UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Vývoj webových aplikací v Javě na cloudové platformě Google App Engine

Bc. Martin Horák

Diplomová práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Martin Horák**
Osobní číslo:	**I09360**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vývoj webových aplikací v Javě na cloudové platformě Google App Engine**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V úvodní části práce bude popis technologií použitých při tvorbě aplikace, konkrétně Cloud computing, WebService a technologie J2EE.
Dále bude popsána objektová databáze, kterou Google prezentuje jako Bigtable, a možnosti přístupu k ní přes standardní API JPA nebo JDO. Spolu s databází je nutné popsat i Google File System (GFS), který databáze využívá pro persistenci dat.

Primárním cílem diplomové práce je praktická aplikace s cloudovou technologií Google App Engine, která se řadí mezi tzv. Platform as a Servis (PaaS) platformy. Součástí práce bude i její srovnání s klasickým webhostingem.

Praktická část bude obsahovat návrh a implementaci webové aplikace pro komunitní knihovnu.
Primárním jazykem bude vietnamština. V rámci aplikace bude rovněž k dispozici lokalizace do českého jazyka.
Návrh bude realizován v jazyce UML.
Aplikace bude umožňovat práci se seznamem evidovaných knih a realizací výpujček.
Produkt bude sloužit vietnamské komunitě v ČR.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

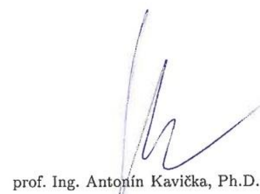
Seznam odborné literatury:

1. VELTE, Toby J.; VELTE, Anthony T.; ELSENPETER, Rober. Cloud Computing : Praktický průvodce. Brno : Computer Press, 2011. 304 s. ISBN 978-80-2513-333-0.
2. SANDERSON, Dan. Programming Google App Engine. First Edition. Sebastopol : O'Reilly Media, 2009. 367 s. ISBN 978-0-596-52272-8.
3. ROCHE, Kyle; DOUGLAS, Jeff. Beginning Java Google App Engine. USA : Apres, 2009. 236 s. ISBN 978-1-4302-2553-9.

Vedoucí diplomové práce: **prof. Ing. Karel Šotek, CSc.**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2012**
Termín odevzdání diplomové práce: **17. května 2013**

L.S.



prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2012

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 23. 8. 2013 Bc. Martin Horák

Poděkování

Rád bych poděkoval panu prof. Ing. Karlovi Šotkovi, CSc. za vedení diplomové práce. Dále bych chtěl poděkovat svým rodičům za podporu při studiu.

Anotace

Diplomová práce se zabývá tvorbou systému veřejné knihovny (SVK) pro vietnamskou komunitu v České republice na cloud computingové platformě Google Application Engine (GAE). V teoretické části jsou shrnuty základní aspekty a mechanismy cloud computingových technologií. Praktická část se zaměřuje na popis vývoje aplikace SVK v prostředí GAE včetně popisu analytické a návrhové části.

Klíčová slova

Bigtable, GAE, Google Application Engine, JDO, Java, Java Data Objects, Knihovna, Knihy, UML, Unified Modeling Language

Title

Development of web applications in Java on cloud computing platform Google Application Engine

Annotation

Diploma thesis is dealing with creation of Public Library System (PLS) for vietnamese community in the Czech Republic on cloud computing platform Google Application Engine (GAE). In theoretical part there are aspects and mechanisms of cloud computing techologies described. Practical part is focusing on description of development PLS application in GAE environment including description of analytical and design part.

Keywords

Bigtable, GAE, Google Application Engine, JDO, Java, Java Data Objects, Knihovna, Knihy, UML, Unified Modeling Language

Obsah

1 Úvod 15
2 Cloud computing 16
2.1 Historie 16
2.2 Definice cloud computingu 16
2.3 Modely nasazení cloud computingu 17
2.4 Servisní modely cloud computingu 18
2.5.1 Virtualizace ve vztahu ke cloud computingu 22
2.5.2 Webové služby 22
2.5.3 Architektura SOA 28
2.6 Přední poskytovatelé cloud computingu 29
2.6.1 Amazon 29
2.6.2 Microsoft 30
2.6.3 Google 32
2.7 Nevýhody cloud computingu 32
3 Java 2 Enterprise Edition (J2EE) 34
3.1 Úvod do J2EE 34
3.2 Servlety 34
3.3 Java Servlet Pages (JSP) 37
3.4 Perzistence dat 37
3.4.1 Java Data Objects (JDO) 38
3.4.2 Java Persistence Api (JPA) 46
4 Google App Engine (GAE) 47
4.1 Představení GAE 47
4.2 Architektura GAE 47
4.2.1 Podporovaná prostředí 48
4.2.2 Podpora jazyka Java pro J2EE na GAE 49
4.2.3 Vyřizování požadavků 49
4.2.4 Restrikce pro aplikace 51
4.3 GAE služby 51
4.3.1 Task Queues 52
4.3.2 Search 53
4.3.3 App Identity 55

4.4 GAE datastore ... 55
4.4.1 Google File System (GFS) ... 55
4.4.2 Bigtable ... 56
4.4.3 Typy GAE datastore ... 57
4.4.4 Porovnání GAE datastore s tradičními relačními databázemi ... 58
4.5 Základní konstrukty GAE datastore ... 58
4.5.1 Entita ... 58
4.5.2 Dotazy v GAE datastore ... 60
4.5.3 Indexy v GAE datastore ... 62
4.5.4 Transakce v GAE datastore ... 65
5 Systém veřejné knihovny ... 67
5.1 Analýza SVK ... 67
5.1.1 Požadavky ... 67
5.1.2 Aktéři ... 70
5.1.3 Případy užití ... 71
5.1.4 Doménový model ... 75
5.2 Návrh SVK ... 76
5.3 Implementace SVK ... 76
5.3.1 Struktura projektu ... 77
5.3.2 Aplikační rámec Struts 2 ... 78
5.3.3 Databázová vrstva ... 81
5.3.4 Výsledná podoba aplikace ... 81
5.3.5 Nasazení aplikace do prostředí GAE ... 82
5.3.6 Testování aplikace ... 83
6 Závěr ... 84
Příloha A – Vybrané artefakty modelu SVK ... 87
Příloha B – Zdrojový kód interceptoru ACLInterceptor.java ... 88
Příloha C – Adresářová struktura DVD ... 90

Seznam zkratek

API	Application Programming Interface
ASP	Active Server Pages
BCEL	Byte Code Engineering Library
CASE	Computer-aided software engineering
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRM	Computer Aided Software Engineering
CRUD	Create Read Update Delete
DaaS	Data as a Service
GA	Google Accounts
GAE	Google Application Engine
GD	Google Data
GFS	Google File System
HMTL	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HR	High Replication
J2EE	Java 2 Enterprise Edition
JDO	Java Data Objects
JDOQL	Java Data Objects Query Language
JDK	Java Development Kit
JPA	Java Persistence Api
JRE	Java Runtime Environment
JSP	Java Server Pages
MVC	Model View Controller
NaaS	Network as a Service
NIST	The National Institute of Standards and Technology
OGNL	Object Graph Navigation Library
PaaS	Platform as a Service
PHP	Hypertext Preprocessor
POJO	Plain Old Java Object
REST	Representational State Transfer
RMI	Remote Method Invocation
SaaS	Software as a Service
SDK	Software Development Kit
SLA	Service Level Agreement
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SQL	Structured Query Language
SSL	Secure Sockets Layer
SVK	Systém veřejné knihovny

UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VPN	Virtual Private Network
WADL	Web Application Description Language
WSDL	Web Services Description Language
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
XML-RPC	XML Remote Procedure Call
XSD	XML Schema Definition

