Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Návrh a implementace webové aplikace s využitím React JS

Michal Honc

© 2019 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Michal Honc

Informatika

Název práce

Návrh a implementace webové aplikace s využitím React JS

Název anglicky

Design and implementation of a web application using React JS

Cíle práce

Diplomová práce je zaměřena na problematiku návrhu a implementace webových aplikací. Hlavním cílem práce je návrh a implementace webové aplikace ve skriptovacím jazyce Javascript s pomocí knihovny React JS. Dílčí cíle práce jsou:
- vymezit základní pojmy problematiky a jejího fungování,
- komparace achitektury Flux s možnými alternativami,
- testování vytvořené aplikace.

Metodika

Metodika diplomové práce se zakládá na studiu odborných informačních zdrojů a jejich aplikaci v praxi. Z toho vychází zpracování teoretické části práce pomocí literární rešerše. Na teoretickou část práce navazuje část praktická, která využívá získané poznatky při návrhu a implementaci webové aplikace pomocí metod softwarového inženýrství. Stanoven bude návrh pro další rozšíření aplikace a na základě syntézy teoretických poznatků a výsledků praktické části práce budou formulovány závěry práce.

Doporučený rozsah práce

60-80

Klíčová slova

React JS, Javascript, REST, Webová služba, Flux

Doporučené zdroje informací

BANKS, Alex; PORCELLO, Eve. Learning React: Functional Web Development with React and Redux. 2017.

CROCKFORD, Douglas. JavaScript: The Good Parts: The Good Parts. " O'Reilly Media, Inc.", 2008.

FOWLER, Martin. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional, 2004.

KLEPPMANN, Martin. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. " O'Reilly Media, Inc.", 2017.

PUREWAL, Semmy. Learning Web App Development: Build Quickly with Proven JavaScript Techniques. " O'Reilly Media, Inc.", 2014.

VRANA, Ivan. Projektování informačních systémů s UML. Česká zemědělská univerzita, Provozně ekonomická fakulta, 2008.

Předběžný termín obhajoby

2018/19 LS – PEF

Vedoucí práce

Ing. Jan Tyrychtr, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 24. 1. 2019

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 1. 2019

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 21. 02. 2019

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Návrh a implementace webové aplikace s využitím React JS" jsem vypracoval samostatně pod vedením vedoucího diplomové 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 diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 27. 3. 2019 ______________________________

Poděkování

Rád bych touto cestou poděkoval Ing. Janu Tyrychtrovi Ph.D. za odborné vedení a pomoc při zpracování diplomové práce.

Návrh a implementace webové aplikace s využitím React JS

Abstrakt

Tato diplomová práce se zabývá problematikou návrhu a implementace moderních webových technologií v knihovně React JS. Hlavní cíl práce je představení moderních technik používaných pří vývoji aplikací na straně klienta. Čtenář se v práci seznámí s problematikou vývoje webových aplikací, která zahrnuje termíny single-page aplikace, server-side rendering či reaktivní programování. Dále jsou popsány jednotlivé návrhové vzory, které se aplikují v moderních projektech. Následně jsou představeny jednotlivé jazyky, knihovny a frameworky použité v praktické části práce. Praktická část je řešena jako návrh a implementace webové aplikace. Nejprve je provedena analýza požadavků, která vycházela z uživatelských potřeb, získaných diskuzí se zadavatelem aplikace. Následně je proveden grafický návrh aplikace v podobě skici, drátěného modelu a finálního grafického návrhu aplikace. Poté byla provedena komparace návrhových vzorů pomocí kvalitativní vícekriteriální analýzy variant. Ve výzkumu byly použity metody focus group, Saatyho metoda vzájemného porovnání a metoda pořadí. Výsledkem bylo vybrání nejvhodnějšího návrhového vzoru, který byl aplikován při vývoji aplikace. Následně proběhl samotný vývoj webové aplikace a jeho následovné otestování.

Výsledkem práce je popis jednotlivých částí procesu návrhu a implementace moderní webové aplikace v knihovně React JS.

Klíčová slova: React JS, JavaScript, REST, Webová služba, Flux

Design and implementation of a web application using React JS

Abstract

This diploma thesis deals with the design and implementation of modern web technologies in the React JS library. The main goal of the work is to introduce modern techniques used in client application development. The reader familiarizes himself with the problem of web application development, which includes single-page application terms, server-side rendering and reactive programming. The thesis also describes individual design patterns that are applied in modern projects. Subsequently, the individual languages, libraries and frameworks used in the practical park of the thesis are presented. The practical part is solved as web application design and implementation. First, a requirement analysis based on the requirements of the discussions with the developer of the application was performed. Consequently, the graphic design of the application is made in the form of a sketch, a wireframe and the final graphic design of the application. Then a comparison of design patterns was performed using qualitative multicriterial analysis of variants. The focus group, Saaty method of mutual comparison and method of order were used in the research. The result was the selection of the most appropriate design pattern that was applied when developing the application. Subsequently, the development of the web application itself and its subsequent testing took place.

The result of the thesis is a description of the individual parts of the process of design and implementation of a modern web application in the React JS library.

Keywords: React JS, JavaScript, REST, Web service, Flux

Obsah

1 Úvod **12**

2 Cíl práce **13**

3 Metodika **13**

3.1 Metoda Focus group pro získání dat 14
3.2 Saatyho metoda párového porovnání 14
3.3 Metoda pořadí pro hodnocení variant 15

4 Teoretická východiska **16**

4.1 Problematika vývoje webových aplikací 16
4.1.1 Single-page aplikace 16
4.1.2 Sever-side rendering 17
4.1.3 Reaktivní programování 17
4.2 Proces vytvoření prototypu 17
4.2.1 Skicování návrhu 17
4.2.2 Drátěný model 18
4.2.3 Prototyp návrhu 18
4.3 Návrhové vzory 18
4.3.1 Architektura datových vrstev 19
4.3.2 Návrhový vzor MVC 20
4.4 Představení jazyka JavaScript 21
4.5 Překlad rozšíření jazyka JavaScript 21
4.6 Knihovna React JS 22
4.7 Transpilace zdrojového kódu 30
4.8 Představení JavaScriptu na serveru 30

5 Vlastní práce **32**

5.1 Analýza požadavků 32
5.1.1 Funkční požadavky 32
5.1.2 Nefunkční (technické) požadavky 32
5.1.3 Specifikace zadaných požadavků 33
5.1.4 Stanovení priorit 34
5.1.5 Výsledná tabulka požadavků 36
5.2 Návrh uživatelského rozhraní 37
5.2.1 Skica návrhu 37
5.2.2 Drátěný model 38
5.2.3 Grafický návrh 39

5.3 Výstupy komparace návrhových vzorů ... 39
5.3.1 Výzkumná otázka a dílčí výzkumné otázky ... 40
5.3.2 Výběrový soubor ... 40
5.3.3 Průběh focus group ... 41
5.3.4 Hodnocení focus group ... 41
5.4 Použité knihovny a frameworky ... 45
5.4.1 NextJS ... 46
5.4.2 CSS a Styled Components ... 46
5.4.3 Jest ... 47
5.5 Konfigurace projektu ... 49
5.5.1 Vývojové prostředí ... 49
5.5.2 Instalace závislostí ... 49
5.5.3 Přidání spouštějících skriptů ... 50
5.5.4 Adresářová struktura ... 51
5.5.5 Komponentová struktura ... 53
5.6 Vývoj ... 53
5.7 Testování ... 67
6 Závěr ... 69
7 Citovaná literatura ... 70

Seznam obrázků

Obrázek 1 – Single-page aplikace (2016) 16
Obrázek 2- Architektura Flux (Tilley, 2014) 19
Obrázek 3- Architektura MVC (Tilley, 2014) 21
Obrázek 4 – Ukázka použití JSX (vlastní zpracování) 23
Obrázek 5 – Výchozí stav aplikace (vlastní zpracování) 24
Obrázek 6 – Funkce pro zavolaní změny stavu (vlastní zpracování) 24
Obrázek 7 – Aktualizovaný stav aplikace (vlastní zpracování) 24
Obrázek 8 – Ukázka props v JSX (2013) 25
Obrázek 9 – Použití props v potomkovi (2013) 26
Obrázek 10 – skica aplikace (vlastní zpracování) 38
Obrázek 16 – drátěný model (vlastní zpracování) 38
Obrázek 17 – grafický návrh (vlastní zpracování) 39
Obrázek 18 – Použití Styled Components v JavaScriptu (vlastní zpracování) 47
Obrázek 19 – Ukázka snapshot testu ve frameworku Jest (vlastní zpracování) 48
Obrázek 20 – Ukázka snapshotu ve frameworku Jest (vlastní zpracování) 48
Obrázek 21 – Příkaz na aktualizování snapshotů (vlastní zpracování) 48
Obrázek 22 – Příkaz na inicializaci NPM projektu (vlastní zpracování) 49
Obrázek 23 – Prvotní podoba soubory package.json (vlastní zpracování) 50
Obrázek 24 – Příkazy na instalaci potřebných knihoven (vlastní zpracování) 50
Obrázek 25 – Skripty pro spuštění serveru (vlastní zpracování) 51
Obrázek 26 – Ukázka struktury složky components (vlastní zpracování) 52
Obrázek 27 – Ukázka importování komponenty (vlastní zpracování) 52
Obrázek 28 – Ukázka struktury po přidání dalších souborů (vlastní zpracování) 52
Obrázek 29 – Globáln styly z knihovny styled-components (vlastní zpracování) 53
Obrázek 30 – Ukázka komponenty Index (vlastní zpracování) 54
Obrázek 31 – Komponenta se strukturou obalové logiky (vlastní zpracování) 54
Obrázek 32 – Vložení globálních stylů (vlastní zpracování) 55
Obrázek 33 – Přidání globálních komponent do stránky (vlastní zpracování) 56
Obrázek 34 – Implementace MenuProvidera (vlastní zpracování) 57
Obrázek 35 – Implementace komponenty zobrazující Menu (vlastní zpracování) 58
Obrázek 36 – Funkce pro změnu stavů pro volání requestu (vlastní zpracování) 59
Obrázek 37 – Ukázka funkce useEffect (vlastní zpracování) 60
Obrázek 38 – Implementace funkce useDebounce (vlastní zpracování) 60
Obrázek 39 – Popsaná struktura dat, které přichází ze serveru (vlastní zpracování) 62
Obrázek 40 – Asynchronní funkce pro volání requestu (vlastní zpracování) 63
Obrázek 41 – Komponenta pro zobrazení listu (vlastní zpracování) 64
Obrázek 42 – Rodičovská komponenta spojující hledací pole a výsledky (vlastní zpracování) 65
Obrázek 43 – Komponenta obsluhující hledací pole (vlastní zpracování) 66
Obrázek 44 – Testovací scénář (vlastní zpracování) 67
Obrázek 45 – Výsledek testu (vlastní zpracování) 68
Obrázek 46 – Ukázka testu, který neprošel (vlastní zpracování) 68

Seznam tabulek

Tabulka 1 – Soupis primitivních PropTypes (de Sousa Antonio, 2015) 28
Tabulka 2 – Soupis kombinovaných primitivných PropTypes (de Sousa Antonio, 2015).. 29
Tabulka 3 – Soupis speciálních PropTypes (de Sousa Antonio, 2015) 29
Tabulka 4 - Vytvořená tabulka požadavků (vlastní zpracování) 36
Tabulka 5 – Saatyho matice pro stanovení vah (vlastní zpracování) 44
Tabulka 6 – Metoda pořadí s váženým vyhodnocení (vlastní zpracování) 45
Tabulka 7 – Rozdělení typů v konvenci BEM (vlastní zpracování)................................... 46

1 Úvod

Dnešní podoba webu je odlišná od nedávné minulosti. Způsob, jakým se nyní vytváří web je též odlišná. Tváří v tvář výzvám, jak řešit těžko udržovatelný imperativní kód napsaný v jQuery, museli vývojáři hledat nové způsoby řízení komplexnosti moderních uživatelských rozhraní. Zapotřebí byly nové knihovny a frameworky zabývající se vytvářením deklarativních, modulárních, rychlých a škálovatelných front-endových aplikací pomocí JavaScriptu. Web již nesloužil jako reprezentace statických dat a nároky na jeho dynamiku se stále zvyšovaly. Tyto nároky zapříčinily vznik single-page aplikací, které ve většině případech nahradili interní systémy a celé nativní aplikace. Tyto jednostránkové webové stránky nabyly popularitu díky rychlému vykreslení HTML a oddělení datové vrstvy od prezentační. Tímto principem se řídí všechny nové knihovny a frameworky. Prvním průkopníkem v JavaScript frameworcích zaměřených na vývoj single-page aplikací byl Backbone následovaný AngularJS, který byl zanedlouho převeden pod společnost Google. AngularJS byl oproti Backbone první framework, který poskytoval kompletní architekturu pro vývoj front-end aplikací. Hlavní vlastností AngularJS byl obousměrný tok dat, který umožňoval změny v reálném čase. V roce 2013 vypustil Facebook knihovnu React, která byla hned od vypuštění populární. Díky přehledné dokumentaci a obrovské komunitě se React stal jedním z lídrů v oblasti vývoji webových aplikací, který svojí jednoduchostí a flexibilitou umožňuje integraci s dalšími knihovnami a frameworky.

Pro každý projekt je důležitá fáze analýzy, která určí, jaký framework, knihovnu či návrhový vzor ve fázi vývoje použít. Výběr neoptimálních vývojářských nástrojů může zpomalit či dokonce ukončit vývoj webové aplikace a následný přepis do jiného frameworku může být velice nákladný z hlediska financí i času. Analytická fáze vývoje webové aplikace je čím dál důležitější a vzhledem k nynějšímu trendu komplexnosti webových aplikací je důležité nepodcenit výběr správných nástrojů.

2 Cíl práce

Diplomová práce je zaměřena na problematiku návrhu a implementace moderních webových aplikací. Hlavním cílem práce je návrh a implementace webové aplikace ve skriptovacím jazyce JavaScript s pomocí knihovny React JS. Dílčí cíle práce jsou:

- vymezit základní pojmy problematiky a jejího fungování
- komparace architektury Flux s možnými alternativami,
- testování vytvořené aplikace
- vyvození závěrů práce

3 Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Následuje zpracování teoretické části práce pomocí literární rešerše. V této části jsou popsána teoretická východiska vývoje webových aplikací, návrhových vzorů, zvolených programovacích jazyků a technologií, které jsou použity v praktické části.

Praktická část navazuje na část teoretickou. Nejprve je provedena analýza požadavků na webovou aplikaci, které byly získány pomocí rozhovorů se zadavatelem. Poté je proveden grafický návrh aplikace ve formátech skica, drátěný model a finální grafický návrh. V další části autor provádí komparaci vybraných návrhových vzorů pomocí kvalitativní vícekriteriální analýzy variant za účelem výběru vhodné varianty pro aplikaci. V analýze jsou využity metody focus group, Saatyho metoda a metoda pořadí. Následuje detailní popis jednotlivých částí aplikace a potřebných implementačních kroků: základní komponenty systému podle vybraného návrhového vzoru, použité knihovny a frameworky, proces konfigurace projektu a vývoj stěžejních komponent aplikace. Praktiky a principy použité při vývoji řešení vychází zejména z modulárního funkcionálního programování. Praktická část je zakončena testováním uživatelského rozhraní, stanovením návrhu pro další rozšíření aplikace a na základě syntézy teoretických poznatků a výsledků práce jsou formulovány závěry práce.

Níže jsou představeny použité specifické metody.

3.1 Metoda Focus group pro získání dat

Focus group neboli skupinový rozhovor demograficky rozmanité skupiny lidí, která se účastní řízené diskuse o konkrétním tématu před jeho realizací, nebo aby poskytovala průběžnou zpětnou vazbu o daném tématu (Focus group - Definition of focus group in US English by Oxford Dictionaries, 2017).

Metoda spadá pod kvalitativní výzkum, jehož cílem je prozkoumat sociální skutečnost odkrytím subjektivních významů (Loučková, 2010). Kvalitativní výzkum citlivě zohledňuje kontext, lokální situaci a podmínky ovlivňující zkoumanou situaci (Zohrabyan, 2014).

Focus group probíhá ve skupině 6 až 12 lidí. V této skupině se vyskytuje i moderátor rozhovoru, který celou diskuzi vede. Pokládá účastníkům otázky ohledně tématu k otestování názorů, parafrázování a jiných psychologických metod. Rozhovor se řídí podle předpřipraveného plánu a trvá 1 až 2 hodiny. Výhoda oproti individuálnímu rozhovoru je skupinový tlak, který motivuje respondenty nahlížet na téma realisticky. Dále ve skupině bývá větší množství nápadů, které pozitivně ovlivňují myšlenky účastníků. Nevýhodou tohoto výzkumu je fakt, že díky malému vzorku zkoumání nelze získané závěry zobecnit. (Loučková, 2010)

3.2 Saatyho metoda párového porovnání

Saatyho metoda je jednou z metod kvantitativního párového srovnání, která umožňuje zúčastněným vyjadřovat své preference i verbálním způsobem, který je jim často bližší oproti numerickému hodnocení (Doubravová, 2009). Získané verbální vyjádření se automaticky převádí na číselnou stupnici, kde zúčastnění hodnotí jednotlivá kritéria pomocí srovnání párů kritérií Saatyho matice. Stupnice využívá následující stupnici:

1 – i a j jsou si rovnocenná,

3 – i je slabě preferováno před j,

5 – i je silně preferováno před j,

7 – i je velmi silně preferováno před j,

9 – i je absolutně preferováno před j.

