České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Matouš Kadrnoška**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Vizualizační komponenty pro automatickou podporu explorativního testování softwaru**

Pokyny pro vypracování:

Navrhněte a implementujte vizualizační komponentu - interaktivní mapa testované aplikace - pro automatickou podporu explorativního testování softwaru. Komponenty budou rozšířením (tj. pluginem) stávající webové aplikace TapirHQ, implementované školitelem. Aplikace TapirHQ je webová aplikace implementované jako SPA (single page application) v JavaScriptu s použitím knihoven React, Redux pro infrastrukturu a SemanticUI pro grafické rozhraní. Základní datovou strukturou, kterou budou navržené komponenty vizualizovat, je mapa testovaného systému.
Detail této struktury bude dodán školitelem. Jejím základem je orientovaný graf, kde hrany reprezentují přechody mezi stránkami testované aplikace a uzly reprezentují jednotlivé stránky s odlišením, zda byla stránka navštívena nebo nikoliv. Mapa by měla být schopná vykreslit bez větších problémů několik desítek uzlů (ale ne více než 100) s malou zátěží prostředí prohlížeče. Mapa bude interaktivní - po kliknutí na uzel se zobrazí v dolní polovině konkrétní metadata uzlu. Mapa se bude průběžně aktualizovat podle aktuálních dat v aplikaci TapirHQ.
K implementaci použijte vhodné technologie pro vizualizaci (použitelné v prostředí React) a přehledné rozložení na ploše (layout engine). Navrhnětě vhodný způsob testování pro vytvořené komponenty.

Seznam odborné literatury:

K. Frajtak, M. Bures and I. Jelinek, "Model-Based Testing and Exploratory Testing: Is Synergy Possible?," 2016 6th International Conference on IT Convergence and Security (ICITCS), Prague, Czech Republic, 2016, pp. 1-6.
http://ieeexplore.ieee.org/document/7740354/

Vedoucí: Ing. Karel Frajták

Platnost zadání: do konce letního semestru 2017/2018

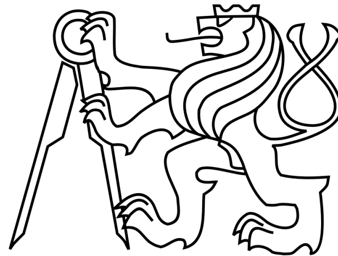
prof. Ing. Jiří Žára, CSc.
vedoucí katedry

L.S.

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 28.11.2016

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Vizualizační komponenty pro automatickou podporu explorativního testování softwaru

Matouš Kadrnoška

Vedoucí práce: Ing. Karel Frajták

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimedia

26. května 2017

Poděkování

Chtěl bych poděkovat panu Ing. Karlu Frajtákovi, za rady ohledně implementace a za vysvětlení principu mateřské aplikace TapirHQ.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 26. 5. 2017 ....

Abstract

The aim of this thesis is creation of a plugin for TapirHQ application, which will be able to visualize a map of a web application. This map has a structure of oriented graph where knots represent individual web pages of the application and links represent the possible transition between them. This precedes a research of available JavaScript visualisation libraries, comparing these libraries and finaly choosing the one which is best suited for the job. Final task is to suggest a method of testing this visualisation component.

Abstrakt

Cílem této práce je vytvořit doplňující komponentu aplikace TapirHQ, která bude vizualizovat mapu stránky. Tato mapa má strukturu orientovaného grafu, kde uzly představují stránky aplikace a hrany reprezentují přechody mezi nimi. Tomu předchází rešerše dostupných vizualizačních JavaScriptových knihoven, jejich srovnání a výběr nejvhodnější z nich. Dalším cílem této práce je navrhnout vhodné testování této grafické komponenty.

Obsah

1 Úvod **1**

2 Cíl práce **2**

3 Struktura práce **3**

4 Rešerše dostupných technologií **4**
4.1 Úvod 4
4.2 Knihovny 4
4.2.1 Dygraph.js 4
4.2.2 D3.js 5
4.2.3 Springy.js 6
4.2.4 Arbor.js 6
4.2.5 Sigma.js 7
4.2.6 Three.js 7
4.3 Závěr rešerše 8

5 Použité technologie **9**
5.0.1 Úvod 9
5.1 React 9
5.2 Redux 10
5.3 TypeScript 10
5.3.1 Ukázka Typescript kódu 10
5.3.2 Ukázka Javascript kódu 11
5.4 D3 11
5.4.1 Force-Directed Graph 12

6 Implementace **13**
6.1 Formát dat 13
6.2 Struktura aplikace 15
6.2.1 Model 15
6.2.1.1 Node 15
6.2.1.2 Link 16
6.2.1.3 DataStorade 16
6.2.1.4 DataReceiver 16
6.2.1.5 NodeFactory 16

6.2.2 View . . . 17
6.2.3 Controller . . . 17
6.2.4 Komunikace mezi React.js a D3.js . . . 18
6.3 Zobrazení dat . . . 18
6.3.1 Charge . . . 20
6.3.2 Center . . . 20
6.3.3 Border . . . 20
6.3.4 Link . . . 21

7 Testování 22
7.1 Úvod . . . 22
7.2 Testování grafických komponent . . . 22
7.2.1 Testování s Jest a Enzyme . . . 24
7.2.2 Testované případy . . . 24

8 Závěr 25

Literatura 26

9 Seznam použitých zkratek 27

10 Instalační a uživatelská příručka 28
10.1 Zprovoznění aplikace . . . 28
10.2 Testování . . . 28
10.3 Vývojové prostředí . . . 28

11 Seznam přiložených souborů 29

Seznam obrázků

4.1 Ukázka DygraphJs 5
4.2 Ukázka knihovny D3 Js 5
4.3 Ukázka knihovny SpringyJs 6
4.4 Ukázka knihovny ArborJs 7
4.5 Ukázka knihovny SigmaJs 7
4.6 Ukázka knihovny ThreeJs 8

6.1 Struktura vstupních dat implementované aplikace 14
6.2 Diagram tříd aplikace 17
6.3 Ukázka hotové mapy vygenerované z testovacích dat 21

7.1 Příklad snapshot testu kde se zobrazilo méně objektů než mělo[2] 23

8.1 Celkový výsledek mé práce 25

Seznam tabulek

4.1 Shrnutí rešerše 8

Kapitola 1

Úvod

Testování je již nedílnou součástí při vytváření jakéhokoliv nového softwaru nebo webové aplikace. Dnes používané techniky testování spoléhají na lidksé síly a intuici. To je v dnešní době velmi časově i finančně náročné. Neexistuje tedy lepší způsob jak aplikace testovat? Nejvhodnější by bylo aby celou aplikaci testoval stroj. Programátor by si pak mohl být jistý, že se v jeho aplikaci opravdu otestovalo vše. Takové techniky bohužel v dnešní době dostupné nejsou, a proto je stále nutné spoléhat na lidi a jejich abilitu otestovat všechny možné scénáře, které mohou v aplikaci nastat. Nešlo by alespoň trochu testerům usnadnit práci a zapojit počítače do procesu testování? Odpověď zní ano. Tato práce se zabývá vytvářením mapy aplikace, která umožní testerům lepší průchod testovanou aplikací. Tato práce vzniká jako součást aplikace TapirHQ, jejíž součástí je rozšíření pro internetový prohlížeč Google Chrome, které analyzuje webovou stránku a najde v ní všechny odkazy a formuláře, které uživatel muže použít aby se dostal na jinou stránku aplikace. Zároveň toto rozšíření nahrává všechny akce, které uživatele přesměrovávají na jinou stránku. Informace o všech prvcích stránky a o všech krocích, které uživatel provedl se ukládají do databáze ve formátu JSON.

