stejná, a ta pak stahovala specifická data (dvě zpáteční cesty na server), tak již první odpověď serveru obsahuje jak aplikační logiku, tak i data. Informace se tedy uživateli zobrazí již při stažení statické stránky (stejná rychlost jako klasický web), po stažením logiky aplikace stane interaktivní, a ony informace již má.

Takovou aplikaci již ale nelze uživateli posílat z mezipaměti. Server nyní musí být chytrý, protože vykresluje aplikaci zvlášť pro každého uživatele, a to s jeho daty a persistencí, stejně jako klasický webový server.

Celou aplikaci je tedy možné používat na starých zařízeních, včetně částí které vyžadují přihlašování, a zobrazovaná data jsou vždy aktuální. Na moderních zařízeních není nikdy nutné stahovat další data zvlášť.

Hlavní nevýhodou je opětovné zvýšení náročnosti vývoje aplikace. Na klientu aplikace běží asynchronně: když chce uživatel zobrazit jinou část aplikace, a ta nemá stažená potřebná data, tak na pozadí pošle serveru žádost, zobrazí zástupná data či uživateli nějakým způsobem řekne že načítá, a čeká na data či na další uživatelovu interakci. Až data ze serveru přijdou, zobrazí je uživateli. Pokud uživatel akci zruší, tak aplikace zruší žádost o data.

Na serveru je však aplikace synchronní: přijde žádost od uživatele, server získá data z databáze po místní síti či z jiného procesu na stejném počítači mnohem rychleji, takže na ně čeká bez posílaní čehokoliv uživateli, a pak výsledek pošle uživateli najednou.

V obou případech se data získávají jinak a odjinud, a aplikace tedy musí umět oboje. Implementace Server-side Renderingu tedy znamená změny napříč celou aplikací. Jedná se tedy o kompletní, ale také nejnáročnější řešení, které se pro některé aplikace nutně nevyplatí.

3.2 Webové technologie

3.2.1 Prezentační část

Je stejná pro statické weby i aplikace - webový prohlížeč od aplikační části (ať již ze serveru či lokálně) dostává informace v univerzálním formátu, který specifikuje jak strukturu, tak obsah. Tímto formátem je na webu HTML (HyperText Markup Language), což je značkovací jazyk, který byl vytvořen v roce 1990 spolu se samotným protokolem HTTP (HyperText Transfer Protocol).

HTML dokument se skládá z vrstvených značek, které specifikují jak v nich obsažené značky, text, a další reprezentovat, a do určité míry také jejich sémantický význam. Prohlížeč ví jak dokument zobrazit, rozumí interaktivním prvkům jako jsou odkazy a tlačítka, a text obohacuje vizuálně pomocí stylů a obrázků.

3.2.1.1 Sémantika

Nechápe však ve většině případů, alespoň bez použití vcelku nově vzniklých (a volitelných) rozšíření jako RDFa, co obsah stránky opravdu znamená – ví že tento text je nadpis třetího řádu a toto je obrázek, ale již nechápe že nadpis je jméno autora článku, a obrázek jeho fotka. Některé sémantické značky existují, například *<article>* obaluje článek, respektive libovolnou samostatnou a uzavřenou jednotku použitelnou mimo tuto specifickou stránku, s vlastní strukturou (hlavička, patička, nadpisy) a vlastními metadaty (například autor).

Vzhledem k tomu, že všechny značky jsou pevně definované standardem, samozřejmě neexistuje značka pro každou možnou část dokumentu. Nejpoužívanější značky *<div>* a *<span>* pouze značí blokový a řádkový element, respektive, a nemají žádný sémantický význam – reprezentují pouze svůj obsah.

Této sémantiky využívají vyhledávače a podobné, ale samotné prohlížeče v této době ne. Rozdíly mezi stránkou vytvořenou sémanticky správně, a stránkou složenou pouze z „divů“, tedy při běžném používání pro většinu uživatelů nejsou. Následující dva kusy HTML (Obrázek 1 a Obrázek 2) se tedy, s odpovídajícími styly, zobrazí v typickém prohlížeči stejně.

První z nich sémanticky definuje hlavičku, nadpis, a obrázek včetně alternativního textu. Zatímco druhý definuje pouze pár nespecifických elementů a jejich vzhled – který například čtečka nemá jak interpretovat.

Obrázek 1, Sémantická hlavička

```
<body>
  <header>
    <h1>
      <img src="logo.png" alt="Logo firmy" />
      Název firmy
    </h1>
  </header>
…
```

Zdroj: Autor

Obrázek 2, Hlavička definovaná vzhledem

```
<body>
  <div>
    <div class="heading">
      <div style="background: url(logo.png);
                  height: 50px; width: 50px;" />
      Název firmy
    </div>
  </div>
…
```

Zdroj: Autor

Sémantice tudíž často není přikládána zdaleka taková váha jako vzhledu, což pak znamená že se na ni nemohou prohlížeče spoléhat. Hlavní tažnou silou pro lepší implementaci jsou tedy pořadí stránek ve vyhledávačích a podobné.

3.2.1.2 Interaktivita

HTML dokument slouží jako prezentační část, a na klasických webech interakce uživatele (na úrovni odeslání formuláře, ne pouhé psaní do textového pole) způsobuje komunikaci s webovým serverem. Odpovědí od serveru je pak další, samostatný HTML dokument.

HTML dokument v sobě také může obsahovat, a ve valné většině případů obsahuje, reference na externí zdroje dat. Na tyto externí zdroje je pak možné odkazovat z různých dokumentů, a prohlížeč je může uchovávat v mezipaměti – pak není nutné je stahovat pokaždé znova. Toto mohou být multimédia jako obrázky, styly pro úpravu vizuální reprezentace různých elementů, a také skripty sloužící jako aplikační logika – ta pak může se strukturou dokumentu manipulovat, reagovat na uživatelskou interakci, a tak dále.

3.2.2 **Aplikační část**

Aplikační logika na vzdáleném serveru, tedy back-end, se vyskytuje jak u webových stránek, tak aplikací. V případě klasických stránek server posílá HTML dokument, zatímco u aplikace posílá data v libovolném formátu, kterému rozumí logika na místním počítači, teda spouštěná v prohlížeči. V případě hybridního řešení, tedy webová aplikace včetně vykreslování na serveru, jsou obě části nezávislé – aplikační logika stránky, ať již v prohlížeči nebo při vykreslování na serveru, komunikuje skrz stejné rozhraní, po místní

síti nebo vzdáleně – back-end je v obou případech použit stejný, a to předchází nutnosti ho vytvářet dvakrát.

Back-end jako takový může používat libovolný server, programovací jazyk, nebo cokoliv jiného, ale jeho implementace není předmětem této práce. Tento back-end poskytuje webové API (Application Programming Interface), tedy rozhraní pro komunikaci front-endu s back-endem po protokolu HTTP.

Front-end, tedy kód spouštěný v prohlížeči, byl donedávna vždy a nutně psaný v programovacím jazyku Javascript. Jeho první verze vznikla v roce 1995, se syntaxí vcelku podobnou Javě (oba jazyky mají syntaxi podobnou C), se kterou však nemá nic společného – název Javascript byl zvolen spíše jako marketingový tah. Javascript se nekompiluje, v prohlížeči je interpretován přímo ze zdrojového kódu. Jedná se o objektově orientovaný jazyk, který nemá pevně stanovené (statické) typy. (5)

Během roku 2017 postupně začaly nové verze všech velkých prohlížečů (Chromium, Firefox, Edge, a Safari, včetně mobilních verzí) podporovat WebAssembly, což je formát strojového kódu, do kterého lze kompilovat ostatní jak Javascript, tak i ostatní programovací jazyky. Výsledek lze pak spouštět v podporujícím prohlížeči na libovolné platformě.

Díky tomu, že je kód kompilovaný, je WebAssembly výkonnější než interpretovaný Javascript. Možnost psát kód v preferovaném programovacím jazyku je další výhodou. Většina vývoje pro web samozřejmě stále probíhá v Javascriptu, a pro většinu projektů není vyšší výkon z použití WebAssembly zajímavý, a větší komplexita vývoje a menší kompatibilita se staršími prohlížeči za něj nestojí. (6)

Interpretovaný Javascript a kompilovaný WebAssembly je možné používat zároveň – lze tedy aplikaci napsat nejprve v Javascriptu, a pokud je potřeba její běh urychlit, je možné přepsat a kompilovat pouze některé kritické části. Tyto WebAssembly moduly lze pak volat jako funkce z přímo z Javascriptu.

Co se týče kompatibility interpretovaného Javascriptu, verze ES (ECMAScript) 5 z roku 2009 je podporována prakticky ve všech dnes používaných prohlížečích, včetně Internet Exploreru 10 a 11. Kód psaný podle novějšího standardu ES 6 (známý jako ES 2015, podle roku vzniku) je možné transpilovat (tedy přeložit z jednoho zdrojového kódu do jiného) do kódu platném v ES 5, a tudíž je ho možné spouštět v prohlížečích které ho nepodporují přímo. (7) (8)

Server či část serveru starajícího se o Server-side Rendering, pokud existuje, spouští stejný kód jako prohlížeč, a tedy zpravidla Javascript, nezávisle na implementaci back-endu.

Ten je možné implementovat také v Javascriptu, pomocí Node.js. Back-end a front-end jsou ale i tak nezávislé. Výhodou je tedy hlavně použití stejného jazyku pro obě části, což je asi nejvíce užitečné pro jednotlivé vývojáře či malé týmy. Populární stránky zpravidla pro back-end používají jiný jazyk. (9)

3.2.3 Databázová část

Samotná databáze žije na vzdáleném serveru, a není tedy pro front-end a tuto práci relevantní. Komunikace s ní, respektive se vzdálenou části aplikační vrstvy která k ní přistupuje, a formáty pro tuto komunikaci použité, ovšem ano. Vzhledem k tomu, že aplikační logika, jak na front-endu, tak na back-endu, je většinou vyvíjená stejnou entitou, je samozřejmě možné používat jakýkoliv formát.

