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

Diplomová práce

Objektově relační mapování Oracle TopLink

Plzeň 2013 Bc. Jan Plaček

Prohlášení

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

V Plzni dne 23. června 2013

Bc. Jan Plaček

Abstract

This diploma thesis deals with technologies for object-relational mapping in Java programming language. After the basic description of the terms used in this domain, the focus is transferred to technologies based on the Java Persistence API. The thesis describes mainly the EclipseLink framework (formerly Oracle TopLink) that is the reference implementation of JPA and the JBoss Hibernate platform. In the next part of this thesis, there is the explanation of these technologies in the practical example of real application for the parcel delivery area. Achieved results are also discussed in the conclusion part of this thesis, as well as the comparison of both frameworks.

Obsah

Kapitola 1

Úvod

Relační databáze jsou stále hojně používaným řešením pro ukládání velkého množství dat. Veškeré výhody tohoto způsobu se ani v současné době stále nepodařilo překonat a jejich hojné nasazení a neustálý vývoj tuto situaci ještě komplikuje. Na straně druhé se ale nelze vyhnout ani masivnímu nástupu objektově orientovaných programovacích jazyků, které do značné míry odráží obraz reálného světa a usnadňují tím jak samotnou analýzu problému a představu o vytvářené aplikaci, tak její samotnou implementaci.

Objektově relační mapování má za úkol zajistit propojení těchto dvou rozdílných světů, tedy světa tabulek relační databáze se světem objektů v samotném programu a uspořit tím čas potřebný pro vývoj aplikací orientovaných na zpracování dat.

Cílem diplomové práce je objasnit problematiku objektově relačního mapování v jazyce Java s ohledem na její aktuální stav a především zhodnotit možnosti, které nabízí specifikace Java Persistence API 2.0. Po vysvětlení základních pojmů z této oblasti a objasnění samotného významu objektově relačního mapování, práce přechází v popis a použití samotného rozhraní Java Persistence API. V další kapitole je představena dvojice rámců EclipseLink a Hibernate, které toto rozhraní implementují a představeny jejich odlišnosti oproti specifikaci.

Další kapitoly jsou již zaměřeny ryze prakticky na samotnou tvorbu reálné aplikace a migraci její datové vrstvy mezi oběma rámci. Na samotný závěr je provedeno zhodnocení obou technologií a představeny rozdíly, které se během implementace podařilo nalézt.

Kapitola 2

Význam objektově relačního mapování

2.1 Objektově orientovaný přístup

Objektově orientované programování (dále jen OOP) umožňuje ve vývoji software zohlednit reálný svět. Původní paradigma imperativního programování je ve většině jazyků podporujících OOP sice stále zachováno, program však již není tvořen pouze sekvencí po sobě jdoucích příkazů, nýbrž systémem provázaných entit (objektů), které komunikují mezi sebou.

Tvorba programu se tak stává z velké části modelováním objektů reálného světa, popisem jejich vlastností a chování, interakce s okolím či způsobů, jakými jsou vzájemně propojeny. Tento objektový přístup umožňuje řešit složité problémy jednoduchým způsobem, zachovává v kódu programu přehled a podporuje znovupoužitelnost.

Programátor ve skutečnosti vytváří pouze předpis pro své objekty, tzv. třídu, na jehož základě je za běhu programu vytvořen libovolný počet instancí. Každá třída, potažmo objekt, obsahuje libovolný počet atributů (vlastností), které reprezentují jeho vnitřní stav. Tyto atributy mohou být tvořeny jak základním datovým typem, tak odkazem na jiné objekty. Dále je možné definovat metody (operace), jež umožňují s objektem pracovat a jeho stav měnit. Třídy mohou být rovněž děděny jedna od druhé, kdy podtřída obsahuje atributy i metody svého rodiče. Z hlediska aplikační logiky je tedy objektový přístup velmi výhodný.

2.2 Serializace

Realizované instance tříd (objekty) jsou uchovávány v operační paměti počítače. Stav těchto instancí, objektů, je však zachován pouze v době běhu programu a po jeho ukončení je často nezbytné tento stav uložit a následně zrekonstruovat pro nový běh, případně tento stav přenést na jiný počítač skrze počítačovou síť.

Postup převodu objektu do podoby vhodné pro paměťové či přenosové médium se nazývá serializace. Zpětná rekonstrukce objektu naopak deserializace. Základním úkolem serializace je tedy zajištění konzistence dat a jejich správné reprezentace při deserializaci, jelikož se třída před deserializací objektů může změnit (např. některé atributy mohou být přidány či odebrány).

Použití serializace pro persistenci objektů je však nevhodné pro jejich následné zpracování. Data jsou uložena v binární podobě a aby je bylo možné opětovně použít, je nezbytné veškerá data před jejich zpracováním deserializovat zpět do paměti. Při velkém množství dat, zejména s ohledem na omezenou kapacitu operační paměti, je takový přístup velmi nevýhodný.

2.3 Relační databáze

Historie relačních databází sahá dle [1] až do roku 1969. Tehdy používaný síťový a hierarchický databázový model se ukázaly nedostatečné vzhledem k množství záznamů, které bylo potřeba do databází ukládat. Dr. Edgar F. Codd z IBM tak vyvinul na základě teorie množin a predikátové logiky relační model [2], který měl veškerá negativa těchto modelů, jako je redundance či inkonzistence, odstranit.

Základním stavebním kamenem relační databáze jsou tabulky (relace). Řádky těchto tabulek pak představují jednotlivé záznamy. Záznam lze dále rozdělit na atributy (sloupce tabulky), jimž je při definici tabulky přiřazen datový typ. Každá tabulka v databázi by rovněž měla mít definován takzvaný primární klíč, tedy vyhrazený sloupec, případně kombinaci sloupců, jejichž hodnotou lze jednoznačně identifikovat každý záznam. Vztahy mezi tabulkami jsou poté realizovány prostřednictvím cizích klíčů, tedy odkazů na primární klíče z jiných tabulek.

Relační databázový model, potažmo relační databáze, mají již čtyřicetiletou historii a v současnosti jsou stále nejpoužívanější formou k uchování velkého množství dat. Ačkoliv byly vyvinuty databáze založené na čistě objektovém principu [3] (například ObjectDB, Db4o, Caché), stále nedosahují kvalit relačních databází a jejich nevýhody, jako je zaměření

na konkrétní programovací jazyk, velké rozdíly mezi jednotlivými produkty v principu ukládání dat i přístupu k datům, obtížná změna schématu či migrace, brání jejich masovému nasazení. I přes to však pomalu nachází uplatnění v některých oblastech, jako jsou zdravotnictví, vědecká a výzkumná pracoviště, CAD a GIS systémy a podobně [4]. Jejich vývoj je však stále na začátku a relačním databázím se v současné praxi nelze vyhnout.

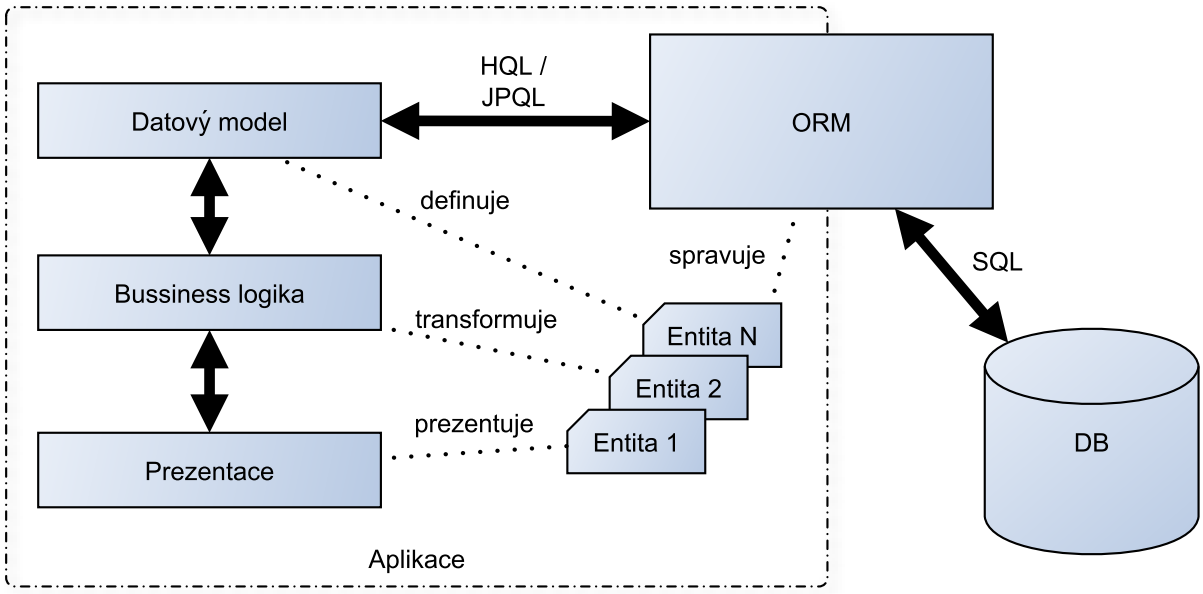
2.4 Objektově relační mapování

Objektově relační mapování (dále jen ORM) je technika, při níž dochází k automatické konverzi mezi záznamy v relační databázi a objekty objektově orientovaného programovacího jazyka. Základní funkcí ORM je tedy zejména zajištění synchronizace mezi aplikačními objekty v paměti počítače a příslušnými řádky tabulek relační databáze. Současně ORM zajišťuje automatickou konverzi mezi datovými typy databáze a programovacího jazyka a usnadňuje zajištění jejich konzistence podporou databázových transakcí a zámků.

Důvodem použití objektově relačního mapování je dle [14] především produktivita při psaní kódu, menší zdrojový kód a v neposlední řadě i nárůst výkonu aplikace, neboť vrstva objektově relačního mapování je sice multiplatformní a nezávislá na použité databázi, ale zároveň může pro tu či onu databázi zajistit optimalizaci dotazů či správné použití JDBC ovladačů.

Objektově relační mapování nabízí široké možnosti z hlediska vývoje aplikace. Je možné generovat třídy programovacího jazyka ze schématu databáze, případně obráceně – ze tříd daného jazyka generovat automaticky tabulky relační databáze, přičemž dědičnost nebo vazby typu M:N jsou automaticky rozkládány na rozkladové tabulky.

Princip vývoje aplikace využívající ORM je znázorněn na obr. 2.1. Objektově relační mapování působí jako prostředník mezi datovou vrstvou aplikace a relační databází. Entity definované v programovacím jazyku jsou automaticky spravovány a synchronizovány s databází a jejich stav je ukládán do tabulek. V případě potřeby jsou data z tabulek ORM poskytovatelem nataženy a v paměti vytvořeny objekty, se kterými lze v programu dále pracovat. Dotazování probíhá přímo na ORM poskytovatele prostřednictvím dotazovacího jazyka podobného SQL, popřípadě na základě definovaných kritérií. ORM poskytovatel dotaz přeloží do SQL podoby vhodné pro použitou databázi a zajistí jeho vykonání.



Obrázek 2.1: Objektově relační mapování

Pro programovací jazyk Java existuje velké množství knihoven pro objektově relační mapování. Nejpoužívanější je patrně Hibernate [15], dále pak například OpenJPA [5] či Ebean [6]. Velmi používaným frameworkem je rovněž Oracle TopLink [7], jehož kód je v současné době sloučen s produktem EclipseLink [18] a stal se tak referenční implementací Java Persistence API.

Kapitola 3

Java Persistance API

Java Persistence API 2.0 (dále jen JPA) [12] je rozhraní pro objektově relační mapování v programovacím jazyce Java, jež je součástí specifikace Enterprise Java Beans 3.0 [8]. Není však svázáno s kontejnerem Java Enterprise Edition a může být použito i v prostředí Java Standard Edition. JPA integruje množství konceptů Java Data Objects [16] a Hibernate. Konfigurace JPA je zajištěna prostřednictvím anotací tříd, případně XML soubory.

3.1 Plain old Java object

Jako Plain old Java object (POJO) jsou označovány běžné třídy jazyka Java. Tento termín byl definován roku 2000 Joshem MacKenzie, Rebbeccou Parsons a Martinem Fowlerem [9]. Jedná se o třídy, které nejsou závislé na žádné externí knihovně či frameworku, tedy neobsahují žádnou specifickou dědičnost, neimplementují specifická rozhraní a neobsahují žádné technologicky specifické anotace. POJO tedy obsahuje pouze atributy a příslušné *gettery* a *settery*. Následující listing 3.1 ukazuje, jak může jednoduchý Plain old Java object vypadat.

```
public class User {
  private String userName;
  private int age;

  public void setUsername(String userName) {
    this.userName = userName;
  }
  public String getUserName() {
```

```
    return this.username;
  }

  public void setAge(int age) {
    this.age = age;
  }
  public int getAge() {
    return this.age;
  }
}
```

Listing 3.1: POJO ukazka

3.2 JavaBean

JavaBean je znovupoužitelná komponenta programovacího jazyka Java. Z hlediska implementace se jedná o běžnou třídu (POJO), která však splňuje následující vlastnosti:

- přístup k atributům třídy je zajištěn prostřednictvím veřejných přístupových metod,
- instanci třídy lze vytvořit voláním bezparametrického konstruktoru,
- třída implementuje rozhraní `Serializable`.

Požadavek na přístupové metody (tzv. *gettery* a *settery*) vyplývá ze samotného principu zapouzdření, tedy skrytí skutečného stavu objektu a poskytnutí přístupu pouze k takové množině atributů či metod, která je skutečně zapotřebí. Takovým přístupem je zajištěna robustnost, stav objektu je konzistentní a validní, lze vynutit kontrolu vstupních dat a výstupní data například transformovat.

Názvy přístupových metod atributů JavaBeanu musí splňovat konvence jazyka Java, tedy název metody např. `getAtribut()` pro *getter* či `setAtribut()` pro *setter*. U přístupové metody vlastnosti datového typu `boolean` je rovněž povoleno použití názvu `isAtribut()`. *Gettery* nesmí obsahovat žádný parametr, *settery* naopak parametr právě jeden, a to pouze hodnotu přiřazovanou příslušnému atributu.

V prostředí Java EE aplikací se velmi často využívá principu reflexe skrze Reflection API. Jedná se o přístup k atributům a metodám objektu v době běhu programu, bez

potřeby znalosti jejich názvů v době kompilace zdrojového kódu. Pro tento druh přístupu k objektům je použití přístupových metod rovněž velmi vhodné až nezbytné.

Aplikační kontejner obyčejně obsluhuje velké množství klientů, přičemž komunikace s klientem probíhá na principu požadavek – odpověď, anglicky *request – response*. Pro každý takový požadavek musí aplikační kontejner vytvořit nový výchozí stav, tedy zavést a provázat potřebné JavaBeany. Bezparametrický konstruktor je zde jednoduché a fungující řešení, jak tvořit nové instance JavaBeanů.

Pro každého klienta lze vytvořit vlastní prostředí, anglicky session, které je zachováno mezi jednotlivými požadavky. Zajištění serializace JavaBeanu je velice důležité z hlediska jeho životního cyklu v aplikačním kontejneru. Požadavky klientských aplikací na aplikační kontejner přichází namátkou a nutnost udržet aktivní session pro každého klienta klade vysoké nároky na operační paměť. Mezi dvěma požadavky jednoho klienta v rámci session tak může být značný časový odstup a je nezbytně nutné, aby aplikační kontejner prováděl persistenci JavaBeanů například na pevný disk.

Aplikační server rovněž nemusí být tvořen pouze jedním strojem. V případě aplikačního clusteru tvořeného více servery je serializace nezbytná také pro zajištění přenosu JavaBeanů mezi jednotlivými uzly.

3.3 Entita

V souvislosti s Java Persistence API je spíše než pojem *JavaBean* používán pojem *entita*. Každá entita tak reprezentuje tabulku relační databáze, sloupce tabulky odpovídají atributům entity a instance entity zde představuje jeden řádek příslušné tabulky. Jedná se opět o běžnou třídu jazyka Java, na kterou jsou kladeny požadavky obdobné jako na JavaBean:

- třída je označena anotací `@Entity` z balíčku `javax.persistence`, nebo je označena jako entita v konfiguračním XML souboru,
- dědičnost je povolena od entitní i neentitní třídy,
- třída ani žádná její metoda nesmí být deklarována jako `final`,
- přístup k atributům třídy je zajištěn prostřednictvím veřejných přístupových metod,
- instanci třídy lze vytvořit voláním bezparametrického konstruktoru,

- třída implementuje rozhraní `Serializable`, pokud s entitou pracuje session bean typu `Remote`.

Další požadavek je kladen na datové typy atributů entitní třídy. Pokud není atribut označen anotací `@Transient`, dojde automaticky k mapování atributu na příslušný sloupec databázové tabulky, přičemž datový typ atributu musí být jedním z těch, které JPA poskytovatel dokáže namapovat:

- primitivní datové typy, jejich obalovací třídy a `java.lang.String`,
- datový typ výčet,
- pole typu `byte[]` a `char[]` či `Byte[]` a `Character[]`,
- `BigInteger` a `BigDecimal` z balíčku `java.math.*`,
- `Date`, `Calendar` z `java.util.*`,
- `Date`, `Time` a `TimeStamp` z `java.sql.*`,
- jiné entity třídy či kolekce entit,
- třídy anotované `@Embeddable`.

Různé implementace JPA (tedy jak EclipseLink, tak Hibernate) umožňují nad rámec specifikace i mapování vlastních datových typů, vždy je však potřeba implementovat příslušné konvertory. Přesto ve většině případů je nabídka JPA dostatečná a mapování složitějších struktur lze obejít skrze rozklad do jiné entity a její vložení anotací `@Embedded` (viz dále v textu).

3.4 EntityManager

EntityManager je jedno z nejdůležitějších[1] rozhraní Java Persistence API. Poskytuje metody pro správu instancí entit jako je řízení životního cyklu, synchronizace entit s databází, použití cache, zamykání, provádění dotazů, transakcí a podobně.

[1] Z pohledu programátora aplikace, která využívá JPA.

3.4.1 Kontext persistence

Každá instance EntityManageru je asociována s tzv. kontextem persistence. Persistentní kontext je množina instancí entitních tříd, jejichž identita je v rámci kontextu unikátní. EntityManager sleduje stav entit v kontextu a provádí jejich aktualizaci a persistenci do databáze, a to dle nastavení flush módu buď automaticky nebo na žádost uživatele. Kontext persistence se dělí na dva typy:

- persistentní kontext ohraničený transakcí (anglicky transaction scoped persistence context),
- rozšířený persistentní kontext (extended persistence context).

Persistentní kontext ohraničený transakcí je řízen aplikačním serverem. Životnost takového kontextu je pak ohraničena trváním jedné Java Transaction API (dále jen JTA) transakce. Po skončení transakce je persistentní kontext zrušen a veškeré instance entit jsou odpojeny. Tento typ kontextu je používán v bezestavových session beanech. Rozšířený persistentní kontext naopak dobou trvání transakce omezen není. Využití nachází naopak ve stavových session beanech.

