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

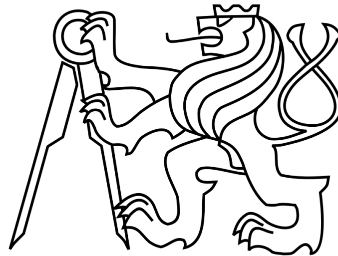
Platnost zadání: do konce letního semestru 2017/2018

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

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

10. ledna 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 09. 1. 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
- 5.4 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 Abstraction . . . 16
  - 6.2.2 Presentation . . . 16
  - 6.2.3 Controller . . . 16
- 6.3 Zobrazení dat . . . 16
  - 6.3.1 Charge . . . 17
  - 6.3.2 Center . . . 17
  - 6.3.3 Border . . . 18
  - 6.3.4 Link . . . 18

7 Testování **20**
7.1 Úvod 20
7.2 Testování grafických komponent 20
7.2.1 React Test Utilities[3] 21
7.2.1.1 21
7.2.1.2 Vykreslení jednoho uzlu 22
7.2.1.3 Vykreslení dvou stejných uzlů 22
7.2.1.4 Vykreslení dvou spojených uzlů 22
7.2.1.5 Vykreslení dvou spojených uzlů a dvou stejných hran 22

8 Závěr **23**

Literatura **24**

9 Seznam použitých zkratek **25**

10 Instalační a uživatelská příručka **26**

11 Seznam přiložených souborů **27**

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 Rozdělení aplikace podle vzoru PAC 15
6.3 Ukázka hotové mapy vygenerované z testovacích dat 19

7.1 Příklad snapshot testu kde se zobrazilo méně objektů než mělo[2] 21

8.1 Celkový výsledek mé práce 23

11.1 Seznam implementovaných souborů 27

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:

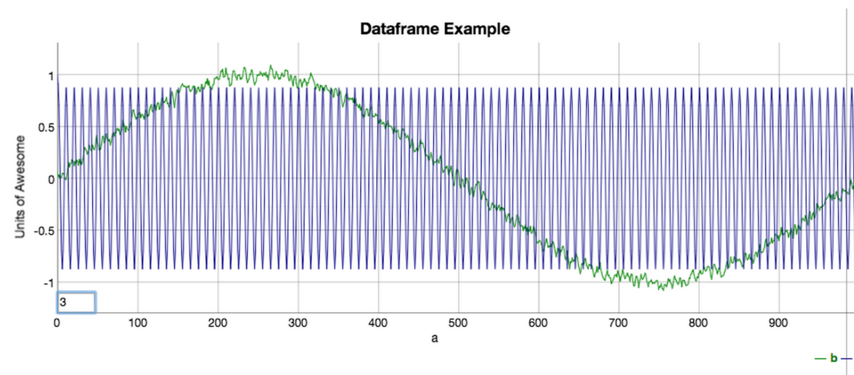
- schopnost vykreslovat nehierarchické struktury dat.
- jednoduchost a čitelnost zdrojového kódu.
- aktualizovanost knihovny
- možnost propojení s frameworkem React.

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,

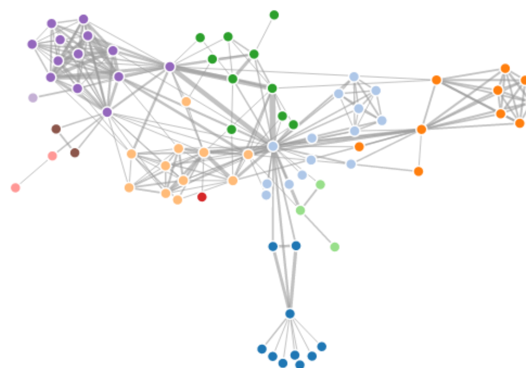


Obrázek 4.1: Ukázka DygraphJs

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.

