VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2018

Václav Gryc

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

SROVNÁNÍ KOMUNIKAČNÍCH TECHNOLOGIÍ A VÝBĚR EFEKTIVNÍHO PROTOKOLU PRO PŘÍSTUP IOT ZAŘÍZENÍ K BACK-END SERVERU S VYUŽITÍM PLATFORMY JAVA

COMPARISON OF COMMUNICATION TECHNOLOGIES AND SELECTION OF EFFECTIVE PROTOCOL FOR ACCESS OF IOT DEVICE TO THE BACK-END SERVER USING THE JAVA PLATFORM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE Václav Gryc

AUTHOR

VEDOUCÍ PRÁCE Ing. Pavel Šeda

SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ | FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Václav Gryc — ***ID:*** 174203
Ročník: 3 — ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Srovnání komunikačních technologií a výběr efektivního protokolu pro přístup IoT zařízení k back-end serveru s využitím platformy Java

POKYNY PRO VYPRACOVÁNÍ:

Práce se bude zabývat běžně používanými komunikačními technologiemi se zaměřením na architekturu REST a protokol SOAP. Cílem práce bude implementace architektury REST a protokolu SOAP v Java EE aplikaci s využitím návrhových vzorů ze softwarového inženýrství a s ohledem na stanovená specifika. Prostřednictvím těchto dvou přístupů se budou data odesílaná na server ukládat do databáze s využitím objektově relačního mapování. Následně budou provedena měření komunikace s implementovanou serverovou stranou, z prostředí s omezenou konektivitou, například s využitím Raspberry PI 3. Student popíše výhody a nevýhody jednotlivých přístupů a zhodnotí efektivní přístup pro komunikaci z prostředí, které má omezenou konektivitu.

DOPORUČENÁ LITERATURA:

[1] RICHARDSON, Leonard a Michael AMUNDSEN. RESTful Web APIs. Beijing: O'Reilly, 2013. ISBN 978-1449358068.

[2] GUPTA, Arun. Java EE 7 essentials. Sebastopol, CA: O'Reilly & Associates, 2013. ISBN 9781449370176.

Termín zadání: 5.2.2018 — ***Termín odevzdání:*** 29.5.2018

Vedoucí práce: Ing. Pavel Šeda
Konzultant: Ing. Petr Černý, devsoft, s.r.o.

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Bakalářská práce se věnuje porovnáním dvou způsobů komunikace s webovou službou. První popsaný způsob komunikace je pomocí protokolu SOAP. Druhý způsob využívá architekturu REST, vzniká nám RESTful webová služba. V práci jsou teoreticky popsány obě možnosti. Hlavním přínosem práce je zjištění, že vhodnější pro komunikaci zařísení se sníženou konektivitou je komunikace pomocí REST API.

KLÍČOVÁ SLOVA

Java EE, Java Persistence API , Hibernate, Webové služby, SOAP, REST API

ABSTRACT

This thesis focuses on comparing two ways of communicating with a web service. The first way to communicate is using SOAP. The second way uses the REST architecture, creating a RESTful web service. Both ways are theoretically described in the thesis. The main benefit of this work for devices with reduced connectivity is the finding that communication with REST API is more appropriate.

KEYWORDS

Java EE, Java Persistence API , Hibernate, Webservices, SOAP, REST API

GRYC, Václav. *Srovnání komunikačních technologií a výběr efektivního protokolu pro přístup IoT zařízení k back-end serveru s využitím platformy Java*. Brno, Rok, 45 s. Semestrální projekt. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Pavel Šeda

Vysázeno pomocí balíčku `thesis` verze 1.6; `http://latex.feec.vutbr.cz`

PROHLÁŠENÍ

Prohlašuji, že svůj semestrální projekt na téma „Srovnání komunikačních technologií a výběr efektivního protokolu pro přístup IoT zařízení k back-end serveru s využitím platformy Java“ jsem vypracoval(a) samostatně pod vedením vedoucího semestrálního projektu a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedeného semestrálního projektu dále prohlašuji, že v souvislosti s vytvořením tohoto semestrálního projektu jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing.Pavlu Šedovi, za odborné vedení, velmi přínosné konzultace, trpělivost a podnětné návrhy k práci. Dále bych rád poděkoval svým rodičům za podporu, kterou mi po celou dobu poskytovali. Také bych chtěl poděkovat Bc.Jaroslavu Hájkovi za pomoc při měření.

Brno ....

podpis autora(-ky)

OBSAH

Úvod **10**

1 Java EE **11**

1.1 Aplikační server . . . 11

1.2 Architektura Java EE . . . 11

2 Rozdělení aplikace **14**

2.1 Databáze . . . 14

2.2 Typy databází . . . 14

2.2.1 Relační databáze . . . 14

2.3 Přístup k relačním databázím v jazyce Java . . . 15

2.4 Java Persistance API . . . 16

2.4.1 Entitní třída . . . 17

2.4.2 Relace . . . 17

2.5 Byznys vrstva . . . 18

2.5.1 Enterprise Java Bean . . . 18

2.5.2 Session Bean . . . 19

2.5.3 Message-driven Bean . . . 19

3 Webové služby **20**

3.1 Hypertext Transfer Protocol . . . 20

3.1.1 HTTP komunikace . . . 20

3.2 Web Service Description Language . . . 20

3.2.1 Struktura WSDL . . . 22

3.3 Simple Object Access Protocol . . . 23

3.3.1 Syntaxe protokolu SOAP . . . 23

3.4 Architektura REST . . . 24

3.4.1 Základní rysy Architektury REST . . . 25

3.4.2 RESTful webové služby . . . 26

3.5 Teoretické srovnání SOAP a REST . . . 27

4 Implementace webových služeb v jazyce Java **28**

4.1 JAX-WS . . . 28

4.2 JAX-RS . . . 29

4.2.1 Životní cyklus . . . 29

4.2.2 Anotace @Path . . . 29

4.2.3 Identifikátor metody požadavku a metody zdrojové třídy . . . 30

5 Webové služby v praxi **31**
5.1 Zadaní příkladu 31
5.2 Serverová aplikace 31
5.2.1 Připojení databáze k aplikaci 31
5.2.2 Návrh SOAP služeb 36
5.2.3 Návrh REST služeb 36
5.3 Klientská aplikace 37
5.3.1 Rozhraní pro komunikaci 37
5.4 Naměřené výsledky 38

6 Závěr **41**

Literatura **42**

Seznam příloh **44**

A Zdrojové kódy aplikací **45**

SEZNAM OBRÁZKŮ

1.1 J2EE Architektura 13
2.1 Vazba M:N 15
2.2 Ukázka ORM pomocí JPA 16
5.1 Vrstvy serverové aplikace 32
5.2 Návrh Databáze 35
5.3 Architektura klientské aplikace 37
5.4 Test odezvy POST metody 39
5.5 Test odezvy GET metody 39

SEZNAM TABULEK

1.1 Základní aplikační komponenty 12
2.1 Názvy anotací pro určení vztahu mezi tabulkami a příklady využití 18
3.1 Metody protokolu HTTP 21
3.2 Stavové kódy 22
3.3 Chyby definované v SOAPu 24
3.4 METODY RESTful webové služby 27
5.1 URI zdrojů aplikace pro ukládání informací uživatelů 37
5.2 Velikosti přenesených zpráv HTTP protokolem 38

ÚVOD

Vztah mezi internetem a jeho uživateli se časem rozrostl a vyvíjel. Zpočátku internet poskytoval většinou statické HTML stránky, které zobrazovali pouze strukturovaný text, obrázky a hypertextové odkazy.

Od 90. let se internet masivně rozrůstá, toto se děje díky masivnímu zvýšení počtu uživatelů. Rostoucí počet lidí také přilákal společnosti, které začaly nabízet více služeb a přesunuly své podnikání do online sféry. Z důvodu nutnosti spotřebovávání a manipulaci s informacemi některé společnosti začaly poskytovat webové služby pro poskytnutí jejích informací z jediného bodu.

Za celou dobu vzniklo více technologií pro webové služby. Weboví vývojáři mají tedy za úkol vybírat správné řešení. Pro webové služby existují dvě nejznámější možnosti a to jsou SOAP a REST, které splňují požadavky pro webovou komunikaci pomocí HTTP protokolu. Tyto dvě technologie existují už téměř 20 let, ale je stále diskutováno, která řešení je pro webové služby vhodnější. Toto lze velmi obtížně porovnávat, jelikož SOAP je protokol a REST je styl návrhu. Každé z těchto řešení má své výhody a také naopak nevýhody. Dá se říct, že každé řešení se hodí na jiný projekt.

Většina dnešních webových služeb využívá REST jako architektonický přístup, který byl původně navržený v roce 2000 Royem Fieldingem v jeho disertační práci. Na druhou stranu existuje spousta projektů, jako jsou obří bankovní systémy, které fungují s komunikačním protokolem SOAP. Tato práce se snaží porovnat a ulehčit rozhodování mezi SOAP a REST.

1 JAVA EE

Platforma Java EE (Java Enterprice Edition) vychází z programovacího jazyka Java SE (Java Standard Edition). Ve skutečnosti je to pouze soubor specifikací pro aplikační server. Cílem tohoto souboru je poskytnout vývojářům výkonnou sadu rozhraní a současně zkrátit dobu vývoje, snížit složitost aplikací a zlepšit výkon aplikací [1].

Aplikace Java EE jsou provozovány na aplikačních serverech, jako jsou IBM WebSphere, GlassFish od společnosti Oracle nebo WildFly od společnosti Red Hat [1]. Aplikační servery mohou pracovat v cloudu nebo v rámci firemního datového centra. Java EE aplikace jsou hostovány na straně serveru, jejich klientské protistrany zahrnují zařízení pro internet (IoT), smartphone, standardní webovou aplikaci, WebSocket nebo dokonce i mikrosystémy běžící v kontejneru Docker [2].

1.1 Aplikační server

Aplikační server poskytuje běhové prostředí pro komponenty, to umožňuje aplikaci soustředit se na byznys logiku a nezabývat se opakovaně stále stejnými, dávno vyřešenými, problémy.

Aplikační server poskytuje další služby jako například management a monitoring. Typicky poskytuje prostředky a rozhraní pro instalaci, spouštění a zastavování aplikací [2]. Aplikační server může spravovat uživatelská oprávnění, přičemž aplikace se o to nestará. Totéž platí o správě databázového připojení - aplikace jen používá existující datové zdroje (Data Source) a nestará se o jeho konfiguraci. Naopak konkrétní databázové dotazy, které mohou být implementovány jak přímo, tak prostřednictvím JPA (Java Persistence API), jsou záležitostí aplikace a nikoliv aplikačního serveru.

Aby aplikační server mohl být prohlášen za Java EE kompatibilní, je nutno provést certifikaci u společnosti `Oracle Corporation` [3]. Vyskytují se i necertifikované servery, které ovšem danou specifikaci splňují. Na druhou stranu všechny certifikované servery nemusí vždy zajišťovat 100% přenositelnost aplikací mezi různými aplikačními servery.