Kapitola 2

Cíl práce

Cílem této práce je vytvořit plugin do aplikace TapirHQ, který vytvoří vizualizaci testované aplikace v podobě orientovaného grafu, ve kterém uzly reprezentují stránky testované aplikace a hrany reprezentují přechody mezi těmito stránkami. Při průchodu testera aplikací se jeho aktivita v aplikaci nahrává a ukládá. To znamená, že máme záznam o tom jaké stránky tester navštívil a co vyplňoval do formulářových polí. Tyto informace jsou zakomponovány do vytvářeného grafu tak, že každý uzel v grafu, je buď navštívený, nebo nenavštívený. Toto se promítá do grafu barevným odlišením uzlů. Dále je důležité odlišit typy přechodů mezi stránkami. Momentálně rozlišujeme dva typy přechodů, klasický HTML odkaz a přechod pomocí odeslání formuláře. Tyto typy se projeví jinou barevnou reprezentací hran grafu.
K implementaci této vizualizace je zapotřebí vybrat vhodnou knihovnu napsanou v jazyce JavaScript, která bude schopná vykreslit strukturu na sebe navazujících stránek a zároveň bude kompatibilní s frameworkem React, který využívá aplikace TapirHQ jako prezentační vrstvu.

Kapitola 3

Struktura práce

Na začátku práce se zabývám rešerší vizualizačních knihoven, které jsou nutnou součástí implementování vizualizační komponenty. Na závěr rešerše shrnu vlastnosti nalezených knihoven a na základě toho vyberu tu, kterou později budu požívat. V další kapitole si představíme knihovny a frameworky, které jsou použity v aplikaci TapirHQ nebo které použijeme v implementaci nového modulu. Poté se budeme zabývat implementací samotného modulu a jeho zakomponování do existující struktury aplikace. A na závěr shrnu možnosti testování vytvořené aplikace.

Kapitola 4

Rešerše dostupných technologií

4.1 Úvod

Tato kapitola se zabývá hledáním nejvhodnější technologie pro zobrazování dat v JavaScriptu. Na začátku srovnáme momentálně nejpopulárnější JavaScriptové knihovny pro vizualizaci dat a poté vybereme tu nejvhodnější. Výslednou knihovnu jsem vybíral podle několika kritérií. Tato kritéria jsou:

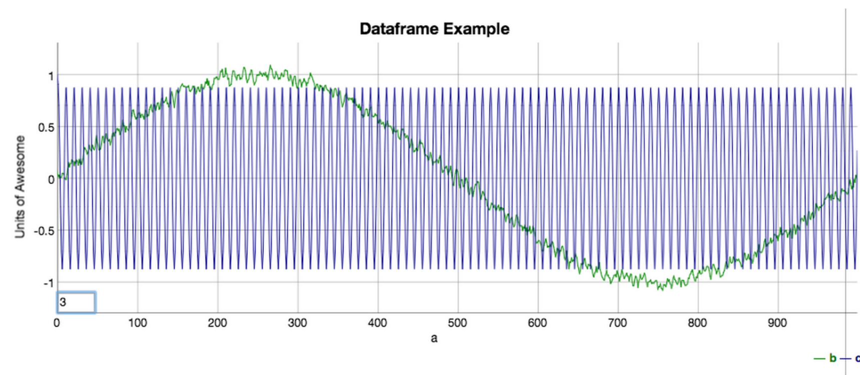
- schopnost vykreslovat nehierarchické struktury dat.
- jednoduchost a čitelnost zdrojového kódu.
- aktualizovanost knihovny
- možnost propojení s frameworkem React.

4.2 Knihovny

4.2.1 Dygraph.js

Dygraph je knihovna pro vykreslovaní grafů na základě dodaných dat. Tato knihovna je vhodná hlavně pro vykreslování grafů kde uživatel chce vidět vzájemnou závislost dvou hodnot. Pro tyto účely vytváří knihovna osy do kterých dosazuje data, která dostane od uživatele. Knihovna nativně podporuje vlastnosti přibližování a posouvání. Knihovna je udržována a pravidelně se na její git commituje, což napovídá, že v budoucnu bude její podpora stále aktivní.

- Umí jen nevhodné typy grafů.
- Existující knihovna pro spojení DygraphJs a React,

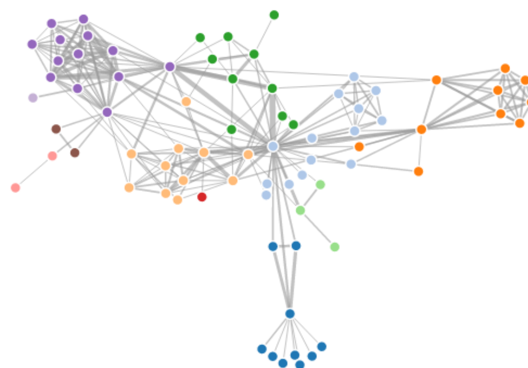


Obrázek 4.1: Ukázka DygraphJs

4.2.2 D3.js

D3 neboli Data-Driven Documents je JavaScriptová open-source knihovna, která se specializuje na vizualizaci dat. Tato knihovna využívá SVG elementy k vykreslování jednotlivých částí grafu. Pro každý kus dat, vytvoří knihovna odpovídající HTML element se kterým dále manipuluje. D3 umožňuje uživateli globálně manipulovat s částmi dat pomocí tzv. selections neboli výběrů DOM elementů pomocí HTML selectorů. U těchto vybraných elementů D3 manipuluje jejich atributy jako je pozice, barva, text apod. D3 obsahuje několik přednastavených typů grafů a možností jak data zobrazovat.

- Vhodná pro nehierarchické typy grafů.
- Existující knihovna pro spojení D3 a frameworku React.
- Pravidelně udržovaná knihovna.

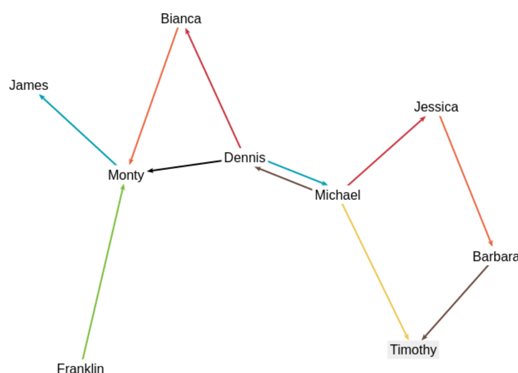


Obrázek 4.2: Ukázka knihovny D3 Js

4.2.3 Springy.js

Springy je malý JavaScriptový framework pro zobrazování tzv. force-directed grafů neboli grafů ve kterých na sebe uzly působí ruznými silami a tím umožňují ideální rozložení uzlů na stránce. Springy je navržen jako velmi jednoduchý a malý framework pracující s HTML Canvas elementem pro vykreslování.

- Vhodná knihovna pro nehierarchické grafy.
- Jednoduchý a intuitivní zápis kódu.
- Žádné zdokumentované propojení s frameworkem React.