Seznam ukázek kódu

Kód 0 – Popisek kódu 14
Kód 1 – Ukázka SOAP zprávy 23
Kód 2 – Popis RESTful webové služby prostřednictvím WADL 27
Kód 3 – Ukázka entitní třídy Kniha v JDO DataNucleus 40
Kód 4 – XML Konfigurace PersistenceManagerFactory v JDO DataNucleus. 41
Kód 5 – Získání objektu typu PersistenceManagerFactory. 41
Kód 6 – Transakce v JDO. 41
Kód 7 – Perzistence objektu typu Kniha. 44
Kód 8 – Získání objektu typu jazyk prostřednictvím metody getObjectById. 45
Kód 9 – Získání všech objektů typu Kniha s použitím extentu a následnou iterací. 45
Kód 10 – Ukázka mapování deskriptoru web.xml pro Java aplikaci. 50
Kód 11 – Získání aktuálního uživatele pomocí služby Users. 52
Kód 12 – Ukázka konfigurace fronty typu push v souboru queue.xml. 53
Kód 13 – Ukázka zařazení úlohy do GAE fronty v Javě. 53
Kód 14 – Vytvoření dokumentu pro objekt typu Kniha. 54
Kód 15 – Vytvoření dokumentu pro objekt typu Kniha. 54
Kód 16 – Vytvoření dokumentu pro objekt typu Kniha. 55
Kód 17 – Ukázka použití parametru v JDO. 61
Kód 18 – Ukázka použití filtru v JDO. 61
Kód 19 – Ukázka použití rozsahu v JDO. 62
Kód 20 – Ukázka použití řazení v JDO. 62
Kód 21 – Obsah souboru datastore-indexes.xml. 64
Kód 22 – Dotaz pro perfect index. 65
Kód 23 – Využití DatastoreService v GAE transakci. 66
Kód 24 – Výřez z konfiguračního souboru tiles.xml. 78
Kód 25 – Výřez z konfiguračního souboru urlrewrite.xml. 79
Kód 26 – Výřez z konfiguračního souboru urlrewrite.xml. 80
Kód 27 – Konfigurační soubor appengine-web.xml pro SVK. 82
Kód 28 – Interceptor ACLInterceptor.java. 89

Seznam obrázků

Obrázek 1 – Schéma vztahu mezi modely nasazení cloud computingu 18
Obrázek 2 – Infrastructure as a Service (IaaS). 19
Obrázek 3 – Platform as a Service (PaaS). 20
Obrázek 4 – Software as a Service (SaaS). 21
Obrázek 5 – Využití SOAP, WSDL a UDDI u webových služeb. 22
Obrázek 6 –Schéma J2EE architektury. 34
Obrázek 7 – Zpracování HTTP požadavku, krok 1. 35
Obrázek 8– Zpracování HTTP požadavku, krok 2. 35
Obrázek 9 – Zpracování HTTP požadavku, krok 2. 36

Obrázek 10 – Zpracování HTTP požadavku, krok 2 36
Obrázek 11– Zpracování HTTP požadavku, krok 2 36
Obrázek 12 – Zpracování HTTP požadavku, krok 2 37
Obrázek 13 – Komunikace při užití architektonického vzoru MVC n platformě J2EE..... 37
Obrázek 14 – Přechody mezi hlavními stavy objektů v JDO. 43
Obrázek 15 – Obsluha požadavku na GAE. 49
Obrázek 16 – Nastavení výkonové třídy frontendu na GAE. 50
Obrázek 17 – Architektura GFS. 56
Obrázek 18 – Milníky potvrzení transakce 66
Obrázek 19 – Funkční požadavky kategorie Uživatelé. 68
Obrázek 20 – Funkční požadavky kategorie Knihy. 68
Obrázek 21 – Funkční požadavky kategorie Výpůjčky. 69
Obrázek 22 – Nefunkční požadavky kategorie Jazykové mutace. 69
Obrázek 23 – Nefunkční požadavky kategorie Podpora. 69
Obrázek 24 – Nefunkční požadavky kategorie Přístupnost. 70
Obrázek 25 – Nefunkční požadavky kategorie Technologie. 70
Obrázek 26 – Interní aktéři. 70
Obrázek 27 – Externí aktéři. 71
Obrázek 28 – Diagram případů užití Knihy. 71
Obrázek 29 – Diagram případů užití Správa účtů. 72
Obrázek 30 – Realizace UC001 (analytické třídy). 73
Obrázek 31 – Realizace UC001 (sekvenční diagram). 74
Obrázek 32 – Realizace UC005 (analytické třídy). 74
Obrázek 33 – Realizace UC005 (sekvenční diagram). 75
Obrázek 34 – Doménový model SVK. 75
Obrázek 35 – Javascriptová část frontendu. 76
Obrázek 36 – Struktura projektu SVK v NetBeans. 77
Obrázek 37 – Ukázka z vyvinuté aplikace SVK. 81
Obrázek 38 – Matice pokrytí funkčních požadavků případy užití. 87
Obrázek 39 – Datový model SVK. 87

Seznam tabulek

Tabulka 1 – WSDL elementy. 25
Tabulka 2 – WADL elementy. 25
Tabulka 3 – Vybrané JDO anotace. 38
Tabulka 4 – Urovně izolace transakcí. 42
Tabulka 5 – Java datové typy pro GAE datastore. 59
Tabulka 6 – Strukturu indexové tabulky EntitiesByKind. 63
Tabulka 7 – Strukturu indexové tabulky EntitiesByProperty. 63
Tabulka 8 – Strukturu indexové tabulky EntitiesByCompositeProperty. 63
Tabulka 9 – Struktura indexové tabulky pro ukládání sekvenci. 64
Tabulka 10 – Jmenné konvence artefaktů analýzy SVK. 67

Tabulka 11 – Hlavní scénář případu užití UC014. 72
Tabulka 12 – Přehled Maven goals pro GAE. 82

Typografické konvence

Diplomová práce používá následující typografické konvence:

- **Tučně** jsou uvedeny nové důležité pojmy.
- *Kurzíva* je použita pro názvy typů, identifikátorů. Takté je použita pro pojmy, které chce autor zdůraznit.
- Pro zkratky se využívají kapitálky (např. GAE).
- Pokud je v textu uveden krátký výřez zdrojového kódu, je použito `neproporcionální písmo`.

Pro komplexní příklad zdrojového kódu je použit následující formát společně s popiskem:

```
public List<Hodnoceni> getList(Set<Key> keys) {
    if (keys.isEmpty()) {
          return new ArrayList<Hodnoceni>();
    }
    Query q = pm.newQuery(Hodnoceni.class, ":p.contains(key)");
    List<Hodnoceni> hodnoceni = (List<Hodnoceni>) q.execute(keys);

    return hodnoceni;
}
```

Kód 0 – Popisek kódu. Zdroj: vlastní

1 Úvod

Diplomová práce pojednává o vývoji webových aplikací pro cloud computingovou platformu Google Application Engine (GAE). Cílem práce je vytvořit funkční systém veřejné knihovny (SVK) pro vietnamskou komunitu v České republice s využitím pokročilých technologií platformy Java 2 Enterprise Edition (J2EE) na GAE.

Kapitola 2 vysvětluje základní principy cloud computingu. Seznamuje čtenáře s jeho historií, klíčovými aspekty, výhodami a nevýhodami. Dále podává informace o hlavních poskytovatelích cloud computingových služeb společně se sumarizací jejich aktuální nabídky produktů.

Kapitola 3 je věnována vybraným technologiím J2EE, zejména pak řešení Java Data Objects (JDO) pro přístup k objektové databázi. Tato kapitola společně s následující představuje teoretický základ pro praktickou část diplomové práce.

Platforma GAE je obsahem kapitoly 4. Obsahuje podrobné popisy základních mechanismů platformy a také postupy integrace vybraných Google služeb do webové aplikace.

Poslední kapitola postupně popisuje jednotlivé fáze tvorby systému veřejné knihovny na GAE. Hlavní důraz je kladen na analytickou a implementační část. Součástí jsou kromě ukázek zdrojových kódů také vybrané diagramy tvořené v modelovacím jazyku Unified Modeling Language (UML).

Od čtenáře nejsou předpokládány žádné předchozí znalosti uvedené problematiky.

2 Cloud computing

Tato kapitola si klade za cíl uvést čtenáře do problematiky cloud computingu a seznámit jej se základními pojmy.

