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

Bakalářská práce

Návrh a implementace serveru pro podporu vizualizace přenosové soustavy

Plzeň 2017

Lukáš Černý

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 28. dubna 2017

Lukáš Černý

Poděkování

Děkuji panu Ing. Richardu Lipkovi, Ph.D. za odborné vedení mé bakalářské práce a za cenné rady, které mi pomohly tuto práci vytvořit.

Abstract

The bachelor thesis follows the ongoing project, which was created in cooperation with the Department of Cybernetics together with the Department of Informatics and Computer Science. The project deals with the visualization of the electric transmission network. This work is part of the second phase of development where the main goal is to design and implement a server application that will provide available data through the API. The application is functionally based on the original prototype, but uses other technologies, and the user must be logged in for the content to be available. The application is implemented in the Java programming language.

Abstrakt

Bakalářská práce navazuje na probíhající projekt, který vznikl ve spolupráci katedry kybernetiky spolu s katedrou informatiky a výpočetní techniky. Projekt se zabývá vizualizací elektrické přenosové sítě. Tato práce je součástí druhé fáze vývoje, kde je hlavním cílem návrh a implementace serverové aplikace, která bude poskytovat data dostupná přes API. Aplikace funkčně vychází z původního prototypu, ale využívá jiné technologie a obsah je dostupný až po přihlášení. Aplikace je implementována v programovacím jazyce Java.

Obsah

1 Úvod **6**

2 Nástroje a technologie při vývoji podnikových aplikací **7**

2.1 Jak vyvíjet podnikové aplikace . . . 7

2.1.1 Java EE . . . 7

2.1.2 Spring . . . 8

2.2 Návrh rozhraní aplikace . . . 8

2.2.1 Swagger . . . 8

2.2.2 Apiary . . . 10

2.3 Aplikační servery . . . 10

2.3.1 Apache Tomcat® . . . 10

2.3.2 Apache TomEE . . . 11

2.3.3 WildFly . . . 11

2.4 Autentizace a autorizace . . . 11

2.4.1 PicketLink . . . 12

2.4.2 Apache Shiro . . . 13

3 Současné navržené řešení **15**

3.1 Architektura . . . 16

3.1.1 Uživatelské rozhraní . . . 16

3.1.2 Server . . . 16

3.2 Popis API . . . 17

3.2.1 Grafová data . . . 18

3.2.2 Modely . . . 22

3.2.3 Přepočítaná data . . . 22

3.2.4 Navržené změny . . . 24

4 Popis implementace **29**

4.1 Navržené řešení aplikace . . . 29

4.1.1 Server a databáze . . . 29

4.1.2 Implementace API . . . 30

4.1.3 Struktura balíků . . . 30

4.2 Identity model . . . 33

4.2.1 Autentizace . . . 34

4.2.2 Autorizace . . . 36

4.3 Klient pro správu uživatelů . . . 38

5 Testování 39
5.1 Jednotkové testy 39
6 Závěr 41
Přehled zkratek 42
Přílohy 43
Literatura 47

1 Úvod

Elektrická síť je složena z velkého množství prvků. Mezi ně patří zdroje energie např. větrné, sluneční elektrárny, ale patří mezi ně i solární panely na domech. U těchto zařízení nelze regulovat výkon a jsou závislé na klimatických podmínkách. To může přinášet problémy, protože přebytky jsou pouštěny do sítě a ostatní elektrárny musejí regulovat výkon. Aby bylo možné regulovat výkon, je zapotřebí dokázat modelovat stav sítě.

Na síť můžeme nahlížet jako na graf, kde jednotlivé uzly reprezentují různá elektronická zařízení (např. transformátor), ale jedná se i o elektrárny. Hrany grafu představují rozvodnou síť, skrz kterou proudí napětí mezi uzly. Elektrické sítě nabývají velkých rozměrů a jejich grafová reprezentace obsahuje velké množství hran a uzlů. Pohlížet na graf jako na celek je obtížné, a proto je dobré mít nástroj pro jeho vizualizaci.

Katedra kybernetiky spolu s katedrou informatiky a výpočetní techniky zahájily projekt, který má za cíl vytvořit webovou aplikaci pro vizualizaci elektrické přenosové sítě. Cílem projektu je graficky zobrazit celou přenosovou soustavu na geografické mapě s možností snadné změny parametrů a následného výpočtu nového stavu sítě a jeho zobrazení. Vizualizovaný graf může nabývat velkých rozměrů, klade se také důraz na efektivitu implementovaných výpočtů a programovací jazyk, který nebude chod aplikace zpožďovat.

Moje práce navazuje na probíhající projekt a přispěl jsem tím, že jsem vylepšil původní verzi prorotypu serveru. Zabýval jsem se tím, jak vylepšit rozhraní komunikace s vizualizací a odolnosti vůči vnějšímu napadení.

2 Nástroje a technologie při vývoji podnikových aplikací

Jedním z bodů vývoje je vytvoření abstraktního návrhu aplikace. Architektura aplikace může být navržena v programech (např. Enterprise Architect), které jsou nezávislé na programovacím jazyku a architektura je popsána diagramem. Z této aplikace lze vygenerovat zdrojové kódy do různých programovacích jazyků a implementaci jednotlivých částí distribuovat mezi vývojáře.

V našem projektu byl zvolen programovací jazyk Java. Podle statistik na serveru `www.tiobe.com` se nachází na prvních příčkách. Jedná se o jeden z nejpoužívanějších programovacích jazyků, existuje pro něj řada hotových technologií a udržovaných knihoven, které jsou k dispozici se svobodnou licencí.

2.1 Jak vyvíjet podnikové aplikace

Rozhodneme-li se vyvíjet podnikové aplikace v Javě, nebudou nám stačit knihovny, které jsou implementovány ve standardní Javě (Java Standard Edition). Abychom rozšířili funkčnost Javy SE, můžeme využít např. frameworku *Spring* nebo využívat technologie dané specifikací *Java EE*.

2.1.1 Java EE

V každé sféře podnikání musí společnosti čelit stále se zvyšujícím požadavkům na rychlejší přístup k datům a bezpečnost bez zvyšujících se nákladů na vývoj. Ve snaze vyhovět těmto požadavkům vznikla nová edice Javy zvaná Java Enterprise Edition, která má za cíl „Napiš jednou, spusť všude“.

Specifikaci Java EE navrhuje *Java Community Process* (viz `www.jcp.org`). Specifikace obsahuje definici, jak mají jednotlivé technologie fungovat, nejedná se o implementaci. Do vývoje jsou zapojeny komerční společnosti, open-source organizace a experti z oboru [1]. V současné době existuje více než 20 aplikačních serverů, na kterých lze spustit aplikace vyvíjené podle specifikace.

Specifikace obsahuje (obr. 2.1) rozhraní pro vývoj webové části, přístup do databáze, řeší problém vkládání závislostí, transakčnost a další technologie, které jsou při vývoji enterprise aplikace důležité [2] [3].

2.1.2 Spring

Spring je framework, který poskytuje sadu modulů pro vývoj aplikací v programovacím jazyce Java. Framework je rozdělen do 20 modulů, které jsou sloučeny do skupin a obslouží všechny vrstvy vícevrstvé aplikace. Spring využívá vlastnosti poslední verze Javy a implementuje technologie dané specifikací Java EE, které jsou dále rozšiřovány [4].

2.2 Návrh rozhraní aplikace

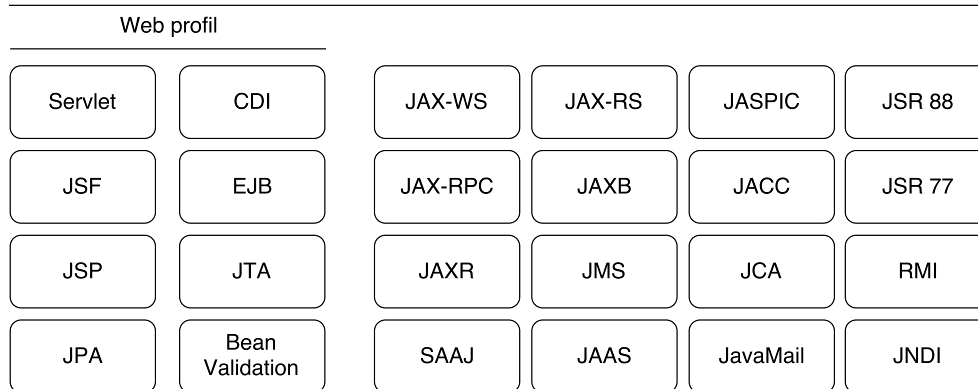
Vývoj může být rozdělen do několika částí. Nejdříve je nutné si definovat pro jaké účely bude aplikace sloužit, o jak rozsáhlý projekt půjde a kolik uživatelů bude k aplikaci přistupovat.

Budeme-li vyvíjet pouze webovou aplikaci, která má být úzce spojena s webovými technologiemi, je možné využít frameworky, které jsou postavené na technologii JSP. Náš projekt navazuje na předešlou část projektu, kde byla grafická (klientská) část oddělena od serverové, data jsou získávána přes API a následně zobrazovaná přes JavaScriptový framework Angular. Oddělení klientské a serverové části je vhodné, protože změna v jedné části nevynucuje změnu i ve druhé části.

Jak jsem psal výše (viz kapitola 2.), i rozhraní mezi klientskou a serverovou částí (API) je potřeba nejdříve definovat a navrhnout před samotnou implementací. Pro tyto účely můžeme využít služby např. *API Blueprint, Swagger* nebo nástroj od českých vývojářů *Apiary*.

2.2.1 Swagger

Swagger je volně šířitelný framework, obsahující sadu třech nástrojů (*Swagger Editor, Swagger Codegen, Swagger UI*) na vytváření a dokumentování RESTfull API [5].



Obrázek 2.1: Technologie specifikace Java EE[1].

Swagger Editor (viz `www.swagger.io/swagger-editor`) je nástroj využívaný pro návrhování a dokumentování RESTfull rozhraní. Dokument vytvořený v editoru je psán ve formátu *YAML*. Díky tomuto formátu je dokumentace čitelná člověkem ale i strojem, který text může dále zpracovat. Do editoru lze také nahrát soubor ve formátu *JSON*. Program je možné spustit online na webových stránkách, na systému *Node.JS*, nebo v kontejneru v aplikaci *Docker* [6].

Swagger Codegen (viz `www.swagger.io/swagger-codegen`) umožňuje navrženou specifikaci vygenerovat do zdrojových kódů různých programovacích jazyků klientské části tak i serverové [7].

Swagger UI (viz `www.swagger.io/swagger-ui`) vizualizuje specifikaci do stukturovaného dokumentu. Jednotlivé cesty rozhraní jsou rozděleny do přehledných boxů, kde je vidět detailní popis vstupních a výstupních dat, zabezpečení a další definované vlastnoti [8].

