Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Portál diabetes.zcu.cz

Plzeň 2017 Marek Rasocha

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 20. 6. 2017

Marek Rasocha

Poděkování

Touto cestou bych rád poděkoval vedoucímu této diplomové práce doc. Ing. Tomáši Koutnému Ph.D. za trpělivost, cenné rady při vedení práce a za čas, který mi při konzultacích věnoval.

Abstract

This thesis extends functionality of the diabetes.zcu.cz portal. The thesis focuses on web architecture, Springs, Apache Wicket and Angular frameworks, the portal's architecture, implementation detail and testing on both PC and mobile devices.

Abstrakt

Tématem této diplomové práce je rozšíření portálu diabetes.zcu.cz. Tato práce nejprve čtenáře seznamuje s některými webovými architekturami. Je představen framework Spring, Apache Wicket, k němuž alternuje Angular. Postupně je čtenáři představena architektura portálu. Praktická část zahrnuje implementaci změn včetně použití knihovny Emscripten. Na závěr je aplikace otestována na desktopu a mobilním telefonu s operačním systémem Android.

Obsah

Seznam obrázků

Seznam zkratek

AGP	Ambulatory Glucose Profile
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AWT	Abstract Window Toolkit
CGMS	Continuous Glucose Monitors
CSS	Cascading Style Sheets
CSV	Comma-separated values
DAO	Data Access Object
DOM	Document object model
EMCC	Emscripten Compiler Frontend
GUI	Graphical User Interface
HQL	Hibernate dotazovací jazyk
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ICD	International Code Designator
IE	Internet Explorer
IoC	Inversion of control
JavaEE	Java enterprise edition
JS	JavaScript
JSON	JavaScript Object Notation
JSP	JavaServer Pages
LLVM	Low Level Virtual Machine

MVC	Model-View-Controller
ORM	Object-relational mapping
SQL	Structured Query Language
SVG	Scalable Vector Graphics
TBB	Threading Bulding Block
URL	Uniform Resource Locator
XHTML	Extensible hypertext markup language
XML	Extensible Markup Language

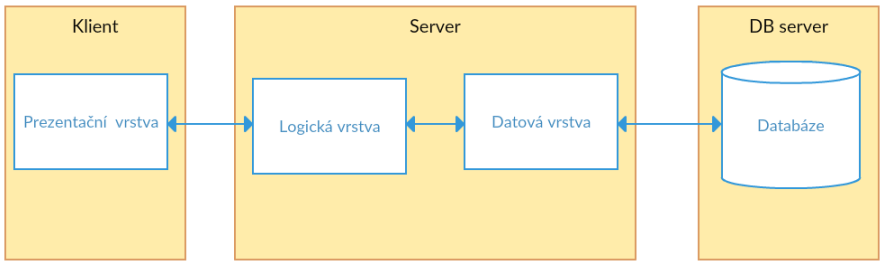
1 Úvod

Tématem této práce je rozšíření portálu diabetes.zcu.cz, jehož úkolem je počítat koncentraci glukózy v krvi pomocí dat naměřených prostřednictvím CGMS senzoru. Tento senzor nosí pacient s podezřením na diabetes určitou dobu a během ní mu je kontinuálně snímána koncentrace glukózy v podkoží a sporadicky (např. třikrát za den) koncentrace glukózy v krvi. Současná implementace neposkytuje dostatečnou vizualizaci výsledků a dostatečné možnosti při nahrávání souboru. Je implementováno pouze základní vykreslování naměřených a vypočtených křivek. Výsledkem této práce bude rozšíření webové aplikace o pokročilejší interpretaci výsledků, implementace stromového filtru a dalších funkcionalit, které bude vedoucí práce požadovat. Důležitým požadavkem je, aby aplikace byla nadále responzivní. To znamená, že místo toho, aby si webovou stránku upravoval uživatel pomocí přibližování, oddalování či posouvání, se stránka přizpůsobí jemu.

Teoretická část práce představí portál, jeho současnou funkcionalitu a základní moduly jeho architektury. Čtenáře seznámí s architekturami webových aplikací, ve kterých je nastíněn princip vícevrstvé architektury a architektonický vzor MVC. Práce vysvětluje princip IoC, představí jeho výhody a také to, jak s IoC pracovat ve frameworku Spring. Dále se teoretická část zabývá frameworky, které lze použít pro implementaci prezentační vrstvy aplikace, jako je Angular a Apache Wicket.

V praktické části se zanalyzují možnosti přenosu serverových částí na stranu klienta. Tato analýza se stane základem pro následnou implementaci tohoto přenosu, k němuž se použije knihovna Emscripten. Dále se implementuje požadovaná rozšiřující funkcionalita, která zvýší použitelnost aplikace.

Na závěr práce se veškerá nová funkcionalita otestuje a důkladně zdokumentuje. Testování aplikace musí probíhat alespoň na desktopu a mobilním zařízení a to minimálně na aktuálních verzí prohlížečů Firefox, Chrome, Internet Explorer, Edge a Opera.



Obrázek 1: Třívrstvá architektura

2 Architektury webových aplikací

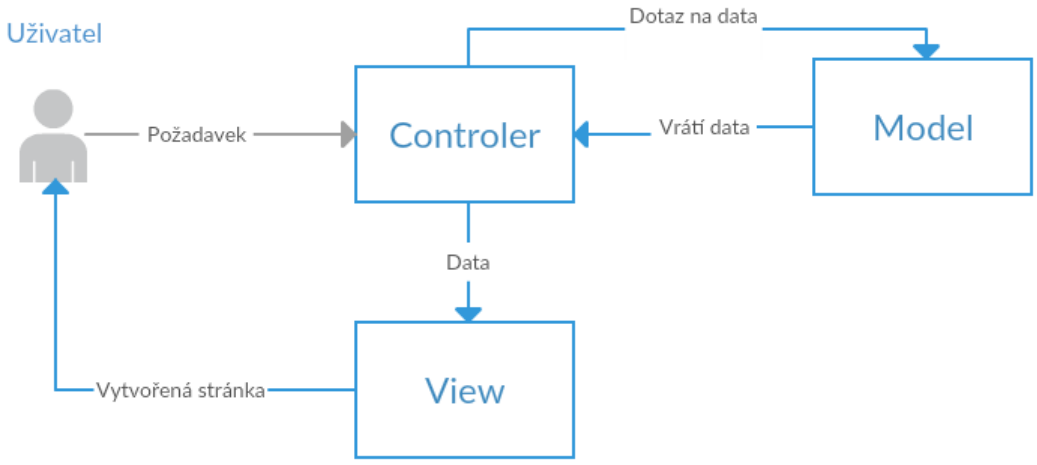
Softwarová architektura je proces definování strukturovaného řešení splňující všechny technické a provozní požadavky[1]. Zvolení vhodné architektury zahrnuje řadu rozhodnutí, která mají značný vliv na kvalitu, výkonnost, udržitelnost a celkový úspěch aplikace. Webová aplikace je aplikace, která je uživateli přístupná přes webové rozhraní. To je obvykle přístupné přes webový prohlížeč. Webový prohlížeč vytvoří konkrétní HTTP požadavek, který pošle na určitý URL server. Server vygeneruje HTML stránku, kterou pošle zpět v odpovědi. Webový prohlížeč pak tuto stránku zobrazí.

2.1 Vícevrstvá architektura

Vícevrstvá architektura umožňuje vývojáři vytvářet flexibilní a znovupoužitelné aplikace [1]. Aplikace je rozdělena do vrstev. Jednotlivé vrstvy komunikují přes rozhraní. Výhodou této architektury je, že se aplikace lépe udržuje, vyvíjí a rozšiřuje. Není třeba modifikovat celou aplikaci naráz, nýbrž po jednotlivých vrstvách. Nejčastěji se používá třívrstvá architektura, obr. 1, ve které je aplikace rozdělena do prezentační, aplikační a datové vrstvy.

K prezentační vrstvě uživatelé přistupují přímo (například pomocí grafického rozhraní). Zajišťuje vstup požadavků a prezentaci výsledků. Tato vrstva je závislá na platformě (webová aplikace, Windows aplikace, mobilní aplikace atd.). Aplikační vrstva tvoří logiku celé aplikace. Zajišťuje výpočty, zpracování dat a operace s nimi. Datová vrstva obsahuje mechanismy pro uchovávání perzistentních dat (souborový systém, databáze) a mechanismy pro přístup k datům.

Výhodou obecné vícevrstvé architektury je nezávislost jednotlivých vrstev. Každá může být implementována nezávisle, může běžet potencionálně na jiném stroji. Výhodou je také možnost znovupoužití komponent.



Obrázek 2: MVC

2.2 MVC

MVC (model-view-controller) je architektonický vzor běžně používaný k implementaci uživatelského rozhraní [1]. Z toho důvodu je často používán pro návrh webových aplikací.[2] Webovou aplikaci rozdělí do tří základních částí:

- model - obsahuje logiku aplikace. V modelu se provádí veškerá důležitá činnost, včetně validace dat. Validace dat nesmí v této části aplikace chybět, a to ani v případě kontroly dat ve view.
- view - zobrazuje data uživateli. Může se jednat o HTML stránku s tagy značkovacího jazyka (např. Wicket, JSP). Zde není validace dat povinná, nicméně je vhodná. Je uživatelsky příjemná a v případě nevalidních dat zabraňuje zbytečnému posílání požadavku na server.[4]
- controller - zabývá se tokem dat v aplikaci. Propojuje model, view a uživatele.

Aplikace funguje tak, že uživatel zadá URL adresu do webového prohlížeče. Ten pošle požadavek, který přijde do controlleru. Ten podle parametrů identifikuje metodu, která se bude vykonávat, a zavolá ji. Data potřebná ke zpracování požadavku zjistí z modelu. Controller musí poslat požadavek na model. Model vyhledá potřebná data a vrátí je controlleru. Controller předá vygenerovaná data view vrstvě, která je vloží do šablony a pošle odpověď uživateli. Uvedený princip je znázorněn na obr. 2.

[1] prezentační vrstvy třívrstvé architektury

2.3 Apache Wicket

Apache Wicket je open source framework pro tvorbu webových aplikacích v programovém jazyce Java. Byl vytvořen v roce 2004. Jeho autory jsou Jonathan Lockea a Miko Mastsumura.[3]

2.3.1 Vlastnosti

Apache Wicket je komponentově orientovaný framework. Komponentový framework se od klasických (např. JSP) liší tím, že na straně serveru generuje model požadovaných stránek. HTML posílané zpět klientovi je získáváno právě z tohoto modelu. Může se mluvit o modelu, který je podobný „inverznímu“ Javascriptovému DOM[2], což znamená[3]:

- je generován na straně serveru,
- je sestaven před tím, než je HTML posláno klientovi,
- HTML je generováno z modelu.

U komponentových frameworků jsou webové stránky a jejich HTML komponenty reprezentovány instancemi tříd. Při vytváření webové aplikace je možné použít objektově orientovaný přístup. Což umožňuje použít dědičnost, skládání objektů a například i efektivnější testování. Komponentově orientované frameworky přinesly do vývoje webových aplikacích stejný druh abstrakce, které poskytují GUI frameworky pro vývoj desktopových aplikací (např. Swing, AWT[3]). Většina těchto webových frameworků skrývá podrobnosti o HTTP protokolu [4].

Jak bylo řečeno, Wicket funguje podobně jako AWT. V AWT existuje *Windows* instance reprezentující fyzické okno. Do této instance jsou vkládány prvky uživatelského rozhraní (přepínače, tlačítka, kreslící okna atp.). Ve Wicketu fyzickou stránku reprezentuje instance *WebPage*, která obsahuje různé HTML komponenty (obrázky, formuláře atd.). V obou frameworcích je základní třída *Component*, která zapouzdřuje všechny prvky grafického rozhraní. Stránka (resp. okno) je pak skládána z různého množství komponent. Takové frameworky poskytují velkou škálu již připravených komponent a samozřejmě umožňují jejich znovupoužitelnost.

[2] Document object model
[3] Abstract Window Toolkit

2.3.2 Struktura aplikace

Wicket aplikace je standardní Java EE[4] webová aplikace, která je nasazena pomocí *web.xml* souboru. Tento soubor musí deklarovat wicket filtr, který odesílá požadavky do wicketovské aplikace. Nastavuje se zde parametr *applicationClassName*, který musí odkazovat na potomka třídy *Application*. Tato třída je zodpovědná za konfiguraci aplikace a nastavení domovské stránky.

Wicket může být spuštěn v módu *Development* nebo *Deployment*. První mód zpřístupňuje speciální vlastnosti ulehčující vývoj aplikace jako je AJAX debugger, vykreslování výjimek, reloading atd. Druhý mód tyto vlastnosti vypne. Nasazení aplikace do produkce musí být vždy v *deployment* režimu. Přepnutí režimů je možné jak ze souboru web.xml, tak i z hlavní aplikační třídy.

Stránka je zde reprezentována potomkem třídy *WebPage*, která musí mít korespondující HTML soubor. Wicket soubor používá jako šablonu k vygenerování jejího HTML. Třída a HTML soubor musí mít shodné jméno a ve výchozím nastavení musí být ve stejném balíčku. Java kód se používá pro psaní logiky aplikace (Controller) a HTML znázorňuje grafickou část aplikace (View). K provázání těchto dvou vrstev se používají tzv. wicket značky, které se vkládají do HTML souboru. HTML obsahující takovéto značky se říká markup. Princip provázání obou vrstev je znázorněn na jednoduchém příkladu, viz obr 3.

Je vytvořena třída *HomePage.java*, která dědí od *WebPage*. V konstruktoru této třídy je přidán štítek s identifikátorem *id* a s textem „Hello WicketWorld!“. Do stránky se vloží metodou *add*. Dále je vytvořen soubor *HomePage.html*, který obsahuje standardní HTML značky. Za zmínku stojí div, který obsahuje atribut *wicket:id*. Hodnota tohoto atributu je shodná s identifikátorem vkládaného štítku v javovské části aplikace (v tomto případě *id*). Tímto dojde k provázání vrstev a do výsledného HTML je vložen zadaný text. Pokud odpovídající *wicket:id* identifikátor není provázán s Javou, tak je vyhozena výjimka *ComponentNotFound Exception*. Výjimka *WicketRuntimeException* je vyhozena právě tehdy, když v markupu není nalezen identifikátor, který je komponentě přiřazen v Javě.

Vzhledem ke komponentovému přístupu je nutné zdůraznit trochu jiný význam odkazů oproti odkazům známých z HTML. V HTML je to odkaz na jiný zdroj (většinou jinou stránku). Zde je odkaz považován za klikací událost. Odkaz je reprezentován třídou *Link*, která obsahuje abstraktní metodu *onClick*. Tuto metodu je nutné

[4]Java enterprise edition

HomePage.java

```
public class HomePage extends WebPage {
   public HomePage() {
      add(new Label("id", "Hello WicketWorld!"));
   }
}
```

HomePage.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Apache Wicket HelloWorld</title>
    </head>
    <body>
        <div wicket:id="id"/>
    </body>
</html>
```

Obrázek 3: Jednoduchý příklad

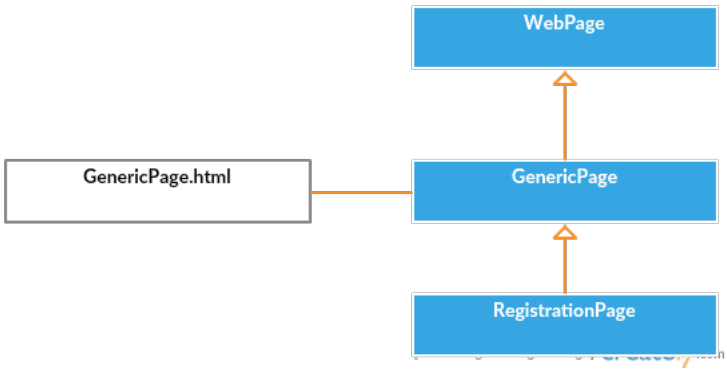
přepsat a tím nadefinovat funkčnost tlačítka. Ve výchozím nastavení server vrací aktuální stránku. Pro přesměrování na jinou stránku se musí použít metoda *setResponsePage*:

```
public class HomePage extends WebPage{
   public HomePage(){
      add(new Link("id"){
          @Override
          public void onClick() {
              setResponsePage(AnotherPage.class);
          }
          });
   }
}
```

2.3.3 Dědičnost

Vzhledem k tomu, že Wicket při vývoji webové aplikace umožňuje objektově orientovaný přístup, je možné využívat i dědičnost. Tu lze využít například při jednotném rozložení stránek. Pro ilustraci je vytvořena generická stránka *GenericPage* a k ní korespondující markup. Dále je vytvořena stránka *RegistrationPage*, která dědí od *GenericPage* a nemá odpovídající markup. V takovém případě je jako markup použit *GenericPage.html*, obr. 4. Tento jev se nazývá markup dědičnost.

Speciální význam má komponenta panel. Panel umožňuje opětovné použití komponent, a to jak na jednotlivých stránkách, tak i na úrovni různých aplikací. Výhodu znovupoužití lze demonstrovat na jednoduchém příkladu. Je vytvořeno menu, které se objevuje na několika stránkách. Pokud je implementováno jako panel, tak je možné ho vkládat na více stránek s jedním markupem. Je-li potřeba menu upravit,



Obrázek 4: Markup dědičnost

Rodič	Potomek	Výsledný markup
<body> Rodič <wicket:child/> </body>	<wicket:extend> Dítě </wicket:extend>	<body> Rodič <wicket:child><wicket:extend> Dítě </wicket:extend></wicket:child> </body>

Obrázek 5: Dědičnost se značkami

tak ho stačí změnit pouze uvnitř daného panelu. Ke každému panelu je asociován právě jeden markup, který musí být umístěn ve značce *wicket:panel*. V panelech lze opět využívat markup dědičnost.

Pří dědění je možné využít i jiný princip, ke kterému je potřeba použít značky *wicket:child* a *wicket:extend*. *Wicket:child* je umístěn v rodičovské třídě a definuje, na jaké místo bude vložen markup z potomka. Markup rodičovského potomka musí být vložen mezi *wicket:extend* značky. Použití těchto značek spolu s výsledným markupem je k vidění na obr.5.

2.3.4 Model

Model je rozhraní, které umožňuje komponentám přistupovat nebo modifikovat své údaje, aniž by znaly, jak jsou udržovány nebo uchovávány[3]. Každá komponenta má nanejvýš jeden model. Každý model může být sdílen mezi více komponentami. Ve Wicketu je model implementace rozhraní *IModel* definující dvě metody: *getObject* a *setObject*, jejichž význam je zřetelný z názvu. Každá komponenta může nastavit/-vrátit svůj model. Komponenta obsahuje metody, které se volají při každé modifikaci

modelu. Výhodou modelů je to, že umožňují dynamicky měnit komponenty. Je-li použit příklad z obrázku 3, tak text ve štítku už měnit nelze. Jedná se o statický text. Pokud má být text dynamický (internacionalizace nebo jen prostá změna textu), tak musí být použit model:

```
public class MyPanel extends Panel{
        String str;
        public MyPanel(String id){
                super(id);
                ...
                add(new TextField("id",new Model(str)));
        }
}
```

Zpočátku není v tomto použití zřejmý žádný problém. Ten však nastává v okamžiku, kdy se do proměnné *str* uloží jiný objekt. Model totiž neustále udržuje referenci na předchozí místo v paměti. To bude mít za následek to, že *TextField* bude zobrazovat zastaralou hodnotu *str*. Tento problém lze vyřešit velmi jednoduše [4]:

```
public class MyPanel extends Panel{
        IModel<String> model;
        public MyPanel(String id){
                super(id);
                ...
                add(new TextField("id",model));
        }
}
```

V tomhle případě si komponenta udržuje referenci na model. Při změně hodnoty modelu se aktualizuje i text, který je vykreslen. Nastavení nové hodnoty modelu se provádí přes metodu *setObject*.

Ve Wicketu existuje několik druhů modelů:

- model - nejjednodušší model sloužící k ukládání statických dat. Nemá žádné speciální vlastnosti.
- stringResourceModel - model, který slouží k internacionalizaci. Načítá lokalizované řetězce dle jejich klíče. Řetězce jsou umístěny v *properties* souboru.

- propertyModel - model, který pracuje s daty ze specifikovaného objektu a jeho atributu. Například uvedený model vrací daný věk osoby.

```
Person person = loadPerson();
new Label("name", new PropertyModel(person, "age"));
```

Tento model dokáže přistupovat k privátním atributům a je null-safe[5]. Lze ho také využít pro formulářové operace, ve kterých může měnit hodnoty pole věk. Při odeslání formuláře na server (tzv. submit) je objekt a atribut věk modifikován na zadanou hodnotu. Důležité je zadanou hodnotu na straně serveru validovat.

- compoundPropertyModel - speciální druh modelu, který využívá dědění. Pokud komponenta potřebuje model, ale nemá ho inicializovaný, tak hledá v rodičovských komponentách model implementující rozhraní *IComponentInheritedModel*. Jakmile ho získá, chová se jako property model.

- loadableDetachableModel - jedná se o model, u kterého je třeba specifikovat metodu *load*. Tato metoda načte všechna data. Na konci zpracování požadavku se volá metoda *detach*, která vyčistí reference na daný objekt.

2.3.5 AJAX

AJAX (Asynchronous JavaScript and XML) mění obsah stránek bez nutnosti jejich kompletního znovunačtení za pomocí asynchronní komunikace se serverem. Webovým aplikacím přináší lepší výkon a interaktivitu[5]. AJAX zahrnuje:

1. standardizovanou prezentaci pomocí XHTML a CSS,

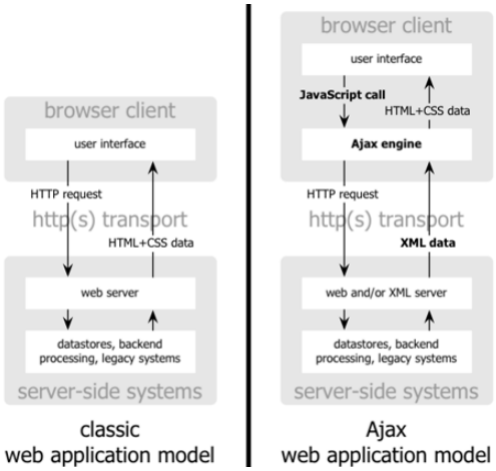
2. dynamické zobrazení a interakci s DOM,

3. výměna dat pomocí XML,

4. asynchronní načítání dat pomocí HTTP požadavku[5].

Javascript propojuje vše dohromady. Klasický webový model funguje tak, že je poslán požadavek na server. Ten klientovi vrátí odpovídající stránku. Tento model snižuje interaktivitu, protože vynucuje překreslení stránky při každém znovunačtení dat. Aplikace používající AJAX vytvoří JS engine, které běží na prohlížeči[5]. Prohlížeč místo webové stránky načítá tento engine, který zobrazí požadovaný obsah.

[5] Ošetřuje nullException



Obrázek 6: Srovnání Ajax modelu s klasickým modelem[5]

Engine pak zajišťuje veškerou interakci s uživatelem (např. ověření dat). Pokud engine potřebuje více dat ze serveru, tak je získá na pozadí bez uzamčení uživatelského rozhraní[5]. Tímto je dosaženo lepší interaktivity aplikace. Rozdíl mezi těmito přístupy je vidět na obr. 6.

Wicketovské AJAX komponenty obsahují zpětná volání (tzv. callbacky), která jsou provedena, když komponenty přijmou určitý AJAX požadavek. Jedním z parametrů těchto metod je rozhraní *AjaxRequestTarget*. Jedna z důležitých metod tohoto rozhraní je *add*, která Wicketu řekne, aby komponentu znovu překreslil s aktualizovaným markupem. Jakým způsobem se zaktualizuje štítek po stisku na odkaz, je zřejmé na následujícím příkladu.

```
Label label = new Label("id", "init");
label.setOutputMarkupId(true);
add(label);
add(new AjaxLink("ajax"){
        @Override
        public void onClick(AjaxRequestTarget target) {
                label.setDefaultModelObject("Change");
            target.add(label);
        }
});
```

Za zmínku stojí to, že překreslovaná komponenta musí mít nastavenou vlajku *outputMarkupId* na *true*. Komponenty jsou vykreslovány podle jejich *id* značky. Vý-

sledkem této ukázky je to, že štítek je modifikován na jinou hodnotu a je překreslen. Pomocí rozhraní *AjaxRequestTarget* lze volat i funkce JavaScriptu.

Důležitou poznámkou je to, že nelze překreslovat všechny komponenty. Nelze překreslovat komponenty opakovačů (*ListView, RepeatingView*...). To lze obejít tím, že je opakovač zaobalen do *WebMarkupContainer*. Tento kontejner překreslit již lze.

2.3.6 Shrnutí

Wicket je komponentový framework podporující znovupoužití a objektově orientovaný přístup. Pro vývojáře je velice příjemný v tom, že vývoj webové aplikace je podobný vývoji desktopové aplikace. Další pozitivem je to, že podporuje několik dalších webových technologií. Ať už se jedná o AJAX, Bootstrap (prostřednictvím knihovny wicket:bootstrap), jQuery či přímá integrace s frameworkem Spring.

2.4 Angular

Angular je framework pro vytváření klientských aplikacích v HTML a JavaScriptu (JS). Byl vytvořen v roce 2009 společností Google[11].

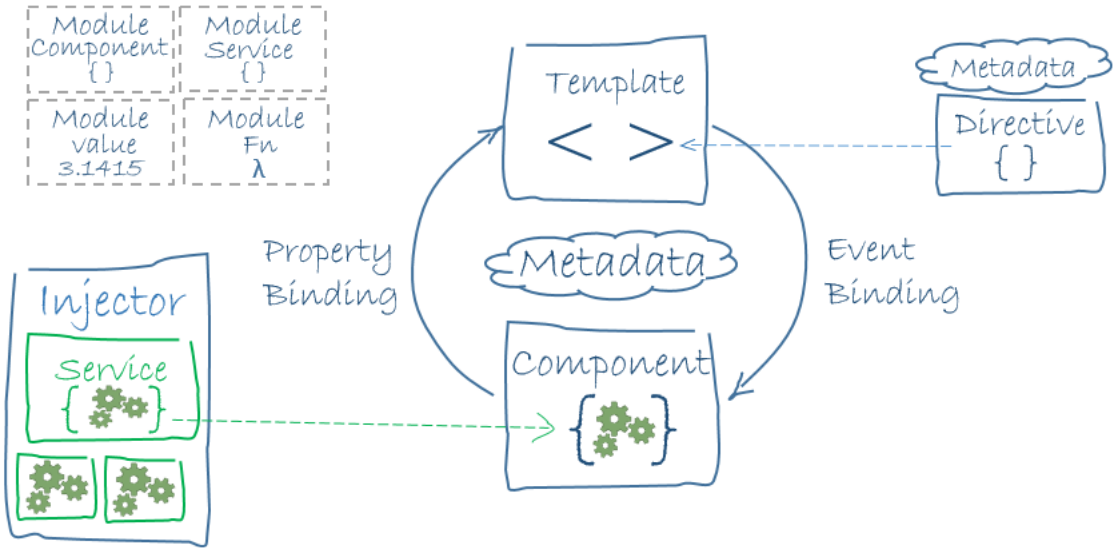
2.4.1 Architektura

Angular využívá komponentový přístup, jehož výhody již byly zmíněny v kapitole 2.3. Aplikace využívá návrhový vzor IoC ke vkládání závislostí do tříd, viz kapitola 2.5.1. Aplikace dodržuje principy MVC architektonického vzoru.

Aplikace se skládá z modulů, služeb, komponent, HTML šablon a určitých direktiv a metadat. Jejich podrobný význam je popsán v dalších kapitolách. Na obrázku 7 jsou zobrazeny části angularské aplikace a komunikace mezi nimi. Hlavní složkou jsou komponenty, která mohou obsahovat metadata. Šablony jsou tvořeny HTML a direktivami, které značkám přiřazují specifické vlastnosti. Šablony s komponenty spolu komunikují pomocí speciálního systému. Do komponent lze natáhnout služby.

Modul

Angularská aplikace obsahuje vlastní modulární systém nazývaný *NgModules*. Tento systém pomáhá organizovat funkčnost aplikace do soudržného bloku. Každá aplikace má minimálně jeden modul, který tvoří kořen aplikace. Aplikace dále může obsahovat další rozšiřující moduly (tzv. feature moduly), které se věnují aplikační



Obrázek 7: Přehled angularské aplikace[11]

doméně. Další důležitou části Angularu jsou tzv. dekorátory. Jsou to funkce modifikující JS třídy. Dekorátor připojí do třídy metadata, aby Angular věděl, jak třídy fungují a co znamenají [11]. Důležitým dekorátorem je *@NgModule*, který slouží pro popis vlastností modulů (jak kořenových tak rozšiřujících). Mezi důležité vlastnosti uvedeného dekorátoru patří:

- deklarace - deklaruje, jaké třídy náleží do tohoto modulu,
- export - množina deklarací, které mohou být použitelné v jiných modulech,
- import - vložení dalších modulů, jejichž exportované třídy jsou potřeba v šabloně importujícího modulu,
- provider - poskytovatelé služeb,
- bootstrap - hlavní aplikační pohled, který může nastavit jen kořenový modul.

Třída označená dekorátorem *@NgModule* říká Angularu, jak má zkompilovat a spustit kód. Angular identifikuje modulové komponenty, direktivy a pipy, které může zviditelnit pro ostatní komponenty atd. JS poskytuje svůj modulární systém, který je rozdílný a nezávislý na systému Angularu[11]. V JS je každý soubor brán jako modul. V souboru lze definovat objekty, které lze označit jako veřejné[6]. Zpřístupnění veřejných objektů se provede importem daného modulu (souboru).

[6] klíčovým slovem public

Je evidentní, že se jedná o dva rozdílné přístupy a autoři Angularu doporučují při psaní aplikace používat oba systémy [11].

Komponenta

Další součástí aplikace je komponenta, která se stará o vykreslení view vrstvy. Zde je definována aplikační logika komponenty, která je uvnitř nějaké třídy např. *HeroListComponent*. Tato třída komunikuje s view prostřednictvím API metod [11]. Angular má pak pod kontrolou vytváření, aktualizování a odstraňování komponent. To je samozřejmě plně závislé na chování uživatele. Komponenty však musí mít odpovídající HTML šablonu, která obsahuje Angularské značky[7]. Dále komponenty potřebují metadata, která říkají jak má Angular zpracovat danou třídu. Ke třídě *HeroListComponent* obsahující aplikační logiku komponenty se musí přidat dekorátor *@Component*. Komponenta může vypadat takto[11]:

```
@Component({
        selector: 'hero-list',
        templateUrl: './hero-list.component.html',
        providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit { ... }
```

Tímto dekorátorem se Angularu říká, že třída pod ním je komponenta. Angular vezme uvedenou konfiguraci komponenty a implementaci třídy a vytvoří požadovanou komponentu s její HTML šablonou. Uvedené konfigurace komponenty znamenají:

- selector - je-li v markupu nalezen tato značka, tak Angular vytvoří instanci komponenty a vloží ji dovnitř této značky. V uvedeném příkladu je komponenta vložena v případě výskytu značky <*hero-list*>, přičemž je vložena právě mezi otevírací a zavírací značku.
- templateUrl - říká, kde se nachází HTML šablona této komponenty.
- providers - zde se specifikují služby, které daná komponenta vyžaduje.

[7] nazýváme markup

Vázání dat

Logika aplikace s view vrstvou je propojena přes tzv. data binding (vázání dat) mechanismus. Do markupu HTML šablony jsou přidány tzv. spojovací značky, které Angularu říkají, jak má spojit obě vrstvy aplikace. Propojení funguje na 4 základních druzích komunikace, které se nejlépe znázorní na tomto příkladu[11]:

```
1 <li>{{hero.name}}</li>
2 <hero-detail [hero]="selectedHero"></hero-detail>
3 <li (click)="selectHero(hero)"></li>
```

Na prvním řádku je znázorněno zobrazení vlastnosti komponenty do šablony. Tomuto způsobu se říká interpolace a píše se do dvojitých složených závorek[11]. Jedná se pouze o svázání směrem od komponenty do DOM. Výsledkem je vypsání jména hrdiny.

Jevu na druhém řádku se říká svázání vlastností (tzv. property binding) a provádí se směrem z komponenty do DOM . Hodnota vybraného hrdiny z rodičovské komponenty se předá do vlastnosti *hero* podřízené komponentě, jejíž selektor je *hero-detail*.

Svázání událostí (tzv. event binding) je vidět na řádku třetím. Na rozdíl od předchozích příkladů se zde jedná o svázání opačným směrem, tedy směrem DOM - komponenta. Po kliknutí je zavolána metoda komponenty s názvem *selectHero*.

Posledním způsobem svázání je obousměrné kombinující svázání vlastností a událostí. Používá se k tomu *ngModel* direktiva. Pro jednoduché vysvětlení principu tohoto svázání je potřeba k předchozímu příkladu předpokládat, že podřízená komponenta se selektorem *hero-detail* obsahuje tento markup:

```
<input [(ngModel)]="hero.name">
```

Model zapříčiní to, že hodnota vlastnosti *hero.name* se dostane do hodnoty vstupního pole podřízené komponenty (jedná se svázání vlastností). Změní-li uživatel hodnotu vstupního pole, tak tato změna proteče do rodičovské komponenty, kde je hodnota pomocí svázání událostí zaktualizována.

Direktiva

Je evidentní, že angularské šablony jsou dynamické, protože jsou vykreslovány a transformovány do DOM podle tzv. direktiv. Direktiva je třída s *@Directive* dekorá-

torem. Často se vyskytují v HTML značce jako atributy[11]. Jsou rozlišovány dva typy direktiv:

1. Strukturální direktivy vkládají, odstraňují a nahrazují elementy v DOM. Například se jedná o **ngFor* či **ngIf*, kde tyto komponenty znamenají cyklus *for* či podmíku *if*.

2. Atributové direktivy upravují vzhled nebo chování existujícího DOM elementu. V šablonách vypadají jako běžné HTML atributy. Jedná se například o *ngModel*, *ngStyle* či *ngClass*.

Speciálním případem direktivy je *@Component*, což je direktiva s šablonou[11].

Poslední částí architektury jsou služby. Pod službou si lze představit libovolnou hodnotu, funkci či vlastnost, kterou aplikace využívá. Typicky je to třída se specifickou funkcionalitou. Například se může jednat o logovací, datovou či libovolnou počítací službu. Služby jsou opět reprezentovány třídami. Tyto třídy lze natáhnout do komponent. Jsou tam vkládány pomocí principů IoC, který je vysvětlen kapitole 2.5.1. Angular využívá vkládání závislostí přes konstruktor[11].

2.4.2 Shrnutí

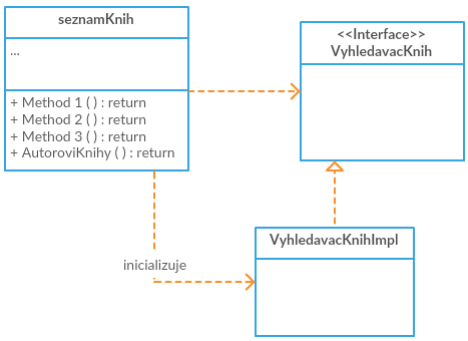
Jedná se o komponentový a javascriptový framework využívající MVC návrhový vzor. Služby jsou do jednotlivých komponent vkládány pomocí principů IoC, což programátorovi pomáhá mít závislosti pod kontrolou. Taková aplikace se totiž lépe testuje, rozšiřuje a udržuje. Jeho největší přednosti je, že Angular pracuje přímo s DOM. To je oproti modifikaci HTML rychlejší[11].

2.5 Spring

Spring je open-source framework, který se používá pro vývoj JavaEE aplikací. Byl vytvořen Rodem Johnsonem v roce 2002 a důvodem jeho vzniku bylo ulehčení vývoje enterprise aplikací[6].

2.5.1 Inversion of Control

Java aplikace se skládá ze spolupracujících objektů, které dohromady tvoří funkcionalitu aplikace. Tyto objekty mají na sobě různé závislosti. Díky těmto závislostem



Obrázek 8: Závislost objektů

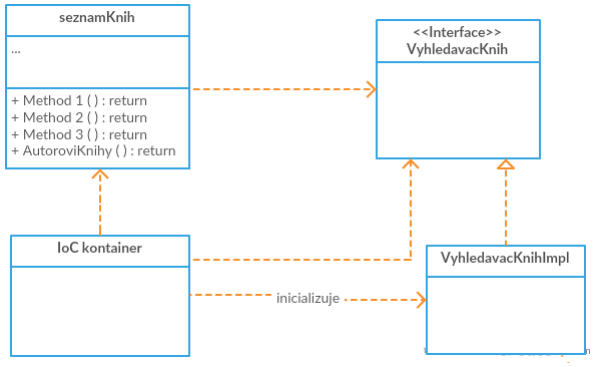
jsou třídy velmi těsně svázány. Změna jedné třídy pak potřebuje zásah do kódu. Inversion of Control (IoC) je návrhový vzor, který se snaží tyto vazby uvolnit. V podstatě funguje tak, že třída nevytváří instance daných tříd sama, ale jsou ji dodány z „vnějšku“. Rozlišujeme několik typů dodání závislosti:

- vkládání přes konstruktor - třídy, do nichž jsou vkládány další instance, musí mít vytvořen konstruktor s potřebnými parametry.
- vkládání přes setry - instance jsou do třídy vkládány přes setr[8].
- vkládání rozhraním - instance, která je do třídy přidána, implementuje rozhraní. Rozhraní objekt očekává v době sestavení programu[6, 7].

IoC lze demonstrovat na jednoduchém příkladu. Existuje třída *SeznamKnih* obsahující několik metod. Implementuje i metodu vracející seznam knih podle autora. V této metodě používá instanci třídy *VyhledavacKnih*, která dané knihy vyhledá. Těsná závislost je patrně v třídě *SeznamKnih*. Pro přehlednost je znázorněna na obr. 8. Těsnou vazbu lze odstranit některou z technik IoC. Například technikou přes setr, kterým se nastavuje atribut *vyhledevacKnih* (přistupuje se přes rozhraní). Tímto se třída *SeznamKnih* zbaví těsné závislosti, protože už nepotřebuje znát implementaci rozhraní. Je ovšem nutné implementovat nějaký IoC kontejner, který třídě *SeznamKnih* nastaví potřebné rozhraní. Grafické znázornění popsané situace za použití IoC je na obr. 9

Ve Springu je beana označení pro objekt, který je řízen z Spring IoC kontejneru. Defaultně je beana typu jediáček. Ve Springu IoC kontejner reprezentuje tzv. aplikační kontext, který je odpovědný za spouštění, konfiguraci a sestavování beanů. Tyto informace získává z konfiguračních metadat. Konfigurace metadat může být zastoupena XML souborem, anotacemi nebo Java kódem.

[8]metoda nastavující daný atribut třídy



Obrázek 9: IoC

Při spouštění aplikace jsou třídy spojeny s konfigurací metadat. Jakmile je aplikační kontext vytvořen a inicializován, tak se aplikace stává plně nakonfigurovanou a spustitelnou [6, 7].

Na výše uvedeném příkladu lze předvést jednotlivé konfigurace metadat.

Konfigurace Anotacemi

Anotace je v Javě zápis, jak přiřadit nějakému elementu (třída, atribut, metoda) jakýsi příznak, metadata či informaci mimo běžný kód. Anotace nemají žádný přímý vliv na fungování kódu[8]. Jak sestavovat beany lze ukázat na příkladu:

```
@Service
public class VyhledavacKnihImpl implements VyhledavacKnih{
      private String name;
      @Autowired
      public VyhledavacKnihImpl(@Value("Michal")String name){
            this.name = name;
      }
      ...
}
@Service
public class SeznamKnih {
      private VyhledavacKnih vyhledavacKnih;
      public SeznamKnih(){}
      ...
      @Autowired
```

```
    public void setVyhledavacKnih(VyhledavacKnih
        vyhledavacKnih){
            this.vyhledavacKnihKnih=vyhledavacKnih;
    }
}
```

Důležitá je anotace *Service* značící, že třída je služba. Anotace *Value* znamená, že proměnná bude naplněna hodnotou Michal. Anotace *Autowired* vloží instanci třídy (závislost) do atributu *vyhledavacKnih*. Lze ho použít na atribut, setr nebo konstruktor. Instanci třídy *SeznamKnih* pak lze získat použitím stejné anotace.

Defaultně je hledání anotací vypnuté. Zapnutí se opět provede příslušnou anotací nebo značkou v XML.

Konfigurace XML

XML soubor obsahuje značku beans, do které se vkládají jednotlivé beany. Jak sestavovat beany lze vysvětlit na následujícím souboru. Třídy vypadají podobně jako v konfiguraci s anotacemi. Třídy neobsahují žádné anotace.