2.1 Historie

Myšlenka konceptu cloud computingu se poprvé objevila v 60. letech. O tom, komu má být její prvotní autorství přisuzováno, se dodnes vedou spory. Podle některých s ní jako první přišel J.C.R. Licklider, americký psycholog a informatik, který se zasloužil například o zajištění financování vývoje armádní sítě ARPANET. Vize J.C.R. Licklidera počítala s tím, že v budoucnosti bude každý uživatel počítače propojen s ostatními uživateli a přistupovat k programům a datům odkudkoli na zeměkouli. Tato předpověď se z dnešního pohledu jeví jako nejvíce přesná. Druhou variantou pak byla vize Lickliderova krajana profesora Johna McCartyho, podle kterého bude výpočetní výkon prodáván stejně jako kterákoli jiná služba. [1]

Jedním z milníků cloud computingu se stalo spuštění portálu *Salesforce.com* v roce 1999, který nabízel podnikové CRM aplikace přes web (služba běží s velkým úspěchem dodnes). O tři roky později následoval start *Amazon Web Services* nabízející cloudové datové uložiště, výpočetní výkon a dokonce i umělou inteligenci ve službě *Amazon Mechanical Turk*. V roce 2006 Amazon spustil *Elastic Compute cloud* (EC2), komerční webovou službu pro malé podniky i jednotlivce. Ty si mohly pronajmout počítače, na kterých bylo možné spouštět vlastní aplikace. [1], [2]

Další hráči vstoupili na trh v roce 2009, mezi nejvýznamnější z nich patří Google. Začaly se nabízet podnikové aplikace pro webové prohlížeče skrz služby podobné *Google Apps*[1]. Cloudové služby hlavních poskytovatelů na trhu shrnuje kapitola 2.5. [1]

K transformaci cloud computingu do dnešní podoby významnou měrou přispěly převážně tyto skutečnosti [1]:

- rozvoj virtualizace,
- rozvoj širokopásmového internetového připojení,
- vytvoření standardů pro spolupráci mezi softwarovými aplikacemi.

2.2 Definice cloud computingu

Definice cloud computingu není zcela přesně vymezena. Americký The National Institute of Standards and Technology (NIST) cloud computing definuje jako model pro umožnění přístupu do sdíleného fondu konfigurovatelných výpočetních zdrojů. [3]

Cloudový model je dle NIST charakterizován pěti hlavními koncepty [3]:

[1] Sada cloudových kancelářských aplikací.

- **On-demand self-service** – uživatel si může sám vyžádat prostředky (výpočetní časové kvantum, síťové datové uložiště) bez nutnosti (lidské) interakce na jeho straně.
- **Broad network access** – prostředky jsou dostupné přes síť prostřednictvím standardních mechanismů (podporováni jsou tencí i tlustí klienti).
- **Resource pooling** – prostředky jsou sdíleny mezi více uživateli. Alokují a dealokují se dynamicky. Uživatel typicky nemá možnost specificky ovlivnit výběr z dostupných prostředků s výjimkou možnosti definovat požadavek na přidělení prostředku na vyšší úrovni abstrakce (např., z které lokace má být prostředek přidělen).
- **Rapid elasticity** – prostředky je možné alokovat a dealokovat pružně v závislosti na situaci. Uživateli se navíc běžně jeví, že cloudové prostředky jsou nevyčerpatelné.
- **Measured service** – systémy založené na cloudu automaticky optimalizují a kontrolují prostředky pomocí k tomu uzpůsobených metrik na vyšší úrovni abstrakce. Úroveň abstrakce je odvozena od typu poskytované služby (např. měření využití datového prostoru, počtu aktivních uživatelů, …).

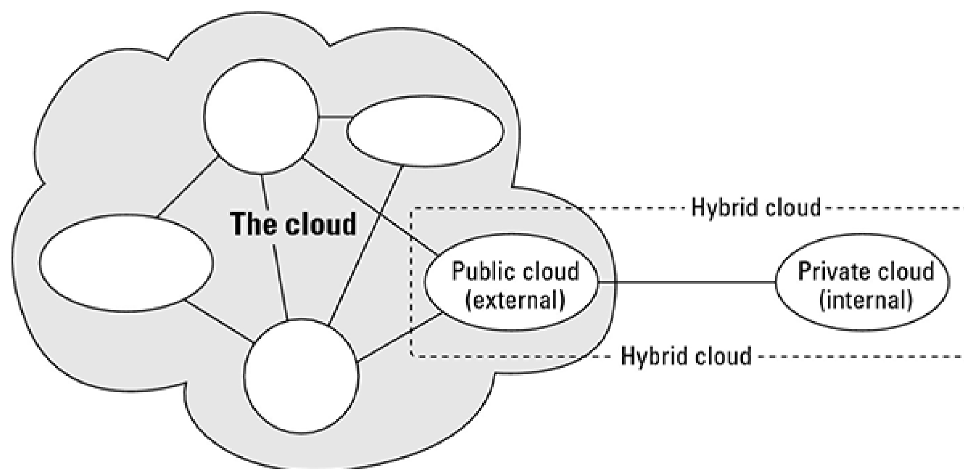
Knižní titul Cloud Computing Bible [4] definuje cloud computing zase jako prostředek přetváření technologií, služeb a aplikací dostupných v síti Internet do samoobsluhovatelné služby. Slovo „cloud“ ve spojení „cloud computing“ pak odkazuje na dva základní koncepty:

- **Abstrakce** – cloud computing abstrahuje od implementačních detailů. Data jsou z pohledu uživatele uložena na neznámém místě, aplikace běží na blíže nespecifikovaných systémech apod.
- **Virtualizace** – cloud computing virtualizuje systémy sdílením prostředků a jejich poolingem [2]. Prostředky jsou zpřístupňovány z centralizované infrastruktury na základě aktuální potřeby uživatelů.

2.3 Modely nasazení cloud computingu

Model nasazení definuje účel cloudu a blíže specifikuje jeho organizaci. Schéma vztahu mezi jednotlivými modely nasazení ukazuje obrázek 1. [4]

[2]Poolingem je myšleno sdružování prostředků do společného fondu.



Obrázek 1 – Schéma vztahu mezi modely nasazení cloud computingu. Zdroj: [4]

Veřejný cloud computing

Infrastruktura cloudu je v tomto případě dostupná komukoli, kdo chce jeho služeb využít. Typicky je vlastněna organizací prodávající cloudové služby. [4]

Privátní cloud computing

Privátní cloudová infrastruktura je vyhrazena pro účely zvolené organizace. Obvykle je spravována přímo vlastnickou organizací nebo třetí stranou. Infrastruktura se může nacházet buďto v prostorách organizace, nebo umístěna externě. [4]

Komunitní cloud computing

Komunitní cloud je takový cloud, který je organizován jako služba pro plnění nějakého společného účelu či funkce. [4]

Hybridní cloud computing

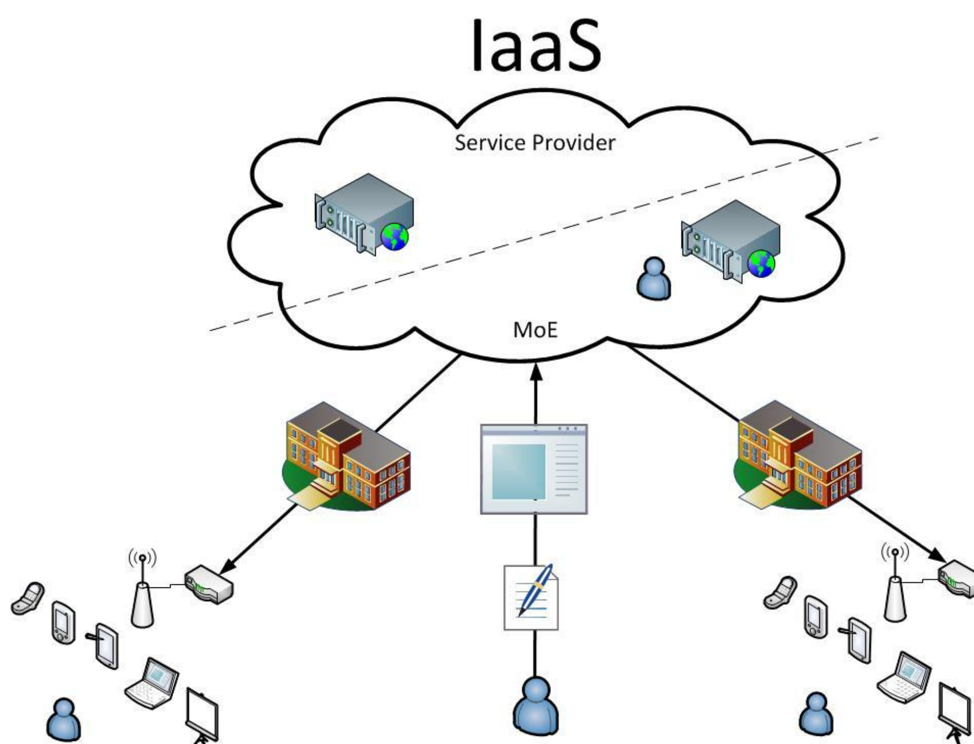
Jedná se o kombinaci předchozích modelů, kdy si jednotlivé entity zachovávají své unikátní vlastnosti. Podstatná je jejicch organizace do společné jednotky. Hybridní cloud často nabízí standardizovaný nebo proprietární přístup k datům a aplikacím. [4]