V praxi je v dnešní době již nejčastěji používán datový formát JSON (JavaScript Object Notation), který, jak název napovídá, vychází ze syntaxe používané pro vytváření objektů v Javascriptu. Je však určený pro použití nezávisle na programovacím jazyku, má svojí vlastní specifikaci, a se syntaxí Javascriptu není úplně shodný. Většina rozdílů se projevuje jako vyšší striktnost JSONu: například nutnost používat uvozovky pro klíče, a pouze dvojité uvozovky pro textové řetězce, zatímco v Javascriptu lze klíče (krom rezervovaných slov) psát bez uvozovek, a řetězce uvozovat jednoduchými uvozovkami (apostrofy). Známým omylem proto je, že JSON je podmnožinou Javascriptové notace. Existují ale určité výjimky, kde validní JSON není validním Javascriptem (10).

Původně bylo používané XML, které ostatně dalo název jak samotnému AJAXu (Asynchronous Javascript And XML), tak také Javascriptovému objektu dlouho používanému pro komunikaci s back-endem, XMLHttpRequest. Další možností je posílat HTML fragmenty, tedy jednotlivé elementy HTML dokumentu, které je pak možné beze změny vkládat do existujícího dokumentu, nebo používat jako šablonu pro data získaná v jiném formátu. (11)

3.3 Ekosystémy pro webové aplikace

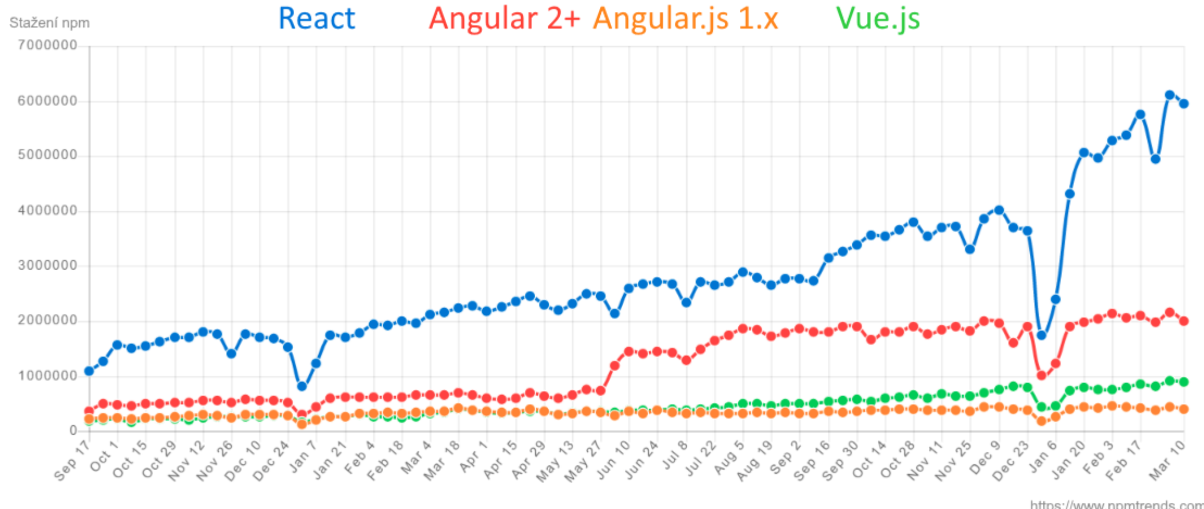
S postupným rozvojem webových aplikací a jejich složitosti vznikla potřeba pro knihovny a frameworky, bez kterých by byl vývoj příliš náročný. V mnoha případech se tomu tak stalo přímo při vývoji oněch aplikací, včetně Reactu vytvořenému pro Facebook.

Okolo těchto knihoven a frameworků pak vznikla různá rozšíření, navazující knihovny, a zdroje informací. Ekosystémem je zde myšlen tento celek.

Tato část práce má za cíl stručně popsat nejpopulárnější frameworky a knihovny pro vývoj *front-endových* webových aplikací. Nespadají sem tedy *back-endové* technologie jako databáze, webové servery, ani jazyky jako *PHP* a *ASP.NET*.

V následujících kapitolách jsou popsány čtyři, momentálně nejpopulárnější: *Angular* ve dvou variantách, které existují zároveň, a v mnohém se liší, *Vue.js*, a finálně *React*. Existují samozřejmě další, ale v porovnání s těmito jsou buď zastaralé, nebo málo rozšířené.

Obrázek 3, Popularita vybraných ekosystémů, počet stažení za týden



Zdroj: npm trends (12)

I vzájemná popularita těchto čtyř toto dobře ilustruje – jak je vidět na Obrázek 3, nejpopulárnější React je cca šestkrát víc stahovaný než třetí nejpopulárnější Vue.js.

3.3.1 **Architektury MV***

Pro vysvětlení rozdílů mezi těmito ekosystémy je však nejprve nutné popsat některé možné architektury pro front-end. *MV** je názvem pro skupinu návrhových vzorů architektury. Zde popsané vzory jsou používané zkoumanými ekosystémy, a rozdíly těchto vzorů tedy pomáhají ilustrovat různé přístupy jednotlivých ekosystémů.

3.3.1.1 MVC

MVC (Model-View-Controller) je historicky prvním MV* návrhovým vzorem; vznikl na konci sedmdesátých let, a ostatní MV* vzory z něj tedy vychází. Verze používaná dnes

se v některých aspektech pochopitelně liší. Hlavním principem je oddělení jednotlivých částí aplikace pro vyšší přehlednost kódu. Aplikaci dělí mezi správu dat (*Model*), uživatelské rozhraní (*View*), a logiku a interakci mezi předchozími (*Controller*). (13, s. 111)

Ačkoliv toto rozdělení značně připomíná dříve zmíněný tříúrovňový model, aplikací je zde myšlen kód spouštěný na uživatelově zařízení, tedy část aplikační vrstvy. Například Model tedy spravuje data v prohlížeči, nezávisle na jejich reprezentaci v databázi na vzdáleném serveru, existuje-li vůbec taková databáze.

Modely uchovávají veškerá data potřebná pro aplikaci, bez ohledu na to jak se budou zobrazovat či zpracovávat. Také mohou data validovat, a vyvolávat události při jejich změně – na tyto události pak může poslouchat libovolné množství views. (13, s. 113)

Views jsou vizuální či textovou reprezentací dat. Mohou reprezentovat model, jeho část, či kombinaci několika. Zobrazují současný stav dat, a zpravidla se aktualizují při jejich změně. Také mohou obsahovat rozhraní pro změnu dat, ale nemění model samotný; informaci o požadované změně předávají controllerům. (13, s. 115)

Obrázek 4, Komunikace v architektuře MVC

Zdroj: Model-View-Controller (MVC) in iOS: A Modern Approach (14)

Controllery zpracovávají vstupy od uživatelů a mění na jejich základě data v modelech. Také mohou na základě změn dat v modelech aktualizovat views, pokud views samy neposlouchají na události o změnách. Stojí tedy mezi předchozími částmi, a zprostředkovávají mezi nimi komunikaci (viz Obrázek 4, Komunikace v architektuře MVC). (13, s. 117)

Díky tomuto rozdělení je zdrojový kód aplikace nejen přehlednější, ale rozdělení také zjednodušuje úpravy – například při chtěné změně vykreslování není nutné měnit modely ani controllery, pouze views. Rozdělení dále usnadňuje testování jednotlivých částí, a snižuje redundanci kódu, znovupoužitím stejných částí pro jiné situace. (13, s. 120)

3.3.1.2 MVVM

MVVM (Model-View-ViewModel) vychází z MVC, respektive z *MVP* (Model-View-Presenter) které vychází v MVC, a používá tedy podobné prostředky pro dosažení víceméně stejných cílů. Modely jsou de facto stejné, s tím že validaci obsahují častěji než v MVC, a formát dat bývá generický, a ne specifický pro potřeby zobrazení. Views mají o něco aktivnější roli, protože nejsou ovládané controllerem, ale samy obsahují část aplikační logiky. (13, s. 125)

Hlavním rozdílem je samozřejmě existence *ViewModelu*, a žádný controller. ViewModel se dá považovat za náhradu controlleru, co se týče interakcí mezi modely a views. Důležitou rolí je převod dat z formátu ve kterém se nacházejí v modelu do formátů chtěných ve views. Na příkladu, v modelu může být datum uloženo jako počet sekund od počátku unixové epochy, a pro potřeby různých views ho viewmodel konvertuje na lidsky čitelné formáty ve správné časové zóně. (13, s. 129)

3.3.1.3 Jiné

Vzhledem k rozšířenosti architektur napříč mnoha programovacími jazyky existuje mnoho dalších variant, a vzhledem k tomu že implementace MVC a podobných se napříč frameworky a knihovnami liší, by šlo i některé realizace považovat za varianty na původní architekturu.

Také architektury *Flux* a *Redux* jsou odlišné, a podrobněji rozebrané ve vlastních sekcích, v kapitole o Reactu.

3.3.2 **Angular**

Angular se dále dělí na dvě varianty, které se liší v mnoha aspektech, včetně přístupu a architektury. Tyto varianty jsou podporované a vyvíjené zároveň.

Obě varianty jsou spravovány Googlem a open-source, dostupné pod MIT licencí (15) (16), a tudíž vyvíjeny společně s nezávislými vývojáři. Jedná se o plnohodnotné frameworky, obsahující všechny potřebné komponenty pro vývoj front-endové aplikace.

3.3.2.1 AngularJS

AngularJS (také známý jako Angular.js a Angular 1.x) je původní varianta, založená na architektuře MVC, ale podporující i MVVM a další varianty. AngularJS vznikl v roce 2010 (17), a brzy se stal velmi populárním díky oboustrannému *data-bindingu* (tedy pokud se hodnota změní v modelu, změní se automaticky i ve view, a obráceně), MVC architektuře, a tomu že byl komprehensivním řešením. (18)

Views aplikace jsou HTML stránky, rozšířené o *direktivy* – speciální atributy pro HTML elementy, řešící data-binding k modelu, vázání na controllery, opakování elementů, a další funkcionalitu která není dostupná v HTML samotném. Controllery jsou psané v Javascriptu, a modely jsou tvořeny proměnnými tamtéž. (19)

3.3.2.2 Angular 2+

Angular 2+ (také pouze Angular, a Angular 5 či brzy Angular 6, dle poslední verze) je kompletně přepsaná, nová varianta Angularu. Vzhledem k problémům s určitými aspekty AngularuJS (20), hlavně pro velké aplikace, se vývojáři rozhodli novou verzi Angular 2 udělat od začátku. Je nekompatibilní s první, a pro přechod z verze 1.x na verzi 2 je tedy nutné celou aplikaci přepsat.