1.2 Architektura Java EE

Architektura Java EE poskytuje služby zjednodušující nejčastější výzvy, kterým vývojáři čelí při vytváření moderních aplikací, v mnoha případech prostřednictvím rozhraní, což usnadňuje používání oblíbených návrhových vzorů a osvědčených postupů přijatých v průmyslu.

Java EE je založena na vícevrstvém distribuovaném aplikačním modelu. Aplikace je rozdělena do následujících vrstev:

- Aplikační vrstva
- Webová vrstva
- Podniková vrstva
- EIS vrstva (Databáze)

Jednotlivé vrstvy se potom skládají z komponent. Komponenta je samostatná funkční softwarová jednotka, která je sestavena do aplikace Java EE s příslušnými třídami a soubory a komunikuje s dalšími komponenty [2].

- Klientské komponenty jsou spouštěny na klientském počítači
- Komponenty webové vrstvy jsou spuštěny na serveru Java EE
- Komponenty podnikové úrovně běží na serveru Java EE
- Podnikový informační systém (EIS) se spouští na serveru EIS

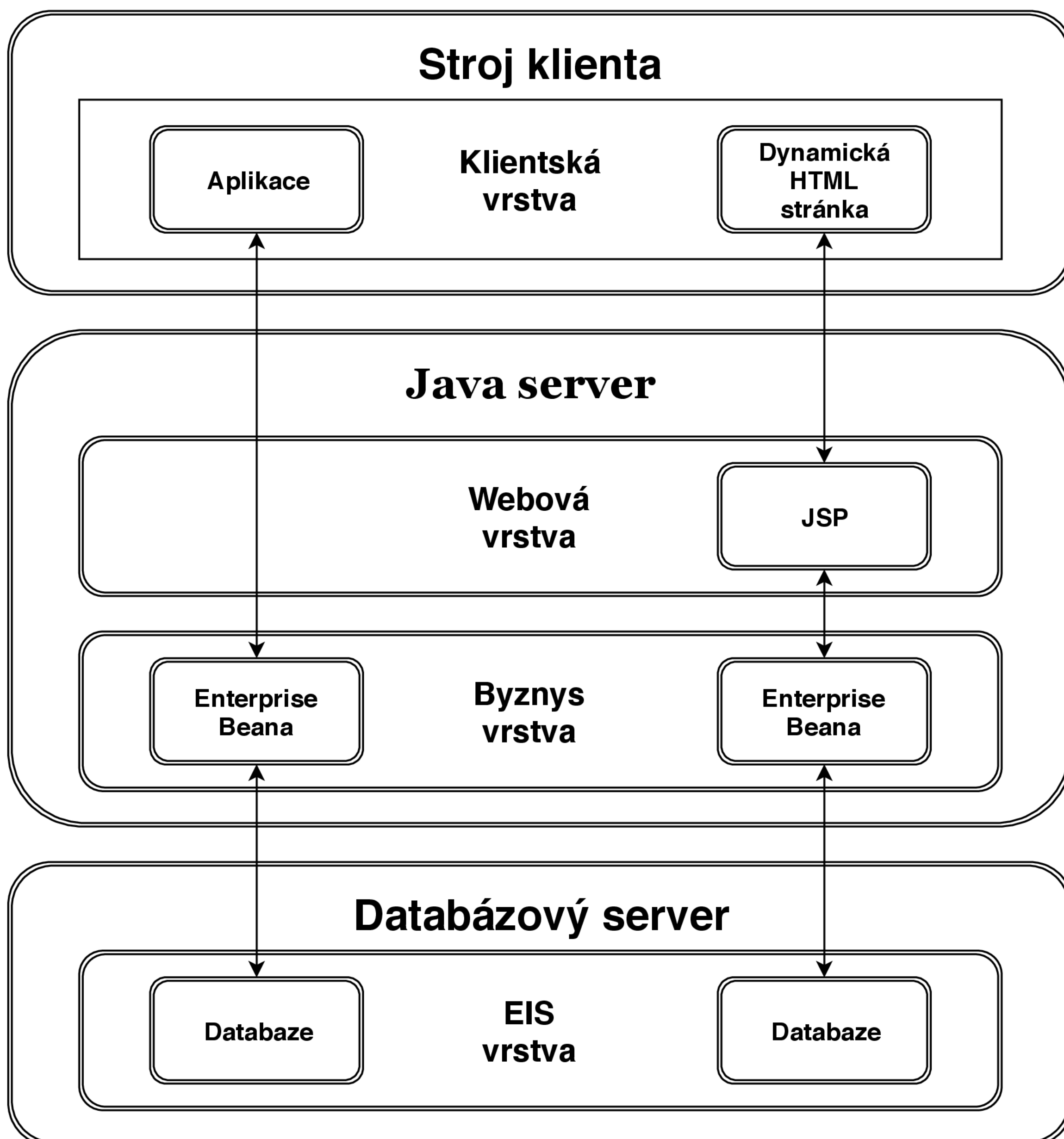
Například jedním z hlavních problémů, s nimiž se vývojáři setkávají je, jak zvládnout požadavky, které přicházejí od webových klientů. Pro zjednodušení této výzvy poskytuje Java EE rozhraní Servlet a JSP (Java Server Pages), které poskytují metody pro aktivity jako je zjištění, co uživatel napsal do textového pole v online formuláři nebo ukládal cookies do prohlížeče uživatele.

Dalším běžným úkolem je ukládání a načítání informací z databáze. Pro řešení tohoto cíle Java EE poskytuje rozhraní JPA, které usnadňuje mapování dat používaných v rámci programu na informace uložené v tabulkách a řádcích databáze. Tvorba byznys logiky aplikace je zjednodušena pomocí specifikace EJB (Enterprise Java Beans) [4].

Základní Java EE komponenty jsou uvedeny v 1.1.

Tab. 1.1: Základní aplikační komponenty

Komponenta	Popis
Servlet	Objekt generující HTTP odpovědi na základě HTTP požadavků
Java Server Pages (JSP)	Generování dynamických dokumentů pro webový prohlížeč
Java Server Faces (JSF)	Implementace uživatelského weového rozhraní
Java Persistence Api (JPA)	Provádí ORM mapování Java objektů vice zde
Enterprice Java Beans (EJB)	Implementuje Aplikační logiku vice v kapitole 3.
Další	Java Transaction Api, Java Message Service API, JavaMail API ...



Obr. 1.1: J2EE Architektura

2 ROZDĚLENÍ APLIKACE

Tato kapitola blíže popisuje jednotlivé vrstvy Java EE aplikace a práci s nimi.

2.1 Databáze

Databáze je nástroj pro trvalé ukládání a uspořádání dat [2]. Data jdou samozřejmě ukládat i do různých textových nebo binárních souborů. Problém ale nastává, když potřebujeme s daty pracovat, například vyhledávat nebo jednoduše seřadit data. Databáze nebo také databázový stroj neslouží pouze k ukládáni dat, je to velice sofistikovaný nástroj, který za nás řeší mnohé operace. Databázové systémy se dělí, podle toho jakým způsobem se v nich ukládají data. Základní typy databází jsou popsány v 2.2.

S databází komunikujeme pomocí jazyka SQL (Structured Query Language), což je velmi srozumitelný dotazovací jazyk.

2.2 Typy databází

Databáze můžeme rozdělit do následujících typů:

- hierarchická databáze,
- síťová databáze,
- relační databáze,
- objektová databáze,
- objektově relační databáze.

2.2.1 Relační databáze

Relační databáze uchovává data ve formě tabulek (relací). Řádky tabulky představují jednotlivé záznamy, sloupce pak konkrétní vlastnosti daného záznamu. Jednotlivá data jsou uložena v buňkách tabulky. Většinou je žádoucí, aby byl každý záznam jednoznačně identifikovatelný. K tomu se v databázích používá takzvaný primární klíč.

Mezi dvěma a více tabulkami může existovat logická vazba (neboli relace), kterou zajišťuje databázový systém. Dvě tabulky jsou provázány pomocí primárních klíčů. Primární klíč z druhé tabulky se nazývá cizí klíč. Bez primárního klíče nelze vytvářet relace s dalšími tabulkami. Relací existuje více druhů, tyto druhy jsou popsány v 2.2.1 [5].

K relačním databázím je přistupováno pomocí jazyka SQL. Pozitivem jazyka SQL je, že je lehce srozumitelný, někdy připomíná i klasický dotaz v anglickém jazyce.

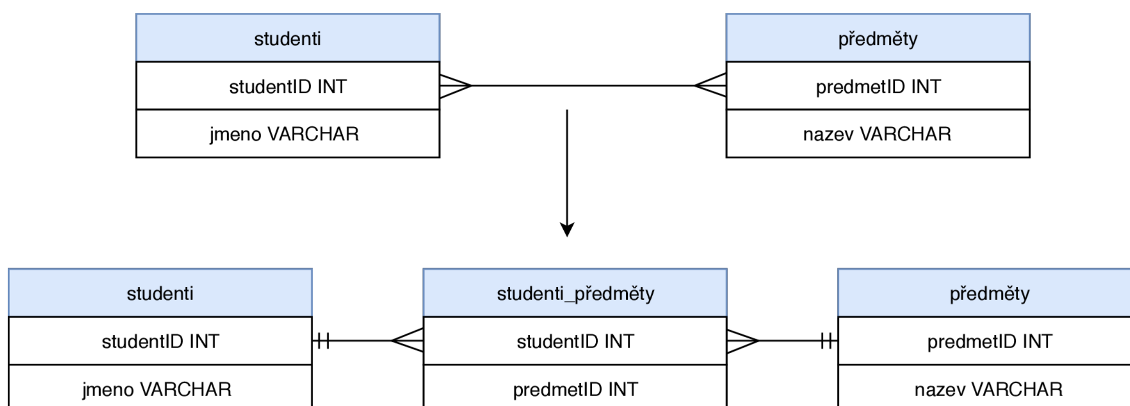
Typy relací (vazeb)

Relace tabulek jsou rozdělovány podle toho, jakým způsobem propojují tabulky.

- **Relace 1 : 1** – Relace „jedna ku jedné“ je nejjednodušší vazba, která se používá. Znamená, že jednomu záznamu v tabulce odpovídá právě jeden záznam v druhé

tabulce. Jako praktický příklad ze života je občan a jeho cestovní pas. Každý občan může mít pouze jeden cestovní pas.
Tahle relace se používá velmi málo a většinou poukazuje na špatně navrženou databázi.

- **Relace 1 : N** – Relace „jedna ku mnoha" je naopak jedna z nejpoužívanějších vazeb. V tomto typu relace pro jeden záznam v nadřazené (referencované) tabulce existuje N záznamu v tabulce podřízené (referencující). Zde je praktickým příkladem panelový dům a osoba, kde v jednom domě může mít trvalé bydliště klidně padesát osob. Naopak každá osoba má pouze jedno trvalé bydliště.
- **Relace M : N** – Relace „mnoho k mnoha" znamená, že více záznamů v první tabulce odpovídá více záznamům v druhé tabulce. Tahle vazba je nejsložitější. Pro reprezentaci M:N vazby musí být vytvořena pomocná tabulka. Tato tabulka je svázána vazbou 1:N mezi oběma tabulkami relace a pomocnou tabulkou. Jako příklad je třeba evidence knih v knihovně a osoby, které si mohou vypůjčit jakoukoli knihu. Nejlépe lze vazbu pochopit na obrázku 2.1.