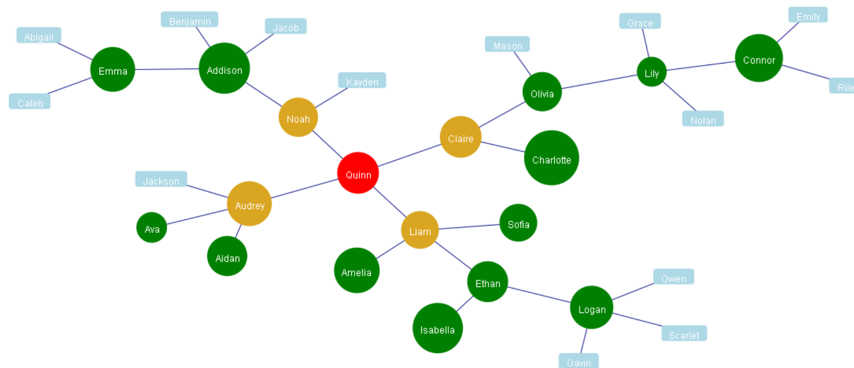


Obrázek 4.3: Ukázka knihovny SpringyJs

4.2.4 Arbor.js

Arbor je knihovna určená pro vykreslování grafů a napsaná pomocí jQuery a web workers. Arbor umí zobrazovat force-directed grafy a sám se stará o obnovování stavu grafu. Zobrazování grafu je ponecháno na uživateli, což znamená, že je možné použít canvas, SVG nebo HTML elementy s určenou pozicí.

- Vhodná knihovna pro nehierarchické grafy.
- Závislá ja jQuery.
- Intuitivní zápis zdrojového kódu.
- Žádné zdokumentované propojení s frameworkem React.

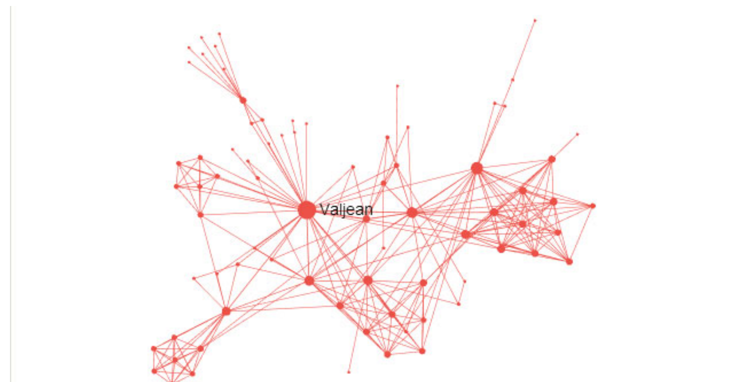


Obrázek 4.4: Ukázka knihovny ArborJs

4.2.5 Sigma.js

Sigma je JavaScriptová knihovna zaměřená pro vizualizaci nehierarchických dat. Pro zobrazování dat je použit HTML canvas nebo WebGL.

- Vhodná knihovna pro nehierarchické grafy.
- Vstupní data ve formátu JSON
- Přehledný a lehce naučitelný styl kódu.
- Existující knihovna pro propojení s frameworkem React.

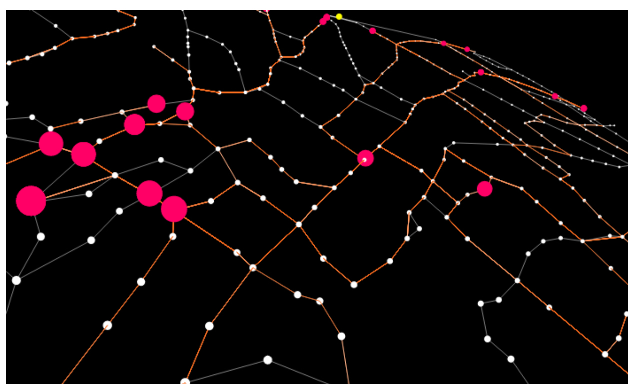


Obrázek 4.5: Ukázka knihovny SigmaJs

4.2.6 Three.js

Three.js je knihovna, která si dává za úkol jednoduše vytvářet 3D vizualizace. Podporuje zobrazování skrz Canvas, SVG, CSS3D i WebGL.

- Velmi rozsáhlá vizualizační knihovna.
- Velmi těžce pochopitelný kód.
- Knihovna je určena spíše pro jiné typy aplikací než kreslení grafů.
- Existující knihovna pro spojení ThreeJs a React.



Obrázek 4.6: Ukázka knihovny ThreeJs

4.3 Závěr rešerše

Knihovna	Vykreslování	React	Čitelnost kódu	Aktualizovanost
DygraphJs	A	A	N	A
D3	A	A	A	A
Springy	N	N	A	A
Arbor	A	N	A	N
Sigma	A	A	A	A
Three	N	A	N	A

Tabulka 4.1: Shrnutí rešerše

Z předcházející tabulky vyplývá, že nejlepšími možnostmi pro implementaci jsou knihovny D3 a SigmaJs. Vzhledem k tomu, že obě splňují všechna kritéria, které jsem stanovil na začátku rešerše, musel jsem se rozhodnout na základě jiných kritérií. Za toto kritérium jsem vybral nalezení vhodného příkladu, který by mi ulehčil se s knihovnou seznámit a naučit se s ní pracovat. Proto jsem se nakonec rozhodl pro knihovnu D3 a její napojení na framework React pomocí knihovny react-d3-library.

Kapitola 5

Použité technologie

5.0.1 Úvod

V této kapitole shrnu technologie, ve kterých je napsána aplikace TapirHQ, ke které se tato práce váže. To zároveň pomůže k vysvětlení některých věcí v implementaci, která je v další kapitole.

5.1 React

ReactJS je opensourcová JavaScriptová knihovna vytvořená v roce 2013 společností Facebook a Instagram. React je knihovna pro vytváření uživatelského rozhraní webových aplikací. Základem Reactu jsou komponenty. Vše co uživatel vidí na obrazovce je komponenta a každá tato komponenta je ve zdrojovém kódu reprezentován třídou. Všechny tyto třídy komponent rozšiřují třídu React.Component, která jim předepisuje metodu render. Tato metoda obsahuje HTML kód a odkazy na další komponenty, které se poté uživateli zobrazují na stránce. Tento koncept zanořování se velice podobá klasickému šablonování. React je ale vyjímečný tím, že na rozdíl od šablon, kde vkládáme kód do předepsaného HTML kódu, vkládáme HTML do zdrojového kódu .

Každý komponent má svůj vlastní vnitřní stav. Tento stav je nativně dostupný pouze komponentu ale samozřejmě je možné ho delegovat dál ručně. Stav komponentu určuje, jaké hodnoty bude komponent vykreslovat. Tento stav je tzv. immutabilní, neboli nelze ho změnit. Tento stav se dá změnit pouze tak, že jej vyměníme za nový, odlišný.

Další vlastností Reactu je immutabilita, neboli znemožnění změny, komponent. Tuto vlastnost React využívá při překreslování stránky. Překreslení stránky může nastat v případech, že uživatel například klikne na interaktivní část stránky. Tato vlastnost napomáhá frameworku překreslovat pouze ty elementy stránky, které byly nějakým způsobem změněny a ne celou stránku, jak tomu je v jiných frameworcích. K tomu slouží tzv. virtuální DOM. To je kopie vygenerovaného DOMu, kterou si React udržuje v paměti. Při změně komponenty v DOMu porovná React svůj virtuální DOM s aktuálním a pokud se nějaké jeho části neshodují, překreslí je.

```
    render() {
       return (
          <div className='applicationMap'>
            <h1>Application Map</h1>
            <Map data={this.state.mapProps}/>
            <Table data={this.state.tableProps}/>
          </div>
       );
    }
}
```

Příklad metody *render*.

