

Agregátor databází veřejných zakázek

Bc. Radek Los

Diplomová práce
2013



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2012/2013

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Radek Los**
Osobní číslo: **A12696**
Studijní program: **N3902 Inženýrská informatika**
Studijní obor: **Informační technologie**
Forma studia: **kombinovaná**

Téma práce: **Agregátor databází veřejných zakázek**

Zásady pro vypracování:

1. Prostudujte informační systémy veřejných zakázek tj. Věstník MMR, elektronická tržiště Gemin, tendermarket.cz, centrumvz.cz, b2bcentrum.cz, vortalgov.cz, ceskytrh.cz, zakazky.praha.eu.
2. Navrhňte databázovou strukturu pro agregaci údajů o zakázkách ze všech výše uvedených zdrojů.
3. Implementujte parsery pro načítání údajů z jednotlivých zdrojů do vaší databáze. Nezapomeňte na to, že stavy jednotlivých zakázek se v čase mění a váš systém tyto změny musí reflektovat.
4. Implementujte portál pro zobrazování a vyhledávání v agregované databázi.
5. Věnujte pozornost zabezpečení.
6. Implementujte systém pro notifikaci registrovaných uživatelů o nových zakázkách na základě jimi zadaných kritérií (oborů, klíčových slov, regionu, atd.).

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **BAUER, Christian a Gavin KING. Hibernate in action. Greenwich: Manning Publications, 2005, xxiii, 408 s. ISBN 19-323-9415-X.**
2. **DEINUM, Marten, Koen SERNEELS, Colin YATES, Seth LADD a Christophe VANFLETEREN. Pro Spring MVC: with Web Flow. New York: Distributed to the book trade worldwide by Springer Science Business Media, 2012, xxviii, 565 p. ISBN 978-143-0241-560.**
3. **KNUDSEN, Patrick Niemeyer and Jonathan. Learning Java. 2nd ed. Beijing: O'Reilly, 2002. ISBN 05-960-0285-8.**
4. **OBE, Regina O. a Leonard HSU. PostgreSQL: Up and Running. Indianapolis, IN: O'Reilly Media, Inc., 2012, xi, 753 p. ISBN 1449326331.**
5. **OBE, Regina a Leo HSU. PostGIS in action. London: Pearson Education [distributor], 2011, xxviii, 492 p. ISBN 19-351-8226-9.**
6. **SVEN LÜPPKEN, Markus Stäuble a Luca Masini REVIEWERS. Spring Web Flow 2 Web development master Spring's well-designed Web frameworks to develop powerful Web applications. Birmingham, U.K: Packt Pub, 2009. ISBN 18-471-9542-3.**

Vedoucí diplomové práce:

Ing. Tomáš Dulík, Ph.D.

Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:

22. února 2013

Termín odevzdání diplomové práce:

22. května 2013

Ve Zlíně dne 22. února 2013



prof. Ing. Vladimír Vašek, CSc.
děkan



doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Cílem práce je vytvoření web aplikace, která získává údaje o zakázkách z elektronických tržišť veřejných zakázek a uloží je do jedné databáze. K databázi uživatelé mohou přistupovat pomocí webové aplikace, která nabízí více funkcí, než originální tržiště. Příkladem je např. zasílání emailových notifikací o nových zakázkách na základě zvolených kritérií.

Klíčová slova: Spring, Java EE, JSF, PostgreSQL, Hibernate, JPA, tržiště veřejných zakázek, web crawler, databáze, webová aplikace, Maven

ABSTRACT

The aim of the thesis is to create Java web crawler, which gets public tenders from Czech market places and store into database. Database is accessed from web application, which offers wider search options than market places. Logged user can save tender criteria and can be notified by emails.

Keywords: Spring, Java EE, JSF, PostgreSQL, Hibernate, JPA, public tenders, web crawler, database, web application, Maven

Tímto bych chtěl poděkovat vedoucímu práce Ing. Tomášovi Dulíkovi, Ph.D. za odborné vedení a konzultace diplomové práce.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze **diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.**

Ve
.....

diplomanta

Zlíně

podpis

Obsah

ÚVOD.....	9
I TEORETICKÁ ČÁST	9
1 VEŘEJNÉ ZAKÁZKY	11
1.1 VĚSTNÍK VEŘEJNÝCH ZAKÁZEK	12
1.2 ZÁKON O VEŘEJNÝCH ZAKÁZKÁCH.....	12
1.3 ZÁSADY ZADÁVÁNÍ VEŘEJNÝCH ZAKÁZEK	13
1.4 ZADAVATELÉ	13
1.4.1 Veřejný zadavatel	13
1.4.2 Dotovaný zadavatel	13
1.4.3 Sektorový zadavatel.....	14
1.4.4 Centrální zadavatel	14
1.5 ZAKÁZKY	14
2 NÁVRH A POPIS APLIKACE	15
3 JAVA	17
3.1 SPRING FRAMEWORK	17
3.1.1 Spring MVC	18
3.1.2 Spring Validation	20
3.2 JAVASERVER FACES.....	23
3.2.1 Konfigurační soubor: web.xml	24
3.2.2 Konfigurační soubor: faces-config.xml	24
3.3 HTML PARSER - JSOUP	25
3.3.1 Požadavky	25
3.3.2 Příklady	26
3.4 JUNIT TESTOVÁNÍ	28
3.4.1 Unit testy s JUnit	28
3.4.2 Anotace JUnit.....	28
3.4.3 Vyhodnocování testů	29
3.4.4 JUnit test suite.....	30
3.4.5 JUnit v Eclipse	30
3.4.6 Parametrizovaný test.....	33
3.4.7 Pravidla.....	34
4 POSTGRESQL	35
4.1 POUŽITÍ TRIGGERŮ	35
4.2 POSTGIS	37
4.2.1 Geometrické objekty.....	37

4.2.2	Kolekce - Collections	40
4.2.3	Geografické objekty	41
II	PROJEKTOVÁ ČÁST	42
5	PROGRAMOVÝ MANUÁL	44
5.1	DATABÁZE	44
5.2	DIGGER	47
5.2.1	Použité API	47
5.2.2	Parsování tržišť a veřejných zakázek	48
5.3	WEBOVÁ APLIKACE	49
5.3.1	Řadiče	50
5.3.2	Vyhledávač	51
5.3.3	Bezpečnost	52
6	POPIS WEBOVÉ APLIKACE	53
6.1	NEREGISTROVANÍ UŽIVATELÉ	53
6.2	REGISTROVANÍ UŽIVATELÉ	56
	ZÁVĚR	57
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	59
	SEZNAM OBRÁZKŮ	60

ÚVOD

Na základě nové legislativy bylo v roce 2012 spuštěno několik elektronických tržišť veřejných zakázek. Bohužel, každé tržiště má jiné možnosti, jiné uživatelské rozhraní a neexistuje žádné centrální místo, kde by bylo možné vyhledávat nové zakázky. Pro dodavatele, kteří se chtějí účastnit veřejných zakázek, je tento stav tristní, protože sledování nových zakázek na všech tržištích znamená vyhradit jednoho zaměstnance jen pro tuto činnost. Naprostá většina malých firem v ČR na to ale nemá kapacity ani prostředky. Zadavatelé zakázek díky tomu mají možnost zakázku „schovat“ na některé málo používané tržiště, kde si jí všimne minimum dodavatelů, což nahrává klientelismu, popř. korupci.

Cílem práce je vytvoření systému pro agregaci nových veřejných zakázek ze všech elektronických tržišť na jediné místo. Systém se skládá ze dvou komponent:

1. klient pro získávání dat ze všech elektronických tržišť veřejných zakázek. Klient stahuje data z tržišť ve formátu *HTML*, data zpracovává a opravuje/doplňuje pomocí veřejně dostupných rejstříků (např. rejstřík ekonomických subjektů, rejstřík adres atd.)
2. webová aplikace pro zobrazování a vyhledávání veřejných zakázek.

Domnívám se, že vytvoření takového systému by mělo zvýšit transparentnost veřejných zakázek.

Práce je rozdělena na část teoretickou a projektovou. V teoretické části se zabývám jak popisem legislativy veřejných zakázek, tak popisem použitých technologií, jako je Java, framework Spring a *JSF* pro pohledovou vrstvu aplikace. Dále popisuji Java *HTML* parser, který je použit pro získání jednotlivých údajů veřejných zakázek, JUnit testování a databázi PostgreSQL.

V projektové části se zabývám dokumentací vytvořených aplikací a to jak uživatelské, tak programové dokumentace. Popisuji zde vytvoření web crawleru a použití API pro opravu dat, převážně adres a doplnění *GPS* souřadnic. U webové aplikace ukazují její možnosti a způsob zobrazení veřejných zakázek.

I. TEORETICKÁ ČÁST

1 VEŘEJNÉ ZAKÁZKY

Veřejné zakázky slouží pro nákup zboží, zadání práce, objednání díla nebo služby veřejným subjektem. Veřejným subjektem je stát, obec, samostatný celek, organizace jimi založené, nebo případně dalším subjektem, který hospodaří s penězi, nebo jinými veřejnými statky, nebo s hodnotami pocházejícími z daní, poplatků či jiných zdrojů veřejného bohatství. Veřejná zakázka je realizovaná na základě smlouvy mezi zadavatelem a jedním nebo více dodavateli. Jedná se o úplatné poskytnutí dodávek či služeb nebo úplatné provedení stavebních prací. Jednou ze stran uzavírajících smlouvu je veřejný zadavatel. [6]

Veřejná zakázka je uzavřena mezi zadavatelem a jedním či více dodavateli na základě smlouvy, jejímž předmětem je úplné poskytnutí dodávek či služeb nebo provedení stavebních prací. Veřejná zakázka musí být uskutečněna na základě písemné smlouvy. Důvodem k zadání veřejné zakázky je veřejná potřeba, či veřejný zájem na pořízení předmětu, služby či na splnění úkolu, jež je jejím cílem a je hrazen z veřejných financí nebo má například stát na veřejném pozemku, nebo, v krajním případě, jinak ovlivňovat veřejný prostor nebo zájem. Veřejný zájem může být skutečný nebo uměle vytvořený, s cílem přichystat pracovní náplň subjektům ve spřáteleném prostředí a zajistit jim tak zakázku a spolehlivý příjem z veřejných zdrojů. [6]

Nejčastějšími podobami veřejných zakázek jsou stavební práce silnic, mostů, budov nebo vodovodních a kanalizačních sítí, služby různého druhu (např. právní služby nebo likvidace komunálního odpadu), veřejné osvětlení, nákup zbraní, IT sítě, dříve i telekomunikační sítě, dopravní služby atp. Veřejnými zakázkami mohou být rovněž projekční práce pro zhotovení stavebních děl, nebo projekty jiného druhu. Specifickým, ale tradičním a velmi starým druhem veřejné soutěže je architektonická soutěž, která je zvláštním předstupněm zadání veřejné zakázky o návrh. Veřejné zakázky jsou pro jejich zhotovitele značně atraktivní, protože stát nebo jiné veřejné subjekty jsou obvykle solventní, v převážné většině případů plní své závazky, zadavatelem se stávají opakovaně a dlouhodobě a veřejné zakázky patří k největším zakázkám. Trvání veřejných subjektů je značně delší, než je to u soukromých subjektů - je téměř neomezené. Důležitým aspektem zvyšujícím atraktivitu je to, že správci veřejného majetku na něj nedbají s takovou důsledností a přísností, jak je tomu u soukromých a bezprostředních vlastníků. [6]

Atraktivita veřejných zakázek vede nezdědky k pokusům o to, aby tyto zakázky nebyly zadávány nejčistším způsobem, tedy takovým, který přináší veřejnému zadavateli (vlastně veřejnému zájmu, a tedy společnosti), nejlepší profit (převzetí nejlepší kvality zakázky oproti vynaložené ceně). Vede také k zadávání nemalé části veřejných zakázek spřátelenému prostředí (ke klientelismu) a stejně tak je tato činnost pravděpodobně provázena i ko-

rupcí. [6, 7]

1.1 Věstník veřejných zakázek

Věstník veřejných zakázek zajišťuje uveřejňování informací povinně uveřejňovaných zadavatelem, které jsou uvedeny v zákoně o veřejných zakázkách a koncesním zákoně a jejichž realizace je upravena v prováděcím nařízení Komise (EU) č. 842/2011 ze dne 19. srpna 2011, kterým se stanoví standardní formuláře pro zveřejňování oznámení v oblasti zadávání veřejných zakázek a kterým se ruší nařízení (ES) č. 1564/2005. [6, 7]

1.2 Zákon o veřejných zakázkách

Zákonem číslo č. 137/2006 Sb. o veřejných zakázkách se měl aktualizovat předchozí zákon č. 40/2004 Sb., který by přijal nové směrnice Evropské Unie. Tato změna měla přinést modernější, flexibilnější, jednodušší právní rámec pro veřejné zakázky. Přijetí nových směrnic Evropské unie a jejich implementace do českého práva ovlivňovala následující směrnice. [6, 7]

- Směrnice Evropského parlamentu a Rady 2004/18/ES ze dne 31. března 2004, týkající se koordinace postupů při zadávání veřejných zakázek na stavební práce, dodávky a služby.
- Směrnice Evropského parlamentu a Rady 2004/17/ES ze dne 31. března 2004, týkající se postupů při zadávání zakázek subjekty, působícími v odvětví vodního hospodářství, energetiky, dopravy a poštovních služeb.