Obr. 2.1: Vazba M:N

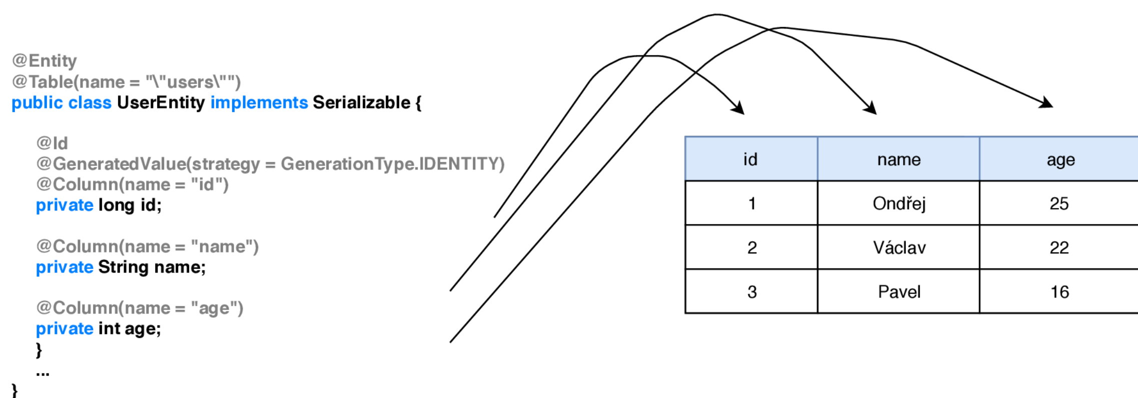
2.3 Přístup k relačním databázím v jazyce Java

Objektově orientovaný jazyk a relační databáze jsou postavené na dvou odlišných způsobech. Vzhledem k tomu, že relační databáze jsou velmi často používané a pro ukládaní dat navíc i velmi výhodné, bylo nutné najít způsoby, jak tyto dva systémy propojit.

Prvním způsobem je JDBC (Java Database Connectivity). JDBC je standardní Java API pro přístup k databázím pomocí SQL. SQL dotaz je JDBC ovladačem předán databázi a výsledky jsou zpět doručeny stejnou cestou. Od JDK 1.1 je JDBC přímo součástí programovacího jazyka Java. Tento přístup má své výhody i nevýhody. Výhodou je jednoznačně možnost využít plného potenciálu SQL databáze (včetně všech potenciálních rozšíření jazyka) a možnost psát jakkoliv složité SQL dotazy. Nevýhodou je nutnost implementace pomocné vrstvy (DAO), která se stará o převod požadavku na SQL dotaz(y) a následně musí mapovat výsledky zpět na objekty. JDBC obecně proti jiným přístupům

poskytuje jen velmi omezenou abstrakci nad databází - se všemi výhodami i nevýhodami z toho plynoucími.

Druhou možností, jak přistupovat v programovacím jazyce Java k databázi je pomocí takzvaného ORM (Object Relation Mapping), které zajišťuje, že data z databáze jsou vnímána jako objekty. Pro práci s databází nemusí být využíván jazyk SQL, tabulky se v programu chovají jako kolekce objektů, se kterými se pracuje běžnými prostředky jazyka. Vývojáři jsou odstíněni od toho, že pracují s relační databází. Standardem pro ORM v programovacím jazyce Java je JPA. Názorná ukázka principu tohoto mapování je vidět na obrázku 2.2.



Obr. 2.2: Ukázka ORM pomocí JPA

2.4 Java Persistance API

JPA (Java Persistence API) je standard popisující programátorské rozhraní (API) a chování knihoven pro objektově-relační mapování. JPA bylo vydáno v roce 2006 společností Sun Microsystems a je open source. Jelikož je pouhou specifikací (interface), k používání potřebuje implementovat hotové řešení. V této práci je využívána implementaci Hibernate, ale existují i jiné implementace například OpenJPA, Eclipse Link a mnoho dalších.[6]

JPA umožňuje vytvářet takzvané entitní třídy, které jsou pomocí anotací mapovány do tabulek databází. Jak entitní třída vypadá je popsáno v 2.4.1.

Kromě standardního JPA lze využít anotací i z proprietního API Hibernate či dalších jiných. Použití proprietních anotaci / API může být v určitých komplexních případech výhodné, nicméně za cenu nepřenositelnosti programu.

K JPA nedílně patří i dotazovací jazyk JPQL (Java Persistence Query Language). JPQL dotaz je oproti SQL vykonáván nad objekty aplikace (entitami viz. 2.4.1). Tyto objekty jsou namapovány na databázové tabulky pomocí JPA anotací. JPQL podporuje i použití parametrů v dotazu, které bychom měli vždy používat, aby nedocházelo ke vniku

chyb typu SQL injection[9]. Dalším důležitým znakem JPQL je, že odstiňuje programátora od specifik konkrétního datového úložiště. Jinými slovy, programátor se nestará o to, zda jsou v konečném důsledku dotazy převáděny nad Oracle nebo MySQL databází. O překlad JPQL dotazů do SQL dotazů se stará ORM manager s příslušnou implementaci dialektu dané databáze v ORM.

2.4.1 Entitní třída

Každá tabulka má svoji entitní třídu, která obsahuje všechny atributy dané tabulky jako proměnné. S položkami tabulek jsou svázány pomocí JPA anotací. Každá entita musí obsahovat anotaci `@Entity` a `@Id`. Všechny anotace spadající pod JPA jsou z balíčku `javax.persistence`, hibernate obsahuje i některé anotace navíc, ty jsou potom z balíčku `org.hibernate`.

Základní anotace jsou:

- **@Table** – umožňuje zadat podrobnosti o tabulce
- **@Column** – slouží k zadání podrobností o sloupci nejčastější atributy jsou name, lenght, nullable a unique
- **@GeneratedValue** – určuje, že hodnota bude automaticky generována (například ze SQL sekvence, identity u primárního klíče atd.).

Pro příklad je vytvořena entitní třída zaměstnance s atributy:

```
1  @Entity
2  @Table(name = "Employee")
3  public class Employee implements Serializable {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private long Id;
7      @Column(name = "email", nullable = false, unique=true)
8      private String email;
9      @Column(name = "age")
10     private int age;
11     //setters & getters...
12 }
```

2.4.2 Relace

Pomocí JPA anotací se mohou také tvořit asociace mezi tabulkami, kromě referencí na tabulky v databázi analogicky vznikají i reference mezi entitními třídami. Základní anotace pro svazování tabulek lze vidět v 2.1:

U každé této reference může být nastaven způsob načítaní:

- **EAGER**
  - Okamžitě načítá, krom dat dané tabulky, i data pro příslušnou asociaci
  - Automaticky se provede join při sestavení dotazu, popř. JPA implementace provede další dotaz pro načtení příslušných dat
  - Tento přístup umožňuje vyčíst všechna data zároveň, nicméně pokud je nepotřebujeme, tak nám načítání těchto dat může prodlužovat dobu provádění a přípravy daného dotazu
- **LAZY** – Data se z tabulky načítají podle potřeby.

Všechny entintní třídy jsou samozřejmě serializovatelné a používají se přímo pro předávání dat prezentační vrstvě.

Tab. 2.1: Názvy anotací pro určení vztahu mezi tabulkami a příklady využití

Anotace	Příklad
@OneToOne	Jeden občan má jenom jedno rodné číslo
@OneToMany	Zaměstnanec pracuje na víc produktech současně
@ManyToOne	Zaměstnanec zastupuje pouze 1 roli ve firmě
@ManyToMany	Firma má více klientů a klient je zákazníkem více firem

2.5 Byznys vrstva

Byznys vrstva zajišťuje veškerou aplikační logiku. Logika byznys vrstvy a rozdělení do dalších podvrstev je popsáno v 5.2.1. Tato kapitola je zaměřena na komponenty EJB (Enterprise Java Beans), ze kterých se byznys vrstva skládá.

2.5.1 Enterprise Java Bean

Enterprise Java Bean je komponenta byznys vrstvy, v anglické literatuře občas nazývána POJO (Plain Old Java Objects) [10], která běží v EJB kontejneru aplikačního serveru. EJB komponenty jsou podobné standardním Java třídám, implementují se stejně, jsou pouze doplněny o několik anotací. EJB vyžadují specifické chování a jsou v nich zakázané některé mechanismy. Například jsou v EJB zakázané takzvané restrikce (používání statických dat třídy). Podle architektury jazyka Java EE mohou EJB pracovat na různých aplikačních serverech a může být prováděna migrace klientů mezi nimi. V aplikačních serverech neexistuje žádný způsob synchronizace statických dat, z toho důvodu se nedoporučuje používat tuto techniku.
Jsou rozlišovány 2 typy EJB komponent:

- **Session Bean**

- **Message-driven Bean**

2.5.2 Session Bean

Session Bean je komponenta běžící na aplikačním serveru, která vykonává úkol pro klienta, který si jej vyžádal. Session beana zapouzdřuje byznys logiku. Instance Session Bean je možné použít z vice části aplikace současně, případně i vice různými klienty [10]. Životní cyklus každé EJB komponenty (tedy i Session Bean) je spravován aplikačním serverem. Přesné detaily jsou nicméně implementačně specifické a aplikační server může využít různé techniky pro optimalizaci správy EJB instanci (například cacheování instanci, pooling atd.) [2], [10].

Rozlišují se tři typy těchto beanů: statefull, stateless a singleton.

- **Stateless** – bean je bezstavový, to znamená, že si nemůže udržovat žádný stav mezi jednotlivými voláními. V průběhu životního cyklu aplikace udržuje AS proměnlivý počet instancí stateless session bean, které uchovává v takzvaných poolech. Když klient vyšle požadavek na tento bean, obslouží ho aktuálně volná instance. Není tedy udržován žádný vztah mezi instancí beanu a klientem. Jsou značeny anotací `@Stateless`.
- **Statefull** – bean je úzce svázán s klientskou relací, kdy v rámci této relace je potřeba udržovat stav relace tj. výsledky zpracování předchozích požadavků zaslaných konkrétním klientem v průběhu této relace Každá její instance je vytvořena a připojena k určitému klientovi a zpracovává pouze jeho požadavky. To znamená, že mezi jednotlivými requesty existuje konverzační stav. Na konci životního cyklu je zavolána metoda Remove a instance je odstraněna. Značí se anotací `@Statefull`.
- **Singleton** – je instanciován pouze jednou za životní cyklus aplikace. Tento druh beanu byl navržen pro případy, kdy je potřeba mít právě jednu EJB instanci sdílenou pro celou aplikaci. Příkladem je cache dat. Označují se anotací @Singleton.