Prvky v Saatyho matici jsou rozděleny diagonálou s identickým poměrem důležitosti (vztah 1). Kritéria pod a nad diagonálou zodpovídají převrácené hodnotě. (Saaty, 2008)

3.3 Metoda pořadí pro hodnocení variant

Výběr vhodného souboru kritérií umožňuje jednoduché a jasné ohodnocení jednotlivých variant podle těchto kritérií. Pokud má rozhodovatel k dispozici ordinální informaci o kritériích, tzn., že je schopen určit pořadí důležitosti kritérií, lze použít pro stanovení vah metodu pořadí. Tato metoda spočívá v tom, že jednotlivé varianty jsou ohodnoceny čísly 1, 2, …, m, kde m je počet variant. Pokud je za nejlepší hodnocení považováno číslo m, pak je kompromisní varianta určena maximalizací součtu dílčích hodnot podle stejného vzorce jako u předchozí metody. Podle principu pořadí však může být nejlépe hodnoceno i číslo 1, potom je vybrána varianta, která dosahuje minimálních hodnot. I tuto metodu lze rozšířit o váhy kritérií. (Doubravová, 2009)

4 Teoretická východiska

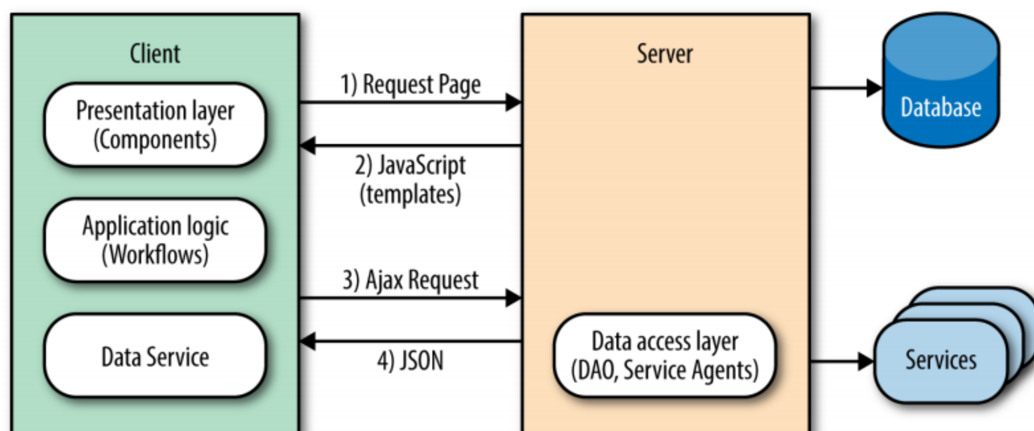
4.1 Problematika vývoje webových aplikací

4.1.1 Single-page aplikace

Single-page aplikace neboli jednostránková aplikace je způsob dodávání zdrojového kódu do prohlížeče, který stránku znovu nenačítá při uživatelově interakci. Jako ostatní aplikace i jednostránková aplikace má pomoci uživateli dokončit žádanou operaci. Flash a Java applety se progresivně vyvíjely do roku 2000. Java byla používána k poskytování komplexních aplikací, které byli schopné nahradit i kancelářské sady. Flash se stal platformou volby pro poskytování propracovaných prohlížečových her a později i videí. Na druhou stranu, JavaScript byl stále odsunut na validace formulářů, pop-up okna apod. Problém byla spolehlivost JavaScriptu, která nebyla konzistentní na populárních prohlížečích. (Mikowski, 2014)

Jak JavaScript vyspěl, většina jeho slabostí byla buď vyřešena nebo zmírněna a jeho výhody převažovaly nad nevýhodami. Mezi hlavní výhody patří (Building isomorphic JavaScript apps, 2016):

- Webový prohlížeč je nejrozšířenější aplikací na světě
- Nasazení JavaScriptu je triviální
- JavaScript je užitečný pro vývoj na různých platformách
- Podpora HTML5, SVG a CSS3 standardů
- Stolní i mobilní zařízení se staly silnějšími



Obrázek 1 – Single-page aplikace (Building isomorphic JavaScript apps, 2016)

4.1.2 Sever-side rendering

Server-side rendering neboli izomorfní JavaScript aplikace sdílí stejný kód mezi klientem prohlížeče a webovým aplikačním serverem. Takové aplikace jsou izomorfní v tom smyslu, že berou na sebe stejnou (iso) formu nebo tvar (morfózu) bez ohledu na to, na jakém prostředí se nachází, ať už je to klient nebo server. Izomorfní JavaScript je dalším vývojovým krokem v jazyce JavaScript. Tento přístup přinesla platforma node.js, která dokáže používat většinu JavaScriptových frameworků. Tím docílí vykreslování šablony na serveru naprosto stejně jako v prohlížeči. Pomocí server-side renderingu se stránka zobrazí korektně i vyhledávacím robotům nebo uživatelům s vypnutým JavaScriptem. Výhodou je výrazné zrychlení prvního načtení aplikace, kdy není nutné čekat na stažení a parsování JavaScriptového kódu prohlížečem. (Building isomorphic JavaScript apps, 2016)

4.1.3 Reaktivní programování

V oblasti výpočetní techniky je reaktivní programování paradigma orientované na datové toky a šíření změn v kódu. To znamená, že statické nebo dynamické datové toky by měly být snadno vyjádřeny a že základní model provádění změn automaticky promítne změny v datovém toku. (The Reactive programming toolkit, 2018)
Aplikace tohoto přístupu přímo nad Document Object Model je časově a výpočetně náročná. Tuto náročnost řeší mechanismus Virtual DOM, obsažený v knihovně React JS.

4.2 Proces vytvoření prototypu

Prototypování je postup vytváření úvodního modelu webových stránek, zaměřuje se především na rozmístění jednotlivých prvků na stránce, vytváření vizuálních priorit a v závěrečné fázi i provázáním jednotlivých stránek.

4.2.1 Skicování návrhu

Skicování je jednou z metod, které lze využít při návrhu nového webu. Jde o metodu ruční kresby na papír, někdy taky na tabuli. Pokud je potřeba vytvářet nový web, nebo provést jen redesign stávajícího řešení, je skicování levná metoda, která pomůže rychle definovat představy o novém webovém rozhraní. Skici je dobré vytvářet zejména v rané fázi návrhu webu, kdy je potřeba zachytit velké množství myšlenek a nápadů bez zbytečného plýtvání časem. Při skicování se vytváří více variant jednotlivých stránek a elementů,

porovnávají se a vyhodnocují klady a zápory jednotlivých prvků a následně se rozkreslují varianty, které mají největší šanci na úspěch.

4.2.2 Drátěný model

Drátěné modely, též známé jako schéma stránky nebo wireframy, jsou vizuální prezentace, která představují kostru rámce webové stránky. (Brown, 2011)

Drátěné modely jsou vytvořeny za účelem uspořádání prvků, které nejlépe splní určitý účel. Účel je zpravidla informován podnikatelským cílem a tvůrčím nápadem. Zobrazuje rozvržení stránky nebo uspořádání obsahu webových stránek, včetně prvků rozhraní a navigačních systémů a jejich spolupráci. (Garrett, 2011)

Drátěné modely se zaměřují na (Brown, 2011):

- Rozsah dostupných funkcí
- Relativní priority informací a funkcí
- Pravidla pro zobrazování určitých druhů informací
- Vliv různých scénářů na obrazovce

4.2.3 Prototyp návrhu

Webové prototypy jsou interaktivní ukázky webové stránky. Ty se často používají ke shromažďování zpětné vazby od zúčastněných stran projektu v raném životním cyklu projektu, než se projekt dostane do konečného vývoje. Webový prototyp je tedy jakákoli maketa nebo demo toho, jak bude vypadat výsledná webová stránka. V podstatě prototyp webových stránek umožňuje zúčastněným stranám projektu vidět, jak vypadá konečný produkt na počátku životního cyklu projektu. Existuje pro to mnoho důvodů (What is a website prototype?, 2017):

1. Dosáhnutí shody ohledně toho, co je a není v rozsahu projektu
2. Vytváření podpory či investice do projektu
3. Testování teorie a myšlenek týkajících se uspořádání a struktury stránky
4. Shromažďování zpětné vazby od uživatelů prostřednictvím testování použitelnosti

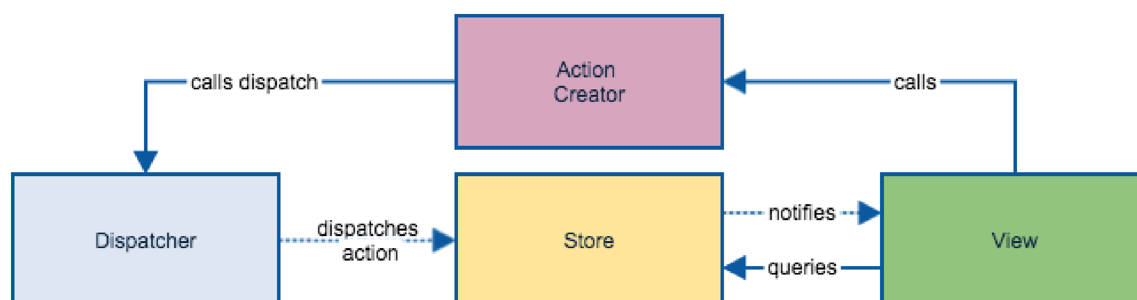
4.3 Návrhové vzory

Mezi základní pilíře architektury jakékoliv aplikace je bezesporu výběr vhodného návrhového vzoru. Primárně návrhové vzory poskytují dva hlavní benefity. Prvním jsou

osvědčená řešení, která poskytují stabilní přístupy k řešení problémů při vývoji softwaru s využitím osvědčených postupů. Druhým benefitem návrhových vzorů je jednoduché znovupoužití. Udává nám vzorec, který můžeme opakované použít a tím urychlit jednotlivé procesy. Výběrem vhodného vzoru můžeme tedy proces vývoje urychlit a standardizovat, což z větší části dokáže zamezit vzniku chyb a celkové zpřehlednění napsaného kódu.

4.3.1 Architektura datových vrstev

Flux je architektura pro vytváření datových vrstev v JavaScript aplikacích. Byla navržena společností Facebook společně s knihovnou React JS. Zaměřuje se na vytváření explicitních a srozumitelných aktualizačních cestách pro data v aplikaci, která zajišťuje zjednodušené sledování změn při vývoji (Tilley, 2014). Průchod dat v architektuře Flux je zobrazen na obrázku č. 2.



Obrázek 2- Architektura Flux (Tilley, 2014)

Je to předpisující model pro uvažování ohledně webových rozhraní. Motivací je domněnka, že rozhraní mají být předvídatelná. Měl by to být deterministický proces, který řídí, jak má být stav dané aplikace prezentován uživateli.

Základní myšlenka je, že data by měla proudit jednostranně prostřednictvím aplikace. To znamená, že akce a transformace dat mohou projít jedním nebo více dispečery a šířit se k View, ale nikdy v opačném směru. Vrstva View není určena pro modifikaci dat přímo. Vrstva musí poslat příkaz pro Dispatcher, čímž se vyvolá změna stavu, která se pak může šířit dále. S tímto přístupem, vrstva View nemá žádnou odpovědnost krom vykreslení aktuálního stavu aplikace. Nepracuje se stavem a ani by neměla (Chambers, 2014).

Dispatcher

Dispatcher je manažer celého procesu. Je to centralizovaný uzel v aplikaci. Přijímá Actions a odesílá z nich data zpět do registrovaných callbacků.

Stores

Store spravuje stav aplikace pro určitou doménu v aplikaci. To znamená, že spravuje jednu sekci aplikace, ukládá v ní data a způsoby, jak se k datům dostat.

Action Creators a Actions

Action Creators jsou kolekce metod, které jsou volány z vrstvy View aby poslaly akce do Dispatcheru. Způsob, jakým je Facebook využívá je, že konstanty typů akcí jsou použity pro definování, jaká akce by měla být použita a jsou poslány společně s daty akce. Uvnitř registrovaných callbacků, tyto akce jsou zpracovány dle jejich typu. Metody mohou být volány s daty akcí jako argumenty.

Controller Views

Controller Views jsou pouze React komponenty, které naslouchají změnám událostí a získávají stav aplikace ze Storu. Poté pošlou data jejím potomkům v props. (Wheeler, 2016)

4.3.2 Návrhový vzor MVC

Model View Controller (MVC) je model architektury softwaru, často používaný v implementaci uživatelského rozhraní. Je to tedy populární volba pro vývoj webových aplikací. Model rozděluje aplikační logiku do tří částí, podporující modularitu, snadnou spolupráci a opětovné využití. Díky tomu jsou aplikace vycházející z MVC flexibilnější (Mozilla MDN, 2005).

Controller

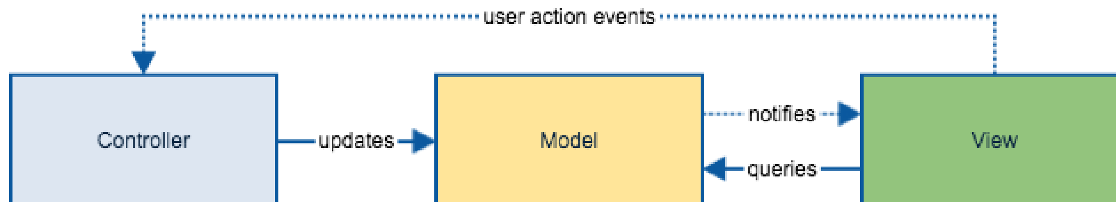
Controller je typicky nejjednodušší část modelu. Když webový prohlížeč pošle HTTP request na server, router předá tento request přiřazené controller akci. Controller přeloží request do akce, což je obvykle koordinuje databázovou akci prostřednictvím modelu a potom odešle (nebo vykreslí) odpověď do View. (Purewal, 2014)

View

View definuje, jak se mají aplikační data vykreslit (Mozilla MDN, 2005). Koncept View je nezbytný k ochraně integrity a rozsahu Modelu (Brown, 2014).

Model

Model je objektem abstrakce prvků v databázi (Purewal, 2014). Je důležité, aby modely nekontaminovaly Modely s jakoukoliv prezentační nebo vykreslovací logikou. Složitější a spornější otázka je vztah mezi Modelem a perzistentní vrstvou (Brown, 2014). Průchod dat v této architektuře je zobrazen na obrázku č. 3.



Obrázek 3- Architektura MVC (Tilley, 2014)

4.4 Představení jazyka JavaScript

JavaScript, spolu s jazyky HTML a CSS, je základní jazyk pro vývoj webových aplikací. Tento jazyk je schopný spuštění ve webovém prohlížeči, a teda je označovaný jako klientský skriptovací jazyk. Jeho nejčastější využití je možné vidět při vývoji webových aplikací. Je to jazyk založený na standardech a specifikaci ECMAScript. (Flanagan, 2011)

V JavaScriptu je možné programovat jak objektově orientovaný, funkcionální tak i procedurálně zapsaný kód. Mezi typické vlastnosti patří asynchronní zpracování, kde neblokující rozhraní orientované na události je vhodné při práci s uživatelským rozhraním.

4.5 Překlad rozšíření jazyka JavaScript

Před započetím programování v Reactu, je potřeba odpovědět na několik otázek. Jak chceme pracovat s překladem JSX a ES6? Jak budeme spravovat externí závislosti? Jak můžeme optimalizovat obrázky a CSS?

Na zodpovězení těchto otázek se objevilo již mnoho nástrojů včetně Browserify, Gulp nebo Grunt. Díky jeho funkcím a širokému využití se nejpoužívanějším nástrojem stal Webpack. (Banks, 2017)

Webpack je definován jako nástroj k vytváření JavaScript balíčků z modulárního JavaScript kódu pro použití v prohlížeči. Umožňuje tedy transformovat, zpracovat, modifikovat či zabalit do balíčku téměř jakýkoliv druh assetu jako modulu.

Konfigurační soubor Webpacku je Node.js modul, který exportuje konfigurační objekt. Můžeme v něm psát jakýkoli JavaScriptový kód interpretovatelný Node.js, což nám

poskytuje mnoho možností. Například můžeme soubory, které chceme nastavit jako vstupy Webpacku, načítat ze souborového systému dynamicky (Janča, 2017).