2.4 Servisní modely cloud computingu

V rámci cloud computingu se rozeznávají různé typy servisních modelů. Tyto se dělí podle služeb, které svým potenciálním uživatelům mohou nabídnout. Mezi hlavní tři typy modelů se řadí IaaS, PaaS a SaaS. Níže jsou uvedeny stručné informace ke každému z nich. [3], [4]

Infrastructure as a Service (IaaS)

Servisní model IaaS nabízí uživateli využití výpočetního výkonu, kapacity datového uložiště či šířku pásma. Uživatel má možnost ovládat a konfigurovat operační systém běžící na infrastruktuře poskytovatele. Schéma modelu SaaS znázorňuje rich picture diagram na obrázku 2. [3], [4]



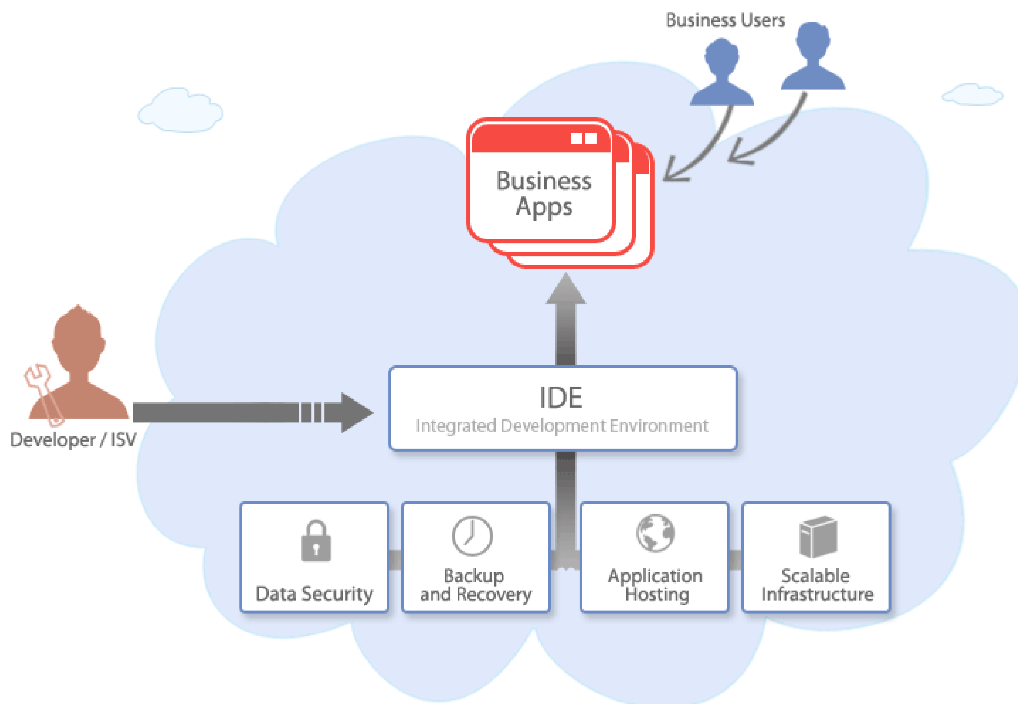
Obrázek 2 – Infrastructure as a Service (IaaS). Zdroj: [5]

Mezi hlavní výhody modelu IaaS patří:

- odpadne potřeba spravovat vlastní podnikovou infrastrukturu,
- minimalizace operačních nákladů,
- cena odvozena od míry užívání prostředků infrastruktury,
- dynamické škálování služeb.

Platform as a Service (PaaS)

Servisní model PaaS umožňuje nasadit aplikaci vytvořenou s použitím programovacího jazyka, knihoven, služeb a nástrojů poskytovatele do jeho cloudové infrastruktury. Infrastruktura sestáva ze sítě, serveru, operačního systému a datového uložiště. Uživatel nemá možnost konfigurovat infrastrukturu poskytovatele ani ji jiným způsobem ovlivňovat. Schéma modelu PaaS je znázorněno formou rich picture diagramu na obrázku 3. [3], [4]



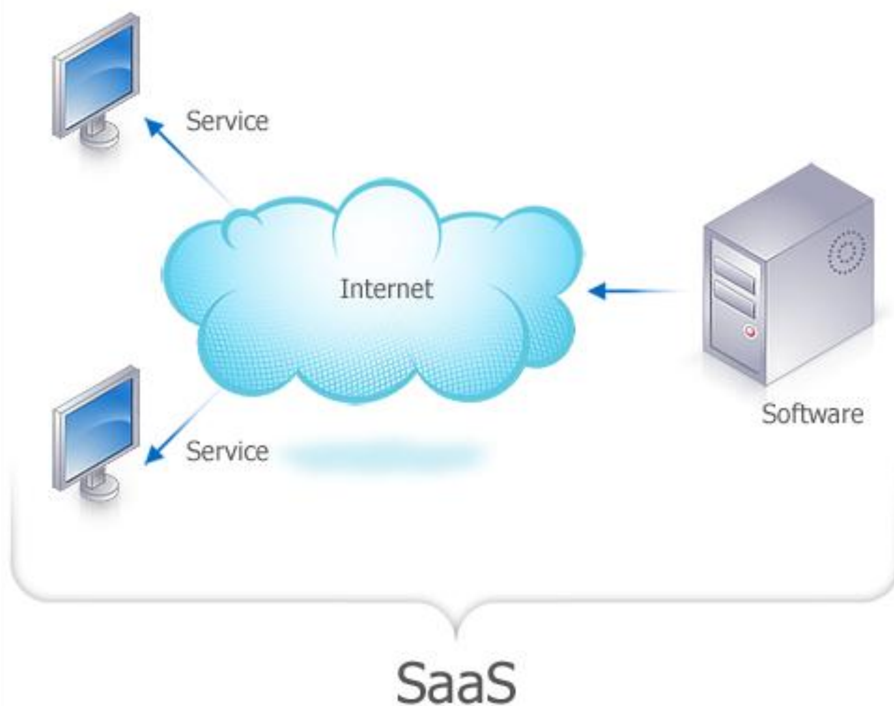
Obrázek 3 – Platform as a Service (PaaS). Zdroj: [6]

Mezi hlavní výhody modelu PaaS patří [6]:

- časová akcelerace uvedení produktu na trh (tzv. *time to market*),
- minimalizace operačních nákladů,
- minimalizace počátečních investic (není nutný nákup hardwaru, instalace a konfigurace prostředí apod.)
- snadná integrace s webovými službami.

Software as a Service (SaaS)

Servisní model SaaS umožňuje uživateli využívat aplikaci či aplikace běžící na cloudové infrastruktuře poskytovatele. Aplikace je přístupná z různých zařízení (typicky tenký klient). Uživatel si aplikaci pouze pronajímá a za tuto službu mu je ze strany poskytovatele periodicky účtován stanovený poplatek. Schéma modelu SaaS znázorňuje rich picture diagram na obrázku 4. [3], [4]



Obrázek 4 – Software as a Service (SaaS). Zdroj: [7]

Mezi hlavní výhody modelu SaaS patří [7], [8]:

- absence poplatku za nákup aplikace,
- odpadá nutnost instalace a aktualizace softwaru,
- zajištění neustálých upgradů aplikace zdarma (poskytovatel je trhem motivován své služby neustále zdokonalovat),
- zpřístupnění původně finančně nedostupných aplikací pro malé a střední podniky,
- uživatel využívá pouze funkcionalitu pronajímané aplikace (nedostane se již k samotné implementaci).