Výše uvedené směrnice přinesly několik nových institutů, které bylo nutno zařadit z hlediska systematiky. Velké množství ustanovení bylo novelizováno. Z tohoto důvodu byla novelizace zákona zamítnuta zákonodárci a vytvořil se zákon úplně nový, který nabyl účinnosti 1. července 2006 a musel obsahovat. [6, 7]

- Celkové zjednodušení a zpřesnění zákona
- Snaha o zohlednění praktických zkušeností
- Vyjasnění základních pojmů zákona
- Podrobnější specifikace jednotlivých zadávacích postupů, jejich zjednodušení a snaha o nastolení větší rovnováhy mezi transparentností a proporcionalitou
- Nově mezi sektorové zadavatele byly zařazeny subjekty poskytující poštovní služby

- Možnost zadávání pomocí společných nákupních subjektů
- Nová forma zadávacího řízení
- Zjednodušení podlimitního řízení
- Rozšíření využití rámcových smluv nejen na sektorové zadavatele
- Elektronizace procesu zadávání
 - Elektronické aukce
 - Dynamické nákupní systémy

1.3 Zásady zadávání veřejných zakázek

Zadavatel musí postupovat podle zákona § 6 a musí dodržovat zásady diskriminace, rovného zacházení a transparentnosti. Tyto zásady vycházejí z práva evropských společenství a je nutné je interpretovat na základě evropského práva ESD. Zásada diskriminace znamená nezvýhodňovat nebo neomezovat žádného dodavatele. Tato zásada je ale nejvíce porušována pomocí kritérií veřejných zakázek.

Zásada rovného zacházení klade povinnost zadavatelům být ke všem dodavatelům neutrální, tedy stanovení podmínek, které jsou pro všechny stejné. Zásada transparentnosti se prolíná celým zadávacím řízením. Je povinností zadavatele veřejně informovat o zadávacím řízení a také odůvodnit svá rozhodnutí o výběru dodavatele. Smyslem všech zásad je odhalit korupční jednání. [6, 7]

1.4 Zadavatelé

1.4.1 Veřejný zadavatel

Je popsán v zákonu č. 40/2004 Sb § 2 odst. 2 ZVZ a je jím Česká republika, územní samosprávný celek, státní příspěvková a jiná právnická osoba. Účelem je uspokojení veřejného zájmu bez průmyslové nebo obchodní povahy. Tento typ zadavatele musí být finančně podporován hlavně státem. Veřejnými zadavateli jsou např. soudy, ČNB, Česká televize, státní zastupitelství atd. [6, 7]

1.4.2 Dotovaný zadavatel

Je vymezen §2 odst. 3 ZVZ. Vztahuje se na právnickou nebo fyzickou osobu zadávající veřejnou zakázku hrazenou z větší části z peněžních prostředků poskytnutých veřejným zadavatelem. [6, 7]

1.4.3 Sektorový zadavatel

Subjekt vykonávající některou zákonem vymezenou relevantní činností v §4 ZVZ v odvětví teplárenství, plynárenství, vodárenství, elektroenergetiky, sítí, dopravním, poštovních služeb atd. [6, 7]

1.4.4 Centrální zadavatel

Podle §3 je veřejný zadavatel umožňující zadavatelům obstarávat služby, zboží a stavební práce, aniž by tito podstupovali zadávací řízení. Nejedná se o společné zadání. Centrální zadavatel může pořizovat zboží či služby během zadávacího řízení a ty poté předprodává zadavatelům. Centrální zadavatel může také podstupovat zadávací řízení na účet zadavatelů, kteří jej k tomu zmocní. V obou případech se ale musí uzavřít písemná smlouva. [6, 7]

1.5 Zakázky

Zakázky jsou děleny podle předpokládané hodnoty podle § 14 ZVZ. V případě, že zakázka přesáhne limity uvedené v prováděném právním předpise a více než 600 000 EUR u dodávky či služby a nebo 5 mil. EUR u stavební práce, jedná se o zakázku nadlimitní. Zakázky, které nepřesáhnou uvedený limit u nadlimitních zakázek, jsou označovány jako podlimitní. [6, 7]

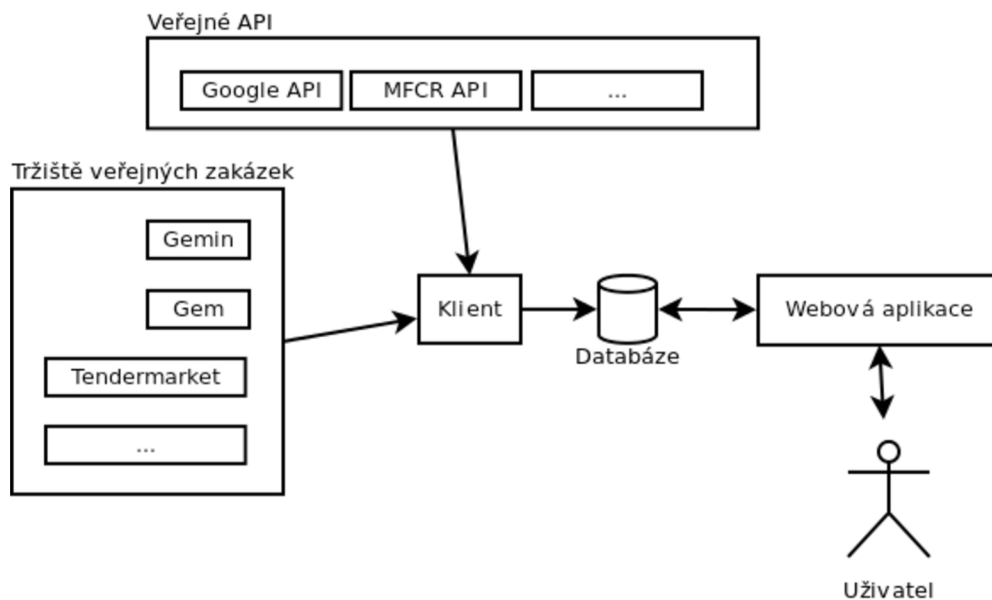
2 NÁVRH A POPIS APLIKACE

Projekt se skládá ze dvou aplikací a to konzolové (dále klient) a webové. Klient je aplikace, která je navržena tak, aby byla spouštěna pomocí plánovače Cron, nebo správci úloh na serveru. Nemá možnost žádné konfigurace spuštění a výstup je pouze log soubor s chybami během získávání veřejných zakázek.

Úloha klienta je pouze ukládat veřejné zakázky do databáze (neprovádí úpravu dat pomocí *SQL* příkazu *UPDATE*), tedy při získání veřejné zakázky se neaktualizuje její stav na základě tržiště, odkud byla získána. Stav se určuje pomocí časové značky uložené v tabulce jako je platnost veřejné zakázky.

Klient také opravuje data a převádí údaje veřejných zakázek na lépe použitelný formát. Například datum a čísla jsou získána jako řetězec, ale klient je převádí na daný formát pro pozdější snadnější manipulaci. Také dohledává a páruje jednotlivé položky s číselníky a klasifikacemi, jako jsou právní formy organizace, CPV, NUTS atd.

Webová aplikace slouží pro zobrazování uložených veřejných zakázek pomocí klienta. Veřejné zakázky jsou zobrazeny podobným způsobem jako na ostatních tržištích. Aplikace nabízí vyhledávání mezi veřejnými zakázkami získaných z jiných elektronických tržišť a přímý odkaz na zdrojovou veřejnou zakázku na tržišti. Přihlášení uživatelé mohou uložit kritéria, o které mají zájem, a poté být emailem upozorněni po přidání nové zakázky odpovídajícími kritérii klientem.



Obrázek 1. Schéma aplikace

Jednosměrné šipky na obrázku 1 znázorňují vazby typu READ-ONLY a dvousměrné READ-WRITE. Obousměrná šípka uživatele znázorňuje uživatelské akce jako je registrace, přihlášení a uložení kritérií pro veřejné zakázky, přičemž webová aplikace nenabízí vložení nové veřejných zakázek do databáze.

Jak klient, tak webová aplikace jsou napsány v jazyce Java z důvodu nasazení na libovolný operační systém. Aplikace používají framework *Hibernate* pro přístup k databázi a webová aplikace je postavena na frameworku *Spring*. Databáze byla zvolena PostgreSQL a to kvůli licenci open source a možnostem doplňků, např. *PostGis*.

3 JAVA

3.1 Spring framework

Spring Framework poskytuje komplexní programovací a konfigurační model pro moderní Java aplikace. Klíčovým prvkem *Spring* je podpora infrastruktury na úrovni aplikace. *Spring* se zaměřuje na enterprise aplikace, takže týmy se mohou soustředit na aplikační úrovni business logiky bez zbytečných vazeb na konkrétní prostředí implementace.

- Flexibilní *dependency injection* pomocí *XML* a použitím anotací
- Prvotní podpora open source projektu jako je *Hibernate* a *Quartz*
- Výkonnou abstrakci pro práci s běžnými Java EE specifikace jako *JDBC*, *SPS*, *JTA* a *JMS*
- Webový framework pro vývoj RESTfull *MVC* aplikací a koncových bodů
- Snadno testovatelný

Spring je navržen modulárně, což umožňuje postupné připojení jednotlivých částí, jako je jádro frameworku nebo podpora *JDBC*. Zatímco všechny *Spring* služby jsou velice dobře kompatibilní s jádrem, může být použito mnoho služeb nebo frameworků, které nespádají pod *Spring*, např. *FreeMaker* pro renderování pohledové vrstvy aplikace a jiné.

Hlavní předností *Spring* je tzv. *dependency injection* (DI) a *aspect-oriented programming* (AoP). *DI* je technika, jejíž podstatou je omezení závislostí při definici objektů uvnitř třídy. Každá využívaná komponenta by měla být nahraditelná jinou, pomocí deklarace jejího rozhraní a využití mutátorů.

Princip AoP je rozdělení funkcionalit aplikace do menších částí, které odpovídají konkrétním požadavkům, nikoli větším celkům.

Podporované prostředí pro nasazení aplikace *Spring* může být *Tomcat* a jiné Java EE servery. *Spring* je také hodně podporováno na cloudových platformách s podporou Javy, např. na Heroku, Google App Engine, Amazon Elastic Beanstalk a VMware Cloud Foundry. Knihovny *Spring* jsou zabudovány do vývojových prostředí, jako Eclipse a NetBeans, a mohou být také použity v projektu *Maven* při použití následujících závislostí. [3]

```
<repository>
  <id>springsource-repo</id>
  <name>SpringSource Repository</name>
  <url>http://repo.springsource.org/release</url>
</repository>
```

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-context</artifactId>  
  <version>3.2.2.RELEASE</version>  
</dependency>
```

3.1.1 Spring MVC

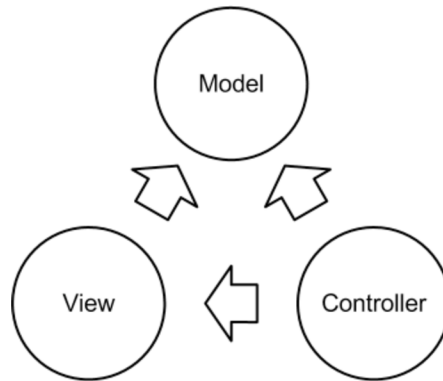
MVC je architektura softwaru, která byla vyvinuta pro potřebu oddělení části kódu od aplikační logiky (model), zobrazování dat (view) a kódu obsluhy (controller). Touto architekturou se aplikace zpřehlední, usnadní budoucí vývoj a umožňuje testování každé části zvlášť.

Model zajišťuje rozhraní mezi daty celé aplikace. Podstata modelu je zapouzdření přístupu k datům. Například nákupní košík v obchodě má dvě základní možnosti, a to přidání položky do košíku a odebrání položky z košíku. Uložení seznamu kupovaných předmětů může být uloženo v databázové tabulce, nebo v uživatelské relaci (session). Obě tyto varianty mají jinou logiku uchování dat, ale model nabízí pevně dané rozhraní, které správu předmětů v nákupním košíku sjednocuje.

Vrstva aplikace *View*, neboli pohled, slouží pro zobrazování uživatelského rozhraní, jako je tomu u *HTML*, *JSF*, *JSP*, *FreeMaker*, atd. Většinou je používán šablonovací systém.

Controller, neboli řadič, zpracovává požadavky od uživatele a podle požadavků volá model a upravuje pohled. Řadič přijímá všechny dotazy od klienta a řídí tok celé aplikace.

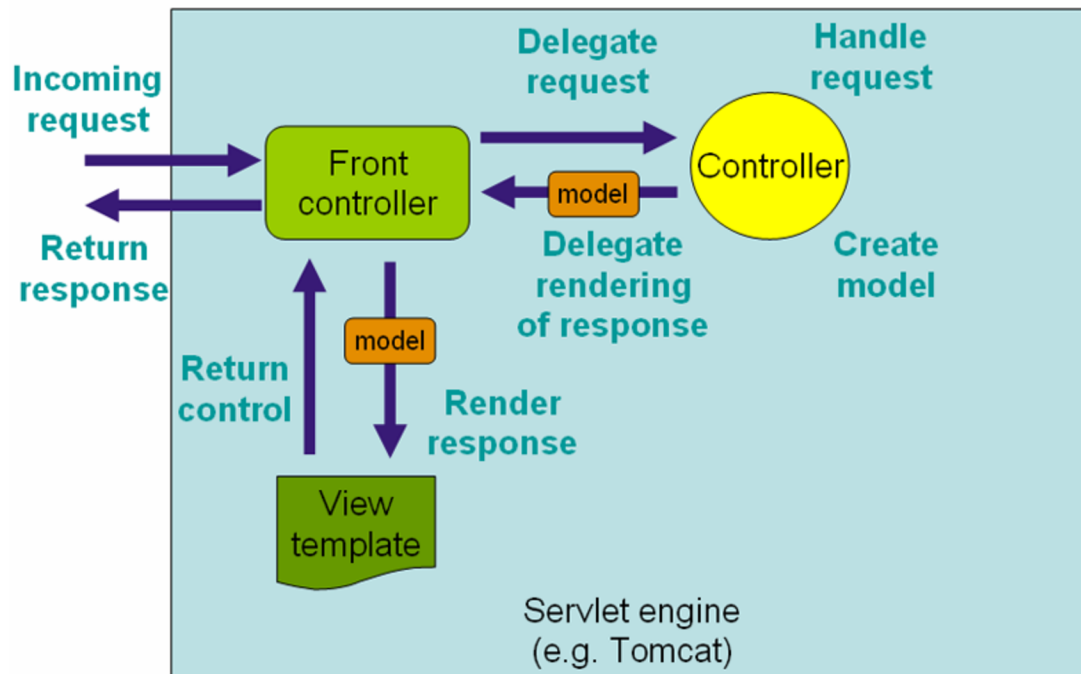
Například přihlašovací formulář je definován pomocí *HTML* tagů ve vrstvě pohledu. Po odeslání formuláře se zavolá řadič, který zpracuje data z *HTTP* požadavku a poskytne je modelu. Ten ověří, zda uživatelské jméno a heslo odpovídá nějakému uživateli. Pokud ano, vrátí potřebná data, která jsou řadičem s použitím modelu uložena do uživatelské relace a dojde k přesměrování na uživatelskou stránku. Pokud není uživatel nalezen, pohled zůstane na přihlašovací stránce a zobrazí se navíc zpráva o špatně zadaných údajích. [3, 4]



Obrázek 2. Diagram MVC [3]

Spring MVC framework je podobně jako ostatní webové frameworky řízen dotazy určenými kolem centrálního servletu, které odesílají žádosti řadičům a nabízejí ostatním funkcionalitu a usnadňuje vývoj webových aplikací. *Spring DispatcherServlet* je kompletně integrován pomocí *Spring IoC* a umožňuje použít každou funkci, kterou *Spring* má.

Zpracování toku žádostí *Spring Web MVC DispatcherServlet* je ilustrováno na následujícím diagramu. *DispatcherServlet* je vyobrazen jako *Front Controller*. [3]



Obrázek 3. Zpracování požadavků v Spring Web MVC [1]

DispatcherServlet je *Servlet*, který je potomkem třídy *HttpServlet* a je deklarován v *web.xml* ve webové aplikaci. Žádost, která by se měla být zpracována *DispatcherServlet*, musí být mapována pomocí *URL* adresy ve stejném souboru. Jedná se o klasickou J2EE servlet

konfiguraci. Příklad deklarace *DispatcherServlet* a mapování lze vidět na následujícím příkladu.