Angular 2+ již není založen na architektuře MVC, ale na vlastní, komponentové. V některých aspektech je stejný jako AngularJS, například použitím HTML souborů s direktivami jako views a oboustranným data-bindingem. V některých je jiný, například logika aplikace se nenachází v controllerech, ale v komponentách, a pro její psaní se místo JavaScriptu používá TypeScript, který se následně kompiluje do Javascriptu pro spouštění v prohlížeči. (21)

3.3.3 **Vue.js**

Vue.js (také pouze Vue, francouzské slovo pro View) je framework založený na architektuře MVVM. Byl vytvořen jako lehčí alternativa k AngularuJS, a je méně rozsáhlý, ale lépe rozšiřitelný dalšími knihovnami. Není vyvíjen žádnou velkou společností (jako

Google u Angularu a Facebook u Reactu), ale pouze nezávislými vývojáři, a je open-source pod MIT licencí (22).

Stejně jako AngularJS pro views v základu používá HTML s přidanými direktivami, a na první pohled je mu velmi podobný, ale celkový přístup je značně jiný: data-binding (a tudíž tok dat) je jednosměrný, což je sice náročnější na vývoj, ale ve velkých a komplikovaných aplikacích přehlednější. Místo controllerů logiku aplikace dělí mezi komponenty, stejně jako nový Angular. (23)

Také je v některých aspektech podobný Reactu, například umožňuje pro views používat JavaScriptové funkce (včetně JSX) místo HTML. Je tedy možné vybrat přístup vhodnější pro danou situaci, a v aplikaci oba přístupy kombinovat. Views v JavaScriptu jsou více rozebrané právě v sekci o Reactu.

3.3.4 **React**

React (také ReactJS nebo React.js) je knihovna určená pro vytváření webových aplikací. Byl vytvořen v době velké popularity AngularuJS, se značně jiným přístupem v mnoha aspektech. Data-binding a tok dat je jednosměrný. Místo rozšiřování HTML a dělení aplikace mezi views v něm a logiku v JavaScriptu se celá aplikace nachází v JavaScriptu, dělená na komponenty.

React samotný se stará pouze o vykreslování a interakci s uživatelským rozhraním; z popsaných MV* architektur žádnou nerealizuje celou, používá se jen jako „V" ve spojení s dalšími knihovnami.

Důsledně okolo něj vznikl celý ekosystém knihoven a rozšíření, které řeší ostatní části aplikace. Je možné používat pouze React bez dalších knihoven, nebo v kombinaci s libovolným počtem z nich, podle potřeby. Také je možné vytvářet si svoje rozšíření a knihovny, pokud ty již existující nesplňují požadavky.

Neexistuje tedy jen jedna, pevně daná a jediná správná cesta, jak React používat, a výběrem Reactu rozhodování zdaleka nekončí. Na jednu stranu je tedy možné použít přesně takovou architekturu a kombinaci knihoven, jaká je pro daný projekt a tým nejlepší; na druhou stranu je samozřejmě možné zvolit špatně.

3.3.4.1 Vykreslování

Jednotlivé části stránky (HTML elementy a React komponenty) jsou v Reactu reprezentovány jako Javascriptové objekty uvnitř kódu, ne v HTML. Tyto objekty je možné

reprezentovat pomocí *JSX*, což je rozšíření Javascriptu, se syntaxí podobnou HTML, či spíše XML – například všechny značky musí být uzavřené, ať již *<p></p>* nebo *
* (24) (25).

Není jej však nutné využít, a před jeho použitím Reactem je JSX přeloženo do volání Javascriptové funkce, a to *React.createElement()* – příklad konverze viz Obrázek 5. (26)

Obrázek 5, Konverze JSX do React.createElement()

```
1  function render() {
2    return (
3      <p>
4        Hello
5        <em>
6          {this.props.name}
7        </em>
8      </p>);
9  }
10
```

```
1  "use strict";
2
3  function render() {
4    return React.createElement(
5      "p",
6      null,
7      "Hello",
8      React.createElement(
9        "em",
10       null,
11       this.props.name
12     )
13   );
14 }
```

Zdroj: Babel REPL (27)

Kód je strukturován do takzvaných komponent, které jsou v kódu definovány buď jako třída, ať již pomocí Javascriptových tříd a nebo skrz volání funkce *React.createClass()*, a nebo jsou definovány jako funkce. Komponenta reprezentovaná funkcí (funkční komponenta) dostává vstupní data od rodiče, nazývané *props* (zkrácená verze properties), a jejím výstupem je Reactový objekt (28). Komponenty reprezentované třídou mohou používat také funkce volané při určitých událostech (načtení komponenty, změna props, atp.) a vnitřní stav komponenty (29) - analogem výstupu funkční komponenty je výstup metody *render()*.

Ať již je komponenta funkční nebo ne, výstup určuje Javascriptová funkce, ve které je tudíž možné používat všechny vlastnosti Javascriptu. Proměnné, smyčky, podmínky, a i volání jiných funkcí, lze také používat přímo uvnitř JSX (viz Obrázek 6).

Obrázek 6, Vlastnosti Javascriptu v JSX

```
1  function render() {
2    let users = this.props.users || [];
3
4    return (
5      <div>
6        <Header>Users</Header>
7        {users.length > 0 ?
8          <ul>
9            {users.map(user =>
10             this.renderUser(user)
11           )}
12         </ul>
13       :
14         <span>There are no users.</span>
15       }
16     </div>
17   )
18 }
19
```

```
1  "use strict";
2
3  function render() {
4    var _this = this;
5
6    var users = this.props.users || [];
7
8    return React.createElement(
9      "div",
10     null,
11     React.createElement(
12       Header,
13       null,
14       "Users"
15     ),
16     users.length > 0 ? React.createElement(
17       "ul",
18       null,
19       users.map(function (user) {
20         return _this.renderUser(user);
21       })
22     ) : React.createElement(
23       "span",
24       null,
25       "There are no users."
26     )
27   );
28 }
```

Zdroj: Babel REPL (27)

Změny ve stavu aplikace nejsou promítány ihned do HTML *Document Object Modelu* (dále DOM), protože jeho manipulace je relativně pomalá a výkonnostně náročná, zejména kvůli překreslování stránky a opětovné aplikaci kaskádových stylů. React si udržuje jeho kopii, takzvaný *Virtual DOM*, který se nestará o styly, vykreslování, a další, a práce s ním je tudíž rychlejší. Porovnáním staré a nové verze Virtual DOMu poté vytvoří záplatu, kterou aplikuje na HTML DOM (30). Tento fakt umožňuje vývojáři používat deklarativní styl programování, tedy říkat co se má změnit, a React vymyslí jak.

3.3.4.2 Struktura

Reactová aplikace se inicializuje tak, že jedna z komponent je předána funkci *ReactDOM.render()* spolu s již existujícím HTML elementem, jehož obsah je nahrazen výstupem oné komponenty. Tato kořenová komponenta zpravidla obsahuje celou aplikaci, a ostatní komponenty jsou jejími potomky. Nejsou ale nutně jejími potomky v DOM – využitím *portálů* lze komponentu vložit mimo kořenový HTML element, ačkoliv pro React je stále potomkem kořenové komponenty, a předávání dat a událostí se chová odpovídajícím

způsobem (31). Toto je vhodné zejména pro dialogová okna a podobné části uživatelského rozhraní, které se mají objevit "nad" ostatními.

Také je ovšem možné pomocí Reactu vykreslovat jen část stránky, a takovýchto komponent se může na stránce nacházet libovolné množství – z pohledu Reactu se jedná o separátní aplikace (32). Možné použití je například při přechodu z jiného ekosystému na React – lze postupně nahrazovat jednotlivé části původní aplikace, a není tudíž nutné přejít najednou.

Každá komponenta má přístup k *props* od svého rodiče (samozřejmě kromě kořenové, která rodiče nemá), ke svému vlastnímu *stavu*, a ke *kontextu* od všech předků. Props jsou jednosměrné: určuje je rodičovská komponenta, a její dětskou komponentou jsou neměnné. Pomocí props se komponentě předají její vstupní hodnoty (v případě funkční komponenty doslova), a ty se uživatelskou interakcí nemění. Pro předání dat z potomka do rodiče se potomkovi pomocí props předá funkce, a použije se zpětného volání. Také lze nastavit požadovaný typ props, a tím vstup částečně validovat.

Na druhou stranu stav je uzavřen pouze uvnitř této komponenty, a lze ho měnit – například hodnota textového pole může být uložena ve stavu.

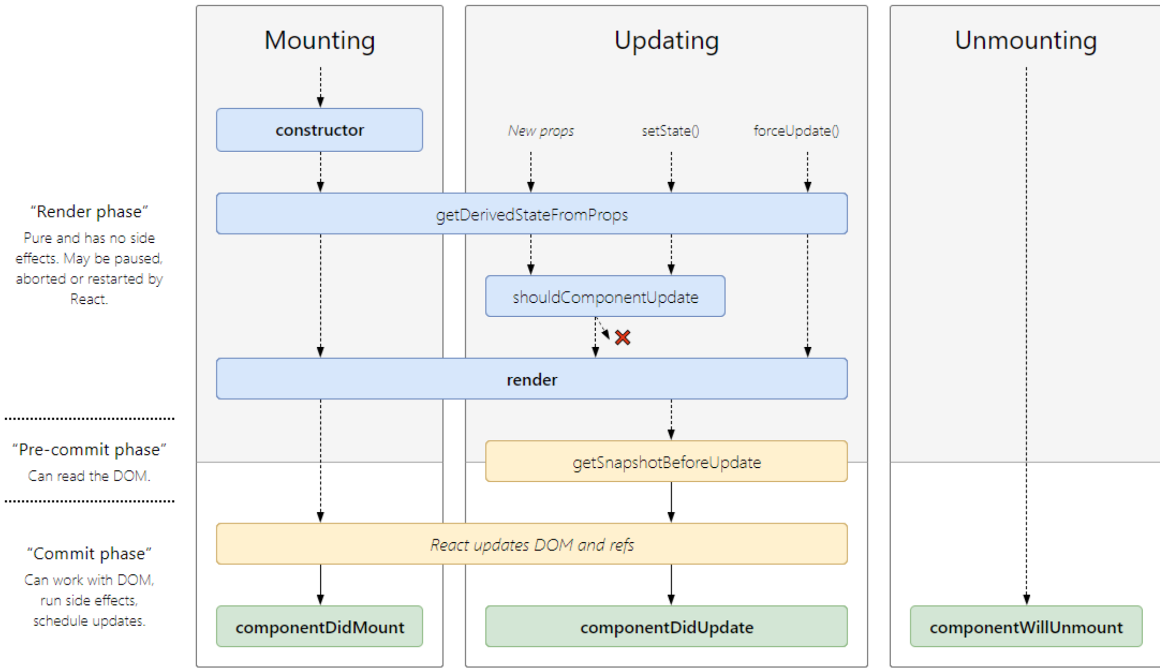
Kontext je podobný props jednosměrností od předka k potomkům, včetně zpětných volání předané funkce. Liší se však používáním kontextu v potomcích. Potomek, který chce přístup ke kontextu jednoho ze svých předků, se musí přihlásit jako odběratel.