Konfigurační objekt obsahuje 4 nejdůležitější klíče:

- entry obsahuje nastavení vstupu Webpacku,
- output obsahuje nastavení výstupu Webpacku,
- module obsahuje nastavení práce nad moduly (zejména nastavení loaderů),
- plugins obsahuje pluginy a jejich nastavení.

Krom zmíněných vlastností dokáže Webpack i následující (Banks, 2017):

- Code splitting
- Minifikace
- Feature Flagging
- Hot Module Replacement

4.6 Knihovna React JS

React je deklarativní JavaScriptová knihovna pro tvorbu uživatelských rozhraní webových aplikací. Je důležité mít na paměti, že React slouží pouze pro vykreslení uživatelského rozhraní, nicméně se v něm nedá vytvořit plnohodnotná aplikace. K vytvoření aplikace je zapotřebí některá state management knihovna (například Redux), abychom mohli spravovat data, a také funkce pro získávání dat (například fetch) (Kolinek, 2017).

React je knihovna, nikoli framework. Důvod, proč React není často nazýván knihovnou je, že se ostatním zdá jako další alternativa k front-end frameworkům. React tedy jako knihovna může být použit v Angularu, či v dalších frameworcích. (Banks, 2017)

Rozšíření syntaxe jazyka JavaScript

React přináší rozšíření syntaxe jazyka JavaScript nazvané JSX. Je podobné šablonovacím jazykům, ale má plnou kapacitu jazyka JavaScript. JSX se kompiluje do React.createElement() metody, která vrátí JavaScript objekt nazvaný "React element".

React DOM používá camelCase konvenci pro pojmenovávání atributů místo klasických názvů v HTML. Příklad může být *tabindex*, který se v JSX použije jako tabIndex. Atribut class je psán jako *className*, protože slovo class je rezervované pro třídu

v JavaScriptu (React - A JavaScript library for building user interfaces, 2013). Ukázka použití props pro komponentu MyButton je zobrazena na obrázku č. 4.

```
<MyButton color="blue" shadowSize={2}>
 Click Me
</MyButton>

// compiles into:

React.createElement(
 MyButton,
 {color: 'blue', shadowSize: 2},
 'Click Me'
)
```

Obrázek 4 – Ukázka použití JSX (vlastní zpracování)

State (stav)

Stav State v Reactu je vestavěná možnost správy dat, které se budou měnit v rámci komponenty. Když se změní stav aplikace, uživatelské rozhraní se znovu upraví tak, aby odráželo tyto změny. Uživatele interagují s aplikací. V aplikaci hledají, navigují, filtrují, vybírají, přidávají, upravují a odstraňují. Když uživatel interaguje s aplikací, State aplikace se změní a tyto změny jsou zpět promítnuty uživateli v UI. State může být vyjádřený v React komponentách jediným JavaScript objektem. Když se, jakkoliv tento objekt změní, komponenta na to zareaguje vykreslením nového UI. (Banks, 2017)

setState

Funkce *setState* přepíše změny Statu komponenty a řekne Reactu, že tato komponenta a její potomci mají být překresleny s aktualizovaným State. *setState* je primární metoda používaná pro aktualizování uživatelského rozhraní jako odpověď na událost či odpověď ze serveru.

Je dobré o *setState* přemýšlet jako o requestu, nežli o okamžitou změnu State komponenty. Pro lepší vnímání výkonu může React zpozdit a poté aktualizovat několik komponent v jediném průchodu. React nezaručuje, že změny stavu jsou okamžitě aplikovány.

setState nemusí vždy okamžitě aktualizovat komponentu. Může aktualizovat dávkovat či zpozdit aktualizaci na později. To dělá čtení Statu ihned po volání *setState* potencionálním úskalí. Namísto toho je doporučeno používat funkci *componentDidUpdate* nebo *setState* callback, které jsou po aktualizaci použity.

setState vždy povede k překreslení, pokud metoda *shouldComponentUpdate* nevrátí *false*. Když jsou použity proměnlivé objekty a podmíněné vykreslení nemůže být implementováno v metodě *shouldComponentUpdate*, volání *setState* pouze v případě odlišnosti nového Statu z předchozího, zabrání nepotřebným překreslením. (React - A JavaScript library for building user interfaces, 2013)

Data reprezentující další State

- Callback, který se dá využít v případě potřeby ihned po aktualizování State

Představme si aktuální State v aplikaci:

```
{
  isHidden: true,
  title: 'Stateful React Component'
}
```

Obrázek 5 – Výchozí stav aplikace (vlastní zpracování)

Poté zavoláme *setState* s argumentem:

```
this.setState({ isHidden: false })
```

Obrázek 6 – Funkce pro zavolaní změny stavu (vlastní zpracování)

React sloučí předchozí a nový State dohromady a výsledkem bude:

```
{
  isHidden: false,
  title: 'Stateful React Component'
}
```

Obrázek 7 – Aktualizovaný stav aplikace (vlastní zpracování)

Vlastnost *isHidden* je aktualizovaná a vlastnost *title* není smazána či aktualizována (Fedosejev, 2015).

Props

V Reactu jsou props zkratka pro properties (vlastnosti). Jedná se o jednotlivé hodnoty nebo objekty obsahující sadu hodnot, které jsou předávány React komponentám na začátku životního cyklu pomocí konvence pojmenování podobné atributům v HTML.

Primárním účelem props v Reactu je poskytování následující funkcionality:

- Předání vlastní hodnot do komponent
- Spuštění změny state
- Použito přes this.props.reactProp v metodě render

Nejlepší je používat typ zápisu camelCase pro formátování vlastních props kvůli dodržení široce adaptovanému standartu.

Ukázka props v JSX komponentě na obrázku č. 8.

```
<Element reactProp="1" />
```

Obrázek 8 – Ukázka props v JSX (React - A JavaScript library for building user interfaces, 2013)

Tento název "reactProp" se stává vlastností připojenou k objektu nativních React props, které existují na všech součástech vytvořených pomocí knihovny React. V rámci definice React komponenty se tyto atributy stávají vlastnostmi připojenými k nativnímu objektu React props v konstruktoru komponenty. Tyto props se však stanou také dostupnými uvnitř *render* metody. Při spuštění jsou to dvě společná místa, která se použijí k přístupu nebo zachycení hodnot uložených v objektu props, které byly během vytváření předány do komponenty. Následné použití předaných props lze vidět na obrázku č. 9.

```
class Element extends React.Component {
  // Component's constructor
  constructor(props) {
    // Required to call original constructor
    super(props); // Props are now accessible from here
    const v = props.hello_i_am_a_prop;
  }
  // This is called when ReactDOM.render is called on <Element />
  render() {
    // And from here
    return <div>{this.props.hello_i_am_a_prop}</div>;
  }
}
```

Obrázek 9 – Použití props v potomkovi (React - A JavaScript library for building user interfaces, 2013)

To nám zajišťuje přístup k props při inicializaci a při každém překreslení dané komponenty (Sidelnikov, 2017).

Hooks

Hooks jsou funkce, které umožňují používat *state* a další funkce Reactu bez nutnosti psaní tříd, které byly do verze *16.7* nezbytné pro plné využití knihovny React a všech jejích životních cyklů. Hooks řeší široké spektrum zdánlivě nesouvisejících problémů v Reactu, které Facebook zaznamenal během pěti let psaní a údržby desítek tisíc komponent. Řešení se zaměřuje především na tyto problémy (React - A JavaScript library for building user interfaces, 2013):

- React nenabízí způsob, jak připojit opakovaně použitelné komponenty
- Komplexní komponenty jsou těžké na porozumění
- JavaScriptové třídy jsou matoucí pro stroje i programátory

Je několik typů Hooks, základní jsou:

useState

const [state, setState] = useState(initialState);

Tento hook vrací hodnotu *state* a funkci, která daný *state* aktualizuje. Během prvního vykreslení je vrácený *state* stejný jako hodnota předaná jako první argument. Funkce *setState* se používá k aktualizaci *state*. Přijme novou hodnotu state a spustí nové vykreslení s aktualizovanou hodnotou. Během následných překreslení bude první hodnota vrácená funkcí *setState* vždy nejaktuálnější reprezentací state.

useEffect

useEffect(didUpdate, shouldUpdate);

Přijímá funkci, která obsahuje imperativní, případně efektivní kód. Mutace, časovače, logování a další vedlejší efekty nejsou povoleny v hlavní části funkční komponenty. Místo toho se použije *useEffect*. Funkce převedená na *useEffect* bude spuštěna poté, co se komponenta vykreslí. Dá se o této funkci přemýšlet jako o únikové cestě z čistě funkčního prostředí Reactu do imperativního. Často efekty vytvářejí prostředky, které je třeba vyčistit dříve, než je komponenta zničena, například Event Listenery. Pro smazání těchto prostředků je potřeba vrátit funkci z *useEffect*.

Výchozí nastavení spustí *useEffect* při každém vykreslení. To může být v některých případech zbytečné, je tedy možné určit v jakých případech se *useEffect* má zavolat, a to pomocí druhého argumentu funkce. Do toho vložíme pole hodnot, které závisí na tom efektu a když se změní hodnota v poli, spustí se i *useEffect*. Pro spuštění *useEffect* pouze po prvním vykreslením vložíme do druhého argumentu prázdné pole.

UseContext

const context = useContext(Context);

Funkce přijímá objekt kontextu (hodnota vrácená z React.createContext) a vrátí aktuální hodnotu kontextu, kterou udává nejbližší poskytoval kontextu. Když se poskytovatel aktualizuje, Hook spustí další vykreslení s novými hodnotami (Abramov, 2018).

Mezi dodatečné patří:

- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHnadle
- useLayoutEffect
- useDebugValue

ReactDOM

React a ReactDOM byly rozděleny do dvou balíčků ve verzi 0.14 kvůli možnosti využití výhod Reactu i v React Native, což je verze Reactu upravená pro nativní aplikace Android a iOS. Jelikož jak Android, tak iOS neobsahují DOM, bylo třeba pro zjednodušení ReactDOM oddělit do vlastního balíčku.

Validace atributů

JavaScript je volně psaný jazyk, což znamená, že proměnné mohou změnit datový typ pro jejich hodnoty. Může se například nejprve nastavit proměnná jako řetězec a později změnit její hodnotu na matici. Neefektivní spravování datových typů proměnných může vést k prodloužení času potřebného k řešení chyb (Banks, 2017).

V aplikaci můžeme zachytit hodně chyb s Typechecking. Pro aplikace se dají využít rozšíření JavaScriptu jako Flow od Googlu či TypeScript od Microsoftu. React má vlastní způsob, jak kontrolovat typy proměnných, a to PropTypes. Princip funguje na přidání vlastnosti propType pro určitou React komponentu.

PropTypes exportují řadu validátorů, které mohou být využity k zajištění příjmu validních dat. Když je přijata nevalidní prop, PropTypes upozorní vývojáře v JavaScriptové konzoli. Kvůli optimalizaci, PropTypes se porovnávají pouze ve vývojovém prostředí.

Primitivní PropTypes

Tabulka 1 – Soupis primitivních PropTypes (De Sousa Antonio, 2015)

Validátor	Popis
array	Prop musí být pole
bool	Prop musí být bool
func	Prop musí být funkce
number	Prop musí být číslo nebo hodnota, která lze být parsována jako číslo
object	Props musí být objekt
string	Prop musí být řetězec

Kombinované primitivní PropTypes

Tabulka 2 – Soupis kombinovaných primitivných PropTypes (De Sousa Antonio, 2015)

Validátor	Popis
oneOfType	Prop musí být jeden z typů: *oneOfType([string, number, instanceOf(Message)])*
arrayOf	Prop musí být matice s určitým typem: *arrayOf(number)*
objectOf	Prop musí být objekt s určitým typem: *objectOf(number)*
shape	Prop musí být objekt s určitou strukturou a typy: *shape({color: string, color: string, fontSize: number})*

Speciální PropTypes

Tabulka 3 – Soupis speciálních PropTypes (De Sousa Antonio, 2015)

Validátor	Popis
node	Prop musí být renderovatelná hodnota: číslo, znakový řetězec, element nebo matice
element	Prop musí být React element
instanceOf	Prop musí být instance dané třídy
oneOf	Prop musí být jedna hodnota z povolených hodnot: oneOf(['News', 'Photos']).

Výchozí atributy

Další způsob, jak zlepšit kvalitu komponent je nasazení *DefaultProps*. Validace probíhá tak, že pokud Prop není poskytnuta, použije se tato výchozí hodnota (Banks, 2017). Výchozí props se nastavují pomocí objektu *defaultProps* (React - A JavaScript library for building user interfaces, 2013).

4.7 Transpilace zdrojového kódu

Většina programovacích jazyků dovoluje kompilování zdrojového kódu. JavaScript je interpretovaný jazyk, prohlížeč tedy interpretuje kód jako text a není tedy potřeba kód kompilovat. Prohlížeče ale nepodporují nejnovější syntaxi ES6 a ES7 a JSX. Protože chceme však používat nejnovější vlastnosti JavaScriptu společně s JSX, potřebujeme způsob, jak konvertovat náš kód, tak aby mu prohlížeč rozuměl. Tento proces je nazván transpilace a Babel je transpilátor k tomu určený. Sebastian McKenzie je jeho autorem. První verze projektu se nazývala 6to5, a byla vypuštěna v roce 2014. 6to5 byl nástroj, který se používal pro konverzi ES6 syntaxe do ES5, která je široce podporována prohlížeči. Jak projekt rostl, zaměřil se na podporu veškerých změn v ECMAScriptu. Přidal také podporu JSX do čistého Reactu. Projekt byl přejmenován na Babel v únoru roku 2015. Babel je používán společnostmi Facebook, Netflix, PayPal, Airbnb a mnoho dalších. Dříve Facebook vytvořil JSX transformátor, který byl jejich standardem, ale později byl nahrazen Babelem.

4.8 Představení JavaScriptu na serveru

Node.js je run-time prostředí na více platformách, které umožňuje vývojářům vytvářet s pomocí JavaScriptu serverové i síťové aplikace (Mozilla MDN, 2005). Node.js vytvořil v roce 2009 Ryan Dahl, jeho následný rozvoj byl sponzorován firmou Joyent, jeho zaměstnavatelem. Node.js se skládá z V8 JavaScript engine. V8 je vysoce výkonný JavaScript a WebAssembly engine od společnosti Google napsaný v jazyce C++. Je použit v prohlížeči Google Chrome v Node.js. Implementuje ECMAScript a WebAssembly and podporuje chod na Windows 7 nebo novější, macOS 10.12+ a Linux distribuce, které používají x64, IA-32, ARM nebo MIPS procesory. V8 může pracovat samostatně nebo může být vložen do libovolné aplikace C ++ (Chrome Developers, 2008).

JavaScript pro své webové aplikace používá například Google, Facebook, Twitter a GitHub. Jelikož Node.js používá JavaScript na straně serveru, přináší tím tyto výhody (Hejtmánková, 2016):

- Vývojáři mohou psát webové aplikace v jednom jazyce, což zjednodušuje vývoj a umožňuje například znovupoužití stejného kódu (mezi serverem a klientem).
- Pro vzájemnou výměnu dat používá Node.js formát JSON, který pochází z JavaScriptu.

- JavaScript se používá v dokumentově orientovaných databázích (MongoDB, CouchDB) a propojení s nimi je velmi snadné.
- Node.js používá „engine“ V8, který dodržuje standard ECMAScript, což zaručuje plnou podporu kódu ze strany všech aktualizovaných prohlížečů.

5 Vlastní práce

5.1 Analýza požadavků

Pro aplikaci, která má mít úspěch mezi uživateli je nutná analýza požadavků, které budou na aplikaci kladeny. Aktivity spojené s analýzou požadavků byly rozděleny do tří fází. V první fázi se autor zaměřil na sběr požadavků, který probíhal diskuzí s identifikovanými uživateli za účelem získání jejich požadavků na aplikaci. Získané požadavky bylo potřeba v druhé fázi projít a určit, zda jsou jasné, kompletní a nerozporují si mezi sebou. Takto identifikované požadavky byly v poslední fázi autorem zaznamenány a zdokumentovány do oblastní funkční a nefunkční. Každému požadavky byla také přiřazena priorita.

5.1.1 Funkční požadavky

Po diskuzi se zadavatelem byly stanoveny následující funkční požadavky na aplikaci:

- Přehled nejčetnějších dotazů
- Hledání souvisejících nejčetnějších dotazů dle kritéria
- Exportování výsledků do souboru
- Výběr sledovaného období
- Geografické filtrování výsledků

5.1.2 Nefunkční (technické) požadavky

Na funkční požadavky bylo navázáno požadavky nefunkčními, které se zaměřují na technický aspekt aplikace. Stanoveny byly tyto požadavky:

- Frontend aplikace bude napsán v jazyce JavaScript s pomocí knihovny React JS
- Aplikace bude dostupná pouze uživatelům z vnitřní sítě (či IP)
- Aplikace bude renderovat na serveru
- Komponenty budou mít napsané snapshot testy

5.1.3 Specifikace zadaných požadavků

Dalším krokem bylo detailnější popsání jak funkčních, tak nefunkčních požadavků. Popis všech požadavků je níže.