2.5.3 Message-driven Bean

Message-driven Beana je integrovaná s technologii JMS (Java Messaging Service), která umožňuje posílat zprávy mezi jednotlivými komponentami a kanály asynchronně. Komunikační kanál musí být vytvořen v konfiguraci aplikačního serveru. Zprávy mohou být posílány i po síti [10].

3 WEBOVÉ SLUŽBY

Webová služba je softwarový systém navržený na podporu interoperability mezi dvěma stroji (klient-server) prostřednictvím sítě [11]. Jednoduše si pod touto definicí lze představit software, který je dostupný přes internet na unikátní URL (Uniform Resource Locator). Webová služba může být napsána v libovolném programovacím jazyce, protože komunikace mezi webovou službou a klientem (stroj, který webovou službu využívá) probíhá ve formátu XML (SOAP) nebo u RESTful webových služeb ve formátu JSON. V obou případech probíhá komunikace přes protokol HTTP (Hypertext Transfer Protocol).

Táto kapitola se zabývá základnímy pojmy a popisuje webové služby využívající protokolu SOAP (Simple Object Access Protocol) a RESTful webové služby využívající CRUD (Create, Read, Update, Delete) operací.

3.1 Hypertext Transfer Protocol

HTTP je internetový protokol aplikační vrstvy a byl původně navržen pro komunikaci mezi prohlížeči a webovými servery [13]. HTTP pracuje na klasickém principu žádost-odpověď (request-response) a je bezstavová, to znamená, že server neudržuje žádná data mezi jednotlivými voláními. Komunikace probíhá tak, že klient naváže spojení (pomocí protokolu TCP (Transmission Control Protocol)), pošle dotaz a čeká, dokud mu server nepošle odpověď. HTTP má specifikována pravidla, jak má vypadat formát žádosti a odpovědi [14].

3.1.1 HTTP komunikace

Každý HTTP dotaz je zahájen HTTP metodou (všechny metody jsou popsány v tabulce 3.1), následuje URL a verze HTTP protokolu. Na dalším řádku pokračují hlavičky, které jsou nepovinné, jediná povinná hlavička je `Host`, která určuje doménu, kam se má posílat dotaz. Dále pak pokračují už jen data, která jsou od hlaviček oddělena jedním volným řádkem.

Každá odpověď potom začíná `HTTP VERB + URI + verze`, dále následuje hodnota stavového kódu a příslušné stavové hlášení. Stavový kód upřesňuje, jakým způsobem byl náš požadavek zpracován. Stavové kódy jsou rozděleny podle charakteru do pěti kategorií. Kategorie můžeme vidět v tabulce 3.2. Na dalším řádku pokračují zase hlavičky a za volným řádkem server vrací samotný HTML dokument.

3.2 Web Service Description Language

WSDL (Web Service Description Language)je jazyk ve formátu XML, který popisuje rozhraní webových služeb. V praxi funguje tak, že pomocí WSDL klient zjišťuje, jaké funkce a jakým způsobem je může volat [12]. WSDL popisuje jména dostupných operací, typy jejich parametrů, návratové hodnoty a také na jakém URL je služba dostupná.

Tab. 3.1: Metody protokolu HTTP

Metoda	Popis
GET	Metoda GET znamená načíst jakýkoli dokument (ve formě entity) identifikovaný jedinečným identifikátorem (URI).
HEAD	Metoda HEAD je totožná s GET, s výjimkou, že server NEMŮŽE v odpovědi vrátit tělo zprávy. Metadata obsažená v záhlaví protokolu HTTP v reakci na žádost o HEAD MUSÍ BÝT totožná s informacemi zaslanými v reakci na požadavek GET. Tato metoda může být použita pro získání metadat o subjektu předpokládaném požadavkem bez přenesení samotného těla. Tato metoda se často používá pro testování hypertextových odkazů na platnost, přístupnost a nedávné úpravy.
POST	Metoda POST se používá k požadavku, aby server původu přijal entitu přiloženou k požadavku jako nový podřízený prostředek identifikovaný požadavkem URI v řádku požadavku.
PUT	Metoda PUT požaduje, aby byl přiložený objekt uložen pod dodaným URI. Pokud URI přiložený k požadavku už existuje, přiložená entita by se měla považovat za upravenou verzi již existujícího entity.
DELETE	Metoda DELETE požaduje, aby původní server odstranil prostředek identifikovaný požadavkem URI.
TRACE	Metoda TRACE se používá k zjištění cíle požadavku na aplikační vrstvě. Cílový příjemce požadavku může odeslat zprávu zpět klientu jako tělo entity s odpovědí 200 (OK). Cílový příjemce je původní server nebo proxy či brána, která přijme v poli hlavičky Max-Forwards nulovou (0) hodnotu. Požadavek TRACE nesmí obsahovat entitu. TRACE umožňuje klientu vidět, co je přijato na druhé straně kanálu a použít data pro testovací a diagnostické informace. Nastavením hodnoty pole hlavičky Max-Forwards umožňuje klientu omezit délku požadavkového kanálu, což je užitečné pro testování kanálu při proxy přeposílání zpráv. V případě úspěchu může odpověď obsahovat neporušený odeslaný požadavek v tělu zprávy s polem hlavičkou Content-Type typu "message/http". Odpověď na tuto metodu nesmí být ukládána do cache.
OPTIONS	Metoda OPTIONS představuje požadavek na informace o možnostech komunikace, které jsou k dispozici v řetězci identifikované žádostí URI. Tato metoda umožňuje klientovi určit možnosti nebo požadavky spojené s prostředkem, nebo schopnostmi serveru, aniž by se jednalo o akci zdroje nebo iniciování získávání prostředků.
CONNECT	Dynamicky navazuje TCP spojení skrze HTTP proxy.

Tab. 3.2: Stavové kódy

Stavový kód	Popis
1xx	Podávají informace.
2xx	Oznamují úspěšnou operaci.
3xx	Oznamují přesměrování požadavku.
4xx	Chyba ze strany klienta.
5xx	Chyba serveru.

Na internetu i ve vývojových prostředích existují nástroje, které umí WSDL dokument vygenerovat z příslušného kódu a naopak.

3.2.1 Struktura WSDL

Dokument WSDL je dokument napsaný v XML, který dodržuje schéma XML WSDL. Kostra WSDL dokumentu může vypadat následovně:

```
1  <definitions>
2     <types>
3       ...
4     </types>
5     <message>
6       ...
7     </message>
8     <portType>
9        <operation>
10          ...
11       </operation>
12    </portType>
13    <binding>
14       ...
15    </binding>
16    <service>
17       ...
18    </service>
19 </definitions>
```

Kořenový prvek všech WSDL Dokumentů je prvek zvaný `<definitions>`, který zapouzdřuje celý dokument.

- **Definice (difinitions)** – Obsahuje některé potřebné jmenné prostory a důležité prvky.
- **Typy (types)** – Deklaruje úplně všechny datové tipy, které mohou být obsaženy ve zprávě.

- **Zprávy (massages)** – Definuje všechny zprávy, které webová služba poskytuje.
- **Typ portu (portType)** – Popisuje rozhraní webové služby. můžeme si to představit, jako rozhraní v programovacím jazyce Java. WSDL dokument může mít jednu nebo více definic typu `<portType>` pro každý web.
- **Mapování (binding)** – Přiřazuje (abstraktní) typ portu a jeho operace určitému přenosovému protokol (například HTTP) a styly kódování zpráv.
- **Operace (operation)** – Popisuje provoz webové služby a používá jednu nebo více zpráv k definování jeho vstupu a výstupu a výstupní. Můžeme si to představit jako metodu v Javě.
- **Služba (service)** – Definuje konkrétní adresu, na níž je daná webová služba dostupná. Nejběžněji se zde udává URL, která slouží k vyvolání dané webové služby.

3.3 Simple Object Access Protocol

SOAP je protokol pro výměnu informací v decentralizovaném, distribuovaném prostředí [2]. Jedná se o protokol založený na XML, který zajišťuje datový přenos pro webové služby. SOAP může být potenciálně použit v kombinaci s řadou dalších protokolů aplikační vrstvy. Nejčastěji se však používá s protokolem HTTP [12].

3.3.1 Syntaxe protokolu SOAP

SOAP je dokument napsaný v XML obsahující následující prvky.

- **Envelope (obálka)** – Je kořenovým prvkem zprávy. Obálka definuje XML jako zprávu SOAP. Obálka také definuje jmenné prostory.
- **Header (hlavička)** – Je volitelný prvek a obsahuje informace o aplikaci (Ověřování, zahájení transakce atd.).
  SOAP definuje v hlavičce tři atributy a to:
  - **mustUnderstand** – Používá se k označení, zda je prvek hlavičky povinný. Pokud tedy u prvku nastavíme `mustUnderstand = "1"`, znamená to, že přijímač zpracovávající hlavičku musí rozpoznat prvek. Pokud přijímač nerozpozná prvek, zpráva bude odmítnuta.
  - **actor** – Používá se k adresování prvku záhlaví na konkrétní koncový bod
  - **encodingStyle** – Slouží k definování datových typů používaných v dokumentu. Tento atribut se může objevit na libovolném elementu a vztahuje se na obsah tohoto elementu a všechny podřízené elementy. Zprávy SOAP neobsahuje výchozí kódování.
- **Body (tělo)** – Obsahuje aktuální zprávu SOAP, na jejichž základě je provedena metoda poskytovaná serverem. SOAP zpráva definuje operace a případně její parametry, které jsou předány od klienta. V případě serverové odpovědi zpráva obsahuje výsledek přijaté operace.

Tab. 3.3: Chyby definované v SOAPu

Název	Popis
VersionMismatch	Zpracovatel nalezl neplatný obor názvů prvku `envelope`
MustUnderstand	Okamžitě podřízený prvek prvku `header`, který nebyl buď chápaný nebo nedodržen zpracovatelem, obsahoval atribut `mustUnderstand = "1"`.
Client	Zpráva byla nesprávně vytvořena nebo neobsahovaly příslušné informace, aby mohla být úspěšná. Obecně platí, že zpráva by neměla být opakována bez změny.
Server	Zpráva nemohla být zpracována z důvodů, že se nepodařilo zpracovat zprávu. Zpracování by například mohlo zahrnovat komunikaci s upstream procesorem, který nereagoval. Zpráva může být úspěšná, pokud bude odeslána později i bez změny.

- **Fault (porucha)** – Reprezentuje chybu uvnitř prvku `body`. To je nezbytné, protože bez standardní chyby by každá aplikace musel vyrobit vlastní, což by znemožnilo generické infrastruktuře rozlišovat mezi úspěchem a selháním. `Fault` může také obsahovat detailní prvek, který poskytuje podrobnosti o chybě, což může pomoci klientům diagnostikovat problém, zejména v případě chybových kódů klientů a serverů. Standardní chyby můžeme najít v tabulce 4.3.