```
<web-app>
  <servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>example</servlet-name>
    <url-pattern>*.form</url-pattern>
  </servlet-mapping>
</web-app>
```

V předchozím příkladu všechny požadavky obsahující řetězec *.form* v adrese budou zpracovány *DispatcherServlet* pojmenovaným *example*.

Dalším krokem je definice řadiče. Řadič je část návrhového vzoru *MVC* (pro bližší specifikaci se jedná o písmeno *C*). Řadič umožňuje přístup do chování aplikace, který je typicky definován jako rozhraní služby. Jedná se o interpretaci uživatelských vstupů a výstupů do určitého modelu, který je reprezentován v uživatelském rozhraní tzv. pohled (HTML). *Spring* zavádí řadič velice abstraktním způsobem umožňující širokou škálu různých druhů řadičů.

Základní architektura řadiče *Spring* frameworku je rozhraní *org.springframework.web.servlet.mvc.Controller*.

```
public interface Controller {
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

Jak je vidět, na rozhraní řadič definuje jednu metodu, která je zodpovědná za řízení požadavku a vrácení správného modelu a pohledu. *ModelAndView* a *Controller* jsou základem pro *Spring MVC* implementaci. Rozhraní *Controller* je velice abstraktní, ale *Spring* nabízí mnoho implementací řadičů, které obsahují mnoho funkcí podle požadavků aplikace. *Controller* pouze definuje základní povinnost pro každý řadič, a to zpracování žádosti a vrácení modelu a pohledu. [3]

3.1.2 Spring Validation

Spring Validation rozhraní slouží pro validaci vstupních dat, které není nijak závislé na implementaci celé aplikace. Toto rozhraní slouží pro validaci objektů a v případě chyby je

vracena pomocí objektu *Errors*. [1]

Pro validaci objektu nám poslouží následující třída *Person*.

```
public class Person {
    private String name;
    private int age;
    // the usual getters and setters...
}
```

Validátor ověří správnost třídy *Person* pomocí implementace metody *org.springframework.validation.Validator* [1].

- *supports(Class)* - Ověří, zda může být validace prováděna s předanou proměnnou.
- *validate(Object, org.springframework.validation.Errors)* - Ověří předaný objekt a v případě chyby ji registruje a předá ji objektu *Errors*.

Implementace třídy validace objektu.

```
public class PersonValidator implements Validator {
    /**
     * This Validator validates *just* Person instances
     */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }
    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

Pro kontrolu jména je použita statická metoda *ValidationUtils.rejectIfEmpty*, která vrátí chyby do instance proměnné *e* a to v případech, kdy předaný řetězec je *NULL*, nebo je prázdný (""). Druhá podmínka ověří, zda je věk osoby příliš vysoký (hodnota je větší jak 110), nebo je záporná. [1]

Je možné implementovat jeden validátor pro ověření všech vnořených objektů, ale je doporučeno definovat více validátorů pro každý takový objekt. Pokud třída *Person* bude rozšířena o vlastnost adresa, která může být použita nezávisle na třídě *Person*, je tedy vhodné definovat vlastní validátor pro ni samotnou. [1]

```
public class PersonValidator implements Validator {
    private final Validator addressValidator;
```

```

public PersonValidator(Validator addressValidator) {
    if (addressValidator == null) {
        throw new IllegalArgumentException("The supplied ↴
            ↴ [Validator] is required and must not be null.");
    }
    if (!addressValidator.supports(Address.class)) {
        throw new IllegalArgumentException("The supplied ↴
            ↴ [Validator] must support the validation of ↴
            ↴ [Address] instances.");
    }
    this.addressValidator = addressValidator;
}
/**
 * This Validator validates Person instances, and any ↴
 * ↴ subclasses of Person too
 */
public boolean supports(Class clazz) {
    return Person.class.isAssignableFrom(clazz);
}
public void validate(Object target, Errors errors) {
    ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
    Person p = (Person) obj;
    if (p.getAge() < 0) {
        e.rejectValue("age", "negativevalue");
    } else if (p.getAge() > 110) {
        e.rejectValue("age", "too.darn.old");
    }
    try {
        errors.pushNestedPath("address");
        ValidationUtils.invokeValidator(this.addressValidator, ↴
            ↴ person.getAddress(), errors);
    } finally {
        errors.popNestedPath();
    }
}
}

```

Chyby validace jsou předány validátorem do objektu *Errors*. V případě použití *Spring Web MVC* může být použit tag `<spring:bind>` k prohlédnutí chybových zpráv, ale také přichází v úvahu získat chyby vlastním způsobem. [1]

Novinkou ve *Spring* verze 3 je tzv. validace *JSR-303 Bean Validation API*. *JSR-303* standardizuje ověřování dat v jazyce Java. Pomocí této *API* se mohou rozšířit vlastnosti tříd o deklarativní validace, které se automaticky provádí při běhu aplikace a plnění objektu hodnotami. Existuje celá řada vestavěných omezení, která mohou být využita a také se mohou definovat vlastní. [1]

```

public class PersonForm {
    @NotNull
    @Size(max = 64)
    private String name;
    @Min(0)
    private int age;
}

```

Spring nabízí plnou podporu *JSR-303 Bean Validation API*. To umožňuje *javax.validation.ValidatorFactory*, nebo *javax.validation.Validator* být aplikován tam, kde je potřeba validace dat v aplikaci.

Spring MVC má schopnost automaticky validovat vstupy řadičů. Pro automatickou validaci vstupních objektů parametr se označí anotací `@Valid`. Framework automaticky ověří správnost data po naplnění objektu daty podle toho, jak byl validátor konfigurován.

```
@Controller
public class MyController {
    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { /* ... */ }
}
```

Instance validace může být volána `@Valid` dvěma odlišnými způsoby. První je volat `binder.setValidator(Validator)` v rámci volání `@InitBinder` v řadiči. Tento způsob dovoluje konfigurovat instanci validátoru pro třídu řadič.

```
@Controller
public class MyController {
    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new FooValidator());
    }
    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { ... }
}
```

Druhým způsobem můžeme volat `setValidator(Validator)` globálně na `WebBindingInitializer`. Tento způsob konfiguruje validátor napříč všemi řadiči. To může být dosaženo snadno pomocí jmenného prostoru *Spring MVC*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <mvc:annotation-driven validator="globalValidator"/>
</beans>
```

3.2 JavaServer Faces

Nativně podporovaný framework pro pohledovou vrstvu ve frameworku *Spring* je *JSP*. *JSP* má omezené možnosti při používání šablon. Nepodporuje komponenty a ukládání jejich stavů. Z toho důvodu jsem se rozhodl použít *JSF*, který je plně podporován frameworkem *Spring*. Jedna z výhod *JSF* je, že z něj vychází mnoho dalších technologií, které nabízí bohatší uživatelské rozhraní pomocí tzv. Ajax komponent a *JSF* se dá o tyto technologie rozšířit či snadno zaměnit.

JavaServer Faces (JSF) je standardní Java framework pro vývoj uživatelského rozhraní webových aplikací. *JSF* zjednodušuje vývoj uživatelského rozhraní, které bývá často kom-

plikované při vývoji webových aplikací. I když je možné psát uživatelské rozhraní pomocí Java Web technologií, jako jsou Java servlety a *JavaServer Pages* bez použití frameworku, mohou tím nastat problémy během vývoje aplikace. [2]

JavaServer Faces je navržen tak, aby zjednodušil vývoj uživatelského rozhraní pro webové aplikace následujícími způsoby:

- Poskytuje komponenty, které jsou orientovány na klienta a nezávislý přístup k vývoji webových uživatelských rozhraní, čímž se zvyšuje produktivita vývojářů a snadnost použití.
- Zjednodušuje přístup a správu dat z webového rozhraní.
- Automaticky spravuje stavy mezi více žádostmi a více klienty.
- Popisuje standardní sadu architektonických vzorů pro webové aplikace.
- Je postaven na MVC architektuře.

JSF běží ve standardním web kontejneru, kterými mohou být *Tomcat* a *Jetty*. *JSF* se může použít s frameworkem *Spring MVC* využitím *ResourceHandlerWrapper*, který je mostem mezi *Spring MVC* a *JSF*. Třída přebírá případy, kdy *JSF* je použito jako pohled v řadiči *Spring MVC* a pro správné získání adres *JSF* šablon, *JavaScript* a *CSS*, jejichž *URL* může být generována pomocí *JSF*. Pokud by nebyla třída použita, *JSF* by nesprávně vygeneroval adresy v závislosti na umístění řadiče. [2]

Webové aplikace využívající *JSF* potřebují konfigurační soubor *faces-config.xml* a *web.xml*.

3.2.1 Konfigurační soubor: web.xml

JSF potřebuje centrální konfigurační soubor, což je *web.xml*, který je uložen v adresáři *WEB-INF* aplikace. Podobně jako je to u ostatních webových aplikací postavených na servletech. V konfiguračním souboru se musí definovat *FacesServlet*, který je zodpovědný za zacházení s *JSF* v aplikaci. *FacesServlet* je centrální řadič pro *JSF* využívající aplikaci. Přijímá všechny požadavky z *JSF* aplikace a inicializuje *JSF* komponenty před tím, než se *JSF* zobrazí. [2]

3.2.2 Konfigurační soubor: faces-config.xml

Poskytuje možnost konfigurovat následující:

- *Managed Bean* - datový element *JSF* aplikace, který je vytvářen dynamicky během běhu aplikace.

- navigace mezi stránkami
- validace dat - použita pro kontrolu vstupů od uživatele
- data konvertory pro převod dat mezi *UI* a modelem

Managed bean je Java třída (POJO), která je deklarována v *faces - config.xml* a může být použita v *JSF* aplikaci. Pro příklad se může definovat Java třída pro osobu. Jakmile se definuje objekt v *faces - config.xml*, mohou se použít atributy osoby v *UI* komponentech *JSF*, např. nastavit hodnotu atributu jméno. [2]

JSF používá *Unified Expression Language (EL)* pro vázání *UI* komponent s atributy objektu nebo metodami. K hodnotám *managed bean* můžeme přistupovat pomocí dvou metod a to buď `#{}` a končí `}`, nebo `#{}` a končí `}`. Výraz *EL* používající syntaxi `#{...}` je okamžitě vyhodnocen. Druhý případ `#{...}` je vyhodnocen pouze tehdy, když je použit, jinak je uložen jako řetězec. [2]

3.3 HTML Parser - Jsoup

Jsoup je knihovna napsána v jazyce Java pro práci s *HTML*. Poskytuje snadné rozhraní pro získání a manipulaci dat použitím *DOM* a *CSS* selektorů. *Jsoup* podporuje také *HTML5* specifikaci a zpracuje *HTML* stejně, jako moderní webové prohlížeče. Je navržen pro vypořádání se s všemi variantami *HTML*, které mohou být na webu použity a ze kterých je vygenerován snadno použitelný strom elementů. [5]

- Analyzovat *HTML* z *URL* adresy, souboru nebo řetězce
- Najít a vyjmout data pomocí procházení *DOM* nebo *CSS* selektorů
- Upravit *HTML* elementy, atributy a text

3.3.1 Požadavky

Aktuální verze je 1.7.2., která může být použita jako *jar* knihovna a nebo také umožňuje následující využití následující závislosti v projektu Maven.

```
<dependency>
  <!-- jsoup HTML parser library @ http://jsoup.org/ -->
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.7.2</version>
</dependency>
```

3.3.2 Příklady

Zpracování *HTML* stránky, která je uložena v proměnné jako řetězec, se použije statická metoda `Jsoup.parse(Stringhtml)`, nebo `Jsoup.parse(Stringhtml,StringbaseUri)` v případě že stránka je načtena ze sítě Internet a chceme získat absolutní adresy z elementů. [5]

```
String html = "<html><head><title>First ↵  
    ↵ parse</title></head><body><p>Parsed HTML into a ↵  
    ↵ doc.</p></body></html>";  
Document doc = Jsoup.parse(html);
```

Metoda `parse(Stringhtml,StringbaseUri)` zpracuje předaný *HTML* kód do objektu typu *Document*. Základní *URI* adresa je použita pro převod relativních adres na absolutní. Proměnná by měla být nastavena na adresu, ze které byl dokument použit. Pokud adresa není známa, nebo kód obsahuje definici ohledně *base* adres, stačí použít metoda `parse(Stringhtml)`. [5]

Z proměnné typu *Document* se poté mohou snadno získat potřebná data pomocí metod *Element*, nebo *Node*.