Přehled nejčetnějších dotazů

Webová aplikace musí zobrazovat základní přehled dotazů, které jsou nejčetnější v určitém časovém období. Dotazy musí být zobrazeny v přehledném seznamu, který bude očíslovaný. Dále musí být zobrazen konkrétní dotaz a jeho četnost v určitém časovém období.

Hledání souvisejících nejčetnějších dotazů dle kritéria

Webová aplikace musí umožnit uživateli hledat související nejčetnější dotazů dle vlastně hledaného dotazu. Vložení dotazu musí být implementováno pomocí hledacího pole. Po vložení dotazu musí být bez potvrzení automaticky zahledán výsledek.

Exportování výsledků do souboru

Každý výsledek musí být možné jednoduše exportovat do souboru formátu PDF. Obsah souboru musí obsahovat:

- Očíslovaný list dotazů
- Hledaný dotaz
- Datum a čas hledání
- IP adresa uživatele
- Odkaz na webovou aplikaci

Soubor musí mít v názvu:

- Název aplikace "Seznam Trends"
- Hledaný dotaz
- Datum a čas hledání

Výběr sledovaného období

Webová aplikace musí umožňovat vlastní výběr období, ve kterém budou nejčetnější dotazy hledány. Rozhraní pro výběr musí umožňovat výběr pomocí:

- Specifického rozmezí dvou datumů
- Rychlý výběr po hodinách a dnech od okamžiku filtrování

Geografické filtrování výsledků

Webová aplikace musí poskytovat uživateli možnost filtrování výsledků dle geografických údajů, ve kterých byly dotazy zadány. Geografické údaje musí být dvou skupin:

- Tuzemsko a zahraničí
- Jednotlivé kraje

Frontend aplikace bude napsán v jazyce JavaScript s pomocí knihovny React JS

Zdrojový kód aplikace bude napsán v jazyce JavaScript. Aplikace bude používat knihovnu React JS v aktuální verzi.

Aplikace bude dostupná pouze uživatelům z vnitřní sítě (či IP)

Z důvodů zachování privátní informace, aplikace bude dostupná pouze po přístupu z vnitřní sítě. Pro přístup do aplikace mimo síť bude potřeba použití virtuální privátní sítě. Dotazujícím se na webovou aplikaci mimo interní síť musí být vstup znemožněn.

Aplikace bude renderovat na serveru

Webová aplikace napsané v knihovně React JS bude renderovat HTML na straně serveru.

Komponenty budou mít napsané snapshot testy

Jednotlivé komponenty v knihovně React JS budou mít napsané snapshot testy pro jednoduché testování změn v kódu.

5.1.4 Stanovení priorit

Specifikované požadavky je třeba prioritizovat z důvodu udržení rozsahu, dodržení rozpočtu a časového rámce projektu. Pro určení priorit pro vybrané požadavky byla vybrána metoda MoSCoW, která poskytuje efektivní počet stupňů priorit. Dále přiřazuje slovní význam jednotlivým stupňům, který odstraňuje jejich abstraktnost. Název metody je odvozen z pojmenování jednotlivých priorit požadavků, které jsou seřazeny podle důležitosti. Po diskuzi se zadavateli byly požadavky rozděleny následovně:

Must have – musí mít

- Přehled nejčetnějších dotazů

- Frontend aplikace bude napsán v jazyce JavaScript s pomocí knihovny React JS
- Aplikace bude dostupná pouze uživatelům z vnitřní sítě (či IP)

Should have – mělo by mít

- Hledání souvisejících nejčetnějších dotazů dle kritéria
- Aplikace bude renderovat na serveru

Could have – bylo by dobré, kdyby mělo

- Výběr sledovaného období
- Exportování výsledků do souboru
- Komponenty budou mít napsané snapshot testy

Won't have this time – zatím nebude mít

- Geografické filtrování výsledků

5.1.5 Výsledná tabulka požadavků

Po rozdělení požadavků na funkční a nefunkční následovaný popisem požadavku se stanovila výsledná priorita každého požadavku. Všechny zjištěné údaje po požadavky byly zaznamenány do tabulky požadavků seřazené dle priorit od nejdůležitějších po nejméně důležité. Nejdůležitějším požadavkem vyplynul hlavní cíl aplikace, a to přehled nejčetnějších dotazů. Požadavek, který se dle priorit zatím splnit nemá, je geografické filtrování výsledků. Přehled všech priorit a jejich typu a priority je zobrazen v tabulce č. 4.

Tabulka 4 - Vytvořená tabulka požadavků (vlastní zpracování)

ČÍSLO	NÁZEV	TYP	PRIORITA
1	Přehled nejčetnějších dotazů	Funkční	musí mít
2	Frontend aplikace bude napsán v jazyce JavaScript s pomocí knihovny React JS	Nefunkční	musí mít
3	Aplikace bude dostupná pouze uživatelům z vnitřní sítě (či IP)	Nefunkční	musí mít
4	Hledání souvisejících nejčetnějších dotazů dle kritéria	Funkční	mělo by mít
5	Aplikace bude renderovat na serveru	Nefunkční	mělo by mít
6	Výběr sledovaného období	Funkční	bylo by dobré, kdyby mělo
7	Exportování výsledků do souboru	Funkční	bylo by dobré, kdyby mělo
8	Komponenty budou mít napsané snapshot testy	Nefunkční	bylo by dobré, kdyby mělo
9	Geografické filtrování výsledků	Funkční	zatím nebude mít

5.2 Návrh uživatelského rozhraní

Grafické uživatelského rozhraní hledá řešení požadavků uživatelů na aplikaci, efektivity, intuitivního rozhraní a zachování firemního designu. Požadavky jsou již shrnuty v kapitole analýza požadavků.

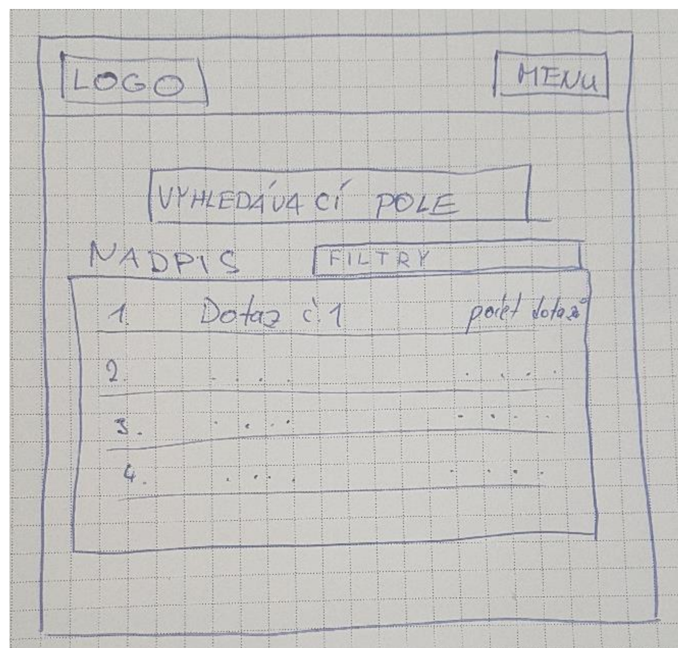
Návrhy uživatelského rozhraní byly vytvořeny v aplikaci Figma. Figma je kolaborační UX design nástroj. Je postaven na platformě Electron a díku tomu podporuje desktop aplikace pro operační systémy Windows, MacOS a Linux. Dále také disponuje webovým rozhraním, ve kterém byly návrhy vytvořeny.

5.2.1 Skica návrhu

První krok v návrhu je vytvoření skici, která je navržena společně se zadavatelem. Zadavatel potřebuje čtyři základní prvky v uživatelském rozhraní:

1. Menu
2. Vyhledávací pole
3. Filtry
4. Seznam dotazů

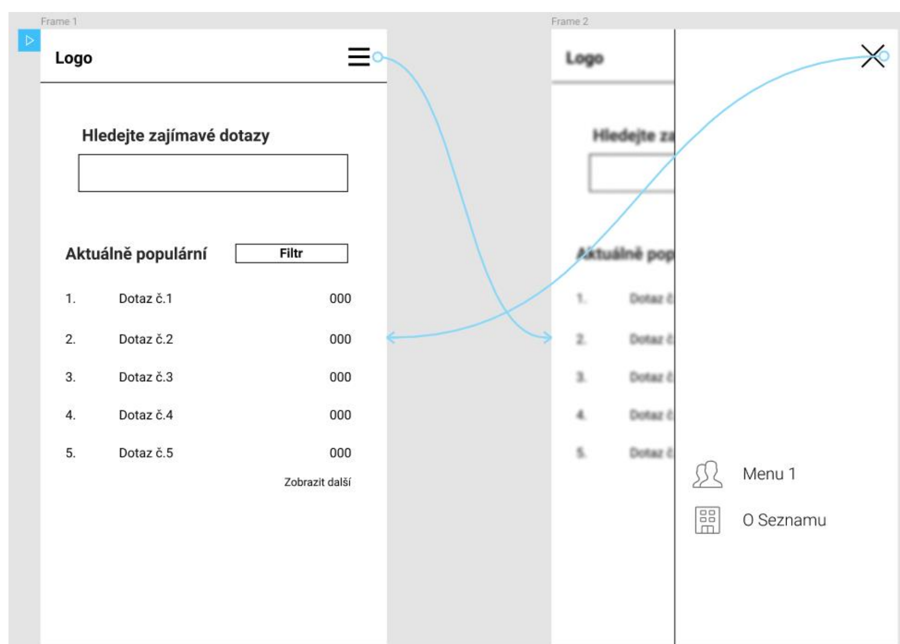
Navrhnutí skici trvalo méně než třicet minut a pomohlo jak zadavateli, tak programátorovi v ucelení využití aplikace a uvědomění si určitých krajních případů, které nebyly zřejmé z požadavků. Navrhnutá skica je zobrazena na obrázku č. 15.



Obrázek 10 – skica aplikace (vlastní zpracování)

5.2.2 Drátěný model

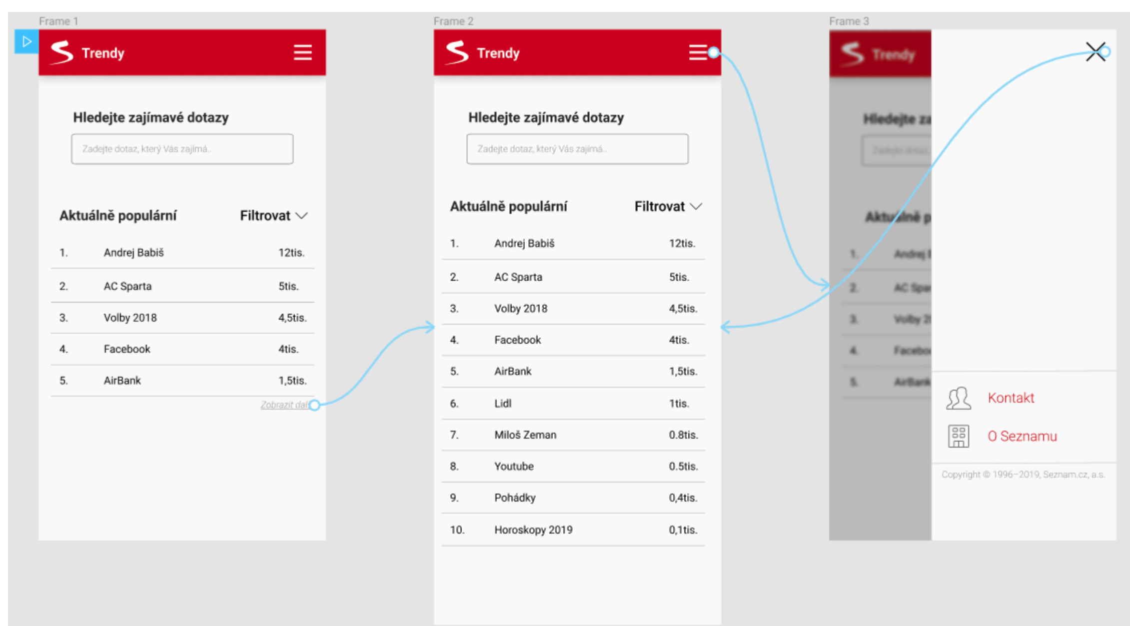
V drátěném modelu byly zpracovány dvě obrazovky, které na sebe navazují. První obrazovkou je úvodní strana, kterou uživatel uvidí po příchodu na stránku a která obsahuje stěžejní prvky navrhnuté ve skice. Druhá obrazovka obsahuje otevřené menu, které se otevře po rozkliknutí tlačítka menu z první obrazovky. Vytvořený drátěný model je vidět na obrázku č. 11.



Obrázek 11 – drátěný model (vlastní zpracování)

5.2.3 Grafický návrh

Po diskuzi drátěného modulu se zadavatelem byl zpracován finální grafický návrh webové aplikace. Ten se opírá o firemní design podle sady nástrojů, která byla vytvořena pro všechny návrháře a vývojáře. Vytvořený návrh obsahuje oproti drátěnému modelu ještě třetí obrazovku, která zobrazuje podobu aplikace po kliknutí na tlačítko "Zobrazit další". Tento návrh je zobrazen na obrázku č. 12.



Obrázek 12 – grafický návrh (vlastní zpracování)

5.3 Výstupy komparace návrhových vzorů

Komparace návrhových vzorů byla provedena pomocí kvalitativní výzkumné metody Focus group. Ta byla provedena za účelem určení kritérií, na které byla posléze aplikována Saatyho metoda z důvodů zjištění preferencí mezi jednotlivými kritérii. Výzkum byl orientován na dvě skupiny osob, a to na skupinu softwarových vývojářů a skupinu softwarových architektů. Mezi komparované varianty byly zařazeny návrhové vzory Flux a MVC. Další návrhové vzory nebyly do komparace zařazeny z důvodu nedostatečného počtu kvalifikovaných osob. Nebyla by tím zajištěna dostatečná diverzifikace v diskuzi. Mezi další návrhové vzory, které by mohly být komparovány, ale nebyly připuštěny, patří architektury MVVC a MVP.

5.3.1 Výzkumná otázka a dílčí výzkumné otázky

Výzkumné otázky se zaměřovaly na názory, zkušenosti a priority vybraných skupin. Pro zpracování komparace byla položena hlavní výzkumná otázka a 3 dílčí výzkumné otázek. Hlavní výzkumná otázka byla "Návrhová architektura Flux je vhodnější pro webovou aplikaci, která je napsaná pomocí knihovny React JS, oproti architektuře MVC". Z této otázky byly odvozeny následující dílčí výzkumné otázky:

- Flux více zjednodušuje hledání chyb oproti MVC
- Flux dbá více na oddělení prezentační a business logiky oproti MVC
- Flux je lépe škálovatelný oproti MVC

Otázky byly sestaveny a koncipovány tak, aby umožnily získat názory respondentů ohledně předních aspektů vývoje front-endové aplikace.

5.3.2 Výběrový soubor

Obsazení focus group byla vedena jediným moderátor, nebyl přítomen pomocný moderátor ani pozorovatel.

Výzkum se zaměřil na dvě cílové skupiny, které jsou na sebe profesně navázané. Byly to skupiny softwarových vývojářů a softwarových architektů. Výběr zmíněných skupin byl záměrný s cílem porovnat názory odborníků, kteří se zabývají návrhem softwarových komponent a odborníky specializující se na jejich samotnou implementaci. Z tohoto výběru vycházely názory jak praktického, tak i teoretického rázu.

Pro obě cílové skupiny byly zavedeny kritéria pro účast ve focus group, které měly za cíl vyfiltrovat nevyhovující respondenty. Ti by mohli negativně ovlivnit průběh výzkumu a tím zkreslit jeho výsledky.

Podmínky pro účast softwarových vývojářů:

- Alespoň dvouleté zkušenosti s vývojem softwaru,
- Alespoň jeden rok zkušeností s vývojem front-end softwaru,
- Zkušenosti s více jak jednou kódovou základnou,
- Znalost českého jazyka na komunikační úrovni

Podmínky pro účast softwarových architektů:

- Alespoň pětileté zkušenosti s návrhem softwarové architektury,
- Není nadřízený některého softwarového vývojáře, který se účastní výzkumu,

- Znalost českého jazyka na komunikační úrovni
- Zkušenosti s návrhem softwarového architektury na front-endové straně

Pro skupinu softwarových architektů bylo osloveno celkem 8 architektů, kteří splňovali zadané podmínky. Podmínky nesplňovali 2 architekti, a to podmínku o neúčasti vývojářů, kteří jsou jejich podřízení. Tato podmínka cílila na možné ovlivnění chování podřízeného vývojáře. S účastí souhlasilo 6 z oslovených architektů. Nevýhodou bylo nezajištění fyzické přítomnosti všech architektů v jedné místnosti, jelikož 2 architekti jsou z Brna a zbytek z Prahy. Propojení proběhlo za pomoci videokonference. Tento postup nijak nenarušil průběh focus group.