3.4.2 Získání instance

Získání instance EntityManageru lze provést dvěma způsoby. Preferovaný způsob v prostředí Java EE je nechat aplikační server injektovat instanci přímo na místo, kde je potřeba (většinou jako atribut session beanu):

```
@PersistenceContext(unitName = "jmenoJednotky")
private EntityManager entityManager;
```

K injektáži rozšířeného persistentního kontextu je potřeba nastavit atribut anotace:

```
@PersistenceContext(type = PersistenceContextType.EXTENDED)
```

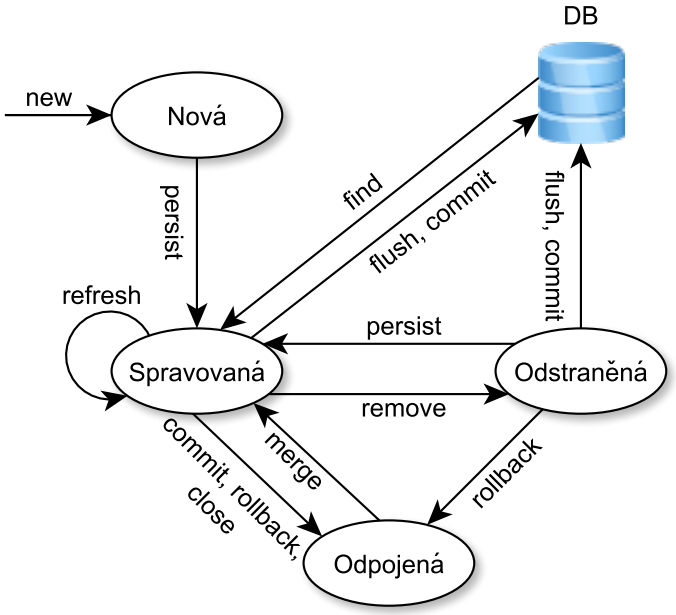
Druhým způsobem, použitelným zejména v prostředí Java SE, je vytvoření instance EntityManageru továrnou:

```
EntityManagerFactory factory = Persistence
  .createEntityManagerFactory("jmenoJednotky");
```

```
EntityManager entityManager = factory.createEntityManager();
```

3.4.3 Řízení životního cyklu entity

Životní cyklus entity je řízen nezávisle na životním cyklu aplikace. Každá entita se nachází v jednom ze stavů na obr. 3.1. Životní cyklus entity je řízen metodami třídy `EntityManager`, které na obrázku odpovídají přechodům mezi jednotlivými stavy.



Obrázek 3.1: Životní cyklus entity

Stav *Nová* odpovídá nově vytvořené instanci entitní třídy. Zařazení entity do kontextu persistence se provede voláním metody `persist()`. Od této chvíle, ze stavu *Spravovaná*, řídí životní cyklus entity EntityManager, jehož prostřednictvím lze provádět synchronizaci kontextu s databází metodou `flush()`. Pro odstraněné entity (metoda `remove()`) je vyhrazen zvláštní stav *Odstraněná*. Fyzické odstranění entity z databáze je nezbytné opět potvrdit metodou `flush()`. Ve stavu *Odpojená* se nachází entita, která je z kontextu dočasně vyřazena, a to buď z důvodu, že synchronizace s databází není nutná (entita je ve stejném stavu jako v databázi) nebo si synchronizaci uživatel nepřeje. Zařazení entity zpět do kontextu lze pak zajistit voláním `merge()` (objektová obdoba SQL příkazu

update). Metoda `refresh()` provede občerstvení entity. Dojde tedy k přepsání lokálních změn hodnotami z databáze.

3.4.4 Další metody

Kromě metod uvedených na obrázku 3.1 nabízí `EntityManager` několik dalších užitečných metod. Mezi ně patří především:

- `contains()` – ověří, zda je zvolená entita v kontextu persistence,
- `detach()` – vyřadí zvolenou entitu z kontextu,
- `clear()` – vyřadí z kontextu všechny spravované entity,
- `isOpen()` – ověří, zda je kontext persistence otevřený,
- `close()` – počká na dokončení všech aktivních transakcí, vyřadí všechny entity a následně uzavře kontext,
- `setFlushMode()` – nastaví flush mód na výchozí hodnotu `FlushModeType.AUTO` nebo `FlushModeType.COMMIT` (synchronizace entit s databází pak probíhá buď automaticky nebo explicitně voláním metody `flush()`),
- `unwrap()` – vrátí instanci zvolené třídy pro přístup ke specifickému API poskytovatele persistence.

Na uzavření EntityManageru metodou `close()` je potřeba dávat pozor, neboť jeho opětovné otevření již není možné a je nezbytné získat novou instanci. Při použití injektáže EntityManageru anotací `@PersistenceContext` se tedy metoda `close()` nesmí volat, jelikož po uzavření EntityManageru již novou instanci není možné obdržet, životnost EntityManageru je zde řízena aplikačním serverem. Naopak při použití instance EntityManageru obdržené továrnou EntityManagerFactory je rozhodnutí, kdy EntityManager uzavřít, ponecháno na uživateli a volání metody `close()` je nezbytné, aby byla uzavřena veškerá realizovaná spojení a uvolněny držené prostředky.

3.4.5 Transakce

Obsluhu transakcí lze v Java Persistence API řešit několika různými způsoby. Nejjednodušším způsobem, a v prostředí Java SE také jediným, je použití rozhraní `EntityTransaction`, které poskytuje následující metody:

- `begin()` – ohraničí počátek transakce,
- `commit()` – ukončí transakci a potvrdí provedené změny,
- `rollback()` – ukončí transakci a vrátí veškeré změny do stavu před spuštěním transakce,
- `isActive()` – vrací, zda je transakce aktivní (transakce je aktivní od spuštění transakce voláním `begin()` do jejího ukončení voláním `commit()` nebo `rollback()`),
- `setRollbackOnly()` – označí transakci pouze pro rollback (zajistí, že i při ukončení transakce metodou `commit()` se místo potvrzení transakce zavolá `rollback()` a veškeré změny jsou zamítnuty),
- `getRollbackOnly()` – vrátí, zda je transakce označena pouze pro rollback.

Instanci třídy implementující rozhraní EntityTransaction lze obdržet voláním metody `getTransaction()` nad instancí EntityManageru. Prostředí Java EE poskytuje rozhraní pro obsluhu transakcí JTA. Toto rozhraní umožňuje definovat a řídit atomické transakce skrze různé zdroje dat. Takovým zdrojem může být například databáze, komponenta J2EE Connector Architecture či služba Java Messaging Services. Jelikož jsou tyto zdroje od sebe izolovány, je potřeba řídit transakce tak, aby po úspěšném provedení commit na jednom zdroji a následném selhání transakce na zdroji jiném, byl na všech zdrojích proveden rollback a data zůstala všude v konzistentním stavu. JTA transakce jsou rozhraním JPA plně podporovány, postačí v souboru persistence.xml v nastavení persistentní jednotky odkázat datový zdroj JTA z aplikačního serveru a zvolit typ transakce JTA:

```
<persistence-unit name="jmenoJednotky" transaction-type="JTA">
  <jta-data-source>jdbc/datovyZdroj</jta-data-source>
  ...
</persistence-unit>
```

Transakce JTA mohou být řízeny buď automaticky na úrovni aplikačního serveru (anglicky Container Managed Transactions, neboli CMT) nebo uživatelsky na úrovni programu (Bean Managed Transactions, BMT). Ve výchozím nastavení JTA transakci řídí aplikační server a je ohraničena voláním metody session beanu. Transakce je započata těsně před voláním metody a commit proveden po jejím skončení. Pokud dojde v běhu

metody k výjimce, aplikační server provede rollback transakce. Problém nastává, pokud je v metodě jednoho session beanu volána metoda jiného session beanu. Má být v takovém případě původní transakce ukončena a spuštěna nová nebo má být volání metody součástí původní transakce? Toto chování lze ovlivnit deklarativním způsobem, totiž anotováním metod session beanu `@TransactionAttribute`. Podrobnosti k jednotlivým hodnotám lze nalézt v [11]. Probíhající transakci lze v kódu kdykoliv označit pro rollback metodou `setRollbackOnly()` třídy `EJBContext`. V takovém případě bude proveden rollback nezávisle na výsledku metody session beanu. Při použití CMT by neměly být v kódu aplikace nikde volány metody ohraničující transakce uživatelsky, tedy například metody `commit()`, `setAutocommit()` nebo `rollback()` rozhraní `java.sql.Connection` nebo `javax.transaction.UserTransaction`.

Pokud je v aplikaci požadována složitá aplikační logika nebo je CMT nevyhovující, je možné použít BMT a ohraničit transakci v kódu session beanu. Patřičné rozhraní poskytuje `javax.transaction.UserTransaction` a jeho instanci lze injektovat do atributu session beanu:

```
@Resource
private UserTransaction ut;
```

Jména metod rozhraní i jejich význam je obdobný jako u `EntityTransaction` (viz výše). Rozdíl je v tom, že transakce může mít širší význam a kromě persistentního kontextu může být commit či rollback propagován na více zdrojů obdobně jako u CMT.

3.4.6 Zámky

Enterprise aplikace bývají mnohdy multiuživatelské, velké množství uživatelů zde často pracuje nad stejnou databází a nemalý důraz je kladen rovněž na konzistenci dat. Tyto podmínky je důležité zohlednit již při návrhu aplikace, neboť pozdější úpravy jsou většinou velmi nákladné a vyžadují razantní úpravy kódu aplikace. Nezbytné je zde především obalit přístup do databáze transakcí. To však zdaleka neřeší veškeré problémy, které mohou v aplikaci nastat. Zejména ukládání do lokální cache nebo odstup mezi načtením dat do aplikace a uložením úprav zpět do databáze představuje problém. Data mohou být například v průběhu práce s aplikací na pozadí změněna jiným uživatelem. K řešení těchto problémů může pomoci použití zámků.

Java Persistence API poskytuje dva různé způsoby zamykání – optimistický a pesi-

mistický. Při použití optimistického způsobu, který je rovněž výchozím, jsou data před zápisem do databáze zkontrolována, a dojde-li v průběhu úprav dat v aplikaci k jejich změně v databázi, je vyhozena výjimka `javax.persistence.OptimisticLockException` a na transakci je zavolán rollback. Optimistické zamykání lze pro entitu zajistit definicí atributu s aktuální verzí:

```
@Version
protected int version;
```

Atributu odpovídá patřičný sloupec s verzí v databázi. Při modifikaci entity je její verze automaticky inkrementována a pokud verze aktualizované entity nesouhlasí s verzí řádku tabulky, je vyhozena výjimka. Atribut s verzí je programově přístupný, nedoporučuje se jej však uživatelsky měnit.

Kromě tohoto základního způsobu zamykání poskytuje JPA další možnosti nastavení `javax.persistence.LockModeType`:

- `OPTIMISTIC` – získá optimistický zámek pro všechny entity obsahující atribut anotovaný `@Version`,
- `OPTIMISTIC_FORCE_INCREMENT` – obdobné jako předchozí, pouze vynutí inkrementaci verze,
- `PESSIMISTIC_READ` – zamkne data dlouhodobým zámkem, zabrání jejich modifikaci či smazání z ostatních transakcí, ale nadále umožní čtení současné verze,
- `PESSIMISTIC_WRITE` – obdobné jako předchozí, pouze navíc znemožní i čtení dat z jiné transakce,
- `PESSIMISTIC_FORCE_INCREMENT` – jako `PESSIMISTIC_READ`, pouze vynutí inkrementaci verze,
- `READ` – totéž co `OPTIMISTIC` (ponecháno kvůli zpětné kompatibilitě),
- `WRITE` – totéž co `OPTIMISTIC_FORCE_INCREMENT` (rovněž pouze kvůli zpětné kompatibilitě),
- `NONE` – žádný zámek není aplikován.

Pro aplikaci zámku `PESSIMISTIC_READ` a `PESSIMISTIC_WRITE` není nutné, aby entita obsahovala atribut verze. Zamknout entitu lze explicitně voláním metody `lock()` rozhraní `EntityManager`. Parametrem je předána instance entity a zvolený typ zámku `LockModeType` z balíčku `javax.persistence`. Typ zámku lze rovněž specifikovat jako parametr při volání metod rozhraní `find()`, `refresh()` nebo jej aplikovat přímo na dynamický dotaz (instance `Query`) metodou `setLockMode()`. Pro pojmenovaný dotaz jej lze nastavit jako parametr anotace:

```
@NamedQuery(name = "applyLockToUser",
  query = "SELECT u FROM User u WHERE u.name = :name",
  lockMode = LockModeType.PESSIMISTIC_READ)
```

K použití pesimistického způsobu zamykání se váže rovněž nastavení doby, po jakou má aplikace čekat na uvolnění zámku. Jedná se o vlastnost `javax.persistence.lock.timeout`. Tuto dobu lze nastavit v souboru `persistence.xml`, případně ji nastavit jako další argument výše uvedených metod dynamického dotazu a `EntityManageru`.

3.5 Multiplicita

Provázanost entitních tříd mezi sebou se označuje jako tzv. multiplicita. Ve své podstatě se jedná o přenesení vazeb mezi tabulkami relační databáze na vazby mezi objekty. Rozhodující v tomto případě je tedy kardinalita vazeb neboli počet realizací (či instancí) entity na obou stranách vazby. Java Persistence API rozlišuje čtyři druhy multiplicit:

- One-to-one – instance právě jedné třídy obsahuje referenci na právě jednu instanci jiné třídy,
- One-to-many – instance jedné třídy obsahuje více referencí na instance jiné třídy,
- Many-to-one – více instancí jedné třídy obsahuje referenci na právě jednu instanci jiné třídy,
- Many-to-many – více instancí jedné třídy obsahuje více referencí na instance jiné třídy.

Tyto multiplicity mohou být realizovány buď jednosměrně (unidirectional), nebo obousměrně (bidirectional). Pokud je tedy multiplicita one-to-many realizována obousměrně,

v rodičovské třídě se jedná o multiplicitu one-to-many, naopak ve třídě podřízené pak o multiplicitu many-to-one. Jednotlivé typy multiplicit jsou dále vysvětleny na ilustrativním příkladu.

3.5.1 Multiplicita one-to-one

V případě multiplicity one-to-one obsahuje každá instance entity referenci na právě jednu instanci entity jiného typu, vazba je tedy jednoznačně určena mezi právě dvěma entitami. V ilustrativním příkladu tedy každý lékař má svou ordinaci a zároveň každá ordinace patří pouze jednomu lékaři (obr. 3.2).

Obrázek 3.2: Multiplicita one-to-one

3.5.2 Multiplicita one-to-many

Pokud jedna instance entity odkazuje na více instancí entity jiného typu, jedná se o multiplicitu one-to-many. Na příkladu lékař-ordinace-pacient si každý lékař vede kartotéku o svých pacientech, přičemž jednoho lékaře může navštěvovat více pacientů. Kartotéka lékaře je tak realizací multiplicity one-to-many (obr. 3.3).

Obrázek 3.3: Multiplicita one-to-many

3.5.3 Multiplicita many-to-one

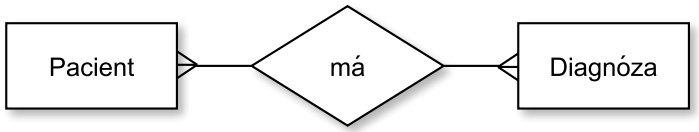
Multiplicita many-to-one je pouze obrácenou multiplicitou one-to-many. Vazba není realizována v rodičovské entitě, ale v entitě podřízené. Multiplicitu na uvedeném příkladu lze

ilustrovat navštívenkou, kterou dostane pacient při návštěvě lékaře. Navštívenek je větší množství, každému pacientovi náleží jedna, ale na všech je uveden stejný lékař (obr. 3.4).

Obrázek 3.4: Multiplicita many-to-one

3.5.4 Multiplicita many-to-many

Jakmile instance rodičovské entity odkazuje na více instancí podřízené entity a na každou podřízenou entitu může zároveň odkazovat více rodičovských entit, jedná se o multiplicitu many-to-many. Každému pacientovi lékař určí diagnózu, pacient může mít i několik problémů zároveň a diagnózy některých pacientů mohou být totožné (obr. 3.5).



Obrázek 3.5: Multiplicita many-to-many

3.6 Anotace entit

Parametry a způsoby mapování entitní třídy do relační databáze lze specifikovat v konfiguračním souboru `orm.xml` nebo anotováním entitní třídy a jejích atributů či metod. V této práci bude dále probrán pouze způsob konfigurace anotacemi, který je pro svoji přehlednost a jednoduchost obecně doporučován.

K označení třídy jako entity a k umožnění její persistence do databáze slouží anotace `@Entity` uvedená nad danou třídou:

```
@Entity
public class User implements Serializable {
  ...
}
```

Každá takto označená třída je mapována na tabulku v databázi se stejným jménem – v tomto případě `User`. Jméno tabulky lze však explicitně uvést jako atribut anotace `@Table(name="Person")`. Dále lze uvést další nepovinné atributy:

- `catalog` – jméno databázového katalogu,
- `schema` – jméno databázového schématu,
- `uniqueConstraints` – pole anotací `@UniqueConstraint` k definici unikátních omezení, které lze využít pouze k definici databázové struktury na základě předpisu entitních tříd.

Jsou-li data entity rozprostřena ve více tabulkách, lze zajistit souběžné načtení či uložení z několika tabulek anotací `@SecondaryTable(name="PersonDetail")`. V obou tabulkách je pak potřeba zajistit stejný primární klíč. Jména sloupců primárního klíče sekundární tabulky je možné uvést v anotaci `@PrimaryKeyJoinColumn`, jejich datové typy a hodnoty však musí v obou tabulkách souhlasit. Jedná se ve své podstatě o způsob mapování vazby 1:1, který je ale značně nevhodný, jelikož jsou data z obou tabulek načítána vždy současně, a to i v případě, že data ze sekundární tabulky nejsou potřeba. Jakmile je to jen možné, je vždy lepší vazbu rozdělit na více entit a odkaz na entitu sekundární tabulky uvést jako atribut primární entity.

3.6.1 Persistentní identita

Každá entita musí mít takzvanou persistentní identitu. Tato identita musí být unikátní a je mapována na primární klíč v databázové tabulce. Definici identity lze zajistit označením atributu entity anotací `@Id`. Identita, potažmo primární klíč, může být nastaven explicitně aplikací při vkládání entity, případně generován automaticky. K tomu slouží

anotace `@GeneratedValue` a její nepovinné atributy `strategy` pro volbu vhodné strategie pro danou databázi a `generator` k nastavení jména generátoru. Jméno generátoru lze zvolit libovolně, strategii lze zvolit jednu z následujících:

- `AUTO` – automatické nastavení dle použité databáze,
- `SEQUENCE` – použití sekvence jako generátoru identity,
- `TABLE` – jako generátor identity je použita tabulka databáze,
- `IDENTITY` – ke generování identity se použije automaticky inkrementovaný sloupec tabulky.