Podobným způsobem může *Jsoup* pracovat pouze s částí *HTML* kódu. Pro práci s částí kódu se použije `Jsoup.parseBodyFragment(Stringhtml)`. Funkce obalí fragment *body* elementem. Pokud je použita metoda `parse(Stringhtml,StringbaseUri)`, výsledek je stejný, pouze kód neobsahuje element *body*. [5]

```
String html = "<div><p>Lorem ipsum.</p>";  
Document doc = Jsoup.parseBodyFragment(html);  
Element body = doc.body();
```

Pro načítání *HTML* kódu ze souboru může být použita metoda `Jsoup.parse(Filein,StringcharsetName,StringbaseUri)`.

```
File input = new File("/tmp/input.html");  
Document doc = Jsoup.parse(input, "UTF-8", ↵  
    ↵ "http://example.com/");
```

Pro získání *HTML* kódu přímo z webové stránky slouží metoda `Jsoup.connect(Stringurl)`. Funkce vytvoří přípojení a v případě chyby vyhodí vyjímku *IOException*, která poté může být zpracována.

```
Document doc = Jsoup.connect("http://example.com/").get();  
String title = doc.title();
```

Pro nalezení elementů v kódu slouží seznam následujících metod:

- `getElementById(String id)`

- `getElementsByTag(String tag)`
- `getElementsByClass(String className)`
- `getElementsByAttribute(String key)` (and related methods)
- `siblingElements()`, `firstElementSibling()`, `lastElementSibling()`, `nextElementSibling()`, `previousElementSibling()`
- `parent()`, `children()`, `child(int index)`

Metody pro získání dat z elementů:

- `attr(String key)` pro získání `attr(String key, String value)` pro nastavení atributu
- `attributes()` získá seznam všech atributů
- `id()`, `className()` a `classNames()`
- `text()` pro získání `text(String value)` pro nastavení hodnoty elementy
- `html()` pro získání `html(String value)` pro nastavení vnitřního *HTML* kódu elementu
- `outerHtml()` pro získání okolního *HTML* elementu
- `data()` pro získání obsahu elementu jako je *script* a *style*
- `tag()` a `tagName()`

Následující ukázka kódu načte *HTML* kód ze souboru a nastaví základní adresu na `http://example.com/`, následně se extrahuje z dokumentu element, který má *id content*. Poté se najdou všechny odkazy v elementu *#content* a získají se jejich adresy a text. [5]

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", ↵
    ↵ "http://example.com/");
Element content = doc.getElementById("content");
Elements links = content.getElementsByTag("a");
for (Element link : links) {
    String linkHref = link.attr("href");
    String linkText = link.text();
}
```

3.4 JUnit testování

Část kódu je psána programátorem je Unit test, který je spuštěn se specifikovanou funkcionalitou v kódu, jenž chceme testovat. Pokrytí kódu a testovaného kódu Unit testem se vyjadřuje procentuálně. Unit testy ověřujeme, zda kód funguje podle požadavků v případě změny pro opravení chyb, nebo rozšíření funkcionality. Velké pokrytí testů kódu dovoluje pokračovat ve vývoji bez nutného manuální testování. Často jsou Unit testy vytvořeny jako samostatný projekt nebo samostatná složka, aby nebyly zaměněny nebo nijak neovlivňovaly celkový projekt. [12, 8]

3.4.1 Unit testy s JUnit

JUnit testy jsou nyní ve verzi 4.X. Jedná se o framework sloužící na testování a používá anotaci pro identifikaci metod, které specifikují samotný test. Každý test je metoda obsažena ve třídě, která je použita pouze pro testování. Často je nazývána jako *Test class*. [12]

```
@Test
public void testMultiply() {
    // Multiply is tested
    Math math = new Math();
    // Check if multiply(10, 5) returns 50
    assertEquals("10 x 5 must be 50", 50, math.multiply(10, ↵
        ↵ 5));
}
```

JUnit test předpokládá, že všechny metody testu mohou být spuštěny v libovolném pořadí, a že jednotlivé testy nezávisí na jiných testech. Pro psaní testu s JUnit se musí před metodu přidat anotace `@org.junit.Test`. Tím se zajistí, že metoda označená anotací slouží pro kontrolu požadovaného výsledku s testovacími daty. [12]

Vývojové prostředí Eclipse nabízí uživatelské rozhraní pro spuštění testů. Kliknutím pravým tlačítkem myši na třídu testu v podnabídce se vybere *Run* → *RunAs* → *JUnitTest*. Mimo Eclipse se může využít třída `org.junit.runner.JUnitCore` pro spuštění testů, nebo spustit testy pomocí příkazů *Maven*.

3.4.2 Anotace JUnit

@Test Anotace `@Test` identifikuje, že metoda je metodou testu.

@Before Metoda je provedena před každým testem. Tato metoda může sloužit pro předpřípravu prostředí.

@After Metoda je provedena po každém testu. Tato metoda může sloužit pro úklid prostředí.

@BeforeClass Metoda je spuštěna jednou, a to před všemi testy. Může být použita pro globální nastavení všech testů, například připojení k databázi. Metody označené touto anotací musí být definovány jako statické pro správnou funkčnost s JUnit.

@AfterClass Metoda je spuštěna jednou, a to poté, co všechny testy skončí. Může být použita pro celkový úklid, například odpojení od databáze. Metody označené touto anotací musí být definovány jako statické pro správnou funkčnost s JUnit.

@Ignore Metoda nebude použita během testování. Může být použita pro přeskočení testu, který ještě nemá být použit, nebo pro testy, které trvají příliš dlouhou dobu, aby byly zahrnuty (debug testů).

@Test(expected=Exception.class) Test skončí chybou v případě, že nevrátí požadovanou výjimku.

@Test(timeout=n) Test skončí chybou v případě, že trvá déle jak n ms.

3.4.3 Vyhodnocování testů

JUnit poskytuje statické metody v třídě *Assert* pro testování jednotlivých kritérií. Tyto metody umožňují zadat chybovou zprávu a specifikovat požadovaný výsledek a aktuální výsledek. V následujícím přehledu je seznam jednotlivých metod. [12]

fail(String) Skončí chybou. Může být použito pro kontrolu, zda část kódu je či není dosažena, nebo ověření vrácení chyby testu před implementací kódu.

assertTrue([message], boolean condition) Zkontroluje, zda podmínka odpovídá pravdě (True).

assertsEquals([String message], expected, actual) Zkontroluje, zda dvě hodnoty jsou stejné. Pro pole kontroluje referenci, nekontroluje, zda obsah pole je stejný.

assertsEquals([String message], expected, actual, tolerance) Zkontroluje, zda čísla s plovoucí desetinou čárkou jsou stejná. Parametr *tolerance* je číslo, udávající přesnost za desetinou čárkou.

serNull([message], object) Zkontroluje, zda objekt je *NULL*.

assertNotNull([message], object) Zkontroluje, zda objekt není *NULL*.

assertSame([String], expected, actual) Zkontroluje, zda obě proměnné odkazují na stejný objekt.

assertNotSame([String], expected, actual) Zkontroluje, zda obě proměnné neodkazují na stejný objekt.

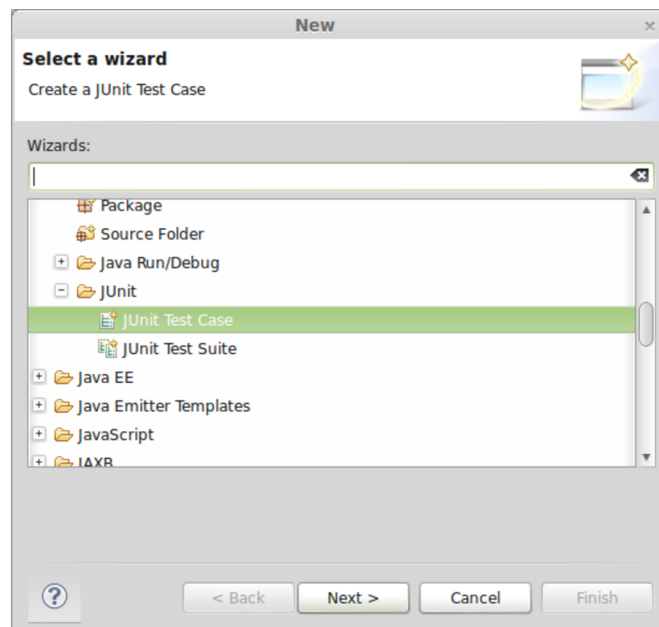
3.4.4 JUnit test suite

Pokud existuje několik testovacích tříd, je možná je kombinovat do jednoho *testsuite*. Spuštěním *testsuite* se spustí všechny testy, obsažené v tomto balíku. Následující ukázka kódu ukazuje *testsuite*, který definuje dvě testovací třídy které měly být spuštěny. Pro přidání další testovací třídy stačí přidat `@Suite.SuiteClasses`. [12]

```
@RunWith(Suite.class)
@SuiteClasses({MyClassTest.class, MySecondClassTest.class})
public class AllTests {
    // ...
}
```

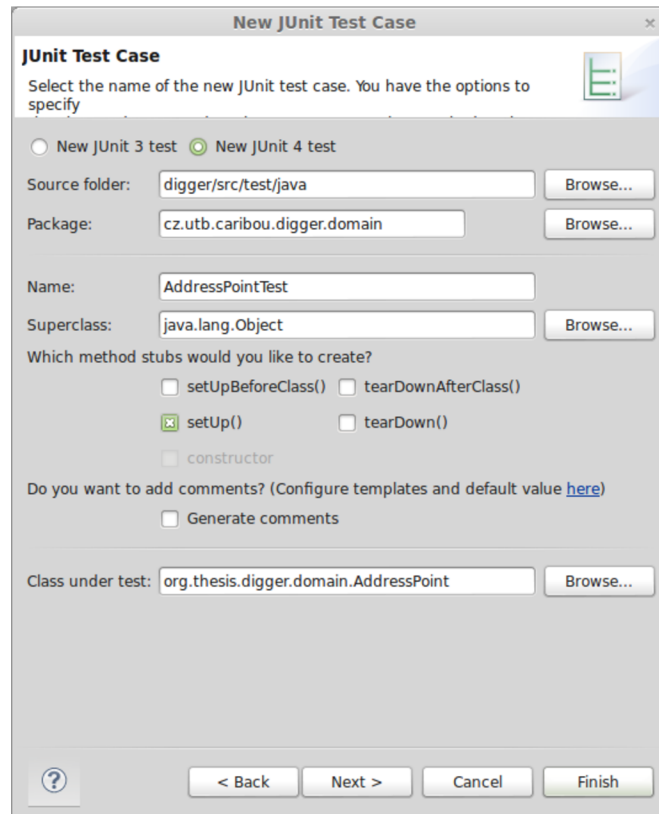
3.4.5 JUnit v Eclipse

Pravým kliknutím na novou třídu v *PackageExplorer* a poté zvolením *New* → *JUnitTestCase*, nebo vybráním JUnit testu v dialogovém okně *New*.



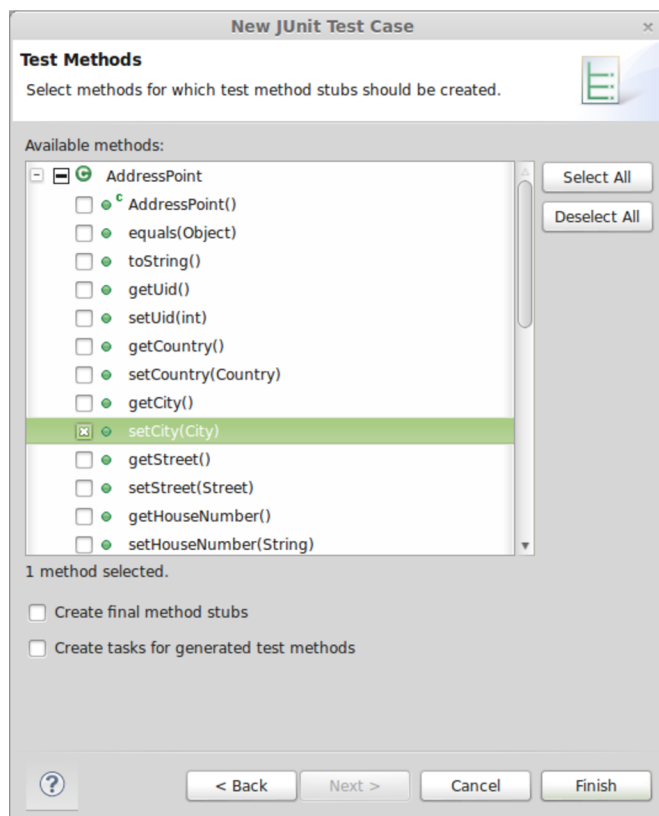
Obrázek 4. Vytvoření nového Test Case

V následujícím okně průvodce musí být označen *NewJUnitAtest* a nastavena správná složka pro uložení nové testovací třídy.



Obrázek 5. Nastavení nového Test Case

V dalším kroku se vybírají metody, které chceme testovat v novém testovací třídě.

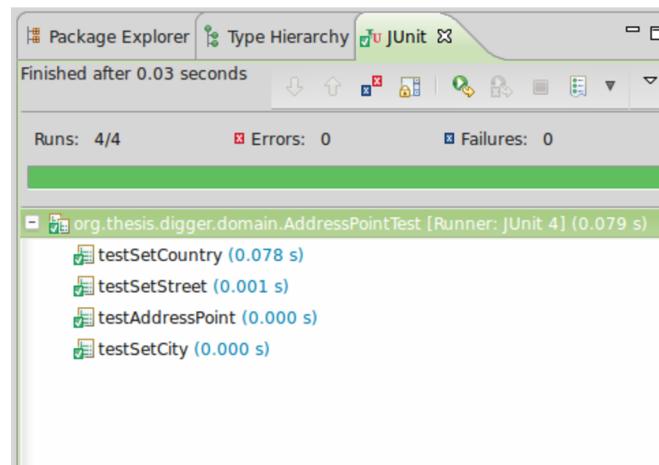


Obrázek 6. Vybrání testovaných metod

Eclipse automaticky vygeneruje šablonu testovací třídy podle zadaných kritérií.