Skupiny softwarových vývojářů se účastnilo 8 vývojářů z celkových 13 oslovených. Podmínky splňovali všichni oslovení. Stejně jako u skupiny softwarových architektů nebyla zajištěna účast všech respondentů z této skupiny v jedné lokalitě. Též byla využita videokonference k zajištění komunikace v reálném čase.

5.3.3 Průběh focus group

Před realizací focus group s výběrovým souborem bylo provedeno zkušební dotazování s juniorními front-end vývojáři. To si kladlo za hlavní cíl ověření relevantnosti výzkumných otázek. Mezi dílčí cíle patřilo ověření časové náročnosti focus group a připravenosti moderátora na měřené dotazování. Výsledek zkušebního dotazování napomohl k určení pořadí jednotlivých výzkumných otázek, které po změně na sebe více navazují a napomáhají respondentům přemýšlet o tématu jako o reálném projektu.

Měřená focus group se uskutečnila v březnu 2019 v rozmezí jednoho týdne. Časové rozmezí bylo dáno volným termínem, který vyhovoval všem respondentům v dané skupině. Mezi jednotlivými focus group nebyly výzkumné otázky změněny kvůli případnému zkreslení výsledků. Celkový čas v obou dotazování byl 2 hodiny a 15 minut. Focus group se skupinou vývojářů trval 1 hodinu a 10 minut. Skupina architektů byla dotazována 1 hodinu a 5 minut.

5.3.4 Hodnocení focus group

Zhodnocení bylo provedeno pro jednotlivé skupiny a následně i v souvislosti obou skupin dohromady.

První hodnocená skupina byla skupina softwarových architektů. Tato skupina se více zaměřila na teoretickou podstatu dotazovaného problému. Mezi často zmiňované odlišnosti obou vzorů patřily především faktory spjaté s návrhem softwarové aplikace, tj. oddělení prezentační a business logiky, škálovatelnost, propojení jednotlivých komponent a náročnost návrhového schéma. Mezi architekty byl slabě preferován návrhový vzor Flux pro front-endovou aplikaci oproti MVC.

Skupina vývojářů se více zaměřila na reálné problémy a praktické ukázky problémů, spjatých s výzkumnými otázkami. Vyskytlo se několik situací, kde byl rozebíraný problém slovně demonstrován na určitém projektu spolu s jeho implementačními nedostatky. To rozšířilo diskuzi do hloubky a přidalo náměty k názorům i pro méně zkušené vývojáře. Výrazněji diskutované byla témata ohledně rychlosti vývoje, řešení chyb, dokumentací a znovu použitelnosti jednotlivých komponent. Skupina vývojářů silně preferovala architekturu Flux oproti MVC.

Obě skupiny, nezávisle na sebe, tedy spíše preferovali návrhový vzor Flux pro front-endové aplikace oproti vzoru MVC. Dle obou diskuzí byly stanoveny kritéria pro výběr návrhového vzoru, které jsou dle zúčastněných skupin zásadní pro vývoj moderní front-endové aplikace. Mezi těmito kritérii byly za pomoci diskuze určeny vazby preferencí, kterým byla přidělena váha dle Saatyho metody párového srovnání. Pro využití této metody byl počet preferencí ustanoven na 6. Jednotlivá kritéria byla srovnána v párech Saatyho matice, která využívala 5 stupňů hodnocení. Pro odhad vah zkoumaných kritérií byla touto metodou získána matice párových porovnání S_{ij}. Zjištěná kritéria pro vložení do Saatyho matice byla tato:

K1 – Znovu použitelnost komponent

Znovupoužitelnost neboli reusability komponent je hlavní praktika modulárního programování. To si zakládá na rozdělení jednotlivých sekcí a prvků do komponent, které se dají opětovně využít v jiné části aplikace. Příkladem může být komponenta pro tlačítko, které má určité styly a vlastnosti. Toto tlačítko lze distribuovat v celé aplikaci nezávisle na sebe. Vždy se pouze vytvoří nová instance dané komponenty. Výhody tohoto návrhu jsou nižší nároky na udržovatelnost, jelikož logika komponenty je pouze na jednom místě v jednom souboru. Tím se snižuje velikost codebase a zjednodušuje případný redesign nebo změna logiky dané komponenty.

K2 – Škálovatelnost

Škálovatelnost nebo schopnost rozšiřování se je žádoucí vlastnost, která reprezentuje schopnost jednoduché obsluhy či rozšíření. Rozšiřitelnost lze dělit na dva typy, horizontální a vertikální. Horizontální se věnuje kvantitativním změnám a vertikální změnám kvalitativním.

K3 – Isomorfismus

Isomorfismus ve smyslu návrhových vzorů se týká schopnosti přehledné a jednoduché stavby provázanosti aplikace, která se stará o renderování aplikace na serveru.

K4 – Směr datového toku

Kritérium pro rozlišení datového toku se zaměřuje na směr daného toku, který ovlivňuje z veliké části architekturu aplikace a práci s ní. Mezi vlastnosti, podle kterých lze rozlišovat výhody a nevýhody jednotlivých směrů patří:

- Jednoduchá představa o aktuálním stavu dat,
- Křivka učení,
- Implementační koncepty,
- Složitost kódu.

K5 – Debugging

Důležité kritérium z hlediska vývoje aplikace je pro vývojáře proces řešení chyb neboli takzvaný debugging. Určité návrhové vzory a přístupy velice zjednodušují práce hledání a opravování chyb, které vzniknou během vývoje.

K6 – Logické oddělení prezentační a business logiky

Posledním vybraným kritériem je logické oddělení kódu a jeho vnitřních částí. Především se odděluje prezentační od business logiky. Jsou tedy komponenty, které řeší zpracování dat a práci s uživatelským rozhraním a komponenty, které pouze řeší vzhled a rozmístěný uživatelského rozhraní. Výhodou oddělení těchto dvou či případně více logik je jednodušší správa kódu a menší zásahy v případném přepisu komponenty.

Saatyho matice pro stanovení vah byla sestavena s těmito preferencemi, které byly zjištěny během focus group s respondenty z obou dotazovaných skupin. Preference byly stanoveny pomocí vzájemného porovnávání vybraných kritérií. Síla těchto preferencí vychází z hodnocení variant, které respondenti uvedli při diskuzi.

Pro všechna kritéria byla následně vypočtena hodnota geometrického průměru pro dané kritérium. Tento mezikrok byl potřebný pro výpočet výsledných vah, které se rovnají podílu geometrického průměru kritéria se sumou geometrických průměrů všech vah. To bylo opět provedeno pro všechna kritéria. Správný výpočet byl ověřen součtem vah, které se správně rovnaly 1. Výslednou tabulku s aplikací Saatyho metody s vypočtenými vahami je zobrazen v tabulce č.5.

Tabulka 5 – Saatyho matice pro stanovení vah (vlastní zpracování)

	K1	***K2***	***K3***	***K4***	***K5***	***K6***	***BI***	***VI***
K1	1	3	5	1/3	1/7	1/5	0,72	**0,10**
K2	1/3	1	3	5	3	5	2,05	**0,27**
K3	1/5	1/3	1	1/7	1/9	1/5	0,24	**0,03**
K4	3	1/5	7	1	5	3	1,99	**0,26**
K5	7	1/3	9	1/5	1	7	1,76	**0,23**
K6	5	1/5	5	1/3	1/7	1	0,79	**0,10**
						Suma	*7,56*	*1,00*

Kritérium s nejnižší váhou byl s hodnotou 0,03 isomorfismus, který neměl silnější preferenci nad jiným kritériem. Následovala dvojice kritérií pro rozdělení logiky a znovu použitelnosti, kterým byla určena totožná váha 0,10. Zbylá kritéria měly určeny s mírnou odchylkou podobné váhy. Nejméně preferovaná váha ze tří byl debugging s váhou 0,23 následovaný směrem datového toku s váhou 0,26 a škálovatelností s hodnotou váhy 0,27.

Dalším krokem bylo stanovení výsledné komparace pomocí metody pořadí, která používá ordinální informace o komparovaných variantách v jednotlivých kritériích. Ordinální informace vycházela z dotazování ve focus group. Obě porovnávané varianty byly ohodnoceny buď 1 nebo 2 za každé kritérium, kde 2 znamená vyšší hodnocení oproti 1. Po aplikaci pořadí a vah kritérií bylo vypočteno celkové hodnocení dané varianty. Pro každé kritérium dané varianty byla vypočtena hodnota pomocí vynásobení hodnoty pořadí a váha konkrétního kritéria. Součet těchto hodnot pro variantu stanovil výsledné vyhodnocení. Vyhodnocení jsou reprezentována v tabulce č.6.

Tabulka 6 – Metoda pořadí s váženým vyhodnocení (vlastní zpracování)

	K1	K2	K3	K4	K5	K6	VYHODNOCENÍ
FLUX	2	2	1	2	2	1	1,863565
MVC	1	1	2	1	1	2	1,136435

Na závěr bylo provedeno porovnání výzkumných otázek a výsledků šetření. Nejprve byly zhodnoceny dílčí výzkumné otázky.

Flux více zjednodušuje hledání chyb oproti MVC

Dle respondentů ve focus group byla tato dílčí otázka potvrzena. Flux opravdu zjednodušuje proces hledání chyb a jejich oprav oproti návrhu MVC. Je to dáno především pevně daným směrem toku dat a pouze jedním zdrojem informací.

Flux dbá více na oddělení prezentační a business logiky oproti MVC

Druhá výzkumná otázka týkající se pohledu na oddělení prezentační a business logiky byla zamítnuta. Šetření vyvodilo závěry, že návrhový vzor MVC opravdu dbá a dopomáhá k přehlednějšímu oddělení prezentační a business logiky oproti vzoru Flux.

Flux je lépe škálovatelný oproti MVC

Poslední dílčí výzkumná otázka, teda otázka škálovatelnosti obou vzoru byla potvrzena. Architektura flux je tedy lépe škálovatelná oproti MVC.

Výsledné hodnocení obou návrhových potvrzuje i hlavní výzkumnou otázku. Návrhová architektura Flux s váženým hodnocením 1,8635 je tedy vhodnější pro webovou aplikaci v React JS oproti MVC s rozdílem 0.727 ve váženém hodnocení provedeném pomocí kvalitativní vícekriteriální komparace.

5.4 Použité knihovny a frameworky

Díky mohutnému rozmachu správců JavaScriptových balíčků se standardizoval proces přidání knihoven třetích stran do projektů s NPM, čímž ulehčuje a zrychluje programovací část. Je ale optimální stanovit hranici, kdy si potřebnou funkcionalitu zajistit vlastním kódem a kdy si stáhnout knihovnu, která řeší daný problém. S narůstajícím počtem knihoven v projektu stoupá i velikost výsledného JavaScriptového bundlu, který zpomaluje přenesení skriptů do prohlížeče a zhoršuje uživatelskou zkušenost. Námi napsaná funkcionalita

samozřejmě taktéž zvýší velikost, ale z pravidla méně kvůli tomu, že řeší jeden problém oproti knihovně, která většinou poskytuje vyšší míru flexibility. V případě knihoven zaměřených na konfiguraci je pro malé a středně veliké projekty lepší využít knihovnu oproti vlastní implementaci.

Pro tento projekt byly použity jen ty nejnutnější knihovny potřebné pro chod aplikace a knihovny sloužící k zjednodušení vývoje a správy kódu. Klíčové jsou knihovny react a react-dom, na jejichž základech byl celý projekt postaven. Co tyto knihovny zajišťují je popsáno v teoretické části práce.

5.4.1 NextJS

Další větší knihovna určená pro konfiguraci je next z projektu Next.js. Next.js je minimalistická knihovna pro statické aplikace vykreslené na serveru pomocí knihovny React. Poskytuje stylovaní a směrováním v základním balíku. Pro funkčnost je potřeba používat jazyk Node.js na straně serveru.

5.4.2 CSS a Styled Components

Kaskádové styly se kvůli rostoucím požadavkům na webové aplikace rychle stávají nepřehledné a náchylné na duplikaci tříd bez použití určité konvence či pravidel pro zápis tříd. Problémem je globální rozsah stylů, který nenásleduje modularitu JavaScript frameworků a knihoven pro psaní kódu. Řešením tedy může být použití konvence pro zápis CSS tříd. Mezi nejpoužívanější patří BEM, který stojí za konvencí typu *Blok, Element* a *Modifikátor*. Příklad pojmenování lze vidět v tabulce č. 7.

Tabulka 7 – Rozdělení typů v konvenci BEM (vlastní zpracování)

Typ třídy	Způsob pojmenování
Blok	.nav
Element	.nav__item
Modifikátor	.nav__item—centered

Tento způsob zajišťuje větší přehled o použitých třídách a nepřímo z jisté části řeší problém globálních kaskádových stylů. Další způsob je použití kaskádových stylů přímo v JavaScriptu. To oproti konvencím zajišťuje přímé rozdělení globálního *scopu*. Mezi přední knihovny pro tento problém patří Aphrodite, Radium a Styled Componets. Styled

Components patří k těm nejpoužívanějším z důvodů jednoduché instalace, velkému množství návodů a dokumentace a přímočarému použití pomocí *template literals, což jsou* řetězcové literály, které umožňují vložené výrazy. Pomocí nich se mohou použít řetězce s více řádky a funkční interpolace řetězců. Ukázku lze vidět na obrázku č. 13.

```
const StyledAnchor = styled.a`
    color: white;
    ${props => props.red && css`
        color: red;
    `}
`;

const Anchor = (props) => (
    <>
        <StyledAnchor href="" red>
            Cerveny odkaz
        </StyledAnchor>
        <StyledAnchor href="">
            Bily odkaz
        </StyledAnchor>
    </>
);
```

Obrázek 13 – Použití Styled Components v JavaScriptu (vlastní zpracování)

5.4.3 Jest

Každý větší projekt zaměřený na vývoj se neobejde bez testů, které se řadí mezi nejlepší praktiky vývoje softwaru. Tato praktika patří mezi ty, které se nemusí ortodoxně dodržovat na extrémní úrovni pokrytí kódu, např. v prvních fází projektu, kde se požadavky často mění. Přesto je testování velmi důležité a někdy i nepostradatelné.

Při používání knihovny React a celkově i jazyka JavaScript se nejvíce rozšířil testovací framework Jest od společnosti Facebook. Je postaven jako nadstavba dalšího dobře známého testovacího frameworku Jasmine. Testování pomocí Jestu bychom mohli rozdělit do tří segmentů, které řeší rozdílné vlastnosti JavaScriptu na straně klienta. Těmi jsou:

Snapshot testy

Snapshot testy řeší podobu reprezentace aplikace ve virtuálním DOMu a zásahy do předešlé reprezentace. Po napsání testu na určitou React komponentu a prvním spuštěním se vygeneruje soubor s příponou test.js.snap, který obsahuje zmíněnou strukturu virtuálního domu, který komponenta v daném stavu vygenerovala, jak lze vidět na obrázku č. 14.

```
it('renders correctly', () => {
    const tree = renderer.create(
        <Link page="http://www.facebook.com">
            Facebook
        </Link>
    ).toJSON();
    expect(tree).toMatchSnapshot();
});
```

Obrázek 14 – Ukázka snapshot testu ve frameworku Jest (vlastní zpracování)

Při dalším spuštění se nově vygenerovaná struktura porovná s prvně vygenerovanou a vypíše případné rozdíly. Ukázku těchto nesrovnalostí lze vidět na obrázku č. 15. Při přidání atributu, elementu, nebo změnou testovacích dat se tedy struktura liší oproti předchozímu snapshotu a Jest označený daný test za neúspěšný a zobrazí rozdíl, který způsobil neúspěch testu.

```
exports[`renders correctly 1`] = `
    <a className="normal"
        href="http://www.facebook.com"
        onMouseEnter={[Function]}
        onMouseLeave={[Function]}
    >
        Facebook
    </a>
`;
```

Obrázek 15 – Ukázka snapshotu ve frameworku Jest (vlastní zpracování)

Pokud si je programátor jistý, že daný rozdíl v snapshotech je žádaný, aktualizuje uložený snapshot pomocí příkazu zobrazeném na obrázku č. 16.

```
> npm run jest --updateSnapshot
```

Obrázek 16 – Příkaz na aktualizování snapshotů (vlastní zpracování)

5.5 Konfigurace projektu

5.5.1 Vývojové prostředí

Tato kapitola se věnuje vývojovému prostředí použitém při tvorbě webové aplikace. Celý projekt je vytvořen v operačním systému Ubuntu 18.04 LTS s pomocí Windows 10, které byly nainstalovány na virtuální stroji pomocí aplikace VirtualBox a oficiálního disku od Microsoftu určeného pro vývojáře. Hlavní motivace testování na Windows bylo ověření správného nastavení relativních cest ve složce projektu a možnost testování na prohlížečích Edge a Internet Explorer 11. Jako IDE bylo vybráno prostředí Visual Studio Code od společnosti Microsoft, které s různými rozšířeními vytváří pohodlný a uživatelsky jednoduchý editor pro vývoj. Jako správce balíčků pro JavaScript byl vybrán výchozí správce pro prostředí Node.js, a to *NPM* (Node.js Package Manager).