Na základě výše popsané funkčnosti Swaggeru jsem se rozhodl tento framework použít pro dokumentaci API projektu. Za největší výhodu považuji vygenerování zdrojových kódů ze specifikace do různých programovacích jazyků.

[1]Zdroj: `https://jaxenter.com/wp-content/uploads/2011/06/Introducing-the-Java-EE-Web-Profile-figure-one.jpg`

2.2.2 Apiary

Budete-li vyvíjet aplikaci, která bude mít rozsáhlou funkčnost přístupnou přes API, doporučuji použít *Apiary*. Jedná se o službu se sadou nástrojů pro návrh specifikace, testování funkčnosti a správu vývojářského týmu [9].

Jelikož se jedná o službu nabízející komplexní nástroje, pro plné využítí je zpoplatněna měsíční částkou. K dispozici je i zdarma verze, která umožňuje přístup k nástrojům pro vytvoření API specifikace a nástrojům pro testování, ovšem nenabízí služby pro řízení týmu.

Apiary Editor nabízí podporu Swaggeru a API Blueprintu. Jedná se o totožný editor, který využívá Swagger (viz sekce 2.2.1.), struktura zápisu je taktéž ve formátu *YAML*. Nevýhodou editoru oproti Swaggeru je nutný stálý přístup na internet. Editor je dostupný pouze z webové stránky na serverech Apiary [10].

Výhodou Apiary jsou nástroje pro testování. Nabízí vytvoření mokovacího serveru a automaticky testovat navržené API ze specifikace.

2.3 Aplikační servery

Pokud aplikace implementuje část technologií popisující specifikace Java EE, je nutné pro spuštění aplikace použít aplikační server. Aplikační servery se rozlišují podle toho, které technologie implementují. Mohou implementovat všechny technologie nebo pouze jejich část (obr. 2.1).

2.3.1 Apache Tomcat®

Tomcat je server implementující pouze technologie *servletů, JSP, EL, WebSocket*. Takto vybavený server je vhodné použít pro jednoduché webové aplikace.

Tomcat obsahuje jen velmi malé množství technologií, které se v dnešních podnikových aplikacích používají. Server je vhodné využít pro malé projekty, kde nebude potřeba dodatečně dodávat velké množství knihoven. Pokud bychom se rozhodli přidat další technologie, je to možné, ovšem aplikace zvětší svoji velikost a bude nutné stále hlídat aktuálnost a stabilitu knihoven. Za těchto podmínek je výhodnější využít server, který bude obsahovat technologie Javy EE.

2.3.2 Apache TomEE

Apache TomEE rozšiřuje funkčnost serveru Apache Tomcat®. Do serveru jsou přidány další technologie Javy EE. Konkrétně se jedná o *CDI, EJB, JPA, JSF, JSP, JSTL, JTA, Servlet, Javamail* a *Bean Validation* [11]. Server je (stejně jako Tomcat) vydáván pod svobodnou licencí *Apache License* v aktuální verzi.

Předchozí verze projektu byla spouštěna na serveru Tomcat. Pro aktuální verzi jsem využil server TomEE, protože implementuje více technologií a nemusí být distribuovány spolu s aplikací. Další výhodu spatřuji v tom, že se jedná o Java EE server. Aplikace může být spuštěna na jiném serveru implementující technologie Java EE.

2.3.3 WildFly

Wildfly (dříve JBoss) je aplikační server vyvíjený společností Red Hat. Server implementuje plnou specifikaci Javy EE a i přesto se jedná o flexibilní, lehký a snadno konfigurovatelný aplikační server. Lehký v tomto smyslu znamená, že je v serveru implementovaná agresivní správa paměti. To má za následek, snížení dat na haldě a minimalizaci způsobených garbage kolekcí. Server je možné konfigurovat přes konfigurační soubory, příkazovou řádku, nativní Java API, požadavky na REST API, JMX bránou nebo přes webové rozhraní. Právě webové rozhraní je přehledné a ze začátku je plně dostatečné pro seznámení se s funkcemi serveru. Pro následné nasazení aplikace v provozu je vhodné využít příkazovou řádku [12].

Aplikační server Wildfly není pro projekt použitý z důvodu jiné konfigurace serveru. Nabízí jiné konfigurační soubory a nastavení, které bylo použité pro server Tomcat a nemohlo by být jednoduše přeneseno. Dalším důvodem je jeho velikost. Wildfly je více jak čtyřikrát větší než TomEE a obsahuje technologie, které v aplikaci využité nebudou.

2.4 Autentizace a autorizace

Zabezpečení aplikace je jeden z bodů, kterému je dobré věnovat větší pozornost při vývoji. Existuje mnoho frameworků, které se zabezpečením zabývají, ovšem výběr toho správného frameworku zavisí na požadavcích. Bude-li se jednat o interní podnikový systém, ke kterému budou přistupovat zaměstnanci z firemní sítě, je vhodné umožňit přihlašování pomocí protokolu *LDAP*,

z Windows přes službu *ActiveDirectory*, nebo přes jakékoliv jiné *SSO*. Ovšem může být umožněno i klasické přihlašování pomocí vytvořeného účtu z jména a hesla. Takto velký framework, který zvládne obsluhu několika služeb, je pro některé aplikace zbytečně složitý, a proto vznikly frameworky, které jsou o některé možnosti přihlašování omezené.

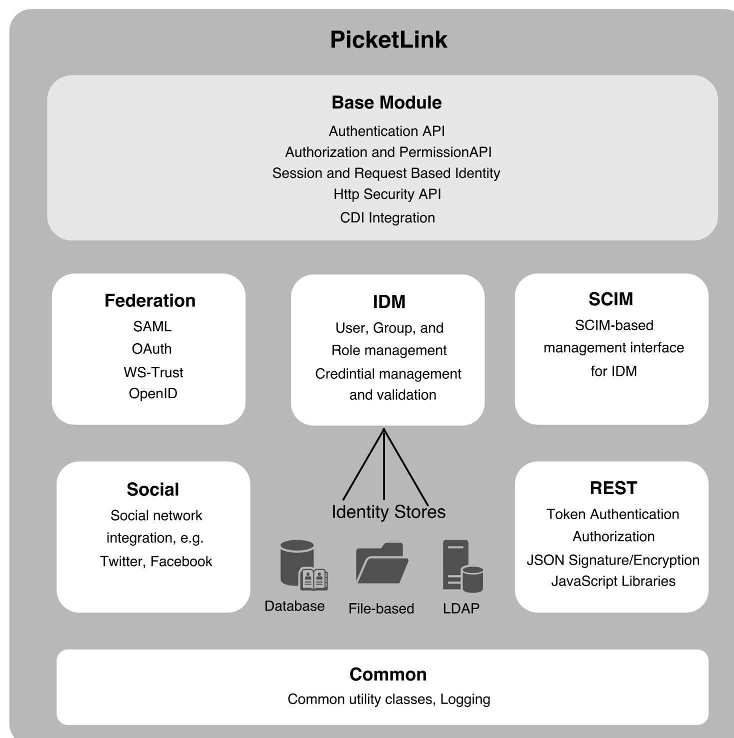
V předchozí verzi aplikace není autentizace uživatele ani autorizace přístupu k obsahu. Jedním z bodů mé práce bylo navrhnout, jak by mohla být tato funkčnost doplněna. Jelikož je celá aplikace přístupná pouze přes API, je jediná možnost přihlášení právě přes tuto službu. Dalším požadavkem byla autorizace obsahu. Uživatel bude mít přístup pouze k modelům sítě, ke kterým má dostatečná oprávnění.

2.4.1 PicketLink

PicketLink je framework určený pro zabezpečení Java EE aplikací. Jelikož je vyvíjený společností Red Hat, je doporučováno jej využívat současně s aplikačním serverem Wildfly.

Framework obsahuje nástroje pro autentizaci uživatelů, autorizaci obsahu a metod, správu uživatelů, práv, skupin, rolí a mnoho dalšího. Jednotlivé funkce jsou rozděleny do modulů (obr. 2.2). Základní modul, označovaný jako **Base Module**, obsahuje rozhraní a základní metody pro autentizaci a autorizaci. Tento modul by se dal označit jako jádro frameworku. K modulu se poté přidává další funkčnost, která už vychází z požadavků. Modul s názvem **IDM** slouží ke správě uživatelů, kontroluje práva a role, vytváří skupiny. Modul je možné rozšířit o **Identity Stores**. Jak už název napovídá, zajišťuje ukládání dat do databáze, LDAP serveru nebo do adresářové struktury [13].

I přesto že použití frameworku je celkem jednoduché, využívá standartní konfigurační soubory a je k dispozici velké množství oficiálních ukázek použití, stále se jedná o rozsáhlý projekt a jeho použití by bylo v naší aplikaci zbytečné.



Obrázek 2.2: Rozdělení frameworku do modulů[2].

2.4.2 Apache Shiro

Dalším bezpečnostím frameworkem pro Java aplikace je Apache Shiro. Rozdíl oproti např. PicketLinku je ten, že Shiro nevyužívá specifikaci Java EE, takže je možné aplikaci jednoduše využít ve webových, desktopových nebo mobilních aplikacích.

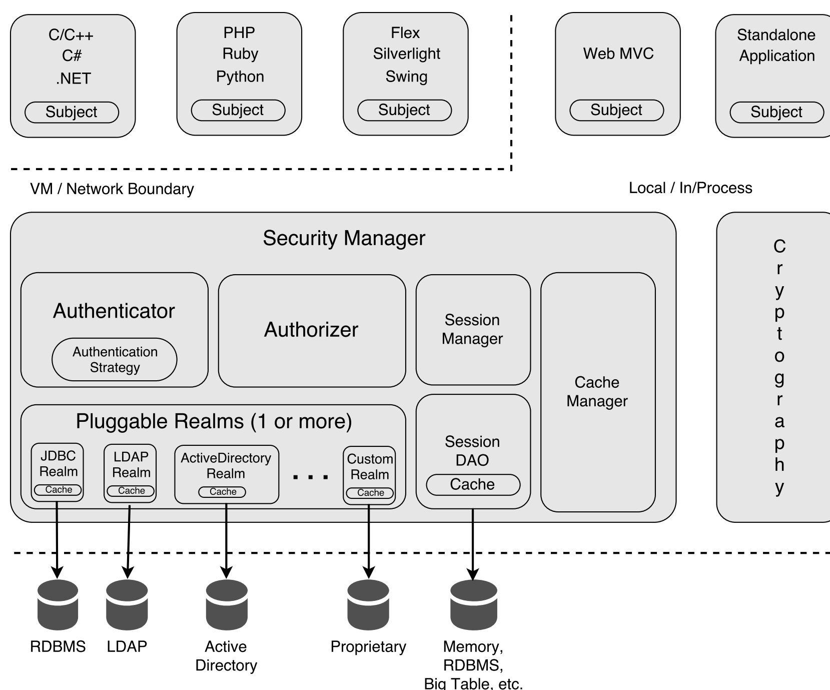
Architektura frameworku je rozdělena do třech částí (obr. 2.3). První je **Subject**. Jedná se o označení pro entitu, která může představovat uživatele nebo službu třetích stran. V našem případě se bude jednat pouze o uživatele. Pro získání uživatele je zapotřebí manažera s názvem **SecurityManager**. Ten zajišťuje všechny procesy, které se týkají autorizace a autentizace, spravuje také uživatelské sezení (v angličtině *session*), do které je možné uložit data, která souvisí s přihlášenou osobou. Poslední vrstva nese název **Realm**. Jedná se o pomyslný most mezi aplikací a zabezpečenými daty. Zabezpečená data mohou být uložena v databázi, na *LDAP* serveru nebo na jakýmkoliv jiném úložišti, ke kterému existuje plugin do Shira.