Ukázka `SOAP` zprávy:

```
1  <SOAP-ENV:Envelope
2   xmlns:SOAP-ENV="..."
3   SOAP-ENV:encodingStyle="...">
4   <SOAP-ENV:Header>
5   <!-- Optional context information -->
6   <!-- eg. security related informations can be here -->
7   </SOAP-ENV:Header>
8   <SOAP-ENV:Body>
9   <m:GetLastTradePrice xmlns:m="some_URI">
10  <tickerSymbol>SUNW</tickerSymbol>
11  </m:GetLastTradePrice>
12 </SOAP-ENV:Body>
13 </SOAP-ENV:Envelope>
```

3.4 Architektura REST

Termín REST zavedl Roy Fielding v roce 2000 [16] v jeho dizertační práci, která mu získala doktorát ve filozofii. Hovořil v ní obecně o webové architektuře definované jako

„Representational State Transfer“ (REST). Tato architektura pojednává o tom, jak by se měla chovat dobře navržena webová aplikace. Podle Fieldinga by ve webové aplikaci měl uživatel postupovat pomocí webových odkazů (odkaz na zdroj), což by mělo za následek zobrazení další stránky (změní se stav aplikace).

REST tedy není žádný protokol, jako třeba SOAP, je to architektonický styl. Roy Fielding byl také spoluzakladatelem protokolu HTTP, proto v němu můžeme vidět rysy architektury REST. Samotná architektura REST není nijak svázaný s žádným protokolem.

3.4.1 Základní rysy Architektury REST

Aby architektura byla považována za REST, měla by dodržovat určité pravidla [2]. Ty systémy, které tyto pravidla dodržují se nazývají RESTful. Zde máme 6 pravidel, která by měla být dodržena:

- Klient-server
- Bezstavovost
- Možnosti využití vyrovnávací paměti
- Jednotné rozhraní
- Vrstvený systém
- Kód na vyžádání

Klient-Server (Client-Server)

Styl klient-server se nejčastěji vyskytuje v architektonických stylech pro síťové aplikace. Serverová část, která nabízí soubor služeb, naslouchá požadavkům na tyto služby. Klientská část, která si přeje, aby služba byla provedena, pošle na server žádost, která je serverem buď odmítnuta nebo provedena. Požadavek, nebo také informace o zamítnutí požadavku je zpět posláno klientovi.

Oddělení uživatelského rozhraní od serveru přináší přenositelnost uživatelského rozhraní mezi platformami a přispívá taky na zjednodušení serverové části. Možná je pro web nejdůležitější však to, že oddělení umožňuje komponentám samostatně se vyvíjet, čímž podporují požadavky na více organizačních domén na internetu.

Bezstavovost (stateless)

Pravidlo bezstavovosti určuje, že na serverové části není povoleno držet žádný stav (relaci). Každá žádost ze strany klienta na server musí obsahovat všechny informace potřebné pro pochopení požadavku a nemůže využívat jakéhokoli uloženého kontextu na serveru. Stav relace je uložen zcela na klientovi.

Tato omezení zlepšují vlastnosti viditelnosti, spolehlivosti a škálovatelnosti. Viditelnost je vylepšena, protože monitorovací systém nemusí přesahovat jediný vztažný bod žádosti, aby zjistil úplnou povahu žádosti. Spolehlivost je vylepšena, protože usnadňuje zotavování z částečných poruch. Škálovatelnost je vylepšena, protože není nutné ukládat stav mezi požadavky, umožňuje serveru rychle uvolnit prostředky a dále zjednodušit implementaci.

Možnost využití vyrovnávací paměti (cacheable)

Aby bylo z části eliminováno zatížení sítě, které nám zapříčiní bezstavovost, je potřeba aby systém umožňoval označování dat. Jako prostředním mezi klientem s serverem pak existuje takzvaná „cache“ ve které jsou uloženy odpovědi na předchozí požadavky, právě pokud jsou tyto data označena (v praxi jsou data označena jako `cacheable/non-cacheable`). Data uložená v cache mohou být znovu použita v reakci na pozdější požadavky, které jsou ekvivalentní a pravděpodobně budou mít stejnou odpověď.

Jednotné rozhraní (uniform interface)

Hlavním rysem, který rozlišuje architektonický styl REST od jiných stylů založených na síti, je jeho důraz na jednotné rozhraní mezi komponentami. To umožňuje komunikaci mezi klientem a serverem bez ohledu na to, na jaké jsou platformě. Jednotné rozhraní s sebou nese i jednu podstatnou nevýhodu a tou je, že odbourává efektivitu, neboť informace jsou převáděny ve standardizované podobě, nikoliv takové, která je specifická potřebám aplikace.

Vrstvený systém (layered system)

Pravidlo vrstvového systému říká, aby architektura byla složena z hierarchických vrstev. Tím omezuje chování komponent tak, že každá součást nemůže "vidět"za vrstvu, se kterou bezprostředně sousedí. Definováni tohoto pravidla značně zjednodušuje složité systémy.

Primární nevýhoda vrstvených systémů spočívá v tom, že ke zpracování dat dochází ke zvýšení režie a latence, čímž se snižuje výkon uživatele Tato nevýhoda může být z části kompenzována sdíleným ukládáním do mezipaměti u zprostředkovatelů.

Kód na vyžádání (code on demand)

Poslední přírůstek do seznamu pravidel je takzvaný kód na vyžádáni. Funkce REST umožňuje rozšíření funkce klienta tak, že stahuje a spouští kód ve formě appletů nebo skriptů. To zjednodušuje komponentu klienta snížením počtu funkcí, které je třeba předem implementovat. Povolení stahování funkcí po nasazení zvyšuje rozšíření systému. Nicméně také snižuje viditelnost a je tedy jako jediné volitelné pravidlo. Úplně předně `REST` definuje, aby tohle pravidlo bylo použitou pouze tam, kde je to výhodné nebo opravdu nutné.

Tato omezení byla volně přeložena z Fieldingove disertační práce[16].

3.4.2 RESTful webové služby

RESTful webová služba je tedy služba, která dodržuje všechna pravidla popsaná v 3.4. Tyto webové služby využívají předem definované metody. Jsou to metody právě z již zmiňovaného HTTP protokolu. Každá z těchto metod má přímo určené, jakou operaci má provést s určitým zdrojem. Zdroj si můžeme představit, jako objekt, který má svoji vlastní URL. Všechny metody a jejich funkci můžeme vidět v tabulce 3.4.

Tab. 3.4: METODY RESTful webové služby

Metoda	Popis	Idenpotentnost
GET	Pouze pro získání reprezentace zdroje (informace).	ANO
POST	Slouží k vytvoření nových podřízených zdrojů.	NE
PUT	Slouží pro aktualizaci existujícího zdroje (zdroj by měl být smazán a znovu nahrán). V případě neexistujícího zdroje muže server pod metodou PUT vytvořit i nový zdroj	ANO
PATCH	Slouží pro aktualizaci zdroje, ale entitu pouze upravuje, nemaže a nenahrává novou jak je to v případě `PUT`.	NE
DELETE	Používá se k odstranění zdroje.	ANO

3.5 Teoretické srovnání SOAP a REST

Nyní už jsou známy dostatečné informace o SOAPu a RESTu na to, aby mohli být teoreticky porovnány.

Hlavním a zcela jasným rozdílem obou přístupů je flexibilita. RESTful webové služby jsou omezeny HTTP metodami, zatímco u SOAP komunikace si můžeme definovat metody vlastní. Na druhou stranu REST využívá celé spektrum HTTP protokolu, zatímco komunikace SOAP je zúžená pouze na využití POST metody protokolu HTTP. Stejně tak využití proxy je komplikovanější, protože jednotlivé proxy členy musí rozumět formátu SOAP zprávy. V případě REST služeb jsou proxy výrazně jednodušší, protože používají čistě HTTP protokol.

Další velkou nevýhodou protokolu SOAP je, že zabaluje svoji zprávu do obálky, tím značně narůstá velikost zpráv a tím pádem zatížení celé sítě. Další nevýhodou obálky je, že omezuje protokol SOAP pouze na formát XML, zatím co u RESTu je zcela jedno, v jakém formátu zpráva bude, nejčastěji se dnes používá formát JSON a to díky rozsáhlým JavaScriptovým klientům, kteří dokáží s tímto formátem velmi efektivně pracovat.

Aby jsme protokolu SOAP pouze nekřivdily tak můžeme říct, že služby využívající SOAP jsou velmi dobře zaběhnuté a používané v řadě aplikací. Podpora SOAP v jednotlivých programovacích jazycích a vývojových prostředích je velmi propracovaná. Vytvoření takové služby může uživatel vykonat pomocí průvodců a většina kódu jako WSDL je vygenerována automaticky. Podobné je to i u klienta webové služby, který jde vygenerovat pomocí pokročilých nástrojů přímo z WSDL. Tudíž programátor pouze napíše službu dle zvyklostí konkrétní implementace a obrovské množství složitého kódu je vytvořeno automaticky.

4 IMPLEMENTACE WEBOVÝCH SLUŽEB V JAZYCE JAVA

Tato kapitola se zabývá vývojem webových aplikací na platformě Java EE. Ukazuje jak používat Java API pro webové služby založené na protokolu SOAP (JAX-WS) a API pro RESTful webové služby (JAX-RS).

4.1 JAX-WS

Pro vytvoření jednoduché webové služby komunikující pomocí protokolu SOAP se v jazyce Java využívalo JAX-RPC API. Od verze jazyka Java 5 bylo aktualizováno a přejmenováno na JAX-WS. JAX-WS přineslo spoustu změn, nejpodstatnější změnou bylo, že API začalo využívat anotace zavedené v jazyce Java 5. Tato podstatná změna výrazně zjednodušila webové služby v jazyce Java.

Rozhraní JAX-WS přineslo nové nástroje: `wsgen` a `wsimport`.

wsgen přečte třídu implementace webových služeb a podle meta dat webových služeb vygeneruje WSDL potřebné pro nasazení webové služby.

wsimport naopak přečte WSDL soubor a vygeneruje třídy, které lze využít pro volání operací webové služby.

Implementace rozhraní JAX-WS

Díky anotacím a nástroji `wsgen` není potřebné konfigurovat žádné XML jak tomu bylo dříve. Příklad kódu jednoduché webové služby, která vrácí text `Hallo world!!`.

```
1  @WebService
2  public class MyService {
3      @WebMethod(operationName = "sayHallo")
4      public String sayHallo(){
5          return "Hallo World!!";
6      }
7  }
```

Nyní tato třída může být spuštěna na libovolném serveru vyhovujícímu standardu Java EE 5. **@WebService** označuje třídu jako provádějící webovou službu. Třídu může být označena také anotací `@WebServiceProvider`. Nikdy ale nemůže používat obě anotace zároveň.

@WebMethod anotace označuje metodu, která je operaci webové služby.

4.2 JAX-RS

Hlavním smyslem JAX-RS je poskytnout programového rozhraní, které ulehčí vývoj RESTful webových služeb. Existují dvě hlavní implementace JAX-RS API.