Změny v kontextu se pak v potomcích projevují nezávisle na přímých rodičích těchto komponent, což může mít dopad jak na výkon, tak i na použitelnost komponent v různých prostředích, například při testování. Používat kontext se tedy doporučuje pouze střídmě (33).

3.3.4.3 Životní cyklus komponent

Dříve zmíněné komponenty tvořené třídou mají několik metod, ve kterých lze pouštět kód před či po určitých událostech, zatímco ty tvořené funkcí je pro zjednodušení a zrychlení nemají. Vizualizace viz Obrázek 7. Pokud je komponenta vykreslována na serveru, spustí se pouze *contructor()* a poté *render()*.

Obrázek 7, Metody životního cyklu



Zdroj: React lifecycle methods diagram (34)

3.3.4.3.1 Vznik

Na klientu se při vytváření komponenty nejprve vygeneruje prvotní stav – zde je rozdíl mezi komponentami tvořenými pomocí Javascriptových tříd, které používají *constructor()*, a mezi těmi tvořenými skrz *createClass()*, které používají *getInitialState()* – je nutné použít tu správnou, ale pak mezi nimi rozdíl není.

Poté proběhne *getDerivedStateFromProps()*, což je statická metoda (nemá tedy přístup k instanci třídy), spouštěná před každým *render()*em, jak nyní při inicializaci, tak posléze při každé změně. Slouží k výpočtu stavu z props.

Následně se spustí první *render()*, na základě kterého se komponenta teprve objeví v DOMu.

Poslední metodou inicializace je *componentDidMount()*, což je doporučené místo jak pro manipulaci elementů DOMu (které až do konce *render()* neexistovaly), tak i pro různou komunikaci – ať již se serverem, nebo s poskytovateli aplikačního stavu, například pro hlášení se k odběru změn modelu. (29)

3.3.4.3.2 Změny

Poté je komponenta plně inicializována, a čeká na změnu. Ta může přijít od rodiče změnou props, a nebo interakcí uživatele, způsobí-li tato změnu stavu. V obou případech se nejprve spustí *getDerivedStateFromProps()*.

Následně může komponenta v metodě *shouldComponentUpdate()* rozhodnout, pomocí svojí návratové hodnoty (true nebo false), zda se má ve vyhodnocování možných změn pokračovat, nebo ne. Hlavním cílem této metody je vyhnutí se zbytečnému překreslování, pokud je předem jasné, že výsledek vykreslení by byl stejný, aby se neplýtvalo strojovým časem.

Alternativou k vlastní implementaci *shouldComponentUpdate()* a ručnímu porovnávání je při vytváření třídy dědit *React.PureComponent* místo *React.Component* – třída založená na *PureComponent* sama porovnává state a props při změně, a ve většině komponent je stejně efektivní jako ruční porovnání. Nejlepším řešením, možným jen u komponent kde nejsou potřeba metody životního cyklu, je použití funkční komponenty.

Pokud se komponenta rozhodne nepokračovat, žádná z následujících metod se již nespouští, a čeká se na další změnu. Při rozhodnutí pro pokračování se nyní spustí *render()*, a vypočítá se nový vzhled komponenty.

Před jejím propsáním do DOMu se však nejprve spustí metoda *getSnapshotBeforeUpdate()*, která na základě již nových props a stavu, ale ještě starého DOMu, může poznamenat data pro pozdější použití.

Teprve poté se pomocí Virtual DOMu výsledek *render()*u porovná se současným stavem, a jejich rozdíl promítne do prohlížeče.

Poslední metodou při změně je *componentDidUpdate()*, která probíhá po dokončení *render()*u, a slouží k manipulaci prohlížečového DOMu, který byl právě změněn. Má dostupné jak staré, tak aktuální props a state, a také data poznamenaná v *getSnapshotBeforeUpdate()*. Toto je, stejně jako *componentDidMount()*, doporučené místo pro různé vedlejší efekty.

3.3.4.3.3 Zánik

Při odpojování a ničení komponenty se, jako úplně poslední krok, zavolá metoda *componentWillUnmount()*. Ta je určená pro úklid jakýchkoliv vedlejších efektů komponenty, tedy hlavně zrušení síťových požadavků a odhlášení se od poslouchání na události, například změn v modelu. (29)

3.3.4.4 Stav aplikace

U jednoduchých aplikací je možné udržovat stav celé aplikace například v kořenové komponentě, a potomkům předávat části stavu skrz props. Stejně tak se uživatelská interakce dostane z potomků do kořenové komponenty skrz zpětná volání. Veškeré síťové dotazy také řeší kořenová komponenta. K tomuto pak není zapotřebí žádných dalších knihoven, v jednoduché aplikaci je pravděpodobně nutné napsat méně kódu, a tak dále.

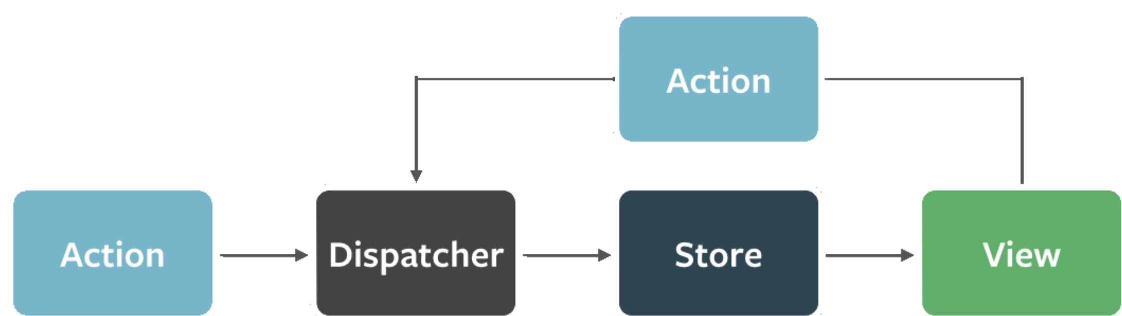
Pro jakoukoliv, byť jen trochu složitější, aplikaci by však tento přístup byl příliš náročný, kořenová komponenta příliš velká, a při větším množství větvení bude, kvůli opakování, kódu příliš. Existuje proto několik knihoven/architektur na manažování aplikačního stavu, ze kterých lze vybírat. Mezi nejznámější a nejpoužívanější se řadí Flux a Redux.

3.3.4.5 Flux

Přímo od Facebooku je již dlouho (červenec 2014) (35) k dispozici Flux, který má blíže k architektonickému návrhovému vzoru než knihovně jako takové. Existují různé implementace, nebo je možné architekturu používat bez dalších knihoven a rozšíření.

Tok dat je zde jednosměrný (Obrázek 8): uživatel či síť vyvolají *akci* (událost), která má daný typ a volitelně další data. Všechny akce zpracovává *dispečer*, u kterého mají jednotlivé sklady (*stores*) zaregistrované typy funkcí které je zajímají. Dispečer všem takovým akci včetně libovolných dat předá, sklady podle akce pozmění svůj stav, a vyvolají událost změny. Na změny ve skladech zas poslouchají jednotlivé komponenty, a ty tedy obnoví svůj stav. (36)

Obrázek 8, Tok dat v architektuře Flux



Zdroj: Flux In Depth Overview (36)

Tato smyčka je nezávislá na stavu jednotlivých komponent: zatímco změna textového pole formuláře pravděpodobně způsobí změnu jen lokálního stavu, odeslání formuláře naopak vyvolá akci a ovlivní stav celé aplikace.

3.3.4.6 Redux

Na druhou stranu Redux je novější (květen 2015) (37), a vznikl jako reakce na některé problémy Fluxu. Některými vývojáři je považován za implementaci Fluxu (38), protože na rozdíl od Fluxu má vlastní implementaci, a to jak obecnou, tak specificky pro React (39), a z Fluxu vychází. Jinými je (podle mě správně) považován za vlastní architekturu, vzhledem k níže rozebraným rozdílům.

Redux je možné používat s jinými knihovnami a frameworky, ale byl vytvořen primárně pro React, a zde se budeme bavit o jejich kombinaci.

Fluxu je celkem podobný, a asi hlavním rozdílem je, že existuje jen jeden sklad. Ten obsahuje celý stav aplikace, tedy veškerá její data. Tento stav je jeden neměnitelný objekt, který se při změně stavu nahrazuje. Toto umožňuje nejen celou aplikaci jednoduše *(de)serializovat* (tedy veškerý stav celé aplikace, zde reprezentovaný jako objekt, převést do formátu co lze ukládat na disk či posílat po síti), ale také se vracet v čase (40).

O jednotlivé sekce tohoto stavu se stará *reducer*, což je funkce, která z předchozího stavu a vyvolané akce deterministicky vytvoří nový stav. Většinou se skládá z nezávislých částí, kterým se také říká reducery, a ty se kombinují, ale v jednoduché aplikaci to může být jen jedna normální funkce. (41)

V porovnání s Fluxem se pro změnu stavu stále vyvolávají akce, ale funkci dispečeru plní samotný sklad, což je možné, protože je vždy jen jeden. Reducer vytvoří nový stav, a komponenty které odebírají změny stavu jsou na to upozorněny. Komponenty lze napojovat na Redux ručně, ale je doporučeno tyto komponenty nechat vygenerovat Reduxem – výsledkem je, že vybrané části stavu dostávají jako props, a vybrané akce mohou volat skrz funkce zpětného volání, také předané skrz props. (42)

3.3.4.7 Routování, rozdělení aplikace

Stejně jako tomu bylo u udržování aplikačního stavu, u jednoduchých či specializovaných aplikací nemusí být nutná nejen knihovna na routování, ale ani routování

samotné. Vlastní řešení může používat podmínky v kořenové metodě, která podle stavu či adresy vykreslí jednu z několika komponent reprezentujících jednotlivé stránky.