[2]Zdroj: `https://docs.jboss.org/picketlink/2/latest/reference/html/What_is_PicketLink.html`

Apache Shiro je vydáváno pod svobodnou licencí *Apache Licence* v aktuální verzi.

Tento framework také nebyl použitý v projektu, protože nenabízí vhodné řešení, které by usnadnilo implementaci. Větší část by musela být implementována a navržené řešení by muselo být změněné tak, aby odpovídalo použití frameworku.

Aplikace využívá jednoduchý způsob autorizace. Z tohoto důvodu nebylo potřeba žádný z výše uvedených frameworků. Při návrhu vlastního řešení jsem využil znalostí, které jsem získal při zkoumání obou frameworků a imlementoval řešení, které by bylo snadné na použití. Výhodu implementace vlastního řešení shledávám ve snadném zásahu při změně návrhu.



Obrázek 2.3: Detailní popis architektury frameworku[3].

3 Současné navržené řešení

Vývoj aplikace je rozdělen do několika fází. První fáze vývoje byla zaměřena více na uživatelské rozhraní. Byla definovaná funkčnost a vytvořeno grafické rozhraní, které umožňuje vizualizovat elektrickou sít a měnit některé vlastnosti jednotlivých prvků v síti. Aby mohla klientská část vzniknout, bylo zapotřebí vytvořit prototyp serveru, který by poskytoval a vypočítával data.

Ve druhé fázi vývoje se zaměřujeme na efektivnější server. Cílem je oddělit maticové výpočty od objektové reprezentace grafu. Výsledkem by měl být robustný server, který bude pracovat rychle a efektivně, umožňovat správu uživatelů, autorizovaný přístup na jednotlivé modely a vylepšovat rozhraní komunikace s vizualizací.

[3]Zdroj: `https://shiro.apache.org/architecture.html`

3.1 Architektura

Architektura aplikace je rozdělena do dvou částí - klientské a serverové. Výměna dat mezi nimi probíhá pomocí *REST API* (viz sekce 3.2.). Jednotlivé části aplikace jsou od sebe oddělené a vytvořené v různých programovacích jazycích. Výhodou řešení je, že uživatelské rozhraní není závislé na struktuře celé aplikace a může být vyvíjeno nezávisle na serveru.

REST API je jednoduché na implementaci a jednoznačně určuje služby na definovaných URL cestách. Uživatelská část nepotřebuje mnoho dotazů na server, současné navržené řešení je tedy pro takové využití vhodné.

3.1.1 Uživatelské rozhraní

Grafické uživatelské rozhraní je v programovacím jazyku JavaScript a využívá *Angular* framework. Aktuální verze umožňuje zobrazení vybraného modelu na interaktivní mapě, která je vytvořená pomocí knihovny *Openlayers*. U jednotlivých prvků sítě lze poté měnit nadefinované vlastnosti a po přepočítání dat vidět, jak změna ovlivnila celou síť.

Do budoucna je naplánováno, že přibude přihlašování uživatelů a jejich správa. Bude možné spravovat modely jednotlivých uživatelů a ukládání verzí. U vizualizované sítě bude možnost přidávat, odebírat prvky sítě nebo je dočasně vypnout.

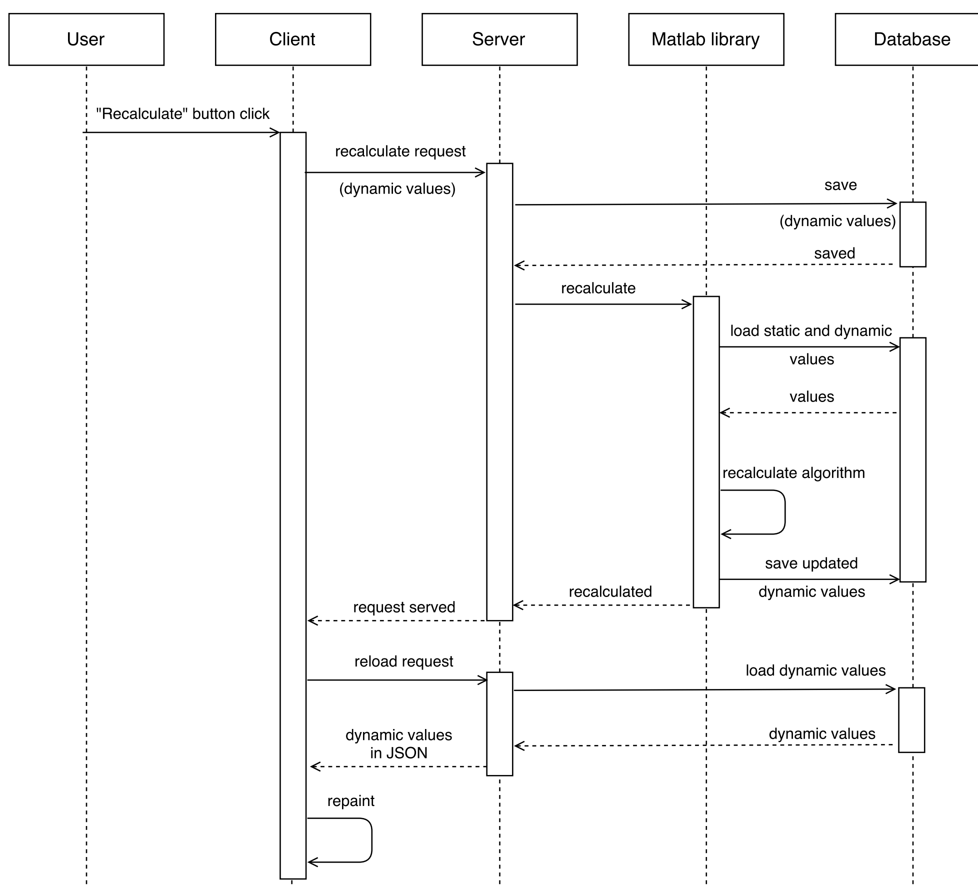
3.1.2 Server

Současná verze serveru je postavená na programovacím jazyku Java a frameworku Spring. Data jsou uložena v databázovém systému *MySQL* a výpočty se provádí v *Matlab* knihovně.

Architektura aplikace je navržena do více vrstev. Horní vrstvu tvoří **Kontrolery**, které naslouchají přístupům z vnější části aplikace na definovaných cestách. Výstupem z kontrolerů jsou data strukturovaná do formátu JSON (viz sekce 3.2.). Získaná data jsou zprcována v **Manažerech**. Ve třídách se v případě potřeby provádí úpravy nad daty a následně se převedou na tzv. DTO objekty. Poslední vrstvou modelu je vrstva přistupující k datům v databázi. Přístup je zajištěn pomocí DAO objektů, které obsahují metody skládající dotazy do databáze. Databázovou vrstvu spravuje ORM framework *Hibernate*. Jedná se o implementaci podléhající specifikaci *JPA*.

Pokud uživatel upraví vlastnosti některých prvků sítě, nastává proces přepočítání (obr. 3.1). Výpočty jsou prováděny pomocí *Matlab* knihovny.

Na diagramu je vidět, že po stisku tlačítka pro přepočítání se změněná data odešlou na server a následně uloží do databáze. Poté se předá požadavek Matlab knihovně, která si načte všechna aktuální data o modelu (statické a dynamické hodnoty) z databáze, sestaví strukturu grafu, vytvoří soustavu rovnic a nad nimi následně provede výpočty. Nově vypočítané hodnoty jsou uloženy do databáze a je předán požadavek serveru, aby si nová data načetl a odeslal zpět uživateli.



Obrázek 3.1: Proces přepočítání.

3.2 Popis API

Data jsou publikovaná na URL, na kterých server naslouchá, ve formátu *JSON*. Jsou rozdělena na statická a dynamická. Statická data představují

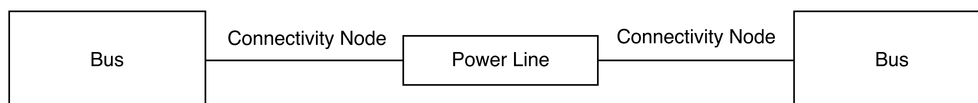
pevné informace o modelu, které nelze měnit. Dynamická data, jak už název napovídá, jsou proměnná data. Jedná se o hodnoty, které lze uživatelsky měnit a mezi tyto hodnoty patří i data, které se přepočítávají.

Na zobrazený model je možné pohlížet ze dvou pohledů a jsou tomu uzpůsobené i cesty pro data. Model s méně detaily je nazýván **highLevel** a podrobnější pohled je **lowLevel**. Rozdělení je zde udělané proto, aby se zredukovalo množství přenášených dat. Data získaná dotazy na highLevel obsahují informace o tom, aby mohl být model vykreslen do grafu a zobrazený na mapě. Při určitém přiblížení se poté načtou data pro lowLevel.

Podrobnější informace o API jsou uvedené v přiloženém souboru s názvem *dokumentace_api.yaml*, který lze zobrazit ve Swagger nebo Apiary editoru. V souboru je detailně popsán každý dotaz na server včetně parametrů a jednotlivých atributů.

3.2.1 Grafová data

Vizualizovaný graf se skládá z uzlů a hran. Uzly představují aktivní prvky, u kterých lze měnit vlastnosti a hrany propojení těchto prvků. Na první pohled by se mohlo zdát, že elektrické vedení je brané jako hrana mezi uzly, ale není to tak. Elektrické vedení je reprezentováno také jako uzel a nazývá se **powerLine**. Hrany grafu představují prvky s názevm **connectivityNode** (obr. 3.2).



Obrázek 3.2: Připojení dvou elektrických zařízení a elektrického vedení.

Graf zobrazíme dotazem na highLevel. Výsledkem dotazu jsou informace o sběrnicích - **buses**, které v přenosové síti značí hlavní typ uzlu, na který se připojují ostatní prvky, elektrickém vedení, které propojuje dvě sběrnice a mají pevně dány vlastnosti a propojovací hrany.