5.5.2 Instalace závislostí

Vývoj v moderních frameworcích a knihovnách je poměrně rychlý a snadný. Je možné vybírat z mnoha předpřipravených projektů (boilerplate), které slouží k rychlému přechodu k programování a téměř nulové konfiguraci. O podstatněji složitější je právě konfigurace projektu, která může při dlouhodobé práci na projektu urychlit jak vývoj, tak i zefektivnit dodávání kódu koncovým uživatelům. Pro tento projekt, nebyl vybrán žádná předpřipravená konfigurace, protože i tato část patří do moderního vývoje webové aplikace.

Jako první krok bylo vytvoření projektu a jeho adresářů pomocí distribuovaného systému správy verzí Github. Po přidání lokálního SSH klíče byl projekt klonován do lokálního úložiště pomocí verzovacího nástroje Git. Po propojení s verzovacím nástrojem bylo potřeba inicializovat projekt pomocí správce JavaScript balíčků NPM příkazem zobrazeným na obrázku č. 17.

```
> npm init
```

Obrázek 17 – Příkaz na inicializaci NPM projektu (vlastní zpracování)

Po vyplnění deskriptivních informací o projektu byl v root adresáři vytvořen soubor *package.json*, který slouží jako hlavní konfigurační soubor NPM. Výchozí nastavení

upřesňuje pouze název projektu, verzi, popis, autora, licenci, startovní JavaScriptový soubor a skripty. Nastavení pro aplikace lze vidět na obrázku č. 18.

```
{
    "name": "SeznamTrends",
    "version": "1.0.0",
    "description": "front-end cast trendoveho hledani v Seznam.cz",
    "main": "main.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Michal Honc",
    "license": "ISC"
}
```

Obrázek 18 – Prvotní podoba soubory package.json (vlastní zpracování)

Následovalo instalování knihoven třetích stran do projektu z oficiálního repositáře NPM, který je odkazován v souboru *.npmrc* v root adresáři. Příkazy na instalování všech knihoven je na obrázku č. 19.

```
> npm install --save react react-dom express enzyme next prop-types
    babel-core nprogress styled-components
> npm install --save-dev babel-jest jest nodemon eslint
```

Obrázek 19 – Příkazy na instalaci potřebných knihoven (vlastní zpracování)

V tuto chvíli se již v adresáři projektu nacházela nový adresář *node_modules*, která obsahuje nainstalované moduly, které lze importovat do zdrojového kódu. Nemá smysl tuto složku commitovat do Gitu, byl tedy vytvořen soubor *.gitignore*, ve kterém se tato složka označila, což zapříčiní, že při commitu bude tento adresář ignorován. Z hlediska konfigurace přibyl soubor *.eslintrc*, který slouží k sémantické validaci kódu přímo v IDE. Eslint pravidla vycházejí z pravidel, které používají vývojáři ve společnosti Airbnb.

5.5.3 Přidání spouštějících skriptů

Soubor package.json je sice už v projektu, ale není zatím možné spustit server, který by aplikaci obsluhoval a hlídal změny. Knihovna next poskytuje velice pohodlné a jednoduché možnosti spuštění serveru i pro hot reloading, které způsobí automatické přeložení změn v kódu po uložení do výsledného bundlu a automatické znovunačtení stránky a tím zrychlí vývoj aplikace. Skripty pro spuštění jsou tři, které jsou na sobě závislé, ale pouze jeden stačí spustit v konzoli. Do package.json je třeba přidat objekt, který je zobrazen na obrázku č. 20.

```
{
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  }
}
```

Obrázek 20 – Skripty pro spuštění serveru (vlastní zpracování)

5.5.4 Adresářová struktura

Adresářová struktura byla rozdělena do podsložek *static, pages, node_modules, lib, server* a *components.*

Static

Adresář static obsahuje všechny statické soubory, na které se aplikace odkazuje. Obsahuje tedy externí CSS soubory, ikony, favicon.png a obrázky. Tato složka je dostupná v cestě */static*

pages

Adresář *pages* je nezbytný pro knihovnu *next*, ta tento adresář potřebuje k zajištění automatického směrování na správné soubory. Ten přidává jednotlivé routy dle názvu souborů uvnitř adresáře. Pro root routu / je tedy potřeba přidat *index.js*, pro /contact je třeba contact.js a tak dále. To poskytuje jednoduché a rychlé nastavení základního směrování.

Node_modules

Všechny nainstalované moduly přes *npm* jsou vloženy právě do tohoto adresáře, ze kterého se jednoduše importují knihovny do celého projektu. Jak již bylo zmíněno v (4.4.2), tato složka je gitem ignorována a existuje tedy pouze na strojích, na kterých je aplikace spuštěna. O to, co a jak nainstalovat se stará *npm* s pomocí *package.json* souboru, který obsahuje list veškerých závislostí, která aplikace potřebuje.

Lib

V lib jsou uloženy čistě JavaScriptové pomocné funkce, které se používají na více místech v aplikaci.

Components

Jádro Reactu je umístěno v adresáři *components*, kam jsou uloženy všechny React komponenty použité v projektu. Komponenty jsou logicky odděleny konvencí, která využívá JavaScriptový způsob importování modulů. Pokud je funkcionalita jedné logické části

uložena pouze do jedné komponenty, tato komponenta nemá vlastní adresář a její rodičovský adresář je *components*. V případě komponenty zajišťující hlavičku stránky by struktura vypadala jako na obrázku č. 21

```
/components
    - Header.js
```

Obrázek 21 – Ukázka struktury složky components (vlastní zpracování)

Takto uložená komponenta se dá ze složky *pages* importovat jako je zobrazeno na obrázku č. 22.

```
import Header from '../components/Header';
```

Obrázek 22 – Ukázka importování komponenty (vlastní zpracování)

Pokud se hlavička rozroste například o navigační menu, vytvoříme v adresáři components podadresář Header, do kterého přesunume Header.js. V adresáři dále vytvoříme soubor index.js, který bude importovat a rovnou exportovat komponentu z Header.js. Nakonec vytvoříme nový soubor Nav.js ve složce Header, jehož struktura je zobrazena na obrázku č. 23.

```
/components
    /Header
        - index.js
        - Header.js
        - Nav.js
```

Obrázek 23 – Ukázka struktury po přidání dalších souborů (vlastní zpracování)

Jelikož lze importovat soubory *index.js* ze složek pouze pomocí odkazu na adresář, a ne na samotný *index.js*, prvotní import z adresáře *pages* bude stále validní.

__tests__

Adresář pro Jest testy je uvnitř každé komponenty a obsahuje jak testovací scénáře, tak výsledné snapshoty testovaných komponent. Specifický název adresáře je kvůli možné kolizi názvů v adresáři. Jest tedy prochází postupně všechny adresáře s tímto názvem prochází a porovnává jednotlivé scénáři uvnitř.

Styles

Styles je podřazena adresáři *components*, jelikož obsahuje informace potřebné pro vizuální stránku komponent. V této složce jsou uloženy sdílené HTML elementy stylované pomocí knihovny *styled-components.* Dále obsahuje adresář *config*, který je zobrazen na obrázku č. 24, se stejnojmenným souborem, obsahujícím objekt globálních CSS pravidel, které jsou dostupný všem stylovaným komponentám v projektu.

```
export default {
    color: {
        label: '#767676',
        success: '#DCF0E3',
        info: '#E1EBFB',
        warning: '#FCEDCF',
    },
    size: {
        paddingSmall: '2rem',
        paddingMedium: '4rem',
    },
    styled: {
        shadow: '0 16px 8px -16px rgba(0,0,0,.08), 0 1px 3px 0 rgba(0,0,0,.08)'
    }
}
```

Obrázek 24 – Globáln styly z knihovny styled-components (vlastní zpracování)

V adresáři config může být v budoucnu více takových konfigurační objektů, které by velice zjednodušili možnost nastavení jiných témat pro jednotlivé uživatele.

Server

Pro konfiguraci serverové části kódu slouží adresář server.

5.5.5 Komponentová struktura

Než začne fáze vývoje aplikace, dobrá praktika je si načrtnout stromovou strukturu komponent uvnitř projektu a popsat jaká data budou kam směřovat a s čím budou souviset. Takový hrubý náčrt dokáže odhalit případné problémy se strukturou a předejít zbytečné implementaci řešení, které je nepřehledné a zmatečné.

5.6 Vývoj

Všechny potřebné konfigurace pro první spuštění projektu jsou v této části již hotové a je možné se pustit do psaní komponent. V první řadě je třeba se zaměřit na adresář *pages*, který stojí za hlavní logikou knihovny *next*. Jak bylo zmíněno v kapitole (4.5.4), jednotlivé soubory ve složce *pages* představují jednotlivé routy, které bude aplikace obsluhovat.

/Pages

Pro root routu tedy je vytvořen soubor index.js jako funkcionální komponenta, která pouze vyrenderuje div s komponentami, které budou na hlavní stránce. V nynější podobě vyrenderuje pouze prázdný div jak lze vidět na obrázku č. 25.

```
import React from "react";
const Index = () => {
  return <div />;
};
export default Index;
```

Obrázek 25 – Ukázka komponenty Index (vlastní zpracování)

Po spuštění lokálního serveru je vidět, že server na root routu odpovídá http status kódem 200 a ve vývojářských nástrojích je viditelně přidaný prázdný div z naší index.js stránky. Server je sice funkční, ale nemá žádnou logickou strukturu, či obal, kterým by každá routa byla obalena, aby nebylo nutné duplikovat komponenty (např. menu nebo hlavičku). K tomu slouží komponenta s fixním názvem _app.js umístěna uvnitř adresáře pages. Podtržítko před názvem souboru je z důvodu přímého rozlišení souborů, které mají být routou. Bez podtržítka by byla vytvořena routa /app. Tato komponenta obsahuje kód zobrazený na obrázku č. 26.

```
import App, { Container } from "next/app";
import Page from "../components/Page";
class MyApp extends App {
  render() {
    const { Component } = this.props;
    return (
      <Container>
        <Page>
          <Component />
        </Page>
      </Container>
    );
  }
}
export default MyApp;
```

Obrázek 26 – Komponenta se strukturou obalové logiky (vlastní zpracování)

Kde App a Container jsou interní komponenty knihovny next a Page je naše ještě nevytvořená root komponenta obsahující menu, hlavičky a další komponenty, které mají být na každé routě. Na ukázce kódu je vidět, že komponenta přijímá další komponentu v props a vkládá jí do Page komponenty. Component je v tomto případě jednotlivá routa. Tedy prozatím pouze root routa index.js.

Page

Aby komponenta určená pro obalení splňovala účel, je potřeba komponenta, která bude to, co se obalí okolo každé routy. Zmíněná byla komponenta Page v adresáři components. Do té je možné vložit všechny konfigurace tak, aby byly přítomny v každé routě. Vloženy taky budou globální nastavení kaskádových stylů a tzv. Theme Provider, který poskytuje globální proměnné pro styly ve stránce zmíněné v kapitole (4.4.2). Theme Provider je jednoduchá komponenta z knihovny styled-components, která obalí všechny komponenty, které mají dědit tento konfigurační objekt. Je také potřeba nastavit takzvaný reset CSS, který přepíše nativní nastavení elementů v každém prohlížeči pro jednodušší stylování a docílení co možná nejmenších rozdílů napříč prohlížeči. To se nastaví přes objekt injectGlobal z knihovny styled-components. Kód nastavený pro aplikace je zobrazen na obrázku č. 27.

```
import { injectGlobal } from 'styled-components';
injectGlobal`
    html {
        box-sizing: border-box;
        font-size: 62.5%;
        font-family: 'Arial CE', Arial, 'Helvetica CE', Helvetica, sans-serif;
        font-weight: normal;
    }
    *, *:before, *:after {
        box-sizing: inherit;
    }
    body {
        padding: 0;
        margin: 0;
        line-height: 2;
    }
`;
```

Obrázek 27 – Vložení globálních stylů (vlastní zpracování)

Hotová komponenta s předpřipravenými komponentami pro meta tagy, menu a hlavičku společně se dvěma stylovanými komponentami StyledPage a Inner je zobrazena na obrázku č. 28.

```
class Page extends Component {
  render() {
    return (
      <ThemeProvider theme={theme}>
        <MenuProvider>
          <StyledPage>
            <Meta />
            <Menu />
            <Header />
            <Inner>{this.props.children}</Inner>
          </StyledPage>
        </MenuProvider>
      </ThemeProvider>
    );
  }
}
```

Obrázek 28 – Přidání globálních komponent do stránky (vlastní zpracování)

MenuProvider

V ukázce kódu je komponenta MenuProvider, což je React komponenta poskytující data přes Context API. Důvod proč používat Context API pro hamburger menu je z důvodu zachování vysoké flexibility a možnosti ovládání otevírání a zavírání v celé aplikaci. Řešení je tedy vytvoření komponenty MenuProvider, která inicializuje Context API a pro všechny komponenty, které jsou potomky MenuProvidera, umožní správu jeho vnitřního stavu. Tím je zachována návrhová struktura Flux, kde MenuProvider je jediný zdroj informací a po změně jeho stavu jsou všechny vnitřní komponenty upozorněny na změnu stavu. Pro použití co nejméně kódu je použita funkce useReducer, která má dva argumenty. Funkci nazvanou reducer, která řeší logiku změnu stavu a vrácení stavu nového a bere dva argumenty, aktuální stav a akci, která udává jeho typ a payload. Druhým argumentem funkce useReducer je objekt, který představuje výchozí stav, který má MenuProvider inicializovat po načtení stránky. Funkce useReducer má jako výstup pole dvou hodnot. První je vždy nový stav, který produkovala funkce reducer a druhá hodnota je funkce, která jako argument bere akci, která se vloží do reduceru. Tento výstup, který je zobrazen na obrázky č. 29, se vloží do props komponenty Context Provider z Context API s názvem value. Komponenta Context Provider dále renderuje svoje potomky.

```
import React from "react";

const MenuContext = React.createContext();
const initialState = { opened: false };

const reducer = (state, action) => {
  switch (action.type) {
    case "close":
      return { ...state, opened: false };
    case "open":
      return { ...state, opened: true };
    default:
      return initialState;
  }
};
const MenuProvider = props => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  return (
    <MenuContext.Provider value={{ state, dispatch }}>
      {props.children}
    </MenuContext.Provider>
  );
};

export { MenuContext, MenuProvider, MenuConsumer: MenuContext.Consumer };
```

Obrázek 29 – Implementace MenuProvidera (vlastní zpracování)

Tento obal je v aplikaci použit na dvou místech. V hlavičce pro zavolání funkce pro otevření menu a v menu pro zavření daného menu. Pro využití Context API v aplikaci je třeba použít funkci useContext ve funkcionální a Context Consumer v komponentě využívající třídu. V menu je použita funkcionální komponenta s využitím useContext funkce. Tato funkce vkládá jako jediný parametr Context importovaný z Menu Providera. Výstupem je objekt, který byl poslán v Menu Provider jako props s názvem value. V hlavičce je postup importování Contextu stejný, jen logika práce s Contextem je jednodušší. Po kliku na tlačítko reprezentující menu se zavolá funkce dispatch z Context API. U menu je postup složitější, protože menu jde zavřít jak pomocí tlačítka s křížkem, které reprezentuje zavření menu, tak kliknutím mimo menu do overlaye.

Tlačítko na zavření menu obsahuje Event Handler onClick, který po spuštění zavolá funkci close, která zavolá dispatch z useContext s typem pro zavření. Tím je tlačítko vyřešeno. Dále je potřeba nastavit Event Listener, který zavolá funkci na zavření menu v případě, že uživatel klikne mimo menu. Event Listener je třeba přidat v případě, kdy je JSX pro komponentu již vyrenderováno. Jelikož je tato komponenta napsaná jako funkce, musíme použít funkci useEffect, která nahrazuje metody componentDidMount a componentDidUpdate v komponentách založených na třídě. Tuto funkci, která přidá Event Listenery chceme zavolat vždy, když se změní stav menu ze stavu zavřený do stavu otevřený.