Pro větší aplikace toto samozřejmě není vhodné. Zdaleka nejrozšířenější knihovnou je React-router, ačkoliv nespadá pod Facebook, a na samotném Facebooku se nepoužívá. Jeho první verze vznikla přibližně rok (květen 2014) (43) po Reactu samotném, a od té doby prošel několika velkými změnami.

Ve verzi 4 funguje v základu víceméně jak je popsáno výše: jako kořenová komponenta je použit <*Router*>, uvnitř něj jsou nadefinovány elementy, které se vykreslují vždy (navigace, hlavička atp.), a také první úroveň <*Route*> komponent, s nadefinovanou adresou a komponentou kterou vykreslují, pokud cesta odpovídá. Vnořené adresy se pak řeší pomocí dalších <*Route*> vnořených někde ve vykreslované komponentě. (44)

Díky tomuto přístupu není nutné znát všechny možné adresy a jejich komponenty předem, a je možné aplikaci stahovat po částech, což umožnuje nejen rychlejší prvotní načtení, ale i rozdělování aplikace podle uživatele, například podle přihlášení, nebo práv. (45)

3.3.4.8 React Native

Za zmínku zajisté stojí i React Native, což je samostatný framework pro vývoj mobilních aplikací. Nejedná se o způsob jak převést webovou aplikaci na mobilní, a views ani výsledná aplikace nijak nepoužívají HTML, ale nativní ovládací prvky.

Pro psaní kódu se i zde používá JavaScript ve verzi *ECMAScript 6*, principy psaní aplikace jsou stejné, a tak dále. Výhodou tedy není použití stejné aplikace pro web i mobilní zařízení, ale použití stejných vývojářů a znalostí pro vývoj obou aplikací. Díky tomu jsou mobilní aplikace opravdu nativní, se všemi výhodami co toto přináší. (46)

3.3.4.9 Licence

React jako takový byl původně stvořen Facebookem pro vlastní použití, ale na jaře 2013 (47) byl uvolněn pro veřejnost jako open-source, nejprve pod Apache licencí (48); tato licence byla později (říjen 2014) změněna na BSD licenci (49), a nedávno (září 2017) na MIT licenci (50). Na vývoji Reactu se tedy podílí celá komunita.

Ostatní knihovny v ekosystému spadají v některých případech také pod Facebook (například Flux (51)), v jiných pod společnosti a týmy používající React (například *react-*

router (52)), nebo i pod nezávislé vývojáře (například *react-redux* (53) a *react-translate-component* (54)). Jejich licence tedy mohou být různé, a je proto nutné kontrolovat licence každého, byť sebemenšího rozšíření, které bychom chtěli použít.

Z tohoto rozdělení vyplývají i další určité nevýhody, nespojené s licencemi jako takovými: libovolné knihovně může skončit vývoj, vývojáři samotného Reactu nemusí sdílet vizi s ostatními vývojáři a určitý způsob použití může přestat být možný, a tak dále.

4 Vlastní práce

4.1 Téma projektu

Cílem této části práce je vytvořit aplikaci postavenou na technologiích probraných v teoretické části, a to specificky webovou aplikaci založenou na knihovně React a architektuře Redux. Aplikace má sloužit jako front-end pro redakční systém WordPress.

4.1.1 WordPress

WordPress je nejpopulárnějším redakčním systémem, používaným na třetině celého internetu (55). Jedná se o svobodný a otevřený software (FOSS), dostupný pod licencí GNU GPLv2 (56), postavený na technologiích PHP a MySQL (57). Jedná se o klasickou webovou aplikaci, kde uživatelská interakce způsobuje načítání celé nové webové stránky, jak bylo rozebráno v teoretické části této práce.

4.1.2 WordPress REST API

WordPress byl pro tuto práci vybrán nejen díky své popularitě, ale také vzhledem k poskytovanému API, které bude sloužit jako back-end pro vyvíjenou aplikaci. Toto API je založeno na architektuře REST, a komunikace s ním probíhá pomocí HTTP dotazů. Tělo dotazu používá formát JSON. (58)

4.2 Požadavky

Z pohledu uživatele se WordPress dělí na dvě části. První je část veřejná, která je dostupná bez přihlášení, zobrazuje příspěvky a stránky, a umožňuje přidávání komentářů. Druhou částí je administrace, která vyžaduje uživatelský účet (jež musí vytvořit administrátor, nelze se pouze zaregistrovat), umožňuje psaní příspěvků, vytváření kategorií, změny nastavení, a další.

Například soukromý blog tedy pravděpodobně bude mít pouze jednoho registrovaného uživatele, a to majitele – nikdo jiný administraci nikdy neuvidí. Toto je reflektováno také v možném nastavení – veřejné části lze lehce měnit vzhled, s několika předinstalovanými tématy, zatímco administrace takovou možnosti nemá. API poskytuje metody pro změny nastavení jinak dostupné skrz administraci, a bylo by tedy teoreticky možné vytvořit

kompletní náhradu, ale složitost projektu by se tím zněkolikanásobila, a přínos by byl minimální.

Vyvíjená aplikace proto bude obsahovat pouze funkcionalitu veřejné části.

4.2.1 **Části aplikace**

Požadovaná funkcionalita byla určena průzkumem instance WordPressu, a také nabízených metod API. Jako hlavní aspekty byly identifikovány příspěvky, kategorie a značky, statické stránky, uživatelé a komentáře.

4.2.1.1 Příspěvky

Hlavní stavební kostkou redakčního systému jsou příspěvky či články (Posts). Příspěvek má název, jednoho autora, datum a čas publikace a poslední změny, obsah, komentáře, náleží do jedné nebo víc kategorií, a může (ale nemusí) mít značky.

Obsah příspěvku může sestávat z odstavců textu, nadpisů, různých oddělovačů, a médií (hlavně obrázků). Z API je tento obsah dostupný ve dvou variantách: jako náhled, což je omezené množství (přibližně odstavec) prostého textu, vhodný například pro seznamy příspěvků, kde by se celý obsah stejně nezobrazoval, a jeho přenos by tedy plýtval časem a daty. Druhou variantou je celý obsah, včetně nadpisů a multimédií, přenášený jako fragment HTML, a tedy vhodný pro vložení do stránky bez jakýchkoliv úprav.

4.2.1.2 Kategorie

Příspěvky jsou řazeny do kategorií (Categories), přičemž příspěvek musí náležet alespoň do jedné z nich – při instalaci je vytvořena kategorie „Nezařazené“, a pokud je příspěvek odebrán ze všech kategorií, automaticky je zařazen do ní. Kategorie tedy obsahuje příspěvky, má název, a dále může mít krátký popis.

4.2.1.3 Značky

Značky (Tags), také známé jako #hashtagy, se od kategorií liší tím, že nejsou povinné – příspěvek nemusí mít žádnou, nebo může mít jednu či několik.

4.2.1.4 Statické stránky

Stránky (Pages) se velmi podobají příspěvkům, mají název, autora, datumy, a stejný typ obsahu. Na rozdíl od příspěvků však nespadají do kategorií a nemají značky. Jejich doporučené použití se liší – v případě webu sloužícího jako prezentace firmy mohou být příspěvky určeny pro novinky a podobné, zatímco stránky by neměly být vázány na specifický čas, ale udržovány aktuální, například „O nás“ a „Kontakt“.

4.2.1.5 Uživatelé

Uživateli (Users) jsou zde myšleni pouze registrovaní uživatelé, ne všichni návštěvníci – také by je šlo nazvat „autoři“, „spisovatelé“ či „novináři“, podle zaměření webu. Z pohledu této aplikace má uživatel jméno, email, obrázek, a může mít (odkaz na) svůj web, krátký popis, a samozřejmě příspěvky které vytvořil. Další aspekty jako heslo, role (pro vytváření a úpravy příspěvků, změny nastavení a podobně) nejsou pro veřejnou část relevantní.

4.2.1.6 Komentáře

Komentáře (Comments) se nachází pouze na příspěvcích. Statické stránky, kategorie, značky, ani uživatelské profily nemohou být okomentovány. Vytvářet komentáře mohou jak registrovaní uživatelé, tak i návštěvníci co účet nemají – v tom případě musí být komentář nejprve schválen v administraci, a teprve poté je veřejně dostupný.

Komentář sestává z textového obsahu, datumu, a autora – ten je definovaný buď skrz své unikátní ID (v případě registrovaného uživatele), nebo skrz jméno, email (neveřejný), obrázek, a potencionálně webový odkaz (veřejný). Komentář také může být označen jako přímá odpověď na jiný komentář.

4.2.2 **Stránky aplikace**

Z těchto informací byla určena požadovaná funkcionalita, organizovaná na jednotlivé stránky aplikace. Nachází se v **Error! Reference source not found.** níže.

Tabulka 1, Stránky aplikace

	Stránka v aplikaci	Účel	Cesta v API
1	Domovská stránka	Odkazy na jednotlivé části aplikace, náhledy	--
2.1	Seznam příspěvků	Všechny příspěvky včetně ukázky textu, chronologicky, stránkování – odkaz na detail	/posts
2.2	Detail příspěvku	Jeden příspěvek, celý obsah a všechny informace, seznam komentářů se stránkováním, možnost vytvoření nového komentáře	/posts
3.1	Seznam kategorií	Všechny kategorie, jejich popis a názvy příspěvků v těchto kategoriích	/categories
3.2	Detail kategorie	Jedna kategorie včetně popisu, všechny její příspěvky se stránkováním, včetně ukázky	/categories
3.3	Seznam značek	Všechny značky (#hashtagy), a názvy příspěvků s těmito značkami	/tags
3.4	Detail značky	Jedna značka, příspěvky se stránkováním a ukázkou	/tags
4.1	Seznam stránek	Všechny statické stránky s ukázkou, chronologicky, se stránkováním	/pages
4.2	Detail stránky	Jedna statická stránka s veškerým obsahem	/pages
5	Uživatelský profil	Profil registrovaného uživatele, seznam příspěvků kterých je uživatel autorem	/users

Zdroj: Autor

4.3 Použité technologie

Jak již bylo zmíněno, základem aplikace jsou technologie *React* a *Redux*. Všechny zde zmíněné knihovny jsou dostupné skrz registr npm, a to pod názvem zde použitým.