Na highLevel můžeme pokládat dva typy dotazů. Oba dotazy vrací data pro stejné prvky, ale zatímco dotaz na **simple** vrátí základní údaje, které jsou potřebné pro správné vykreslení grafu, **detail** vrátí úplné informace.

Příklad níže ukazuje (ukázka 3.1), jak může vypadat dotaz na highLevel **simple**. Sběrnice obsahuje název, GPS souřadnice, úroveň napětí, popis a připojené hrany. Připojené hrany se značí **connectivities** a každý prvek v síti je obsahuje. Obsahují počáteční a koncový uzel hrany. U elektrického vedení se nachází úroveň přenášeného napětí, popis a připojené hrany. Parametr **idModel** určuje na jaký model se dotazujeme a úroveň napětí zvolíme parametrem **voltageLevel**.

```
Cesta: /vps-devel/api/graph/highLevel/simple?idModel=1&voltageLevel=400
Výsledek:
{
  "buses"            : [{
    connectivities         : [{
            idBaseObject        : "12637"       // identifikátor počátačního prvku
            idConnectivity      : "153"         // identifikátor hrany
            idConnectivityNode  : "13190"       // identifikátor konečného prvku
    }]
    descriptionVoltageLevel : "U400V"           // textový popis úrovně napětí
    gpsLatitude            : 49.85069982335782 // zeměpisná šířka
    gpsLongitude           : 11.82164342679259 // zeměpisná délka
    idVoltageLevel         : 400               // identifikátor úrovně napětí
    mRid                   : "12637"           // jednoznačný identifikátor prvku
    name                   : "Bus: 1101"       // jméno směrnice
  }]
  "connectivityNodes" : [{
    connectivities         : []                // pole hran
    mRid  : "13190"                            // jednoznačný identifikátor prvku
  }]
  "powerLines"       : [{
    connectivities         : []                // pole hran
    descriptionVoltageLevel : "U400V"           // textový popis úrovně napětí
    idVoltageLevel         : 400               // identifikátor úrovně napětí
    mRid                   : "13271"           // jednoznačný identifikátor prvku
    name                   : ""                // jméno elektrického vedení
  }]
}
```

Ukázka. 3.1: Dotaz na méně podrobný pohled (highLevel).

Dotazem končícím na **/highLevel/detail** získáme detailní informace o grafu. Struktura výsledku a parametry jsou stejné jako u dotazu na **simple**. Dodatečné informace se zobrazují až při kliknutí na prvek v síti, není tedy nutné, aby byly načteny současně při vykreslování vizualizované sítě.

Pokud bude počet prvků dostatečně malý, bude proveden dotaz na detailnější pohled grafu. Podrobnější data získáme dotazem na **lowLevel**. Stejně jako u highLevelu se dotazem na **simple** (ukázka 3.2) načtou data nutná pro vykreslení uzlů a dotazem na **detail** načteme podrobnější informace.

Detailnější pohled zobrazuje generátory - **generatingUnits**, které představují zdroj energie a na sběrnici jich může být připojeno několik. U generátoru lze měnit vlastnosti. Zátěže - **loads**, které spotřebovávají energii. Jejich vlastnosti jsou pevně dány, přepínače - **switches** a elektrické vedení - **powerLines**. Význam jednotlivých atributů u prvků je stejný jako u dat z dotazu na **/highLevel/simple**.

Podrobnější informace jsou taktéž stejné jako u highLevelu a dotazujeme se přes **detail**. Výsledkem dotazu jsou data, která mají stejnou strukturu dat jako u **lowLevel/simple**, ovšem s podrobnějšími informacemi.

```
Cesta: /vps-devel/api/graph/lowLevel/simple?idModel=2&voltageLevel=400000
Výsledek:
{
  "generatingUnits" : [{
      connectivities          : []
      descriptionVoltageLevel : "U400kV"
      gpsLatitude             : 49.124244
      gpsLongitude            : 16.124513
      idVoltageLevel          : 400000
      mRid                    : "4686295088136782643"
      name                    : "CEDAL_12:gen"
  }]
  "loads"           : [{
      connectivities          : []
      descriptionVoltageLevel : "U400kV"
      gpsLatitude             : 50.029144
      gpsLongitude            : 15.453515
      idVoltageLevel          : 400000
      mRid                    : "4686295088136782637"
      name                    : "CECHV_12:load"
  }]
  "powerLines"      : [{
      connectivities          : []
      descriptionVoltageLevel : "nonspec"
      gpsLatitude             : 50.635664
      gpsLongitude            : 14.456551
      idVoltageLevel          : 0
      mRid                    : "4686295088136781913"
      name                    : "CBAB__1_CBAB__5_1"
  }]
  "switches"        : [{
      connectivities          : []
      descriptionVoltageLevel : "U400kV"
      gpsLatitude             : 49.784776
      gpsLongitude            : 18.508337
      idVoltageLevel          : 400000
      mRid                    : "4686295088136781879"
      name                    : "CALB__1_CALB__1_5:1:swt"
  }]
}
```

Ukázka. 3.2: Dotaz na podrobnější pohled (lowLevel).

Při lowLevel pohledu je k dispozici také dotazování přes **aux** (ukázka 3.3). Výsledkem jsou dodatečné informace. Konkrétně se jedná o transformátory - **powerTransformers**. Na sběrnici jich může být několik a jejich parametry lze měnit. Posílají se také informace o přepínačích. Jedná se o stejná data, která získáme simple dotazem.

```
Cesta: /vps-devel/api/graph/lowLevel/aux?idModel=3&voltageLevel=400000
Výsledek:
[{
  "powerTransformers" : [{
      connectivities          : []
      descriptionVoltageLevel : "nonspec"
      gpsLatitude             : 50.260576
      gpsLongitude            : 5.0550398
      idVoltageLevel          : 0
      mRid                    : "4686295105780449352"
      name                    : "BACHEN6_BACHEN1_1"
  }]
  "switches"          : [{
      connectivities          : []
      descriptionVoltageLevel : "U400kV"
      gpsLatitude             : 50.260576
      gpsLongitude            : 5.0550398
      idVoltageLevel          : 400000
      mRid                    : "4686295105780449338"
      name                    : "BACHEN1_BACHEN1_1:1:swt"
  }]
}]
```

Ukázka. 3.3: Dotaz na dodatečná data pro detailnější pohled.

Každý model může mít různé hodnoty napětí. Požadavek na tuto cestu vrací všechny možné úrovně napětí, které jsou přepínatelné v uživatelském rozhraní (ukázka 3.4).

```
Cesta: /vps-devel/api/graph/voltageLevels?idModel=1
Výsledek:
[{
  "idVoltageLevel" : 400      // identifikátor úrovně napětí
  "description"    : "U400V"  // textový popis úrovně napětí
}]
```

Ukázka. 3.4: Dotaz pro výpis všech úrovní napětí daného modelu.

Cesty končící **pictureInfo** a **picture** spolu vzájemně souvisí (ukázka 3.5). Zatímco dotaz na **pictureInfo** vyhledá informace o obrázku a sestaví cestu pro jeho stažení, cesta končící **picture** obrázek fyzicky stáhne. Parametry

dotazu lze ovlivnit číslem změny (parametr **idChangeBatch**) a parametrem **mRid** určujeme, na jaký prvek se dotazujeme.

```
Cesta: /vps-devel/api/graph/pictureInfo?idChangeBatch=1&mRid=12673
Výsledek:
{
  "path"   : "/api/graph/picture?mRid=14" // cesta pro obrázek k prvku s mRid 14
  "xScale" : 1                            // zvětšení na ose X
  "yScale" : 1                            // zvětšení na ose Y
}

Cesta: /vps-devel/api/graph/picture?mRid=14
Výsledek: obrázek zakódovaný v Base64
```

Ukázka. 3.5: Dotaz pro získání informací o obrázku a stažení obrázku.

3.2.2 Modely

Při vstupu na hlavní stránku se zobrazí přehled všech dostupných modelů. Dotaz vrací seznam všech modelů, které jsou aktuálně dostupné (ukázka 3.6).

```
Cesta: /vps-devel/api/model/getAll
Výsledek:
[{
  createdOn : 1478475123000 // datum v milisekundách
  description : null        // popis grafu
  idModel : "1"             // identifikátor modelu
  name : "Model 1"          // název modelu
}]
```

Ukázka. 3.6: Dotaz pro výpis všech modelů.

3.2.3 Přepočítaná data

U prvků sítě je možné měnit vlastnosti a sledovat změny pro celou síť. Aby se tak stalo, musí se data přepočítat (dotaz **recalculate**). Na server se pošle dotaz, který obsahuje seznam změn. Každá změněná vlastnost má svůj identifikátor, maximální a minimální hodnotu, základní a vypočítanou hodnotu. Druhý dotaz (dotaz **changes/{changeBatch}**) vrací opět stejný seznam změněných vlastností, ale s již už vypočítanými hodnotami (ukázka 3.7).

```
Cesta: /vps-devel/api/recalculate
Cesta: /vps-devel/api/data/changes/{changeBatch}
Tělo nebo výsledek:
[{
    "valueId"         : 18,   // identifikátor vlastnosti
    "value"           : 666,  // změněná hodnota
    "minValue"        : 0,    // minimální nastavitelná hodnota
    "maxValue"        : 0,    // maximální nastavitelná hodnota
    "enumerateValue"  : 0     // vypočítaná hodnota
}]
```

Ukázka. 3.7: Dotaz pro přepočítání dat a dotaz pro získání dat dané změny.

Dotaz vrací výčet všech vlastností a identifikátorů, které se zobrazují u jednotlivých prvků sítě (ukázka 3.8).

```
Cesta: /vps-devel/api/data/enumerateValues
Výsledek:
{
  "variables"  : [{
    description : ""               // popis proměnné
    idVariable  : 4                // identifikátor proměnné
    name        : "ReactivePower"  // název proměnné
  }]
  "valueTypes" : [{
    description : ""             // popis typu hodnoty
    idValueType : 4              // identifikátor typu hodnoty
    name        : "limit"        // jméno typu hodnoty
  }]
  "enumerates" : [{
    description : ""             // popis vlastnosti
    idEnumerate : 6              // identifikátor vlasnosti
    idVariable  : 1003           // identifikátor proměnné
    name        : "U3kV"         // jméno vlasnosti
    originalId  : 3000           //
    value       : 3000           // hodnota
  }]
}
```

Ukázka. 3.8: Dotaz pro získání všech vlastností a identifikátorů.