- Jersey
- RESTEasy

JAX-RS využívá anotací ze standardu Java 5, které umožňují definovat jednotlivé zdroje a funkce. Anotace jsou prováděny přímo na POJO a tím se z tohoto objektu stane služba.

Službu si tedy můžeme představit jako třídu v jazyce Java, která má alespoň jednu metodu označenou anotací `@Path` nebo takzvaným „indikátorem metody požadavku" (Request method designator), který je podrobněji popsán v 4.2.3. Taková třída se nazývá třídou zdrojovou.

4.2.1 Životní cyklus

Instance zdrojové třídy je vytvářena pro každý požadavek. Zdrojová třída prochází při obsloužení požadavku následujícími kroky:

- konstrukce,
- vložení závislostí,
- zavolání odpovídající metody a obsloužení požadavku,
- objekt je dostupný pro garbage collection.

Po vytvoření instance zdrojové třídy jsou nastaveny její vlastnosti opět pomocí anotací:

- @MatrixParam,
- @QueryParam,
- @PathParam,
- @CookieParam,
- @HeaderParam,
- @Context.

4.2.2 Anotace @Path

Anotace `@Path` definuje relativní URI (Uniform Resource Identifier), na kterém bude dostupná služba definované touto anotací. Anotace `@Path` může definovat i vlastní proměnné, které jsou za běhu aplikace nahrazeny aktuálními hodnotami. Tyto proměné jsou v URI uzavřené ve složených závorkách. Na následujícím příkladu můžeme vidět URI šablonu například pro vybrání ID uživatele:

```
1 @Path("/users/{userid}")
```

Při zavolání služby `users/1` dojde k přiřazení hodnoty 1 do proměnné `userid`. Pro načtení této proměnné například jako parametru metody použijeme anotaci `@PathParam`. Tyto hodnoty proměnné je možné omezit pomocí regulárních výrazu. V našem případě s uživateli by to vypadalo následovně:

```
1  @Path("/users/{userid:[0-9]}")
```

Zavoláním takovéto služby s atributem nepatřícím mezi hodnoty 0-9 včetně, by byla oznámena chyba 404 (Not found).

4.2.3 Identifikátor metody požadavku a metody zdrojové třídy

Identifikátor metody požadavku je běhová anotace, která svazuje metody zdrojové třídy s jednotlivými metodami `HTTP` protokolu. Tyto anotace vycházejí přímo z názvů metod: **@GET**, **@POST**, **@HEAD**, **@PUT** a **@DELETE**. V praxi tedy při zavolání `HTTP` metody `GET`, na adrese odpovídající šabloně obsahu anotace `@path`, bude zavolána metoda identifikována anotací `@GET`.

5 WEBOVÉ SLUŽBY V PRAXI

V této kapitole je popsán vývoj serverové a klientské aplikace. Tyto dvě aplikace implementují JAX-RS API i JAX-WS API. V závěru této kapitoly je provedeno měření komunikace mezi serverovou a klientskou aplikací.

5.1 Zadaní příkladu

Pro demonstraci byl zvolen systém pro ukládání informací o zaměstnancích na pracovišti. Systém umožňuje registrovat zaměstnance a zaznamenávat atributy o něm. Tyto atributy mohou být modifikovány nebo mazány. Je také možné vyhledávat pomocí atributů, nebo mazat uživatele.

Po registraci uživatele dojde k uložení uživatele do MySQL databáze. Po registraci je možné k tomuhle uživateli přidávat nebo mazat data o jeho životních funkcích jako jsou stres, srdeční tep, váha a procento tuku.

Serverová část se stará o práci s databází, implementuje jak rozhraní JAX-RS, tak JAX-WS. Aplikace je schopna komunikovat prostřednictvím sítě internet na koncových bodech popsaných v 5.2.2 a 5.2.3.

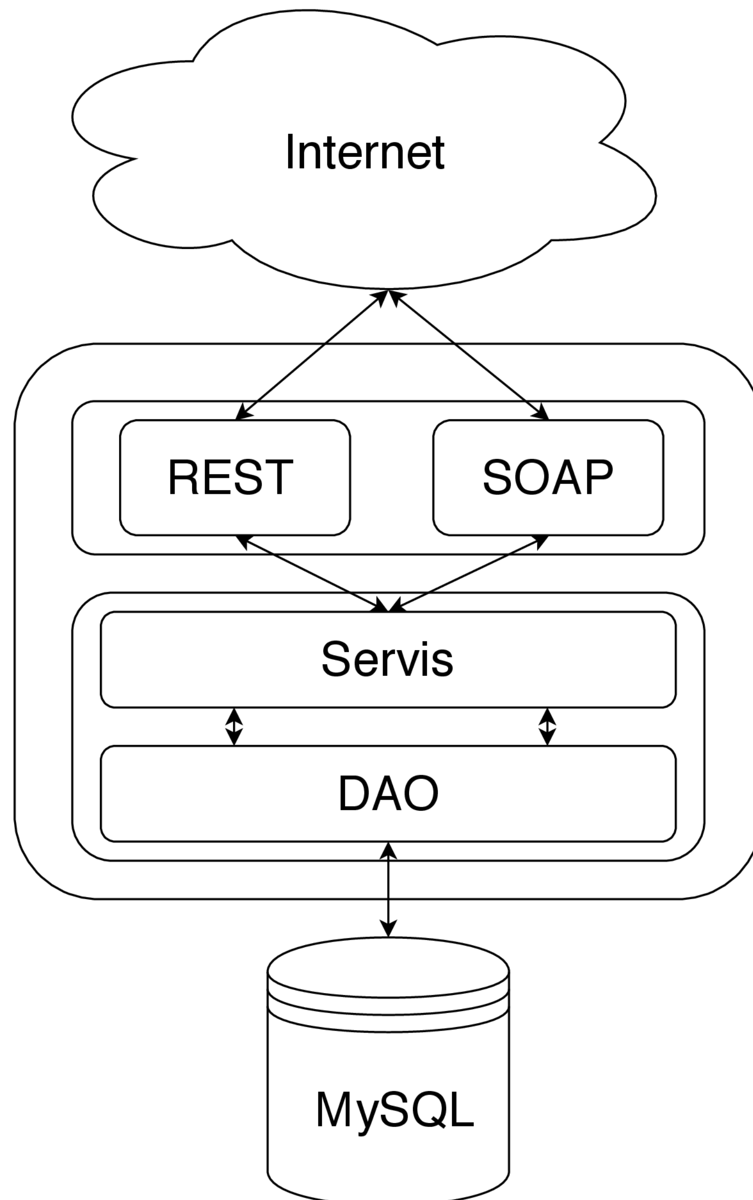
Klientské aplikace jede na Raspberry Pi a je připojená k síti internet pomocí technologie EDGE. Klientská aplikace provádí měření délky komunikace a vykresluje graf pro porovnání přístupu přes obě rozhraní.

5.2 Serverová aplikace

V každém větším projektu je vhodné analyzovat problém a rozdělit ho na menší podproblémy. K tomu slouží rozdělení programu do vrstev. Oddělíme tím logiku aplikace od práce s databází a od komunikačních API. Rozdělení demonstrační aplikace lze vidět na 5.1.

5.2.1 Připojení databáze k aplikaci

Před samotným použitím JPA, popsaným v 2.4, se musí vykonat konfigurační nastavení. Jako první je potřeba aplikaci propojit s databází. To lze provést přidáním modulu do aplikačního serveru, nainstalováním potřebného ovladače pro JDBC a nastavením data source [7].



Obr. 5.1: Vrstvy serverové aplikace

Modul se přidá tím, že ve složce `/modules/system/layers/base` se vytvoří adresáře `com/mysql/driver/main`, sem se poté vloží stažený JDBC ovladač (například Mysql Connector 5.1.33) a vytvoří se soubor `module.xml`.

Obsah souboru `module.xml` vypadá následovně:

```
<module xmlns="urn:jboss:module:1.3" name="com.mysql.driver">
    <resources>
        <resource-root path="mysql-connector-java-5.1.33.jar" />
    </resources>
    <dependencies>
        <module name="javax.api"/>
        <module name="javax.transaction.api"/>
    </dependencies>
</module>
```

Do konfiguračního souboru aplikačního serveru (ve WildFly `home/bin/standalone.sh`) je potřeba přidat následující kód k přidání ovladače a data source databáze:

```
<datasources>
    <datasource jndi-name="java:jboss/datasources/server"
    pool-name="serverDS" enabled="true" use-java-context="true">
        <connection-url>
            jdbc:mysql://localhost:3306/database_bp
        </connection-url>
        <driver>mysql</driver>
        <security>
            <user-name>root</user-name>
            <password>1234</password>
        </security>
    </datasource>
</datasources>
<drivers>
    <driver name="mysql" module="com.mysql.driver">
    <driver-class>com.mysql.jdbc.Driver</driver-class/drive>
</drivers>
```

Poté už jen stačí vytvořit persistence unit (persistetní jednotka). Ta se nastaví vytvořením souboru `META-INF/persistence.xml` v adresáři zdrojových souborů (typicky adresář `src`, takže na vrcholu hierarchie balíčků), čím je zabezpečeno, že ho IDE zkopíruje do kořenového adresáře v `CLASSPATH` [7].