Dle použité strategie lze zvolený generátor doladit v anotaci `@SequenceGenerator` nebo `@TableGenerator`. Jedná se především o nastavení počáteční hodnoty generátoru, velikosti alokace pro zvýšení výkonu při vkládání většího množství entit a podobně. Zde je potřeba experimentovat a ověřit funkčnost s použitou databází. Problémem v databázi Oracle 11g se například ukázalo nastavení cache u sekvencí (výchozí hodnota 20), které způsobovalo inkrementaci hodnoty sekvence o hodnotu cache při každém vložení nové entity do databáze, ačkoliv ručním výběrem z pseudosloupce `NEXTVAL` byla sekvence inkrementována správně. Řešením může být cachování sekvencí vypnout nastavením parametru `NOCACHE`, v takovém případě však samozřejmě dochází k degradaci výkonu.

3.6.2 Základní mapování

Mapování základních datových typu jazyka Java probíhá plně automaticky a atributy entity, které se nemají mapování účastnit musí být anotovány jako `@Transient`, v opačném případě budou považovány za stejnojmenné sloupce patřičné tabulky. Výchozí způsob mapování lze změnit prostřednictvím dalších anotací. Anotace `@Basic` obsahuje atribut `fetch` k případnému vynucení opožděného načítání nastavením hodnoty `FetchType.LAZY` a dále booleovský atribut `optional`, jež umožní u ne-primitivních datových typů zakázat nebo povolit do patřičného sloupce tabulky uložení hodnoty `NULL`. Anotace `@Column` slouží k doladění mapování přímo vzhledem ke struktuře tabulky. Nabízeny jsou tyto nepovinné atributy:

- `name` – jméno sloupce tabulky,
- `unique` – zda má být nad sloupcem vytvořen unikátní index,

- `nullable` – zda je povoleno do sloupce vkládat hodnotu `NULL`,
- `insertable` – zda je sloupec zahrnutý v generovaném SQL insert dotazu,
- `updatable` – zda je sloupec zahrnutý v generovaném SQL update dotazu,
- `columnDefinition` – specifikace SQL pro definici sloupce,
- `table` – jméno tabulky, ve které se sloupec nachází,
- `length` – délka pro sloupce obsahující řetězec,
- `scale`, `precision` – rozsah a desetinná přesnost pro sloupce obsahující číslo.

Mapování atributů obsahujících datum nebo čas pro datový typ `java.util.Date` nebo `java.util.Calendar` se provádí anotací `@Temporal`. V jejím atributu `value` lze nastavit mapování pouze času (hodnota `TIME`), data (`DATE`) nebo celé časové značky (`TIMESTAMP`).

Pole primitivních datových typů se ukládají jako takzvaný Large Object (LOB) uvedením anotace `@Lob`. Na základě typu pole je pak automaticky rozhodnuto, zda se použije SQL datový typ `Clob` pro řetězce a pole znaků, nebo `Blob` pro pole bytů. Anotací `@Lob` lze vyjma polí označit i jakýkoliv jiný atribut, jehož datový typ implementuje rozhraní `Serializable`. Serializace i deserializace probíhá samozřejmě automaticky.

Výčtový datový typ `enum` lze mapovat buď ordinální hodnotou instance prvku výčtu nastavením anotace `@Enumerated(value=EnumType.ORDINAL)` nebo jako řetězec obsahující celý název prvku (`value=EnumType.STRING`).

3.6.3 Multiplicita

Vazby mezi entitami jsou podstatou celého ORM. V databázi jsou tyto relace mezi tabulkami vyjádřeny v podobě primárních a cizích klíčů, v podobě objektů by však tento způsob byl zoufale nevhodný. Objektově relační mapování má tedy za úkol zajistit realizaci vazeb v paměti přímo, a to prostřednictvím referencí mezi jednotlivými objekty.

Jednotlivé typy vazeb lze namapovat vazebními anotacemi `@OneToOne`, `@ManyToOne`, `@OneToMany` a `@ManyToMany`. Atribut entity pro vazby one-to-one a many-to-one je v případě jednosměrného mapování realizován pouhou referencí na rodičovskou entitu. Naopak vazby one-to-many a many-to-many se implementují v rodičovské entitě kolekcí referencí na podřízené entity. Tento způsob mapování se nazývá jednosměrný. Zde je nutné dodat, že

rodičovskou entitou je vždy nazývána entita, ve které je vazba realizována, a tedy jí příslušející mapovaná tabulka obsahuje sloupec s cizím klíčem. Pokud se název vazebního atributu rodičovské entity neshoduje s názvem sloupce tabulky obsahující cizí klíč, je nezbytné jej uvést v anotaci `@JoinColumn(name="MY_COLUMN")` nad příslušným atributem, v opačném případě je vazba realizována automaticky. Vazba many-to-many je svým způsobem specifická, neboť ji nelze realizovat bez takzvané rozkladové tabulky. Jméno rozkladové tabulky se uvádí v anotaci `@JoinTable` spolu s názvy vazebních sloupců v atributech `joinColumns` a `inverseJoinColumns`:

```
@Entity
public class Author implements Serializable {
  @Id
  private int id;

  @ManyToMany
  @JoinTable(name="AUTHOR_BOOK",
    joinColumns=@JoinColumn(name="ID_AUTH"),
    inverseJoinColumns=@JoinColumn(name="ID_BOOK")
  )
  private Collection<Book> books;
   ...
}
```

Všechny typy vazeb se však mohou mapovat i obousměrně, tedy jak na entitě rodičovské, tak na entitě podřízené, přičemž vazbě one-to-many v rodičovské entitě z principu odpovídá v entitě podřízené vazba many-to-one a obráceně, a je samozřejmě nezbytné v podřízené entitě použít odpovídající anotaci. Jelikož v podřízené entitě není k dispozici informace o vazebním sloupci, přidává se k anotaci vazby jméno vazebního atributu rodičovské entity (atribut `mappedBy`). K příkladu výše by tedy v entitě `Book` byla vazba na druhé straně realizována následovně:

```
@Entity
public class Book implements Serializable {
  @Id
  private int id;
```

```
  @ManyToMany(mappedBy="books")
  private Collection<Author> authors;
   ...
}
```

V tomto příkladu je použita generická kolekce a název rodičovské entity je zde uvozen v lomených závorkách. Pokud by kolekce byla deklarována bez uvedení datového typu prvků, JPA poskytovatel by neměl možnost rodičovskou entitu objevit. V takovém případě se nadřazená entita uvádí jako další atribut anotace:

```
@ManyToMany(mappedBy="books", targetEntity=Book.class)
private Collection authors;
```

Dalším důležitým parametrem vazby je způsob její inicializace. Ve výchozím nastavení jsou všechny vazby, jež jsou realizovány pouhou referencí na jinou entitu inicializovány okamžitě (takzvaný *eager* přístup) a při dotazu na rodičovskou entitu je okamžitě dotažena z databáze k ní příslušná entita podřízená. V místě, kde je vazba realizována kolekcí, je naopak výchozím způsobem opožděné načítání (*lazy* přístup) a odkazované entity jsou poskytovatelem JPA dotaženy až při prvním programovém přístupu do příslušné kolekce. Toto výchozí chování je samozřejmě možné změnit nastavením atributu vazební anotace na `fetch=FetchType.LAZY` nebo `fetch=FetchType.EAGER` a vynutit tak v případě potřeby okamžité či opožděné načtení.

3.6.4 Kaskádní operace

Operace prováděné nad rodičovskou entitou nejsou ve výchozím nastavení propagovány na podřízené entity. K zajištění databázové integrity může být vhodné některé operace provádět kaskádně, k tomu slouží atribut `cascade` u vazební anotace. K dispozici jsou tyto možnosti:

- `CascadeType.PERSIST` – společně s nadřazenou entitou budou persistovány i podřízené entity,
- `CascadeType.REMOVE` – pokud je nadřazená entita odstraněna, podřízené entity budou také současně odstraněny,

- `CascadeType.REFRESH` – operace refresh bude prováděna kaskádně, s nadřazenou entitou se aktualizují i podřízené entity,
- `CascadeType.MERGE` – aktualizace podřízených entit bude vyvolána současně s aktualizací nadřazené entity,
- `CascadeType.ALL` – všechny operace jsou prováděny kaskádně.

S kaskádními operacemi je potřeba zacházet opatrně a používat pouze tam, kde mají své opodstatnění. Pokud je například mapována vazba faktura – řádek faktury, kaskáda zde má rozhodně smysl a je výhodné aktualizovat celou fakturu jako celek a při jejím odstranění odstranit i všechny její řádky. Naopak ve vazbě oddělení – zaměstnanec může být nebezpečné při zrušení oddělení odstranit všechny zaměstnance, kteří zde pracují.

V JPA verze 2.0 byl přidán k vazebním anotacím `@OneToOne` a `@OneToMany` další booleovský parametr `orphanRemoval`, který slouží k odstranění sirotků. Pokud je podřízená entita odstraněna, či je jinak ztracena reference na podřízenou entitu přímá reference nastavena na hodnotu `null` u vazby one-to-one nebo je entita odstraněna z příslušné kolekce u vazby one-to-many), bude při synchronizaci nadřazené entity provedeno odstranění podřízené entity. Jedná se tedy o jakési usnadnění práce, kdy na odstraňované entity není nutné volat explicitně `remove()`, ale JPA poskytovatel vše zařídí automaticky.

3.6.5 Zahrnutí tříd

Anotováním třídy entity jako `@Embeddable` bude zamezeno její samostatné persistenci do databáze. Takto označená entita však může být součástí jiné entity, a to v podobě jejího atributu. Atribut je nezbytné označit anotací `@Embedded`.

Tento způsob mapování je vhodný k definici složeného primárního klíče. Pole, jež mají být jeho součástí, je třeba oddělit do samostatné `Embeddable` třídy, a tu následně v podobě atributu s anotací `@EmbeddedId` použít jako identitu jiné entity.

K dispozici je rovněž podobné zahrnutí nepersistentní kolekce `Embeddable` třídy prostřednictvím anotace `@ElementCollection`. Ve své podstatě se jedná o mapování vazby 1:N s tím rozdílem, že `Embeddable` entita na vícečetné straně nemusí mít identitu, jelikož není mapována do databáze přímo, ale pouze s rodičovskou entitou. Použití toho způsobu mapování však nemá žádné reálné opodstatnění a lze jej plně nahradit mapováním vazby `OneToMany`.

3.6.6 Dědičnost

Natavení mapování entit využívajících dědičnost se provádí anotací `@Inheritance`. Zde jsou nabízeny tři možné strategie (atribut `strategy`):

- `SINGLE_TABLE` – výchozí hodnota, celá hierarchie dědičnosti je mapována do jedné tabulky, k určení potomka se použije sloupec určený v anotaci `@DiscriminatorColumn`, jehož hodnotu pro každého potomka lze specifikovat anotací `@DiscriminatorValue`,
- `TABLE_PER_CLASS` – každý potomek je mapován do své vlastní tabulky, každá tabulka obsahuje sloupce pro všechny atributy potomka, včetně těch, které podědí od svých předků,
- `JOINED` – každý potomek je mapován do své vlastní tabulky, ale tabulka obsahuje pouze sloupce k atributům, které jsou do potomka přidány, primární klíč této tabulky je zároveň cizím klíčem v tabulkách předků.

Je-li třída z hierarchie označena jako abstraktní, lze anotací `@MappedSuperclass` potlačit její persistenci do databáze, ale zároveň zajistit, že atributy této třídy budou mapovány v jejích potomcích. Další podrobné informace k mapování dědičnosti lze nalézt v JPA specifikaci [12].

3.6.7 Callback metody

Callback metody jsou vyvolány vždy před a po provedením nějaké akce EntityManagerem. Na metody entity lze použít následující anotace:

- `@PostLoad` – po volání metody `find()`, `refresh()` nebo vykonání Java Persistence Query Language dotazu (dále jen JPQL),
- `@PrePersist`, `@PostPersist` – před a po volání `persist()`,
- `@PreUpdate`, `@PostUpdate` – před a po aktualizaci (volání setteru nebo `merge()`),
- `@PreRemove`, `@PostRemove` – před a po odstranění entity voláním `remove()`.

Vykonání callback metod lze rovněž z entity delegovat na jinou třídu. Postačí entitní třídu označit anotací `@EntityListeners(Posluchac.class)` a ve třídě Posluchac poté označit libovolné metody se signaturou `void metoda(Entita e)` výše uvedenými anotacemi. Zaregistrovat lze rovněž globálního posluchače, který reaguje na callback metody

všech entit. Signatura metod posluchače je poté `void metoda(Object o)`. Globálního posluchače však nelze registrovat anotací, nýbrž pouze v souboru `orm.xml` (listing 3.2).[2]

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener class="cz.balicek.Posluchac"/>
    <entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Listing 3.2: Registrace globálního posluchače

3.7 JPA dotazy

3.7.1 Dotazy JPQL

Dotazování v Java Persistence API probíhá speciálním jazykem Java Persistence Query Language, který pokrývá základní možnosti jazyka SQL a je speciálně upraven pro dotazování v objektovém prostředí. JPQL je podmnožinou Hibernate Query Language (dále jen HQL) [15], tedy jakýkoliv JPQL dotaz je možné interpretovat jako HQL, ale opačně obecně nikoliv.

Jednoduchý JPQL dotaz, který vybere entitu `User` dle uživatelského jména vypadá následovně:

```
SELECT u FROM User u WHERE u.username = :username
```

Syntaxe je podobná jako u jazyka SQL, ovšem místo názvů tabulek se vždy uvádí názvy entitních tříd. Dále je vytvořena proměnná `u`, což je nezbytné pro další manipulaci s entitou, tedy její vybrání z databáze v klauzuli `SELECT` či odkazování na atributy entity v klauzuli `WHERE`. Odkaz na atribut se provádí skrz tečkovou notaci, jak je v objektovém prostředí běžné. Slovo uvozené dvojtečkou je považováno za parametr, jehož hodnota se nastavuje programově před vykonáním dotazu. Základní rozdíl JPQL a HQL je patrný již v tomto případě, neboť v HQL není klauzule `SELECT` povinná a dotaz může začínat klauzulí `FROM`.

[2] V souboru `orm.xml` je možné zaregistrovat i posluchače pro jednotlivé entity.

Přesto je obecně doporučeno i u jazyka HQL klauzuli `SELECT` uvádět, neboť dotaz neztrácí na přehlednosti a je na první pohled jasné, co je jeho výsledkem.

Jazyk JPQL podporuje většinu klíčových slov jazyka SQL, není tedy problém zapsat i složitější dotaz:

```
SELECT u FROM User
u WHERE u.active = true
  AND :minOrders < (SELECT COUNT(o)
        FROM u.orders
o WHERE o.type = cz.example.OrderType.WEB
      )
ORDER BY u.lastActivity DESC
```

Uvedený dotaz by vrátil seznam aktivních uživatelů, jejichž počet webových objednávek je větší než `minOrders`, entity jsou seřazeny sestupně dle času poslední aktivity. Na dotazu je patrné, že JPQL umožňuje použití řazení, pod-dotazů či agregačních funkcí. Dále je zde ukázáno použití filtrování dle výčtového datového typu (`OrderType`), které je nezbytné zapisovat plně kvalifikovaným způsobem, tedy s uvozením všech balíčků. Zajímavý je rovněž způsob, jakým je pod-dotaz napojen. Výběr entit `Order` je omezen přímo výběrem z vazebního atributu `u.orders` entity `User`. Typ a parametry vazby jsou již jednou specifikovány v podobě anotací entity, v tomto případě tedy není nutné napojení skrze cizí klíč explicitně uvádět. Veškeré možnosti jazyka JQPL spolu s dalšími podrobnostmi lze nalézt v [11].

3.7.2 Dynamické dotazy

JPQL dotaz lze v nejjednodušším případě vytvořit dynamicky při jeho spuštění v místě, kde je k dispozici instance `em` třídy `EntityManager`:

```
Query query = em.createQuery("SELECT u FROM User u");
List result = query.getResultList();
```

Výsledkem dotazu je negenerický list, bylo by tedy nezbytné při obsluze výsledků dotazu provádět přetypování na vybíranou entitu. Proto JPA poskytuje ještě kromě dotazu `Query`

ještě jeho typovanou variantu `TypedQuery`:

```
TypedQuery<User> query = em.createQuery("SELECT u FROM User u",
  User.class);
List<User> result = query.getResultList();
```

Návratovou hodnotou volání metody `getResultList()` je seznam instancí entit. Pokud je výsledkem dotazu pouze jedna instance, lze v JPA 2.0 volat metodu `getSingleResult()`. Ta vyhazuje výjimky `NoResultException` a `NonUniqueResultException` v případě, že výsledek dotazu nevrací žádnou entitu nebo je entit více. Následující příklad rovněž osvětluje způsob mapování pojmenovaných parametrů dotazu:

```
Query query = em.createQuery("SELECT u FROM User
u WHERE u.id = :userId");

query.setParameter("userId", 1);
User user = query.getSingleResult();
```

Vykonání akčního dotazu je rovněž možné, má však smysl pouze v ojedinělých případech, neboť lze stejnou funkčnost zajistit přímo metodami instance `EntityManager` nad příslušnou entitou. K vyvolání akčního dotazu slouží funkce `executeUpdate()`, výsledkem je počet aktualizovaných nebo smazaných instancí:

```
Query query = em.createQuery("DELETE FROM User
u WHERE u.username = ?1");

query.setParameter(1, "test");
int deleteCount = query.executeUpdate();
```

Příklad odstranění uživatele „test“ rovněž ukazuje použití očíslovaných parametrů. Ty jsou namísto dvojtečky uvozeny otazníkem a místo zadání jména parametru se používá číslování.

Použití dynamických dotazů má význam pouze v případě, že řetězec dotazu je sestaven v době běhu programu a při kompilaci není známý (dotaz se například mění v závislosti na uživatelském vstupu). Je potřeba si uvědomit, že řetězec dotazů musí být při každém

spuštění zkompilován JPA poskytovatelem do SQL pro danou databázi a teprve následně spuštěn. Z hlediska výkonu a pro lepší přehlednost zdrojového kódu je tedy mnohem výhodnější používat pojmenované dotazy.

3.7.3 Pojmenované dotazy

Pojmenovaný dotaz je běžný JPQL dotaz, který je staticky definován na úrovni zdrojového kódu entity. Jelikož jeho řetězec není k dispozici až v době běhu v místě vykonání, ale již při překladu samotného zdrojového kódu, může být při zavedení aplikace do aplikačního serveru jednoduše předkompilován do SQL a jeho vykonání tak urychleno. Pojmenovaný dotaz se zapisuje v anotaci `@NamedQuery` nad definici třídy entity:

```
@Entity
@NamedQuery(name="User.findAll", query="SELECT u FROM User u")
public class User implements Serializable {
  ...
}
```

Pokud je pojmenovaných dotazů k jedné entitě definováno více, musí být obaleny anotací `@NamedQueries`:

```
@NamedQueries({
  @NamedQuery(name="User.findAll",
    query="SELECT u FROM User u"),

  @NamedQuery(name="User.findByUsername",
    query="SELECT u FROM User u WHERE u.username = :username"),
})
```

Vyvolání pojmenovaného dotazu je obdobné jako u dotazu dynamického. V místě, kde je k dispozici `EntityManager` se zavolá jeho metoda `createNamedQuery()`:

```
User user = em.createNamedQuery("User.findByUsername")
    .setParameter("username", "test")
    .getSingleResult();
```

3.7.4 Kriteriální dotazy

Do Java Persistence API verze 2.0 byl přidán další možný způsob skládání dotazů, a to na základě takzvaného JPA Criteria API. Výhodou tohoto typu dotazů je, že nejsou zapisovány ve formě jazyka podobného SQL, ale přímo v jazyce Java. Programátor může být tedy částečně osvobozen od studia JPQL. Další výhodou je eliminace syntaktických chyb v dotazu, jelikož veškeré nesrovnalosti jsou odhaleny při překladu zdrojového kódu. K sestavení a vyvolání jednoduchého kriteriálního dotazu `SELECT u FROM User u` se používá instance třídy `CriteriaBuilder`:

```
CriteriaBuilder builder = em.getCriteriaBuilder();

CriteriaQuery<User> query = builder.createQuery(User.class);
query.select(query.from(User.class));

TypedQuery<User> typedQuery = em.createQuery(query);
```

Výsledkem je tedy opět instance `TypedQuery`, jehož metodami lze dotaz vykonat a obdržet požadované entity obdobně jako u pojmenovaného nebo dynamického dotazu.