Dotaz slouží pro získání aktuálních vypočítaných dat všech vlastností prvků sítě (ukázka 3.9). Výsledkem jsou data, která obsahují aktuální hodnotu, rozmezí možných hodnot, identifikátor vlastnosti a další zobrazované hodnoty.

```
Cesta: /vps-devel/api/data/values/3
Výsledek:
[{
    description     : ""                    // popis vlastnosti
    enumerateValue  : null                  // vypočítaná hodnota
    idValue         : 11701                 // identifikátor vlastnosti
    mRID            : "4686295105780449389" // jednoznačný identifikátor
    maxValue        : 162000000             // maximální hodnota
    minValue        : 50000000              // minimální hodnota
    name            : ""                    // jméno vlastnosti
    temporary       : 0                     // dočasná vlasnost
    value           : 94800003              // aktuální hodnota
    valueType       : 1                     // identifikátor typu hodnoty
    variable        : 3                     // identifikátor typu proměné
}]
```

Ukázka. 3.9: Dotaz pro získání všech vlastností sítě.

Na úvodní obrazovce je možné vybrat nejen model, ale také metodu provádění výpočtů. V matlab knihovně jsou implementovány různé metody výpočtů. Po vybrání modelu a metody výpočtu se informace uloží do uživatelského sezení (ukázka 3.10). Vyvolání dotazu způsobí uložení modelu s id **2** a výpočty se budou provádět metodou s id **1**.

```
Cesta: /vps-devel/api/data/model/2/1
```

Ukázka. 3.10: Dotaz pro uložení identifikátoru a metody výpočtu.

3.2.4 Navržené změny

V nové verzi serveru bude obsah přístupný až po přihlášení. Přihlášení se bude provádět přes token v hlavičce dotazem na autentizační cestu (ukázka 3.11). Služba bude vracet autorizační token a výsledek přihlášení.

```
Cesta: /api/auth/apitoken/login
Výsledek:
{
  "result": "SUCCESS",                    // výsledek přihlášení
  "message": "User admin has been logged.", // textová hláška popisující výsledek
  "auth-token": <token>                   // token pro autorizaci
}
```

Ukázka. 3.11: Dotaz pro přihlášení do aplikace.

Zavoláním služby pro odhlášení dojde k odhlášení uživatele ze serveru a smazání jeho sezení (ukázka 3.12).

```
Cesta: /api/auth/apitoken/logout
Výsledek:
{
  "result": "SUCCESS",                          // výsledek odhlášení
  "message": "User admin has been logged out." // textová hláška popisující výsledek
}
```

Ukázka. 3.12: Dotaz pro odhlášení z aplikace.

Kvůli redukci dat přibyla možnost požádat si o kompletní seznam všech číselníků, které jsou v aplikaci k dispozici (ukázka 3.13). V samotných datech se již jen odkazuje na číselné hodnoty. Tím se minimalizuje množství přenášených dat, protože není potřeba přenášet celý textový řetězec enumu. Výsledek se generuje automaticky, a to tak, že se ze všech dostupných číselníků složí slovník, kde klíčem je název číselníku a hodnoty představují pole atributů. Ze slovníků jsou poté vygenerovaná data v JSON formátu.

```
Cesta: /api/enum/model
Výsledek:
{
  "coordinatesType"     : [{
    "name"  : "JTSK", // název souřadnice
    "value" : "2"     // identifkátor
  }
  "connectStatus"       : []
  "phaseTapChangerType" : []
  "windingType"         : []
  "windingConnection"   : []
  "busType"             : []
}

Cesta: /api/enum/user
Výsledek:
{
  "role"        : [{
      "name"  : "ADMIN"   // název role
  }, ...]
  "permission"  : [{
      "name"  : "READ",   // název oprávnění
      "value" : "4"       // číselná hodnota oprávnění
  }, ...]
}
```

Ukázka. 3.13: Dotaz pro výpis všech čísellníků patřící k modelu.

Jedná se o úpravy původní cesty -*/vps-devel/api/model/getAll*. Změna je zde proto, aby cesta měla stejnou logiku jako nové cesty, které jsou podmíněné autorizací. Struktura výsledných dat zůstává zachovaná (ukázka 3.15).

```
Cesta: /api/model/all
```

Ukázka. 3.14: Úprava cesty pro výpis všech modelů.

Zde se jedná o úpravu přístupových cest (ukázka 3.15). Pomocí těchto nových cest je možné jednoduše provádět jednotně autorizaci. Výsledná data zůstávají nezměněná.

```
Cesta: /api/model/{modelID}/*
   /highLevel/simple
   /lowLevel/simple
   /highLevel/detail
   /lowLevel/detail
   /voltageLevels
```

Ukázka. 3.15: Úprava cest pro přístup ke grafu.

Pokud načítaná síť bude nabývat velkých rozměrů, budou i přenášená data velká. Z tohoto důvodu přibyla možnost pracovat pouze s jedním prvkem v síti přes jednoznačný identifikátor (ukázka cest 3.16). Cesta dotazu je rozdělena podle toho o jaký typ prvku se jedná a následuje operace, kterou nad prvkem chceme provádět. Parametrem všech dotazů (kromě **insert**) je identifikátor prvku.

```
Cesta:
  /api/model/{modelID}/bus/*
  /api/model/{modelID}/load/*
  /api/model/{modelID}/generatingUnit/*
  /api/model/{modelID}/powerLine/*
  /api/model/{modelID}/powerTransformer/*
  /api/model/{modelID}/switch/*

      /info
      /update
      /delete
      /insert
```

Ukázka. 3.16: Cesty pro práci s jedním prvkem v síti.

API je rozšířené také o práci s uživatelem. Správu uživatelů může provádět pouze uživatel, který má právo ADMIN. Každý požadavek je kontrolován a pokud není uživatel administrátor, je mu přístup zamítnut. Ukázka 3.17 ukazuje, jak zobrazit informace o konkrétním uživateli. Výsledek požadavku

na druhé cestě obsahuje stejnou strukturu dat, ovšem je dotázána na informace o sobě sama. Tento požadavek je jediný, který lze provést bez administrátorského přístupu.

```
Cesta: /api/user/info?userID=1
Cesta: /api/user/current
Výsledek:
{
  "id": 751,              // identifikátor uživatele
  "username": "admin",    // přihlašovací jméno
  "hash": <hash>,         // otisk hesla
  "firstName": "",        // jméno
  "lastName": "",         // příjmení
  "allowed": true,        // povolení v systému
  "email": "",            // email
  "role": "ADMIN",        // přidělená role
  "models": [             // pole modelů s oprávněním
    {
      "modelID": 151,     // identifikátor modelu
      "name": "Model 1",  // název modelu
      "permissions": [    // pole práv k modelu
        "READ",
        "WRITE"
      ]
    }
  ]
}
```

Ukázka. 3.17: Dotaz pro získání informací o konkrétním a aktuálně přihlášeném uživateli.

Zobrazení všech uživatelů je možné dotázáním se na cestu v ukázace 3.18. Výsledkem je pole o stejné struktuře dat, jako při dotazu na konkrétního uživatele.

```
Cesta: /api/user/all
```

Ukázka. 3.18: Dotaz pro výpis všech uživatelů.

Přidání, odebrání a aktualizace uživatele je možná přes dotazy v ukázace 3.19. Výsledkem všech těchto dotazů je informace o úspěchu či neúspěchu akce. U dotazů na přidání a aktualizaci dat, se data vkládají do těla dotazu a obsahují stejnou strukturu jako při dotazu na informace o uživateli. Při dotazu na smazání se v parametru dotazu musí nacházet parametr **userID**, který určuje, jaký uživatel bude smazán.

```
Cesta: /api/user/*
          /insert
          /update
          /delete

Výsledek:
{
  "result":  string,   // výsledek
  "message": string    // textová hláška popisující výsledek
}
```

Ukázka. 3.19: Dotaz pro výpis všech uživatelů.

Poslední možností, která lze u uživatele měnit, je přidávání nebo odebírání oprávnění na jednotlivé modely (ukázka 3.20). Dotaz musí obsahovat parametr **userID**, který určuje, jakému uživateli budou práva změněna a v těle dotazu se nachází data ve formátu JSON.

```
Cesta: /api/user/changeModelPermission?userID=<userid>
Tělo:
[{
  "modelID": 151,     // identifikátor modelu
  "name": "Model 1",  // název modelu
  "permissions": [    // pole práv k modelu
    "READ"
  ]
}]
Výsledek:
{
  "result":  string,   // výsledek
  "message": string    // textová hláška popisující výsledek
}
```

Ukázka. 3.20: Dotaz pro změnu oprávnění na model.

4 Popis implementace

Cílem mé práce bylo navrhnout a implementovat server, který bude poskytovat služby dostupné přes API. Práce měla navázat na prototyp serveru, který vznikl v první fázi projektu a rozšířit funkčnost. Původní server využíval technologie a knihovny z frameworku Spring, který aplikaci výrazně zvětšoval. Jedním z požadavků bylo vytvořit aplikaci, která bude obsahovat jen nejnutnější knihovny. Nová verze serveru tedy nevychází z původní verze, ale je navržená od základu a využívá jiné knihovny a technologie.

4.1 Navržené řešení aplikace

Aplikace je postavená na technologiích specifikace Java EE a logika je rozdělena do vícevrstvé architektury. Správu závislostí a sestavování zajišťuje program **Maven**. V konfiguračním souboru *pom.xml* jsou definované základní vlastnosti aplikace - verze Javy, název a verze projektu. Jsou zde uvedené všechny dodatečné knihovny a profily - **production** a **development**.

4.1.1 Server a databáze

Aplikace běží na aplikačním serveru TomEE. Server již v sobě obsahuje většinu potřebných knihoven, proto tedy aplikace může využívat pouze API specifikace. Z toho plyne, že přeložená aplikace nabývá měnších rozměrů a s mírnou změnou konfigurace (viz sekce 6.) může být spuštěna na jiném aplikačním serveru.

Databázová vrstva využívá technologii JPA. V aplikačním serveru se nachází implementace **OpenJPA**, která pokrývá všechny požadavky databázového přístupu. Konfigurace databáze se nachází v konfiguračním souboru *persistence.xml*. V tomto souboru je odkaz na identifikátor datového zdroje, je zde vypnuté automatické skenování doménových tříd a zapnutá funkce automatického vytváření tabulek v databázi.

Připojení do databáze je pod správou aplikačního serveru. TomEE nabízí konfigurační soubor *tomee.xml*, kde jsou definované jednotlivé datové zdroje. Pro data byl zvolen databový systém MySQL. Požadavek na tento databázový systém vychází z požadavků KKY. Obsahuje dvě databáze - **users** a **models**. V databázi models jsou uloženy informace o modelech, celé

topologii sítě a identifikátory uživatelů s nastavenými právy k modelu. V databázi users se nachází informace o uživatelích a základní informace o modelu s nastavenými právy. Rozdělení dat na dvě databáze je zde důležité pro zachování uživatelských oprávnění při smazání databáze s modely. Nastavená práva na modely mohou být obnovena z uživatelské databáze.