Jako druhý argument této funkce bude pole s jedním prvkem a to stavem. To řeší situaci, kdy se tato funkce zavolá. Co tato funkce řeší je, že když se zaregistruje klik na stránce, ověří se, jestli element, na který uživatel kliknul je obsažen v obalujícím elementu. Jestli není, víme, že uživatel klikl mimo menu a tím pádem můžeme menu zavřít. Potřebujeme tedy zjistit závislost dvou DOM elementů. Lze použít JavaScriptovou funkci contains, která vrátí true v případě, že element, nad kterým byla funkce zavolána obsahuje element, který se pošle jako argument. Obalující element menu uložíme pomocí funkce useRef, který spárujeme s daným elementem, pomocí props ref. Druhý element, ten, na který uživatel kliknul získáme z JavaScriptové události vyvolané kliknutím z event.target. Výsledný kód pro komponentu Menu je vidět na obrázku č. 30. Pro odstranění Event Listeneru je nakonec nutné z funkce useEffect vrátit funkci, která ho odstraní.

```
const Menu = props => {
  const modalEl = React.useRef(null);
  const { state, dispatch } = React.useContext(MenuContext);

  if (state.opened === false) return null;

  const listener = e => {
    if (modalEl && modalEl.current && !modalEl.current.contains(e.target)) {
      close();
    }
  };
  const close = () => dispatch({ type: "close" });

  React.useEffect(() => {
      window.addEventListener("click", listener);
      return () => window.removeEventListener("click", listener);
    }, [state.opened]
  );

  return (
    <StyledWrapper ref={modalEl}>
      <Button onClick={close}>
        <Icon icon="cross" el={StyledCloseButton} />
      </Button>
    </StyledWrapper>
  );
};
```

Obrázek 30 – Implementace komponenty zobrazující Menu (vlastní zpracování)

Results

Nejužitečnější komponenta v aplikaci je bezesporu list dotazů. Ten zobrazuje deset nejčetnějších dotazů dle kritéria. Kritériem je buď obecný přehled nejčetnějších dotazů či klíčové slovo, které uživatel zadá. Jednotlivé dotazy v listu jsou složeny z čísla umístění, názvu dotazu a počtu hledání. Nejdříve je potřeba si navrhnout strukturu této komponenty, která odhalí případné problémy dříve, než se napíše zdrojový kód. To může mít za následky

zvýšení kvality kódu a rychlejší implementaci. Komponenta má na starosti následující problémy k vyřešení:

- Zavolání API requestu na backend
- Upravení dat, které z API přijdou v odpovědi
- Řešení i možné chyby, která může přijít z API
- Rozdělení stavů komponenty na načítání, chyba a úspěšné načtení dat z backendu
- Vyrenderování listu s daty z backendu
- Nastylování vyrenderovaného listu
- Umožnění donačtení dalších dat při kliku na tlačítko s textem "zobrazit další"

To je příliš mnoho problémů, které kdyby řešila pouze jedna komponenta, znepřehlednila by danou komponentu. Taková komponenta by byla náročná na znovu použití a snížila by její modularitu, která je esenciální pro React. Komponenta je tedy rozdělena na dvě. První řeší business logiku a druhá vizuální stránku komponenty a její interakci s uživatelem. Souborová struktura je navržena a je tedy možné připravit návrh jednotlivých kroků k dosažení cíle komponenty.

Komponenta po inicializaci načte výchozí hodnoty do funkcí useState a pokusí se s nimi vyrenderovat prvotní podobu UI. Funkce useState komponenta potřebuje tři, které jsou vidět na obrázku č. 31. První, která pracuje s listem výsledků, které přijdou jako odpověď na API requestu. Její výchozí hodnota je prázdné pole. Druhá funkce zajišťuje práci s chybovým stavem. Ten je nastaven jako false při inicializaci. Třetí funkce je podobná druhé, jelikož zajišťuje přepínání stavu načítání. Tento stav se postupně mění při odeslání API requestu a jeho zpracování.

```
const [list, useList] = React.useState([]);
const [isError, useIsError] = React.useState(false);
const [isFetching, useIsFetching] = React.useState(true);
```

Obrázek 31 – Funkce pro změnu stavů pro volání requestu (vlastní zpracování)

Po inicializaci komponenty a prvním vyrenderováním výchozích hodnot potřebujeme zavolat API request na backend s dotazem na data. Jelikož je potřeba aby tato komponenta volala API request jak pro obecné, tak i pro specifikované data, je třeba uzpůsobit logiku tomuto faktu. Specifická data budou volána v případě, kdy uživatel vyplní hledací pole. Ihned po načtení výchozího UI je třeba zavolat zmíněný API request. K tomu

slouží funkce useEffect, která se volá v různé momenty, které specifikuje druhý argument této funkce. V případě poskytnutí prázdného pole jako druhý argument se funkce useEffect zavolá pouze jednou po první vyrenderování. Toto řešení by zamezilo dalšímu zavolání API requestu v případě, kdy uživatel zahledá dotaz pomocí hledacího pole. Je tedy potřeba vložit hodnotu, která se bude měnit vždy, když uživatel změní hodnotu hledacího pole, což je samotná hodnota hledacího pole. Tato hodnota je vidět na obrázku č. 32.

```
React.useEffect(() => {
  // Call API request
}, [props.query]);
```

Obrázek 32 – Ukázka funkce useEffect (vlastní zpracování)

Problém s touto implementací ale je, že při každé změně hodnoty hledacího pole, tj. přidání či odebrání znaku, by se ihned zavolal API request. Takové chování je uživatelsky nepřívětivé a rušivé. Krom uživatelské zkušenosti je problém i technického rázu, kdy počet requestů může zpomalit stránku a zvýšit nápor na backend server. Je tedy třeba posílat requesty až po uplynutí určitého množství času po tom, kdy uživatel přestane psát. Pro tuto funkcionalitu si vytvoříme vlastní React 60oko, který se dá použít kdekoliv v aplikaci. Tento 60oko bude funkce, která jako argumenty bude brát hodnotu a časové zpoždění v milisekundách, které určuje, jak dlouho čekat, po tom, co uživatel přestane psát, na zavolání requestu. Využijí se funkce useState a useEffect, kde useState bude mít výchozí hodnotu tu, která přijde jako první argument funkce a vracet bude změněnou hodnotu, pokud uplyne více milisekund od poslední změny, než kolik je hodnota druhého argumentu funkce. Celá logika je vidět na obrázku č. 33.

```
import React from "react";

export default function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = React.useState(value);

  React.useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);
    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}
```

Obrázek 33 – Implementace funkce useDebounce (vlastní zpracování)

Tuto funkci je potřeba importovat do naší komponenty pomocí ES6 modulů. Dále je třeba určit časovou konstantu, která se pošle funkci useDebounce a která určí, po jaké době se má zavolat další request, když uživatel dopíše dotaz. Rozšířená je hodnota 500 milisekund, kterou adoptovalo mnoho webových aplikací. Naše aplikace se bude řídit stále relevantní studií (ELLIOT, 1982). firmy IBM, která doporučuje hodnotu 400 milisekund jako časovou hranici pro interakci s uživatelským prostředím. Tato hodnota je tedy poslána jako druhý argument funkce useDebounce. První argument bude text, který se má opozdit. Vložen je tedy hledaný výraz přicházející do komponent z props. Výsledek funkce je uložen do proměnné debouncedQuery, který se bude posílat jako druhý argument funkce useEffect a tedy determinant, kdy zavolat API request. Zároveň se tato proměnná posílá v requestu jako hledaný výraz.

Pro poslání API requestu se používá nativní knihovna fetch, která nahrazuje XMLHttpRequest. Fetch poskytuje vetší flexibilitu při práci se síťovými requesty. Jelikož je JavaScript asynchronní jazyk čili po zavolání requestu kód nečeká na jeho výsledek, je zapotřebí přidat logiku, která zachytí odpověď a předá jí příslušnému handleru. Toto je možné řešit třemi způsoby, a to použitím callback, promise nebo async/await. Pořadí, ve kterém byly způsoby vypsány je též pořadí, ve kterém se začaly používat. Nejstarší je tedy použití callback, což je návratová funkce, která se zavolá po přijetí odpovědi. Toto řešení je jednoduché, ale programátorsky velice nepřívětivé. V případě mnohonásobných requestů se tyto callbacky zanořují a vytváří takzvaný callback hell, což je slovní spojení pro více než dva vnořené callbacky. Tento problém řeší promise, což je objekt, který představuje případné dokončení (nebo selhání) asynchronní operace a její výslednou hodnotu. Nejmodernější způsob je pomocí syntaxe async/await, která určí asynchronní funkcí pomocí operátoru async před funkcí. To samotné ale nestačí, je třeba ještě určit, která logika je ta, na kterou má funkce počkat. To se určí pomocí operátoru await. Tato syntaxe byla přidána v EcmaScript7 a velice se rozšířila i bez podpory většiny prohlížečů díky překladači Babel.

V naší komponentě results je využit způsob async/await, který zpřehlední kód při ekvivalentní funkcionalitě jako u callback a promise. Proměnná req slouží jako odkaz na anonymní funkci pro request, která se použije na více místech v komponentě results. Před zavoláním requestu je potřeba aktualizovat stav komponenty, konkrétně stavy načítání a chyba. V první fázi nastavíme stav načítání na true a chybu na false. To zapříčiní přerenderování komponenty, která v rozhodovací větvi vyrenderuje sub-komponentu

určenou pro načítání. Uživatel v této fázi vidí text informující ho o aktuálním stavu. Po nastavení aktuálního stavu, je zabalena v try catch bloku funkce fetch, která volá URL s debounce hodnotou. Tento fetch je zabalen do dvou klíčových slov await. První zajišťuje překlad odpovědi serveru do formátu JSON a druhý čeká na konečnou odpověď, kterou uloží do proměnné response. Komponenta očekává data ze serveru v datové struktuře popsané jazykem YAML, který lze vidět na obrázku č. 34, což je formát pro serializaci strukturovaných dat.

```yaml
data:
    - {title: string, hits: number}
meta:
    timestamp: number
    from: number
    to: number
```

Obrázek 34 – Popsaná struktura dat, které přichází ze serveru (vlastní zpracování)

Tuto odpověď je nutné dále zpracovat, a to seřazením a vytažením určitého počtu prvků. Důležitá data pro komponentu results jsou obsažena v atributu data. Je to pole výsledků obsahující nejčetnější hledání za určité časové období. Toto pole je nutné nejdříve seřadit od nejčetnějších až po nejméně četné. Seřazená data jsou již ukládána v pravidelných intervalech v databázi, ze které je back-end posílá v odpovědi na front-end. Data se v prvotní fázi projektu často mění, a to jak logika ukládání dat do databáze, tak i logika back-endu. Seřazení na front-endu je tedy bráno jako pojistka. Implementačně je seřazení provedeno pomocí metody nad objektem Array, a to sort, která jako jediný argument očekává funkci s parametry prvního a druhého prvku v poli, které porovnává. Vetší prvek je posunut v poli výše a postup se opakuje.

Dále je nutné upravit počet záznamů, které v listu komponenta zobrazí. Pole má ve výchozím nastavení dvacet prvků. Naše komponenta má zobrazit prvních pět nejčetnějších a po rozkliknutí tlačítka "Zobrazit další" rozšířit seznam výsledků o dalších pět. Z efektivních důvodů je výhodnější při requestu, který je zobrazen na obrázku č. 35, uložit již prvních deset nejčetnějších výsledků, a to pak rozdělit již v komponentě na straně front-endu. Rozdělení je provedeno pomocí metody slice, která je nad objektem Array. Metody slice vrací nové pole prvků v rozmezí indexů, které se jí pošlou jako první a druhý argument.

Po úpravě dat již změníme stav komponenty na načítání false a do useList pošleme upravené pole prvků. V případě, kdyby nastala jakákoliv chyba, bude zachycena v bloku

catch, kde nastaví stav načítání na false a na chybu na true. Na tuto změnu zareaguje uživatelské rozhraní, které vypíše chybovou hlášku a tlačítko pro opětovné poslání requestu. Mezi chyby, které mohou nastat patří interní chyba na straně serveru (kód 5xx), nedosažitelný server (kód 4xx) či chyba způsobená zastaralým prohlížečem, který nezná funkce použité v bloku try.

```
const req = async () => {
  const fetchUrl = `${url}${debouncedQuery}/repos`;
  useIsFetching(true);
  useIsError(false);
  try {
    const response = await (await fetch(fetchUrl)).json();
    const out = response.data
      .sort((a, b) => b.hits - a.hits)
      .slice(0, MAX_ITEM);
    useIsFetching(false);
    useList(out);
  } catch (e) {
    useIsFetching(false);
    useIsError(true);
  }
};
```

Obrázek 35 – Asynchronní funkce pro volání requestu (vlastní zpracování)

Po změně prvků ve stavu useList v komponentě, je komponenta překreslena s daty, které přišly ze serveru. Logika zobrazení dat už závisí na komponentě ResultsList, které se předá list v props *list*.

ResultsList

Tato komponenta ručí za správné zobrazení dat, které jí přijdou z komponenty *results*. Dále ručí za funkcionalitu rozbalování více výsledků po kliknutí na tlačítko "Zobrazit další". Ve fázi, kdy je komponenta zabalena, zobrazuje maximálně pět prvků z listu. Horní hranice počtu zobrazených prvků po rozkliknutí tlačítka na rozbalení není nijak omezena a plně závisí na rodičovské komponentě. Komponenta potřebuje uchovávat jeden stav, a to stav, zdali je komponenta rozbalena či nikoliv. K tomu slouží funkce useState s proměnnými *isExtended* a *setIsExtended*, která má výchozí hodnotu proměnné *isExtended* nastavenou jako *false*. Hodnota stavu je užita jako podmínka, zda list vykreslit celý, či snížit počet renderovaných prvků pomocí metody slice. To je veškerá logika, která je obsažena v komponentě ResultsList.

Další funkce komponenty je vykreslit list. Komponenta je rozdělena na dvě části, a to list prvků a tlačítko na rozbalení listu. Jelikož jsou tyto dva elementy odlišné, nelze je zanořit v JSX do sebe. Zanořit je můžeme do elementu typu *div*, který bude obsahovat jak

list, tak tlačítko. Tímto způsobem u menších komponent často dochází k takzvanému zamoření DOM zbytečnými elementy, které zpomalují webovou stránku a znepřehledňují hledání chyb. Jelikož dle JSX pravidel musí mít komponenta vždy jako návratovou hodnotu jeden JSX element či pole elementů, zanoříme list a tlačítko do *Fragmentu*. List využívá HTMl tag *ol* pro číslovaný list a *li* jako jednotlivé prvky listu. Pro jednoduché stylování se využije funkce z kaskádových stylů pro jednorozměrný model rozložení, která nabízí prostorovou distribuci elementů v sekvenci. Tato funkce se nazývá Flexible Box Module, obvykle označovaná jako flexbox. Hlavní výhoda flexboxu oproti dvojrozměrnému rozložení elementu je jeho jednoduchost a parciální podpora i v prohlížeči Microsoft Internet Explorer 11. V případě, že je hodnota stavu *isExtended* false, vyrenderuje se i tlačítko s textem "Zobrazit další". Toto tlačítko má navěšený event handler, a to *onClick*, který registruje jakýkoliv klik myší či klávesnice. Po kliku se zavolá anonymní funkce, která změní hodnotu proměnné *isExtended* na true. Tato logika je zobrazena na obrázku č. 36.

```
const ResultsList = ({ list }) => {
  const [isExtended, setIsExtended] = React.useState(false);
  const listToRender = isExtended ? list : list.slice(0, NON_EXTENDED_ITEM);
  return (
    <>
      <StyledList>
        {listToRender.map((item, index) => {
          return (
            <StyledListItem key={item.id}>
              <span>{index + 1}.</span>
              <span>{item.full_name}</span>
              <span>{item.size}tis.</span>
            </StyledListItem>
          );
        })}
      </StyledList>
      {!isExtended && (
        <StyledButton onClick={() => setIsExtended(true)}>Zobrazit další</StyledButton>
      )}
    </>
  );
};
```

Obrázek 36 – Komponenta pro zobrazení listu (vlastní zpracování)

SearchInput

SearchInput je komponenta, starající se o funkcionalitu hledacího pole ve stránce. Dále také o distribuci hodnoty ve hledacím poli, které slouží již popsané komponentě Results. Než se začne psát zdrojový kód komponenty, je nutné najít způsob, jakým distribuovat hodnotu ve hledacím poli do komponenty Results. Jelikož tyto dvě komponenty

nemají jasně definovat strukturu typu rodič-potomek, je nutné použít sofistikovanější způsob přeposílání dat. V aktuální strukturální podobě projektu se naskytují dvě možnosti:

1. Obalení hranice obou komponent v pomocné komponentě s funkcí useContext
2. Přidání funkce useState do komponenty index.js, která obaluje celý obsah stránky

První varianta poskytuje vyšší míru flexibility, ale také složitější implementaci, která bude méně přehledná a více náchylná na chyby při případné úpravě projektu. Implementováno by to bylo vytvořením komponenty SearchContext v adresáři /components. Tento SearchContext by se importoval do komponenty Page, starající se o rozložení top level prvků na stránce. Mezi top level komponenty patří globální styly, meta tagy, menu, a hlavička. Přidání SearchContextu do Page komponenty by snížilo přehlednost v komponentě. Varianta číslo dvě, tedy přidání funkce useState do komponenty index.js se oproti první variantě drží již vytvořených komponent a nepřidává další soubory či komponenty do projektu. Jelikož jde o jednoduchou funkci, která se potřebuje splnit ke správné komunikaci jednotlivých komponent, je výhodnější použít tento způsob přenosu dat.