Kapitola 4

Rámce pro ORM

Rámců pro objektově relační mapování v jazyce Java je poměrně rozsáhlé množství. Liší se samozřejmě poskytovaným rozhraním, způsobem práce a vnitřní implementací. Postupem času byla některá rozhraní standardizována, mezi ně patří například Java Data Objects (JDO) [16] nebo právě popisované Java Persistence API, které vzniklo oddělením od standardu EJB.

Některé rámce implementují rozhraní vlastní, jedná se například o produkty MyBatis [17] či Ebean [6], který nabízí jakousi jednoduchou alternativu k JPA či JDO. Vlastní cestou jde rovněž Hibernate, který je již delší dobu nejpoužívanějším produktem. V posledních verzích však Hibernate nabízí rovněž podporu JPA a jeho srovnání s EclipseLink je tedy nasnadě.

4.1 Hibernate

JBoss Hibernate je historicky nejúspěšnější řešení ORM v Javě. V dnešní době jde již o široké portfolio produktů zaměřených na vývoj datové vrstvy aplikace založené na relační databázi. Hibernate nabízí nástroje pro clusterizaci dat, fulltextové vyhledávání napříč doménovým modelem, validaci Java Beanů, vývojové nástroje a další technologie. Jedná se o ucelený produkt, který má širokou podporu ze strany vývojových nástrojů, aplikačních kontejnerů i vývojářů jiného softwaru (např. Spring). Hibernate je k dispozici kromě verze pro programovací jazyk Java i ve verzi pro Microsoft .NET, což může představovat určitou výhodu z hlediska nároků na znalosti programátora.

4.2 EclipseLink (Oracle TopLink)

Referenční implementace Java Persistence API 2.0, EclipseLink, vznikla uvolněním zdrojového kódu produktu Oracle TopLink [7]. Od verze TopLink 11g je EclipseLink jeho součástí jako poskytovatel persistence a TopLink jako takový se stává distribucí produktu EclipseLink spolu s nástrojem Coherence, který umožňuje L2 cachování a replikace dat v clusteru na bázi P2P. Kromě ORM řešení, EclipseLink nabízí technologie pro mapování objektů na XML a technologii DBWS, což je technologie pro definici webových služeb postavených na relační databázi deklarativním způsobem [18].

4.3 Rozšíření oproti JPA

V následujících sekcích budou popsána některá rozšíření ORM EclipseLink oproti specifikaci Java Persistence API 2.0, které uvádí [13], a provedeno srovnání s odpovídajícím řešením Hibernate, pokud existuje.

4.3.1 Multi-tenancy

Jedním z rozšíření, které EclipseLink nabízí oproti JPA specifikaci, je podpora pro takzvané multi-tenancy aplikace. Ve své podstatě se jedná o aplikace nabízející služby většímu množství uživatelů s odděleným datovým prostorem.

Datový model aplikací tohoto typu je většinou založen buď na oddělených databázích nebo schématech pro každého uživatele, nebo na společných tabulkách pro všechny uživatele, které jsou vertikálně rozděleny a každému uživateli odpovídá určitá množina řádků. V aplikační logice je pak nezbytné zajistit každému uživateli přístup pouze k jemu vymezené části datového prostoru. Řešení datového modelu je zde většinou omezeno na použití cizího klíče z tabulky uživatelů. Tento cizí klíč je pak součástí většiny dotazů takové aplikace, což zanáší do aplikace zbytečné složitosti.

V případě EclipseLink jsou multi-tenancy aplikace podporovány od verze Indigo 2.3. Řešení je velice jednoduché a postačí entitu označit anotací `@Multitenant`. Parametrem této anotace je pak zvolená strategie:

- bez parametru – tabulka entity je multiuživatelská,
- `SINGLE_TABLE` – tabulka je dělena horizontálně a veškeré dotazy jsou filtrovány dle identifikátoru uživatele, přičemž patřičný sloupec tabulky je specifikován anotací `@TenantDiscriminatorColumn`,
- `TABLE_PER_TENANT` – tabulky jednotlivých uživatelů jsou odděleny, a anotací `@TenantTableDiscriminator` lze dále určit dělení dle schématu, prefixu nebo sufixu tabulky,
- `VPD` – Virtual Private Database – veškeré dotazy jsou filtrovány, za filtrování však odpovídá patřičná databáze, která musí tuto technologii podporovat.

Identifikátor uživatele se pak předává jako parametr při vytvoření instance `EntityManager`u:

```
Map<String, Object> properties = new HashMap<String, Object>();
emProperties.set("eclipselink.tenant-id", "tenantId");
EntityManager em = emf.createEntityManager(properties);
```

U Hibernate se identifikátor uživatele zanáší rovněž již při otevření databázové session:

```
Session session = sessionFactory.withOptions()
        .tenantIdentifier(tenantId).openSession();
```

Zde je potřeba dodat, že instance `Session` u Hibernate představuje paralelu k rozhraní `EntityManager` z JPA 2.0, a pokud je v kódu injektována jeho instance, není problém obdržet příslušnou továrnu a session s podporou multi-tenancy vytvořit takto:

```
SessionImpl emSession = (SessionImpl) entityManager.getDelegate();
SessionFactory sessionFactory = emSession.getSessionFactory();
Session session = sessionFactory.withOptions()
  .tenantIdentifier(tenantId).openSession();
```

Způsob, jakým budou data uživatelů oddělena, lze v konfiguraci Hibernate nastavit vlastností `hibernate.multiTenancy`. Podporovány jsou prozatím strategie oddělených schémat (nastavení `SCHEMA`) a databází (nastavení `DATABASE`). Vždy je ale potřeba implementovat poskytovatele připojení, který zařídí obdržení specifického spojení pro přihlášeného uživatele. Strategie založená na vertikálním dělení tabulek (`DISCRIMINATOR`) je plánována až v Hibernate verze 5.0 [19], přesto lze i v současné verzi tuto strategii implementovat s Hibernate poměrně elegantně, a to použitím filtrů.

Prostřednictvím anotací entity `@FilterDef` a `@Filter` nebo `@FilterJoinTable` lze specifikovat filtry pro horizontální dělení tabulky a po nastavení filtru patřičné session již bude filtrování záznamů probíhat automaticky:

```
session.enableFilter("filterName")
  .setParameter("tenantFilter", "value");
```

4.3.2 Generátory identity

Kromě základních JPA strategií pro generování identity entity uvedených v kapitole 3.6.1 EclipseLink nabízí navíc generátor `@UuidGenerator` (Universally Unique Identifier). Jedná se o pseudonáhodně generovanou posloupnost 16 bytů, která je globálně unikátní. V databázi se ukládá jako pole bytů nebo 32 bytový řetězec a uplatnění nachází zejména v distribuovaných systémech, kde odpadá potřeba centralizovaného generátoru identity. Jako primární klíč databázové tabulky jej však z důvodu snížení výkonu nelze obecně doporučit (zejména čas potřebný pro jeho generování, větší datový tok, vyšší paměťové nároky, špatná optimalizace indexů pro datový typ pole či řetězec).

Hibernate rovněž podporuje generování identity jako UUID, dále pak GUID pro databáze MySQL a Microsoft SQL, strategii *increment*, při které se provádí pouze inkrementace maximální hodnoty identity, generátory založené na HiLo algoritmu a v neposlední řadě také strategii *select*, kde je identita přiřazena na základě volání databázového triggeru. Zde je paleta ORM Hibernate oproti EclipseLink velmi rozsáhlá.

4.3.3 Entity pouze pro čtení

EclipseLink anotací `@Readonly` lze nad entitou specifikovat, že přístup k ní je omezen pouze pro čtení, čímž se razantně zvýší výkon dotazů, neboť taková entita nebude při

výběru z databáze zařazena do persistentního kontextu. Přístup pouze pro čtení nalézá uplatnění například při migraci dat z jedné databáze na druhou, provádění replikace či transformace dat, zálohování či přesun dat z produkční databáze do datového úložiště.

Hibernate také podporuje přístup pouze pro čtení, celé řešení je zde však poněkud odlišné. Entity, jejichž třída je definována jako neměnná (anglicky *immutable*), jsou automaticky označeny pouze pro čtení. Přístup lze dále nastavit pro celou session metodou `Session.setDefaultReadOnly()`, pro prováděný dotaz `Query.setReadOnly()` nebo `Criteria.setReadOnly()`, případně pro entitu `Session.setReadOnly(entity, true)`. Nastavení přístupu k asociačním vazbám lze nalézt v [15].

4.3.4 Konvertory a transformace

EclipseLink rovněž rozšiřuje základní implementaci JPA o několik dalších záležitostí. Jednou z nich je možnost definice vlastních konvertorů pro úpravu mapování některých datových typů:

- `@Converter` – zavedení vlastní implementace konvertoru, obsahuje atributy pro nastavení jména konvertoru a třídy implementující rozhraní `Converter`, jejíž metody budou pro konverzi vyvolány,
- `@TypeConverter` – nastavením vstupního a výstupního datového typu umožňuje jednoduchou konverzi primitivních datových typů mezi datovým typem sloupce databáze a atributu entity,
- `@ObjectTypeConverter` – kromě konverze datového typu umožňuje i modifikovat obsah,
- `@StructConverter` – slouží ke konverzi mezi datovým typem `java.sql.Struct` a `java.lang.Object` (vhodné například pro geometrické datové typy PostgreSQL),

Zajistit konverzi definovaným konvertorem lze uvedením anotace `@Convert("myName")` nad vybraným atributem entity, kde `myName` je zvolené jméno konvertoru.

Konvertory mohou být velmi užitečné, vhodným užitím může být nastavení mapování booleovského datového typu jazyka Java pro databáze, ve kterých není implementován (například Oracle 11g), na číselné nebo znakové datové typy.

Ve výjimečných případech, kdy je potřeba provádět složitější transformace dat, například mapování několika sloupců databáze na jeden atribut entity, je řešením implementace transformátorů. Anotaci `@Transformation` lze dále doplnit o `@ReadTransformer`

a `@WriteTransformer` a specifikovat třídy nebo metody, které budou transformaci zajišťovat. Jelikož je ale transformace ve většině případů možné zajistit pouze přístupovými metodami entity nebo definicí konvertorů, není tento přístup v praxi příliš užíván.

Hibernate nabízí k mapování ve složitějších případech užití vlastních datových typů a v implementovaných metodách jednoho z rozhraní `Type`, `BasicType`, `UserType` nebo `CompositeUserType` provést požadované konverze. Následným zanesením třídy vlastního typu do konfigurace Hibernate je dále vše zajištěno automaticky.

Kapitola 5

Java Server Faces

Java Server Faces (dále JSF) je technologie založená na oddělení prezentační vrstvy webové aplikace od vrstvy aplikační. Uživatelské rozhraní je definováno XML šablonami, ze kterých je následně generován HTML a JavaScript kód. JSF umožňuje mapování stavu Java Beanů na komponenty uživatelského rozhraní, obsluhu událostí, validaci a konverze uživatelského vstupu, internacionalizaci a rovněž definici navigace po jednotlivých stránkách. Veškeré možnosti, jaké má programátor k dispozici, se odvíjí zejména od použité implementace, potažmo množiny komponent, technologie jako taková je tedy modulární a jednoduše rozšiřitelná. Velkou výhodou JSF je rovněž zabudovaná podpora technologie AJAX (Asynchronous JavaScript and XML). Podrobné informace, včetně množství praktických příkladů lze nalézt v [20].

5.1 Facelets

Šablony k JSF lze definovat jako Java Server Pages (JSP) nebo od JSF 2.0 prostřednictvím takzvaných Facelets. Jedná se o šablony postavené čistě na technologii XML, nejsou tedy kladeny žádné další nároky na znalost specifických vlastností JSP. Facelet šablona je obyčejný XML soubor s definicí doctype XHTML 1.0 Transitional, ve kterém lze používat běžné XHTML značky. Další možnosti a funkce jsou k dispozici definicí jmenného prostoru a prefixu v záhlaví dokumentu:

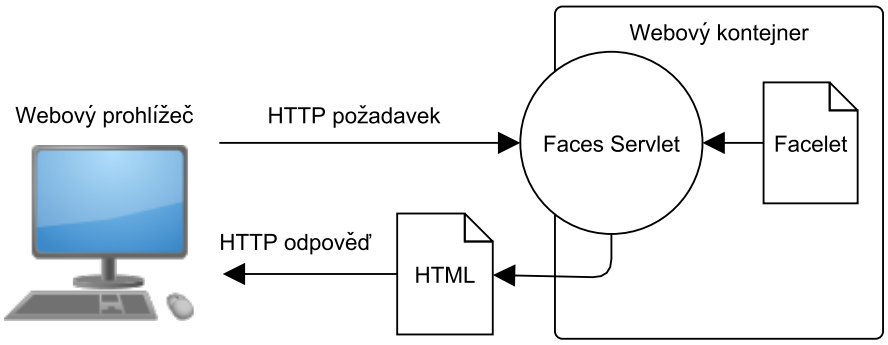
- JSF Facelets Tag Library (prefix ui) – základní značky pro tvorbu šablon (takzvaných templates), používá se k definici částí obsahu,
- JSF HTML Tag Library (prefix h) – značky k renderování běžného HTML jako jsou

formulářové prvky, tabulky, odkazy a podobně,

- JSF Core Tag Library (prefix f) – značky pro přístup k nastavení JSF jádra (například validace, konverze či Ajax),
- JSTL Core Tag Library (prefix c) – značky jádra JSTL (JavaServer Pages Standard Tag Library), zejména podmínky, cykly či obsluha výjimek,
- JSTL Functions Tag Library (prefix fn) – obsahuje funkce JSTL pro práci s řetězci.

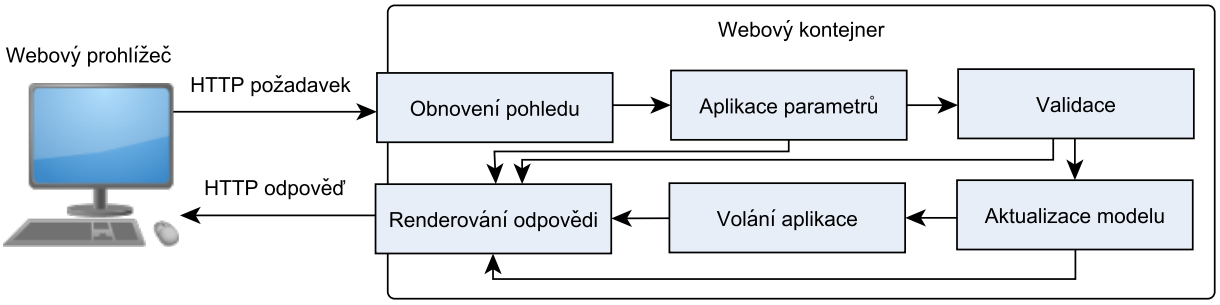
5.2 Životní cyklus požadavku

Základním principem JSF je mapování veškerých klientských požadavků na takzvaný Faces Servlet (obr. 5.1), jež je součástí implementace JSF. Ten pak dle konfiguračního souboru `faces-config.xml` nebo anotací uvedených ve zdrojovém kódu vybere podle URL adresy požadavku vhodný JavaBean, modifikuje jeho stav na základě parametrů HTTP a vyvolá patřičnou metodu tohoto beanu. Výsledek volání metody a nový stav beanu je použit k vyrendrování odpovědi prostřednictvím JSF šablony a data ve formě HTML a JavaScriptu odeslána zpět klientovi.



Obrázek 5.1: Java Server Faces

Před nastavením nového stavu JavaBeanu je prováděna validace předaných parametrů a pokud jsou nalezeny jakékoliv nesrovnalosti, veškeré chybové zprávy jsou posbírány a odeslány klientovi zpět v pohledu, na kterém se právě nachází. Životní cyklus požadavku lze rozdělit do následujících stavů (obr. 5.2):



Obrázek 5.2: Životní cyklus JSF

- Obnovení pohledu – JSF implementace obnoví patřičný pohled ze session, případně založí pohled nový, pokud se jedná o úvodní požadavek,
- Aplikace parametrů – proběhne načtení parametrů požadavku a jejich aplikace na jednotlivé komponenty ve stromu komponent, pokud dojde kdekoliv k výjimce, přechází se do fáze renderování odpovědi,
- Validace – proces validace vstupu, jednotlivé atributy komponent jsou validovány vůči nastaveným pravidlům, rovněž proběhnou případné konverze, v případě výjimky je renderována odpověď s chybovými hláškami,
- Aktualizace modelu – procházením stromu komponent jsou všechny mapované atributy přeneseny do příslušných Java Beanů prostřednictvím jejich setterů, nastane-li výjimka, přechází se opět rovnou k sestavení odpovědi,
- Volání aplikace – probíhá vyvolání akčních metod na Java Beanech, obsluha aplikačních událostí a na základě jejich výsledku se rozhodne o případném přechodu na jiný pohled,
- Rendrování odpovědi – finální sestavení HTTP odpovědi a její odeslání na klienta.