5.2 Redux

Stejně jako React je Redux také JavaScriptovou knihovnou. Tak jako se React zabývá pouze front-endovou vrstvou, Redux má na starosti jedinou věc, udržovat stav aplikace. Tento stav obsahuje všechna data, která potřebujeme mít v aplikaci uložena. Tento stav je neměnný. Dá se pouze přepsat stavem novým. Způsob jakým se provádějí změny v aplikaci je založený na tzv. akcích. Akce, je běžný JavaScriptový objekt, který popisuje co se má změnit a jak se to má změnit. Tento koncept přináší velkou výhodu. Tím, že každá změna stavu je reakcí na vyslanou akci, nám seznam vyslaných akcí zaznamenává přesný průběh co se v aplikaci dělo a tím i jednoduchý způsob jak reprodukovat chyby.

```
{
    type: 'DECREMENT'
    payload: 1
}
```

Toto je ukázka jak může vypadat nejjednoduší akce, která mění stav aplikace. Akce musí obsahovat pole se jménem *type*, které určuje o kterou akci se jedná. Zbytek dat už je libovolný.

5.3 TypeScript

TypeScript je programovací jazyk, který je nadstavbou nad jazykem Javascript. Jak název napovídá, jedná se, narozdíl od JavaScriptu který je netypový o typový jazyk. Kód psaný v TypeScriptu se kompiluje do běžného JavaScriptu. Tento kompilátor nám při procesu provede typovou kontrolu a sdělí případné chyby. Tento jazyk jsem použil vzhledem k mé přechodzí zkušenosti a zároveň kvůli své preferenci typových jazyků.

5.3.1 Ukázka Typescript kódu

```
public markVisited(visitedPages :string[]){
        var iNode :Node;
```

```
        var ii,jj :number = 0;

        for(iNode of this.nodeSet){
            for(ii = 0; ii < visitedPages.length; ii++){
                if(iNode.getUri() == visitedPages[ii]){
                    iNode.setVisited(true);
                    break;
                }
            }
        }
    }
```

5.3.2 Ukázka Javascript kódu

```
markVisited(visitedPages) {
        var iNode;
        var ii, jj = 0;
        for (iNode of this.nodeSet) {
            for (ii = 0; ii < visitedPages.length; ii++) {
                if (iNode.getUri() == visitedPages[ii]) {
                    iNode.setVisited(true);
                    break;
                }
            }
        }
    }
```

Jak je vidět z ukázky, zkompilovaný typescriptový kód je přehledný a čitelný. Takže k dalšímu vývoji aplikace TapirHQ není zapotřebí uchovávat zdrojové kódy v jazyku Typescript ale stačí pouze jejich zkompilované obrazy.

5.4 D3

D3.js je vizualizační knihovna napsaná v jazyce JavaScript. Její název je zkratkou **D**ata-**D**riven-**D**ocuments. Jejím cílem je pomoci programátorům s vizualizací dat jejich aplikací. Data to mohou být jakákoliv. D3 má několik přednastavených typů zobrazování neboli layoutů. Jedním z nich je i Force layout, který jsem v práci použil, a proto ho v této práci vice přiblížím. D3 knihovna využívá k zobrazování dat elementy HTML a SVG, které dynamicky přidává, odebírá a upravuje na webové stránce tak, aby výsledná stránka odpovídala vstupním datům.
D3 manipuluje DOM objekty pomocí výběrů, které provádíme na základě CSS selektorů. Ke každému výběru přiřadíme sadu dat, kterou chceme zobrazit. D3 nabízí speciální syntaxi která se stará o to aby byla všechna data namapována na DOM objekt. Tato syntaxe přináší tři metody, které jsou součástí tzv. General Update Patternu[1].

[1] https://bl.ocks.org/mbostock/3808218

- *data* Tato funkce spojí data s elementy DOMu. Pokud již v průběhu vizualizace změníme data, tato funkce se postará, že D3 změní správné DOM elementy.
- *enter, exit* Tyto dvě funkce se starají o to, aby každý kus dat, měl svůj vlastní element. Pokud po přidání nových dat zavoláme funkci *enter* na výběr DOM elementů, určujeme tím co chceme aby se dělo s daty, které k sobě ještě žádný DOM objekt nemají přiřazený. Naopak funkcí *exit* určujeme, co se stane s DOM objekty, se kterými už nejsou spojená žádná data.

```
nodeElements = d3.select("svg").selectAll(".node")
            .data(this.nodeArray);

nodeElements = nodeElement.enter()
            .append("g")
            .append("circle")
            .attr("r", 5)
            .attr("fill", function(d){
                if(d.visited){
                    return("grey");
                }
                else{
                    return("black");
                }
            })
            .attr("class", "node");
```

Ukázkový kód vytvoří pro nová data DOM objekt *g*, který je podřazený objektu *svg* a přidá k němu nový HTML tag *circle*, kterému upraví argumenty. Konkrétně zvolí parametr *r* 5 pixelů ale jen u těch objektů, které ještě tento parametr nemají. To nám ovlivňuje funkce *enter*, která z celé množiny objektů vybere pouze ty nově přidané.

5.4.1 Force-Directed Graph

Force-Directed Graph je způsob zobrazování svázání navzájem propojených dat. Mapa aplikace, která je výsledkem této práce má přesně stejnou strukturu. Uzly, které jsou navzájem spojeny reprezentují webové stránky, a spojnice mezi nimi, které budeme nazývat hrany, reprezentují možné přechody mezi těmito stránkami. Force-Directed layout funguje tak, že všechny uzly, které jsou spojené hranou, na sebe působi přitažlivou silou a uzly, které spojené nejsou se navzájem odpuzují. Tyto změny se projeví do pozice uzlů v grafu a ty se zase projeví změnou velikosti sil, kterými uzly působí. Pokud budeme tento stav neustále aktualizovat, dosáhne graf rovnovážného stavu.

Kapitola 6

Implementace

6.1 Formát dat

Data obdržená z aplikace TapirHQ jsou ve formátu JSON[1]. Zdrojem dat v aplikaci TapirHQ je databáze MongoDB, která běží na serveru implementovaným v jazyku C#. Protože toto nastavení je určeno pro platformu Windows a tento modul byl vyvíjen na platformě Linux a zároveň protože aplikace TapirHQ je stále ve vývoji a tento server není veřejně přístupný, byl přístup do databáze nasimulován čtením dat ze souborů, ve kterých se nachází export z dat z databáze.
Tato data pocházejí z rozšíření webového prohlížeče Google Chrome, které analyzuje webovou stránku a najde v ní všechny odkazy a formuláře které při odeslání odkáží uživatele na další stránku. Tato data jsou následně odeslána na server a uložena do databáze. Na obrázku níže můžete vidět ukázku dat určených k vytvoření mapy. K vykreslení orientovaného grafu znázorňujícího mapu aplikace potřebujeme pouze URL počáteční stránky a poté URL stránek dostupných z této počáteční. Tyto hodnoty jsou na obrázku zvýrazněny barevně. Konkrétně se jedná o položku *uri*, ta označuje adresu počáteční stránky. Dále je v datech položka *forms*, která obsahuje pole, ve kterém jsou uloženy informace o včech formulářích na stránce. V těchto informacích je položka *action*, která obsahuje adresu, na kterou bude uživatel přesměrován v momentu odeslání formuláře. Na stejné úrovni jako pole formulářů je položka *linkGroups*, která obsahuje pole s informacemi o odkazech na další stránky. Adresy těchto stránek se nacházejí v položce *location*. Všechny tyto adresy, se zpracují a použijí k vytvoření výsledného grafu.