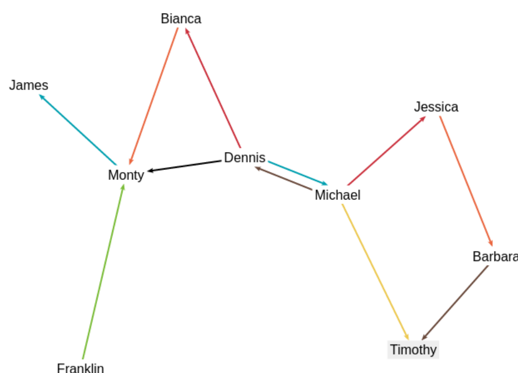


Obrázek 4.2: Ukázka knihovny D3 Js

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.

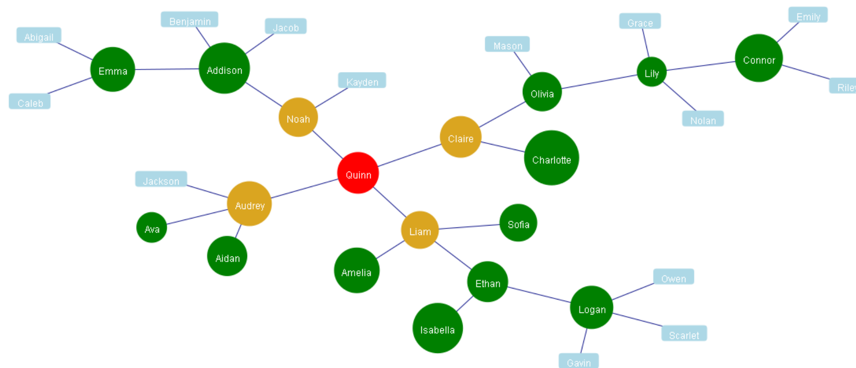


Obrázek 4.3: Ukázka knihovny SpringyJs

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.

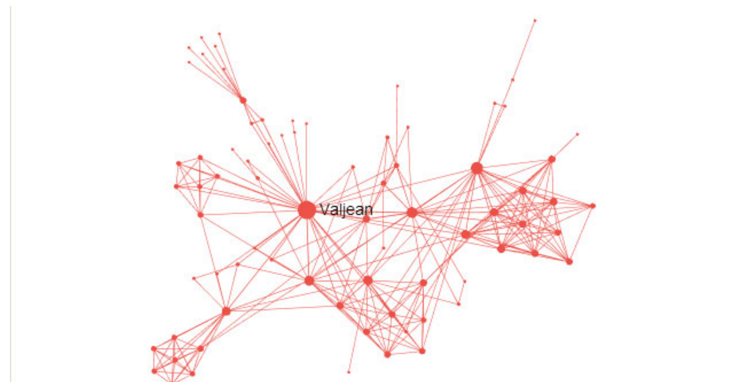


Obrázek 4.4: Ukázka knihovny ArborJs

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.

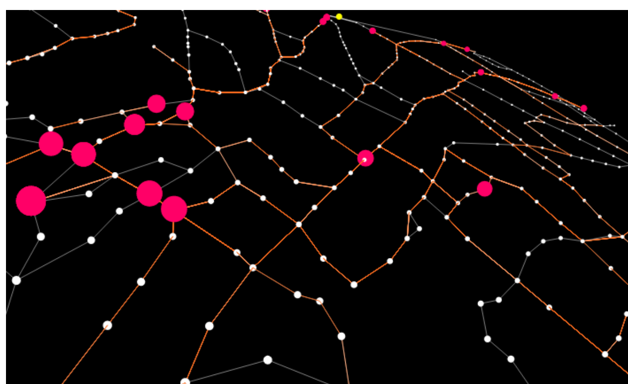


Obrázek 4.5: Ukázka knihovny SigmaJs

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 komponent a každý tento komponent je ve zdrojovém kódu reprezentován třídou. Všechny 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 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. Jako programátor, naučený na programovacích jazycích C a Java mi typová kontrola velmi vyhovuje. a to je důvodem, proč jsem se rozhodl svoji práci implementovat právě v tomto jazyce.

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].

- *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.

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

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 přístupny, byl přístup do databáze nahrazen čtením dat ze souborů, které odpovídají záznamům v databázi.
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.