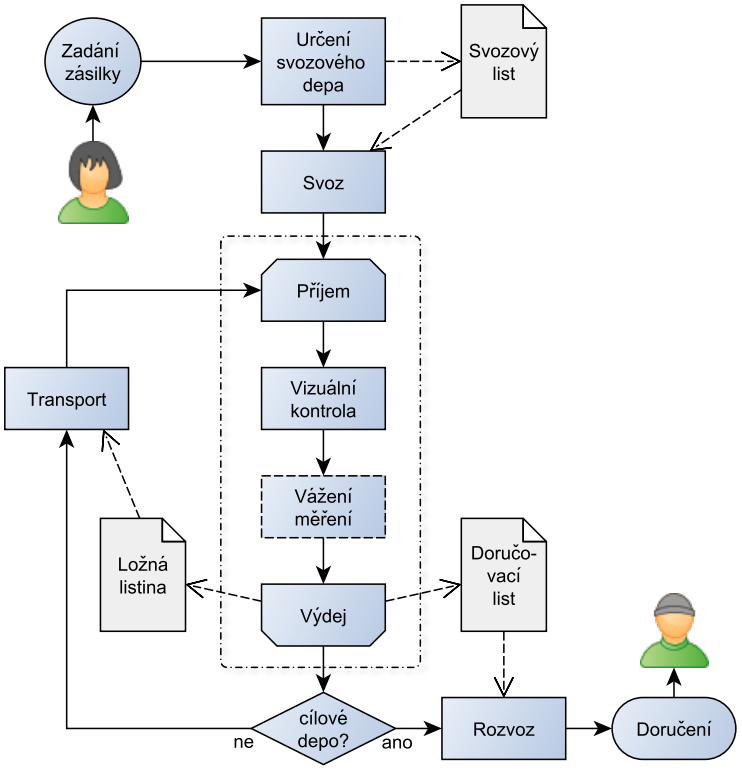
Kapitola 6

Analýza aplikace

6.1 Požadavky

Pro objektivní posouzení vlastností objektově relačního mapování EclipseLink slouží vytvořená referenční aplikace. Jedná se o jednoduchý informační systém pro společnost zabývající se přepravou kusových zásilek.

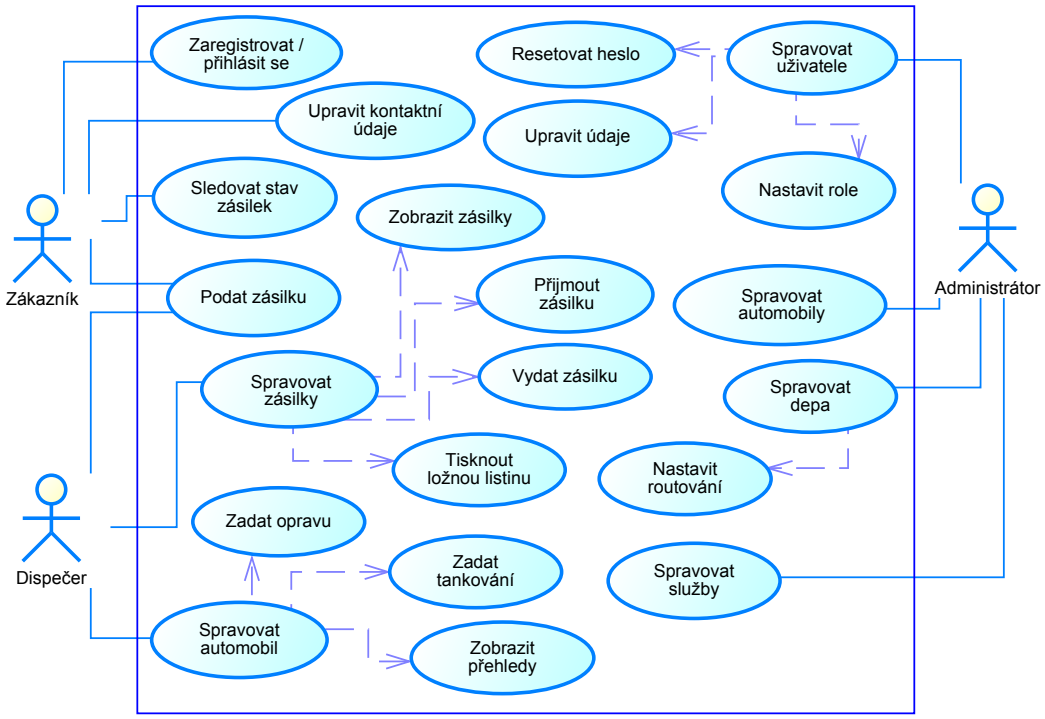
Aplikace zajišťuje celý životní cyklus kusové zásilky od jejího vložení do systému zákazníkem či pracovníkem společnosti, naložení zásilky řidičem, odvozem na svozové depo, průchod překladištěm a následné doručení příjemci skrze depo distribuční. Aplikace rovněž nabízí systém sledování zásilek Track and Trace i evidenci automobilů včetně informací o opravách či čerpání pohonných hmot. Model obchodního procesu, který je aplikací podpořen je na obr. 6.1.



Obrázek 6.1: Obchodní proces

K objasnění rozsahu implementace byl sestaven diagram užití (obr. 6.2), který popisuje možnosti systému vzhledem k různým uživatelským rolím. Stanoveny byly tyto tři uživatelské role:

- Administrátor – spravuje nastavení a kmenová data systému jako jsou práva uživatelů, routovací tabulky pro směrování zásilek, seznam řidičů, automobilů a dep, poskytuje podporu uživatelům,
- Dispečer – hlavní uživatel systému, koordinuje činnost na depu, provádí příjem a výdej zásilek, tiskne seznamy zásilek, zobrazuje přehledy, zajišťuje rovnoměrné vytížení automobilů a zadává k nim informace o tankování, údržbě a opravách,
- Zákazník – zadává zásilky ke svozu a kontroluje stav odeslaných zásilek, rovněž má možnost registrace a po přihlášení do systému vyšší komfort při práci s aplikací.



Obrázek 6.2: Diagram užití aplikace

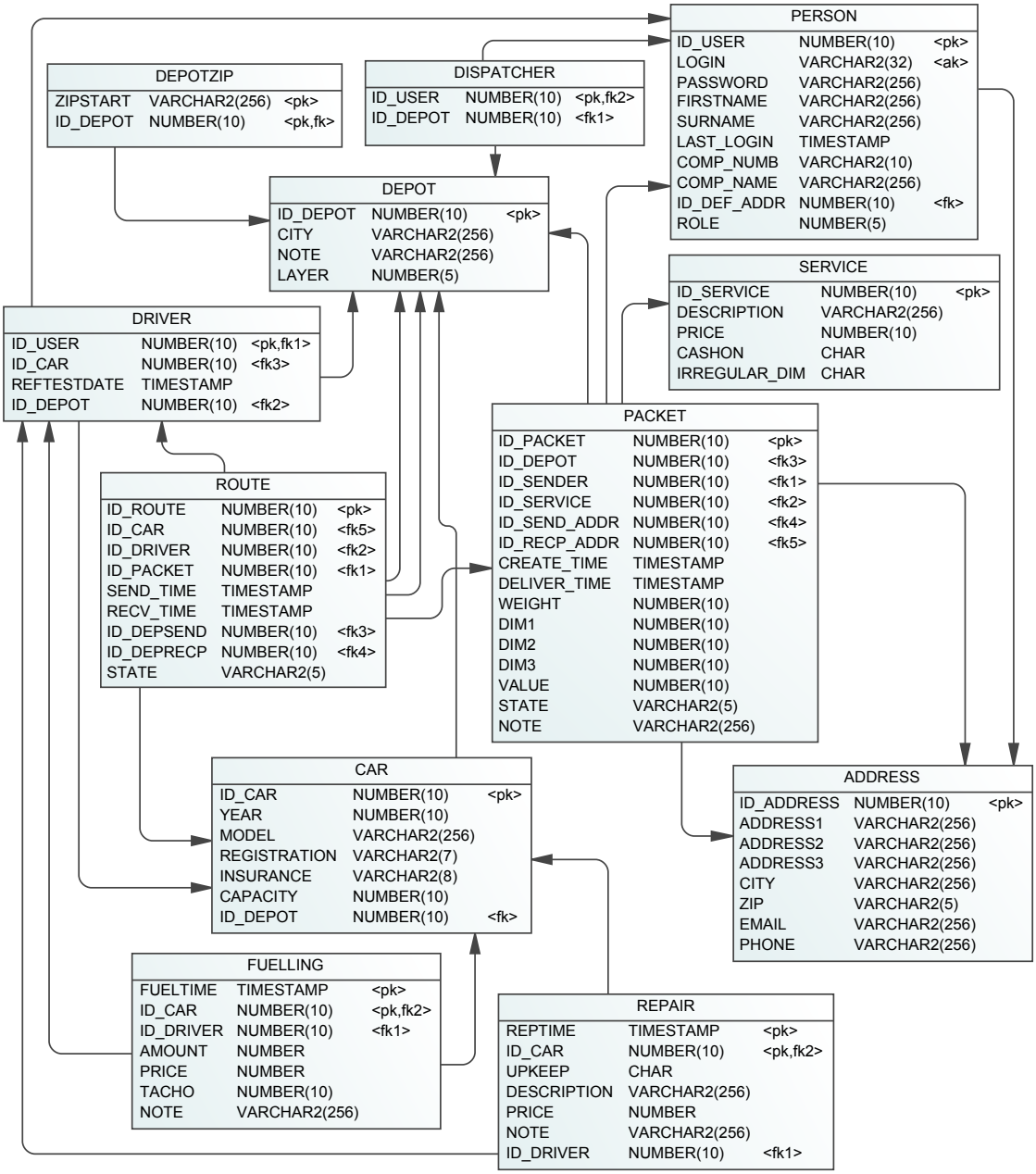
Z diagramu na obr. 6.2 jsou patrné rovněž případy užití aplikace vzhledem k jednotlivým uživatelským rolím. Případy užití aplikace pro nepřihlášeného uživatele se v zásadě neliší od případů užití přihlášeného uživatele s rolí *Zákazník*. Po přihlášení je uživateli umožněna změna kontaktních údajů zadaných při registraci, které jsou automaticky vyplněny při odeslání zásilky, rovněž má uživatel přehled o všech odeslaných zásilkách, včetně podrobného přehledu o stavu zásilky, který není nepřihlášeným uživatelům skrze sledování zásilky přístupný.

Na realizovanou aplikaci jsou kladeny především mimofunkční požadavky, jelikož jejím hlavním cílem je demonstrace možností objektově relačního mapování EclipseLink. Účelem implementace je zejména nabytí praktických zkušeností v následujících oblastech:

- tvorba aplikace využívající Java Persistence API,
- možnosti objektově relačního mapování EclipseLink,
- srovnání EclipseLink s konkurenční knihovnou Hibernate,
- problémy při migraci datové vrstvy mezi různými ORM.

6.2 Datový model

Aplikace je realizována ve dvou verzích. Datová vrstva první verze je postavena na objektově relačním mapování EclipseLink, která je pro druhou verzi migrována na knihovnu Hibernate. Implementační detaily jsou shrnuty v následující části práce. Fyzickému modelu datové vrstvy odpovídá obr. 6.3.



Obrázek 6.3: Fyzický datový model (exportováno z programu PowerDesigner)

Persistence veškerých registrovaných uživatelů systému probíhá do tabulky *Person*. Evidovány jsou údaje jako je jméno a příjmení osoby, přihlašovací jméno, heslo, datum a čas posledního přihlášení, údaje o společnosti a adresa. Každý uživatel má rovněž nastavenu roli, kterou je možné změnit pouze v administrační části aplikace. Každý uživatel systému může mít v jednu chvíli přiřazenu pouze jednu roli. Systém umožní nastavení následujících uživatelských rolí:

- Běžný uživatel – výchozí role, nastavena při registraci uživatele, jedná se o běžného uživatele systému,
- Dispečer – pracovník depa,
- Řidič – pouze fiktivní role, řidiči nebude umožněno přihlášení do aplikace,
- Administrátor systému.

Tabulky *Driver* a *Dispatcher* slouží k uložení dalších informací u zaměstnanců společnosti. Evidováno je depo, kde je pracovník přiřazen a u řidiče dále svěřený automobil a datum absolvovaného školení řidičů. Vazební vztah mezi těmito tabulkami a tabulkou *Person* je 1:1 vzhledem k primárnímu klíči všech tabulek, což je identifikační číslo uživatele. Zařazení zaměstnance na více dep najednou je tedy vyloučeno.

Depa společnosti jsou zanesena v tabulce *Depot*. Záznam je omezen na město, kde se depo nachází a libovolnou poznámku. Pole *Layer* slouží k zařazení depa do hierarchie pro směrování zásilek, tedy například *Layer=0* značí depo místní, *Layer=1* by bylo depo krajské a *Layer=2* depo centrální. Spolu se seznamem obslužných poštovních směrovacích čísel z tabulky *Depotzip* tak lze zajistit směrování zásilky přes libovolný počet dep.

Aktuální údaje o zásilce jsou drženy v tabulce *Packet*. Jedná se o identifikátory odesílatele, adresy odesílatele, adresy příjemce, zvolené služby z tabulky *Service*, dále pak datum a čas podání, datum a čas doručení, hmotnost, rozměry, hodnotu, libovolnou poznámku a poslední stav zásilky včetně identifikátoru posledního depa, kterým zásilka prošla. Tabulka *Service* je pouze číselník služeb, které společnost nabízí. Jejím obsahem je textový popis služby, cena služby, informace zda se jedná o službu s dobírkou a zda je služba určena pro zásilky s nepravidelnými rozměry.

Pro každý převoz zásilky z jednoho depa na jiné nebo mezi zákazníkem a depem je vytvořen záznam v tabulce *Route*. Záznam obsahuje identifikátor automobilu, řidiče, zásilky, časovou značku předání zásilky řidiči, časovou značku přijetí zásilky na depo nebo převzetí zákazníkem, identifikátory depa pro příjem či výdej a původní stav zásilky. Aplikace vždy nastaví zásilce jeden z následujících stavů:

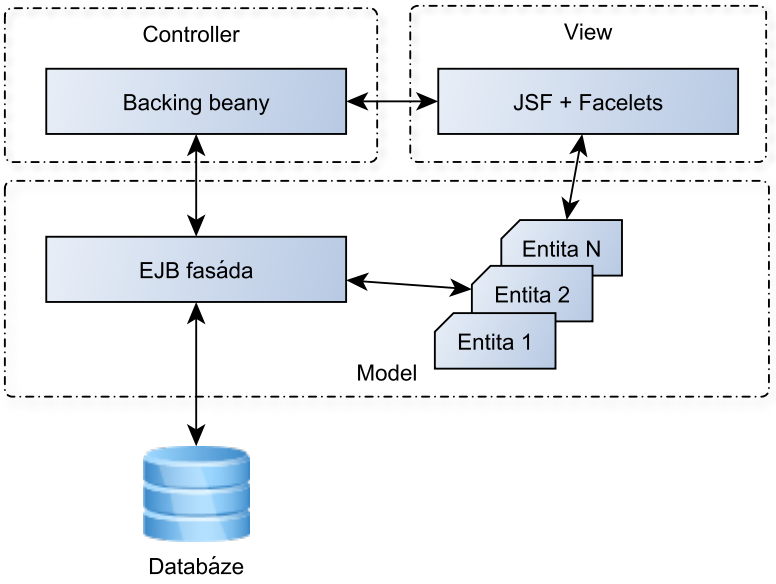
1. *Nová* – nová zásilka v systému, čeká na potvrzení a zadání příkazu k vyzvednutí,
2. *Vyzvedávána* – pro zásilku bylo naplánováno vyzvednutí a určen řidič, který zásilku od odesílatele převezme,
3. *Vyzvednutá* – pro zásilku byl proveden příjem na depu, měření, vážení a dále přechází buď automaticky rovnou do stavu *K doručení* nebo pokračuje transportem na překladiště se stavem *Ve svozu*,
4. *Ve svozu* – zásilka je přepravována na centrální překladiště,
5. *V překladišti* – zásilka se nachází v centrálním překladišti,
6. *V rozvozu* – zásilka je přepravována z překladiště na cílové depo,
7. *K doručení* – zásilka se nachází na cílovém depu a čeká na výdej řidiči,
8. *Doručována* – zásilka je doručována řidičem cílového depa,
9. *Doručena* – zásilka byla doručena příjemci.

Veškeré adresy zadané do aplikace, tedy jak adresy uživatelů, tak adresy odesílatele a příjemce zásilky, jsou ukládány do tabulky *Address*. V této tabulce se rovněž udržují případné kontaktní údaje jako je e-mail nebo telefonní číslo.

Aplikace umožňuje evidenci automobilů v tabulce *Car*. Zaznamenat lze model automobilu, rok uvedení do provozu, registrační značku, číslo pojistné smlouvy, kapacitu nákladového prostoru a identifikátor depa, ke kterému je automobil evidován. Tabulky *Fuelling* a *Repair* slouží k uchování událostí o tankování, opravách a údržbě vozidel.

6.3 Architektura aplikace

Aplikace je navržena s ohledem na třívrstvou architekturu. Aplikační vrstva je oddělena od datové vrstvy pro provedení snadné migrace mezi oběma implementacemi Java Persistence API, EclipseLink a Hibernate. Na obr. 6.4 je navržena architektura aplikace na bázi návrhového vzoru MVC [21].



Obrázek 6.4: Architektura aplikace

Základem datové vrstvy jsou entitní třídy, které odpovídají tabulkám v relační databázi ve výše uvedeném datovém modelu. K nim je implementována EJB fasáda zastřešující veškeré potřebné operace prováděné s entitami jako je jejich načítání z databáze, ukládání, editace či odstranění. Aplikační logiku zajišťují backing beany, na které bude navázána prezentační vrstva ve formě JSF šablon.

Kapitola 7

Implementace

7.1 Entitní třídy

Implementace aplikace probíhala od datové vrstvy směrem k aplikační a následně prezentační. Po návrhu fyzického modelu v programu Case PowerDesigner byl model exportován do databáze Oracle, odkud byly dále vývojovým prostředím Netbeans vygenerovány entitní třídy. Obdobnou funkcionalitu nabízí i konkurenční prostředí Eclipse, je však potřeba mít instalovánu verzi pro Java Enterprise Edition nebo patřičný plugin pro práci s JPA. Generátor entit v produktu Netbeans se ovšem jevil v určitých ohledech „schopnější“ a výsledný kód byl byl mnohem blíže zamýšlenému cíli. Současně s generováním entitních tříd byla rovněž vytvořena persistentní jednotka a jako poskytovatel persistence vybrána knihovna EclipseLink pro JPA 2.0.

Ačkoliv byly vygenerované entity použitelné, s ohledem na efektivitu vývoje i běh výsledné aplikace je vhodné některé způsoby mapování v entitních třídách dále upravit. V prvé řadě se jedná o způsob generování identity vzhledem k zamýšlené databázi. Pro databázi Oracle je například velmi žádoucí nastavit generátor identity na bázi sekvence.

Dalším problémem v databázi Oracle je, vzhledem k absenci booleovského datového typu, způsob ukládání hodnot pro dvoustavovou logiku. Pro jednoduchou implementaci prezentační vrstvy jsou dvoustavové hodnoty v aplikaci mapovány prostřednictvím vlastního výčtového datového typu, který je dále rozšířen o textovou hodnotu čitelnou pro člověka, přičemž *getter* a *setter* příslušného výčtového atributu lze dále rozšířit o přístup prostřednictvím datového typu boolean. Použitý výčtový datový typ je mapován do databáze skrze svou ordinální hodnotu.

Tento způsob umožňuje provést vcelku jednoduchým způsobem i případnou lokalizaci hodnot *ano* či *ne* a zároveň zachovat standardní způsob přístupu k danému atributu.

Vhodnou úpravou generovaných entit je také odstranění nadbytečných vazeb. Vazební atributy realizované na obou stranách nejsou vždy zapotřebí a z hlediska výkonu a nutnosti vazbu udržovat u obou entit, je účelné je odstranit.