[1] JavaScript Object Notation

▼ _id {1}
$oid : 57e374fe506e4b5a00a00278
uri : /my_view_page.php
hasPendingChanges : false
▼ created {2}
date : 2016-09-22T06:06:54.1802577Z
▼ user {2}
_id : kfrajtak
name : Karel FrajtÃ¡k
versionId : 57aa2499003029e88bea16e6
▼ forms [1]
▼ 0 {5}
action : /jump_to_bug.php
method : post
name : value
▶ actionElements [1]
▶ inputElements [1]
▼ linkGroups [1]
▼ 0 {2}
name : linkGroup0
▼ links [1]
▼ 0 {5}
value : value
local : true
location : /my_view_page.php
target : value
type : link
▶ pendingChanges [1]
▶ metaData {1}

Obrázek 6.1: Struktura vstupních dat implementované aplikace

6.2 Struktura aplikace

Aplikace byla vyvíjena jako samostatný modul, téměř nezávislý na aplikaci TapirHQ, Aplikace TapirHQ slouží jako poskytovatel prostředí pro běh aplikace a jako zdroj dat. K tomu aby byla aplikace nezávislá potřebuje si udržovat informaci o zobrazovaných date ve vlastní paměti. Zdroj dat aplikace TapirHQ nelze použít z toho důvodu, že naše aplikace nebude zobrazovat všechna dostupná data ale pouze malou část. K této zobrazované části dat bude možné další data přidávat, a proto, aby byla dodržena unikátnost zobrazovaných dat, je potřeba o této množině dat udržovat informace samostatně.
Data určená k zobrazování a uložená v paměti dále budou muset být zformátována do struktury, kterou očekává knihovna D3, která se stará o vykreslování dat na stránku. Data k zobrazování ale nejsou jedinou informací, kterou potřebujeme k vykreslování. Dalším důležitým kusem skládanky je informace, jakým způsobem se tato data budou zobrazovat. Konkrétně tím mám na mysli například informace o velikosti zobrazované mapy, barvy jakými se bude mapa vykreslovat, jakým způsobem se budou zobrazovat jednotlivé typy spojení atd. Všechny tyto konfigurační data je vhodné mít na jednom místě aby bylo možné je jednoduše měnit. Jednotlivé funkční části je dobré mít implementované v úzkých celcích, a proto jsem využil objektově orientovaného kódu, který toto rozdělení umožňuje.
Za nejvýhodnější rozdělení celků považuji rozdělit kód na část, která udržuje potřebná data, část, která tyto data vykresluje a část, která zprostředkovává komunikaci mezi nimi. Toto rozdělení je již praxí natolik zavedené, že už má vlastní název. Jedná se o rozdělení podle návrhového vzoru MVC, tedy Mode, View a Controller. Část View se stará pouze o zobrazování a o akce provedené uživatelem, na zobrazované stránce. Ideálně by měla fungovat jako automat, který pouze vykresluje data a zpracovává akce, bez žádné aplikační logiky. Data do této části se dostávají z Controlleru. Ten se chová jako prostředník, který předává data, formátuje data a stará se o správné provedení uživatelských akcí. Poslední vrstva Model se stará o uložení a vydávání dat na popud Controlleru. Každý z těchto celků hlouběji popíši v následujících kapitolách.

6.2.1 Model

Vrstva mode je určená k udržování dat, která aplikace používá, v našem případě data, která se budou vykreslovat na obrazovku. K funkci cele aplikace potřebujeme aby modelová vrstva uměla několik věcí. Obdržet data z nějakého zdroje, uložit data, vrátit data na požádání a zajistit unikátnost udržovaných dat. K tomu jsem naimplementovat několik tříd.

6.2.1.1 Node

Třída reprezentující jednotlivé uzly grafu. Instance třídy Node v sobě udržují informace o své url, infomaci jestli už byla stránka, reprezentována tímto uzlem, navštívena testerem. Tato informace se promítne do výsledné barvy uzlu v grafu. Dále tato třída obsahuje svoje unikátní uid jako string a unikátní číslo d3Index použité v číslování uzlů v knihovně D3. Metody této třídy povětšinou nastavují nebo čtou výše zmíněné hodnoty. Výjímkou je metoda *toString*, která vrátí textovou reprezentaci tohoto uzlu.

6.2.1.2 Link

Jedná se o jednoduchou datovou strukturu, která definuje spojení dvou uzlů v grafu. Ke svému vzniku třída potřebuje dvě instance třídy Node, které slouží jako zdroj a cíl spojení, které je definováno. Dále obsahuje metody pro určení začátku a konce spojení, metodu pro porovnání dvou instancí třídy Link a dále také uchovává informaci o typu spojení,k teré tato třída reprezentuje. Tím se ve finálním zpracování grafu odliší přechody mezi stránkami pomocí formuláře nebo klasického odkazu.

6.2.1.3 DataStorade

Hlavni tridou modelove vrstvy je trida DataStorage. Tato trida zajistuje spravny chod ostatnich potrebnych trid modelove vrstvy a hlavne se stara o udrzovani seznamu uzlu a seznamu hran v pameti. Zaroven zajistuje unikatnost svych dat, pro pripad, ze se objevi pozadavek na pridani jednoho uzlu dvakrat. Trida DataReceiver si udrzuje odkaz na instanci tridy DataReceiver, ktera se stara o prejimani dat z tridy Controller a o ukladani dat do tridy DataStorage. Data se ve tride ukladaji pomoci datovych struktur Set, neboli mnozina. Tuto datovou strukturu jsem vybral pro jeji jednoduchost ovladani.

6.2.1.4 DataReceiver

Tato třída funguje jako prostředník mezi třídou Controller a třidou DataSrorage. Na starost má předání dat třídě DataStorage ve správném formátu.

6.2.1.5 NodeFactory

Tato třída vytváří nové instance třídy Node, která reprezentuje uzel z dat dodaných do třídy DataReceiver. Do třídy DataReceiver se data dostávají ve formátu JSON a tato třída má za úkol z tohoto formátu vytvořit jednu nebo více instancí třídy Node. Pro tvorbu tříd Node takovouto cestou jsem se rozhodl, protože to umožňuje jednoduché rozšíření aplikace v případě, že bude nutné přidat informace v těchto třídách. Změna pak proběhne pouze v této třídě ostatní kód zůstane netknutý. V případě změny rozhraní musí nové třídy implementovat tyto metody aby byla zachována funkčnost.