Nastavení persistence.xml:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
       http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
3    <persistence-unit name="com.mycompany_ServerBp_war_1.0-SNAPSHOTPU"
         transaction-type="JTA">
4      <jta-data-source>java:jboss/datasources/server</jta-data-source>
5      <exclude-unlisted-classes>false</exclude-unlisted-classes>
6      <properties>
7        <property name="javax.persistence.schema-generation.database.action"
             value="drop-and-create"/>
8        <property name="hibernate.show_sql" value="true"/>
9      </properties>
10   </persistence-unit>
11 </persistence>
```

Takovýchto jednotek může může být v systému více. V demonstračním příkladu stačí pouze jedna perzistentní jednotka [7].

Implementace JPA

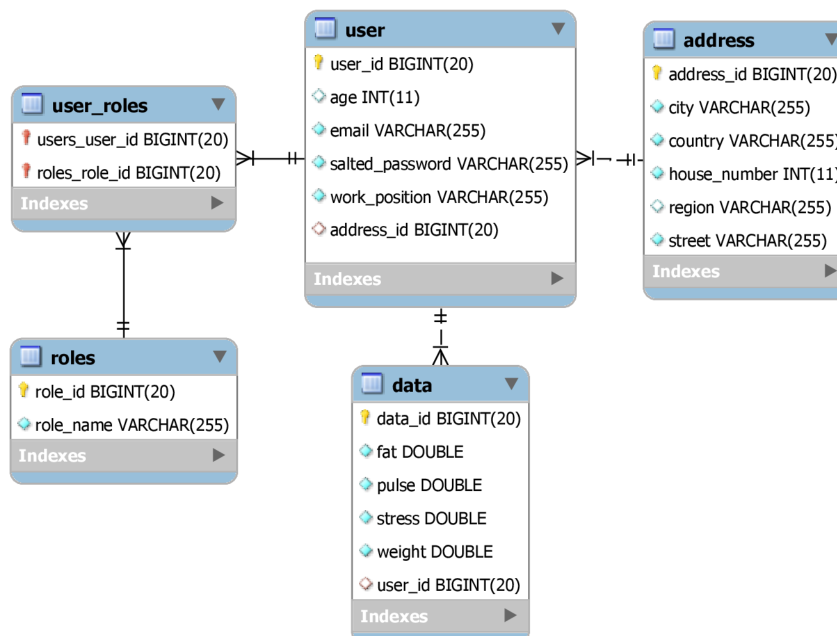
Propojení databáze s aplikací je realizováno pomocí technologie JPA. Všechny entitní třídy jsou vytvořeny pomocí anotací JPA, stejně jako všechny dotazy jsou psané v jazyce JPQL za použití objektů poskytovaných JPA. Databázový model si lze prohlédnout na obrázku 5.2.

Pro každou tabulku je zvlášť vytvořena entitní třída, která obsahuje všechny položky dané tabulky v databázi jako proměnné dané třídy. Pomocí JPA anotací jsou svázány z položkami tabulek. Zároveň zde jsou vytvořeny asociace mezi tabulkami, tím nám kromě referencí na tabulky v databází vznikají zároveň analogické reference mezi jednotlivými třídami v JPA.

Byznys vrstva

Byznys vrstva slouží k provádění operací a požadavků klientů. Téměř veškerá datová logika aplikace je prováděna právě v byznys vrstvě. Ve složitějších aplikacích se byznys vrstva rozděluje ještě do dalších podvrstev.

Klíčovou odpovědností DAO vrstvy je práce s databází - vyšší vrstvy na databázi nemají žádné napojení, veškerá interakce se děje právě pomocí DAO vrstvy. Používají se zde entitní třídy, jednak pro ukládání a načítání dat jednotlivých tabulek z/do databáze, ale zároveň i pro předávání dat vyšší servisní vrstvě. Veškeré operace se provádí pro perzistentní jednotku. Perzistentní jednotku můžeme referencovat pomocí resource injection



Obr. 5.2: Návrh Databáze

a anotací `@PersistenceUnit` a `@PersistenceContext`. Resource injection spočívá v připojení anotace k proměnné dané třídy určitého pevně daného typu. Aplikační server poté provede tzv. „injection“ – neboli nastavení této proměnné na instanci implementující třídy. Tyto resource injection je možné samozřejmě provádět pouze v určitém kontextu (v našem případě v EJB).

Umožňují nám snadný přístup k prostředkům aplikačního serveru bez nutnosti složitého JNDI vyhledávání (které jsme museli používat v Java EE před verzí 5). `@PersistenceUnit` reprezentuje referenci na objekt typu EntityManagerFactory, který slouží jako factory třída pro vytváření objektů typu EntityManager. To je ovšem ve většině případů nepraktické nebo nežádoucí, častěji necháváme vytvoření entitního manažeru na aplikačním serveru. K tomu nám slouží anotace `@PersistenceContext`, která se postará o resource injection objektu typu EntityManager, který můžeme rovnou začít používat pro operace nad JPA [9].

Příklad vyhledání uživatele v databázi podle primárního klíče:

```
1  @PersistenceContext
2  EntityManager em;
3  public UserEntity getUserByFind(long id) throws ResourceExceptions {
4      UserEntity user = em.find(UserEntity.class, id);
5      if (user == null) {
6          throw new ResourceExceptions.ResourceNotFoundException();
7      }
8          return user;
9  }
```