4.3.1 react-create-app

Oficiálně doporučenou cestou pro tvorbu aplikací založených na Reactu je příhodně nazvaný nástroj *react-create-app*. Při svém spuštění vytvoří základní kostru aplikace, s již nastaveným sestavováním kódu, transpilací Javascriptu do ES5, lokálním vývojovým serverem, a exportem pro nasazení na produkční server.

Vnitřně aplikace používá Babel pro transpilaci do ES5 a podporu JSX, a Webpack pro sestavení mnoha jednotlivých Javascriptových souborů se zdrojovým kódem a všech závislostí do výsledného „balíčku“, tedy jednoho až několika málo Javascriptových souborů obsahujících výsledný minifikovaný a kompatibilní kód, a také pro organizaci a modifikaci různých dalších souborů (například obrázků).

Toto je však pro vývojáře neviditelné, pokud se nerozhodne z react-create-app „vystoupit“, což umožňuje detailní nastavení všech součástí, ale značně komplikuje aktualizace a změny. Dokud je vývojář „uvnitř“ react-create-app, stačí aktualizovat jediný balíček *react-scripts*, který obsahuje veškerou konfiguraci a závislosti, nutné pro výše popsanou (a další) funkcionalitu.

Pro přidávání dalších závislostí a jejich nastavování není potřeba vystupovat, a v případě této aplikace tomu nebylo učiněno.

4.3.2 **redux**

Redux součástí react-create-app není, a bylo tedy nutné ho nainstalovat zvlášť. Vzhledem k tomu, že samotný Redux nezávisí na Reactu, a lze používat s libovolným frameworkem či bez, do této kategorie spadá také rozšíření *react-redux*. To se stará o vytváření komponent napojených na stav a akce.

4.3.2.1 redux-saga

Pro vyvolávání vedlejších efektů (komunikaci se serverem a podobné) byla zvolena knihovna *redux-saga*. Pomocí ní se definují asynchronní funkce (ságy), které poslouchají na vyvolané akce, čtou stav, vyvolávají vlastní funkce, a vytváří vedlejší účinky.

4.3.3 **react-router-dom**

Routování obstarává již zmíněný react-router, přičemž pro routování v prohlížeči se instaluje balíček *react-router-dom*. Alternativou je *react-router-native*, pro React Native.

4.3.4 **SuperAgent**

Pro usnadnění komunikace po síti, a lepší kompatibilitu se starými prohlížeči než nativní *fetch*, byla zvolena knihovna *SuperAgent*.

4.3.5 styled-components

Grafický design a vzhled aplikace nejsou předmětem práce, ale úplně je ignorovat také nejde. Knihovna *styled-components* používá de facto standardní CSS (volitelně podporuje jednodušší syntaxi pro skládání stylů), vložené přímo do Javascriptu jako textový řetězec, a z nich generuje unikátně pojmenované třídy.

Vyhýbá se tím konfliktům, a zpříjemňuje vývoj, protože vzhled komponenty je definován přímo v ní, a ne ve velkém souboru stylů někde mimo. Toho lze také samozřejmě dosáhnout tím, že každá komponenta má svůj CSS soubor, ale řešení pomocí styled-components je elegantnější.

4.3.6 prop-types

Kontrola typů props předávaných komponentě byla dříve součástí Reactu samotného, ale byla vytknuta do vlastní knihovny, nazvané *prop-types*. Zmíněna je zde pouze pro kompletnost.

4.3.7 moment

Knihovna pro práci s datumy a časy *moment* byla přidána během vývoje, protože nativní třída *Date()* je vcelku zastaralá a nepříliš přizpůsobitelná. Aplikace by bez ní fungovala víceméně stejně, jen s méně čitelnými formáty datumů.

4.4 Implementace

Valná většina kódu se nachází uvnitř adresáře *src*, a následující sekce popisují jeho obsah.

4.4.1 Moduly

Aplikace je rozdělena na samostatné moduly, každý z kterých se stará o jeden aspekt definovaný v požadavcích. Takový modul obsahuje svoje akce (respektive jejich typy), ságy, reducer, funkce pro komunikaci s API WordPressu, a dva typy komponent. První typ jsou takzvané Containery, tedy komponenty napojené na Redux. Druhým typem jsou prezentační komponenty, které pouze dostávají props od rodiče, a o Reduxu ani neví.

Tyto moduly jsou Posts, Categories, Pages, Comments, a Users. Například modul Pages obsahuje následující soubory:

4.4.1.1 api.js

Sestává z funkcí *getList(page, amount)* a *getDetail(id)* - obě pomocí knihovny SuperAgent posílají dotaz na WordPress REST API. GetList stahuje pole statických stránek bez plného obsahu (pro rychlejší přenos), a podporuje stránkování. Součástí odpovědi je také počet všech stránek, umožnujíc aplikaci vědět zda jsou všechny staženy, či může posílat dotazy pro další. GetDetail stahuje jednu stránku, včetně jejího plného obsahu.

4.4.1.2 constants.js

Obsahuje typy akcí, v tomto případě pro dvě možné skupiny akcí, *LIST* a *DETAIL*. Napříč celou aplikací skupiny akcí ze stejného tria: *REQUEST*, *SUCCESS*, a *FAIL*, značící požadavek o získání či poslání dat, úspěch, a neúspěch, respektive. Toto samozřejmě platí pouze pro akce mající co do činění se sítí.

4.4.1.3 DetailContainer.jsx a ListContainer.jsx

Komponenty napojené na Redux. Sestávají z vnitřní a vnější části, přičemž obě jsou komponenty, ale zvlášť nemají využití, takže se navenek tváří jako jedna. Každý Container by mohl být rozložený do dvou souborů, ale lépe se s ním pracuje v jednom.

Vnitřní reaguje na props a jejich změny, volí vhodnou prezentační komponentu, a sama neví o Reduxu. Vnitřní nemá definované tělo (ať již třídou či funkcí), ale je generována Reduxem, z funkcí říkajíc jaké části stavu a volání akcí předávat vnitřní části jako props, a z názvu oné vnitřní komponenty.

Specificky DetailContainer.jsx se stará o zobrazení detailu stránky, a to voláním akce pro získání dat, a předáváním těchto komponentě Page.jsx. Naopak ListContainer.jsx se stará o seznamy, a to jak náhled pouze prvních pár názvů (viz domovská stránka), tak i seznam jednotlivých statických stránek s náhledy obsahu a stránkováním.

4.4.1.4 index.js

Samotný index.js de facto nic nedělá, slouží pouze pro odlišení „veřejných“ a „soukromých“ částí modulu a hezčí importy, viz Obrázek 9. Importy bez index.js jsou stále

možné, a jde tedy importovat i komponenty které v index.js zmíněny nejsou, protože jsou určeny pro využití uvnitř.

Obrázek 9, Importy

```
// skrz index.js
import { PagesList, PageDetail } from './pages';

// bez index.js
import PagesList from './pages/ListContainer';
import PageDetail from './pages/DetailContainer';
```

Zdroj: Autor

4.4.1.5 Page.jsx

Toto je prezentační komponenta, která podle props zobrazuje buď indikátor načítání dat, zprávu o neexistující stránce (při ručním zadání ID skrz adresní řádek prohlížeče, nebo při mrtvém odkazu), náhled, nebo celý obsah stránky. Jak náhled tak celý obsah se nachází ve stejném souboru, protože sdílí vzhled a rozložení.

Ve chvíli kdy má komponenta k dispozici pouze náhled, a ne celý obsah, tedy v okamžik kdy uživatel byl na seznamu stránek a z náhledu stránky přešel na detail, se náhled zobrazí první, a při načtení zbylých dat jej komponenta vymění za plný obsah, bez jakéhokoliv problikávání či podobného.

4.4.1.6 reducer.js

Jak bylo zmíněno v sekci o Reduxu, výsledný Reducer je jediná funkce, ale zpravidla se kombinuje z několika menších, a toto je ona menší část. Zajímají ji pouze akce pro statické stránky, a stará se o část stavu spadající jim, vše ostatní ignoruje.

Pages reducer mění stav při následujících akcích: *LIST_SUCCESS* získané pole stránek ukládá na dvě části – skutečný obsah do slovníku *byId*, a samotná id získaných stránek na odpovídající místa v poli *all*, kde mají zatím nestažené stránky vynechaná místa. *DETAIL_SUCCESS* pouze ukládá staženou stránku do slovníku *byId*, a poslední *DETAIL_FAIL* do slovníku ukládá informaci o tom, že daná stránka neexistuje, a uživateli se má zobrazit zpráva o tomto, a ne indikátor načítání.

4.4.1.7 sagas.js

Ságy se zde nacházejí tři: jedna mateřská, která obaluje ostatní ságy, určuje kdy se mají spouštět – nejen jako důsledek které akce, ale také jestli pro každou akci, jen první a blokovat do ukončení běhu, nebo zda při další akci v průběhu předchozí dotaz zrušit, a místo něj spustit nový. Taková mateřská sága se nachází v každém modulu.

Další dvě ságy se starají o stažení seznamu a detailu. Jejich průběh je, ve většině případů napříč celou aplikací, víceméně stejný: nejprve je sága spuštěna skrz *REQUEST* akci, a podívá se do storu, zda jsou tato data již stažena – díky tomu není třeba existenci dat řešit složitě skrz props v Containerech, které vytvoří *REQUEST* akci vždy při změně props, zpravidla při navigaci.

Pokud data stažena v tuto chvíli nejsou (vůbec, či částečně – například pokud je stažen náhled, ale akce požaduje detail), sága zavolá odpovídající funkci z api.js se správnými parametry, čeká na výsledek. Pokud jsou data úspěšně stažena, pošle je spolu s akcí *SUCCESS* reduceru, pokud proces selže, chybu odchytí, a vyšle *FAIL* akci. Tím jeden průběh této ságy končí, a čeká se na další *REQUEST*.