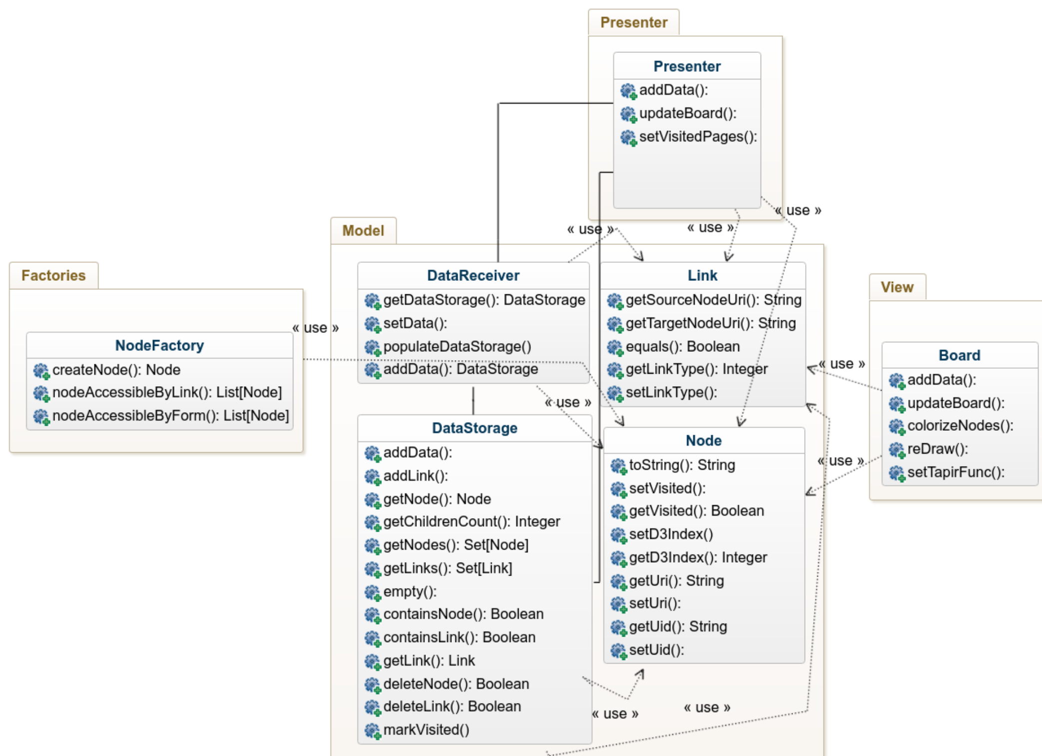
- toString
- getVisited
- setVisited
- getUri
- equals

6.2.2 View

- Tato vrstva se stará o zobrazování dat uživateli a zachytává všechno co uživatel v aplikaci chce udělat. Implementována je pouze jediná třída Board, která se stará o vše potřebné. Tím je myšleno zobrazování dat, notifikace controlleru o uživatelských akcích a stylování výsledné mapy. Uživatelské akce jsou definovány na každém uzlu mapy. V případě, že uživatel přejede myší přes uzel je zavolána funkce napojená na událost *mouseover*. Ta zvýrazní a barevně oddělí všechny hrany, které buďto v daném uzlu začínají nebo končí. Další událostí reagující na uživatelský podnět je kliknutí na uzel. Tato událost zavolá funkci controlleru, která ji dále deleguje až do funkce componenty frameworku React, který ji zpracuje tak, že na obrazovce ukáže detaily o uzlu.

6.2.3 Controller

- Controller funguje jako prostředník mezi datovou vrstvou a zobrazovací vrstvou. Stará se o veškerou logiku aplikace a přeposílá data do částí, do kterých je potřeba. V naší aplikaci deleguje zprávy od prezentační vrstvy do obalového React komponentu, kvůli aktualizaci tabulky s informacemi o uzlu při kliknutí na uzel. Hned při startu aplikace se mezi React komponentou a Controllerem vytvoří jakýsi komunikační tunel, který je realizovaný polem, ve kterém obě části předávájí odkazy na své metody pro účely komunikace mezi těmito komponenty.



Obrázek 6.2: Diagram tříd aplikace

6.2.4 Komunikace mezi React.js a D3.js

Aby bylo možné, nějakým způsobem ovlivňovat vykreslování mapy aplikace, bylo zapotřebí vymyslel způsob jakým se budou předávat instrukce z React komponenty do modulu starajícího se o vykreslování. K tomu jsem použil soubor *.../ApplicationMap/components/Map/node.js*, který je nutný ke běhu knihovny react-d3-library. V tomto souboru se definuje prostředí knihovny D3 a zároveň se vytváří třída Controller. Některé proměnné v tomto souboru se exportují jako modul pomocí metody *module.exports*, která je zabalí do jednoho objektu, který je možné si vyžádat kdekoliv v kódu. Mezi exportované hodnoty jsem zařadil několik funkcí controlleru, které slouží k ovládání chování celé mapy. Konkrétně se jedná o následující metody:

- updateFunc - funkce volaná pomocí časovače v React komponentě. Stará se o synchronizaci vykreslování mapy s obnovováním stavu komponenty. Je volána přibližně 30x za vteřinu, což zajišťuje iluzi plynulého pohybu pro lidské oko.
- passDataFunc - funkce, která vykreslí novou mapu s daty, které jsou v argumentu této funkce.
- addDataFunc - funkce, která se používá pro přidání dat do, již existující, mapy.
- setVisited - funkce, která se postará o barevné rozlišení uzlů reprezentujících již navštívené adresy.

```
module.exports = {
    "node": node,
    "updateFunc": updateFunc,
    "passData": passDataFunc,
    "setVisited": setVisitedPagesFunc,
    "addData": addDataFunc,
    "tapirFunc": tapirBoardTransmitter
};
```

Reálna ukázka exportovaného pole hodnot a funkcí z modulu node.js

6.3 Zobrazení dat

Jak již bylo řečeno v předchozích kapitolách pro zobrazení dat byla použita knihovna D3 a v ní implementovaný Force-Directed layout. Pro přehledné rozvržení dat bylo zapotřebí spojit několik sil působících napříč grafem. Tyto síly se implementují jako separátní funkce a následně se spojí do jednoho objektu nazvaného simulace. Každý kus dat, který D3 přijme do své simulace je objekt, ke kterému D3 přidá následující hodnoty.

- vx - vyjadřuje rychlost, kterou se uzel v grafu pohybuje ve směru osy X
- vy - vyjadřuje rychlost, kterou se uzel v grafu pohybuje ve směru osy Y

- x - vyjadřuje momentální pozici uzlu v ose X
- y - vyjadřuje momentální pozici uzlu v ose Y

D3 si vytváří vlastní časovač, který spustí hned po definici simulace. Tento časovač volá funkci *simulation.tick* implementovanou ve třídě *View*, která počítá nové pozice a rychlosti uzlů a obnovuje podle toho DOM. Protože je tato aplikace součástí aplikace TapirHQ, která je implementována s frameworkem React, který si manipuluje DOM sám, byla použita knihovna react-d3-library[2] která se stará o konverzi vygenerovaného DOM objektu knihovnou D3 do DOM objektu vhodného pro React. Časovač, který D3 používá pro obnovování DOMu přestává plnit svojí funkci protože obnovování DOMu obstarává React. Toto se projeví v nehezkou animaci s trhanými pohyby. D3 ovšem nabízí možnost simulaci krokovat pomocí funkce *tick*. Odkaz na tuto funkci je proto delegován skrz Controller až do React komponenty, kde je vytvořen časovač, který obnovuje stav komponenty, čímž automaticky říká Reactu aby aktualizoval DOM a zároveň volá tuto delegovanou funkci.

```
this.simulation = this.d3.forceSimulation()
            .nodes(this.nodeArray)
            .force("charge", this.d3.forceManyBody().strength(-50))
            .force("center", this.d3.forceCenter(this.width / 2, this.height / 2))
            .force("border", this.borderForce.bind(this))
            .force("link", this.d3.forceLink().id(function (d) {
                return (d.uri);
            })
                .links(this.linkArray)
                .distance(function (d) {
                    if((d.distance*3 + 10) < this.minimalLinkDistance){
                        return(this.minimalLinkDistance);
                    }
                    else{
                        return(d.distance*3+10);
                }
            }.bind(this)))
            .on("tick", this.ticked.bind(this));
```

V ukázkovém kódu je vidět jakým způsobem se definují síly, které výsledná simulace využívá k rozvržení uzlů. Každá síla je předána do simulace se svým pojmenováním *charge*, *center*, ukazatelem na funkci s implementací síly *this.borderForce* a s možnými argumenty, které upřesňují jak se bude síla chovat.