[1] JavaScript Object Notation

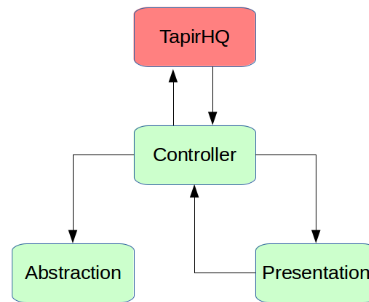
```
▼ _id {1}
    $oid : 57e374fe506e4b5a00a00278
  uri : /my_view_page.php
  hasPendingChanges : false
▼ created {2}
    date : 2016-09-22T06:06:54.1802577Z
  ▼ user {2}
      _id : kfrajtak
      name : Karel FrajtÃ¡k
  versionId : 57aa2499003029e88bea16e6
▼ forms [1]
  ▼ 0 {5}
      action : /jump_to_bug.php
      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 nejdříve vyvíjena, jako samostatný modul a až později byla zapracována do aplikace TapirHQ. K psaní aplikace byl použit návrhový vzor PAC[2].

Tento návrhový vzor rozděluje zdrojový kód do několika navzájem nezávislých částí a definuje, které části spolu mohou komunikovat.



Obrázek 6.2: Rozdělení aplikace podle vzoru PAC

[2] Presentation-Abstraction-Control (https://en.wikipedia.org/wiki/Presentation-abstraction-control)

Jak vidíme z obrázku aplikace je rozdělena na tři části, které hlouběji popíšeme v následujících kapitolách.

6.2.1 Abstraction

Tato vrstva se stará o data aplikace a všechny potřebné funkcionality s nimi spojenými. V této vrstvě jsou implementovány třídy, které slouží jako data předávaná knihovně D3, která je obalí

6.2.2 Presentation

- Tato vrstva se stará o zobrazování dat uživateli a zachytává všechno co uživatel v aplikaci chce udělat.

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 realizovany polem, ve kterém obě části předávájí té druhé odkazy na svoje metody.

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*, která počítá nové pozice a rychlosti uzlů a obnovuje 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[3] která se stará o konverzi vygenerovaného DOM objektu knihovnou D3 do DOM objektu vhodného pro React.

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

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.

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.

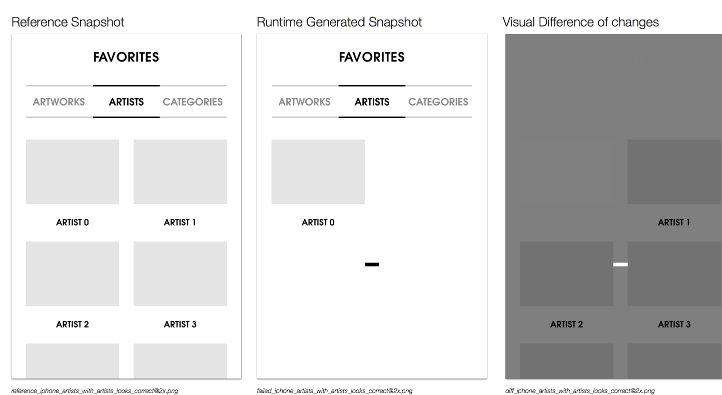
```
private borderForce(alpha) {
    this.nodeArray.forEach(function (d) {
        if (d.x > this.width - this.nodeRadius) {
            d.x = this.width - this.nodeRadius;
            d.vx *= -1;
        }

        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.

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 srkz formulář, ostatní reprezentují přechod normálním odkazem.