```
10  public UserEntity getUserByJpql(long id) throws ResourceExceptions{
11      try {
12          UserEntity newUser = em.createQuery("SELECT user FROM UserEntity user
                WHERE (user.id = :id)", UserEntity.class)
13              .setParameter("id", id).getSingleResult();
14          } catch (NoResultException e) {
15              throw new ResourceExceptions.ResourceNotFoundException();
16          }
17          return false;
18      }
```

Další částí byznys vrstvy je servisní vrstva, ta zpracovává veškerou logiku pro obsloužení klienta. Pro připojení DAO vrstev se používá Dependency Injection. Ty nám umožňují nevytvářet globální proměnné nebo statické třídy DAO vrstvy, pomocí anotace `@Injection` z knihovny `javax.inject.Inject` vytvoří závislost na určité beaně z DAO vrstvy a už nepotřebuji vytvářet instanci. Byznys vrstva je vytvořena pomocí EJB komponent. EJB komponenty jsou popsány v kapitole 2.5.1.

5.2.2 Návrh SOAP služeb

Z kapitoly 3.3 víme, že u soapu pracujeme s metodami objektu. V praxi nám tedy stačí obalit POJO anotací `@websrvices` a nad každou operaci (metodu tohoto objektu) vložit anotaci `@WebMethod`. U této anotace můžeme použít atribut `operationName` pro nastavení jména koncového bodu. Vše ostatní nutné pro komunikaci s takovouto službou (převod do SOAP zpráv, vygenerování WSDL atd.) zajistšťuje JAX-WS.

5.2.3 Návrh REST služeb

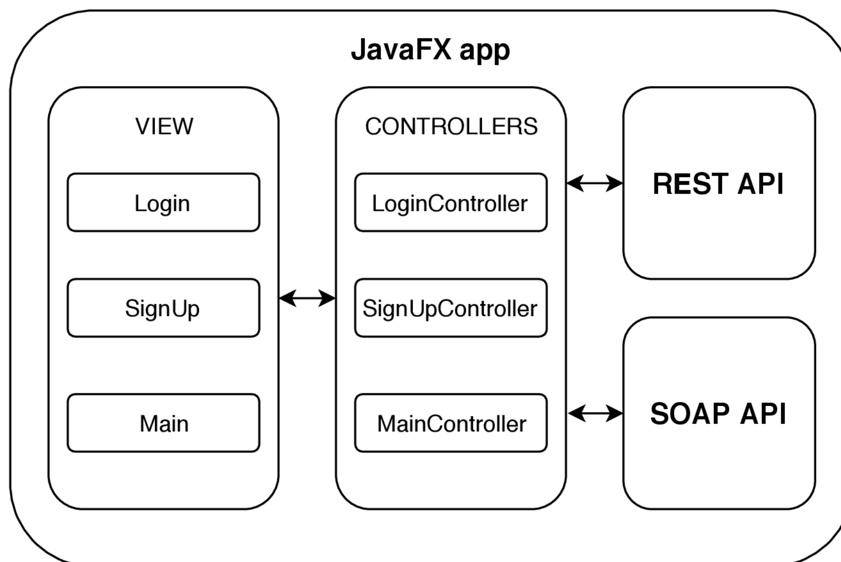
U RESTful webových služeb je situace o něco složitější, protože se pracuje se zdroji a operacemi nad nimi a ne přímo s metodami objektu. Je tedy důležité dobře navrhnout adresy zdrojů podle smyslu metod a vyvarovat se špatného použití filozofie REST. V 5.1 je popsán návrh zdrojů pro demonstrační úkol.

Tab. 5.1: URI zdrojů aplikace pro ukládání informací uživatelů

URI	HTTP metoda	Význam
/users	GET POST	Vrátí pole všech uživatelů Registrace nového uživatele
/users/{id}	GET PUT DELETE	Vrátí detail uživatele Upgrade existujícího uživatele Vymaže uživatele
/users/{id}/address	GET	Vrátí adresu uživatele
/users/{id}/data	GET	Vrati data uživatele
/users/{id}/data/{dataId}	GET	Vrátí detail data uživatele
/users/search	GET	Vyhledávání uživatele, v URI bude následovat ? a výhledávaný výraz. příklad: /users/search?age = 25

5.3 Klientská aplikace

Klientská aplikace je naprogramovaná ve frameworku JaveFX. JavaFX nebyla tématem téhle bakalářské práce, proto se práce bude dále zabývat pouze komunikací mezi klientem a serverem. Návrh klientské aplikace lze vidět na 5.3.



Obr. 5.3: Architektura klientské aplikace

5.3.1 Rozhraní pro komunikaci

Komunikace pomocí SOAP protokolu v programovacím jazyce Java je velice jednoduchá, díky JAX-WS a nástroji `wsimport` který přímo generuje třídy z přiloženého WSDL. Díky

tomuto procesu absolutně odstíněna od definování adres služby a návratového formátu. Komunikace pro vývojáře tedy připomíná volání metod z nějaké knihovny.

V případě REST komunikace se musí pracovat s adresou zdroje, specifikovat jaká HTTP metodu je potřeba využít a jaká data očekávat. I když RESTEasy nabízí vcelku jednoduché API, práce se SOAP službami je přece jen mnohem jednodušší a intuitivnější.

Toto se jedná ale hlavně programovacího jazyka Java, například v Javascriptu je situace úplně jiná. V Javascriptu se musí složitě vytvářet SOAP zprávu v XML, což není nijak příjemné. V případě REST si Javascript vystačí s jednoduchou knihovnou, která poskytuje základní podporu asynchronního volání HTTP metod. Navíc je možno pomocí hlavičky `Accept` požádat o JSON, který je pro práci s daty v Javascriptu mnohem efektivnější.

Další výhodou REST klienta je, že negeneruje žádné třídy, používá pouze objekty z RESTEasy a tím pádem zabírá mnohem méně místa.

5.4 Naměřené výsledky

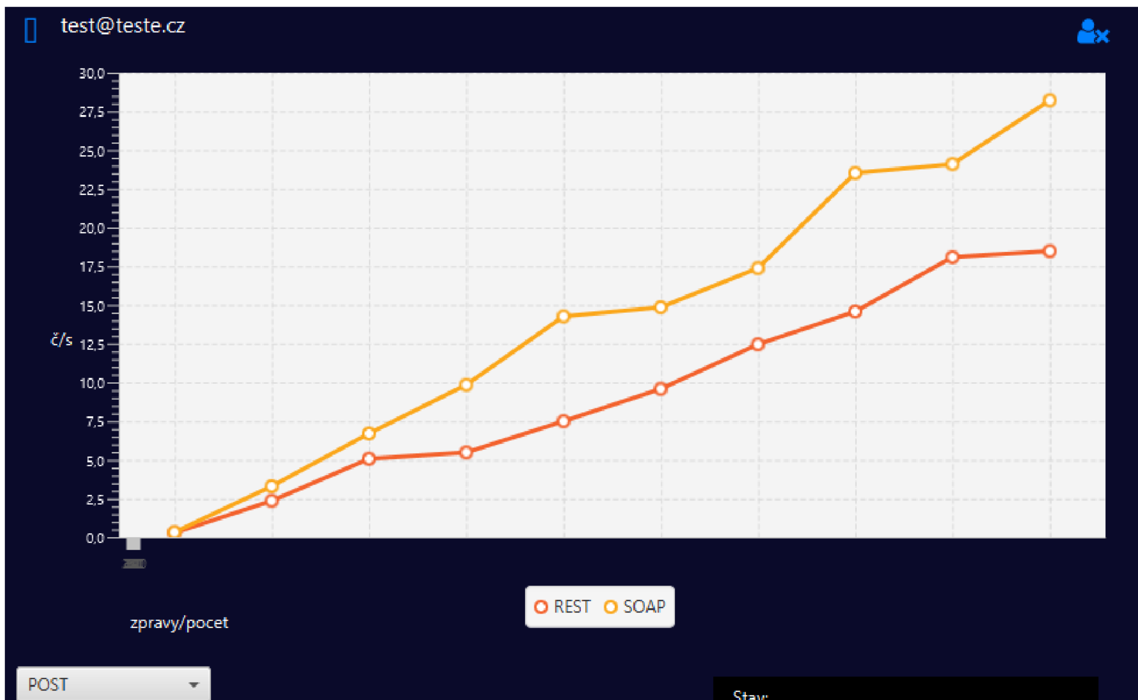
Měření odezvy bylo provedeno na IoT zařízení Raspberry Pi2, připojené k síti internet prostřednictvím technologie EDGE. Z RPi byla odesílána data v intervalech od 1 zprávy po 100 zpráv. Klientská aplikace měřila odezvu komunikace a nakonec byly hodnoty odezvy vykresleny do grafu, který lze vidět v 5.4 a 5.5. Na stolním PC, který sloužil jako server, byl spuštěn program wireshark, pomoci kterého byly měřeny velikosti zpráv.

Podle teoretické části bylo očekáváno, že REST bude efektivnější. Jelikož testování probíhalo na zařízení s omezenou konektivitou, měl by být vidět i rozdíl ve velikosti odezvy. V tabulce 5.2 jsou zobrazeny velikosti odeslaných zpráv.

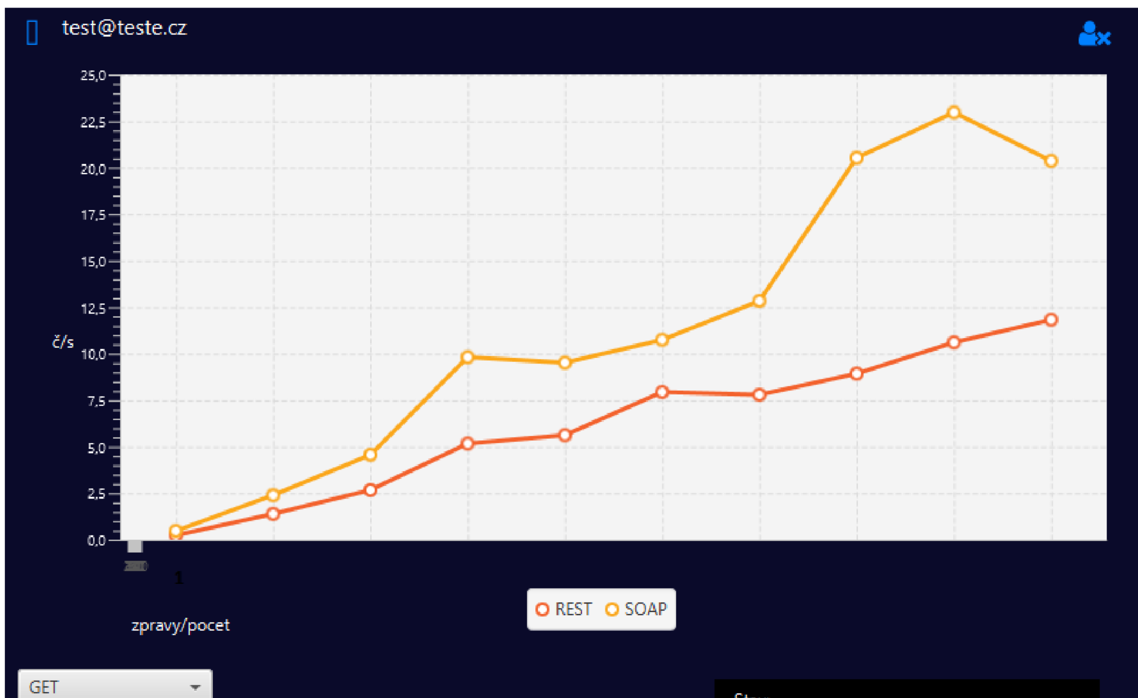
Tab. 5.2: Velikosti přenesených zpráv HTTP protokolem

Technologie	Metoda	Request/Resonse	Velikost (bytes)
SOAP	GET	Request	201
SOAP	GET	Response	321
REST	GET	Request	0
REST	GET	Response	106
SOAP	POST	Request	318
SOAP	POST	Response	321
REST	POST	Request	96
REST	POST	Response	106

Test ukazuje, že i když bylo posláno malé množství dat, SOAP protokol je obalil obálkou, která zabírala několikanásobek místa. Z tabulky 5.2 jde tedy vidět, že výměna jedné informace, která v případě komunikace pomocí RESTfull webových služeb zabírá 106 bytů, u protokolu SOAP potřebuje přenést přes internet 552 bytů. To je skoro pětinásobná hodnota. Připojení EDGE má maximální datový přenos 6,5 kB/s [17], takže teoreticky odeslání 100 zpráv by mělo mít odezvu 8,5 sekund. Stejné množství zpráv by mělo systému



Obr. 5.4: Test odezvy POST metody



Obr. 5.5: Test odezvy GET metody

využívající REST zabrat pouze 1,63 sekund. Rozdílný čas je tedy 6,87 sekund. V grafu 5.5 je rozdílný čas cca 10 sekund. Je jasné, že přenos nikdy nebude mít vždy stejnou maximální rychlost, ta je ovlivněna zatížením sítě. Výsledek ale potvrzuje, že RESTfull webové služby jsou rychlejší a méně náročné na zatížení sítě.

6 ZÁVĚR

Tato práce se zabývala srovnáním komunikačních technologií pro webové služby. Cílem bylo naprogramovat server poskytující webové služby pomocí REST API a protokolu SOAP, vyzkoušet si obě technologie v praxi, porovnat je a vyhodnotit, který přístup je vhodnější pro komunikaci mezi IoT zařízením a serverem. V první části práce byl popsán programovací jazyk Java EE, následně bylo vysvětleno, co jsou to webové služby a byly popsány přístupy k nim, jak pomocí protokolu SOAP, tak REST API.

Další část práce se zabývala implementací webových služeb v programovacím jazyce Java EE. Pro webové služby bylo využito rozhraní JAX-WS a JAX-RS. Tyto rozhraní byly podrobněji popsané. Byla naprogramována serverová aplikace, která zpracovává udaje o zaměstnanci a umí tyto údaje poskytovat klientu. Dále byla naprogramována klientská aplikace, která dovoluje registraci nového uživatele, přihlášení do systému a testování doby trvání komunikace pomocí RESTfull webových služeb a protokolu SOAP pro HTTP metody GET a POST. Aplikace vykresluje názorný graf, kde lze oba přístupy jednoduše porovnat. Bylo potvrzeno, že REST využívá celé spektrum HTTP protokolu a je tedy mnohem flexibilnější a ohebnější oproti protokolu SOAP.

V bakalářské práci bylo prováděné měření velikosti těla HTTP metody pomocí programu Wireshark a bylo zjištěno, že v případě SOAPu je přenášeno pětkrát více dat, než v případě RESTu. U IoT zařízení s omezeným připojením k internetu se tato skutečnost výrazně projeví i na rychlosti komunikace.

I když implementace protokolu SOAP v programovacím jazyce Java se zdá mnohem jednodušší a intuitivnější, pro komunikaci IoT zařízení s omezeným připojením k internetu se jeví jako vhodnější přístup pomocí RESTfull webových služeb hlavně díky menšímu zatížení sítě.

LITERATURA

[1] ASHMORE, Derek Clark. The Java EE architect's handbook: how to be a successful application architect for Java EE applications. Second edition. Bolingbrook, IL: DVT Press, 2013. ISBN 978-097-2954-884.

[2] GONCALVES, Antonio. Beginning Java EE 7. Berkeley, CA: Apress, 2013. Expert's voice in Java. ISBN 143024626x.

[3] JENDROCK, Eric., Ricardo CERVERA-NAVARRO, Ian EVANS, Kim HAASE a William MARKITO. The Java EE 7 tutorial. Fifth edition. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN 978-032-1980-083.

[4] Introduction to the Java EE Architecture. Introduction to the Java EE Architecture [online]. 1994-2005 [cit. 2017-11-04]. Dostupné z: http://pawlan.com/monica/articles/j2eearch/

[5] HARRINGTON, Jan L. Relational database design and implementation: clearly explained. 4th edition. Cambridge, MA: Elsevier, 2016. ISBN 978-0128043998.

[6] KEITH, Mike a Merrick SCHNICARIOL. Pro JPA 2: Mastering the Java Persistence API. New York: Apress, 2009. ISBN 978-1430219569.

[7] LINWOOD, Jeff. a Dave. MINTER. Beginning Hibernate. 2nd ed. New York: Apress, c2010. ISBN 978-1-4302-2851-6.

[8] MCKENZIE, Cameron. Hibernate made easy: simplified data persistence with Hibernate and JPA (Java persistence API) annotations. 3rd print. S.l.: Hiberbook.com, 2008. ISBN 9780615201955.

[9] Pavel Palát: Java EE organizér – softwarová architektura, bakalářská práce, Brno, FIT VUT v Brně, 2009[cit. 2017-11-12]

[10] ANDREW LEE RUBINGER AND BILL BURKE. Enterprise JavaBeans 3.1. 6th ed. Beijing: O'Reilly, 2010. ISBN 9780596158026.

[11] BOOTH, David, et al. W3C [online]. 11. 2. 2004 [cit. 2017-11-20]. Web Services Architecture. Dostupné z WWW: <http://www.w3.org/TR/ws-arch/>.

[12] KALIN, Martin. Java Web services: up and running. Second edition. Sebastopol, California: O'Reilly, 2013. ISBN 9781449365110.

[13] GOURLEY, David. a Brian. TOTTY. HTTP: the definitive guide. Sebastopol, CA: O'Reilly, 2002. ISBN 9781565925090.

[14] LUDIN, Stephen. Learning HTTP/2O', CA: O'Reilly, 2017. ISBN 9781491962442

[15] RICHARDSON, Leonard a Michael AMUNDSEN. RESTful Web APIs. Beijing: O'Reilly, 2013. ISBN 978-1449358068.

[16] Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. s.l. : University of California, Irvine, 2000

[17] Sajel Kumar DAS. Mobile handset design. Singapore: John Wiley & Sons (Asia), 2010. ISBN 9780470824696.

SEZNAM PŘÍLOH

A Zdrojové kódy aplikací **45**

A ZDROJOVÉ KÓDY APLIKACÍ

Zdrojové kódy serverové a klientské aplikace jsou dostupné na serveru GitHub. Odkaz pro serverovou aplikaci je: https://github.com/vendybike/UserServer.git. Pro Klientskou aplikaci: https://github.com/vendybike/UserClientFX.git.