[2] http://react-d3-library.github.io/

6.3.1 Charge

Síla *d3.forceManyBody* ovlivňuje všechny uzly. Pokud je dodaný parametr *strength* kladný všechny uzly jsou k sobě přitahovány a pokud je parametr záporný, všechny uzly se navzájem odpuzují. Absolutní hodnota parametru *strength* odpovídá síle jakou na sebe uzly působí. Protože se ve výsledku snažím o co možná nejpřehlednější uspořádání uzlů, zvolil jsem parametr záporný aby bylo zaručeno, že každý uzel bude mít kolem sebe co možná největší volný prostor.

6.3.2 Center

Abychom udrželi celý graf pohromadě, D3 nabízí další ze svých funkcí *forceCenter*, která přitahuje všechny uzly do bodu definovaného jejími parametry. Pro přehledné rozložení je nejlepší aby tento bod byl ve středu objektu do kterého kreslíme, kterým je v tomto případě HTML objekt SVG s dimenzemi *[this.width, this.height]*.

6.3.3 Border

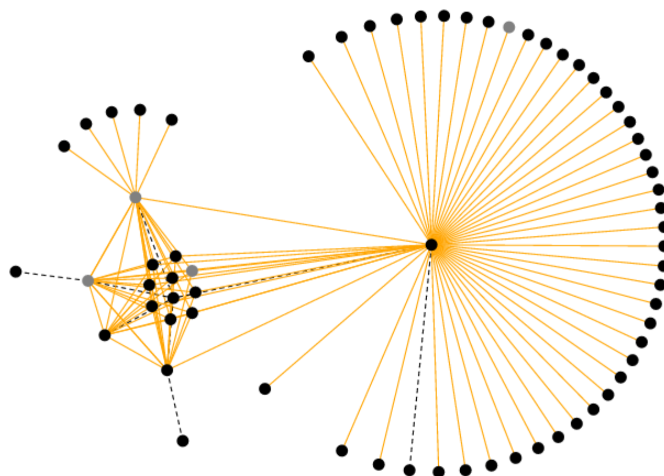
Síla border je mnou implementovaná funkce, která udržuje všechny uzly v mezích obalujícího SVG objektu.

```
private borderForce(alpha) {
    this.nodeArray.forEach(function (d) {
        if (d.x > this.width - this.nodeRadius) {
            d.x = this.width - this.nodeRadius;
            d.vx *= -1;
        }

        if (d.x < this.nodeRadius) {
            d.x = this.nodeRadius;
            d.vx *= -1;
        }


        if (d.y > this.height - this.nodeRadius) {
            d.y = this.height - this.nodeRadius;
            d.vy *= -1;
        }

        if (d.y < this.nodeRadius) {
            d.y = this.nodeRadius;
            d.vy *= -1;
        }
    }.bind(this));
}
```

Příklad kódu reprezentující sílu. Konkrétně se jedná o sílu, která drží všechny uzly uvnitř obdélníku.

6.3.4 Link

Síla link se stará o to, že uzly, které jsou v grafu spojeny hranou, na sebe působí přitažlivou silou nebo odpudivou silou, tak aby byla zachována vzdálenost mezi uzly, která je dodána jako parametr této funkce. Uzly spojené hranou se tedy chovají jako by mezi sebou měly pružinu, která jim nedovolí se dostat ani moc blízko sebe ani moc daleko od sebe.



Obrázek 6.3: Ukázka hotové mapy vygenerované z testovacích dat

Na obrázku je vidět výsledný vzhled mapy vygenerované z testovacích dat. Šedivé uzly značí stránky, které již tester navštívil. A přechody jsou také barevně i typově odlišeny. Černá čerchovaná značí přechod skrz formulář, ostatní reprezentují přechod normálním odkazem.

Kapitola 7

Testování

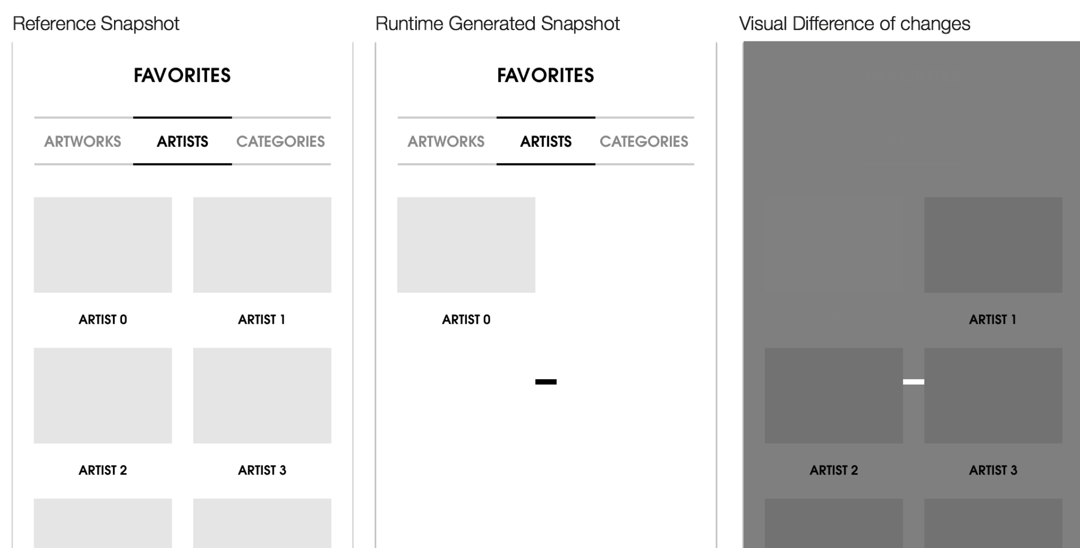
7.1 Úvod

V dnešní době se používají počítače už téměř všude a na běh programů a webových aplikací spoléhá spousta lidí. A ať chceme nebo nechceme, všichni děláme chyby. Některé chyby jsou zcela zanedbatelné a nikdy se na ně nepřijde a někdy mohou mít chyby velké následky. To samozřejmě platí i pro obor vývoje počítačových aplikací. Proto vznikl koncept testování softwaru, který z velké části pomáhá se vyvarovat zbytečným chybám.

7.2 Testování grafických komponent

Pro testování grafických komponent je dnes již velká řada nástrojů a postupů. Naivně by se zdálo, že grafické prvky půjdou testovat pouze beta testingem[1] ale není tomu tak. V dnešní době se již dají i grafické komponenty testovat automatickými metodami. Stejně jako klasické testy zdrojových kódů aplikací, které testují jednotlivé metody a porovnávají jejich vracené hodnoty s předem definovanými, očekávanými hodnotami, lze dnes porovnávat grafické výstupy aplikací s jejich požadovaným vzhledem. Tato metoda testování se nazývá *Snapshot testing*. Tato práce je však napsaná ve frameworku React, který jak bylo již zmíněno v předchozích kapitolách, využívá k vykreslování svého obsahu metodu *render*. Tato metoda již ve zdrojovém kódu obsahuje HTML elementy nebo odkazy na další komponenty a tyto elementy vrací jako svoji návratovou hodnotu. To nám dává možnost nechat si vrátit obsah určený k vykreslování do proměnné a tím i možnost otestovat tento kód bez nutnosti vykreslování.