Poslední úpravy se týkají datových typů atributů. Rozsahy datových typů jazyka Java a mapované relační databáze nejsou vždy rovnocenné a je tedy nezbytné volit určitý kompromis, a to buď na jedné nebo na druhé straně. Například mapování databázového datového typu `Number` nemusí být vždy v programu implementováno datovým typem `BigDecimal` a v mnohých případech postačí použít `long` nebo `int`.

Java Persistence API umožňuje i opačný způsob definice datové vrstvy. Programátor zahájí implementaci entitních tříd, nastaví parametry mapování a při prvním spuštění aplikace jsou v nakonfigurované databázi automaticky vytvořeny veškeré tabulky, sekvence i indexy. Dle zvolené strategie jsou při dalším spuštěním do databáze zaneseny buď pouze změny schématu, nebo je schema databáze kompletně odstraněno a vytvořeno znovu.

7.2 EJB fasáda

K implementovaným entitním třídám je vygenerována vývojovým prostředím také kostra EJB fasády. Jejím základem je generická třída `AbstractFacade` zajišťující nad databází základní CRUD operace (create, read, update a delete). Ke každé entitní třídě je automaticky vytvořen rovněž EJB bean, který rozšiřuje třídu `AbstractFacade` a lze do něj postupně přidávat další potřebné metody. Není nezbytné ponechat v aplikaci generované EJB beany ke všem entitním třídám. Pro zpřehlednění byly entity logicky seskupeny do souvisejících celků a zachovány pouze některé.

Do každého EJB beanu je aplikačním kontejnerem injektovaná instance `EntityManageru` v podobě soukromého atributu a veškeré operace související s datovou vrstvou jsou prováděny v jeho veřejných metodách.

Dotazy JPA jsou definovány přímo nad příslušející entitou ve formě anotace `@NamedQuery`, což zvyšuje bezpečnost aplikace, odolnost proti útokům typu SQL injection a v neposlední řadě zůstává kód velmi dobře čitelný. Tvorba dotazů jde velmi rychle, neboť jsou psány právě do místa, kam logicky přísluší a programátor má tak při psaní dotazu na očích potřebné atributy entity. I když je JPA dotaz zapisován jako řetězec, tedy do uvozovek, vývojové prostředí provádí automatické doplňování syntaxe.

7.3 Uživatelské rozhraní

Uživatelské rozhraní aplikace je postaveno na specifikaci Java Server Faces 2.0 a referenční implementaci Mojarra [22]. Jelikož specifikace JSF definuje uživatelské rozhraní pouze principiálně a Mojarra nenabízí, kromě samotné implementace, téměř nic navíc, nabízí se použití některé z knihoven obsahující další komponenty.

Implementací JSF či komponentových frameworků existuje velké množství, zde je přehled pěti nejoblíbenějších[1]:

- Jboss Richfaces [23] – open-source knihovna JSF komponent s podporou Ajaxu, podporuje rovněž validaci vstupu na straně klienta, aktualizaci klienta metodou push na bázi Java Messaging Service či sadu pro vývoj vlastních komponent,
- Jboss Seam 3 [24] – platforma pro vývoj webových aplikací v Javě s LGPL licencí, nabízí integraci technologií JPA, JSF, Ajax a EJB 3.0 do kompletního řešení, podpora společností RadHat byla bohužel ukončena a vývoj některých součástí byl přesunut pod skupinu Apache, jiné byly ukončeny,
- Primefaces [25] – knihovna velkého množství ajaxových JSF komponent s licencí Apache, aktivní vývoj, značně rostoucí oblíbenost (dle Google Trends), doporučeno vývojovým týmem Spring,
- MyFaces [26] – alternativní implementace JSF od Apache, k dispozici je rovněž několik knihoven komponent,
- ICEfaces [27] – knihovna JSF komponent založená na kódu Primefaces, obsahuje menší množství komponent.

S ohledem na výše uvedené vlastnosti jednotlivých produktů byla pro implementaci uživatelského rozhraní vybrána knihovna Primefaces, která prochází opravdu aktivním vývojem a v průběhu tvorby aplikace bylo vydáno několik nových verzí. Veškeré vážné problémy, které se objevily tak bylo možné vyřešit pouze aktualizací knihovny. Primefaces nabízí největší počet JSF komponent z uvedených rámců a postupně jsou přidávány další.

Pro celou aplikaci je vytvořena šablona `template.xhtml` rozdělená na hlavičku, patičku a obsah stránky. Ta je základem všech dalších webových stránek aplikace. Pro každou uživatelskou roli je definována stránka vlastní. Jednotlivé případy užití příslušné role jsou dále

[1] Řazeno dle počtu výsledků vyhledávače Google ke dni 7.5.2013 na dané klíčové slovo.

členěny na karty (Primefaces komponenta `tabView`), přičemž obsah každé karty je oddělen do zvláštního souboru `*.xhtml`, který je na danou kartu vložen značkou `ui:include`.

Aplikace hojně využívá AJAX a karty uživatelského rozhraní jsou načítány opožděně.

7.4 Backing beany

Webové stránky uživatelského rozhraní jsou vázány na takzvané „backing beany“. Při vývoji aplikace se osvědčilo členění backing beanů dle uživatelských rolí, nicméně postupem času bylo zapotřebí provést refaktoring a některé dále rozdělit dle případů užití. Každá stránka je tedy vázána na několik beanů uložených v session a ty jsou dále společně propojeny skrze vkládání závislostí (anglicky dependency injection).

Samotné vkládání závislostí je součástí jak JSF, tak i specifikací EJB a Contexts and Dependency Injection (CDI). Situace v Java EE 6 se tedy komplikuje a programátor má na výběr z alternativních anotací z následujících balíčků:

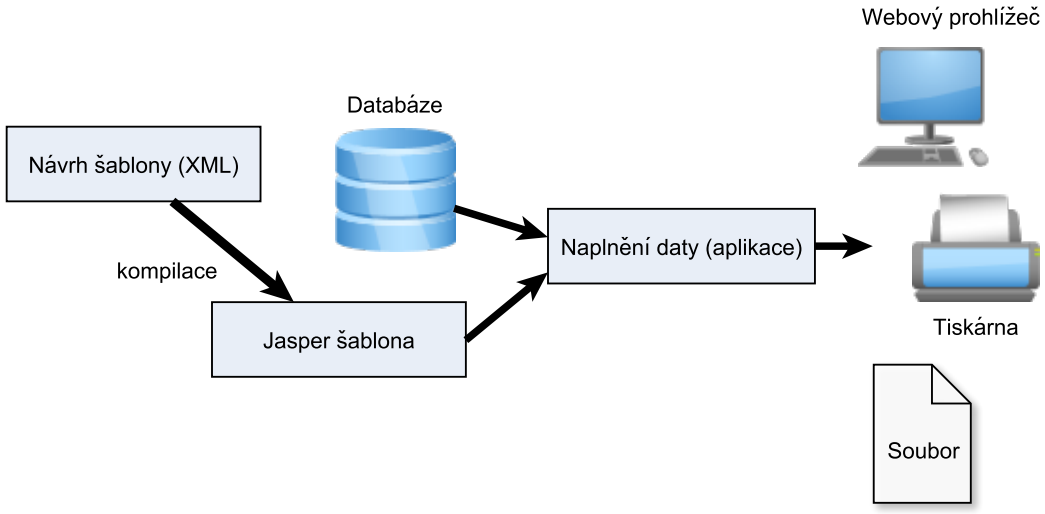
- JSF: javax.faces.bean.*
- CDI: javax.inject.*
- CDI: javax.enterprise.context.*
- EJB: javax.ejb.*

V prostředí plnohodnotného aplikačního serveru je na místě opustit vkládání závislostí anotacemi z JSF specifikace a pro backing beany použít pouze anotace CDI `@Named` a `@RequestScoped`, případně `@SessionScoped` pro zařazení backing beanu do session. Na výslednou funkčnost JSF to nemá žádný vliv a vzhledem k propojení CDI i EJB je možné v backing beanech používat i EJB anotace jako je například `@PostConstruct` nad metodou a jednoduše tak zajistit inicializaci beanu hodnotami z databáze, neboť v době volání konstruktoru beanu při injektáži EJB fasády ještě není příslušný atribut inicializován.

7.5 Reporting

Tiskové výstupy z aplikace jako jsou ložné listiny, doručovací a svozové listy jsou zajištěny open-source knihovnou JasperReports [28]. Šablony pro reporting na bázi XML lze přehledně navrhovat v programu iReport postaveném na platformě Netbeans IDE, ale je k dispozici i jeho obdoba ve formě pluginu do vývojového prostředí Eclipse.

Šablony navržených reportů jsou následně zkompilovány do binární podoby a uloženy v souborech s příponou `*.jasper`. Zkompilovanou šablonu je dále možné použít ve vyvíjené aplikaci, naplnit daty a následně prezentovat v rozličných formátech. Podporovány jsou formáty HTML, XML, PDF, Excel, Word či OpenOffice. Výstup reportu lze rovněž směrovat na tiskárnu prostřednictvím `JasperPrintManageru` či pouze zobrazit náhled uživateli třídou `JasperViewer`, viz obr. 7.1.



Obrázek 7.1: Použití JasperReports

K naplnění šablony daty slouží datový zdroj. JasperReports podporuje datové zdroje ve formě CSV souboru, XML i databáze skrze JDBC či Hibernate. V implementované aplikaci je použit `JRBeanCollectionDataSource` založený na kolekci JavaBeanů. Tím je zajištěna nezávislost na použitém datovém zdroji a reporting se při migraci datové vrstvy aplikace obejde bez jakýchkoliv úprav.

7.6 Konfigurace a použité nástroje

Veškeré ladění aplikace, včetně jejího testování probíhalo v operačním systému Microsoft Windows 7 Home Premium x86 na Java Development Kit 1.7 (32 bit) a aplikačním serveru GlassFish 3.1.2 (pro Java EE 6), a to především z důvodu jeho silné integrace s vývojovým prostředím Netbeans 7.3, ve kterém byla aplikace vyvíjena. Jelikož je server GlassFish referenční implementací aplikačního kontejneru pro Java aplikace, odpadají při vývoji jakékoliv problémy s případnou nekompatibilitou nebo nestandardním chováním oproti specifikaci

Java EE. Pro nasazení aplikace do reálného prostředí lze samozřejmě zvolit jinou platformu. Změnám v konfiguraci aplikačního kontejneru i nasazované aplikace a důkladnému testování se tak jako tak vyhnout nelze, tudíž by použití jiného serveru dle aktuálních potřeb nemělo činit potíže.

Jako datová platforma byla zvolena databáze Oracle Database 11g Express Edition, která, dle průzkumu trhu společností Gartner [29], zaujímá vedoucí pozici na trhu s relačními databázemi. Objektově relační mapování je zajištěno knihovnou EclipseLink 2.3.0 a dále pak Hibernate 4.1.9 Final.

K implementaci prezentační vrstvy aplikace využívá množinu komponent PrimeFaces 3.4.1 rozšiřující JSF implementaci Mojarra 2.1 a pro generování reportů a tiskových sestav je použita knihovna JasperReports verze 5.0.

7.7 Možná rozšíření

Aplikace byla implementována s ohledem na vyzkoušení možností objektově relačního mapování, pro její případné nasazení by samozřejmě bylo nutné zajistit podrobnou analýzu požadavků v reálném prostředí a provést příslušné úpravy datové i aplikační vrstvy. Případná rozšíření aplikace by se mohla týkat především oblasti bussiness reportingu, jelikož v prostředí praxe je nejvyšší důraz kladen na hospodářské výsledky a celkovou finanční zátěž subjektu. To je konec konců ve většině případů i hlavním důvodem nasazení informačních systémů do prostředí podniku.

Oblasti jako je monitoring výkonnosti zaměstnanců na depu či řidičů, přehledy počtu odeslaných zásilek za určitý časový úsek, sledování doby doručení zásilky, to vše jsou záležitosti, které by měl reálně nasazený systém pokrývat. Prostor pro další implementaci lze spatřit i v oblasti systémové integrace a napojení na další informační systémy firmy, např. rozhraní pro fakturaci.

Reálná aplikace by se samozřejmě neobešla bez speciálního rozhraní pro řidiče vybavené mobilními terminály pro skenování čárových kódu zásilek. Výdej a příjem na depu by po nalepení etiket probíhal již velice rychle.

Kapitola 8

Srovnání rámců

8.1 Rozdíly v implementaci

Migrace aplikace na platformu Hibernate proběhla téměř bez potíží. Celý proces začíná úpravou konfiguračního souboru `persistence.xml`. Třídu poskytovatele persistence je potřeba změnit z původní třídy EclipseLink:

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

na třídu poskytovatele pro Hibernate:

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

Rovněž je nutné změnit sekci `<properties>`, kde se provádí nastavení specifické právě pro použitý poskytovatel persistence. Vyžadováno je nastavení dialektu pro použitou databázi:

```
<property name="hibernate.dialect"
    value="org.hibernate.dialect.Oracle10gDialect"/>
```

Hibernate lze konfigurovat také konfiguračním souborem `hibernate.cfg.xml`, který je obdobou souboru `persistence.xml` z JPA. Pokud by byly v projektu oba soubory, vlastnosti nastavené v souboru `persistence.xml` mají vyšší prioritu. Po nastavení classpath ke všem potřebným knihovnám by již aplikace měla jít nasadit do aplikačního serveru a následně spustit. Po provedení základních testů se však projevily některé problémy, které si vyžádaly zásah do kódu aplikace.

Celkem závažný problém se vyskytl při opožděném načítání kolekcí. Hibernate na rozdíl od EclipseLinku neumožňuje opožděné načítání, pokud je uzavřen `EntityManager`, kterým byla původní data načtena. Vzhledem k tomu, že aplikace používá rozšířený persistentní kontext a správa instancí třídy `EntityManager` je ponechána na aplikačním serveru, nelze ovlivnit, kdy je jeho instance uzavřena. Aplikační server bohužel uzavírá tuto instanci okamžitě po vyřízení požadavku a pro další požadavek již vytváří instanci novou. Při pokusu o opožděné načtení atributu v dalším požadavku je vyhozena `LazyInitialization-Exception`. Změnit způsob načítání z opožděného na okamžitý bohužel není možné. Situace u EclipseLink je výrazně lepší a při dalším požadavku EclipseLink provede opožděné načtení i skrze novou instanci `EntityManager`u.

Řešení tohoto problémů existuje několik. Úplné vypnutí opožděného načítání není možné, vzhledem k problému „N+1“, kdy je databáze zbytečně zatížena velkým množstvím dotazů. Entity by zde byly načítány kaskádně a množství dotazů na databázi rostlo exponenciálně s hloubkou zanoření. Efektivním řešením může být držení instance `EntityManager` v session, zde však nastává problém s uvolněním spojení při trvalém odpojení klienta a je potřeba, vzhledem k blokovaným prostředkům, nastavit session poměrně krátkou trvanlivost, což může být v rozporu s dalšími požadavky na aplikaci. V implementované aplikaci je načítání kolekcí ponecháno jako opožděné a řešení je omezeno pouze na jejich ruční načtení v době před jejich použitím.

Zajímavá odlišnost v implementaci obou knihoven se projevila při persistenci výčtového datového typu. Atribut entity byl anotován `@Enumerated(ORDINAL)`, přičemž mapování bylo nastaveno pro sloupec databáze s datovým typem `char`. Cílem bylo zajistit mapování jednoduchého výčtu pro dvoustavovou logiku s hodnotami „no“ a „yes“, který by v databázové tabulce představoval hodnoty „0“ a „1“. V případě EclipseLink toto mapování fungovalo bez problémů, pro Hibernate však bylo nezbytné změnit datový typ sloupce v databázi na `number`. Podobných rozdílů mezi oběma rámci bude pravděpodobně mnohem více.

8.2 Rozdíly ve výkonu

V implementované aplikaci se neprojevily mezi oběma poskytovateli ORM žádné výkonové rozdíly. Z tohoto hlediska jsou tedy knihovny EclipseLink i Hibernate srovnatelné. Je však třeba dodat, že výkon je v tomto případě hodnocen čistě subjektivně, neboť pro objektivní měření výkonu by bylo potřeba do hodnocení zahrnout rovněž veškeré vnější prostředí a

jeho celkovou variabilitu, včetně srovnání výkonu v různých aplikačních kontejnerech. Zde samozřejmě hraje určitou roli i použitá databáze, portfolio SQL dotazů, množství uživatelů vykonávajících dotazy současně či samotná verze použité knihovny pro ORM.

Různé zdroje uvádí výkonové rozdíly mezi oběma knihovnami v extrémních případech tisíců dotazů maximálně v řádu stovek milisekund ve prospěch jednoho či druhého. V reálném prostředí tedy nemá smysl tento faktor brát při rozhodování v úvahu a zaměřit se spíše na použitelnost, kompatibilitu, či případnou cenu toho nebo onoho řešení.

8.3 Doporučení

Při použití rozhraní Java Persistence API se obě řešení ukázala jako rovnocenná a záleží tedy víceméně na okolnostech, kterou ze dvou testovaných implementací zvolit. Pro nově vznikající projekty lze doporučit použití JPA v kombinaci s EclipseLink, jakožto standardizovaného řešení s poměrně silnou jistotou do budoucna a podporou silných korporací jako je Eclipse Foundation či Oracle. Naopak pro stávající projekty se téměř nevyplatí migrovat z Hibernate na cokoliv jiného, vzhledem k obdobné funkcionalitě, a to i nad rámec JPA, rozsáhlé komunitě i dokumentaci a současné znalosti programátorů.

Kapitola 9

Závěr

Objektově relační mapování jako takové se jeví bez větších problémů a ve srovnání s programováním aplikace nad standardním JDBC rozhraním je vývoj mnohem rychlejší. Tato pružnost je sice vykoupena určitou časovou investicí do studia, nicméně při implementaci větších projektů se rozhodně vyplatí. ORM lze svým způsobem chápat i jako určitou pojistku do budoucna, neboť ani migrace aplikace na jinou databázi nemusí být problém a jedná se zde v podstatě o vyřešení základní konfigurace a případné malé úpravy v datové vrstvě.

Velkou výhodou objektově relačního mapování je oproštění programátora od nízkoúrovňových problémů. Není potřeba řešit problémy typu „jakým způsobem data získat“, ale lze se plně soustředit na aplikační logiku. Výsledkem je tedy čistější a přehlednější kód i rychlejší vývoj aplikace jako takové.

Cílem práce bylo představit ORM EclipseLink a objasnit odlišnosti od knihovny Hibernate, k čemuž do značné míry přispěla dvojí realizace ukázkové aplikace. Nastíněna byla rovněž řešení problémů, které se mohou při migraci aplikace vyskytnout. Na závěr bylo provedeno kritické srovnání obou produktů a doporučen postup při jejich výběru.

Je evidentní, že obě řešení (tedy jak EclipseLink, tak Hibernate) jsou vyspělé produkty, které prochází neustálým vývojem a v oblasti informačních technologií je mnohdy velmi obtížné učinit rozhodnutí či odhadnout jeho případné následky. Zde je ovšem situace velmi usnadněna, neboť existují standardy a specifikace jako je například JPA, které mohou rozhodování značně ulehčit. Konec konců, jak se při implementaci ukázalo, ani jedno z nabízených řešení nemusí být vyloženě špatné a v případě jakýchkoliv problémů je možné jednoduše provést migraci datové vrstvy z jednoho poskytovatele persistence na druhý.