4.4.1.8 Ostatní moduly

V ostatních modulech jsou soubory všech těchto typů, ačkoliv počet Containerů a prezentačních komponent se liší. Za zmínku zde stojí modul Categories, který v sobě zastřešuje jak kategorie, tak i značky, vzhledem k jejich téměř identické implementaci. Také modul Comments, který umožňuje vytvoření nového komentáře a jeho odeslání na server, kde reducer při úspěšném odeslání komentář ukládá do storu, bez nutnosti stahovat seznam znovu.

4.4.2 **Zastřešující kód**

Tedy soubory nespadající do modulů. Ty pojmenované analogicky k součástem modulů, tedy **reducers.js** a **sagas.js**, slouží pro zkombinování jednotlivých reducerů do hlavního, a spuštění mateřských ság, respektive.

Podobnou funkci poskytuje soubor **Routes.jsx**, který obsahuje možné adresy pro celou aplikaci, interpretuje z nich proměnné, a vykresluje ten správný Container ze správného modulu. Svým způsobem tedy kombinuje Containery do jedné komponenty.

Home.jsx je komponenta vykreslovaná pro domovskou stránku. Není Containerem, protože sama nekomunikuje s Reduxem a neřeší data, pouze importuje a vykresluje Containery z modulů, včetně dříve zmíněného *PagesList* (který ze vykresluje názvy a odkazy na prvních pár statických stránek), protože Container může být zanořen libovolně hluboko ve stromu komponent, a komponenty nad ním o něm nemusí vědět. Toho využívají i některé Containery v modulech, například *PageDetail* v hlavičce vykresluje obrázek a jméno autora – toto však nedělá sám, pouze zná autorovo ID, a předává jej *UserDetail*u, který si sám pošle požadavek o data.

index.js je vstupní místo aplikace, importuje všechny její části, včetně většiny souborů v této sekci. Vytváří Redux store, váže na něj ságy, inicializuje routování, registruje serviceWorker (níže), vkládá globální styly, a finálně vkládá React aplikaci do dokumentu.

App.jsx je vždy vykreslovaná komponenta, která obsahuje horní menu s odkazy na části aplikace. Mohla by také obsahovat patičku nebo jiné části stránky.

serviceWorker.js registruje aplikaci jako progresivní webovou aplikaci, umožňující jí ukládat do dlouhodobé prohlížečové cache, takže se při následujících návštěvách spustí ihned, a do určité míry funguje i offline – tato aplikace se offline nemůže připojit k WordPress API, a tudíž stáhnout příspěvky a podobně, a redakční systém bez obsahu není příliš zajímavý. Při opakovaných návštěvách online se ale použije kód z cache, a ihned se tedy stahují příspěvky – aplikace funguje se stejnou rychlostí jako při navigaci mezi adresami v již načtené aplikaci.

styles.js obsahují jak globální styly, tedy styly pro celý HTML dokument a styly pro všechny elementy určitého typu (například odkazy), tak i malé ne-úplně-komponenty používané napříč aplikací. Například *Button*, základní styly po všechny tlačítka, lze používat tak jak jsou, nebo modifikovat pro různá specifická tlačítka.

NotFound.jsx slouží jako routa pro nevalidní adresy, a také jej vykresluje například *PageDetail* v případě neexistujícího ID.

utils.js obsahuje různé pomocné funkce využívané napříč aplikací, a předchází duplikaci kódu.

4.4.3 Adresář _global

V tomto adresáři, začínající podtržítkem protože se nejedná o modul, se nachází různé komponenty používané napříč aplikací, uvnitř Containerů a jiných komponent – na rozdíl od komponent jako *App.jsx* a *Home.jsx*, které jsou používané při inicializaci aplikace, a jako

route, respektive. V podadresáři *icons* se nachází jak komponenta *Icon.jsx*, tak i jednotlivé obrázky používané jako ikony. A *index,js* opět exportuje ostatní komponenty pro snadnější import, s našeptáváním od editoru kódu, jinde v aplikaci.

4.4.4 Soubory mimo src

Adresář **public** obsahuje **index.html**, což je HTML dokument, do kterého se aplikace vkládá. Lze zde definovat různé tagy v hlavičce, přiložit externí skripty jako Google Analytics, nebo například vložit CSS soubor se styly, pokud tyto nejsou definovány přímo v aplikaci. V případě této aplikace je **index.html** de facto nezměněn, krom tagu *<title>*. Dále se zde nachází ikona **favicon.ico**, a také **manifest.json**, který pro progresivní webovou aplikaci definuje název, ikonu, a další, například pro její uložení na domovskou obrazovku mobilního zařízení.

Do adresáře **build** se aplikace sestavuje, do adresáře **node_modules** se instalují balíčky z npm, a jak skripty pro sestavení, tak i balíčky ke stažení, jsou určeny v **package.json**.

Soubory **.env** a **.env.development** definují proměnné prostředí pro nastavení aplikace, více rozebrané v pozdější sekci. Také je lze nastavit v příkazové řádce a přímo v operačním systému.

Ostatní soubory jsou git a nebo pro Github, a aplikace samotná je nepoužívá.

4.5 Spuštění aplikace

Demo aplikace je dostupné na adrese https://david-pavel.github.io/react-wordpress/, přičemž jako API využívá oficiální demo WordPressu pro vývojáře pracující s jeho API, dostupném na adrese https://demo.wp-api.org/.

Zdrojový kód se nachází na adrese https://github.com/david-pavel/react-wordpress. V branchi gh-pages se nachází sestavená verze aplikace, která běží jako výše zmíněné demo.

4.5.1 Proměnné

Pro změnu nastavení je třeba aplikaci sestavit znovu – aby aplikace fungovala z libovolného webového serveru, včetně těch které pouze podávají statické soubory, nelze proměnné načítat za běhu. Tyto proměnné jsou:

REACT_APP_ADMIN obsahuje adresu administrace, pouze pro odkaz v horní liště.

REACT_APP_BACKEND obsahuje adresu REST API.

REACT_APP_ROUTER_BROWSERHISTORY jakákoliv pravdivá hodnota (neprázdný textový řetězec) zapíná podporu libovolných cest pro jednostránkové aplikace, a tu musí podporovat webový server. Jinak aplikace používá cesty v hash parametru, který se neposílá na server.

REACT_APP_ROUTER_BASE existuje pro případ běhu aplikace s libovolnými cestami, ale mimo kořenový adresář serveru, tedy například *example.com/blog*.

4.5.2 Lokálně

Pro sestavení je potřeba Node.js (https://nodejs.org/), přičemž aplikace byla vyvinuta na verzi 11.

V adresáři s aplikací poté stačí spustit nejprve „npm install“, a poté „npm start“, a spustí se lokální vývojový webový server s aplikací, který používá hodnoty z **.env.development,** a podporuje libovolné cesty.

Pro sestavení nejprve „npm install“, a poté „npm run build“. Sestavená aplikace se uloží do adresáře **build**, odkud je potřeba ji přesunout na webový server.

5 Zhodnocení a doporučení

Díky omezenému rozsahu požadavků na funkcionalitu, a tím pádem poměrné jednoduchosti aplikace, je výsledek svižný a lehký. Sestavená aplikace se vším všudy, kód, HTML, styly, a závislosti, s jedinou výjimkou map minifikovaného kódu (které slouží pouze pro ladění sestaveného kódu, a uživateli je netřeba poskytovat), má pouhých 369 kilobajtů. Po kompresi GZip, která je podporovaná de facto vším (včetně Internet Exploreru 6), dokonce 110 kilobajtů.

Uživateli se tedy aplikace stáhne a spustí rychlostí porovnatelnou s klasickým webem, a všechny další interakce jsou samozřejmě rychlejší.

5.1 Budoucí cíle

Přidání administrace je mimo rozsah pro tuto aplikaci, výsledný projekt by byl mnohem rozsáhlejší a složitější, a jeho účel by byla kompletní náhrada WordPress rozhraní, ne rozšíření. Aplikace ale má několik dosažitelných cílů pro budoucí vývoj.

Lokalizace a přehazování jazyků nebylo vyvinuto, protože obsah je v jednom jazyce, a mít několik odkazů a nadpisů v jazyce jiném by nedávalo smysl. Aplikace by ale mohla další jazyky podporovat při sestavení, bez nutnosti změn textů přímo v komponentách.

S lepší znalostí CSS by jistě šlo vylepšit vzhled, a s použitím proměnných, ať již v CSS nebo v Javascriptu, by bylo možné implementovat různá grafická témata bez duplikace kódu. Například světlé a tmavé téma, měnitelné za běhu aplikace, pro čtení ve dne a v noci.

WordPress i jeho REST API, obsahují mnohem více funkcionality, než je v aplikaci využito. S lepší znalostí WordPressu samotného by šlo lépe určit další možná rozšíření.

6 Závěr

Přehled ekosystémů a jimi používaných architektur ukázal rozdíly mezi dřívějšími řešeními jako Angular.js, vytvořenými v době jednodušších aplikací, pro které je obousměrný tok dat vhodný, a řešeními vytvořenými na základě problémů které se ukázaly při jeho použití v nových, komplikovanějších aplikacích. Také rozdělení aplikace na komponenty je použité ve všech zkoumaných ekosystémech, včetně nového Angularu, který se jinak drží oboustranného toku dat.

Ekosystém Reactu je oproti jiným, stabilnějším ekosystémům, například v oblasti vývoje desktopových aplikací, velmi rozdílný. Hlavní výhodou je bezpochyby rozlehlost ekosystému a velké množství dostupných informací – pokud se při vývoji objevil problém, ve většině případů stačilo zadat správnou formulaci do vyhledávače, a objevil se článek nebo diskuze mezi vývojáři, velmi často včetně řešení.

Rychlé tempo změn se ukázalo ještě rychlejší, než bylo očekáváno. Rozdíly mezi Reactem, a dřívějšími ekosystémy jako Angular.js, byly samozřejmě velké, ale překvapivě velké rozdíly se objevily i mezi Reactem v době počátku této práce, a Reactem teď na konci. Verze 16.3 z dubna 2018 kompletně změnila životní cyklus komponenty, a relevantní sekci práce bylo potřeba přepsat. V případě čerpání informací z knih, místo oficiální dokumentace by se vývojář velmi divil, proč příklady z knihy nefungují.

Stejně jako informace poskytoval ekosystém i knihovny, komponenty a nástroje v nepřeberném množství. Problémem není toliko najít řešení, ale spíše které řešení z těch mnoha dostupných, často vyvíjených souběžně, zvolit.