```
<beans ...>
    <bean id="seznamKnih" class="cz.zcu.kiv.example.
        SeznamKnih">
            <property name="vyhledevacKnih" ref="vyhledavac
                "/>
      </bean>
    <bean id="vyhledavac" class="cz.zcu.kiv.example.
        VyhledevacKnihImpl">
        <constructor-arg type ="java.lang.String" value="
            Michal"/>
      </bean>
</beans>
```

Jsou zde dvě beany, které jsou identifikovány unikátním id. Beana *vyhledavac* vytvoří instanci třídy *VyhledevacKnihImpl*, který bude hledat autory se jménem Michal. Předpokládejme, že tato třída obsahuje konstruktor obsahující jméno hledaného autora. Tento parametr konstruktoru nastavíme právě značkou *constructor-arg*, ve kterém specifikujeme typ a jeho hodnotu. Druhá beana vytvoří instanci třídy

SeznamKnih. Za povšimnutí stojí, že zde není specifikován žádný argument konstruktoru. Třída musí implementovat defaultní konstruktor. Nicméně se zde objevuje značka *property*, která slouží k nastavení atributu třídy přes setr. Název atributu třídy odpovídá hodnotě *name*. Atribut *ref* nás odkazuje na příslušnou beanu, jehož instance je do tohoto atributu vložena.

Z příkladu je vidět, že Spring podporuje jak vkládání přes setr, tak i přes konstruktor.

2.5.2 Shrnutí

Spring je framework, který je organizován asi do 20 modulů. Tyto moduly jsou seskupeny do několika skupin (jádro, přístup k datům, web, testování, aspektově orientované programování atd). Z výčtu lze usoudit, že pokrývá širokou škálu částí Java EE aplikací. Jeho princip je založen na IoC, který nám umožňuje mít závislosti celé aplikace pod kontrolou. Výhodou je i to, že je framework velice dobře zdokumentován.

2.6 Emscripten

Emscripten je open-source projekt, kterým lze zkompilovat kód (psaný v jazyce C/C++) do JavaScriptu (JS) a tím použít nativní kód na webu[10].

2.6.1 Překlad

Emscripten umožňuje přeložit téměř jakýkoliv přenosný céčkový kód. Může se jednat o grafické, zvukové či souborové aplikace. Výstupním formátem je defaultně asm.js[10], což je vysoce optimalizovaná množina javascriptů poskytující abstrakci podobné C/C++ virtuálnímu stroji (celočíselné a desetinné aritmetické operace, ukazatele, velké binární haldy atd.)[10, 9]

Hlavním nástrojem pro překlad je tzv. Emscripten Compiler Frontend (EMCC). Jedná se o alternativu ke standardnímu C/C++ kompilátoru (např. gcc). Emcc využívá tzv. Clang, který zkonvertuje C/C++ kód do LLVM[9] IR. EMCC dále použije tzv. Fastcomp, který tvoří jádro výchozího Emscripten kompilátoru. Jeho úkolem je zkonvertovat LLVM (vytvořené pomocí Clang) do JS [10].

[9] Low Level Virtual Machine - kompilátorová infrastruktura optimalizující programy [10]

2.6.2 Omezení

Je zřejmé, že výsledný skript musí běžet na jakémkoliv webovým prohlížeči. JS a webový prohlížeč přece jen překládaný kód omezují. Program nesmí obsahovat[10]:

- aplikace s více vlákny sdílející mezi sebou stav. To je dáno tím, že JS umožňuje pouze vlákna (webworkeři) nesdílející stejný stav.
- kód spoléhající na architekturu velkého endianu. Emscripten požaduje, aby architektura hosta byla malého indianu[10]. JS sice dodržuje řazení bajtů podle hostitele, ale LLVM potřebuje vědět na jakou endianitu cílí.
- kód používající nízkoúrovňové funkce.
- kód skenující registry nebo zásobník.
- kód obsahující instrukci assembleru.

Většinu těchto omezení přenositelná aplikace splňuje, nicméně je potřeba brát na omezení ohled. Omezení programátora obecně není dobrá praxe (vede k chybám). Zmíněná omezení vznikají především principem JS a funkcionalit webových aplikacích. Několik těchto principů se snaží autoři Emscriptenu obejít. Jako je například problém se souborovým systémem. Kód běžící v prostředí prohlížeče je izolován a nemá přímý přístup na lokální souborový systém[11]. Emscripten poskytuje knihovny implementující virtuální souborový systém, díky kterému lze pracovat se soubory. Soubory se však nevyskytují fyzicky na disku.

2.6.3 Alternativa

Alternativou k Emscriptenu je Cheerp. Cheerp umožňuje překlad C/C++ do JavaScriptu. Výsledný kód je optimalizovaný jak do velikosti, tak do výkonu. Výsledný kód je multiplatformní, takže ho lze použít na libovolném prohlížeči. Má omezení stejná jako emsripten[15]. V práci se dále používá Emscripten a to z toho důvodu, že za ním stojí velká firma (Mozilla).

[10] 99% strojů připojených k internetu [10]
[11] Z důvodu bezpečnosti

2.7 Testování

Testování se snaží najít chyby programu, dříve než je software nasazen. Výsledkem otestování aplikace není tvrzení, že program neobsahuje žádnou chybu. Ale výsledkem testování je, že nebyla nalezena chyba aplikace. Pokud chyba projde až do produkční verze, pak aplikace utrpí na důvěře a uživatel ji v konečném důsledku nebude muset chtít používat. Testování by se mělo provádět od začátku projektu, protože dřívější odhalení chyby snižuje náklady na její opravu[13].

2.7.1 Testování webových aplikací

U webových aplikací je kladen větší důraz na vzhled a použitelnost. Pokud webová aplikace bude pomalá a nepřehledná, uživatel už ji znovu nenavštíví. Dále budou popsány hlavní techniky testování webových aplikací.

Funkční testování

Funkční testování zahrnuje otestování všech odkazů v aplikaci, databázové spojení, otestování všech formulářů, otestování cookies atd. Obecně se testuje veškerá funkcionalita aplikace.

Pod testováním odkazů si lze představit testování:

- odchozích odkazů z celé aplikace,
- všech interních odkazů,
- odkazů k posílání emailů,
- zda existuje nějaká stránka, ze které se nelze přesměrovat jinam, tzv. osiřelá stránka[12].

Testování formulářů spočívá v ověření existujících validátorů. Vyplňují se tedy pole se správnými a špatnými údaji a kontroluje se, zdali server požadavek přijme. Dále je důležité validovat HTML a CSS.

Velice důležitým bodem funkčního testování je testování databáze. Musí se kontrolovat konzistence dat. Kontrolujeme, zda do databáze lze zapsat špatná či neúplná data. Testuje se i integrita dat, kdy se kontroluje, zda byla uložena taková data, která se očekávala.

Testování použitelnosti

Dobře použitelná webová aplikace je taková aplikace, ve které je uživatel schopný provést nějakou akci, aniž by musel dlouho čekat, moc přemýšlet či hledat. Pokud uživatel nenarazí na použitelnou aplikaci, tak pravděpodobně odejde a bude hledat jinou.

Důležitým aspektem použitelnosti je navigace. Navigace znamená, jak uživatel prochází, rozumí a pracuje s webovými stránkami. Testování navigace zahrnuje testování, zda aplikace je jednoduchá na použití, poskytované instrukce jsou jednoznačné, zda hlavní menu je na každé stránce. Dále je důležité zajistit, aby stránky byly konzistentní. Pod tím se lze představit například to, že je-li hlavní menu nahoře, tak nahoře bude vždycky (nebude na jiné stránce např. vpravo).

Dalším aspektem je testování obsahu, který musí být snadno srozumitelný, neobsahující pravopisné chyby, má dobře nastavenou škálu barev atd. Obsah a a text odkazů by měl mít smysl, obrázky se zobrazují správně atd.

Testování rozhraní

Dalším aspektem je otestování všech rozhraní. Obvykle každá aplikace obsahuje serverové a databázové rozhraní. Toto testování zahrnuje kontrolu, zda veškeré chybové zprávy serveru generují chybové zprávy uživateli (např. email neexistuje). Kontroluje se, co se stane v případě výpadku jedné z napojených komponent.

Testování kompatibility

Webová aplikace je specifická tím, že může běžet na desktopu, telefonu či tabletu. Aplikace může běžet na mnoha webových prohlížečích např. Firefox, Opera, Chrome, Safari, Edge či Internet Explorer atd. Na všech těchto zařízeních a aplikacích je nutné otestovat responzivnost a správnou funkčnost aplikace.

Testování výkonu

Webová aplikace by měla být schopná pracovat i při větším zatížením. Při testování výkonu by se mělo testovat, zdali aplikace je schopna zpracovávat více požadavků najednou, zda je systém udržitelný i ve špičce, zda se k jedné stránce může dostat více uživatelů atd.

Dalším druhem testování výkonu je webové testování napětí, kdy se aplikace cíleně dostává mimo stanovené limity (přetížení serveru či databáze, špatné vstupy, mnoho požadavků atd.). Poté se sleduje, jak se aplikace chová.

Testování bezpečnosti

Testuje se, zda se uživatel může dostat k cizímu obsahu. Zda se lze dostat do autorizované sekce bez autorizace. Zda lze do vstupních polí vložit nevalidní kód. Kontroluje se ochrana proti botům (např. CAPTCHA a její náhodnost). Důležité je, aby se všechny transakce, chyby a pokusy o narušení logovaly do nějakého souboru na serveru.

2.7.2 Shrnutí

Testování webové aplikace je důležitou součástí jejího vývoje. Testování umožní nalézt chybu ještě před nasazením aplikace. Aplikaci je možné testovat ručně a automaticky. Při použití ručního testování je potřeba psát scénáře testů (alespoň ručně), aby vývojář věděl, jak k chybě došlo. Při automatizaci testů není toto třeba řešit, nicméně implementace testů bývá často pracnější. Při vývoji aplikace v týmu je automatické testování nezbytné, jelikož se provádí více úprav naráz, šetří se čas vývojářů/testerů.

Existuje mnoho knihoven k testování. Na klientskou část se hodí například Selenium či logování. Na serverovou část se hodí například Unit testy, které podporuje mimo jiné i Spring. Server webové aplikace musí každopádně podporovat logování, aby bylo možné kontrolovat jeho běh, popř. dohledávat jeho chyby.

3 Portál diabetes.zcu.cz

Portál diabetes.zcu.cz je webová aplikace, jejíž úkolem je počítat a vizualizovat koncentraci glukózy v krvi. Díky těmto statistikám se uživatel dozví, zda má zvýšenou koncentraci glukózy v krvi a z něho plynoucí diabetes neboli cukrovku. Uživatelé této aplikace mohou být pacienti, doktoři, vědci či naprostí laici. Tato aplikace umožňuje uživateli nahrát data z CGMS zařízení. CGMS zařízení je senzor pro měření koncentrace glukózy v krvi a podkoží. Koncentrace v podkoží je měřena kontinuálně, zatímco koncentrace v krvi sporadicky (např. tři měření za den). Portál pak z těchto dat dokáže spočítat křivku glykemie v krvi. Registrovanému uživateli portál umožňuje procházet historii měření, stažení grafu či jeho tisk. Anonymní uživatel může své údaje pouze zobrazit. Po opuštění stránky už se ke svým údajům nedostane.

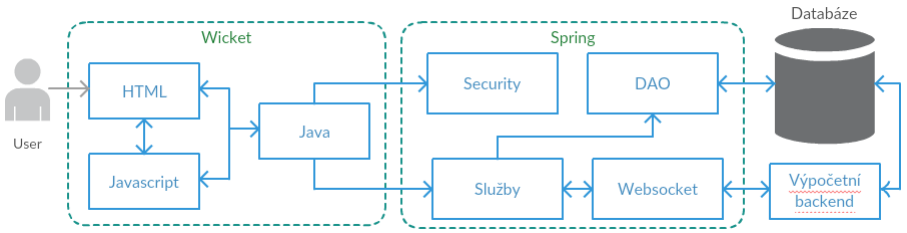
3.1 Architektura aplikace

Portál je nyní tvořen 4 základními moduly, obr 10: výpočetní backend, databaze, wicket modul a spring modul. Uživatel pracuje s jednotlivými stránkami, které jsou reprezentovány HTML. V těchto stránkách může provádět interakce. Tyto interakce se mohou provádět na straně klienta (JS) nebo se přenášet až na stranu serveru (Java). Tato vrstva aplikace se nazývá prezentační a dodržuje principy MVC. Prezentační vrstva je psána pomocí frameworku Apache Wicket, viz kapitola 2.3.

Vrstva pracující s databází se nazývá datová. Databáze se používá PostgreSQL a přistupuje se k ní přes DAO[12]. DAO využívá frameworku Spring podporující Hibernate. Hibernate je framework poskytující objektově relační mapování (tzv. ORM). Což znamená, že javovské objekty jsou mapovány na entity v relační DB. Tímto mapováním je Hibernate nakonfigurován tak, že ví, jakým způsobem se mají transformovat objekty do databáze a naopak. V DAO jsou implementovány veškeré HQL dotazy (Hibernate dotazovací jazyk)[13]. Výhoda použití Hibernate spočívá v tom, že v případě změny databázového systému není potřeba měnit HQL dotazy v aplikaci.

Propojení datové a prezentační vrstvy zajišťují tzv. služby. Tyto služby využívají principů IoC ve frameworku Spring. Tyto služby implementují práci s DAO vrstvou, použití anonymizeru a parseru. Anonymizer nahrané soubory anonymizuje a

[12]Data Access Object neboli objekt pro přístup k datům
[13]Jsou odvozeny od SQL



Obrázek 10: Architektura portálu

uloží na server. Parser nahraný soubor zanalyzuje, získá z něj potřebná data. Služba využívající parser pak pomocí DAO uloží získaná data do DB. Poslední službou je implementace klienta, která zajišťuje komunikaci s výpočetním backendem, viz bod 3.2

3.2 Výpočetní backend

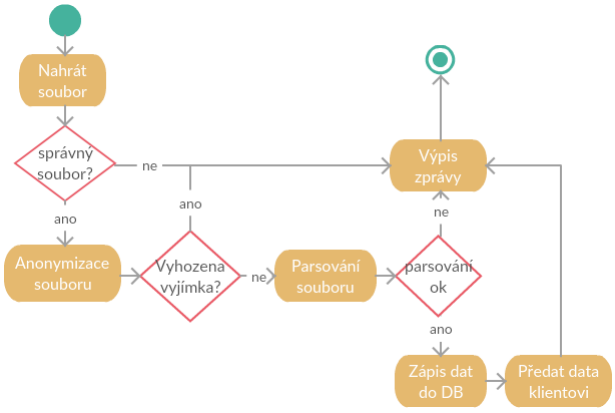
Z obrázku 10 je vidět, že aplikace komunikuje přes *websocketClienta* s výpočetním backendem. Výpočetní backend pracuje asynchronně a zcela nezávisle na webové aplikaci. Výpočetní backend je v projektu komponenta typu černá skříňka, tzv. black-box komponenta. Funkcionalita výpočetního backendu je zapouzdřena. Je známo, že na vstupu výpočetní backend obsahuje buffer, do kterého vkládá požadavky. Odpověď výpočetního backendu na požadavek je typu:

- požadavek přijat
- buffer je plný
- špatný požadavek

Výpočetní backend pak zpracovává požadavky asynchronně. Po zpracování požadavku ukládá spočítané údaje do databáze. Po zapsání do databáze posílá klientovi zpětnou vazbu o výsledku výpočtu. Klient podle této zpětné vazby nastavuje vlajky segmentu. Vlajky klasifikují segment do jedné ze skupin: výsledek k dispozici, výsledek nedostupný a čekání na výsledek.

Klient posílá výpočetnímu backendu požadavky přes websocket. Obsahem zprávy je řetězec typu JSON[14] s identifikátorem segmentu a identifikátorem metody, se kterou má výpočetní backend počítat. V aktuálním řešení je identifikátor metody pevně zadán, takže si nelze metodu počítání vybrat. Klient obsahuje tzv. handler (obslužná

[14]JavaScriptový objektový zápis



Obrázek 11: Nahrání dat

rutina), který zpracovává veškeré přijaté odpovědi od výpočetního backendu. Klient je schopný na chybové odpovědi reagovat. Je-li špatný požadavek, zapíše do databáze informaci o chybě, v případě plného bufferu posílá zprávu okamžitě znovu.

3.3 Nahrání dat

Nahrání dat je prováděno pomocí formuláře, do kterého se vkládá CSV soubor. Tento soubor obsahuje koncentraci glukózy v podkoží a i v krvi. V ojedinělých případech může mít uživatel koncentraci glukózy v krvi a podkoží oddělenu [15]. Z tohoto důvodu formulář obsahuje dvě pole pro nahrání souborů. Registrovaný uživatel ještě může před nahráním vložit poznámku.

Jak se provádí upload, je znázorněno na obr. 11. Soubor je uživatelem nahrán a je poslán na server. Tam je ověřeno, jestli se jedná o požadovaný CSV soubor. Soubor se musí anonymizovat, což je proces odstranění osobních údajů ze souboru. Soubor se následně uloží na server. V dalším kroku je soubor rozparsován. Výsledkem parsování je pole segmentů rozdělené podle časové osy. Segment už obsahuje naměřené hodnoty v krvi či podkoží a další potřebné údaje. Do segmentu se přidají další potřebné údaje, jako je například zadaná poznámka či účet, pod kterým byl segment nahrán. V dalším kroku se provede zápis dat do databáze. Jsou zapsány všechny segmenty, které byly výsledkem parsování. Posledním krokem je předání segmentů klientovi, který obstará poslání požadavku na výpočetní backend, viz bod 3.2.

Nutno zmínit, že rozdíl mezi nahráním jednoho a dvou souborů je minimální. Anonymizují se oba soubory a parser pak pracuje se dvěma soubory. Jinak veškerý

[15] Zvláštní soubor pro krev a podkoží

princip zůstává stejný.

Při anonymním uploadu nastává menší rozdíl. Aplikace čeká, dokud výpočetní backend nespočítá výsledky, protože statistiky je možné zobrazit až po obdržení výsledků. Anonymní uživatel ale nemá účet, kde by se mohl ke svým statistikám dostat. To je samozřejmě nepohodlné, protože uživatel může čekat dlouho. Nicméně anonymní uživatel funguje jen k vyzkoušení aplikace.

3.4 Prezentace výsledků

Prezentace výsledků je prováděna pomocí panelu, který vždy zobrazuje datum měření, počet údajů z kůže, podkoží a zadanou poznámku. Panel pak může mít tři stavy:

- ok - spočítání proběhlo v pořádku. Tento druh panel zpřístupňuje tlačítko pro vizualizaci grafu. Ten se zobrazí ve vyskakovacím okně a je vykreslen pomocí javascriptové knihovny *Highcharts*.
- chyba - panel informuje, že výsledek nebylo možné spočítat. Došlo k nějaké chybě.
- čekání - panel nemá ještě k dispozici výsledky od výpočetního backendu.

Na stránce se může vyskytovat 10 těchto panelů. Pokud se na stránku nevejdou, je vykreslena navigace, pomocí které lze přepínat mezi segmenty.

Na desktopu se v pravé části zobrazuje panel s grafem (nemusí se nutně vykreslovat do okna). Tento panel má implementovaný tzv. affix. Ten způsobí to, že graf se při rolování stránky posouvá. Graf momentálně vykresluje pouze křivku s naměřenými hodnotami v podkoží a křivku s vypočítanou koncentrací glukózy v krvi.

Momentálně je implementováno vykreslování grafu pro tři metody (difusion1, difusion2 a difusion3). Nicméně je možné mluvit pouze o dvou metodách, protože difusion2 je pouhým rozšířením metody difusion1.

4 Analýza stávajícího řešení

V této části páce se identifikují ty části serveru, které lze přesunout na stranu klienta pomocí Emscripten. Při přesunu na klienta je nutné brát ohled na otevřenost kódu, limity mobilních zařízení, bezpečnosti atp. Z pohledu architektury aplikace, popsané v bodě 3, lze přesunout tyto komponenty: anonymizér, parser, vykreslování grafu a výpočetní backend.

4.1 Anonymizér

Nahrané soubory se ukládají na serveru, kde slouží ke sběru dat. Tyto soubory mohou obsahovat osobní údaje, proto je nutné soubor projít a tyto údaje smazat. Dále je vygenerován unikátní název souboru. Anonymizací musí soubor projít vždy, protože není dovoleno udržovat osobní údaje, a je nutné zajistit unikátnost souboru.

Přesunutí na klienta je možné. Principy JS umožňují procházet CSV soubor a mazat požadované údaje. Teoreticky by bylo možné vygenerovat i unikátní jméno souboru, které by se mohlo skládat z času v milisekundách a z určitého řetězce (např. login). Lze tedy říci, že přesunutí je realizovatelné.

Další části analýzy je otázka vlivu na server. Obecně při vytváření webových aplikací je potřeba vstupní data validovat. Není možné spoléhat na to, že přes klienta neprojdou špatná data. JS lze obejít a klientská validace nemusí být 100%. Upload souboru je speciálním případem vstupních dat. Z těchto důvodů se nelze spoléhat na to, že veškerá potřebná funkcionalita je na klientovi provedena. Důsledkem tohoto tvrzení je to, že anonymizér nelze ze serverové části odstranit.

Poslední částí analýzy je otázka výkonu. V případě anonymizéru je otázka jednoduchá a zní: „Není zbytečné zatěžovat klienta funkcionalitou, která musí být na serveru stejně provedena“? Odpověď je ještě prostší: „Ano je“.

Výsledkem analýzy přesunu anonymizéra na klienta je konstatování, že se jedná o zbytečné zatěžování klienta. Z tohoto důvodu se tento přesun implementovat nebude.

4.2 Parser

Stejně jako v případě anonymizéru je přesun parseru možný. Principy JS nám umožňují tento přesun. Na klientovi by bylo možné získat data ze souboru a poslat je na server. Server by neměl mít s přijetím dat a jejich zapsáním problém. Nicméně je

potřeba se zamyslet nad bezpečností a integritou dat. Integrita dat se s určitostí nemůže zaručit, jelikož může během přenosu dojít k chybě. Další problém se týká bezpečnosti. Na server mohou dojít špatná a nesmyslná data. Na straně serveru by bylo nutné implementovat nějaký validátor, který by data ověřil. V takovém případě by bylo možné takové řešení implementovat. Nicméně klient by musel posílat jak data, tak i soubor, což je dáno požadovaným sběrem dat (souborů).

Dále je nutné zohlednit výkonnostní hlediska. Klient by prošel soubor a zkonstruoval data. Nicméně tato data by byla potřeba na serveru znovu projít a nějakým sofistikovaným způsobem zvalidovat a ověřit integritu. To by se provádělo porovnáváním s obdrženým souborem. Je evidentní, že přesunem na klienta nesnížíme zatížení serveru. Posledním hlediskem je, že posílat data a soubor je neefektivní.

Z důvodu neefektivního přenosu dat po síti, nesnížení zatížení serveru a v konečném důsledku zbytečného zatěžování klienta je výsledkem analýzy konstatování, že přesun parseru na klienta se nevyplatí a implementovat se nebude.

4.3 Výpočetní backend

Výpočetní backend provádí paralelní výpočty pomocí TBB [16]. Jednotlivá vlákna sdílí mezi sebou stav, což naráží na možnosti Emscriptenu a principy JS, viz kapitola 2.6.2. Proto není možné výpočetní backend v aktuální podobě přesunout.

Pokud se bude uvažovat se sériovou verzí programu, tak se bude narážet na stejné problémy jako v předchozích případech. Server by musel kontrolovat správnost dat, což by tentokrát bylo s velkou pravděpodobností algoritmicky jednodušší. To z toho důvodu, že validace by běžela na cca deseti parametrech. Tyto parametry jsou typu *float*, takže jejich validace je jednoduchá. V dnešní době už je i pravděpodobné, že mobilní zařízení mají dostatečný výkon pro tento výpočet. Na druhou stranu není žádoucí, aby aplikace zbytečně zatěžovala mobilní zařízení náročnými výpočty, a tím spotřebovávala větší množství energie.

Tato analýza není příliš podrobná, z důvodu nemožnosti použití překladu paralelního kódu se sdíleným stavem. V případě sériového kódu nastávají obavy s velkým zatížením mobilních zařízení. Z těchto důvodu se přesun výpočetního backendu provádět nebude.

[16] Intelovká knihovna Threading Bulding Block

4.4 Vykreslení

Poslední analyzovanou částí je vykreslení. Vykreslení se v aktuální době provádí na klientovi pomocí JS a knihovny *HighChart*. Nicméně se předpokládá implementace nových grafů, které mají být interaktivní a mají pracovat s více metodami. Momentálně jsou data vkládána do polí s naměřenými a spočtenými hodnotami. Tyto pole jsou parametry JS funkce, která provádí vykreslení. O zbytek vykreslení se stará JS, a proto zde určitě půjde použít Emscripten.

Lze říci, že se jedná o výstupní funkcionalitu, jelikož server pošle data a už dál neví, jak se na klientovi výsledky prezentují. Nehrozí zde tedy žádná nebezpečí poškození serveru. Z logiky věci je jasné, že nemůže dojít ani ke snížení serverového výkonu. Data se stejně posílat musí. Otázka přetížení klienta také není na místě, protože zvládá-li klient vykreslení teď, není důvod pochybovat o budoucnu. Vykreslení je pořád stejné, takže složitost se nezmění.

Neexistuje jediná pochybnost o nevhodnosti řešení. Překlad by přinesl výhodu odstranění závislosti na externí knihovně HighChart. Z těchto důvodů se rozhodlo, že se vykreslování bude překládat pomocí Emscriptenu. O přesunu nelze z uvedených důvodů mluvit.

4.5 Shrnutí

Byla provedena analýza zahrnující možnosti přesunu části aplikace na klienta. U vstupních aplikací nastává problém s validací parametrů. Důsledkem toho není možné serveru „ulevit“, jen by byl zbytečně zatěžován klient. Na přesunutí jsou výhodnější výstupní operace, které není nutné validovat. Ze zkoumaných subjektů lze přesunout pouze vykreslování.

5 Přesun činnosti na klienta

V kapitole 4 byla provedena analýza zkoumající, ve kterých částí aplikace je vhodné použít Emscripten. V úvahu připadá pouze vykreslování, které bude kompletně předěláno. Dále se do něj budou implementovat nové funkcionality jako je Parkesova a Clarkova chybová mřížka. Dále se požaduje graf AGP [17], výpis statistik, generování hodnot do CSV a stažení vygenerovaného SVG souboru. Při implementaci vykreslování je třeba brát v potaz to, že dojde k rozšíření nahrání dat. Bude možné vybrat více postupů výpočtu.

5.1 Analýza řešení

Při návrhu řešení je třeba zakomponovat omezení plynoucí z principů Emscriptenu, viz kapitola 2.6.2. Je třeba brát v úvahu, že ne všechny externí knihovny tyto podmínky mohou dodržovat. Z tohoto důvodu je při návrhu řešení předpokládáno, že externí knihovny (jako Qt, SDL2...) nebudou použity. Předpokládá se, že generování SVG souboru, parsování JSON či další funkcionality budou v kompetenci programátora.

Vstupem programu jsou proměnné typu *string* obsahující data ve formátu JSON. Bude se jednat o proměnné:

- ist - JSON pole objektů obsahující naměřené hodnoty v podkoží. Takový objekt obsahuje dvojici atributů *date* a *value* reprezentující čas ve formátu *RRRR-MM-DDThh:mm:ss* a hodnotu měření.
- blood - obsahuje úplně stejné pole objektů jako předchozí proměnná, s tím rozdílem, že zde se jedná o naměřená data z krve.
- par2 - JSON objekt obsahující spočítané výsledky metodou difusion2.
- par3 - JSON objekt obsahující spočítané výsledky metodou difusion3.
- strings - JSON objekt obsahující lokalizační řetězce

5.1.1 JSON parser

Po obdržení parametrů je provedeno jejich převedení do odpovídajících objektů. Pro naměřené hodnoty bude vytvořen objekt *Value* obsahující stejné atributy jako JSON.

[17] Ambulatory Glucose Profile

Atribut čas bude uložen v struktuře *time_t*. Pro parametry jsou vytvořeny objekty (*diff2* a *diff3*) odpovídající objektům v serverové části aplikace.

Pro parsování naměřené hodnoty se implementuje metoda *parseValue*. Tato metoda vrací instanci objektu *Value* s odpovídajícími atributy. Dále budou implementovány metody pro parsování příslušných výsledků.

5.1.2 SVG generátor

SVG je škálovatelná vektorová grafika, která se definuje pomocí XML formátu. Lze proto jednoduše generovat. V aplikaci není třeba psát obecný SVG generátor. Generátorem půjdou generovat jen ty elementy, které budou potřeba k vykreslení všech grafů a statistik (text, přímka, cesta, elipsa atd.).

Nicméně je třeba se zamyslet, jak bude implementována interpolace či aproximace křivky zadanými body. SVG poskytuje speciální element *path*, který již aproximaci provádí. Implementuje Bézierovy křivky a to jak kvadratickou, tak kubickou verzi. Kvadratická verze potřebuje tři body, kde druhý bod se nazývá kontrolní. Kontrolní bod určuje směr křivky a tímto bodem křivka většinou neprochází. Nekontrolními body křivka prochází. Kubická verze je používanější a potřebuje 4 body, kde dva z nich (vnitřní body) jsou kontrolní. Vzhledem k tomu, že se jedná o zdravotnické znázornění, kde je přesnost vykreslení důležitá, bylo rozhodnuto, že i přes menší oblibu je lepší použít kvadratickou verzi. Křivka bude procházet více body.

5.1.3 Grafy

Výsledné křivky znázorňují koncentraci glukózy v krvi. Počítají se z naměřených hodnot z podkoží, z krve a z parametrů dané metody. Tyto parametry byly spočítané výpočetním backendem. Pro metody difusion2 a difusion3 jsou metody napsány v JS. Stačí je jen přepsat do C++ syntaxe.

Důležité je dodat, že je požadováno, aby se vykreslovaly všechny možné křivky. Tzn. křivka naměřených hodnot v podkoží, výpočty pomocí všech metod, pro které jsou k dispozici údaje. A samozřejmě body naměřených hodnot v krvi.

Bude implementován graf[18] vykreslující tyto křivky, kdy na ose x je čas a na ose y koncentrace. U tohoto typy grafu je potřeba, aby bylo možné přepnout měřítko časové osy na jeden den. Důsledkem této změny vznikne několik křivek, které budou zobrazovat vývoj glukózy v různých okamžicích dne.

[18] tzv. defaultní graf

Parkesova a Clarkova chybová mřížka zobrazuje chybu spočtených dat. Vezme naměřenou a vypočtenou hodnotu z krve a vloží ji na souřadnicový systém.

Dalším grafem, který se implementuje je graf AGP. Tento graf znázorňuje vývoj koncentrace glukózy v denním horizontu v rozsahu jednotlivých kvartilů. Vezme se každá přímka z defaultního grafu s denní mřížkou. Z těchto přímek graf sestaví pro každý časový okamžik pole všech naměřených hodnot. V každém takovém bodě pak spočítá hodnotu všech kvartilů, které znázorňují vývoj glukózy. Výsledný graf bude znázorňovat přímky, které jsou dány jednotlivými kvartily v daných bodech. Medián znázorňuje odhadovaný vývoj glukózy.

U všech grafů se předpokládá, že bude možné si volit, které křivky budou zobrazeny. Takovýto graf půjde ve zvolené podobě stáhnout. Dále se předpokládá možnost přepnutí jednotek mmol/l a mg/dL, kde platí: $mg/dl = 18 * mmol/l$. Posledním požadavkem je, aby šlo ve webové aplikaci rozumně manipulovat, to znamená umožnění zoomu a panu.

5.1.4 Přístup

Před samotnou implementací je nutné rozhodnout, jaký přístup se bude využívat. Je na výběr mezi klasikou C++ aplikací, která bude mít minimální rozdíly oproti emstripten verzi nebo aplikací implementující využívající veškeré možnosti emsriptenu. První varianta spočívá v tom, že bude implementováno veškeré vykreslení a přidaná funkcionalita (generování, zneviditelňování atd.) bude implementována přímo v JS. Druhá varianta by implementovala veškerou funkcionalitu pomocí emsriptenu pomocí knihovny HTML5, kterou Emscripten poskytuje. Výhodou první varianty je oddělení vykreslování od úprav, šetření výkonu (není třeba překreslovat graf, když stačí jen odstranit/zneviditelni elementy z DOM) . Největší výhodou je ale to, že kód C++ aplikace bude spustitelný. Druhá varianta udržuje překládaný kód celistvý, nicméně ladění je komplikovanější. To z toho důvodu, že se musí neustále překládat. Právě z toho důvodu se rozhodlo, že se s emstriptenem bude překládat pouze jednoduché vykreslování a zbytek se implementuje mimo překládaný kód.

5.2 Řešení

Vykreslování se implementuje nejprve jako klasická C++ aplikace, kdy je generovaný SVG soubor uložen na disk a odtud kontrolováno vykreslení. Později se aplikace

upravuje do podoby potřebné pro překlad. Samotná integrace této samostatní části aplikace do portálu je popsána v bodě 6.

Před začátkem implementace je třeba definovat, jak bude probíhat předávání dat do vstupu aplikace. Známými formáty pro přenos dat je XML a JSON. XML slouží pro přenos strukturovaných dat. Význam jednotlivých části dat je dán pomocí XML značek. JSON slouží také pro přenos dat. Lze říci, že JSON je reprezentován mapou, kde každému klíči je přiřazena nějaká hodnota. Tato hodnota může být objekt, pole, číslo atd. Je známo, že XML dokáže lépe vystihnout kontext dat, ale za cenu většího přenosu dat (zejména značek).

V aplikaci je potřeba identifikovat data tak, aby se dal specifikovat jejich význam (data z krve, podkoží, parametry metody diffusion2 atd.). K tomu plně dostačuje JSON, protože význam dat lze specifikovat metodou klíčem-hodnota.

5.2.1 Popis implementace

Je implementována metoda *parseInput*, která provádí veškeré parsování vstupních JSON řetězců. Z této metody jsou volány další podřízené metody, které rozparsují dané naměřené hodnoty a výpočetním backendem spočtené výsledky. Tato metoda vrací největší hodnotu měření, která se používá při vykreslování normalizované osy y. Rozparsovaná data jsou předána metodě *getVectorMap*, která provede výpočet nový hodnot. Vstupní naměřená data a výsledné vektory výpočtu se uloží do mapy podle předem daných klíčů. Zde se nastavuje také to, jaká data si uživatel přeje vidět. To je potřeba pro graf AGP, který se musí přepočítat vždy.

Dále je třeba rozparsovat zadané lokalizovatelné řetězce reprezentující popisky grafů. Tyto řetězce jsou v JSON přiřazeny jako klič-hodnota, a proto výsledkem parsování bude mapa. V ní se bude hledat text popisku grafu podle klíče.

Pro vykreslení jednotlivých grafů jsou použity metody:

- *generate_graph* - pro defaultní graf,
- *generate_day* - pro defaultní graf s denní mřížkou,
- *generate_parkes* - Rarkesův graf,
- *generate_clark* - Clarkův graf,
- *generate_agp* - AGP graf.

Parametry těchto metod jsou mapa s daty, maximální y-ová hodnota, mapa řetězců a typ jednotek. Metoda Parkesova grafu ještě obsahuje signalizaci, jestli se jedná o diabetes typu 1. Podle této proměnné se volí, jestli se vykreslí mřížka přesností.

Vykreslování jednotlivých grafů využívá tvary SVG generátoru pro vykreslení všech svých potřebných útvarů. Všechny tvary jsou normalizovány a to tak, aby výsledný graf byl roztažen přes celou svoji šířku.

5.2.2 Emstripten

Zcela fungující C++ aplikace potřebuje modifikaci, aby fungovala v JS tak, jak je požadováno. V první řadě je třeba vytvořit metody, které budou pomocí Emscriptenu zviditelněny pro ostatní skripty. To se provede pomocí klíčového slova *extern*. Ukázka takové metody pro defautní graf je následující:

```
extern "C" long graph_main(char* ist, char* blood, char*
   param2, char* param3, char* stringsJson, int mmol) {
    reset_global_var();
    ...
    double max_value = parseInput();
    stream << generate_graph(getVectorMap(istVisible),
       max_value, parseString(stringsJson),mmol);
    ...
    return resultSvg.size();
}
```

Hned z hlavičky funkce lze vidět, že vstupními parametry jsou zmíněné JSON řetězce. Posledním parametrem se určuje, v jakých jednotkách se má graf vygenerovat[19]. Před startem funkce je potřeba resetovat všechny globální proměnné, protože jejich hodnotu si JS pamatuje z předchozího běhu, což bylo zjištěno testováním, kdy program vykresloval neočekávané křivky.

Nejprve tato metoda byla implementována tak, že vracela již výsledný řetězec. V určité fázi vývoji se stávalo, že se vrátil řetězec s neočekávanými znaky. Bylo zjištěno, že to je dáno velikostí řetězce.

Emsctripten požaduje, aby proměnné s velkým množstvím dat byly alokovány pomocí *malloc* před jejím naplněním[10]. Proto tato funkce je modifikována do ta-

[19] mmol/l nebo mg/dl

kové podoby, kdy vrací velikost vygenerovaného řetězce a řetězec se uloží do globální proměnné. Důsledkem toho se implementuje další externí metoda *write_result*, která vrací už požadovaný SVG řetězec.

Na následujícím příkladu je znázorněno, jak je možné v případě úspěšného překladu metodu *graph_main* používat:

```
1 graph = Module.cwrap('graph_main', 'number', ['string','
    string','string','string','string','number']);
2 clark_result = Module.cwrap('write_result', 'string', []);
3 var result_size = graph(ist,blood,par2,par3,strings,mmol);
4 var graph_svg = Module._malloc(result_size);
5 graph_svg = graph_result();
6 ...
7 Module._free(graph_svg);
```

Na prvním a druhém řádku se načtou metody z překládané části. To za pomocí instrukce *cwrap*. Parametry jsou v pořadí: název funkce, typ výstupní hodnoty a pole typů vstupních parametrů. Na třetím řádku se zavolá funkce, která vygeneruje SVG řetězec a vrátí jeho velikost. Na čtvrtém řádku se alokuje požadovaný počet bajtů a na řádku pátém do této proměnné zapíšeme vygenerovaný SVG řetězec. Na úplný konec se nesmí zapomenout na uvolnění alokované paměti.

Popsaným způsobem je upraven veškerý C++ kód, a to pro všechny metody zmíněné v bodě 5.2.1. V C++ kódu se stejným principem aplikuje i generování CSV řetězce.

5.2.3 Vedlejší funkcionalita

Jak bylo zmíněno v analýze řešení, tak Emscriptem se překládá pouze ta nejnutnější část. Veškeré stahování, překreslování a přibližování je psáno v JS a to v souboru *chartMain.js*.

Zneviditelnění některých elementů se provádí pomocí jejich změnění atributu *visibility* v DOM. To je umožněno pomocí jednotných identifikátorů, které jsou generovány v překládané verzi JS. Výjimku tvoří AGP graf, který je nutné překreslovat vždy[20]. Proto AGP graf a jeho elementy mají rozdílné identifikátory oproti ostatním křivkám. Generování SVG souboru se provádí tak, že se vytvoří DOM z vygenero-

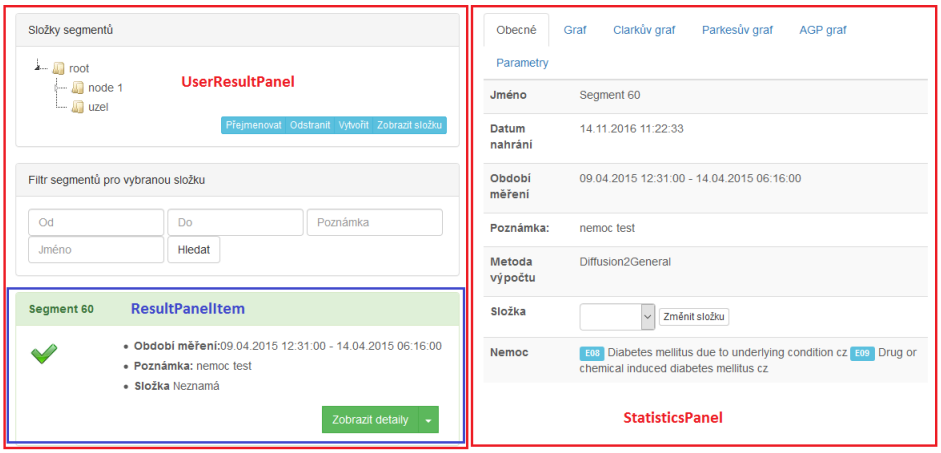
[20] kvartily se musí přepočítat

vaného SVG řetězce. Z něj se opět odstraní nechtěné elementy. Zbylé části se uloží do SVG souboru.

Generování CSV souboru musí probíhat v překládaném kódu. Je třeba provést všechny výpočty, které zjistí další hodnoty glukózy. Spočtená a naměřená data jsou spolu s parametry vygenerována do výsledného souboru.

Další vedlejší funkcionalitou je změna jednotek. Při požadavku na změnu jednotek musí být překreslen celý graf. To z důvodu nových popisků a výpočtu statistik. Absolutní chyba je určována v definovaných jednotkách. Na rozdíl od relativní chyby, která je vyjádřena v procentech.

Přibližování, oddalování a manipulace s grafem je prováděna pomocí knihovny *svg_pan_zoom*.



Obrázek 12: Nová verze PersonalPage

6 Nová funkcionalita

Před začátkem implementace bylo rozhodnuto, že se bude pokračovat ve vývoji portálu ve stejné architektuře jako nyní. To z toho důvodu, že se nevyplatí předělávat celou aplikaci.

Nová funkcionalita se týká několika částí. První část se zabývala pře-implementací *PeronalPage*, nová verze je na obrázku 12. K ní se dostane pouze uživatel, který je přihlášen. Může v ní nalézt všechny svoje nahrané segmenty a zobrazit si jeho podrobnosti, statistiky atp. Součástí této stránky je i filtrování, seskupování do skupin a změna parametrů.

Další změny se týkaly stránky *UploadPage*, ve které je možné nahrát CSV soubor. Nová podoba této stránky pro vědce je vidět na obr.13.

6.1 PersonalPage

PersonalPage je rozdělena do dvou na desktopu stejně širokých sloupců. Každý sloupec je reprezentován jedním panelem. Vlevo lze nalézt *ResultPanel*, zatímco vpravo je *DrawingPanel*. *ResultPanel* zobrazuje filtry, vyskakovací okna a jednotlivé segmenty. Každý segment je reprezentován *ResultPanelItem*, kde si stiskem na správné tlačítko může uživatel zobrazit odpovídající *DrawingPanel* z pravé části stránky. *DrawingPanel* vykreslí uživateli graf, který zobrazuje koncentraci glukózy v krvi. Uživatel si může zobrazit *DrawingPanel* i ve vyskakovacím okně, přičemž vyskakovací okno je na mobilu jedinou možnou variantou zobrazení grafu. Původní rozložení stránky odpovídá novému rozložení na obrázku 12. V nové verzi je přidán stromový filtr a

jednoduchý *DrawingPanel* je nahrazen *StatisticsPanelem*.

Druhou úpravou je modifikace *ResultPanelu* tak, aby bylo možné použít stromový filtr. Tato změna se nazývá úprava filtru.

6.1.1 Úprava statistik

Úkolem je modifikovat pravý panel tak, aby zobrazoval více informací o segmentu, vykresloval grafy z bodu 5 a zobrazoval spočtené parametry. To lze interpretovat tak, že do aktuálního *DrawingPanel* se přidají záložky, které budou zobrazovat informace, parametry a vizualizovat výsledky. Nutno podotknout, že tento panel se vykytuje ještě na stránce s nahráváním souboru.

Vzhledem ke komponentovému přístupu by se změna panelu projevila na všech místech aplikace. Aktualizovaný panel bude zobrazovat komplexní výsledky, které zobrazují různé statistiky. Původní název panelu by byl zavádějící. Proto se *DrawingPanel* nahradí *StatisticsPanel* ve všech částí aplikace. Panel se statistikami zapouzdřuje veškeré znázornění výsledků, statistik a měření. Panel pro kreslení je použit pouze pro vykreslování grafů, které byly implementovány v bodě 5.

Na rozdíl od původního řešení, se v nové verzi vykreslují 4 grafy (původní graf, Clarkův, Parkesův a AGP). Pro každý tento graf je nutné implementovat načítání dat, řetězců, vytváření JSON atd. Do vykreslování se vždy posílá stejný JSON obsahující stejná data. Liší se pouze volaná metoda a lokalizační řetězce. Z důvodu omezení duplicity kódu je kreslící panel modifikován na abstraktní třídu, která získá potřebná data z databáze, a lokalizační řetězce z abstraktní metody *getStringsByLocale.* Dále sestaví JSON řetězec a zavolá požadovanou JS metodu, jejíž parametrem je právě sestavený JSON. Název volané metody je specifikován v konstruktoru. Každý panel obsahující již konkrétní graf od této abstraktní třídy dědí.

Panel se statistikami bude dále obsahovat podrobnosti o segmentu jako je: jméno, datum nahrání, období měření, poznámka. Poslední funkcionalitou je vykreslení všech parametrů, které spočítal výpočetní backend. Uživatel typu vědec má umožněno tyto parametry měnit.

Vytvořený panel je vidět v pravé části obrázku 12. Lze si všimnout, že veškeré statistiky jsou zaobaleny do záložek, které využívají principů AJAX.

6.1.2 Úprava filtrování

Úkolem je modifikace *ResultPanelu* tak, aby bylo možné použít stromový filtr. Tato úprava je ve skutečnosti složitější, než doopravdy vypadá, a to z následujícího důvodu. *ResultPanel* se vyskytuje na třech místech aplikace se vždy mírně odlišnou funkcionalitou. Například umožňuje zobrazení panelu se statistikami pouze v okně. Někde není umožněna filtrace, někde se zobrazuje jen určitý počet výsledků a někde jsou zobrazena tlačítka navíc. Rozdílná funkcionalita se týká také jednotlivých segmentových panelů se jménem *ResultPanelItem*, kde přihlášený uživatel má více možností. Funkcionalita obou panelů je určována podle nastavení vlajek v konstruktoru. Vzhledem k tomu, že tento přístup je dosti zmatečný a nepřehledný, dojde nejprve k vytvoření abstraktních tříd a následné dědičnosti.

Vytvoří se třída *AbstractResultPanelItem,* která definuje základní vzhled jednotlivé položky výsledku. Podle údajů v databázi určuje, jestli výsledek je správný, špatný nebo dosud nespočítaný. Dále se modifikuje třída *ResultPanelItem*, která je potomkem předchozí abstraktní třídy. Tato třída implementuje požadované rozšíření jako je přepočet parametrů, nahlášení chyby, či rozlišuje zobrazení výsledků v okně nebo panelu. Toto rozlišení je důležité kvůli responzivitě webu, protože výsledky v panelu lze zobrazit jen na desktopu, viz obrázek 12.

Dále se vytvoří *AbstractResultPanel*, který implementuje společnou funkcionalitu a vkládá společné prvky. Například vloží vyskakovací okno obsahující zobrazení panelu se statistikami a jeho zviditelnění. V aplikaci pak lze rozeznávat tři druhy panelů z výsledky:

1. anonymní - Slouží pro anonymní prezentaci výsledků. Zobrazí jednotlivé položky výsledku. Anonymní uživatel má možnost zobrazit statistiky pouze v okně.

2. základní - Slouží k prezentaci výsledků okamžitě po nahrání. Zobrazují se pouze 3 položky a lze je zobrazit pouze v okně.

3. uživatelský - Slouží k podrobné prezentaci výsledků, která je použita v *PersonalPage*. Zde jsou vykresleny veškeré nahrané segmenty a jejich výsledky. Uživatel může statistiky zobrazit jak v panelu, tak ve vyskakovacím okně. Na středních a menších obrazovkách je možnost zobrazení pouze v okně. Vzhledem k velkému počtu segmentů je zde potřeba filtrovat.

V tomto okamžiku už je možné upravit filtrování v uživatelském panelu. Ve výchozí verzi se nachází pouze filtrování pomocí jména, času a poznámky. Úkolem je pro-

pojit tento filtr s filtrem novým, který bude reprezentován stromem. Požaduje se, aby uživatel mohl seskupovat segmenty do skupin. Ve stromu si bude moci vybrat skupinu, kterou chce zobrazit. Na tuto skupinu bude moci použit původní filtr.

Pro uživatele je pravděpodobně nejznámější seskupování pomocí složek. Se složkami se jistě setkal každý uživatel nějakého zařízení s operačním systémem. Také je vhodné použít i strom, který se svým vzhledem blíží k tomu z Windows. Dobrou volbou se zdá být knihovna *jsTree*, kterou lze jednoduše integrovat do aplikace. Interakce stromu a serveru lze zajistit pomocí technologie AJAX.

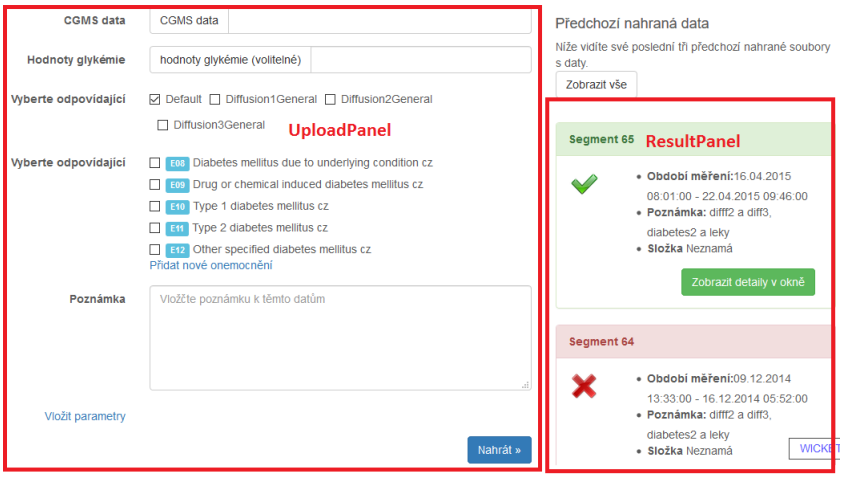
Aplikace v původní podobě se složkami nepracuje a ani s jejich implementací nepočítá. Z tohoto důvodu je nutné nejprve rozšířit relační model databáze. Vytvoří se tabulka *folder*, která mimo jiné udržuje identifikátor, jméno, jakému účtu naleží a nadřazenou složku. Dále je potřeba doplnit informaci, do jaké složky daný segment spadá. Do tabulky segment se tedy přidá údaj *folder*.

Wicketovské části aplikace se týkají změny v uživatelském *ResultPanelu.* Zde je potřeba vložit *TreePanelScript*, který zapouzdřuje veškerou funkcionalitu a i vzhled stromového filtru (panel s názvem složky segmentů na obr. 12). Tento stromový panel obsahuje metodu *onChange*, která je volána v případě změny vybrané složky. Tuto metodu lze přepsat a případně překreslit požadovanou komponentu nebo vložit data původnímu filtru. Nejdůležitější částí stromového panelu je třída *TreeAjaxBehavior*, která zajišťuje komunikaci s klientskou částí. Implementuje reakci na tlačítka smazat, vytvořit, přejmenovat, přesunout a vybrat. Na klienta je třeba poslat JSON s daty, které se budou zobrazovat. Tato data tvoří stromovou strukturu a podle ní *jsTree* sestaví strom. Nakonec se z důvodu přehlednosti zobrazují pouze složky a nikoliv segmenty. Aby bylo možné měnit umístění segmentu ve složce, tak se musí modifikovat *StatisticsPanel.* Ten teď obsahuje řádkový formulář, ve kterém lze tuto akci provést.

Komunikace mezi klientskou a serverovou částí musí být podrobně zdokumentovaná. Její popis lze nalézt v bodě 8.

6.1.3 Možnost přepočítání

Úkolem je upravit aplikaci tak, aby bylo možné jednotlivé segmenty přepočítat resp. poslat znovu požadavek na výpočetní backend. Dále je požadavek, že vědec si bude moci vybrat typ metody výpočtu. Služba, která se stará o komunikaci s výpočetním backendem, již obsahuje implementovanou metodu pro poslání požadavku o vý-



Obrázek 13: Nová UploadPage

počet segmentu. Nicméně aktuální datový model neudržuje informaci o tom, jakou metodou byl *segment* počítán. Proto tabulku segment rozšíříme o daný sloupec.

Do uživatelského panelu s výsledky je vloženo tlačítko pro přepočítání. Pokud je uživatelem vědec, tak toto tlačítko zobrazí vyskakovací okno pro přepočítání. Obsahem okna je formulář, který obsahuje všechny dostupné metody počítání. Po výběru metody je odeslán požadavek na výpočetní backend. V případě, že se nejedná o vědce, tak je požadavek poslán rovnou. Metoda počítání je brána z tabulky *segment* (je-li příslušný sloupec prázdný, tak se použije výchozí metoda počítání). Před odesláním požadavku je potřeba vynulovat pomocné vlajky a uložit metodu výpočtu.

6.2 UploadPage

Původní stránka pro nahrávání souboru obsahovala pouze formulář pro vložení souboru a poznámky. Cílem úpravy stránky je rozšíření možnosti nahrávání, a to především na uživatele typu vědec. Je požadována implementace výběru metody výpočtu, uchování typu nemoci, zadávání parametrů manuálně a možnosti přidat novou nemoc.

Na obrázku 13 lze vidět rozvržení této stránky. Stránka se skládá z nahrávacího panelu a panelu s výsledky. Panel s výsledky implementuje základní funkcionalitu a upravován a refaktorován je v bodě 6.1.2.

6.2.1 Refaktoring

UploadPanel se v aplikaci objevuje na několika místech. Panel má pokaždé mírně odlišnou funkcionalitu, která je rozlišována pomocí vlajek v konstruktoru. Stejně jako v předchozích případech je tento panel refaktorován, protože kód je nepřehledný.

Vznikne anonymní třída UploadPanel, která implementuje základní principy nahrávání. To znamená, že dostane soubor, který anonymizuje, uloží na server, rozparsuje, uloží data včetně poznámky do databáze a předá data k výpočtu. Jsou vytvořeny dva potomci této třídy:

1. anonymní nahrání - Po tomto nahrání se čeká, dokud nejsou k dispozici všechny výsledky nahraných segmentů. Získané výsledky jsou uživateli vykresleny pod formulář nahrávání. Jedná se pouze o ukázkovou funkcionalitu aplikace, tak není třeba tento upload dále rozšiřovat.

2. uživatelský upload - Zde se na výsledky nečeká, pouze se aktualizuje panel s nimi. Ve výchozím stavu má uživatelské nahrávání navíc pouze poznámku. Tento druh uploadu je dále rozšiřován.

6.2.2 Výběr metod

Cílem této úpravy je možnost volby metody výpočtu. Předpokládá se, že bude možnost vybrat více metod najednou. Dále je požadováno, aby přístup k této funkcionalitě měl pouze vědec. Při implementaci je potřeba klást důraz na jednoduchost změny metody či rozšíření o metodu novou.

Musí se tedy vytvořit vstupní pole pro zaškrtávání více položek, které je pojmenováno *CheckField*. Jeho funkcionalita je taková, že obsahuje popisek a seznam zaškrtávacích políček, přičemž jich může být zaškrtnuto více. Toto pole je vhodné udělat generické, protože ho bude možné využít v jiných místech a projektech. Toto pole obsahuje v konstruktoru modely *modelCheck* a *viewModel*. *ViewModel* složí k zobrazení všech zaškrtávacích tlačítek, které obsahují i popisek. *ModelCheck* složí k výchozímu nastavení zaškrtnutí a k vrácení všech zaškrtnutých možností.

Pro rozlišení metod je použit výčtový typ, který obsahuje všechny dosud známé metody. Jednotlivé položky načítají svoje specifické řetězce z properties souboru. Změna názvu metody je jednoduchá, jelikož stačí změnit tento soubor, a není třeba rekompilovat. Při vložení nové metody je do výčtového typu třeba definovat novou položku. To není optimální z důvodu nutné rekompilace. Výsledky z výpočetního

backendu jsou v závislosti na zvolené metodě ukládány vždy do jiné databázové tabulky. Z tohoto důvodu bude nutné kód upravovat. A to hlavně část pracující s prezentací výsledků, kde bude minimálně potřeba načítat data i z nově vytvořené tabulky. Z tohoto důvodu není možné se rekompilaci vyhnout, tudíž řešení s výčtovým typem je plně dostačující.

Teď už je možné přidat vytvořené pole do formuláře a upravit metodu uploadu. Během její modifikace dochází k problému se službou posílající požadavky na klienta. Její rozhraní neumožňuje výběr metody, takže je třeba její funkčnost modifikovat. Do této služby se bude předávat jak identifikátor segmentu, tak i počítaná metoda.

Od této doby je možné, že daný segment bude obsahovat více výsledků. Z tohoto důvodu je třeba k tomu uzpůsobit funkcionalitu všech částí aplikace, které pracují s výsledky. Například se jedná o prezentaci výsledků, přepočítávající tlačítko atd. Rozšíří se databázová tabulka *segment* o sloupec, který si pamatuje počítanou metodu. Je-li metod více, tak tento sloupec obsahuje řetězec MORE. Ten signalizuje, že se má služba pro zisk výsledků podívat do všech tabulek s výsledky. Dále veškerá prezentace výsledků musí pracovat s listem výsledků.

6.2.3 Manuální zadání parametrů

Tato funkcionalita do značné části vyplývá z předchozí úpravy. Požadavkem je, aby uživatel, opět pouze vědec, mohl zadat parametry ručně. Znamená to, že se nebude posílat žádný požadavek na výpočetní backend.

Ke splnění požadavku je potřeba vytvořit formulář, který umožní vložit výsledky manuálně. Tyto výsledky nelze vložit bez segmentů, takže právě rozšiřující se funkcionalita musí být součástí uploadu. Pod nahrávací formulář se vloží odkaz, který zobrazí vyskakující okno. V tomto okně si uživatel může nastavit parametry. Jakmile jsou parametry nastaveny, tak už uživatel nemá možnost nastavit volbu metody. Je zobrazena volba metody, která odpovídá zadaným parametrům. Pokud by si to rozmyslel, tak ve vyskakovacím okně má možnost resetovat parametry. Po této akci může nahrávat klasický způsobem.

Upload manuálně nahraných výsledků probíhá stejně. Uživatel může napsat poznámku a musí nahrát data atd. Po provedení základních funkcí uploadu se uloží výsledky do databáze. Oproti klasickému nahrávání se neprovádí posílání požadavku na výpočetní backend. Tato změna nijak neovlivňuje funkcionalitu dalších

částí aplikace.

6.2.4 Nemoci

Cílem této změny je umožnit zadání druhu nemoci. Ta je důležitá při vykreslování Parkesova chybového grafu. Dále se požaduje, aby vědec měl možnost vkládat nové onemocnění.

Aktuální datový model není připraven na toto rozšíření, a proto je ho třeba modifikovat. Vytvoří se nová tabulka *disease*, která bude obsahovat identifikátor (id), ICD[21] a popis nemoci v českém a anglickém jazyce. Id je primárním klíčem tabulky a je typu long. Je inkrementálně zvětšován. ICD je mezinárodní identifikátor nemoci. Pomocí tohoto identifikátoru lékaři rozlišují druh nemoci. Popisy těchto nemocí se udržují v obou jazycích kvůli lokalizaci a slouží pro znázornění nemoci neznalému uživateli. Dále je třeba rozšířit tabulku *segment* o nemoci. Těchto nemocí může mít uživatel více, stejně jako jednu nemoc může mít více lidí. Vazba mezi těmito tabulkami je M:N.

Formulář pro upload se rozšíří o podobné vstupní pole, jako v kapitole 6.2.2. Vytvoří se třída *DiseaseField*, která zmíněné pole dědí a mírně přepíše jeho funkcionalitu. Do seznamu klasického pole je třeba přidat odkaz, který zobrazí vyskakovací okno pro vytvoření nové nemoci (v případě vědce). Zde je jednoduchý formulář, který uživateli vynutí zadat ICD a popis nemoci v angličtině. Zadání českého popisu není nutné. Pokud český popis není zadán, nemoc se vypisuje anglicky i na česky lokalizovaných stránkách.

Tato funkcionalita vyžaduje i upravit panel se statistikami (bod 6.1). To tak, aby se zvolené nemoci uživateli zobrazovaly. Do tabulky s obecnými informacemi se přidá řádek, který dané nemoci vypíše, viz obr. 12.

6.3 WebSocket klient

WebSocketClient je služba, která reprezentuje klienta komunikujícího s výpočetním backendem přes websocket. Tato část aplikace se nachází na serveru a jejím nedostatkem je její závislost na pořadí spouštění výpočetního backendu a serveru. Před spuštěním serveru musí již být výpočetní backend zapnut. Server pak naváže spojení a komunikuje. Po odpojení výpočetního backendu již není schopný spojení znovu na-

[21] Mezinárodní klasifikace nemocí dle Světové zdravotnické organizace

vázat. Tato vlastnost je nevhodná z důvodu vynucování pořadí spouštění a nutného restartu serveru po odpojení výpočetního backendu. Proto je požadována taková úprava, která tento problém odstraní.

Další problém je ten, že se klient pokouší neustále posílat zprávu, i když je buffer výpočetního backendu plný. Vzhledem k tomu, že dochází ke zbytečnému zatěžování obou části systému, je potřeba tento problém vyřešit.

6.3.1 Přetěžování

Cílem této modifikace klienta je ošetření zbytečného zatěžování serveru a výpočetního backendu. Při analýze této změny je potřeba brát v úvahu, že výpočetní backend může být odpojen i delší dobu. Z tohoto důvodu je potřeba brát ohled i na velikost čekací fronty, ve které se udržují požadavky na odeslání zpráv.

První možností je považovat stav „plná fronta“ za chybný. To znamená, že uživatel dostane hlášku, že výsledek není k dispozici. Segment si díky předchozí implementaci může kdykoliv přepočítat. Druhá cesta vede přes časování, kde se udržuje plánovaný čas odeslání požadavku. Nastane-li správný časový okamžik, vlákno pro odeslání zpráv se vzbudí a pokusí se zprávu odeslat.

První případ sníží zatížení obou systémů na minimum. Na druhou stranu může uživateli ukazovat velké množství chyb, což pravděpodobně povede ke snížení použitelnosti aplikace. Z tohoto důvodu nelze toto řešení považovat za správné. Plánování nám poskytuje se vyhnout aktivnímu čekání, které je ve výchozí verzi. Jeho implementací se docílí zbytečného zatěžování obou částí systému. Nicméně zatěžování je větší než v prvním případě. Z pohledu velikosti čekací fronty je toto řešení nevhodné, a to v případě dlouhodobého výpadku. Pokud by bylo možné zamezit možnému nárůstu velikosti fronty, bylo by toto řešení vhodné. Zamezení velikosti fronty lze provést zkombinováním obou řešení. V praxi to znamená, že se bude provádět plánování. Jakmile se dosáhne daného množství pokusů, tak se považuje výsledek za špatný.

Plánování se provede tak, že při vkládání fronty je nastaven aktuální čas. Z této vlastnosti je jasné, že se vždy odesílací vlákno při vložení zprávy notifikuje. V případě, že výpočetní backend je přetížen nebo není připojen, je odeslání přeplánováno a inkrementuje se počítadlo pokusů. Přeplánování se provede tak, že k aktuálnímu času se přičte *timeout* vynásobený počtem pokusů. *Timeout* a počet pokusů jsou definovány v properties souboru, takže po změně těchto konstant není kód nutné re-

kompilovat.

Jakmile odesílací vlákno dokončí svoji iteraci odeslání, tak najde nejbližší čas odeslání. Následně se do tohoto času samo uspí. Během spánku ho samozřejmě může vzbudit jiné vlákno a to zmíněným vložením zprávy.

6.3.2 Připojování

Cílem této práce je dosáhnout nezávislosti pořadí spouštění a eliminace restartování serveru po ztrátě spojení s výpočetním backendem. Řešením tohoto problému je vyjmout navazování spojení do vlastního vlákna a následného zachycování callbacků reflektující s odpojením, ztrátou spojení atp.

Je vytvořena třída *HandshakeThread*, která se v pravidelných intervalech snaží navázat spojení. Jakmile je navázáno, vlákno se uspí. Vzbuzeno je až vláknem pro posílání zpráv a to tehdy, kdy je zachyceno odpojení nebo ztráta spojení s výpočetním backendem. Interval pro navazování spojení je opět získán z properties souboru.

6.3.3 Zotavení serveru

Během implementace požadovaných funkcionalit bylo zjištěno, že klient nijak neřeší možnost pádu serveru. Sice aplikace umožňuje u nespočteného výsledku nahlásit chybu. Dokonce u vědce je možnost u takového výsledku požádat o přepočítání. I přes tuhle již implementovanou funkcionalitu není tohle řešení elegantní. Z důvodu lepšího uživatelského pohodlí je do klienta implementována metoda, která okamžitě po volání konstruktoru, načte z databáze všechny nespočtené segmenty. Ty připraví ke spočtení na nejbližší půlnoc.

6.4 Skript

Poslední částí práce je vytvoření skriptu pro UNIX a Windows systémy, který ze zdrojových souborů vytvoří soubory pro nasazení na webu. Při spuštění skriptu se předpokládá, že jsou instalovány všechny potřebné softwary a že jsou řádně nastaveny. Je třeba mít nainstalováno:

- GIT - pro stažení aktuálního repozitáře,
- Maven - pro sestavení WAR[22] souboru,

[22] Webový archiv

- Emscripten - pro překlad C++ kódu,
- WinSCP - v případě Windows slouží k přenosu na server.

Ve skriptu se nacházejí čtyři proměnné, které je třeba modifikovat podle vzdáleného stroje. Prvním je *webapps*, která značí cestu složce Tomcatu. Proměnná *cop* značí složku pro ukládání záloh. *WarName* značí jméno WAR souboru, který se nasazuje na server. Poslední proměnná je *session* a slouží pro navázání spojení se vzdáleným serverem. K tomuto nasazení dochází pomocí programu WinSCP. Na operačním systému Linux se používá příkaz *sftp*, který je součástí shellu. Je potřeba upozornit na správnost těchto proměnných, které jsou závislé na operačním systému serveru. Pokud server běží na Linuxu, používají se dopředné lomítka (/). V případě Windows je potřeba zpětná lomítka zdvojovat (\\).

V prvním kroku jsou nejdříve staženy aktuální verze repozitářů. V celém projektu jich je pět. Vývoj webové aplikace probíhá pouze v *glucose-levels-prediction.1.* V dalším kroku se skript přepne do složky, ve které je nahrán C++ projekt. Zde se provede překlad do Javascriptu. K tomu překladu se používá *makefile*, který má klasickou strukturu. Obsahuje pravidla, která se mají vykonat před tím, než se provede vygenerování JS. Příkaz (em++) pro generování javaScriptu obsahuje parametr *EXPORTED_FUNCTIONS*, který zviditelní uvedené funkce pro ostatní javascripty. *Makefile* se vykoná příkazem *call emmake make.* Tímto příkazem vznikne soubor *paint.js*, který je potřeba překopírovat do složky s *webapps*.

V dalším kroku se pomocí *mavenu* provede sestavení *war* souboru. Tento soubor je následně přejmenován na požadovaný název (proměnná *warName)*. Teď je potřeba soubor přenést na vzdálený server. Naváže se spojení a soubor se vloží do složky uvedené v proměnné *webapps.* Dále se soubor překopíruje i do záloh. K jménu je přidán ještě aktuální datum a čas.

Po nastavení uvedených proměnných jen stačí spustit skript v příkazové řádce nebo terminálu. Sytém s proměnnými byl zvolen z důvodu stálosti jejich nastavení (nastaví se jednou a už se nemění).

7 Testování

Úkolem testování je otestovat robustnost a responzivnost aplikace. Na desktopu se bude testovat na následujících prohlížečích: Chrome 58.0.3, Firefox 53.0.3, Opera 45.0, Internet Explorer 11, Edge 38.14393.1066.0. Na telefonu bude aplikace otestována na Chrome 58.0.3.

Testování probíhá průběžně během vývoje. V této fázi se testuje pouze na prohlížeči Firefox. Testování se provádí manuálně, obyčejným proklikáváním aplikace. V případě chyby je psán test log, který popisuje, jak k dané chybě došlo. Jakmile jsou všechny chyby opraveny, tak dochází k nahrání změn do repozitáře.

7.1 Chyby ve funkčnosti

Chyby jsou nalezeny v Parkesově grafu, kde se špatně vykreslují body pro metodu diffusion 3. Chyba nastává pro každý nahraný segment. Druhou chybou je špatné počítání statistik u výchozího grafu. Statistiky znázorňují přesnost aproximace přímky v daných bodech. Požadavkem však bylo zobrazení statistik chyb vůči naměřeným hodnotám v krvi. Dále se pro absolutní chybu nepřepínaly jednotky. Všechny uvedené chyby byly opraveny v C++ kódu, který byl následně znovu přeložen do JS.

Nejvíce problémů bylo nalezeno ve stromovém filtru. Dochází k tomu, že složku nelze vytvořit do kořene, složku jde přesunout mimo kořen, při mazání je vyhozena výjimka. Tyto chyby lze opravit jednoduchou úpravou JS a *TreeAjaxBehaviour.* Nejzásadnější chybou je, že při mazání a přejmenování složek se změny neprojeví na jednotlivých filtrovaných položkách. To znamená, že se provede například přejmenování, ale změny se neprojeví na ostatních místech stránky. Tato chyba je nejzásadnější a potřebuje refaktorovat celou komponentu se stromem. Je třeba upravit práci s modely a aktualizovat je při jakékoliv práci se složkami.

7.2 Testování v různých prohlížečích

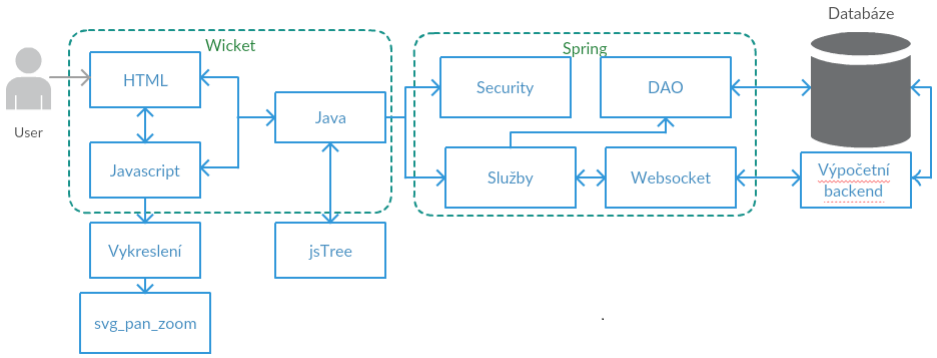
Funkčnost sice byla testována na prohlížeči Firefox, nicméně je nutné ji otestovat i na ostatních prohlížečích. Frameworky Wicket a Bootstrap jsou multiplatformní. Proto je pravděpodobné, že již v této části aplikace chyba nebude. Problémem může být především JS, který je potřeba důsledně otestovat.

Na backendu je nalezena chyba s porovnáváním lokále. Na Chromu se objevuje angličtina i přes nastavenou češtinu. Problém je vyřešen tím, že se již porovnávají řetězce jazyků oproti porovnávání řetězců lokále. Po opravě se uvedený problém nevyskytuje ani na ostatních prohlížečích.

Největší problémy nastaly dle očekávání na prohlížeči IE a Edge. Zde nefunguje jak ukládání souborů, tak JS vykreslení. To z toho důvodu, že nebyly použity multiplatformní příkazy. Tyto chyby byly opraveny modifikací souboru *chartMain.js*. Další problém nastává v případě knihovny pro přibližování grafu v prohlížeči Edge. Tento prohlížeč není totiž knihovnou podporován. Popularita Edge je v řádech jednotkách procent [14]. Z tohoto důvodu padlo rozhodnutí, že nová implementace přibližování se nevyplatí. Funkcionalita bude upravena pouze tak, aby nedocházelo k chybnému vykreslení.

7.3 Testování na telefonu

Při testování na telefonu byla testována responzivita aplikace. Veškeré vizuální nedostatky, včetně správného zarovnávání komponent, jsou v průběhu této fáze odstraněny. Dalším krokem testování na telefonu je ověření funkcionalit. Stejně jako v předchozím případě je potřeba řádně otestovat veškeré JS. Během této fáze vzniká problém se skrýváním elementů v grafech. Chybí totiž inicializovat posluchače i na událost *touchstart*. Do této doby se event *onClick* vykonával z SVG souboru. Teď je C++ kód i *chartMain.js* upraven tak, že je možné ukrytí elementů měnit bez překladu C++ kódu. To je děláno přes posluchače událostí. Posluchač reaguje na událost *click* a *touchstart*.



Obrázek 14: Nová architektura

8 Architektura

V bodě 6 bylo rozhodnuto, že se z důvodu časové náročnosti, efektivity a dalších zdrojů nevyplatí měnit architekturu rozšiřující se aplikace. Architektura se od původní architektury výrazně neliší. Oproti původní architektuře jsou přidány dvě knihovny. A to knihovna pro manipulaci s SVG obrázkem a knihovna implementující strom složek. Aktuální architektura je na obr. 14. Z obrázku je zřejmé, že oproti původní verzi je vykreslení vyjmuto z modulu Javascript. To má hned dva důvody. První je nutnost znázornění principu manipulace s obrázkem. Druhým důvodem je potřeba specifikace části, která byla překládána pomocí emstriptenu. Z obr. 14 je také vidět, že se i nadále používají technologie Wicket a Spring.

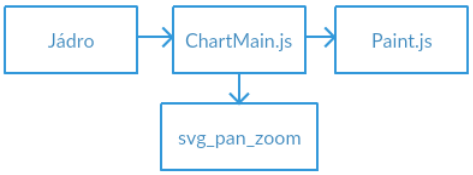
Celou architekturu je nutné důkladně zdokumentovat. V dokumentaci není důležité popisovat, jak je specifická komponenta implementována, ale jaké je mezi jednotlivými komponentami rozhraní. Princip generování stránek ve frameworku Wicket je popsán v bodě 2.3. Princip vkládání závislostí pomocí IoC je popsán v bodě 2.5.1. Služba má přesně definovanou a zapouzdřenou funkcionalitu. Její rozhraní je důkladně okomentováno pomocí JavaDoc přímo v kódu.

8.1 Vykreslení

Vykreslení se skládá ze čtyř hlavních komponent, obr. 15:

1. jádro - reprezentuje wicketovské jádro aplikace. Odtud se volají odpovídající javascriptové metody s požadovanými parametry.

2. chartMain - Propojuje funkcionalitu vykreslení s jádrem. Jsou zde implementovány metody, které jádro může volat. ChartMain může volat jednotlivé me-



Obrázek 15: Komponenty vykreslení

tody, které provedou vykreslení. Vytváří instance *svg_pan_zoom*, které poté obstarají manipulaci s obrázkem.

3. paint - Implementuje generování SVG řetězců. Vznikne překladem za využití emstriptenu.

4. svg_pan_zoom - Obstarává manipulaci s obrázkem

Nejzásadnější roli hraje komponenta ChartMain, která propojuje veškeré části vykreslení. Implementuje funkce pro vizualizaci požadovaného grafu (obecný, Parkes, Clark, AGP) a doplňující akce (stažení souborů, skrývání oblastí).

8.1.1 Hlavní funkcionalita

Význam parametrů a funkcionalita těl funkcí je vždy stejná. Těla funkcí se liší jen ve volání funkcí definovaných v modulu Paint.js. Ukázka propojení je zobrazena na obecném grafu. Zavolá se funkce *startCalculations*, která obsahuje parametry:

- json - JSON s potřebnými daty v požadovaném formátu.
- chartId - jedinečný identifikátor okna, do kterého se má graf vykreslit.
- inWindow - charakterizuje, jestli se zobrazuje graf v okně.

JSON se skládá z několika objektů:

- ist - obsahuje pole naměřených hodnot v podkoží. Naměřená hodnota je charakterizována objektem, který obsahuje *date* ve formátu *YYYY-MM-DDThh:mm:ss* a *value*. Ta je udána jako desetinné číslo. Příkladem ist může být:

  ```
  "ist":[{"date":"2015-04-16T08:01:00","value":13.2},{"date
      ":"2015-04-16T08:06:00","value":13.2}]
  ```

- blood - obsahuje pole naměřených hodnot v krvi. Formát stejný jako v podkoží.

- par2 - obsahuje všechny údaje o výsledcích, které se počítaly metodou diffusionGeneral2. Obsahuje atributy: p, cg, k... Dále může obsahovat atribut *visible*, který uvedené údaje zneviditelní pro vykreslení.

- par3 - obsahuje všechny údaje o výsledcích, které se počítaly metodou diffusionGeneral3. Obsahuje atributy: p, cg, kpos, kneg, tau... Také může obsahovat vlajku *visible.*

- strings - obsahuje mapu řetězců ve formátu klič-hodnota. Tyto řetězce jsou používány k popisování grafu. Aby byl řetězec zobrazen na správném místě, musí mít správný klíč (viz příslušný grafový panel v Java kódu).

JSON může být doplněn o číslo, které značí typ diabetu. Toto číslo se nazývá *disease*. Může nabývat hodnot 1 - diabetes typu 1 a jiné číslo - diabetes typu 2. Pokud toto číslo není uvedeno, předpokládá se diabetes typu 1.

Dalšími metodami pro vykreslení grafů jsou: *startAgp*, *startParkesGrid*, *startClarkGrid*.

Postup volání metod komponenty Paint je již popsán v kapitole 5.2.2. Vstupem metod pro generování grafů je více JSON řetězců reprezentující položky popsané v předchozím bodě. Pomocnými parametry je i vlajka pro typ jednotek. Je-li tato vlajka 1, tak hodnoty budou v mmol/l. Ze sekvence metod využívající Emscripten se získá SVG řetězec charakterizující daný soubor. Jakmile je tento řetězec získán, tak je zavolána metoda *generateSvg*, která z daného řetězce udělá DOM. Tento objekt je vložen do elementu uvedeného v *chartId* (tento element musí existovat).

Dále se inicializuje knihovna *svg_pan_zoom*, která se provede:

```
var panelInstance = svgPanZoom(svg,{
        zoomEnabled: true,
        controlIconsEnabled: true,
        fit: true,
        center: true,
        minZoom: 0.1
});
```

Zde se nastavují dané funkcionality přibližování. Například se jedná o umožnění zoomu, nastavení minimálního přiblížení, centrování, zpřístupnění tlačítek. Do *svgPanZoom* se jako první parametr dává SVG element, se kterým se má manipulovat.

název	id v legendě	id křivky	barva
naměřená koncentrace v krvi	blood_click	bloodCurve	červená
naměřená koncentrace v podkoží	ist_click	istCurve	modrá
metoda diffusion2	diff2_click	diff2Curve	oranžová
metoda diffusion2	diff3_click	diff23Curve	zelená

Tabulka 1: Identifikátory grafu

8.1.2 Vedlejší funkcionalita

V kapitole 5.2.1 bylo rozhodnuto, že pomocí Emscriptenu bude překládána pouze základní funkcionalita. Rozhodnutí nijak neovlivňuje komunikaci jádra s ChartMain. Jádro vždy zavolá příslušnou metodu *(download, download_csv, change_unit*). Ta provede danou funkcionalitu, která je popsána v bodě 5.2.3.

Speciálním případem grafu je AGP. Tento graf spočítá pro určitý denní okamžik všechny dostupné koncentrace (vypočítané koncentrace v krvi a naměřené koncentrace v podkoží). V každém tomto okamžiku počítá jednotlivé kvartily, které pak zobrazuje. Pokud si uživatel přeje zneviditelnit nějaká data, tak tato data musí být odebrána i z jednotlivých časových okamžiků. Musí být také přepočítány kvartily. Z tohoto důvodu je nutné graf vždy překreslovat. Zneviditelnění parametrů par2 a par3 lze provést přidáním atributu *visible* do příslušného JSON řetězce. Aby bylo možné skrývat i naměřené hodnoty z podkoží, je k parametrům funkce *agp_main* (z modulu Paint.js) přidána vlajka charakterizující viditelnost těchto hodnot (pro výpočet jsou potřeba, ale jednotlivé hodnoty se nezahrnují do výsledného grafu). Je-li nastavena na 0, dojde k zneviditelnění naměřených hodnot v podkoží.

U ostatních grafů se zneviditelňování křivek provádí pomocí manipulace s DOM. K zobrazení zneviditelnění se provádí změnou barvy příslušného řetězce, který je také získán z DOM. Po kliku na řetězec dojde k zneviditelnění části grafu a zšedivění řetězce. Pro tuto funkcionalitu jsou důležité identifikátory, viz tab. 1. V prvním sloupečku je název elementu, kterého se daný identifikátor týká. Druhý je identifikátor v legendě a třetí obsahuje identifikátor elementu, který zaobaluje vykreslenou křivku. Ve třetím je uvedena barva, kterou se uvedené elementy zobrazí.

8.2 Strom

Strom se skládá ze tří částí, viz obr.16: Jádro, knihovna jsTree a Tree.js, který první dvě komponenty integruje. Je potřeba upozornit na obousměrnou komunikaci. Lze

Obrázek 16: Komponenty stromu

totiž se stromem manipulovat pomocí tlačítek, ale i pohybem myši (například přesun složky). Tlačítka volají metody Tree.js z jádra, zatímco druhý příklad pomocí komponenty Tree.js zachycuje zpětné volání (tzv. callback) jsTree. Změny jsTree je potřeba propagovat až do jádra. V jádru je důležité veškeré změny validovat. Pod tím si lze představit například kontrolování toho, zda je manipulováno se složkou náležící správnému uživateli, zda daná složka existuje atd. Na straně serveru (v jádře) je implementována třída *TreeAjaxBehavior*, která zajišťuje tuto obousměrnou komunikaci. Komunikace je zajištěna pomocí AJAX. Aby se dalo se stromem komunikovat, musí být nejprve inicializován.

8.2.1 Vytvoření stromu

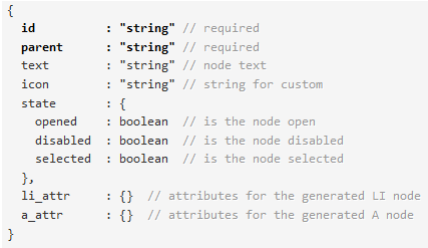
Jádro vytvoří strom voláním metody drawTree (z komponenty jsTree). Tato metoda vytvoří instanci stromu z komponenty jsTree. Metoda potřebuje znát tři parametry:

- id - identifikátor elementu, do kterého se strom vkládá.
- url - na jakou URL se budou posílat AJAX požadavky. Slouží při „zpětné“ komunikaci.
- data - JSON řetězec reprezentující data.

Data jsou reprezentována jednotlivými uzly a jsou uspořádána do stromové struktury, podle které se pak strom vykresluje. Jednotlivé uzly musí mít přesný formát, který je zobrazen na obr. 17. Identifikátor musí být jedinečný.

Aplikace se skládá ze tří druhů uzlů. První je kořen, se kterým nelze nijak manipulovat a je vrcholem celého stromu. Druhý je složka, která může být mazána, přejmenována, přesunuta atd. Posledním typem je segment, který lze přejmenovat a přesunout. Identifikátory se pak skládají z prvního písmene jejich druhu (r, f, s) a čísla (primární klíč z databáze). Tímto složením se docílí unikátnosti identifikátoru a zároveň je umožněna klasifikace druhu uzlu.

V této metodě se i definují případy, kdy se mají volat callbacky. Například s kořenem nelze hýbat, a proto je pro něj potřeba volání callbacku zakázat. Nebo je třeba



Obrázek 17: Data uzlů stromu

ošetřit, že segment nemůže být rodičem. Dále zde lze definovat rozšiřující funkce stromu, jako je například: přidání drag&drop plugin, kontextové menu, řadící plugin atd.

Je nutné upozornit na to, že vrcholy typu segment nejsou ve finální fázi stromu vidět. Důvodem tohoto rozhodnutí byla přehlednost při velkém počtu segmentů.

8.2.2 Funkcionalita stromu

Pro komunikaci z jádra do Tree.js jsou vytvořeny metody:

- *my_remove_node* - odstraní uzel. V parametru je identifikátor mazaného uzlu.
- *my_rename_node* - připraví uzel k přejmenování. Text u uzlu je připraven k modifikaci. V parametru je identifikátor přejmenovávaného uzlu.
- *show_tree* - z vybraného uzlu vezme všechny segmenty, které pošle zpět na server (jádro). V parametru je identifikátor procházeného uzlu.

Pro opačnou komunikaci je na serveru implementována třída *TreeAjaxBehavior*, která obsahuje metodu *respond*. Tato metoda reaguje na příchozí zprávy, například mění údaje v databázi, vybírá zvolené segmenty nebo si pamatuje vybraný uzel. Zpráva může obsahovat tyto parametry:

- *method* - signalizuje, jaká metoda se má provést. Může obsahovat tyto údaje: *selected*, *move*, *rename*, *create*, *delete*, *deselect* a *show*.
- *id* - číselný identifikátor uzlu.
- *type* - typ uzlu. Může nabývat hodnot *segment*, *root*, *folder*.
- *parent* - id rodičovského uzlu.
- *parentOld* - id starého uzlu.

- *children* - identifikátory dětí.
- *name* - nové jméno
- *segments* - identifikátory vybraných segmentů.

Každý zpráva vždy musí obsahovat parametr *method*, podle kterého je charakterizována činnost serveru. Každá zpráva ještě obsahuje atribut *id* a *type* (s výjimkou *show* a *deselect*).

Metoda *selected* si zapamatuje zvolený uzel, zatímco *deselect* resetuje zvolenou volbu. *Move* obstarává přesun a požaduje atributy *parent* a *parentOld*. *Rename* přejmenovává zvolený uzel a potřebuje atribut *name*. *Create* vytvoří složku a očekává atribut *parent*. *Delete* maže složku a obsahuje *parent* a *children*. Součástí *Show* jsou všechny segmenty (atribut *segment*) vyskytující se v dané složce (včetně zanoření). Takto může vypadat ukázka zprávy s přejmenováním:

```
var wcall = Wicket.Ajax.get({ u: url, "ep":[
        {"name":"method","value":"rename"},
        {"name":"id","value":node.id},
        {"name":"type","value":node.type},
        {"name":"name","value":node_position}
]});
```

8.3 Výpočetní backend

Komunikace mezi výpočetní backendem a websocket klientem se provádí pomocí Websocketů. Obsahem zprávy je JSON řetězec. Klient posílá požadavek na výpočetní backend v této podobě:

```
{
        "Command":"EnqueueTimeSegment",
        "Parameters":
        {
                "TimeSegmentDbId":1,
                "Method:"Diffusion2General"
        }
}
```

Do atributu *TimeSegmentDbId* se nastavuje identifikátor segmentu. Do atributu *method* se vkládá metoda měření.

Výpočetní backend odpovídá klasickými HTTP status kódy. Ty jsou obsahem JSON řetězce, který je posílán v těle odpovědi. Odpověď vypadá takto:

```
{
        "Result":400,
        "Comment":"Malformed, incomplete or unknown command.",
        "TimeSegmentDbId":1
}
```

Result vrací zmíněný HTTP status kód a *TimeSegmentDbId* vrací id segmentu, kterého se tato zpráva týká. Zpráva může vracet tyto status kódy:

- 200 - příkaz přijat a počítá se
- 202 - příkaz přijat a zařazen do fronty
- 303 - segment byl spočítán
- 400 - neznámý nebo špatný příkaz,
- 410 - segment nebyl spočítán,
- 429 - fronta je plná

8.4 Properties

Důležitou částí jsou i hodnoty jednotlivých konstant, které lze v aplikaci nastavit. Tyto konstanty jsou načítány z properties souboru a obsahuje tyto konstanty:

- farmer.url - URL výpočetního backendu
- farmer.timout - čas v milisekundách mezi jednotlivými pokusy o navázání spojení s výpočetním backendem.
- failed.attempt - počet pokusů o poslání zprávy. Po kolika pokusech je segment označen za chybný.
- failed.timeout - čas v milisekundách mezi jednotlivými pokusy o poslání zprávy na výpočetní backend.

- nemoci - dále jsou zde specifikovány názvy českých a anglických popisů nemoci.

9 Závěr

Hlavním úkolem této práce bylo implementovat požadované změny portálu a to tak, aby byla zachována responzivnost webové aplikace. Té bylo dosaženo pomocí frameworku Bootstrap, který poskytuje třídy zajišťující responzivnost. Práce spočívala v seznámení se s webovými technologiemi, mezi které patří především Wicket a Spring. S oběma technologiemi jsem se během studia již setkal. Z toho důvodu pro mě nebyl problém s jejich používáním. Je ovšem pravda, že jsem objevil nové možnosti těchto technologií. Ve Wicketu se jedná například o komunikaci z javascriptu na server. Ve Springu jsem se poprvé setkal s používáním Websocketů. Musím říct, že mě velice překvapilo, s jakou jednoduchostí je lze díky frameworku Spring použít.

Největší výzvou jednoznačně byla práce na generování grafů, a to kvůli knihovně Emscripten. Přizpůsobení kódu tak, aby ho bylo možné přeložit a nasadit do provozu, přineslo celou řadu problémů. Nicméně nakonec se tuto část práce podařilo implementovat s požadovanou funkcionalitou. Myslím si, že se velice povedl graf AGP, který je vizuálně hezký, přehledný a obsahuje vysokou míru informace.

Do budoucna by si aplikace zasloužila implementování manipulace s obrázkem i na prohlížeči Edge. Dále by si zasloužila modernější a propracovanější vzhled. Pokud by se práce měla rozšiřovat, tak by se vyplatilo zvážit implementaci automatických testů, protože už v této fázi existuje poměrně hodně scénářů.

I přes uvedený problém s prohlížečem Edge lze konstatovat, že práce byla splněna v celém svém rozsahu.

Reference

[1] Microsoft Application Architecture Guide [online]. 2nd Edition. Amazon, 2009 [cit. 2017-04-09]. ISBN 9780735627109. Dostupné z: https://msdn.microsoft.com/en-us/library/ff650706.aspx

[2] A web application based on the MVC architecture using the Spring Framework - Panchal, Hardikkumar B.. California State University, Long Beach, ProQuest Dissertations Publishing, 2016. [cit. 13.06.2017]. Dostupné z:http://search.proquest.com/openview/9f37d42e3389fe60e583f9b02babf3da/1?pq-origsite=gscholar&cbl=18750&diss=y

[3] LONGO, João Sávio Ceregatti a Felipe Fedel PINTO. Instant Apache Wicket 6. 1th edition. Birmingham, UK: Packt Publishing, 2013. ISBN 1783280026.

[4] RASOCHA, Marek. Responsivní design portálu pro správu organizací. Plzeň, 2015 [cit. 17.04.2017]. bakalářská práce (Bc.). ZÁPADOČESKÁ UNIVERZITA V PLZNI. Fakulta aplikovaných věd

[5] Ajax: A New Approach to Web Applications. Jesse James Garrett [online]. James Garrett, 2005 [cit. 17.04.2017]. Dostupné z: https://pdfs.semanticscholar.org/c440/ae765ff19ddd3deda24a92ac39cef9570f1e.pdf

[6] Spring Framework Reference Documentation [online]. Copyright © 2004 [cit. 17.04.2017]. Dostupné z: http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/index.html

[7] SCHAEFER, Chris; HO, Clarence; HARROP, Rob. Pro Spring. Apress, 2014.

[8] MORDANI, Rajiv a Linda DEMICHIEL. Common Annotations for the Java ™ Platform [online]. 2016 [cit. 13.6.2017]. Dostupné z: http://download.oracle.com/otn-pub/jcp/common_annotations-1_3-mrel3-spec/jsr-250.pdf?AuthParam=1497352493_0a0005516bf9dd05af5e3b5e0e0465f5

[9] asm.js. asm.js [online]. Dostupné z: http://asmjs.org/spec/latest/#introduction

[10] Main — Emscripten 1.37.9 documentation.· GitHub Pages [online]. Copyright © Copyright 2015, . [cit. 18.04.2017]. Dostupné z: http://kripken.github.io/Emscripten-site/index.html

[11] Angular Docs. Angular Docs [online]. Dostupné z: https://angular.io/guide/architecture

[12] Web Testing: Complete guide on testing web applications — Software Testing Help. Software Testing Complete Guide — Software Testing Help [online]. 2017 [cit. 21.04.2017]. Dostupné z: http://www.softwaretestinghelp.com/web-application-testing/

[13] LI, Yuan-Fang; DAS, Paramjit K.; DOWE, David L. Two decades of Web application testing—A survey of recent advances. Information Systems, 2014

[14] Internet Explorer Browser. W3Schools Online Web Tutorials [online]. Dostupné z: https://www.w3schools.com/browsers/browsers_explorer.asp

[15] Cheerp - C++ for the Web. [online]. Copyright © 2013 [cit. 24.06.2017]. Dostupné z: http://leaningtech.com/cheerp/index.html

Přílohy

DB schéma