Literatura

[1] HERNANDEZ, Michael J. *Database Design for Mere Mortals: A Hands-on Guide to Relational Database Design.* Third Edition. Ann Arbor: Addison-Wesley, 2013. ISBN 03-218-8449-3.

[2] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM.* vol. 13, issue 6, s. 377-387. DOI: 10.1145/362384.362685. Dostupné z ⟨http://portal.acm.org/citation.cfm?doid=362384.362685⟩.

[3] NOVÁK, Vítězslav. Využití návrhového vzoru adaptér pro odstínění rozdílů v API objektových databází *Acta academica karviniensia*. 2/2013, s. 47-53. Dostupné z ⟨http://www.slu.cz/opf/cz/informace/acta-academica-karviniensia/casopisy-aak/aak-rocnik-2013/docs-2-2013/Novak.pdf⟩.

[4] PROCHÁZKA, Jaroslav. *Objektově orientované databáze* [online]. ©2004, 3.3.2004 [cit. 2013-03-17]. Dostupné z ⟨http://www.dbsvet.cz/view.php?cisloclanku=2004030301⟩.

[5] APACHE SOFTWARE FOUNDATION. *OpenJPA project* [online]. 2013 [cit. 2013-03-16]. Dostupné z ⟨http://openjpa.apache.org/⟩.

[6] AVAJÉ. *Ebean ORM* [online]. 2013 [cit. 2013-03-17]. Dostupné z ⟨http://www.avaje.org/⟩.

[7] ORACLE CORPORATION. *Oracle Press Release: Oracle Releases Oracle® TopLink® 11g Based on EclipseLink* [online]. ©2008, September 22, 2008 [cit. 2013-03-17]. Dostupné z ⟨http://www.oracle.com/us/corporate/press/017498_EN⟩.

[8] ORACLE CORPORATION. *Enterprise JavaBeans Technology* [online]. 2013 [cit. 2013-03-18]. Dostupné z ⟨http://www.oracle.com/technetwork/java/javaee/ejb/index.html⟩.

[9] FOWLER, Martin. *POJO* [online]. ©2013, [cit. 2013-03-18]. Dostupné z ⟨http://www.martinfowler.com/bliki/POJO.html⟩.

[10] ORACLE CORPORATION. *Java TM Platform, Enterprise Edition 6 API Specification* [online]. ©2011, 2011-02-10 [cit. 2013-03-23]. Dostupné z ⟨http://docs.oracle.com/javaee/6/api/⟩.

[11] JENDROCK, Erick, Ricardo CERVERA-NAVARRO, Ian EVANS, Devika GOLLAPUDI, Kim HAASE, William MARKITO a Chinmayee SRIVATHSA. *The Java EE 6 Tutorial* [online]. ©2013, 2013-01 [cit. 2013-03-23]. Dostupné z ⟨http://docs.oracle.com/javaee/6/tutorial/doc/⟩.

[12] DEMICHIEL, Linda. *JSR 317: JavaTM Persistence 2.0* [online]. ©2009, 10 Dec, 2009 [cit. 2013-03-28]. Dostupné z ⟨http://jcp.org/en/jsr/detail?id=317⟩.

[13] THE ECLIPSE FOUNDATION. *Java Persistence API (JPA) Extensions Reference for EclipseLink, Release 2.4* [online]. ©2012, 04/02/2013 14:09:19 [cit. 2013-05-16]. Dostupné z ⟨http://www.eclipse.org/eclipselink/documentation/2.4/jpa/extensions/toc.htm⟩.

[14] BAUER, Christian a Gavin KING. *Hibernate in action*. Greenwich: Manning Publications, 2005, xxiii, 408 s. ISBN 19-323-9415-X.

[15] THE HIBERNATE TEAM. *HIBERNATE - Relational Persistence for Idiomatic Java: Hibernate Reference Documentation 4.1.12.Final* [online]. ©2004 Red Hat, Inc., 2013-04-25 [cit. 2013-05-16]. Dostupné z ⟨http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html_single/⟩.

[16] ORACLE CORPORATION. *Java Data Objects (JDO)* [online]. 2013 [cit. 2013-04-20]. Dostupné z ⟨http://www.oracle.com/technetwork/java/index-jsp-135919.html⟩.

[17] MYBATIS.ORG. *MyBatis* [online]. ©2013 [cit. 2013-04-21]. Dostupné z ⟨http://mybatis.github.io/mybatis-3/⟩.

[18] ECLIPSE FOUNDATION *EclipseLink project* [online]. ©2013 [cit. 2013-04-21]. Dostupné z ⟨http://www.eclipse.org/eclipselink/⟩.

[19] THE HIBERNATE TEAM. *Hibernate Developer Guide* [online]. ©2011 Red Hat, Inc., 2013-04-25 [cit. 2013-05-03]. Dostupné z ⟨http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html/ch16.html⟩.

[20] LEONARD, Anghel. *JSF 2.0 Cookbook*. Birmingham: Packt publishing, 2010. ISBN 978-1-847199-52-2.

[21] FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, ©2003, xxiv, 533 s. ISBN 03-211-2742-0.

[22] ORACLE. *Java.net: The Source for Java Technology Collaboration* [online]. ©2013, 2013-04-26 [cit. 2013-05-04]. Dostupné z ⟨https://javaserverfaces.java.net/⟩.

[23] JBOSS. *RichFaces* [online]. ©2013, [cit. 2013-05-04]. Dostupné z ⟨http://www.jboss.org/richfaces⟩.

[24] JBOSS. *Seam Moving Forward* [online]. ©2013, 06. Mar 2013 [cit. 2013-05-04]. Dostupné z ⟨http://www.seamframework.org/⟩.

[25] CALISKAN, Mert a Oleg VARASKIN. *PrimeFaces Cookbook*. Birmingham: Packt Publishing, ©2013, January 2013, 328 s. ISBN 978-1-84951-928-1.

[26] APACHE SOFTWARE FOUNDATION. *Apache MyFaces* [online]. ©2013, April 1, 2013 [cit. 2013-05-04]. Dostupné z ⟨http://myfaces.apache.org/⟩.

[27] ICESOFT TECHNOLOGIES. *ICEfaces EE Overview* [online]. ©2013, 2013 [cit. 2013-05-04]. Dostupné z ⟨http://www.icesoft.org/java/products/ICEfaces-EE/overview.jsf⟩.

[28] JASPERSOFT. *JasperReports Library: Open Source Java Reporting Library* [online]. ©2013, 2013 [cit. 2013-05-04]. Dostupné z ⟨http://community.jaspersoft.com/project/jasperreports-library⟩.

[29] GRAHAM, Coleen aj. *Market Share: All Software Markets, Worldwide, 2012* [online]. ©2012 March 29, 2013 [cit. 2013-04-13]. Dostupné z ⟨http://www.gartner.com/id=2398815⟩.

Příloha A

Přehled zkratek

BMT – Bean Managed Transaction, transakce řízená v kódu session beanu.

CMT – Container Managed Transaction, transakce řízená aplikačním serverem.

CRUD – Create, read, update, delete – množina základních operací prováděných nad tabulkami transakční databáze.

EJB – Enterprise Java Bean, specifikace pro tvorbu serverových komponent v jazyce Java Enteprise Edition.

HQL – Hibernate Query Language, dotazovací jazyk pro objektově relační mapování Hibernate.

JPA – Java Persistence API, rozhraní pro persistenci objektů v programovacím jazyce Java.

JPQL – Java Persistence Query Language, dotazovací jazyk v prostředí Java Persistence API.

JSF – Java Server Faces, specifikace jazyka Java k vytváření uživatelského rozhraní webových aplikací.

JSP – Java Server Pages, technologie pro tvorbu dynamicky generovaných webových stránek na straně serveru pro jazyk Java.

JTA – Java Transaction API, rozhraní pro obsluhu transakcí při výměně informací v distribuovaném systému v prostředí Java EE.

Příloha B

Uživatelský manuál

B.1 Instalace aplikace

Aplikace je distribuována na přiloženém médiu ve formátu webového archivu `diploma.war`. Ke svému běhu vyžaduje instalován aplikační server Oracle Glassfish 3.1.2 a systém řízení báze dat Oracle 11g Express Edition. Před instalací aplikace je nezbytné vytvořit v aplikačním serveru connection pool a JTA datasource. K tomu slouží skript `asadmin` z podadresáře `bin` z instalačního adresáře aplikačního serveru, pro který je na přiloženém médiu vytvořen konfigurační soubor:

```
glassfish/bin/asadmin add-resources glassfish-resources.xml
```

V konfiguračním souboru je samozřejmě nejprve potřeba zkontrolovat a případně upravit přístupové údaje do databáze jako je adresa a název databáze či uživatelské jméno a heslo. K samotnému nasazení aplikace slouží příkaz `deploy`:

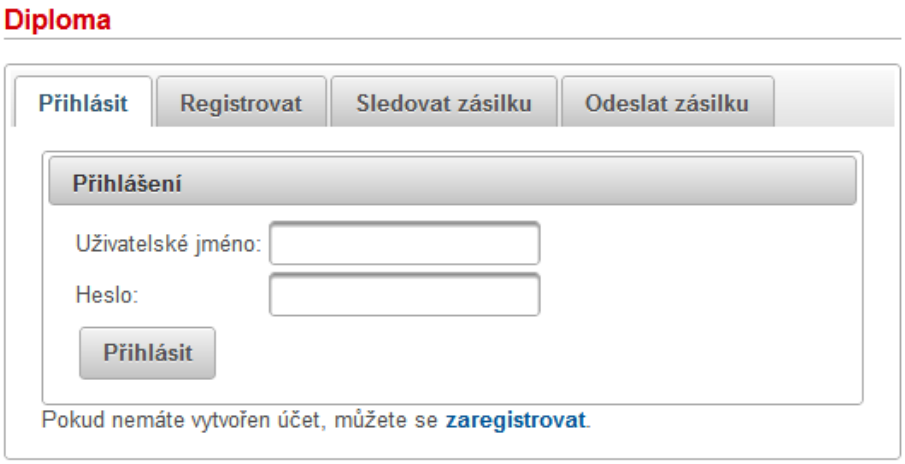
```
glassfish/bin/asadmin deploy diploma.war
```

V případě neexistence veškerých tabulek a sekvencí v databázi dojde při spuštění aplikace k jejich automatickému vytvoření. Přesto je součástí média SQL skript, který umožní tabulky vytvořit a provést jejich naplnění daty pro testování. Skript lze zavést v příkazovém řádku administračního programu `sqlplus`, který je součástí instalace Oracle 11g:

```
SQL> @diploma-db-dump.sql
```

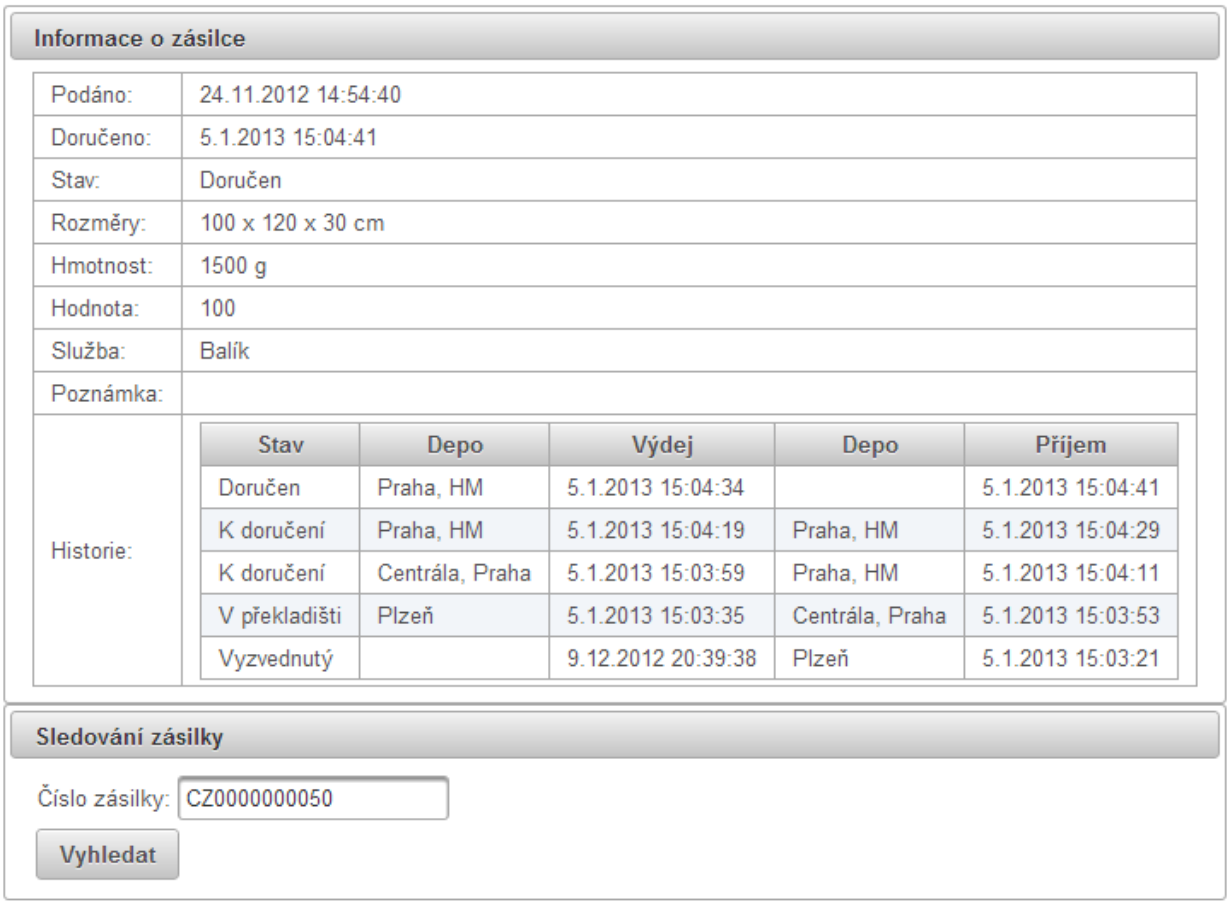
B.2 Běžný uživatel

Základní obrazovka aplikace (na obr. B.1) je určena nepřihlášeným uživatelům. Po vyplnění uživatelského jména a hesla a kliknutí na tlačítko *Přihlásit* je uživatel přesměrován do klientské sekce. Pokud uživatel nemá vytvořený účet, na kartě *Registrovat* je možné skrze jednoduchý registrační průvodce vytvořit nový uživatelský účet.



Obrázek B.1: Přihlášení do aplikace

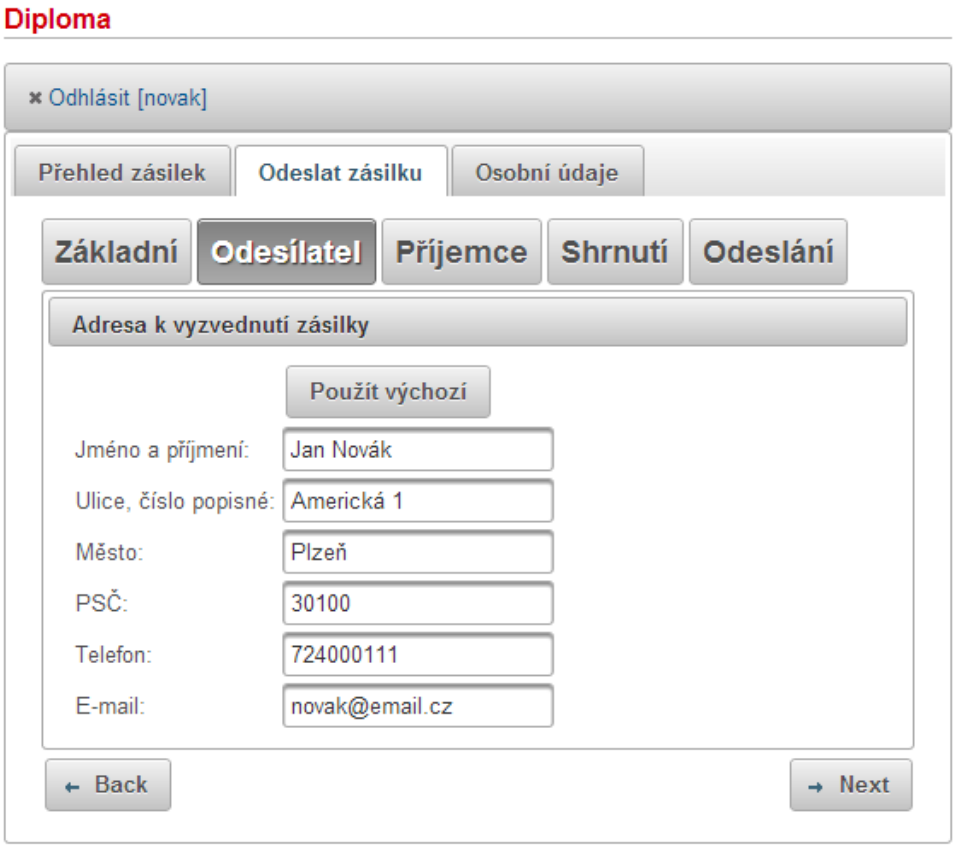
Karta *Sledovat zásilku* slouží k přístupu do systému Track&Trace pro sledování zásilek. Zadáním čísla zásilky uživatel získá přehled o stavu, v jakém se sledovaná zásilka právě nachází, včetně historie stavů a dalších detailů (obr.B.2).



Obrázek B.2: Sledování zásilky

Pro rychlé podání zásilky slouží karta *Odeslat zásilku*. Zde uživatel vyplní základní údaje o zásilce, adresu odesílatele a příjemce a po závěrečném potvrzení a vložení zásilky do systému je uživateli nabídnut rychlý odkaz pro sledování stavu a historie. Zadaná zásilka je okamžitě zobrazena na příslušném depu v seznamu pro vyzvednutí.