Další servisní modely

Kromě uvedené trojice se lze setkat ještě s dalšími typy servisních modelů:

- **Network as a Service (NaaS)** – Servisní model NaaS je zaměřen na poskytování síťových a transportních služeb a to jak směrem do cloudu, tak i ven. NaaS obvykle zahrnuje také protokol VPN.
- **Datastore as a Service (DaaS)** – Servisní model DaaS poskytuje uživatelům přístup ke cloudové databázi prostřednictvím k tomu určeného API. Dnes jsou již podporovány jak SQL, tak i NoSQL databázové enginy.

2.5 Technologie stojící za cloud computingem

V této podkapitole jsou zmíněny nejdůležitější technologie, na kterých cloud computing staví.

2.5.1 Virtualizace ve vztahu ke cloud computingu

Pojmy virtualizace a cloud computing jsou v běžné praxi často zaměňovány. Hlavní rozdíl je zejména v tom, že zatímco cloud computing je spíše přístup k poskytování služeb koncovému uživateli, virtualizace je pouze jednou ze služeb, kterou lze v rámci cloud computingu nabízet. Samotná virtualizace totiž neobsahuje servisní vrstvu, v rámci které by výpočetní kapacity bylo možné nabídnout koncovému uživateli. [9]

Aby bylo možné virtualizaci nabízet jako službu, je třeba zapojit kombinaci vhodných nástrojů, procesů a architektury. Hovoří se o tzv. **orchestraci.** Právě orchestrace umožňuje správu, koordinaci a řízení komplexních počítačových systémů. Pro cloud computing se tak stává esenciální technikou. [9]

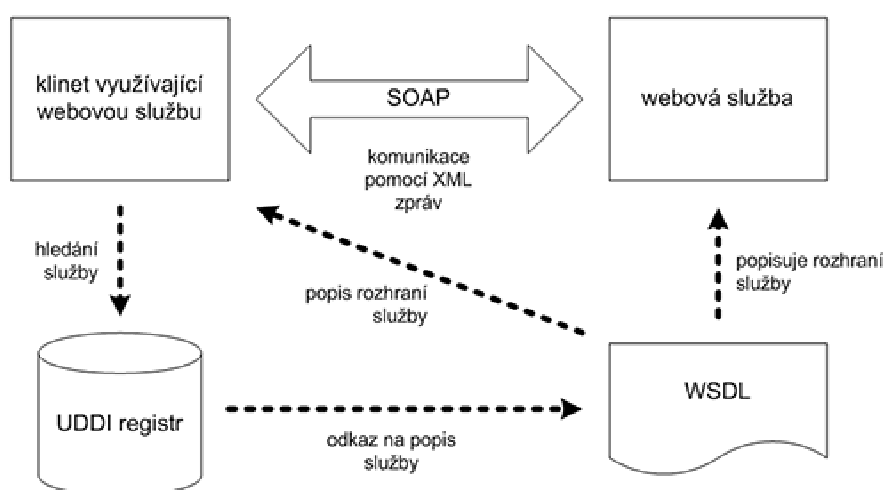
2.5.2 Webové služby

Webové služby jsou orientované na zasílání zpráv a jsou nezbytné pro realizaci distribuované výpočetní architektury, která je jádrem cloud computingu. Hlavní výhoda webových služeb spočívá ve využívání standardních webových protokolů (HTTP, XML, TCP/IP) a především možnosti jejich vzájemné komunikace bez nutnosti zabývat se implementačními detaily. [10]

Webové služby poskytují svým uživatelům [10]:

- funkce přes standardní webový protokol (obvykle SOAP, REST nebo XML-RPC),
- popis svého rozhraní (obvykle přes WSDL nebo WADL),
- jejich snadné nalezení prostřednictvím UDDI, případně jiného registru webových služeb.

Výše zmiňované technologie tvoří základní stavební kameny webových služeb. Vzájemnou spolupráci SOAP, WSDL a UDDI pro ilustraci zachycuje obrázek 5.



Obrázek 5 – Využití SOAP, WSDL a UDDI u webových služeb. Zdroj: [11]

SOAP

SOAP je zkratka pro Simple Object Access Protocol. Jedná se o standard pro komunikační protokol webových služeb (vznikl ještě před standardy WSDL a UDDI). SOAP pro své zprávy definuje vlastní XML formát. Pracuje na principu *peer-to-peer*, přičemž zpráva představuje jednosměrný přenos informace od odesílatele k příjemci. Nespornou výhodou SOAP (stejně jako REST) je, že zprávy se při komunikaci mezi aplikacemi přenesou přes síťově firewally mnohem pravděpodobněji, než kdyby aplikace komunikovaly přímo. HTTP protokol, který SOAP pro přenos využívá, je totiž ve firewallech v drtivé většině případů povolen. SOAP navíc nabízí i podporu transakcí. [8], [11], [12]

Standardní scénář komunikace prostřednictvím SOAP probíhá v těchto krocích [11]:

1. Iniciátor zašle zprávu aplikaci.
2. Aplikace na základě obsahu zprávy obslouží požadavek.
3. Aplikace vytvoří SOAP zprávu obsahující odpověď na požadavek a zašle ji iniciátorovi.

SOAP zpráva je jednoduchý XML dokument s kořenovým elementem *envelope*. V kořenovém elementu se nacházejí tagy *header* a *body* reprezentující hlavičku a tělo zprávy. Hlavička je nepovinná a obvykle obsahuje identifikační a autentizační údaje uživatele. Tělo obsahuje samotnou přenášenou informaci. V těle se dále může vyskytovat nepovinný element *fault* používaný pro přenos informací o chybách. Příklad jednoduché SOAP zprávy znázorňuje kód 1. [11], [12], [8]

```
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>
</soap:Envelope>
```

Kód 1 – Ukázka SOAP zprávy. Zdroj: [13]

REST

REST (Representational State Transfer) byl poprvé představen v disertační práci jednoho ze spoluautorů protokolu HTTP, Roye Fieldinga. Jedná se o architekturu rozhraní pro distribuované prostředí. Hlavní rozdíl v REST oproti SOAP či XML-RPC spočívá v orientaci na data, nikoli na procedury. Zatímco webové služby se zabývají definicí procedur a protokolů pro jejich volání, REST specifikuje možnosti přístupu k datům. Webové služby, které REST API využívají, se běžně označují jako RESTful. [14]

Pomocí REST lze snadno přistupovat ke zdrojům. Zdrojem se rozumí nejen data, ale také stavy aplikace. Zdroj je charakterizován jednoznačným URI identifikátorem. [14]

REST definuje čtveřici metod funkčně shodnou se známým akronymem práce s daty CRUD (Create, Read, Update, Delete) [14]:

- GET – Jak bylo řečeno, každý zdroj lze jednoznačně identifikovat podle jeho URI. Metoda GET protokolu HTTP zajistí získání přístupu ke zdroji. K požadavku lze navíc připojit tzv. *query parametry*, které se píší za otazníkem. Pokud bychom chtěli získat např. seznam pracovníků firmy z oddělení s ID 1, sestavili bychom následující HTTP požadavek: `GET /oddeleni?id=1 Host: nejakafirma.cz`. Tento přístup zřejmě není žádná novinka a na internetu se využívá běžně (webové prohlížeče tento požadavek sestavují ze zadaného URI).
- POST – Tato metoda slouží pro vytvoření dat. Opět se nejedná o nic neznámého. Využívají ji totiž minimálně HTML formuláře. Je zřejmé, že v okamžiku volání této metody ještě není známo URI zdroje (ten neexistuje). Používá se tedy společný koncový bod.
- DELETE – Metoda slouží pro odstranění zdroje nacházejícím se na zadaném URI. Odstranění seznamu zaměstnanců z předchozího příkladu by tedy zajistil takovýto HTTP požadavek: `DELETE /oddeleni?id=1 Host: nejakafirma.cz`. Tato metoda může být problematická. Většina HTTP a HTML nástrojů (formuláře) ji nepodporuje. V praxi se proto nahrazuje metodou POST s příslušným parametrem implikujícím mazání.
- PUT – Používá se pro změnu zdroje. Rámcově je podobná metodě POST s tím rozdílem, že URI zdroje je již známo. Co se týče podpory, PUT trpí stejným problémem jako DELETE. Je tedy také nahrazován nejčastěji metodou POST.