Obrázek 6.3: Ukázka hotové mapy vygenerované z testovacích dat

Kapitola 7

Testování

7.1 Úvod

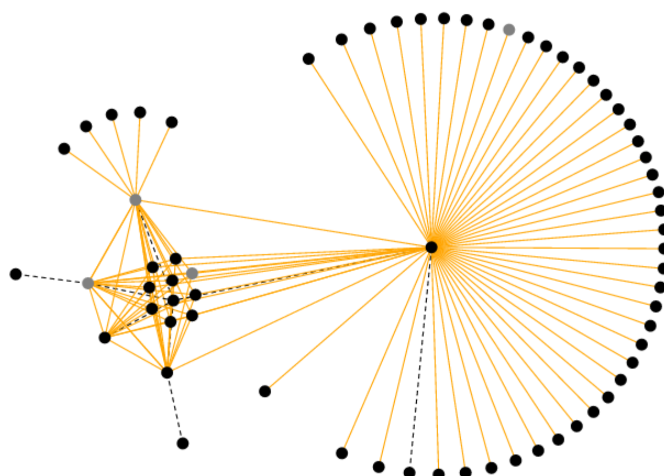
V dnešní době se používají počítače už téměř všude a na běh programů a webových aplikací spoléhá spousta lidí. A ať chceme nebo nechceme, všichni děláme chyby. Některé chyby jsou zcela zanedbatelné a nikdy se na ně nepřijde a někdy mohou mít chyby velké následky. To samozřejmě platí i pro obor vývoje 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 společnost facebook přišla se způsobem, testování na základě rozeznávání obrazu. Snapshot testing[1] je způsob testování aplikace pomocí porovnávání vizuálního vzhledu komponentů webové stránky ze screenshotu. Funguje tak, že testovací framework vyrenderuje obsah webové stránky do PNG obrázku. Poté co se kód změní vyrobí framework nový obrázek a porovná ho s původním. Pokud se liší, test neprošel. Tento způsob je však určen pro testování projektů jiného typu, než je poždé jiná mapa aplikace.

Běžné testování softwaru spočívá v tom, že vezmeme jednu funkci ze zdrojového kódu, uvědomíme si co všechno bychom chtěli aby uměla a otestujeme jí tak, že jí voláme s různými argumenty a porovnáváme vrácené hodnoty s hodnotami, které očekáváme. Tento princip porovnávání jde napojit i na grafický výstup webové stránky. Ten je totiž pouze projekcí HTML kódu. Tato aplikace je však implementována ve frameworku React, který tvoří výsledný HTML kód pomocí funkce *render*. To nám umožňuje využívat klasicé techniky jako při testování funkcí aplikace.

[1] Testování aplikace lidmi



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

7.2.1 React Test Utilities[3]

Pro testování aplikace jsem našel knihovnu *react-addons-test-utils*[2]. Zároveň nám dovoluje simulovat akce, jako kliknutí myši, které v normálním běhu aplikace mohou přijít pouze od uživatele.

Testované případy Dříve než začneme něco testovat je důležité si uvědomit, co od finální aplikace požadujeme. Proto vytvoříme seznam testovacích scénářů. Pro testování použijeme knihovnu *react-addons-test-utils*, která je schopná vyrenderovat html kód koponenty do objektu, který poté můžeme zkoumat. Při každém testu nastavíme hodnoty proměnné *props* tak, aby se na stránce objevilo, co potřebujeme. Uděláme odhad a poté porovnáme data v DOMu. Vzhledem k tomu, že vizualizace je pokaždé jiná nemůžeme hledat přesné shody mezi dvěma obrazy DOMu. Proto navrhuji počítat HTML objekty spojené s daty z vizualizace. To znamená HTML objekt *line* pro hrany a *circle* pro uzly. Pokud by to bylo zapotžebí můžeme zjemnit naše testování až na atributy daných objektů.

7.2.1.1

HTML objekty *line* mají atributy *class* sX tY lZ. Kde X, Y, Z jsou přirozená čísla.

- sX - X je index uzlu, ze kterého hrana vychází.
- tY - Y jen index uzlu, do kterého hrana míří.
- lZ - Z je typ hrany. Pokud hrana reprezentuje skok na další stránku skrz formulář má hodnotu 0. Pokud z klasického odkazu tak 1. A poku je možnost obou variant, tak hodontu 2.