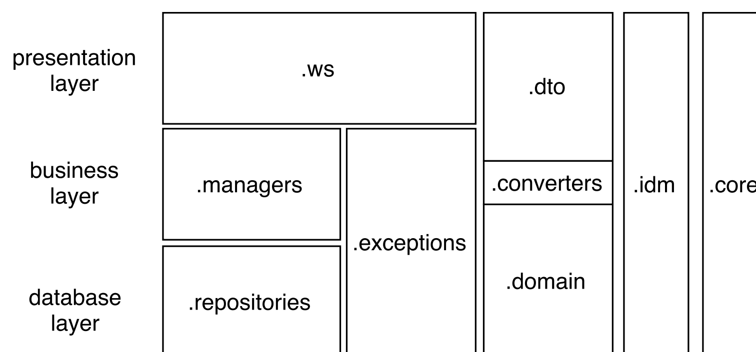
4.1.2 Implementace API

Služby dostupné přes API vycházejí z návrhu původního serveru a mírně je rozšiřují. Cesty původního API byly změněny, ovšem struktura dat zůstala zachována. Aplikace využívá rozhraní pro práci s RESTFull webovými službami. V aplikačním serveru je obsažena knihovna **Johnzon**, která implementuje technologii **JSR 353**. Tato technologie umožňuje generování z objektového návrhu JSON data [14]. Knihovna Johnzon také obsahuje funkčnost technologie **JAX-RS**. Knihovna **Apache CXF** zajišťuje implementaci této technologie [15]. Vytvoříme-li dotaz na předem definovanou cestu, tyto knihovny umožňí zavolání obslužné metody.

V aplikaci existují dotazy typu GET a POST a dvě metody přístupu - autorizovaný a neautorizovaný. Povolené typy metod jsou definované ve třídě **CORSFilter**. Neautorizovaný přístup je umožněn pouze dotazům pro přihlášení do aplikace (autentizace). Ostatní dotazy podléhají restrikcím (autorizace). Celý proces autentizace od vytvoření požadavku z klientské aplikace po dokončení požadavku je popsaný v sekci 4.2.1. Jelikož jsou všechny ostatní požadavky podmíněné autorizací, je putování požadavků ostatních dotazů velice podobné a liší se pouze v manažerech, kde je vykonána logika požadavku. Proces je popsán v sekci 4.2.2.

4.1.3 Struktura balíků

Struktura aplikace je logicky rozdělena do balíků (obr. 4.1). Všechny balíky začnají názvem **cz.zcu.kiv.vps**.



Obrázek 4.1: Zobrazení balíků ve vrstvách.

- **.domain** - v balíku se nachází všechny doménové objekty využívané v grafové části. Tyto třídy využívají databázi *models* a jsou vygenerované programem z relačního databázového modelu navržený KKY.
- **.dto** - balík obsahuje kopii doménových objektů. Tyto objekty se používají při přenosu přes síť a mohou obsahovat jinou strukturu. Např. některé hodnoty mohou být skryty nebo být reprezentovány v jiném formátu.
- **.database** - v balíku jsou třídy obsahující logiku při volbě databáze.
- **.converters** - nachází se zde třída **ModelConverter**, která obaluje knihovnu **ModelMapper** pro převod doménového objektu na DTO. Knihovna je nastavena tak, aby převod byl prováděn přes reflexi. V konfiguraci jsou poté přidané výjimky např. číselníky se v DTO převádí na číslicovou hodnotou, v doménových objektech je enum hodnota.
- **.core** - tento balík obsahuje třídy, které nastavují parametry aplikace nebo se využívají napříč celou aplikací.
  - Třída **Application** definuje jaký bude počátek cesty pro služby dostupné přes API.
  - Třída **ApplicationConfig** načítá konfigurační soubor s názvem **config.properties**, který obsahuje počáteční konfiguraci. Konfigurační soubor obsahuje přihlašovací údaje pro systémový účet, základní locale a názvy hlaviček a cookies.
  - Třída **JacksonConfig** nastavuje vzhled odchozích dat ve formátu JSON. V aplikaci se využívá knihovna **Jackson**, která dovoluje definovat vzhled odchozích dat na jediném místě.

  - Třída **Messages** načítá soubor s uživatelskými hlášky podle aktuálního locale. Správné načtení lokalizovaného souboru je dané nastavením locale v konfiguračním souboru.

- **.exceptions** - obsahuje vlastní aplikační výjimky. Vlastní výjimky se využívají pro přehlednost kódu a snažší obsluhu manažerů a repozitářů. Výjimky jsou použité v situacích, kdy na určitou událost nelze reagovat jinak než chybovým stavem a následná chybová hláška je propagovaná uživateli.

- **.idm** - **Identify Model** - balík obsahuje logiku autentizace a autorizace. Obsahuje vlastní doménové objekty, výjimky a nástroje pro přehlednost kódu.

- **.managers** - balík obsahující rozhraní a implementaci obchodní logiky aplikace. Třídy jsou volány z kontrolerů nebo z jiného manažeru a využívají služeb databázové vrstvy přístupem přes repozitáře.

- **.repositories** - třídy spravující dotazy pro přístup do databáze. Třídy využívají API perzistentní vrstvy.

- **.utils** - statické třídy zpřehledňující kód a jsou využívány pro opakující se části kódu.

- **.ws** - **Web Services** - třídy obsluhující webové služby. V tomto balíku se nachází další podbalíky:

  - **.annotations** - vlastní anotace vytvořené za účelem zpřehlednění kódu a rozšíření funkčnosti.

    * anotace **AuthorizationPath** - definuje počátek cesty pro dotaz na autorizaci. Anotaci lze použít pouze nad třídou.

    * anotace **ModelIdentification** - označuje v metodě parametr, který představuje identifikátor modelu. Anotaci lze použít pouze pro kontrolery, které pracují s modelem.

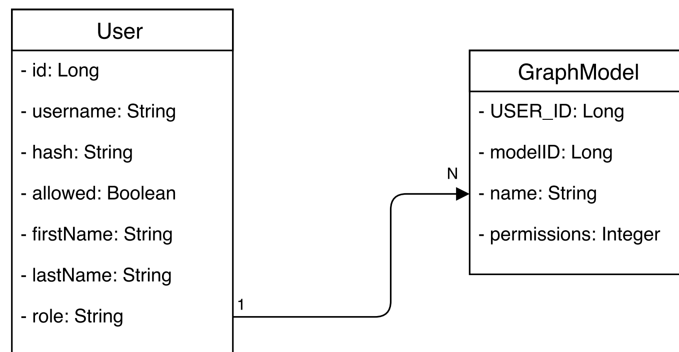
  - **.controllers** - rozhraní a implementace kontrolerů naslouchající na API cestách.

  - **.enums** - pomocné enumy zpřehledňující kód.

  - **.filters** - implementace servletových filtrů filtrující požadavky na server.

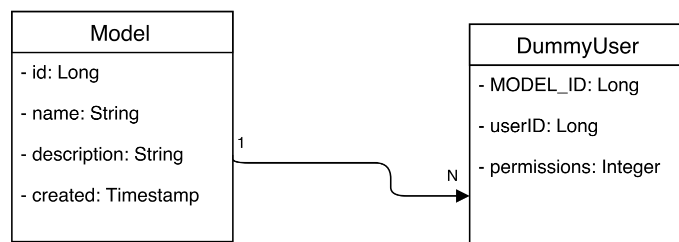
4.2 Identity model

Balík z názvem IDM obsahuje logiku pro správu uživatele - autorizace, autentizace, přidávání, odebírání a změnu práv uživatele. Uživatel je reprezentován třídou **User**, která obsahuje uživatelské jméno, otisk hesla, jméno, příjmení, email, povolení v systému, roli a seznam modelů, ke kterým má právo (obr. 4.2). Jelikož jsou uživatelská data a data o modelech uložené v různých databázích, práva přidělená uživateli na jednotlivé modely se uchovávají v obou databázích.



Obrázek 4.2: Třída User s relací na model.

V databázi s modely je situace opačná. K modelu jsou přiřazeny uživatelé, kteří mají na daný model právo. Třída **Model** obsahuje uživatele s nastavenými právy (obr. 4.3). Tento návrh byl zvolen kvůli jednodušší práce při načítání modelů, které náleží uživateli.



Obrázek 4.3: Třída Model s relací na uživatele.

Aby byly uživatelská oprávnění v obou databázích v konzistentním stavu, propisují se změny do obou databází současně. Proces slučování probíhá tak, že se z databáze načtou všechny modely, ke kterým má uživatel aktuálně právo a modely, u kterých je změna. Tyto dva seznamy se sloučí a nad každým modelem se provede akce (ukázka 4.1). U všech změněných modelů dojde k uložení do databáze.

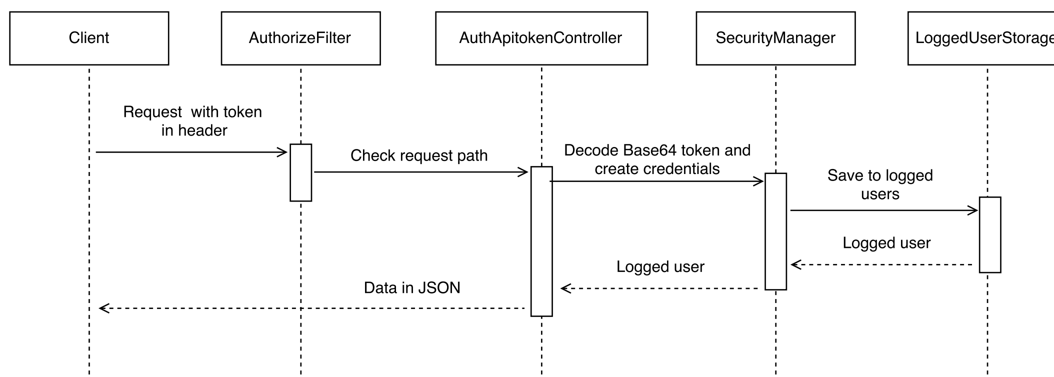
```
loop -> modely s aktuálními právy + modely s novými právy
    if model je v seznamu změn
        if uživatel již nějaké oprávnění obsahuje
            změna oprávnění na nové
        else
            přidání oprávnění
    else
        odebrání oprávnění
```

Ukázka. 4.1: Pseudo kód popisující proces slučování.

4.2.1 Autentizace

Autentizace je proces ověření pravosti přihlašovacích údajů (anglicky credentials). Přihlašovací údaje se skládají z jména a hesla. Z důvodu bezpečnosti se heslo nepřenáší jako prostý text, ale je vytvořen otisk hašovací funkcí **SHA256**. Uživatelské jméno a hash poté slouží k ověření identity.

Proces autentizace je znázorněn na diagramu (obr. 4.4) a vysvětlen níže.



Obrázek 4.4: Diagram procesu autorizace.