```
public class AddressPointTest
{
    @Before
    public void setUp() {
        fail("Not implemented");
    }
    @Test
    public void testAddressPoint() {
        fail("Not implemented");
    }
    @Test
    public void testSetCountry() {
        fail("Not implemented");
    }
}
```

Po implementaci testovacího kódu a spuštění testů Eclipse zobrazí výsledek testovací třídy. Všechny testy na obrázku 7 prošly, což je znázorněno zeleným status barem a nulovým počtem *Errors* a *Failures*.



Obrázek 7. Výsledek Test Case

3.4.6 Parametrizovaný test

JUnit testy podporují použití parametru v testovacích třídách. Tato třída může obsahovat jednu nebo více metod, které jsou spouštěny s jinými parametry. Třída se označí jako parametrizovaná `@RunWith(Parameterized.class)` anotací.

Taková testovací třída musí obsahovat statickou metodu označenou `@Parameters`. Tato metoda generuje a vrací kolekci polí. Každý prvek této kolekce je použit jako parametr pro testovanou metodu.

Také se musí vytvořit konstruktor, ve kterém se budou ukládat hodnoty pro každý test. Počet prvků v každém poli, poskytovaném metodou označené `@Parameters`, musí odpovídat počtu parametrů v konstruktoru třídy. Třída se vytváří pro každý parametr a zkušební hodnoty jsou předány prostřednictvím konstruktoru třídy.

Následující ukázka kódu ukazuje příklad pro parametrizovaný test u metody násobení, která byla použita výše. Test vytvoří pole čísel 1, 5, 121, která se postupně předávají pomocí konstruktoru do členské proměnné `multiplier`, a poté se testuje jejich součin v `@Test` metodě. [12]

```
@RunWith(Parameterized.class)
public class MathParameterizedClassTest {
    private int multiplier;
    public MathParameterizedClassTest(int testParameter) {
        this.multiplier = number;
    }
    // Creates the test data
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] {{1}, {5}, {121}};
        return Arrays.asList(data);
    }
}
```

```
}
@Test
public void testMultiplyException() {
    Math math = new Math();
    assertEquals("Result", multiplier * multiplier,
        math.multiply(multiplier, multiplier));
}
}
```

3.4.7 Pravidla

Pomocí anotace `@Rule` můžeme vytvořit objekt, který dokáže být použit a být nastaven v testovací metodě. Následující ukázka kódu specifikuje, která zpráva výjimky bude očekávána během spuštění. [12]

```
public class RuleExceptionTesterExample {
    @Rule
    public ExpectedException exception = ↵
        ↵ ExpectedException.none();
    @Test
    public void throwsIllegalArgumentExceptionIfIconIsNull() {
        exception.expect(IllegalArgumentException.class);
        exception.expectMessage("Negative value not allowed");
        ClassToBeTested t = new ClassToBeTested();
        t.methodToBeTest(-1);
    }
}
```

JUnit poskytuje několik užitečných implementací pravidel. Pro vlastní napsání svého pravidla se musí použít `TestRule` interface. Další příklad zobrazuje, jak třída `TemporaryFolder` umožní nastavit soubory a složky, které budou automaticky odstraněny po dokončení testu. [12]

```
public class RuleTester {
    @Rule
    public TemporaryFolder folder = new TemporaryFolder();
    @Test
    public void testUsingTempFolder() throws IOException {
        File createdFolder = folder.newFolder("newfolder");
        File createdFile = folder.newFile("myfilefile.txt");
        assertTrue(createdFile.exists());
    }
}
```

4 POSTGRESQL

PostgreSQL je výkonná open source objektově relační databáze. Má za sebou více než 15 let vývoje a ověřenou architekturu, která získala výraznou pověst pro svou spolehlivost, integritu dat a pro správnost. Může být instalována na všech operačních systémech, včetně Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), a Windows. Je plně ACID kompatibilní a má plnou podporu cizích klíčů, spojení, pohledů, triggerů a procedur, které mohou být psány ve více jazycích.

Zahrnuje většinu SQL:2008 datových typů, jako jsou *INTEGER*, *NUMERIC*, *BOOLEAN*, *CHAR*, *VARCHAR*, *DATE*, *INTERVAL* a *TIMESTAMP*, a také podporuje ukládání binárních objektů, jako obrázky, hudba nebo video. Má nativní programové rozhraní pro C/C++, Java, .Net, Perl, Python, Ruby, Tcl a ODBC.

Jako enterprise databáze PostgreSQL se může pochlubit sofistikovanými funkcemi, mezi které patří Multi-Version Concurrency Control (MVCC), obnovy podle časového bodu, asynchronní replikace, vnořené transakce (body návratu), on-line/horké zálohy, sofistikovaný plánovač dotazů, optimalizér a psaní před protokolování pro odolnost proti chybám.

Podporuje mezinárodní znakové sady, vícebajtové kódování znaků Unicode, velká a malá písmena a formátování. Je vysoce škálovatelná a to ve velkém množství dat, která mohou být řízena vysokým počtem uživatelů ve stejnou dobu. PostgreSQL systémy v produkčních prostředích mohou spravovat více než 4TB dat. [10]

4.1 Použití triggerů

Trigger je speciální funkce, která se váže na tabulku, nebo na pohled definovanou událostí. PostgreSQL disponuje dvěma typy triggerů - statement-trigger a row-level trigger. Pro upřesnění spustí dotaz, který ovlivní 1500 řádků. První typ triggeru se spustí pouze jednou, zato druhý typ se spustí pro každý ovlivněný řádek.

Typy triggerů se také liší podle času spuštění. Můžeme definovat trigger, který je spuštěn před provedením příkazu *BEFORE* nebo poté, co se příkaz provede - *AFTER*. Tyto dvě události mohou být provedeny pouze pro tabulky. Pro pohledy se používá *INSTEADOF* událost, která se provede místo normální akce.

Trigger je vázán na akci, která se provádí s tabulkou nebo s pohledem. Akce mohou být *INSERT*, *UPDATE* a nebo *DELETE*.

Pro definici triggerů v PostgreSQL se používají speciální funkce, které nemají žádné vstupní parametry a vrací pouze speciální typ *trigger*. Triggery se definují jako standardní funkce a proto je možnost použít více než jeden trigger. Pokud je použito více triggerů pro jednu tabulku nebo pohled, tak jsou spouštěny podle abecedního pořadí názvu triggerů.

Triggery mohou být psány i v jiném jazyce než *PL/pgSQL* a to například *Python* při použití jazyka *PL/Python*. [10]

Následující trigger je napsán v jazyce *PL/pgSQL* a je vázán na tabulku *user_account*. Při přidání nového řádku, nebo editaci v případě, že heslo (sloupec *password*) není zašifrováno, provede funkci *crypt*, která heslo zašifruje.

```
CREATE OR REPLACE FUNCTION trg_crypt_users_pass()
RETURNS TRIGGER AS
$BODY$
DECLARE
BEGIN
    IF substr(NEW.password, 1, 3) <> '$1$' THEN
        NEW.password := crypt(NEW.password, gen_salt('md5'));
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';
CREATE TRIGGER trg_crypt_users_pass BEFORE
INSERT OR UPDATE ON "user_account"
FOR EACH ROW EXECUTE PROCEDURE trg_crypt_users_pass();
```

Pro snadné zpracování dat z API, která mohou být ve formátu *JSON* se může v triggeru použít i jazyk *PL/Python*. Trigger níže je ukázkou pro získání a doplnění geografických souřadnic pomocí Google API.

```
CREATE FUNCTION trg_get_location() RETURNS trigger AS $$
try:
    import urllib
    import json
    api_url = 'http://maps.google.com/maps/api/geocode/
    json?address={city}&components=postal_code:{postal_code}'
    jsonurl = urllib.urlopen(api_url.format(city = ↵
    ↵ TD['new']['name'], postal_code = TD['new']['zip']))
    json_data = json.loads(jsonurl.read())
    location = ↵
    ↵ json_data['results'][0]['geometry']['location']
    TD['new']['location'] = 'Point({lng} ↵
    ↵ {lat})'.format(**location)
    return 'MODIFY'
except:
    return None
$$ LANGUAGE plpythonu;
CREATE TRIGGER trg_get_location BEFORE
INSERT OR UPDATE ON "city"
FOR EACH ROW EXECUTE PROCEDURE trg_get_location();
```

4.2 PostGIS

PostGIS je rozšíření pro databázi PostgreSQL, které přidává možnost práce s geografickými objekty. PostGIS nabízí mnoho funkcí zřídka se nacházejících v jiných konkurenčních databázích jako je Oracle Locator/Spatial a SQL Server. PostGIS vychází z OpenGIS *Simple Features Specification* pro SQL a byl oceněn *Types and Functions*.

Vývoj PostGIS začal jako *Refractions Research* pod licencí open source projekt pro prostorové databáze (spatial database). Komunitou vyvíjející PostGIS přidává nové funkce pod GNU licencí. Aktuální verze PostGIS je 2.0. [9]

4.2.1 Geometrické objekty

Následující SQL příkaz vytvoří tabulku *geometries*, která obsahuje jméno, v našem případě datového typu, a geografickou pozici uloženou pomocí datového typu *geometry*. Tabulka je poté vyplněna různými geometrickými typy, jako jsou bod, přímka, polygon a kolekce geometrických objektů. [11]

```
CREATE TABLE geometries (
  name varchar,
  geom geometry
);
-- insert some data
INSERT INTO geometries VALUES
('Point', 'POINT(0 0)'),
('Linestring', 'LINESTRING(0 0, 1 1, 2 1, 2 2)'),
('Polygon', 'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))'),
('PolygonWithHole', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),
(1 1, 1 2, 2 2, 2 1, 1 1))'),
('Collection', 'GEOMETRYCOLLECTION(POINT(2 0),
POLYGON((0 0, 1 0, 1 1, 0 1, 0 0)))');
```

Vývoj *PostGIS* se zaměřuje na definování reálných objektů pomocí popisu jejich tvarů a fixního rozlišení, čímž získáme reprezentaci objektů. Standardní *SQL* implementace pracuje pouze s dvourozměrnými objekty, ale při použití *PostGIS* můžeme vytvořit až čtyřrozměrné reprezentace objektů. [11]

Pro získání základních informací ohledně každého objektu se mohou použít následující funkce.

- *ST_GeometryType(geometry)* vrátí typ geometrického objektu
- *ST_NDims(geometry)* vrátí počet rozměrů (dimenzí) objektu

Bod - Point Bod představuje jediné místo na Zemi. Tento bod je specifikován jednou souřadnicí (včetně obou dvou, tří nebo čtyř rozměrů). Body se používají k reprezentaci objektů, kdy přesné údaje, jako je tvar a velikost, nejsou důležité v cílovém rozsahu.

Například města na mapě světa mohou být popsána jako body, zatímco mapa jednoho státu může představovat města pomocí polygonů. [11]



Obrázek 8. Reprezentace bodů v PostGIS [11]

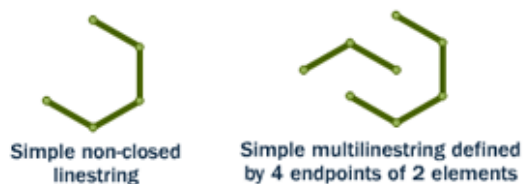
Funkce pro práci s typem *Point*:

- $ST_X(geometry)$ vrátí X souřadnici
- $ST_Y(geometry)$ vrátí Y souřadnici

Tedy pro získání souřadnic se může použít následující dotaz.

```
SELECT ST_X(geom), ST_Y(geom) FROM geometries WHERE name = ↵
↵ 'Point';
```

Přímka - Linestrings Linestrings může definovat cestu z jednoho místa do druhého a má podobu uspořádané série dvou, či více bodů. Silnice a řeky jsou obvykle zastoupeny jako *linestrings*. Datový typ *linestring* je uzavřen pokud začíná a končí ve stejném bodě. Pokud se přímka dotýká, nebo se kříží v určitém bodě, nedojde k jejímu uzavření. [11]



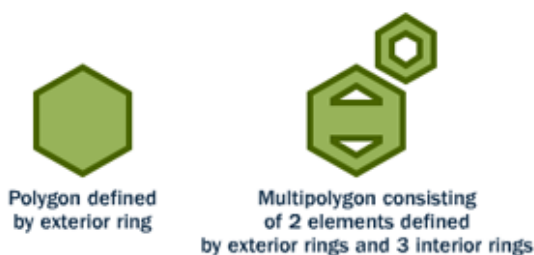
Obrázek 9. Reprezentace přímky v PostGIS [11]

Například adresy v městě mohou být popsány několika *linestrings*, kde každý segment cesty bude definován jinými parametry.

- $ST_Length(geometry)$ vrátí délku přímky
- $ST_StartPoint(geometry)$ vrátí první souřadnici přímky
- $ST_EndPoint(geometry)$ vrátí poslední souřadnici přímky
- $ST_NPoints(geometry)$ vrátí počet souřadnic přímky

Mnohostěn - Polygon Polygon popisuje oblast. Jeho vnitřní hranice je reprezentována pomocí prstence¹⁾. Tento prstenec se skládá z uzavřeného *linestring*. Vykrojení děr do polygonu je také definováno jako prstenec. [11]

Polygon je použit u objektů s podmínkou definovaných tvarů a rozměrů. Města, parky, části měst, atd. jsou často reprezentovány pomocí polygon, kde měřítko je dostatečně vysoké, aby mohly být popsány pomocí oblastí. Řeky a cesty mohou být rovněž v některých případech popsány pomocí tohoto datového typu. [11]

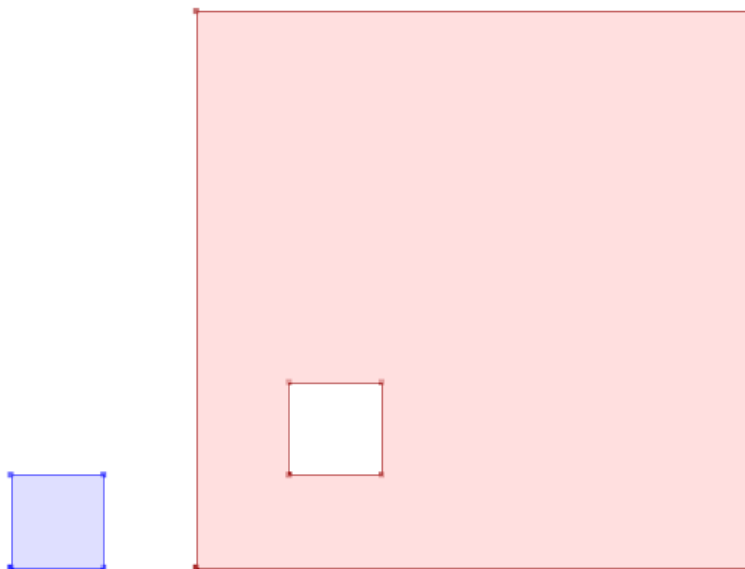


Obrázek 10. Reprezentace polygomu v PostGIS [11]

Následující ukázka definuje čtverec o rozměru stran 1. Druhý polygon je čtverec o rozměru stran 10 s vykrojeným čtvercem na pozici [1, 1] až [2, 2]. Výsledný geometrický objekt je znázorněn v obrázku 11.