Na místě je srovnání technologie REST, jejíž popularita stále roste, se SOAP. Kromě kritéria flexibility a podpory transakcí REST převažuje především v těchto oblastech [8]:

- Snižuje zatížení serveru a zrychluje reakční dobu. Toto je umožněno především díky cachování přenášených dat.
- Nižší režie údržby relací.
- Požadavek na RESTful službu lze odeslat z adresního řádku libovolného prohlížeče (stejně tak přijmout odpověď). Není nutné doprogramovávat klienta jako v případě SOAP.

REST byl využit při realizaci webové služby vracející seznam knižních titulů SVK (viz kapitola 5).

WSDL

WSDL (Web Service Description Language) popisuje zprávy a způsob jejich výměny. Je to jazyk pro popis rozhraní webových služeb. WSDL jednoznačně specifikuje, jak má vypadat zpráva s požadavkem a zpráva s odpovědí. Notace WSDL je založena na XML Schema. WSDL je od roku 2007 doporučováno konsorciem W3C. [11], [12]

WSDL dokument pro popis webové služby využívá elementy shrnuté v tabulce 1.

Tabulka 1 – WSDL elementy. Zdroj: [15]

Element	Popis
<types>	Slouží jako kontejner pro datové typy, se kterými webová služba pracuje.
<message>	Slouží pro popis parametrů operací, které webová služba nabízí.
<portType>	Nejdůležitější element WSDL. Popisuje webovou službu, její operace a zprávy. Operace se dle WSDL dělí do čtyř typů: • One-way – operace může přijmout zprávu, ale nevrací odpověď. • Request-response – standardní model odpověď-požadavek. • Solicit-response – operace může odeslat požadavek a bude čekat na odpověď. • Notification – operace může odeslat požadavek a nebude čekat na odpověď.
<binding>	Pro každý *portType* definuje formát dat a používaný protokol.

Právě element *binding* zajišťuje vysokou Flexibilitu WSDL. Lze jej použít například i pro SMTP servery.

WADL

WADL (Web Application Description Language) je alternativou k WSDL. Stejně jako WSDL je založené na XML Schema. Pro popis RESTful služeb je patrně vhodnějším kandidátem, protože se přímo zaměřuje na popis webových aplikací založených na protokolu HTTP (daní za to je ovšem menší flexibilita než v případě WSDL). Obecně je jednodušší jak na pochopení, tak na zápis. WADL podporuje u vybraných elementů křížové reference na další elementy, což redukuje duplicitu (není třeba definovat již jednou definované). Reference mohou směřovat také na elementy v externích WADL dokumentech. Pak je třeba použít nikoli ID elementu, ale jeho URI. [16]

Kořenovým elementem WADL dokumentu je *application*, jehož nejdůležitější podelementy obsahuje tabulka 2 (každý ve výskytu 0..N). Popis dalších elementů lze najít v oficiálním W3C dokumentu WADL [16].

Tabulka 2 – WADL elementy. Zdroj: [16]

Element	Popis
<doc>	Slouží pro popis elementu, do kterého je zanořen. Jeho účel je tak ryze dokumentačního charakteru.
<resources>	Jedná se o kontejner zdrojů, které aplikace obsahuje. Disponuje atributem *base* definujícím kořenové URI obsažených zdrojů.
<resource>	Popisuje množinu zdrojů, z nichž každý je identifikovatelný svým jedinečným URI. Tento element má následující volitelné atributy:

	• *id* – jednoznačně identifikuje *resource* element. • *path* – obsahuje relativní URI zdroje (absolutní URI se složí s pomocí atributu *base* nadřazeného elementu *resources*). • *type* – seznam křížových referencí na *resource_type*. • *queryType* – definuje typ obsahu query komponenty[3] URI. Výchozí hodnota je *application/x-www-form-urlencoded*. Element *resource* může dále obsahovat následující volitelné potomky: • *method* – popisuje vstup a výstup metody HTTP protokolu (GET, POST, …), které lze aplikovat na daný zdroj. O to se starají vnořené elementy *request* a *response*. • *resource* – popisuje další (podřízené) zdroje v aktuálním obsažené. • *param* – má atribut *style*, který může nabývat jedné z hodnot *template*, *matrix*, *query* nebo *header*.
`<resource_type>`	Popisuje množinu metod, které společně definují chování zdroje. Disponuje atributem *id* a následujícími (volitelnými) potomky: • *param* – má jediný atribut *style*, který může nabývat hodnoty *query*, nebo *header*. • *method* – viz popis výše u `<resource>`. • *resource* – popisuje další (podřízené) zdroje tohoto typu zdroje.
`<representation>`	Popisuje formát dat zdroje. Může být vnořený do elementu *request*, *response*, případně *application* (pak se jedná o globání definici, na kterou se odkazuje). Důležité atributy jsou: • *id* – odkaz na globální definici formátu dat. • *mediaType* – typ obsahu, který je vrácen (např. XML). • *element* – pokud jsou data v XML, specifikuje kvalifikovaný (plný) název kořenového elementu (ten také popisuje dokument u *grammars*, viz níže).
`<grammars>`	Popisuje formát dat přenášených během vykonávání HTTP metod, tedy jak má vypadat odpověď, reakce na chybu apod. Obvykle se pro popis formátu využívá XML Schema, ovšem

[3] Jedná se o část mezi ? a koncem URI.

	dovoleny jsou i další alternativy, jako např. RelaxNG. Obsahuje libovolné množství vnořených elementů *include* s atributem *href*, který odkazuje na popisný dokument.