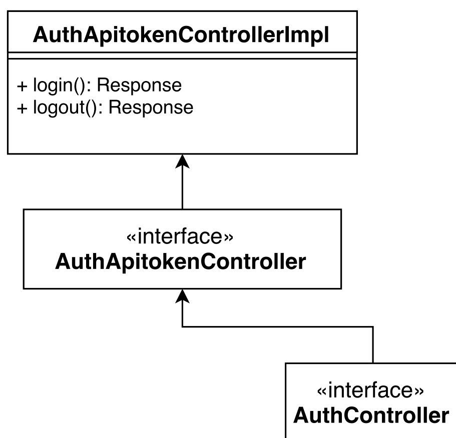
- Klient vytvoří dotaz na URL cestu pro autentizaci. V hlavičce uvedeme **Base-auth** token. Token je zakódován kódováním **Base64** a má následující strukturu: **username:hash**. Pokud nebude token uveden, bude klient upozorněn příslušnou hláškou.
- Požadavek přijme **AuthorizeFilter**, který podle zadané cesty rozhodne, zda se jedná o požadavek na autentizaci nebo autorizaci.
- **AuthApitokenController** načte autentizační token z hlavičky dotazu a z cookie **SID**. SID slouží jako jednoznačný identifikátor pro uživatelské sezení. Identifikátor je zde důležitý pro to, aby mohl být uživatel

přihlášen z více zařízeních. Tímto se zamezí opětovnému příhlášení ze stejného sezení. Údaje pro příhlášení a odhlášení se zabalí do objektu **UsernamePasswordCredentials**, kde jako parametry jsou token s SID.

- **SecurityManager** ověří správnost přihlašovacích údajů, vytvoří autorizační token a datum platnosti. Autorizační token je hash vytvořen funkcí **SHA256** z příhlašovacího jméno (userName) a SID. K uživateli se vytvoří **Session** objekt, který nese dočasné informace o uživatelském sezení a pomocí autorizačního tokenu je uživatel a sezení uloženo mezi přihlášené uživatele jako objekt **LoggedUser**. Platnost tokenu je nastavena na 20 minut a při každém dotazu na server je automaticky prodloužen.

- V odpovědi pro klienta je do hlavičky přidán token a data obsahují token a výsledek přihlášení. Pokud by byla autentizace neúspěšná, je klientovi poslána příslušná hláška.

Cesta pro autentizaci je **/api/auth/apitoken/** a je navržená tak, aby jí bylo možné rozšířit (obr. 4.5). Nejspodnějším kontrolerem je **AuthController**, který značí, že se jedná o autentizaci. Obsahuje anotaci **AuthorizationPath**, která definuje jak bude cesta začínat (v tonto konkrétním případě */auth/*). Tento kontroler dědí **AuthApitokenController**, který přidává do cesty */apitoken/* a implementace kontroleru provádí logiku přihlašování a odhlašování (viz výše) přes token.



Obrázek 4.5: UML diagram autorizace.

4.2.2 Autorizace

Autorizace je proces, kdy uživatel přistupuje k datům, ke kterým je potřeba oprávnění. Oprávnění je možné získat rolí nebo právy pro daný model.

Každý vytvořený uživatel musí mít definovanou roli z enumu **Role**. Role jsou definované dvě - *ADMIN* a *CUSTOMER*. Uživatel s rolí administrátor má přístup ke všem datům a má přístup i do nastavení uživatelů. Uživatelský přístup podmíněný rolemi je možný přidáním anotace **RequiresRoles** (ukázka 4.2). Lze ji přidat na metodu i třídu. Pokud je anotace přidaná, před provedením metody je spuštěn příslušný intercepter, který ověří, zda přistupující uživatel má danou roli.

```
@RequiresRoles(ADMIN)
public void secretMethod() {
  // povolen přístup uživateli s rolí ADMIN
}

@RequiresRoles({ADMIN, CUSTOMER})
public void secretMethod() {
  // povolen přístup uživateli s rolí ADMIN a CUSTOMER
}
```

Ukázka. 4.2: Použití anotace pro povolení přístupu přes role.

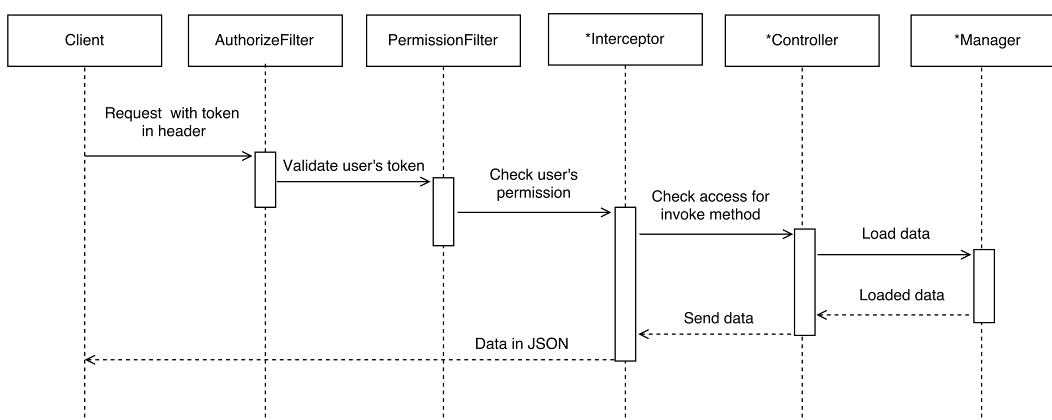
Druhou možností, jak povolit přístup k datům je povolení přístupu pro určitou operaci. Toto řešení je vhodné v situaci, kdy chceme uživateli povolit přístup k určitému modelu, ale chceme zamezit, aby jej mohl měnit. Každý uživatel tedy musí mít definované oprávnění k modelu. Práva jsou definované v enumu **Permission** a obsahuje název povolení s číselnou reprezentací - **READ - 4** a **WRITE - 2**. Čísla se používají při ukládání do databáze a názvy při odeslání přes síť. Omezení na metodu nebo třídu je možné přídáním anotace **RequiresPermissions**, kde oprávnění je uvedené jako parametr (ukázka 4.3).

```
@RequiresPermissions(READ)
public void secretMethod() {
  // povolen přístup uživateli s právem ke čtení
}

@RequiresPermissions({READ, WRITE})
public void secretMethod() {
  // povolen přístup uživateli s právem ke čtení a zápisu
}
```

Ukázka. 4.3: Použití anotace pro povolení přístupu přes oprávnění k modelu.

Proces autorizace je znázorněn na diagramu (obr. 4.6) a vysvětlen níže.



Obrázek 4.6: Diagram procesu autorizace.

- Klient vytvoří dotaz na URL cestu, která vyžaduje autorizaci. V hlavičce dotazu musí být uveden **X-Auth-Token**. Pokud token nebude uveden, není možné ověřit uživatele, přístup bude odmítnut a klientovi bude odeslána příslušná hláška.
- Na serveru je zavolán **AuthorizeFilter**, který zkontroluje, jestli je dotaz prováděn na obsah, který vyžaduje autorizaci. Pokud ano, načtou se údaje o přihlášeném uživateli a je provedena kontrola, zda je token v aktuálním čase stále platný.
- Pokud token je platný, zavolá se **PermissionFilter**. V cestě je dovolený pouze identifikátor modelu a *all*. Cokoliv jiného bude považované za špatně položený dotaz. Z dotazu se načte model, ke kterému přistupujeme a ověří se, zda má uživatel právo pro přístup k modelu.
- Data mohou podléhat různým úrovním oprávnění. Jestliže je dotaz položen na data, které nepodléhají oprávnění a nevyžadují určitou roli,

je zavolán příslušný kontroler. V kontroleru jsou načteny vstupní data, které jsou předány příslušnému manažeru, kde je provedena požadovaná operace. Operace může představovat načtení dat z databáze, kde se následně provede převod z doménových objektů na DTO, nebo například se může jednat o proces přepočítání grafových dat.

Pokud ale data podléhají nějakým restrikcím, je před kontrolerem zavolán interceptor (viz anotace *RequiresPermissions* a *RequiresRoles*), který prověří přístup pro danou operaci.

- Dotaz směřovaný klientovi obsahuje požadovaná data, popřípadě chybovou hlášku a v hlavičce se nachází opět autorizační token.

4.3 Klient pro správu uživatelů

K serverové části vznikla i klientská aplikace, která spravuje uživatele. Klientská aplikace využívá navržené API serverové aplikace a ověřuje funkčnost řešení. Pro uživatelský přístup je nutné se nejdříve přihlásit. Přihlašování probíhá přes jméno a heslo a přihlašovaný uživatel musí mít přidělené právo ADMIN. Po přihlášení se zobrazí tabulka s uživately, kde u každého z nich lze jednoduše upravit údaje např. jméno, email, roli nebo heslo. U každého uživatele je tlačítko, které zobrazí okno se všemi modely a přidělenými právy zobrazené jako zaškrtávací tlačítko.

Aplikace je napsaná v programovacím jazyce JavaScript a využívá knihovny - **JQuery**, **JQuery UI**, **JSGrid** a **SHA256**. Knihovna JSGrid vytváří tabulku, kde je možné upravovat řádky změnou v buňkách. Knihovna také využívá obě zmíněné JQuery knihovny. SHA256 knihovna je využívána pro vytvoření otisku hesla, které se následně s uživatelským jménem zakóduje kódováním Base64 a odešle na server.

5 Testování

5.1 Jednotkové testy

V aplikaci jsou vytvořeny jednotkové testy přesahující 1 000 řádek kódu, které testují nejproblematičtější části.

Převod doménových tříd na DTO je operace, při které nesmí dojít k chybě. Žádná hodnota se nesmí při převodu ztratit nebo změnit. Testy pokrývají převod všech doménových tříd a ověřují např. správnost převodu hodnoty číselníku na konkrétní číselník.

Přihlášení uživatele probíhá přes token, který se skládá z přihlašovacích údajů. Token je nutné dekódovat a získat potřebné údaje. Test popisující tuto funkčnost ověřuje, že nástroj funguje korektně pro různé chyby, které mohou při vytvoření tokenu vzniknout.

Druhou nejdůležitější operací je změna uživatelských práv na model. Test obsahuje všechny možnosti, které mohou vzniknout. Například přidávání, odebírání, úprava práv.

V aplikaci se nachází několik nástrojů, které usnadňují práci. Jedná se například o nástroj pro převod enumu do textového řetězce ve formátu JSON nebo převod oprávnění z čísla na pole oprávnění v textové podobě.

Vytváření nových instancí je pod správou kontejneru jednotlivých technologií. Tyto třídy nejsou otestované jednotkovými testy, ale využívají nástroje, které obsahují výkonnou část, a ty již otestované jsou. K severové aplikaci vznikla i klientská aplikace umožňující správu uživatelů. Touto aplikací je ověřeno, že filtry, interceptry ověřující role nebo například dynamické vkládání závislostí funguje korektně.