```
POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))
POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),
(1 1, 1 2, 2 2, 2 1, 1 1))
```

¹⁾Prstenec v tomto případě není brán jako kružnice, ale uzavřený objekt, přeloženo z anglického originálu „ring“.



Obrázek 11. Ukázka polygonu [11]

- $ST_Area(geometry)$ vrátí oblast polygonu
- $ST_NRings(geometry)$ vrátí počet prstenců (často jedna, pokud polygon neobsahuje díry)
- $ST_ExteriorRing(geometry)$ vrátí okolní prstenec jako *linestring*
- $ST_InteriorRingN(geometry, n)$ vrátí konkrétní hranu jako *linestring*
- $ST_Perimeter(geometry)$ vrátí obvod všech prstenců

4.2.2 Kolekce - Collections

Existují celkem čtyři typy kolekce, které seskupují více geometrických objektů do množiny, např. pro popis dvou pozemků, které jsou rozděleny cestou.

- *MultiPoint* kolekce bodů
- *MultiLineString* kolekce přímek
- *MultiPolygon* kolekce polygonů
- *GeometryCollection* různorodé kolekce geometrických objektů

Funkce pro práci s kolekci geometrických objektů.

- $ST_NumGeometries(geometry)$ vrátí počet částí kolekce

- $ST_GeometryN(geometry, n)$ vrátí konkrétní část
- $ST_Area(geometry)$ vrátí celkovou oblast všech polygonů
- $ST_Length(geometry)$ vrátí celkovou délku všech lineárních částí

4.2.3 Geografické objekty

Časté jsou k dispozici údaje, ve kterých jsou zeměpisné souřadnice nebo zeměpisná délka a šířka použita. Narozdíl od souřadnic Mercator, UTM nebo Stateplane, geografické souřadnice nejsou stejné jako kartézské souřadnice. Zeměpisné souřadnice nepředstavují lineární vzdálenost. Tyto sférické souřadnice popisují souřadnice úhlové na Zemi. Ve sférických souřadnicích je bod určen pomocí úhlu otáčení od referenčního poledníku (zeměpisná délka) a úhel od rovníku (zeměpisná šířka). [11]

Operace se zeměpisnými souřadnicemi jsou podobné jako s geometrickými souřadnicemi a mohou to být průnik, „obsahuje“, vzdálenost mezi body, nebo objekty atd.

Pro příklad zde jsou souřadnice měst Paříž a Los Angeles.

- Los Angeles: POINT(-118.4079 33.9434)
- Paris: POINT(2.3490 48.8533)

Následující výpočet vzdálenosti mezi Los Angeles a Paříží ukazuje použití standardní PostGIS kartézské funkce $ST_Distance(geometry, geometry)$. Hodnota bude předána jako geografická souřadnice *POINT*.

```
SELECT ST_Distance(
  ST_GeographyFromText('POINT(-118.4079 33.9434)'), -- Los ↗
  ↙ Angeles (LAX)
  ST_GeographyFromText('POINT(2.5559 49.0083)') -- ↗
  ↙ Paris (CDG)
);
```

Výsledek operace je číslo 9124665.26917268. Všechny geografické operace pracují se soustavou SI a vrácená délka je v metrech, tedy vzdálenost mezi Los Angeles a Paříží je 9124.665Km.

Za účelem uložení geometrických dat do tabulky geometry je třeba nejdříve překonvertovat data do EPSG: 4326 (zeměpisná délka/šířka). Funkce $ST_Transform(geometry, srid)$ převede geometrické souřadnice na geografické.

```
CREATE TABLE nyc_subway_stations_geog AS
SELECT
  Geography(ST_Transform(geom, 4326)) AS geog,
  name,
  routes
FROM nyc_subway_stations;
```

II. PROJEKTOVÁ ČÁST

5 PROGRAMOVÝ MANUÁL

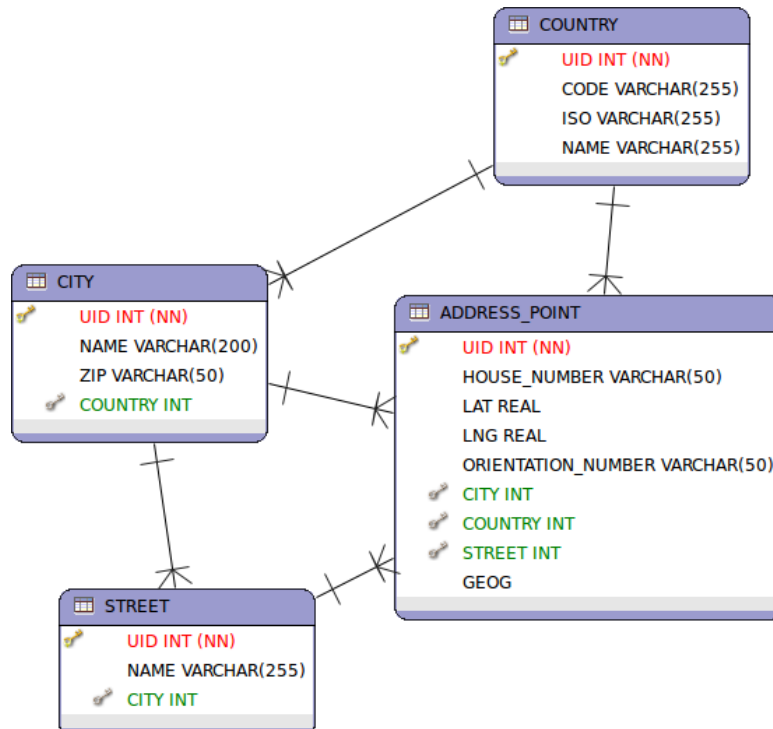
V diplomové práci jsem vytvářel dva projekty *digger* a *webapp*. Oba jsou napsány v jazyce Java. Projekty jsou zapouzdřeny v baličku *cz.utb.caribou* a jsou postaveny na kompilačním nástroji *Maven*. Pro správu zdrojových kódů jsem použil nástroj *Git*.

5.1 Databáze

Pro databázi používám PostgreSQL verzi 9.2 s doplňkem PostGIS pro ukládání geografických dat. Všechny tabulky jsou pojmenovány podle prvního pádu jednotného čísla a pro oddělovač mezi slovy je použito podtržítka (*address_point*, *tender*), stejně tak u názvu sloupců tabulek. Pouze u názvů spojovacích tabulek je použito množné číslo prvního pádu (*tender_items*).

Schéma tabulky *address_point* zobrazuje obrázek 12. Sloupce tabulky se odkazují pomocí cizích klíčů na tabulky obsahující země, ulice a města. Sloupec v tabulce *country* nazván *code* je zkratka státu podle *ISO 3166-1 alpha-2* normy (CZ, CA, ...).

Tabulka *address_point* obsahuje geografickou pozici každého bodu, která se získává pomocí *Google API*, a uložení poštovního směrovacího čísla a čísla popisného.



Obrázek 12. Diagram struktury adres bodu

Na obrázku 5.1 je znázorněn diagram tabulky *tender*. Tabulka *tender* se odkazuje na *organizaci* a ta obsahuje seznam všech zadavatelů. Tabulka *legal_form*, neboli právní forma zadavatele, je jeden z číselníků, další číselníky (např. NUTS, CPV, ..) v této verzi databáze nejsou použity z důvodu rozdílného zobrazení na jednotlivých tržištích. Číselníky budou doplněny po získání většího množství dat.

Tabulka *tender_market* upřesňuje, z jakého tržiště byla veřejná zakázka stažena. *tender* obsahuje kompozitní unikátní klíč - *tender_market* a *tender_market_uid*. Druhý sloupec v klíči je hodnota ID v tržišti.

Použití relace m:n, pro předměty a části veřejné zakázky (tabulky *item* a *part*) je použito pro omezení stejných předmětů v tabulce.

5.2 Digger

Digger je konzolová aplikace, která slouží pro stahování veřejných zakázek z tržišť. Jednotlivé veřejné zakázky ukládá do databáze PostgreSQL pomocí frameworku Hibernate a *JPA Entity Manageru*. Aplikace je vyvinuta pro jednorázové spouštění (cron job), a to bez jakýchkoliv parametrů.

Hlavní třída má název *Digger* a inicializuje třídu *CrawlerManager*, která obsahuje kompletní logiku stahování veřejných zakázek a parser pro každé tržiště. Jednotlivé parsery jsou přidány do třídy *CrawlerManager* jako seznam, který se postupně prochází a přidává parsované zakázky do databáze. *CrawlerManager* reportuje dva typy chyb.

První typ je chyba vrácena parserem, tedy pokud parser narazí na políčko, které neví, do jakého sloupce v databázi ho zařadit, a nebo došlo k chybě v serializaci dat. Chyba serializace je například špatný formát datumu, např. YYYY-MM-DD místo YY-MM-DD. Tento typ chyby není kritický, ale popisuje stabilitu parseru a to, jak je schopen reagovat na jednotlivé údaje ve veřejné zakázce na tržišti. Stav chyby se ukládá do textového souboru, kde je uložena adresa veřejné zakázky, která chybu vyvolala a popis chyby.

Druhý typ chyby je v samotném *CrawlerManager*. Tento typ chyby je závažnější, protože ovlivňuje všechny parsery. Stav se také ukládá do textového souboru, ale veřejná nabídka během níž chyba nastala, se ukládá jako serializovaný objekt pro rychlejší opětovné zpracování. Chyba může nastat i v případě kolize identických unikátních klíčů, např. pokusu o vložení stejné zakázky do databáze.

5.2.1 Použité API

Pro získání více informací nebo jejich opravení využívá aplikace veřejných API. Třídy jsou umístěny v balíčku *cz.utb.caribou.digger.api* a v případě chyby vyhazují výjimku *ApiException*.

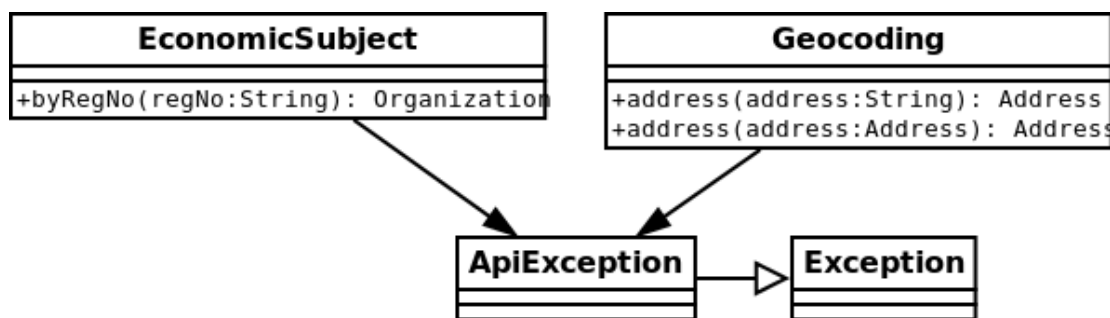
Google Geocoding API ²⁾ Třída získává geografická data z Google Geocoding API v3. API a je použita pro každou adresu, která se ukládá do databáze pro dohledání geografických souřadnic a opravení údajů, které by zvedaly redundanci v databázi. Každé tržiště používá jiný způsob vkládání adres a je možné stejnou adresu zadat i v jiné formě, např. „Václavské náměstí“ a „Václavské nám.“ udává stejné místo, ale dva různé názvy. Použitím této API je každá adresa převedena na stejný formát a poté uložena do databáze.

²⁾<https://developers.google.com/maps/documentation/geocoding/>

Omezení API je pouze 2500 dotazů za den. V případě, že limit je dosažen, třída vyhodí *ApiException*.

ARES - Administrativní registr ekonomických subjektů ³⁾ Tržišť veřejných zakázek nezobrazují všechny informace zadavatelů. Pro podrobnější informace se používá ARES Ministerstva financí. Během parsování z veřejných zakázek se získá pouze IČO, které je použito pro dohledání dalších informací týkajících se zadavatele.

Limit na počet dotazů se liší podle doby, kdy byl dotaz odeslán, a to na 1000 dotazů od 8:00 do 18:00 a na 5000 dotazů během noci. Také hrozí zablokování služby při posílání stejných nebo náhodných dotazů.



Obrázek 14. Diagram tříd pro API

5.2.2 Parsování tržišť a veřejných zakázek

CrawlerManager je hlavní třída parseru. Řídí její kompletní chod a pořadí spuštění jednotlivých parserů. Metoda *run* prochází jednotlivé parsery a v první řadě zpracovává tržiště a získá seznam adres zakázek, které přibyly od posledního spuštění aplikace. Po získání seznamu se začne zpracovávat seznam nových zakázek, které jsou posléze uloženy do databáze.

Každé tržiště má definovanou třídu, která je potomkem třídy *GenericParser*. Tato třída má dvě virtuální metody *queue(String)* a *content(string)*. První metoda slouží pro parsování tržiště a získání seznamu všech nových veřejných zakázek, které jsou přidávány do objektu *TenderContentQueue* jako seznam webových adres.

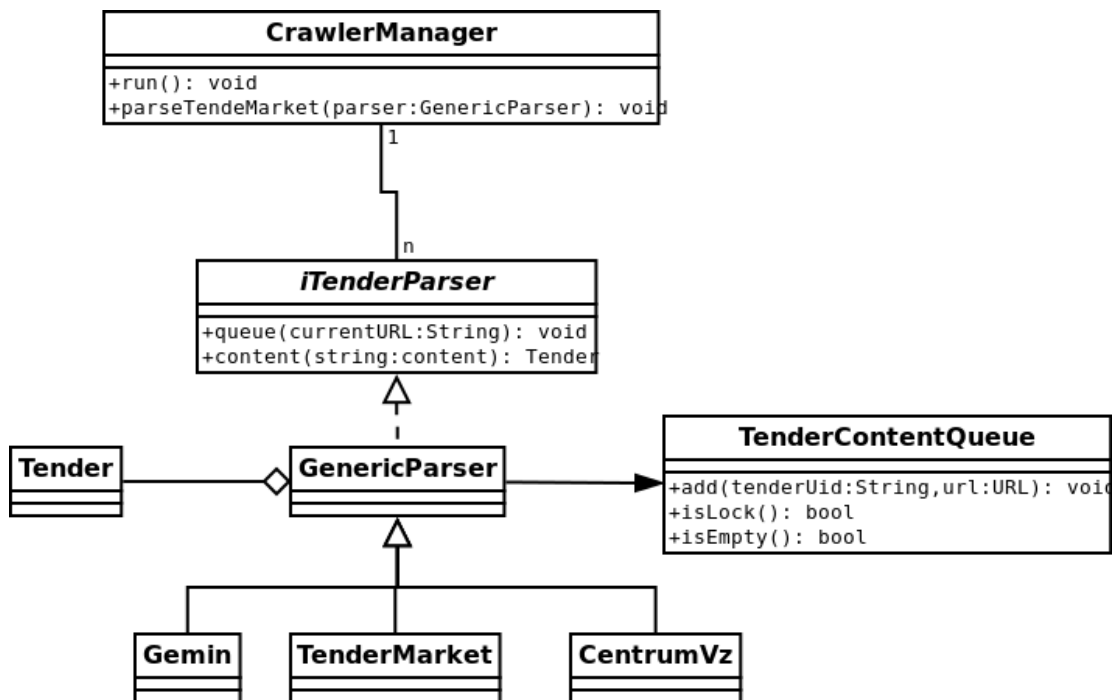
Metoda pro získání seznamu veřejných zakázek musí také nastavit tzv. aktuální stránku se seznamem veřejných zakázek. Pokud není nastavena, tak se po čtení seznamu parser

³⁾<http://www.info.mfcr.cz/ares/ares.html.cz>

nedostane na další stránku zakázek, aby nedošlo k uložení některých zakázek.

Druhá metoda slouží pro parsování jedné veřejné nabídky. Webová stránka s veřejnou nabídkou je předána z *CrawlerManager* jako řetězec, který je zpracován a pomocí *Jsoup* se získají její jednotlivé údaje. Poté funkce vrací objekt typu *Tender*, který je poté uložen do databáze v *CrawlerManager*.

TenderContentQueue je třída obsahující seznam adres veřejných zakázek, které byly načteny během parsování tržišť. Tento seznam se poté prochází a získávají se potřebné veřejné zakázky. Při přidání nové zakázky do seznamu se ověří, zda není zakázka se stejnou adresou a jejím jedinečným číslem z tržiště již uložena v databázi. Pokud je uložena, nastaví se vnitřní zámek metody a ukončí se parsování tržiště na straně *CrawlerManager*. Metoda pro získání uložené adresy vrátí první vloženou zakázku a poté ji vymaže ze seznamu, takže objekt funguje jako zásobník.



Obrázek 15. Diagram tříd pro parsování zakázek

5.3 Webová aplikace

Webová aplikace je napsána v jazyce Java a používá framework *Spring*. Pro přístup k databázi stejně jako *Digger* slouží framework *Hibernate*. Aplikace je postavena na návrhovém vzoru *MVC*, konkrétně *Spring MVC* framework, kde pohledová vrstva aplikace je *JSF* framework a vrstva modelu je *POJO* a *JPA*.

HTML je psáno podle standardu páté verze a *CSS3*. Pro snadnější stylování stránky používám *Twitter Bootstrap*, který je volně stažitelný a použitelný i na komerčních projektech. *Twitter Bootstrap* obsahuje mnoho JavaScriptových komponent napsaných v *JQuery* pro rychlejší vývoj. Mezi komponenty patří modální okna, různé dialogy, titulky, rozbalovací nabídky atd. Také obsahuje styly pro tlačítka, sadu ikon a snadné nastavení šablony webu, které jsou optimalizovány pro stolní počítače, tablety i mobilní telefony.

5.3.1 Řadiče

Každý řadič, neboli *Controller* je potomkem *BaseController*, který inicializuje *@ModelAttribute*. Tento atribut je vytvořen tehdy, když uživatel není přihlášen a není relace uložena v *Session*. V opačném případě si řadič sám dokáže najít již vytvořený atribut modelu. Řadiče mají podobnou strukturu a pomocí anotace *@RequestMapping* definují adresu, pod kterou se daná metoda volá.