[2]https://facebook.github.io/react/docs/test-utils.html

7.2.1.2 Vykreslení jednoho uzlu

- Vstupní hodnota: Data pro jeden samotný uzel.
- Očekávaná hodnota: Jeden HTML objekt *circle*

7.2.1.3 Vykreslení dvou stejných uzlů

Tento test testuje, že pokud nám v datech přijde uzel se stejnými hodnotami, naše aplikace ho nevykreslí dvakrát.

- Vstupní hodnota: Dvakrát data se stejným uzelm
- Očekávaná hodnota: Jeden HTML objekt *circle*

7.2.1.4 Vykreslení dvou spojených uzlů

- Vstupní hodnota: Data pro dva uzly a jednu hranu
- Očekávaná hodnota: Jeden HTML objekt *line* a dva objekty HTML circle.

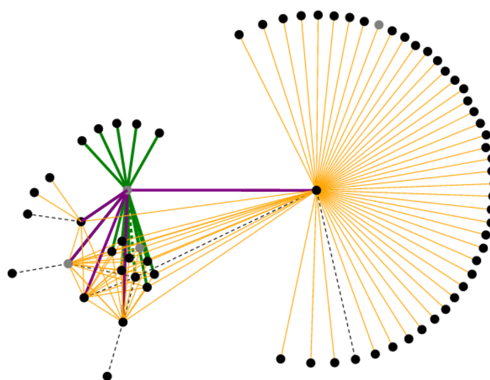
7.2.1.5 Vykreslení dvou spojených uzlů a dvou stejných hran

- Vstupní hodnota: Data pro dva uzly a dvě stejné hranu
- Očekávaná hodnota: Jeden HTML objekt *line* a dva objekty HTML circle.

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 umožňěno 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. A nakonec jsem navrhnul způsob, kterým se aplikace může testovat.



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

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

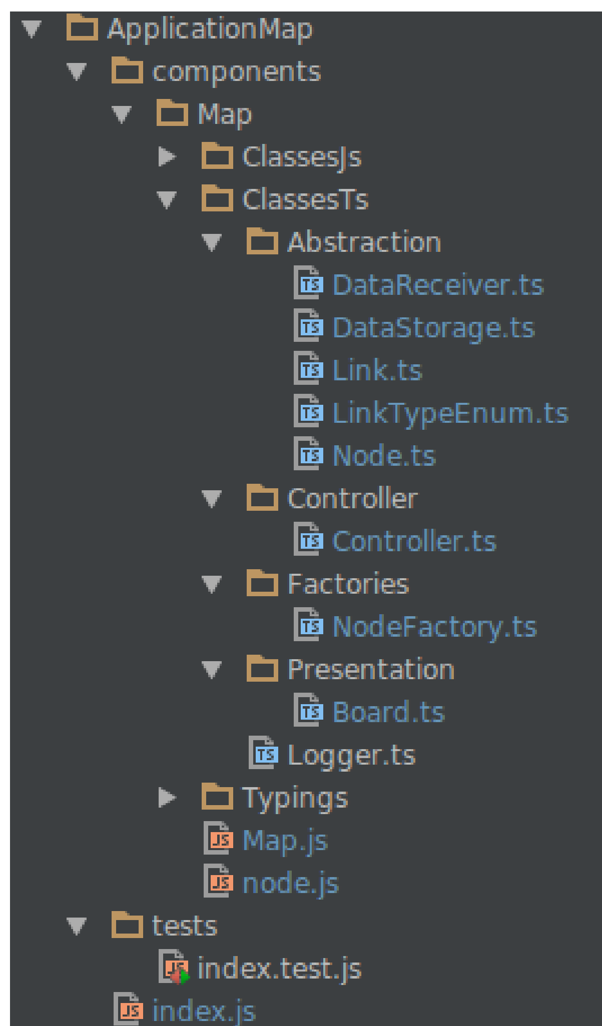
```
npm install
```

Potom co byly všechny balíčky nainstalované, musí se jediný soubor z adresáře hack zkopírovat místo souboru ./node_modules/debug/src/node.js.

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

Kapitola 11

Seznam přiložených souborů



Obrázek 11.1: Seznam implementovaných souborů