Pokoušel jsem se o implementci frameworku **Arquillian**. Pomocí tohoto frameworku lze spustit aplikaci na embedovaném aplikačním serveru, který obsluhuje kontejnery jetnotlivých technologií. Pomocí tohoto frameworku lze vytvářet požadavky na server a simulovat tím běh aplikace. Bohužel se mi nepovedlo tento framework nastavit tak, aby byl použitelný a mohly být

provedeny integrační testy. Využití Arquillian frameworku jsem chtěl otestovat průběh dotazu od API po databázi. Tato funkčnost není automaticky otestována, ale lze ji ověřit dodanou klientskou aplikací.

6 Závěr

Výsledkem mé práce je aplikace postavená na technologiích specifikace Javy EE. Architektura je logicky rozdělena do vrstvech a konfigurační soubory jsou uložené na standartních cestách. Aplikace rozšiřuje a mírně upravuje služby, které jsou dostupné přes API. Každý požadavek na server je podmíněný autorizačním tokenem, který uživatel získá při autentizaci. K serverové části aplikace jsem vytvořil jednoduchou klientskou aplikaci, která umožňuje spravovat uživatele.

V rámci této práce jsem se seznámil s nástroji a technologiemi pro vývoj podnikových aplikací. Naučil jsem se navrhovat vícevrstvé aplikace a konfigurovat aplikační server TomEE. Celou práci jsem vytvořil ve vývojovém prostředí IntelliJ IDEA, které usnadňuje vývoj, protože nabízí rozsáhlou správu projektu. Nabízí např. kontrolu konfigurací nejpoužívanější frameworků, což značně urychluje práci při odhalování chyb.

Druhá fáze vývoje je zaměřena především na efektivnější server. Serverová část byla rozdělena do dvou bakalářských prací. Moje část se zaměřovala na architekturu serveru, použité technologie, autorizaci a autentizaci. Druhá část práce je zaměřena na přístup k datovému modelu a grafové reprezentaci. Implementace druhé části je zpožděna a aktuálně není možné otestovat, zda druhá fáze splnila očekávání a bylo docíleno urychlení výpočtů.

Přehled zkratek

- CDI (Context And Dependency Injection) - technologie pro vkládání závislostí mezi komponentami
- EJB (Enterprise Java Beans) - technologie pro řízení komponent
- JPA (Java Persistence API) - technologie umožňující práci s databází
- JSF (JavaServer Faces) - technologie pro vývoj webových aplikací
- JSP (JavaServer Pages) - technologie umožňující vkládání Java kódu do HTML stránek
- JSTL (JavaServer Pages Standard Tag Library) - knihovna pro vývoj webových aplikací
- JTA (Java Transaction API) - technologie pro správu transakcí
- LDAP (Lightweight Directory Access Protokol) - protokol pro ukládání a přístup k informacím o uživatelích
- SSO (Single Sign-On) - systém jedinečného přihlášení
- ORM (Object-relational mapping) - objektově relační mapování
- DTO (Data transfer object) - objekt pránášející data doménového objektu mezi procesy
- DAO (Data access object) - objekt umožňující přístup do databáze

Přílohy

Překlad a spuštění aplikace

Pro úspěšné sestavení je vyžadováno mít JDK ve verzi 8 a program *Maven*, který aplikaci sestaví do **.war** balíku. Sestavení aplikace se provede přes příkaz v ukázace 6.1. Sestavená aplikace se nachází na:
ADRESÁŘ_PROJEKTU/target/vps.war.

```
mvn clean install -P development
```

Ukázka. 6.1: Příkaz pro sestavení aplikace.

Připojení k databázím spravuje aplikační server. Konfigurace datových zdrojů se nachází v souboru **TOMEE/conf/tomee.xml** (ukázka 6.2). Následně je nutné přidat do složky **TOMEE/lib** MySQL JDBC ovladač. Ukázka datového zdroje v konfiguračním souboru.

```
<Resource id="ModelsDS" type="javax.sql.DataSource">
  JdbcDriver  com.mysql.jdbc.Driver
  JdbcUrl     jdbc:mysql://localhost/models
  UserName    root
  Password    root
  jtaManaged  true
</Resource>

<Resource id="UsersDS" type="javax.sql.DataSource">
  JdbcDriver  com.mysql.jdbc.Driver
  JdbcUrl     jdbc:mysql://localhost/users
  UserName    root
  Password    root
  JtaManaged  true
</Resource>
```

Ukázka. 6.2: Nastavení datového zdroje v konfiguraci TomEE.

Sestavenou aplikaci uložíme do složky **TOMEE/webapps** a spustíme server (ukázka 6.3).

```
./bin/catalina.sh start
```

Ukázka. 6.3: Spuštění aplikačního serveru TomEE.

Postupy při obsluze aplikace

Změna aplikačního serveru

Aplikaci lze spustit také na jiném aplikačním serveru, který splňuje požadavky specifikace Java EE. Server může obsahovat knihovny, které jsou již v projektu přidány, poté stačí nastavit u dané závislosti v *pom.xml* scope na *provided*. Problém nastane také, pokud nový server nebosahuje ORM framework OpenJPA. Při změně frameworku je nutné změnit v *persistence.xml* OpenJPA klíč pro vytvoření tabulek do databáze na klíč, který odpovídá funkčnosti v novém frameworku.

Přidání doménové třídy nebo číselníku

Pokud by byla potřeba přidat novou doménovou třídu, musí být přidána do balíku **.domain** a povoleno její mapování v konfiguračním souboru *persistence.xml* (ukázka 6.4).

```
<persistence-unit name="ModelsDataSource" transaction-type="JTA">
    ...
    <class>cz.zcu.kiv.vps.domain.Model</class>
    <class>cz.zcu.kiv.vps.domain.DummyUser</class>
    ...
</persistence-unit>
```

Ukázka. 6.4: Konfigurační soubor JPA.

Nová třída musí obsahovat kopii v podobě DTO. Pro všechny změny, které jsou mezi doménovou třídou a DTO musí existovat odpovídající přepis definovaný v konvertoru (ukázka 6.5).

```
modelMapper.createTypeMap(Integer.class, CoordinatesType.class)
        .setConverter(e -> CoordinatesType.findByValue(e.getSource()));

modelMapper.createTypeMap(CoordinatesType.class, Integer.class)
        .setConverter(e -> e.getSource().getValue());
```

Ukázka. 6.5: Nastavení pravidel pro převod číselníku na hodnotu a obráceně.

Následně v testech vytvořit příslušný test pro danou třídu a otestovat, zda se třídy správně navzájem převádí (ukázka 6.6).

```
Bus entity = new Bus();
entity.setBusType(BusType.SWING_BUS);
...
BusDTO dto = ModelConverter.convert(entity, BusDTO.class);
assertEquals(entity.getBusType().getValue(), dto.getBusType().intValue());
...
```

Ukázka. 6.6: Převod doménové třídy na DTO.

Přidání nového číselníku (enumu) je velice podobné jako přidání doménové třídy. Více není nutné dělat, protože při požadavku na všechny číselníky se přes reflexi načtou všechny z balíku a převedou na data v JSON formátu.

Přidání nového repozitáře

Repozitáře jsou navržené tak, aby byly jednoduché na obsluhu. Nový repozitář musí být oddělený od třídy **AbstractRepository**, které obsahuje generický parametr. U nově vzniklé třídy bude tento parametr nahrazen konkrétní doménovou třídou, ke které se nové repozitory vztahují. Jelikož aplikace obsahuje dvě spojení do databáze, musí být určeno, ke které bude přistupovat. Databáze se vybírá anotací **Database**, která se přidává nad třídu a parametrem je název databáze (ukázka 6.7).

```
@Stateless @Database(MODELS)
public class ModelRepositoryImpl extends AbstractRepository<Model>
                                    implements ModelRepository
{}
```

Ukázka. 6.7: Správné vytvoření nového repozitáře.

Přidání nové databáze

Přidání nebo také odebrání databáze je záležitost několika po sobě jdoucích kroků. V konfiguračním souboru *tomee.xml* se přidá nový *Resource* (viz sekce 6.) Následně se provede přidání nového záznamu (*persistence-unit*) v konfiguračním souboru *persistence.xml* viz sekce 4.1.1.). Dalším krokem je ve třídě **DatabaseResources** vytvoření třídy *EntityManager* s anotací *PersistenceContext* a správně vyplněným jménem. Poslední krok je poskytnutí databáze repozitářům. Toho je docíleno tak, že je vytvořen *getter* s anotací

Produces a s anotací *DatabaseQualifier*, kde je v parametru uvedený o jakou databázi se jedná.

Změna jazyka uživatelských hlášek

Základním jazykem pro uživatelské hlášky je angličtina. Soubor s hláškama je uložený ve zdrojích - **LANG/Messages_en_US.properties**. Soubor se načítá podle *Locale*, které je v konfiguračním souboru *config.properties*.

Literatura

[1] *Java EE* [online]. Oracle, 2017. [cit. 2017/27/02]. Dostupné z: http://www.oracle.com/technetwork/java/javaee.

[2] *Java EE* [online]. 2017. [cit. 2017/03/12]. Dostupné z: http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html.

[3] Gupta, A. *Java EE 7 Essentials*. O'Reilly Media, inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2013. ISBN 978-1-449-37017-6.

[4] *Spring framework* [online]. 2017. [cit. 2017/02/03]. Dostupné z: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/.

[5] *Swagger Documentation* [online]. 2017. [cit. 2017/03/03]. Dostupné z: http://swagger.io/docs/.

[6] *Swagger Editor* [online]. 2017. [cit. 2017/03/04]. Dostupné z: https://github.com/swagger-api/swagger-editor.

[7] *Swagger Codegen Documentation* [online]. 2017. [cit. 2017/03/04]. Dostupné z: https://github.com/swagger-api/swagger-codegen.

[8] *Swagger UI* [online]. 2017. [cit. 2017/03/04]. Dostupné z: https://github.com/swagger-api/swagger-ui.

[9] *Apiary Products* [online]. 2017. [cit. 2017/03/05]. Dostupné z: https://apiary.io/products.

[10] *Apiary Editor* [online]. 2017. [cit. 2017/03/05]. Dostupné z: https://help.apiary.io/tools/apiary-editor/.

[11] *Apache TomEE* [online]. Apache, 2017. [cit. 2017/27/02]. Dostupné z: http://tomee.apache.org/apache-tomee.

[12] *About Wildfly* [online]. 2017. [cit. 2017/03/09]. Dostupné z: http://wildfly.org/about/.

[13] *PicketLink Documentation* [online]. 2017. [cit. 2017/03/08]. Dostupné z: http://docs.jboss.org/picketlink/2/latest/reference/html/.

[14] *JSR 353* [online]. 2017. [cit. 2017/04/26]. Dostupné z: http://docs.oracle.com/javaee/7/tutorial/jsonp.htm.

[15] *Johnzon* [online]. 2017. [cit. 2017/04/26]. Dostupné z: https://johnzon.apache.org/johnzon-jaxrs/project-info.html.