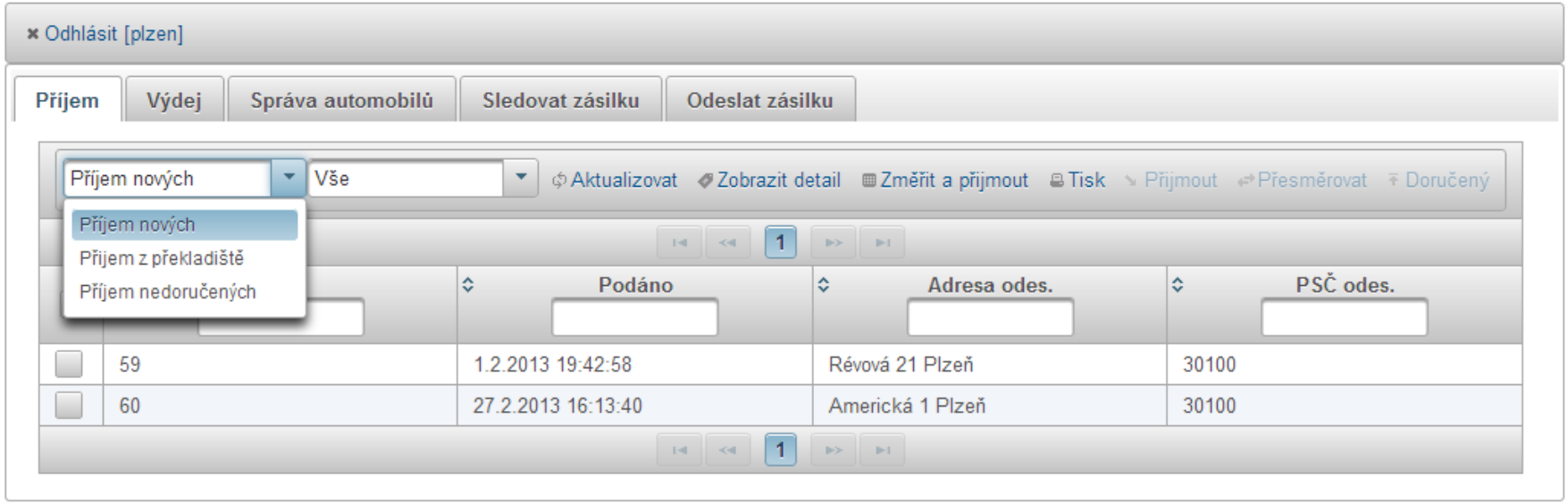
Po přihlášení do aplikace uživatel získává na kartě *Přehled zásilek* okamžitý přehled nad odeslanými zásilkami a při podání nové zásilky je k dispozici tlačítko *Použít výchozí*, kterým lze ve formuláři (obr. B.3) předvyplnit údaje o odesílateli na základě dat zadaných při registraci. Údaje lze později upravit na kartě *Osobní údaje*.



Obrázek B.3: Odeslání zásilky

B.3 Dispečer

V případě úspěšného přihlášení je dispečer směrován na kartu *Příjem*. Rozbalovací menu v horní části okna slouží k výběru způsobu příjmu, tedy zda je prováděn příjem nových zásilek, zásilek z překladiště, nebo zpětný příjem zásilek nedoručených. Filtrování lze dále upřesnit výběrem vozidla či řidiče v následujícím roletovém menu, případně vyplněním filtru nad příslušným sloupcem tabulky. Další možnosti na hlavní liště jsou aktivovány či deaktivovány zvolením způsobu příjmu. Tlačítkem *Zobrazit detail* lze k vybranému balíku zobrazit podrobnosti, jako je adresa odesílatele, příjemce, historie stavů a podobně. Klepnutím na *Změřit a přijmout* je zobrazen dialog pro vložení či úpravu naměřených údajů a dokončení příjmu. Přijatá zásilka je uskladněna na depu a umožněno její následné vydání z karty *Výdej*. Příjem zásilek zvážených na jiném depu lze uspíšit tlačítkem *Příjem*, kdy aplikace nevyžaduje zadání dalších podrobností.

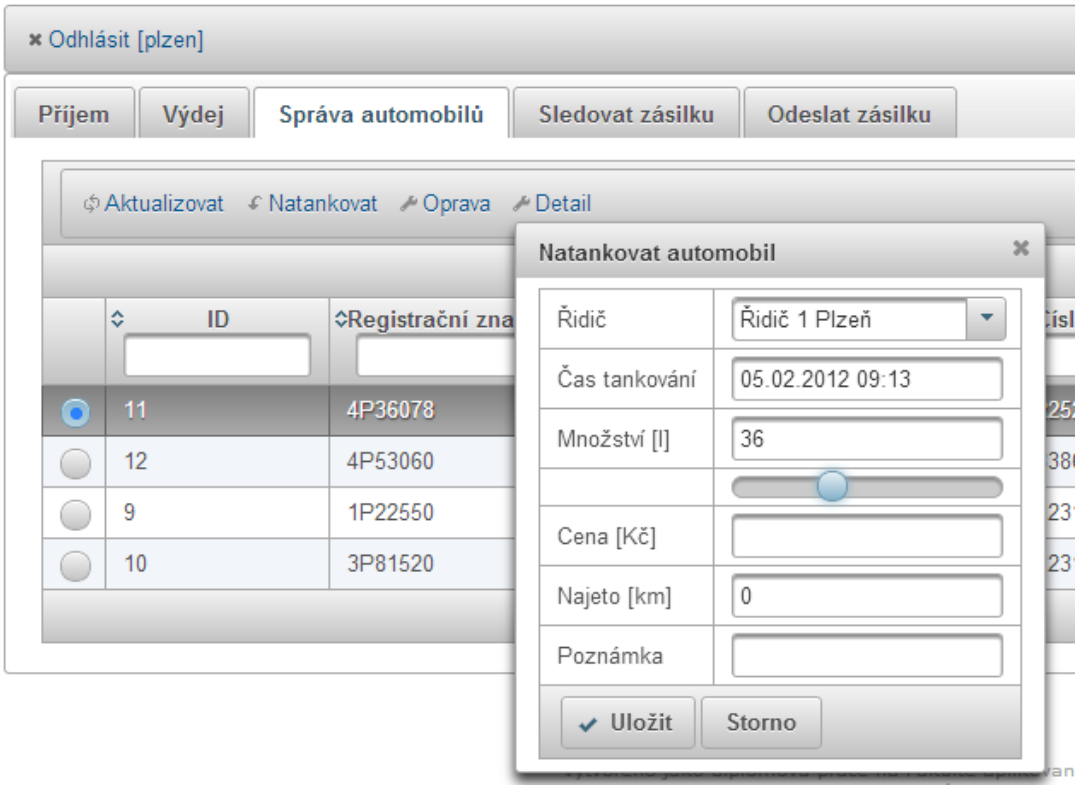


Obrázek B.4: Příjem zásilek

Tlačítko *Tisk* zajistí pro vybrané zásilky vytištění polepovacích etiket s čárovými kódy, příkazem *Přesměrovat* lze upravit adresu dodání a zásilku při příjmu rovnou směrovat na jiné depo a poslední tlačítko, *Doručený*, provede označení zásilky jako doručené. Taková zásilka bude v aplikaci skryta.

Na kartě *Výdej* probíhá výdej zásilek z depa obdobným způsobem jako jejich příjem. Tlačítkem *Naložit* lze vybrané zásilky vydat zvolenému řidiči a provést nakládku na vozidlo. Příkaz *Zadat svoz* slouží k potvrzení svozu podané zásilky. Svozový i nákladový list obsahující seznam zásilek pro řidiče se tiskne tlačítkem *Tisk*.

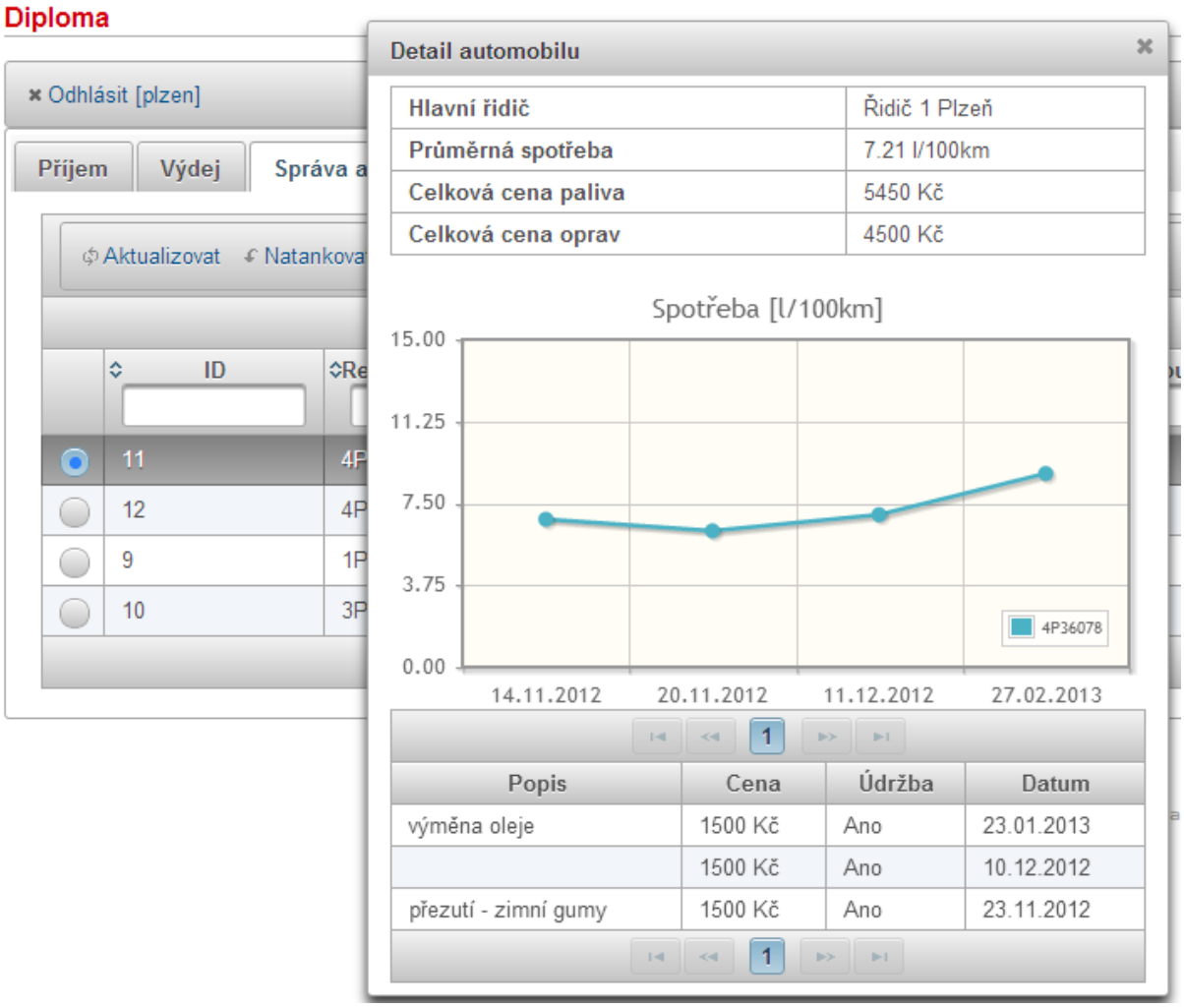
Evidence automobilů, které náleží depu přihlášeného dispečera, se provádí na kartě *Správa automobilů* (obr. B.5). Systém rovněž umožňuje k vedenému automobilu vložit opravu či údaje o údržbě (tlačítkem *Oprava*).



Obrázek B.5: Zadání tankování automobilu

Podrobné informace o zadaných událostech lze zobrazit tlačítkem *Detail*. Vyvolané dialogové okno obsahuje součty celkových nákladů na palivo, opravy a údržbu a průměrnou spotřebu automobilu. Tankování automobilu ve formě grafu průměrné spotřeby je zobrazeno uprostřed a opravy či údržba automobilu se nachází v tabulce v dolní části okna.

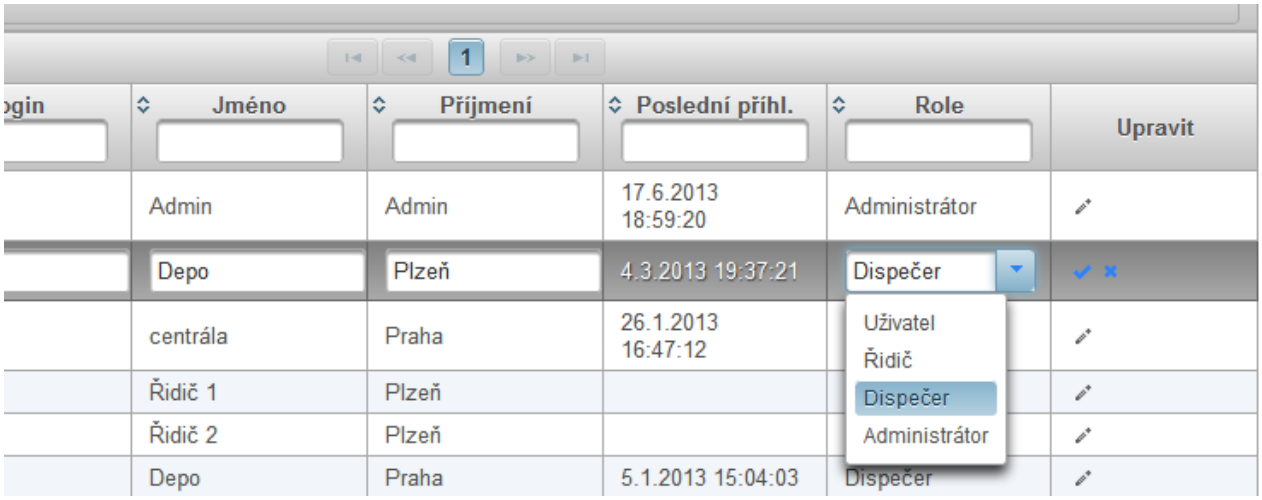
Dispečer má k dispozici rovněž karty pro sledování a podání zásilky, a to zejména z důvodu rychlého přístupu k těmto funkcím bez nutnosti odhlášení z aplikace.



Obrázek B.6: Přehled automobilu

B.4 Administrátor

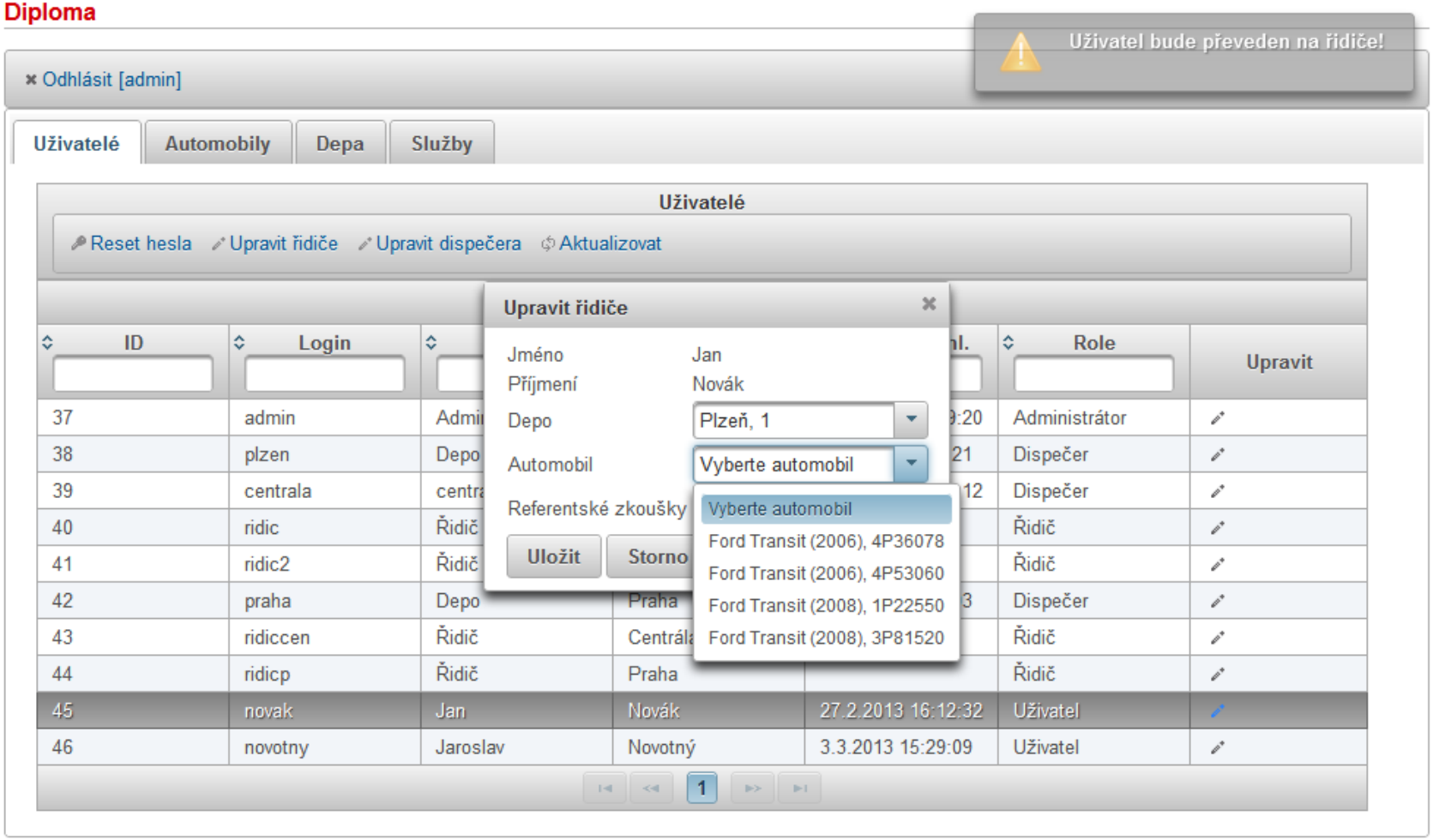
Panel administrátora na obrázku B.8 obsahuje několik karet pro úpravy kmenových dat. Karta *Uživatelé* slouží k úpravě údajů o uživatelích. Do editačního režimu lze přepnout libovolný řádek tabulky kliknutím na ikonu tužky ve sloupci *Upravit* (obr. B.7). Provedenou úpravu je následně nutno potvrdit nebo zamítnout kliknutím na příslušnou ikonu.



Obrázek B.7: Úprava uživatele

Dále je možno nastavit další parametry k vybraným rolím prostřednictvím tlačítek *Upravit řidiče* (obr. B.8) a *Upravit dispečera*. V případě, že se role uživatele liší od právě přidělované role, administrátor je upozorněn vyskakovací hláškou v pravém horním rohu okna.

Na kartách *Automobily*, *Depa* a *Služby* se nachází další tabulky pro úpravy kmenových dat, přičemž způsob práce je obdobný jako na kartě *Uživatelé*. Kromě základních údajů k příslušné doméně se jedná především o přiřazení automobilů na jednotlivá depa, nastavení routovacích tabulek, vlastností služeb a podobně.



Obrázek B.8: Panel administrátora – úprava řidiče

Příloha C

Obsah přiloženého média

Přiložené médium má následující adresářovou strukturu:

- `Projects` – adresář obsahující projekty pro vývojové prostředí NetBeans, obsahuje verzi projektu `diploma`, která používá knihovnu EclipseLink a `diplomahb` ve verzi pro knihovnu Hibernate,
- `Screens` – obrázky uživatelského rozhraní aplikace,
- `Sql` – skripty SQL pro založení tabulek naplněných testovacími daty,
- `TextImg` – soubory obrázků použité v textu práce v plném rozlišení či vektorovém formátu,
- `TextPdf` – samotný text práce ve formátu PDF,
- `TextTex` – text práce ve formátu Latex,
- `Xtras` – knihovny a případné další soubory popsané v souboru `readme.txt`,
- `readme.txt` – popis adresářové struktury a informace k instalaci a zprovoznění aplikace.