Příklad použití WADL pro popis RESTful webové služby ukazuje kód 2.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
    <doc xmlns:jersey="http://jersey.java.net/"
jersey:generatedBy="Jersey: 1.8 06/24/2011 12:17 PM"/>
    <doc title="Web Service description">
        REST web service contains 2 resources:
        1) List of all books
        2) Book search
    </doc>
    <grammars>
        <include href="/public/xsd/bookListResponse.xsd"/>
        <include href="/public/xsd/bookSearchResponse.xsd"/>
    </grammars>
    <resources base="http://libraryforpeople.appspot.com/rest/">
        <resource path="/book">
            <resource path="/search/{searchText}">
                <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
name="searchText" style="template" type="xs:string"/>
                <method id="findBooks" name="GET">
                    <response>
                        <representation mediaType="application/xml"/>
                    </response>
                </method>
            </resource>
            <resource path="/list">
                <method id="getBookList" name="GET">
                    <response>
                        <representation mediaType="application/xml"/>
                    </response>
                </method>
            </resource>
        </resource>
    </resources>
</application>
```

Kód 2 – Popis RESTful webové služby prostřednictvím WADL. Zdroj: vlastní

UDDI

UDDI neboli Universal Description, Discovery and Integration je specifikace pro centralizovaný registr sdružující informace o webových službách založený na XML. Hlavním účelem UDDI je umožnit uživatelům dotazování registru prostřednictvím SOAP zpráv (možná je také komunikace přes CORBA nebo Java RMI protokol) za účelem získání WSDL dokumentů popisujících rozhraní hledaných služeb. UDDI lze rozdělit na část představující byznys registr uchovávající metainformace o registrovaných službách

a na množinu WSDL *port typů*[4] umožňujících manipulaci a procházení tohoto registru. [12]

UDDI byznys registr se skládá z následujících komponent [12]:

- Bílé stránky – obsahují informace o poskytovateli služby (jméno, adresa, e-mail, případně další kontakt).
- Žluté stránky – obchodní zaměření poskytovatele služby (portfolio produktů poskytovatele, popis oboru apod.).
- Zelené stránky – technická data o dané službě (např. formalizovaný popis rozhraní).

UDDI se nesetkalo s příliš velkým úspěchem. Mnoho velkých firem jako IBM nebo Microsoft své veřejné UDDI uzly již zavřelo.

2.5.3 Architektura SOA

SOA neboli Service Oriented Architecture je architektura sestávající z *black box*[5] komponent propojených volnou vazbou. SOA se zaměřuje na tvorbu byznys aplikací. Příslušné byznys procesy jsou pak realizovány orchestrací SOA komponent. Nejedná se o nic jiného než o sadu metodik a doporučení, které by měla daná byznys aplikace a její SOA komponenty splňovat. [17]

Mezi základní principy SOA patří znovupoužitelnost, granularita, modularita, orientace na komponenty a interoperabilita. SOA komponenty by měly respektovat zavedené standardy. Každá komponenta či soubor komponent poskytuje nějakou službu. [17]

Jsou definovány čtyři základní typy architektur SOA [17]:

- **Architektura služeb** – jedná se o návrh konkrétní služby, který zahrnuje také všechny zdroje službou používané (typicky databáze, softwarové komponenty, XML schémata, …). Z hlediska dokumentace je nejdůležitější formalizovat kontrakt služby (co přesně se služba zavazuje poskytnout).
- **Architektura složení služeb** – jak bylo řečeno, jedním ze základních principů SOA je orientace na komponenty. Konkrétní služba se tedy může skládat z více podružných služeb s tím, že jejich implementační detaily nejsou podstatné.
- **Architektura soupisu služeb** – soupis služeb sestává ze služeb, které automatizují konkrétní byznys proces. Doporučeno je dokumentovat služby odděleně od byznys procesu pro lepší identifikaci případných technických problémů (např. výkonových).
- **Servisně orientovaná podniková architektura** – sdružuje předchozí jmenované architektury s některou z podnikových technologií pro správu dat.

---

[4] V jazyce WSDL se jedná o pojmenovanou množinu abstraktních operací společně s popisem zpráv, které využívají.

[5] Pojem *black box* (černá skříňka) značí komponentu, která skrývá své implementační detaily. Pro uživatele je důležité vědět pouze to, jaký vstup má dodat a jaký výstup je očekáván.

2.6 Přední poskytovatelé cloud computingu

Tato podkapitola je zaměřena na sumarizaci služeb předních poskytovatelů cloud computingu, a sice společností Amazon, Microsoft a Google. Informace zde uvedené vycházejí především z publikací [4], [8] a aktuální nabídky příslušného poskytovatele (v uvedených publikacích se nezřídkakdy nacházejí neaktuální údaje).

2.6.1 Amazon

Amazon je oprávněně považován za průkopníka a leadera v oblasti poskytování cloud computingových služeb. Níže je uvedeno jeho hlavní produktové portfolio.

Amazon Elastic Compute Cloud (Amazon EC2)

Jedná se o cloudovou webovou službu nabízející výpočetní kapacitu s proměnným rozsahem. Nabízí jednoduché webové rozhraní pro získání a konfigurování výpočetní kapacity. Amacon EC2 umožňuje provozovat například Microsoft Windows Server 2003, 2008, 2008R2 a 2012. Dovoluje také nasazení technologií ASP.NET, Silverlight a Internet Information Server (IIS).

Dále je možné spouštět aplikace pro operační systém Windows, ať už se jedná o webové aplikace, hostování webových služeb či aplikace provádějící náročné výpočty. Zahrnuta je také podpora databázových enginů SQL Server Express, SQL Server Standard a SQL web s účtováním na časové bázi (účtuje se každá započatá hodina, kdy je k databázi přistupováno).

Amazon Simple Storage Service (Amazon S3)

Amazon S3 představuje minimalistické úložné řešení. Umožňuje uložit a získat libovolný objem dat prostřednictvím jednoduchého webového rozhraní. Data jsou uložena ve stejné infrastruktuře, jakou Amazon využívá pro své byznys aplikace (např. obchod *amazon.com*).

Mezi funkcionalitu Amazon S3 se řadí:

- Možnost číst, zapisovat a odstraňovat datové objekty velikosti od 1 B po 5 TB. Počet objektů, které lze nahrát, není omezen.
- Každý objekt je uložen ve struktuře zvané *bucket*. Uživatel si může zvolit, v kterém regionu bude *bucket* fyzicky uložen (vhodné pro optimalizaci latence).
- Objekty uložené v daném regionu jej nikdy neopustí, dokud si uživatel explicitně nezažádá o jejich transfer do jiného regionu.
- Autentizační mechanismus s podporou nastavení práv pro jednotlivé objekty. Objekt může být nastaven jako privátní, veřejný nebo lze udělit práva pro přístup specifickým uživatelům.
- Standardizovaná REST a SOAP rozhraní pro přístup k datům.

Amazon SimpleDB

Amazon SimpleDB je flexibilní nerelační uložiště dat. Poskytuje databázové funkce pro dotazování a indexování dat. Je napojena na výše zmiňované služby Amazon EC2 a Amazon S3. Výhodou je především nízká administrativní zátěž.

Amazon CloudFront

Jedná se o webovou službu zaměřenou na poskytování statického a streamovaného obsahu. Amazon CloudFront je snadno integrovatelný s ostatními Amazon webovými službami. Lze jej samozřejmě využít i jako samostatné řešení. Cena je určena dle zvoleného programu za 1 GB přenesených dat. Samotné uložení dat v Amazon datacentrech je zdarma.

Amazon Simple Queue Service (Amazon SQS)

Amazon SQS nabízí škálovatelnou frontu pro ukládání zpráv. Zaměřuje se především na distribuované systémy. Tato služba je úzce spojená s Amazon EC2. Hlavním účelem služby je umožnit počítačům v síti snadný přístup ke čtení zpráv, jako i jejich odesílání, bez nutnosti instalace speciálního softwaru nebo konfigurace firewallu.

Elastic Block Store

Služba Elastic Block Store nabízí datové svazky, které lze kdykoli připojit k běžící Amazon EC2 instanci. Data ve svazku jsou perzistentní a přístupná ze sítě nezávisle na životním cyklu EC2 instance.

Mezi hlavní funkcionalitu Elastic Block Store patří:

- Tvorba datových svazků o velikosti 1 GB až 1 TB, které lze připojit jako zařízení do libovolné Amazon EC2 instance.