[1] Testování aplikace lidmi



Obrázek 7.1: Příklad snapshot testu kde se zobrazilo méně objektů než mělo[2]

7.2.1 Testování s Jest a Enzyme

K testování aplikace jsme zvolil testovací framework Jest a Enzyme od společnosti Facebook. Tento framework disponuje třemi hlavními metodami k testování komponent. Všechny tři převádějí vygenerovaný DOM do textové podoby, kterou lze prohledávat pomocí funkcí, které vyhledávají různé html elementy podle jejich selectorů. Tyto tři metody, se liší počtem volání nativních funkcí Reactových komponent, které tyto metody volají.

- shallow: Metoda renderuje pouze jednu mateřskou komponentu a na této komponentě volá metody *constructor* a *render*.
- render: Volá pouze metodu *render*, ale vyrenderuje všechny komponenty vnořené do mateřské komponenty.
- mount: Metoda, která zavolá metody *constructor*, *render*, *componentDidMount*. Zároveň vyrenderuje, všechny vnořené komponenty.

Pomocí těchto metod, se dá otestovat každá klasická React komponenta. Knihovna, která utváří z kódu knihovny D3 Reactovou komponentu ale nevrací klasickou komponentu. Při použití kterékoliv z předchozím metod se vrací následující kód.

```
<div class="Map"><div><div class="react-component"><div></div></div></div><p><!-- react-text: 5
Nodes: <!-- /react-text -->
<!-- react-text: 6 -->0<!-- /react-text --></p><p><!-- react-text: 8 -->Links: <!-- /react-text
<!-- react-text: 9 -->
0<!-- /react-text --></p><p>
<!-- react-text: 11 -->URL: <!-- /react-text --><!-- react-text: 12 --><!-- /react-text --></p>
```

Tento kód zahrnuje pouze obalující *div* elementy a komentáře *react-text*, které react používá k manipulaci se svým DOMem. Tímto způsobem tedy bohužel nejde vygenerovanou mapu otestovat. Jediné testy, jsou možné na určení zda-li se komponenta opravdu začala renderovat, nebo pokud komponenta vrátila chybovou hlášku. Oba tyto testy se nachazí v souboru *../app/containers/ApplicationMap/tests/index.test.js*. Jsou spustitelné pomocí příkazu *npm run test-jest – app/containers/ApplicationMap/tests/index.test.js* . Zbytek testovacích scénářů jsem tedy otestoval jako betatester pouze kontrolou vykreslování v prohlížeči.

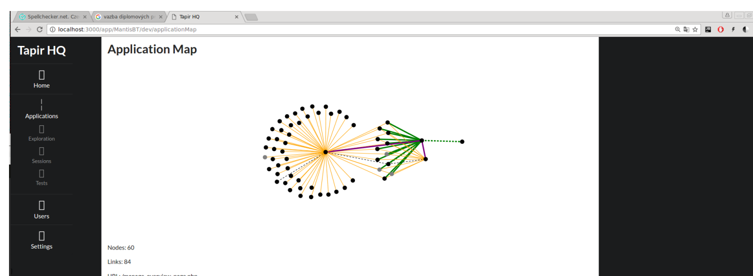
7.2.2 Testované případy

- Vykreslení jednoho uzlu
- Vykreslení dvou uzlů se stejným URL. Byl vykreslen pouze jeden uzel.
- Vykreslení dvou spojených uzlů.
- Vykreslení dvou spojených uzlů a dvou stejných hran

Kapitola 8

Závěr

V této práci jsem analyzoval dostupné JavaScriptové knihovny pro vizualizaci nehierarchických dat. Na základě kritérií a vhodnosti pro řešení dané úlohy jsem vybral knihovnu D3 pro implementaci vizuální komponenty. Tuto vizuální komponentu jsem neprogramoval v prostředí jazyku JavaScript a zakomponoval do již existující aplikace TapirHQ využívající JavaScriptový framework React. Toto bylo umožněno pomocí knihovny react-d3-library. Výslednou vizualizaci jsem upravoval pomocí aplikování sil na uzly grafu, které jsou součástí knihovny D3 a zároveň jsem i implementoval vlastní síly tak, aby bylo výsledné rozvržení uzlů co možná nejpřehlednější. Nakonec jsem zhodnotil možné testovací metody nové komponenty, vybral nejvhodnější způsob a pokusil se ho zrealizovat. Po zjištění, že tento způsob nedokáži realizovat jsem použil způsobu betatestingu a dotestoval zbývající testovací scénáře.



Obrázek 8.1: Celkový výsledek mé práce

Literatura

[1] C. Pojer. React tree snapshot testing. https://facebook.github.io/jest/blog/2016/07/27/jest-14.html.

[2] O. Therox. Snapshot testing. https://www.objc.io/issues/15-testing/snapshot-testing/.

[3] React test utilities. https://facebook.github.io/react/docs/test-utils.html.

Kapitola 9

Seznam použitých zkratek

HTML HyperText Markup Language

DOM Document Object Model

SVG Scalable Vector Graphics

D3 Data Driven Documents

JSON JavaScript Object Notation

npm Node.js package panager

⋮

Kapitola 10

Instalační a uživatelská příručka

10.1 Zprovoznění aplikace

Ke zprovoznění kódu je zapotřebí mít nainstalované prostředí Node.js[1] a program npm. V adresáří s daty projektu zadáme následující příkazy.

```
npm install
npm start

// var tty = require('tty');
```

10.2 Testování

Testování mapy se spouští pomocí příkazu:

```
npm run test-jest -- app/containers/ApplicationMap/tests/index.test.js
```

Je ale možné, že pří průběhu testu nastane problém s parsováním. V takovém případě je nutné přidat do kořenového adresáře aplikace soubor .babelrc, který má následující obsah:

```
{
  "presets": ["es2015", "react", "airbnb"]
}
```

10.3 Vývojové prostředí

Aplikace byla vyvíjena v IDE PhpStorm v. 2017.1 a v IDE IntelliJ IDEA ULTIMATE v. 2016.1.2 na platformě Linux Mint 17.2 KDE.
Aplikace byla spouštěna v prohlížeči Google Chrome ve verzi 57.0.2987.110.

[1] https://nodejs.org/en/

Verze npm : 3.10.10

Verze knihoven jsou uvedeny v souboru *package.json*.

Kapitola 11

Seznam přiložených souborů

```
./ApplicationMap/index.js
./ApplicationMap/routes.js
./ApplicationMap/components/Map/ClassesJs/Model/Node.js
./ApplicationMap/components/Map/ClassesJs/Model/DataStorage.js
./ApplicationMap/components/Map/ClassesJs/Model/DataReceiver.js
./ApplicationMap/components/Map/ClassesJs/Model/Link.js
./ApplicationMap/components/Map/ClassesJs/Controller/Controller.js
./ApplicationMap/components/Map/ClassesJs/Factories/NodeFactory.js
./ApplicationMap/components/Map/ClassesJs/Logger.js
./ApplicationMap/components/Map/ClassesJs/View/Board.js
./ApplicationMap/components/Map/Map.js
./ApplicationMap/components/Map/node.js
./ApplicationMap/reducer.js
./ApplicationMap/tests/index.test.js
```