To samé platí ostatně i pro ekosystémy samotné. React je sice momentálně nejpopulárnější, ale to není automaticky záruka, že je i nejlepší. Jak ekosystémů, tak jejich konkurenčních součástí existuje mnoho, protože každý řeší trochu jiné problémy, o něco jiným způsobem.

7 Seznam použitých zdrojů

(1) GOOGLE LLC. Understanding web pages better. *Google Webmaster Central Blog*. 2014. Dostupné také z: https://webmasters.googleblog.com/2014/05/understanding-web-pages-better.html

(2) GOOGLE LLC. Understanding rendering on Google Search. *Google Developers* [online]. b.r. [cit. 2019]. Dostupné z: https://developers.google.com/search/docs/guides/rendering

(3) MOLDOVAN, Alex. Demystifying server-side rendering in React. *FreeCodeCamp*. 2018. Dostupné také z: https://medium.freecodecamp.org/demystifying-reacts-server-side-render-de335d408fe4

(4) ABRAMOV, Dan. Server Rendering. *Redux Docs* [online]. b.r. [cit. 2019]. Dostupné z: https://redux.js.org/recipes/serverrendering

(5) TERLSON, Brian. ECMAScript® 2018 Language Specification. *Ecma international*. 2018. Dostupné také z: https://www.ecma-international.org/ecma-262/9.0/index.html

(6) MOZILLA. WebAssembly. *MDN web docs* [online]. b.r. [cit. 2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly

(7) DEVERIA, Alexis. Can I Use ECMA. *Caniuse.com* [online]. b.r. [cit. 2019]. Dostupné z: https://caniuse.com/#search=ecma

(8) ZAYTSEV, Juriy. ECMAScript 5 compatibility table. *Compat-table* [online]. b.r. [cit. 2019]. Dostupné z: https://kangax.github.io/compat-table/es5/

(9) W3TECHS. Usage of server-side programming languages for websites. *W3Techs* [online]. b.r. [cit. 2019]. Dostupné z: https://w3techs.com/technologies/overview/programming_language/all

(10) HOLM, Magnus. JSON: The JavaScript subset that isn't. *The timeless repository*. 2018. Dostupné také z: http://timelessrepo.com/json-isnt-a-javascript-subset

(11) KOCH, Peter-Paul. The AJAX response: XML, HTML, or JSON?. *Quirksmode*. 2005. Dostupné také z: https://www.quirksmode.org/blog/archives/2005/12/the_ajax_respon.html

(12) POTTER, John. React vs angular vs vue vs @angular/core. *Npm trends* [online]. b.r. [cit. 2018-03-07]. Dostupné z: http://www.npmtrends.com/react-vs-angular-vs-vue-vs-@angular/core

(13) OSMANI, Addy. *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. 1. vyd. Mountain View: O'Reilly Media, 2012.

(14) PERES, Rui. Model-View-Controller (MVC) in iOS: A Modern Approach. *Ray Wenderlich*. 2016. Dostupné také z: https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach

(15) GOOGLE LLC. Angular.js License. *Github*. 2018. Dostupné také z: https://github.com/angular/angular.js/blob/master/LICENSE

(16) GOOGLE LLC. Angular Licence. *Github*. 2018. Dostupné také z: https://github.com/angular/angular/blob/master/LICENSE

(17) GOOGLE LLC. Angular.js v0.9.0 release. *Github*. 2010. Dostupné také z: https://github.com/angular/angular.js/releases/tag/v0.9.0

(18) CORDLE, Chris. Why Angular 2/4 Is Too Little, Too Late. *Medium*. 2017. Dostupné také z: https://medium.com/@chriscordle/why-angular-2-4-is-too-little-too-late-ea86d7fa0bae

(19) GOOGLE LLC. Conceptual Overview. *AngularJS Documentation* [online]. b.r. [cit. 2018]. Dostupné z: https://docs.angularjs.org/guide/concepts

(20) EIDNES, Lars. AngularJS: The Bad Parts. *Lars Eidnes' blog*. 2014. Dostupné také z: https://larseidnes.com/2014/11/05/angularjs-the-bad-parts/

(21) RANGLE.IO. Components in Angular. *Rangle's Angular Training Book* [online]. b.r. [cit. 2018]. Dostupné z: https://angular-2-training-book.rangle.io/handout/components/

(22) YOU, Evan. Vue.js License. *Github*. 2017. Dostupné také z: https://github.com/vuejs/vue/blob/dev/LICENSE

(23) YOU, Evan. Comparison with Other Frameworks. *Vue.js Docs* [online]. b.r. [cit. 2018]. Dostupné z: https://vuejs.org/v2/guide/comparison.html

(24) FACEBOOK, INC. JSX In Depth. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/jsx-in-depth.html

(25) FACEBOOK. Draft: JSX Specification. *Facebook, Github Pages* [online]. b.r. [cit. 2018]. Dostupné z: https://facebook.github.io/jsx/

(26) FACEBOOK, INC. React Without JSX. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/react-without-jsx.html

(27) MCKENZIE, Sebastian. Babel REPL. *Babel* [online]. b.r. [cit. 2018]. Dostupné z: https://babeljs.io/repl/

(28) FACEBOOK, INC. Components and Props. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/components-and-props.html

(29) FACEBOOK, INC. React.Component. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/react-component.html

(30) FACEBOOK, INC. Reconciliation. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/reconciliation.html

(31) FACEBOOK, INC. Portals. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/portals.html

(32) FACEBOOK, INC. Integrating with Other Libraries. *React Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://reactjs.org/docs/integrating-with-other-libraries.html#integrating-with-other-view-libraries

(33) FACEBOOK, INC. Context. *React Docs* [online]. b.r. [cit. 2018]. Dostupné z: https://reactjs.org/docs/context.html#before-you-use-context

(34) MAJ, Wojciech. React lifecycle methods diagram. *Wojtekmaj.pl* [online]. b.r. [cit. 2019-03-07]. Dostupné z: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

(35) FACEBOOK, INC. Flux Initial Commit. *Github*. 2014. Dostupné také z: https://github.com/facebook/flux/commit/45c856e78783f132f491d9a374a3895a5ee1076c

(36) FACEBOOK, INC. Flux In Depth Overview. *Facebook Github Pages* [online]. b.r. [cit. 2018]. Dostupné z: https://facebook.github.io/flux/docs/in-depth-overview.html

(37) ABRAMOV, Dan. Redux Initial commit. *Github*. 2015. Dostupné také z: https://github.com/reactjs/redux/commit/8bc14659780c044baac1432845fe1e4ca5123a8d

(38) ABRAMOV, Dan. Prior Art. *Redux Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: http://redux.js.org/docs/introduction/PriorArt.html

(39) ABRAMOV, Dan. React-redux. *Github* [online]. b.r. [cit. 2018-03-07]. Dostupné z: https://github.com/reactjs/react-redux

(40) ABRAMOV, Dan. Three Principles. *Redux Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: http://redux.js.org/docs/introduction/ThreePrinciples.html

(41) ABRAMOV, Dan. Reducers. *Redux Docs* [online]. b.r. [cit. 2018]. Dostupné z: http://redux.js.org/docs/basics/Reducers.html

(42) ABRAMOV, Dan. Usage with React. *Redux Docs* [online]. b.r. [cit. 2018-03-07]. Dostupné z: http://redux.js.org/docs/basics/UsageWithReact.html#implementing-container-components

(43) REACT TRAINING. React-router first commit. *Github*. 2014. Dostupné také z: https://github.com/ReactTraining/react-router/commit/987de78deb9687f15133188f2e8e51ffd653794d

(44) REACT TRAINING. Philosophy. *React Router Docs* [online]. b.r. [cit. 2018]. Dostupné z: https://reacttraining.com/react-router/core/guides/philosophy/nested-routes

(45) REACT TRAINING. Code Splitting. *React Router Docs* [online]. b.r. [cit. 2018]. Dostupné z: https://reacttraining.com/react-router/web/guides/code-splitting

(46) MANGIN, Alexis. What are the main differences between ReactJS and React-Native?. *Medium*. 2016-12-28. Dostupné také z: https://medium.com/@alexmngn/from-reactjs-to-react-native-what-are-the-main-differences-between-both-d6e8e88ebf24

(47) FACEBOOK, INC. React Initial public release. *Github*. 2013. Dostupné také z: https://github.com/facebook/react/commit/e9e6b9b9b7558f1bc972f5cfb7b396d396a5508f

(48) FACEBOOK, INC. React License. *Github*. 2013. Dostupné také z: https://github.com/facebook/react/blob/e9e6b9b9b7558f1bc972f5cfb7b396d396a5508f/LICENSE

(49) FACEBOOK, INC. React License. *Github*. 2014. Dostupné také z: https://github.com/facebook/react/blob/dcf415c2b91ce52fd5d4dd02b70875ba9d33290f/LICENSE

(50) FACEBOOK, INC. React License. *Github*. 2017. Dostupné také z: https://github.com/facebook/react/blob/b765fb25ebc6e53bb8de2496d2828d9d01c2774b/LICENSE

(51) FACEBOOK, INC. Flux License. *Github*. 2014. Dostupné také z: https://github.com/facebook/flux/blob/master/LICENSE

(52) REACT TRAINING. React-router License. *Github*. 2018. Dostupné také z: https://github.com/ReactTraining/react-router/blob/master/LICENSE

(53) ABRAMOV, Dan. React-redux License. *Github*. 2017. Dostupné také z: https://github.com/reactjs/react-redux/blob/master/LICENSE.md

(54) ANDERT, Martin. React-translate-component License. *Github*. 2014. Dostupné také z: https://github.com/martinandert/react-translate-component/blob/master/LICENSE

(55) W3TECHS. Usage of content management systems for websites. *W3Techs* [online]. b.r. [cit. 2019]. Dostupné z: https://w3techs.com/technologies/overview/content_management/all

(56) WORDPRESS. GNU Public License. *WordPress* [online]. b.r. [cit. 2018]. Dostupné z: https://wordpress.org/about/license/

(57) WORDPRESS. Requirements. *WordPress* [online]. b.r. [cit. 2019]. Dostupné z: https://wordpress.org/about/requirements/

(58) WORDPRESS. REST API Handbook. *WordPress Developer Resources* [online]. b.r. [cit. 2019]. Dostupné z: https://developer.wordpress.org/rest-api/