Než se vytvoří soubor SearchInput.js, je potřeba přidat logiku do zmíněné komponenty index.js. K uchování hodnoty ve hledacím poli a ke změně hodnoty pole je využita funkce useState, ze které vychází dvě proměnné, *query* pro uložení hodnoty pole a *setQuery* její editaci. Obě proměnné se posílají komponentě SearchInput v props a pouze proměnná *query* se posílá komponentě Results, jelikož se o editaci hodnoty ve hledacím poli nestará. Zajímá jí pouze její hodnota. SearchInput potřebuje na druhou stranu vědět, jak hodnotu ve hledacím poli, aby jí v poli fyzicky zobrazila, tak funkci, která tuto hodnotu změnu po uživatelově interakci s hledacím polem. Výchozí hodnotou hledacího pole je prázdný textový řetězec. Komponenta Index je zobrazena na obrázku č. 37.

```
import SearchInput from "../components/SearchInput";
import Results from "../components/Results/Results";

const Index = () => {
  const [query, setQuery] = React.useState("");

  return (
    <>
      <SearchInput query={query} setQuery={setQuery} />
      <Results query={query} />
    </>
  );
};
```

Obrázek 37 – Rodičovská komponenta spojující hledací pole a výsledky (vlastní zpracování)

SearchInput komponenta je složena ze stylovaného elementu div, který obsahuje hledací pole a ikonku. Jelikož se po vložení hledaného dotazu automaticky, po vypršení funkce useDebounce, odešle API request a uživateli se načtou výsledky, není potřeba tlačítko pro potvrzení hledaného dotazu. Ikonka v tomto případě slouží k lepší orientaci uživatele na stránce a jednoznačnému určení jaký význam vyhledávací pole má. Logika práce s textovým polem v Reactu je odlišná od použití čistého JavaScriptu. Opět se dají použít dva přístupy, a to kontrolovaný a nekontrolovaný. Druhý přístup se více blíží přístupu práce s textovým polem v čistém JavaScriptu, neboli hodnota pole je zpracovávána prohlížečem a komponenta nemá o hodnotě v poli žádné informace. Tu zjistí až pomocí přidané reference na element v případě, kdy uživatel provede určitou interakci či v časovém intervalu. Nevýhoda tohoto přístupu je omezená práce s hodnotou a používání principů, které nekorespondují s filozofií Reactu. První přístup neboli přístup kontrolovaný je takový, který dává komponentě přehled o každé změně hodnoty pole. Tento přístup je náročnější na porozumění výhod, které komponentě přináší. Komponenta v jednoduchosti řídí element, určuje jeho hodnotu a rozhoduje, co se provede při změně. Změna může být přidání či odebrání znaků z pole. Změnu komponenta zaregistruje pomocí přidaného event handleru onChange, který je zavolán po změně stavu hledacího pole. Při změně se zavolá funkce s argumentem *event*, který reprezentuje objekt události. Tento objekt obsahuje informace o události a funkce, které se nad událostí dají zavolat. Dále také obsahuje všechny atributy elementu a jejich hodnoty což je pro naši komponentu podstatné. Z objektu je potřeba si tedy vytáhnout objekt obsahující hodnoty atributů. Tím je objekt *target*, který obsahuje i hodnotu *value*. Ta reprezentuje aktuální hodnotu hledacího pole, potřebnou pro aktualizování vnitřního stavu komponenty index.js. Komponentu SearchInput lze vidět na obrázku č. 38.

```
const SearchInput = (props) => (
  <InputWrapper>
      <Input
          type="input"
          value={props.query}
          onChange={(e) => props.setQuery(e.target.value)}
          placeholder="Hledejte.."
      />
      <Icon icon='search' el={StyledIcon} />
  </InputWrapper>
);
```

Obrázek 38 – Komponenta obsluhující hledací pole (vlastní zpracování)

5.7 Testování

Aplikace byla otestována pomocí knihovny Jest s použitím snapshot API metod pro testování výsledné HTML struktury a připojených event handlerů. Jelikož aplikace neobsahuje náročnější logické úkony a veškerá mutace datového stavu je obsažena na straně serveru, není proto žádoucí přidávat unit testy jednotlivých komponent na straně klienta. Unit testy v prezentační vrstvě nepřináší podstatné výhody oproti snapshot testům, které jsou prosté na vytvoření a jejich pokrytí lze rapidně zvýšit díky snapshot testům celé aplikace. Napsáním snapshot testů pro root komponentu se zároveň ošetří isomorfní část aplikace, která vyrenderuje na serveru pouze to nutné pro zobrazení stránky.

Jediná napřímo testovaná komponenta pomocí snapshot testů je ResultsList, která zajišťuje stěžejní prezentační logiku aplikace a závisí na ní celá aplikace. Jelikož tato komponenta přijímá v props pole jednotlivých záznamů, je potřeba tento list pro snapshot testy zafixovat. Kdyby byly hodnoty proměnné, při každém spuštění testů by testy neprošly kritériem, jelikož například název prvního dotazu by byl odlišný. Test tedy posílá v props fixní pole s daty a pomocí knihovny *react-test-redenrer* převádí celou komponentu na čisté JSX, které porovnává s vytvořeným snapshotem. Testovací scénář je popsán v obrázku č. 39.

```
import React from 'react';
import ResultsList from '../ResultsList';
import renderer from 'react-test-renderer';

it('ResultsList render 5 items', () => {
    const list = [
        {
            id: 'xu19das0',
            full_name: 'Dotaz 1',
            size: 12,
        },
        // ...
    ]
    const tree = renderer.create(
        <ResultsList list={list} />
    );
    expect(tree.toJSON())
        .toMatchSnapshot();
});
```

Obrázek 39 – Testovací scénář (vlastní zpracování)

Vytvořený soubor s testem je nutné spustit a samotný snapshot vytvořit. Spuštění se provádí v příkazové řádce pomocí balíku NPM. Výsledek je zobrazen v přehledné formě opět v příkazové řádce, která je zobrazena na obrázku č. 40.

```
> npm run test ./components/Results/__tests__/ResultsList.test.js
PASS  components/Results/__tests__/ResultsList.test.js
  ✓ ResultsList render 5 items (23ms)

 › 1 snapshot written.
Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 written, 1 total
Time:        2.271s
Ran all test suites matching /.\/components\/Results\/__tests__\/ResultsList.test.js/i.
```

Obrázek 40 – Výsledek testu (vlastní zpracování)

Je dobré vždy po vytvoření snapshot se pokusit kontrolovaně a záměrně upravit kód. V tom případě by měl na stejné pozici vzniknout problém v souboru se snapshotem. Tím ověříme, že snapshot funguje správně a je možné upravený kód vrátit do původního stavu. Ukázka výsledku snapshotu je vidět na obrázku č. 41.

```
 FAIL  components/Results/__tests__/ResultsList.test.js
  ● ResultsList render 5 items
    expect(value).toMatchSnapshot()
    Received value does not match stored snapshot "ResultsList render 5 items 1".
    - Snapshot
    + Received
# ...
    @@ -65,11 +65,11 @@
            <span>
              5
    -         .
    +         ,
            </span>

      36 |     );
      37 |     expect(tree.toJSON())
    > 38 |         .toMatchSnapshot();
         |          ^
      at Object.toMatchSnapshot (components/Results/__tests__/ResultsList.test.js:38:10)
```

Obrázek 41 – Ukázka testu, který neprošel (vlastní zpracování)

6 Závěr

Hlavním tématem Diplomové práce byl návrh a implementace webové aplikace pomocí knihovny React JS. Jelikož je tato knihovna dominantní v odvětví vývoje webových aplikací a zároveň toto odvětví vykazuje vysokou míru dynamiky, bylo cílem vytvořit aktuální přehled postupu návrhu a implementace webové aplikace v knihovně React JS.

V teoretické části práce byla představena problematika vývoje webových aplikací. Mezi tuto problematiku patří i grafického prototypování a návrhové vzory. Dále byly přiblíženy knihovny a frameworky spjaté s vývojem moderní webové aplikace. Detailněji byla popsána knihovna React JS, která se ve značné míře využívá v praktické části práce.

Praktická část popisuje analýzu požadavků a návrh grafického rozhraní webové aplikace, která předchází části samotné implementace. V implementační části byla nejprve provedena komparace návrhových vzorů pomocí kvalitativní vícekriteriální analýzy variant a metody focus group, ze které vzešly kritéria pro komparaci. Těmto kritériím byla stanovena váha za pomoci Saatyho metody párového porovnání a z následné aplikace metody pořadí vyšlo hodnocení komparovaných variant. Následovalo představení použitých knihoven a frameworků ve vývojové fázi. Před vývojem aplikace proběhla konfigurace projektu, která byla rozdělena do několika fází. Implementace byla zakončena pomocí automatických testů, které se soustředily na reprezentační část aplikace. Testy pokrývají pouze nutné části, které autor plánuje rozšířit.

V rámci Diplomové práce vznikla webová aplikace Seznam Trends, která je určena po interní účely společnosti Seznam.cz, a.s. Přínos aplikace je její informační hodnota o nejčetnějších dotazech, které uživatelé hledají na fulltextovém vyhledávání společnosti. Tyto informaci bude mít k dispozici především zpravodajská divize společnosti v projektech Seznam Zprávy a Televize Seznam. Webová aplikace by měla usnadnit reportérům rozhodování o relevanci témat pro reportáže a články.

7 Citovaná literatura

ABRAMOV, Dan a Sophie ALPERT, 2018. React Today and Tomorrow. In: *YouTube* [online]. Lake Las Vegas, Nevada, USA: React Conf [cit. 2018-11-15]. Dostupné z: https://youtu.be/V-QO-KO90iQ

BANKS, Alex a Eve PORCELLO, 2017. *Learning React: functional web development with React and Redux*. First edition. Sebastopol, CA: O'Reilly Media. ISBN 14-919-5462-0.

BROWN, Daniel M., 2011. *Communicating design: developing web site documentation for design and planning*. 2nd ed. Berkeley, CA: New Riders. ISBN 978-0321712462.

BROWN, Ethan, 2014. *Web Development with Node and Express* [online]. 1st. Sebastopol: O'Reilly Media [cit. 2018-09-02]. ISBN 978-1-491-94930-6. Dostupné z: my.safaribooksonline.com

Building isomorphic JavaScript apps: Fromconcept to implementation to real-world solutions, 2016. 1st. California: O'Reilly. ISBN 978-1491932933.

CHAMBERS, Gary, 2014. *Understanding Flux* [online]. London, Velká Británie: Medium [cit. 2018-09-01]. Dostupné z: https://medium.com/@garychambers108/understanding-flux-f93e9f650af7

Chrome Developers [online], 2008. USA: Google [cit. 2019-01-14]. Dostupné z: https://v8.dev/

DE SOUSA ANTONIO, Cássio, 2015. *Pro React* [online]. 1st. New York: apress [cit. 2018-09-08]. ISBN 978-1-4842-1260-8. Dostupné z: http://file.allitebooks.com/20160130/Pro%20React.pdf

DOUBRAVOVÁ, Hana, 2009. *Vícekriteriální analýza variant a její aplikace v praxi* [online]. České Budějovice [cit. 2019-03-14]. Dostupné z: https://theses.cz/id/6citbe/downloadPraceContent_adipIdno_11361. Diplomová práce. JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH. Vedoucí práce Ing. Jana Friebelová, Ph.D.

FEDOSEJEV, Artemij, 2015. *React.js Essentials* [online]. První. Birmingham, UK: Packt Publishing Ltd. [cit. 2018-11-14]. ISBN 978-1-78355-162-0. Dostupné z: https://www.amazon.com/React-js-Essentials-Artemij-Fedosejev-ebook/dp/B00YSILZRW

FLANAGAN, David, 2011. *JavaScript: the definitive guide*. 6th ed. Sebastopol, CA: O'Reilly. ISBN 978-0-596-80552-4.

Focus group - Definition of focus group in US English by Oxford Dictionaries, 2017. In: *Oxford Dictionaries* [online]. United Kingdom: Oxford [cit. 2019-03-03]. Dostupné z: https://en.oxforddictionaries.com/definition/us/focus_group

GARRETT, Jesse James, 2011. *The elements of user experience: user-centered design for the Web and beyond*. 2nd ed. Berkeley, CA: New Riders. Voices that matter. ISBN 978-0321683687.

HEJTMÁNKOVÁ, Kateřina, 2016. *Využití MongoDB s Node.js* [online]. ČR [cit. 2019-01-14]. Dostupné z: https://vskp.vse.cz/49348_vyuziti_mongodb_s_nodejs. Diplomová práce. VŠE. Vedoucí práce Palovská, Helena.

JANČA, Marek, 2017. Webpack – moderní Web Development. *Ackee* [online]. Praha: Ackee [cit. 2019-01-14]. Dostupné z: https://www.ackee.cz/blog/moderni-web-development-webpack/

JAŠPROVÁ, Alena, 2008. *Použitelnost a přístupnost webových portálů*. Praha. Diplomová práce. Univerzita Karlova v Praze. Vedoucí práce Ing. Martin Souček, Ph. D.

KOLINEK, David, 2017. *Využití frameworků Redux a React při vývoji webových aplikací*. Praha. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Iva Stanovská.

LOUČKOVÁ, Ivana, 2010. *Integrovaný přístup v sociálně vědním výzkumu*. Praha: Sociologické nakladatelství (SLON). Studijní texty (Sociologické nakladatelství). ISBN 978-80-86429-79-3.
MIKOWSKI, Michael a Josh POWELL, 2014. *Single page web applications: JavaScript end-to-end*. Shelter Island, NY: Manning. ISBN 978-161-7290-756.
Mozilla MDN [online], 2005. USA: Mozilla [cit. 2019-01-14]. Dostupné z: https://developer.mozilla.org
NIELSEN, Jakob, 2012. Usability 101: Introduction to Usability. In: *Nngroup* [online]. USA: nngroup [cit. 2019-03-01]. Dostupné z: https://www.nngroup.com/articles/usability-101-introduction-to-usability/
O přístupnosti, 2008. *Přístupnost.cz: otevřete svůj web všem* [online]. Česko: pristupnost [cit. 2019-03-01]. Dostupné z: http://www.pristupnost.cz/o-pristupnosti/
PAI, Sunil, 2018. Difference between Class and Hooks. In: *Twitter* [online]. USA: Twitter [cit. 2019-01-20]. Dostupné z: https://twitter.com/threepointone/status/1056594421079261185?s=20
PUREWAL, Semmy, 2014. *Learning web app development*. First edition. Beijing: O'Reilly. ISBN 14-493-7019-5.
React - A JavaScript library for building user interfaces [online], 2013. Menlo Park, California: Facebook [cit. 2018-09-02]. Dostupné z: reactjs.org/docs
SAATY, Thomas L. a Kirti PENIWATI, 2008. *Group decision making: drawing out and reconciling differences*. Pittsburgh, PA: RWS Publications. ISBN 978-1-888603-08-8.
SIDELNIKOV, Greg, 2017. React.js Props Tutorial. *Medium* [online]. **2017**(1), 1 [cit. 2019-01-21]. Dostupné z: https://medium.com/@js_tut/react-js-props-tutorial-a3aceb69999c
The Reactive programming toolkit, 2018. In: *The Art of Service* [online]. USA: theartofservice [cit. 2019-03-01]. Dostupné z: https://store.theartofservice.com/the-reactive-programming-toolkit/
TILLEY, Michelle, 2014. What is Flux?. *Fluxxor* [online]. Gaffney: fluxxor [cit. 2018-09-02]. Dostupné z: http://fluxxor.com/what-is-flux.html
Ukázky nedostatků přístupnosti, 2008. In: *Přístupnost* [online]. Česko: pristupnost [cit. 2019-03-01]. Dostupné z: http://www.pristupnost.cz/pristupnost-webu-statni-spravy/pristupnost-webovych-stranek-statni-spravy
What is a website prototype?, 2017. In: *Experienceux* [online]. USA: Experienceux [cit. 2019-03-01]. Dostupné z: https://www.experienceux.co.uk/faqs/what-is-a-website-prototype/
WHEELER, Ken, 2016. *Getting To Know Flux, the React.js Architecture* [online]. USA: Scotch [cit. 2018-09-02]. Dostupné z: https://scotch.io/tutorials/getting-to-know-flux-the-react-js-architecture
ZOHRABYAN, Hripsime, 2014. *Dlouhodobý pobyt v zahraničí a jeho vliv na postoje studentů Fakulty sociálních studií Masarykovy univerzity k migrantům žijícím v České republice* [online]. Brno [cit. 2019-03-03]. Dostupné z: https://is.muni.cz/th/u5ig1/Diplomova_prace_Zohrabyan.pdf. Diplomová práce. Masarykova univerzita. Vedoucí práce PhDr. Michal Vašečka, Ph.D.