```
@SessionAttributes({ "userAccountSession" })
public class BaseController
{
    @ModelAttribute("userAccountSession")
    public UserAccountSession getUserAccountSession() {
        return new UserAccountSession();
    }
}
```

Atribut modelu je předáván parametrem každé metody a je automaticky frameworkem *Spring* doplněn. Následující kód ukazuje je řadič pro zobrazení pohledu organizací. První metoda *list* zobrazí seznam uložených organizací do pohledu *organization list*. Druhá metoda zobrazí detail organizace se všemi jejími veřejnými zakázkami.

```
@Controller
public class OrganizationController extends BaseController
{
    @RequestMapping("/organization")
    public ModelAndView list(@ModelAttribute(value = ↵
        ↵ "userAccountSession") UserAccountSession ↵
        ↵ userAccountSession) {
        ModelAndView mv = new ModelAndView("organization/list");
        mv.addObject("user", userAccountSession);
        OrganizationDao organizationDao = new OrganizationDao();
        mv.addObject("organizationList", ↵
            ↵ organizationDao.getList());
        return mv;
    }
    @RequestMapping("/organization/{organizationUid}")
    public ModelAndView detail(@ModelAttribute(value = ↵
        ↵ "userAccountSession") UserAccountSession ↵
        ↵ userAccountSession, @PathVariable int ↵
        ↵ organizationUid) {
        ModelAndView mv = new ↵
            ↵ ModelAndView("organization/detail");
    }
}
```

```

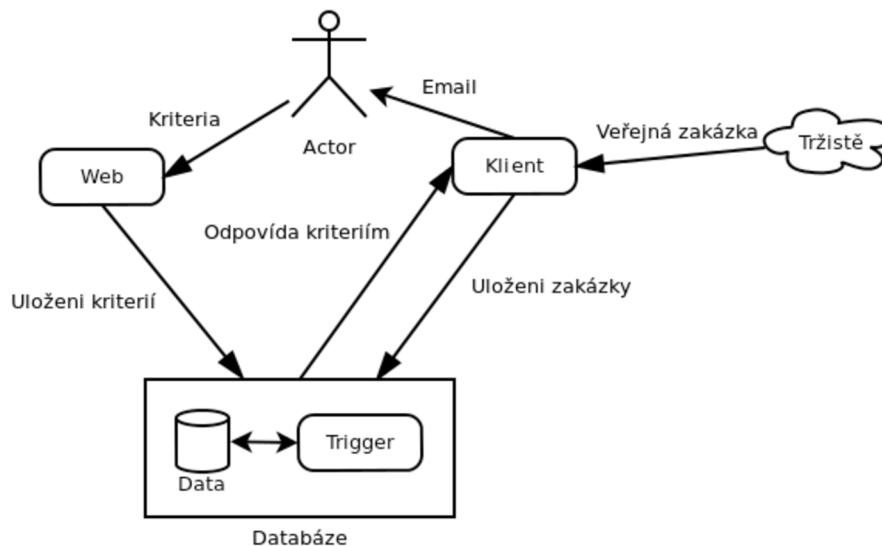
mv.addObject("user", userAccountSession);
OrganizationService organizationService = new ↵
    ↵ OrganizationService();
Organization organization = ↵
    ↵ organizationService.searchById(organizationUid);
TenderService tenderService = new TenderService();
mv.addObject("organization", organization);
mv.addObject("tenderList", ↵
    ↵ tenderService.searchBy(organization));
return mv;
}
}

```

5.3.2 Vyhledávač

Vyhledávací formulář slouží pouze pro vyhledání ve veřejných zakázkách, tedy není implementován vyhledávač pro organizace. Vyhledávací kritéria jsou název veřejné zakázky, druh zakázky, název zadavatele, IČO zadavatele a rozsah ceny. Kritéria jsou předána do metody třídy *TenderService* a poté jsou nalezeny odpovídající zakázky v databázi.

Logika nalezení veřejných zakázek podle uložených kritérií je navržena čistě na straně databáze. Kritéria se uloží do tabulky a po každém vložení nové zakázky je spuštěn trigger, který spáruje nově přidané zakázky s kritérii uživatele. Tato dvojice je pak uložena do tabulky fronty, která je poté klientem rozesílána a dvojice označena jako splněna. Schéma uloženého hledání je znázorněno na následujícím obrázku.



Obrázek 16. Schéma uloženého hledání

5.3.3 Bezpečnost

Databáze uchovává hesla k účtům každého registrovaného uživatele. Tato hesla by se z bezpečnostních důvodů neměla ukládat v textové podobě jako byla obdržena od uživatelů, v případě exportu a zálohy databáze, která se ukládá do textového souboru, nebo při vniknutí nepovolenou osobou do databáze. Pro ukládání hesla používám trigger, jenž byl popsán v kapitole o triggerech.

Při vkládání, nebo editaci nových záznamů do tabulky s uživateli jsou hesla uložena bezpečně a není je možné zjistit bez tzv. *Rainbow table*. Pro nalezení konkrétního uživatele můžeme použít následující příkaz.

```
SELECT * FROM user_account WHERE username = 'radek' AND ↵
    ↵ crypt('heslo', passwd) = passwd;
```

Pro zjednodušení tohoto příkazu jsem si vytvořil operátor pro definovaný datový typ *password*.

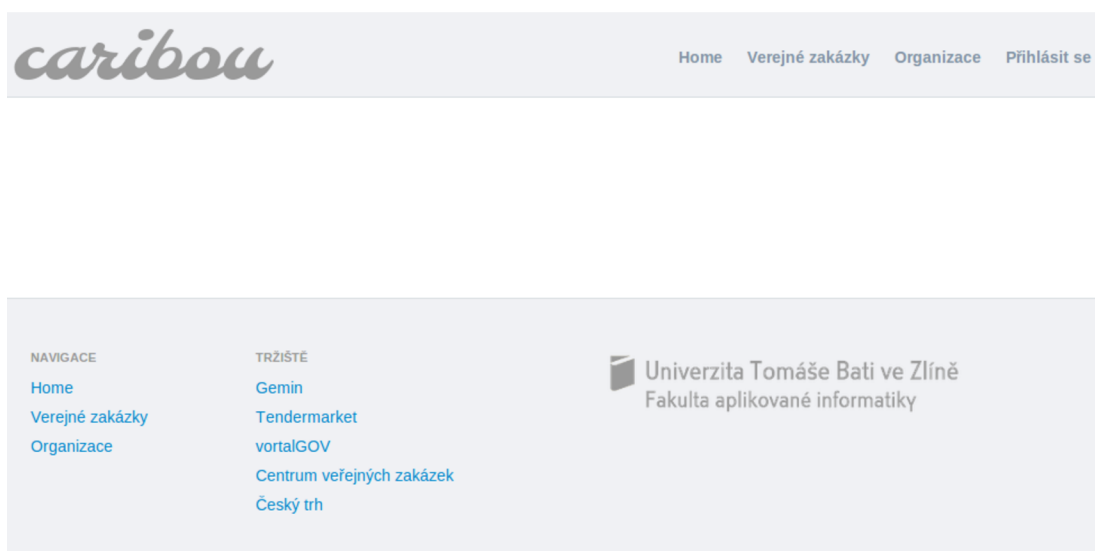
```
-- definice vlastního typu
CREATE DOMAIN password as TEXT;
alter table 'user_account' alter column 'password' type ↵
    ↵ password;
-- vytvoření funkce pro operátor
CREATE OR REPLACE FUNCTION password_eq(passwd1 password,
passwd2 password) RETURNS bool as $BODY$
DECLARE
    left_crypted bool;
    right_crypted bool;
BEGIN
    left_crypted := (substr(passwd1, 1, 3) = '$1$');
    right_crypted := (substr(passwd2, 1, 3) = '$1$');
    IF (left_crypted) AND (NOT right_crypted) THEN
        RETURN crypt(passwd2, passwd1)::TEXT=passwd1::TEXT;
    END IF;
    IF (NOT left_crypted) AND (right_crypted) THEN
        RETURN crypt(passwd1, passwd2)::TEXT=passwd2::TEXT;
    END IF;
    RETURN passwd1::TEXT = passwd2::TEXT;
END;
$BODY$ language plpgsql immutable;
CREATE OPERATOR = (
    leftarg = password,
    rightarg = password,
    negator = <>,
    procedure = password_eq
);
```

Funkce ověří, který z parametrů je zašifrován, a poté porovná originál s šifrovaným parametrem a vrátí pravdu, nebo nepravdu. Příkaz po vytvoření operátoru pro ověření uživatele se zjednoduší:

```
SELECT * FROM user_account WHERE username = 'radek' AND ↵
    ↵ 'heslo' = passwd;
```

6 POPIS WEBOVÉ APLIKACE

Šablona webové aplikace vychází z návrhů *Twitter Bootstrap*. V horní části stránky je hlavička a logem stránky a horizontální navigací. Hlavička je oddělen šedou čarou, pod kterou je umístěn obsah webové aplikace. Pod obsahem je stejně nastýlována patička obsahující navigaci stránky, seznam elektronických tržišť a logo Univerzity Tomáše Bati ve Zlíně. Šablona webové stránky má šířku a $970px$ a výška zabírá celou oblast monitoru. Patička je umístěna pod spodním okrajem stránky, obsah stránky není dostatečně dlouhý, aby zabral celou výšku monitoru.



Obrázek 17. Šablona webové stránky s prázdným obsahem

6.1 Neregistrovaní uživatelé

Po načtení webové stránky se zobrazí uvítací obrazovka s vysvětlením projektu a odkazem na registraci nového uživatele. Nepřihlášený uživatel může procházet a vyhledávat v uložených zakázkách, ale není mu umožněno hledání uložit.

Seznam zakázek je znázorněn na obrázku 18. V horní části je umístěn vyhledávací formulář, pod kterým je tabulka zakázek seřazených podle času přidání do databáze. V seznamu je zobrazen název veřejné zakázky, jako odkaz na její detail a odkaz na detail organizace.

Název veřejné zakázky Druh zakázky

Název dodavatele IČO zadavatele Cena Kč Kč

Název	Zadavatel
Propagace a prezentace ČR Správy státních hmotných rezerv	Správa státních hmotných rezerv
Hygienický materiál	NÁRODNÍ BEZPEČNOSTNÍ ÚŘAD
Spotřební materiál pro IT (pro ÚVN)	ÚSTŘEDNÍ VOJENSKÁ NEMOCNICE PRAHA
Nákup čistících a hygienických prostředků	Vězeňská služba České republiky
Dodávky kuchyňských potřeb pro závodní jídelnu	Ministerstvo zahraničních věcí
Archivní krabice a desky	Státní oblastní archiv v Plzni
OB7113-046, toner Q7553X	Ministerstvo zahraničních věcí
Dodávka tonerů pro tiskárnu HP Color LJ CP 5225	MINISTERSTVO FINANČÍ
Pytle na odpad, papírové ručníky	Vězeňská služba České republiky
Monitory 10ks	Ústav zemědělské ekonomiky a informací
Počítače, monitory a tiskárny (pro VVÚf)	ÚSTŘEDNÍ VOJENSKÁ NEMOCNICE PRAHA
Dodání písku a štěrku	Vězeňská služba České republiky
Ostrava - papír xerox formát A4	Vězeňská služba České republiky
Nákup licenční podpory 115 licencí operačních systémů RedHat Linux	MINISTERSTVO ZEMĚDĚLSTVÍ
Kancelářské a hygienické potřeby včetně rozvozu na 4 pobočky v Praze	Úřad práce České republiky
ÚP Ostrava - dodávka zárovek a zářivek pro OP Frýdek-Místek, referát Karviná	Úřad pro zastupování státu ve věcech majetkových
Kancelářské a hygienické potřeby včetně rozvozu na 4 pobočky v Praze	Úřad práce České republiky
Kancelářské potřeby - balíček	Vězeňská služba České republiky
ÚP Ostrava - Záruční a pozáruční servis a opravy služebních vozidel odloučeného pracoviště Šumperk a referát Jeseník	Úřad pro zastupování státu ve věcech majetkových
Monitory Dell 24" a 27"	NÁRODNÍ BEZPEČNOSTNÍ ÚŘAD
Individuální výuka anglického jazyka	Správa státních hmotných rezerv
Kancelářské potřeby VV Hradec Králové	Vězeňská služba České republiky
IV VS - Kurz Vyhrazená technická zařízení - plynová, zdvihací, elektrická, tlaková	Vězeňská služba České republiky
Kancelářský papír pro ÚP ČR v Jablonci nad Nisou	Úřad práce České republiky
Kancelářský papír A4 pro ÚP ČR v Liberci	Úřad práce České republiky

Obrázek 18. Seznam uložených veřejných zakázek

Po kliknutí na název zakázky se zobrazí její detail (obrázek 19). Vedle nadpisu se nachází známka s názvem tržiště odkud byla zakázka získána a odkazem na původní zakázku. Pod nadpisem lze vidět detail zadavatele veřejné zakázky a pod ním samotný obsah veřejné

zakázky, v konkrétním případě předpokládaná hodnota, druh veřejné zakázky, atd. Název zadavatele odkazuje na jeho detail (20).

Propagace a prezentace ČR Správy státních hmotných rezerv Gemin

Zadavatel

Název	Správa státních hmotných rezerv	Správa státních hmotných rezerv
IČO	48133990	Šeříková 1/616
Právní forma	Organizační složka státu	Praha, 15000

Předpokládaná hodnota	CZK585,000.00
Druh VZ	Dodávky (zboží)
Typ smlouvy	Smlouva o dílo
VZ zadávaná na základě	Ne
Výsledkem zadávacího řízení	Uzavření jednorázové smlouvy

Předmět zakázky

Stručný popis	Předmětem veřejné zakázky je vytvoření kreativního konceptu včetně harmonogramu plnění a jeho následné plnění. Cílem veřejné zakázky je propagace a prezentace ČR Správy státních hmotných rezerv (dále jen SSHR nebo zadavatel) a její přiblížení veřejnosti. Pro účely hodnocení nabídky musí být z návrhu uvedené v nabídce patrná vize a představa uchazeče o způsobu propagace a jejím dodání/plnění.
----------------------	--

Položky předmětu VZ	Název	Množství	Hodnota
	Pronázační služby	1 0	160000 0

Obrázek 19. Detail uložené veřejné zakázky

Další položka v menu zobrazuje seznam organizací, který je vytvořen podobným způsobem jako tabulka zakázek s tím, že v levém sloupci je název organizace a pravém její IČO. Název organizace slouží jako odkaz pro zobrazení jejího detailu (20).

Správa státních hmotných rezerv

Název	Správa státních hmotných rezerv	Správa státních hmotných rezerv
IČO	48133990	Šeříková 1/616
Právní forma	Organizační složka státu	Praha, 15000

Veřejné zakázky

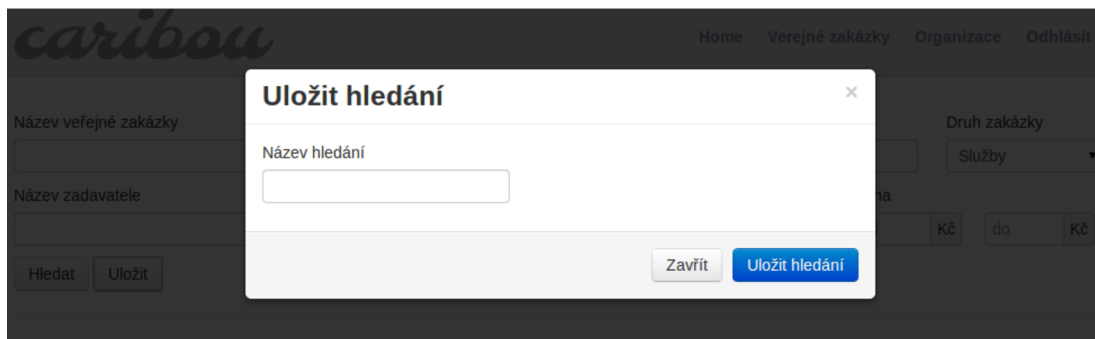
Název	Typ
Propagace a prezentace ČR Správy státních hmotných rezerv	Dodávky (zboží)
Individuální výuka anglického jazyka	Služby

Obrázek 20. Detail organizace

Detail organizace zobrazí bližší informace o sídle společnosti a seznam veřejných zakázek uložených v databázi.

6.2 Registrovaní uživatelé

Registrovaní uživatelé získají možnost své hledání uložit do databáze a dostávat automatické emailové notifikace podle zadaných kritérií při přidání nové zakázky klientem. Vedle tlačítka „Hledat“ se zobrazí tlačítko „Uložit“, které po kliknutí zobrazí modální okno pro zadání názvu uloženého hledání (obrázek 21).



Obrázek 21. Modální okno s uložením hledání

Všechna uložená hledání se poté dají zobrazit v seznamu (obrázek 22). K seznamu uložených hledání se plánuje přidat nové funkcionality jako možnosti editace uloženého hledání, čímž je myšlena změna kritérií, přejmenování a zobrazení seznamu odpovídajících veřejných zakázek.

Název hledání	Uloženo
Uloženého hledání 1	10. 5. 2013
Uloženého hledání 2	14. 5. 2013

Obrázek 22. Uložené hledání

ZÁVĚR

Cílem práce bylo vytvoření klienta stahující data z tržišť veřejných zakázek a vytvoření centrální webové aplikace pro procházení veřejných zakázek. Hlavním cílem práce je zvednout transparentnost zadaných zakázek, protože stát uzákonil vytvoření více elektronických tržišť přičemž neexistuje žádné centrální místo, kde by zakázky daly hledat. Tím se mohou některé zakázky skrýt na méně známých tržištích a díky tomu je „schovat“ pro jiné společnosti, které mohou zmeškat výběrové řízení.

V teoretické části rozebírám právo veřejných zakázek, použité Java technologie a jejich konfigurace a vývoj webové aplikace pomocí frameworku *Spring* s pohledovou vrstvou *JSF*. V druhé části je popsána uživatelská a programová dokumentace vytvořených aplikací.

Vytvořená aplikace pomocí klienta přistupuje přes *HTTP* protokol na tržiště veřejných zakázek a pomocí *Jsoup* zpracovává obsah *HTML* webové stránky a extrahuje data, např. název veřejné zakázky, zadavatel atd. Tato data jsou poté opravena a doplněna o další informace z veřejných API a uloženy do databáze PostgreSQL. Klient je napsán v jazyce Java jako konzolová aplikace, tak aby byla spouštěna pomocí cron, nebo správce úloh v operačním systému. Webová aplikace slouží pro procházení a vyhledávání uložených veřejných zakázek pomocí klienta. Přihlášení uživatelé mohou být automaticky informováni emailem o přidání nových veřejných zakázek podle jejich zadaných kritérií.

Reference

- [1] Springsource.org, 2012.
- [2] E. Burns and C. Schalk. *JavaServer Faces 2.0, The Complete Reference*. JavaServer Faces 2.0: The Complete Reference. McGraw-Hill Education, 2009.
- [3] Marten Deinum, Koen Serneels, Colin Yates, Seth Ladd, and Christophe Vanfleteren. *Pro Spring MVC*. Distributed to the book trade worldwide by Springer Science Business Media, c2012.
- [4] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, c2003.
- [5] Jonathan Hedley. Jsoup java html parser, 2013.
- [6] Radek Jurčák. *Zákon o veřejných zakázkách*. C.H. Beck, 2. vyd. edition, 2011.
- [7] Michal Krenk. *Zadávání veřejných zakázek*. Galén, 1. vyd. edition, 2005.
- [8] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.
- [9] Regina Obe and Leo Hsu. *PostGIS in action*. Pearson Education [distributor], c2011.
- [10] Regina O. Obe and Leonard Hsu. *PostgreSQL: Up and Running*. O'Reilly Media, Inc., 2012.
- [11] Paul Ramsey. Opendegeo, 2012.
- [12] Petar Tahchiev and Vincent Massol. *JUnit in action*. Manning, 2nd ed. edition, c2011.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ACID	Atomicity, Consistency, Isolation, Durability
API	Application programming interface
AoP	Aspect-oriented programming
CSS	Cascading Style Sheets
DI	Dependency injection
DOM	Document Object Model
EL	Unified Expression Language
ESD	Evropský sodní dvůr
HTML	Global Positioning System
JDBC	Java Database Connectivity
JMS	Java Message Service
JPA	Java Persistence API
JS	JavaScript
JSF	JavaServer Faces
JSP	JavaServer Pages
JTA	Java Transaction API
MVC	Model view controller
SPS	Spy Structure
SQL	Structured Query Language
SRIB	Spatial Reference System Identifier
UI	User interface
URI	Uniform resource identifier
URL	Uniform resource locator
UTM	Universal Transverse Mercator
XML	Extensible Markup Language
ZVZ	Zadavatel veřejné zakázky

Seznam obrázků

Obr. 1. Schéma aplikace	15
Obr. 2. Diagram MVC [3]	19
Obr. 3. Zpracování požadavků v Spring Web MVC [1]	19
Obr. 4. Vytvoření nového Test Case	30
Obr. 5. Nastavení nového Test Case	31
Obr. 6. Vybraní testovaných metod	32
Obr. 7. Výsledek Test Case.....	33
Obr. 8. Reprezentace bodů v PostGIS [11].....	38
Obr. 9. Reprezentace přímky v PostGIS [11].....	38
Obr. 10. Reprezentace polygону v PostGIS [11].....	39
Obr. 11. Ukázka polygonu [11].....	40
Obr. 12. Diagram struktury adres bodu	44
Obr. 13. Diagram veřejné zakázky	46
Obr. 14. Diagram tříd pro API	48
Obr. 15. Diagram tříd pro parsování zakázek	49
Obr. 16. Schéma uloženého hledání	51
Obr. 17. Šablona webové stránky s prázdným obsahem	53
Obr. 18. Seznam uložených veřejných zakázek	54
Obr. 19. Detail uložené veřejné zakázky	55
Obr. 20. Detail organizace	55
Obr. 21. Modální okno s uložením hledání	56
Obr. 22. Uložené hledání	56