- Datový svazek se chová jako neformátované blokové zařízení s uživatelsky volitelným jménem a svým vlastním rozhraním. Uživatel má možnost nad daným svazkem vytvořit zvolený souborový systém, čímž může optimalizovat výkon EC2 pro určitý typ úlohy.
- Každý datový svazek je automaticky replikován. Data jsou tak zabezpečena proti případnému výskytu hardwarového selhání.
- Je možné vytvořit snapshot datového svazku a uložit jej v rámci služby Amazon S3. Snapshot je možné dále použít jako výchozí bod při vytváření nových datových svazků nebo jednoduše pro účely archivace.
- K dispozici je monitorování a přehledné statistiky jednotlivých datových svazků.

2.6.2 Microsoft

Hlavním produktem společnosti Microsoft v oblasti cloud computingu je sada služeb v rámci architektury Azure Service Platform. K dalším produktům této společnosti zaměřeným především na sdílení obsahu mezi uživateli se řadí především Windows Live a Office 365.

Azure Services Platform

Jedná se o platformu hostovanou v datových centrech společnosti Microsoft (evropský region obsluhují datacentra v Irsku). Azure Services Platform nabízí vlastní operační systém společně s dostatečně širokou paletou vývojových nástrojů. Mezi hlavní komponenty platformy se řadí:

- **Windows Azure** – cloudový operační systém poskytující prostředí pro vývoj, hostování a správu služeb.
- **SQL Services** – rozšiřuje možnosti databáze Microsoft SQL Server do sféry cloudu. Obsahuje řadu integrovaných funkcí pro relační dotazy, analýzu, synchronizaci apod.
- **.NET Services** – sada služeb zaměřená na vývojáře. Obsahuje komponenty využitelné pro aplikace založené na cloudu. Jak lze očekávat, je zde velká podoba s architekturou .NET Framework.
- **Live Services** – centrum pro vývojáře obsahující aktuální dokumentaci, popis API či tutoriály.

Windows Live

Windows Live představuje sadu online služeb zaměřených na podporu komunikace a sdílení dat mezi uživateli. Součástí je bezplatný software Windows Live Essentials zjednodušující vyhledávání digitálního obsahu napříč různými zařízeními.

Office 365

Office 365 je cloudová sada kancelářských aplikací. Jedná se o kombinaci produktů z rodiny Microsoft Office a dalších, jako např. Exchange Online, Sharepoint Services, Forefront nebo Lync Online. Výhodou balíku Office 365 je snadné sdílení a synchronizace dokumentů.

Exchange Online je podniková poštovní služba založená na technologii Microsoft Exchange Server. Nabízí jak webového klienta, tak i podporu Outlooku. Kromě nástrojů pro prací s poštou nabízí také sdílený kalendář, úkolovník, kontakty apod.

Sharepoint Services představuje sadu nástrojů, jejichž cílem je usnadnit spolupráci mezi uživateli v týmu. Tento produkt sestává z webových komponent založených na technologii ASP.NET figurujících jako doplňky pro webové aplikace. Tyto komponenty se pak konfigurují tak, aby cílový web mohl sloužit jako pracovní základna pro příslušný tým. Členové týmu se mohou podílet na diskuzích, spolupracovat na dokumentech.

Forefront je integrovanou sadou produktů pro ochranu a správu identit a také řízení přístupu. Řízení identit a přístupu k nim je postaveno nad infrastrukturou Active Directory[6]. V rámci Forefront (Forefront Identity Manager 2010, Forefront Unified Access Gateway 2010) se využívají zejména technologie SSL, VPN a DirectAccess.

[6] Jde o implementaci adresářových služeb LDAP společnosti Microsoft.

Lync Online je další službou zaměřenou na kolaboraci uživatelů, která jim může napomoci v jejich běžných pracovních aktivitách. Lze jej zakoupit buď v rámci sady Office 365 nebo samostatně. Hlavními funkcemi Lync Online jsou hlasová komunikace, video konference, zasílání zpráv ostatním uživatelům, možnost využít virtuální tabule pro prezentace a další.

2.6.3 Google

Nabídka společnosti Google pro cloud sestává z trojice Google Apps, Google App Engine a Google Marketplace. Google App Engine se zabývá samostatně kapitola 4, proto jsou zde stručně popsány pouze zbývající dva produkty.

Google Apps

Stejně jako Office 365 je i Google Apps cloudová sada kancelářských aplikací s bezplatnou zkušební verzí zaměřená nejen na podnikatelské subjekty (donedávna byla k dispozici i verze zdarma pro vlastní doménu, tato nabídka však byla ke konci roku 2012 zrušena[7]). Vyniká hlavně jednoduchou konfigurací a správou. [18]

Portfolio služeb Google Apps je rozmanité od webového e-mailového klienta Gmail přes kalendář až po Google Hangouts. Služby jsou rozděleny do balíčků podle typu koncového zákazníka na [18]:

- Google Apps pro firmy,
- Google Apps pro vzdělávání,
- Google Apps pro vládu.

Google Marketplace

Google Marketplace nabízí aplikace třetích stran, které jsou díky Google Apps API snadno integrovatelné do původních Google aplikací. Stejně tak mohou přistupovat ke stejným datům. Nabídka Google Marketplace je široká od aplikací pro účetnictví přes marketingové nástroje až po řízení projektů.

2.7 Nevýhody cloud computingu

Přes zřejmý přínos cloud computingu v různých oblastech IT je na místě sumarizovat i jeho hlavní nevýhody [4], [8]:

- Oproti standardnímu řešení v podobě vlastní IT infrastruktury či vlastního vývoje softwaru může cloud trpět **nedostatkem customizace**.
- Vzhledem k síťové povaze cloudu je nutné počítat s **vyšší latencí** při zpracovávání požadavků. Cloudové řešení nemusí být vhodné také v případě, kdy je nutné přenášet velké množství dat.
- Stejně jako v celé síti Internet je cloudová komunikace **bezstavová**. V závislosti na požadavku na synchronizaci je pak nutné využívat middleware v podobě

[7] Zdroj: http://www.zive.cz/bleskovky/google-apps-zdarma-konci-za-gmail-na-vlastni-domene-se-nove-plati/sc-4-a-166685/default.aspx

transakčních manažerů, service brokerů apod. Tato skutečnost může znamenat významný výkonnostní pokles.

- Cloudové řešení obecně ještě není dostatečně vyspělé, proto se lze ojediněle setkat s **nespolehlivostí** či dočasnou **nedostupností** služeb.
- Posledním a často nejzávažnějším argumentem proti cloud computingu je **hrozba bezpečnostního rizika**. Data procházejí či jsou uložena v systému, nad kterým jejich vlastník nemá plnou kontrolu. Zapezpečení dat v cloudu se obvykle řeší kombinací těchto metod:
    - Šifrování – data jsou v cloudu ukládána v šifrované podobě.
    - Autentizační procesy – pro přístup k datům musí uživatel zadat své uživatelské jméno a heslo.
    - Autorizační postupy – přístup k datům se obvykle dělí dle autorizační hierarchie, kdy např. níže postavený zaměstnanec může přistupovat pouze k omezenému segmentu dat.

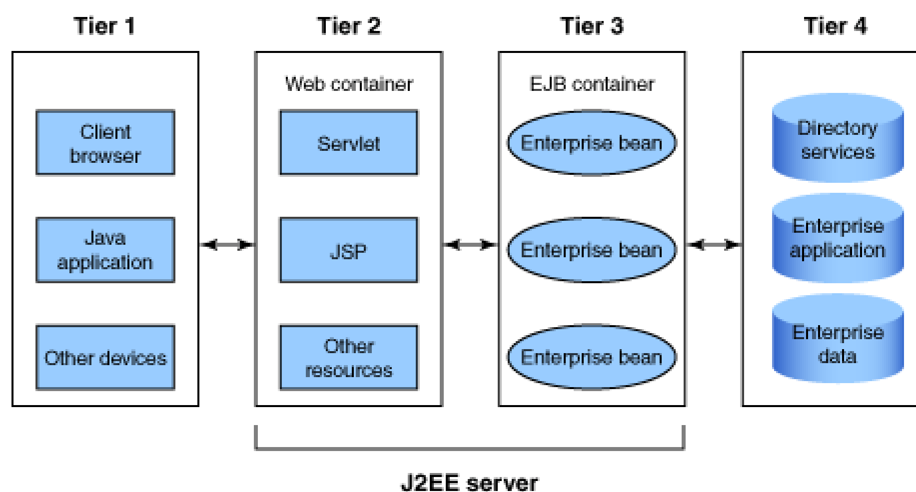
3 Java 2 Enterprise Edition (J2EE)

Tato kapitola je zaměřena na popis platformy Java 2 Enterprise Edition (J2EE) a jejich vybraných aspektů. Jelikož pod J2EE spadá celá řada technologií, kapitola se omezí pouze na popis těch, které jsou relevantní z hlediska vypracování praktické části diplomové práce. Větší část kapitoly je věnována technologii Java Data Objects (JDO).

3.1 Úvod do J2EE

Jedná se o otevřenou platformu postavenou na dobře zavedených standardech. Její využití je velmi široké, od vývoje webových aplikací až po budování komplexních podnikových řešení založených na komponentové architektuře. [19]

J2EE využívá čtyřvrstvou architekturu sestávající z **klientské**, **webové**, **EJB** a **databázové** vrstvy. Každá vrstva poskytuje aplikaci odlišnou funkcionalitu. API většiny J2EE komponent jsou typicky svázána s jednou konkrétní vrstvou (výjimkou tvoří např. Java XML API). Ilustrační schéma J2EE architektury zachycuje obrázek 6. [19]



Obrázek 6 – Schéma J2EE architektury. Zdroj: [20]

J2EE je založena na kontejnerech. Kontejner operuje vždy na jedné z výše jmenovaných vrstev a tvoří prostředí pro komponenty. Komponentu platformy J2EE si lze zjednodušeně představit jako balíček Java tříd, které poskytují určitou funkcionalitu. [19]

3.2 Servlety

Servlet je komponenta, která rozšiřuje možnosti webového serveru. Jedná se o Java třídu implementující rozhraní Java Servlet Api, díky kterému může servlet odpovídat na klientské požadavky. Nejčastěji se v případě servletů uvažují požadavky přicházející přes protokol HTTP nebo HTTPS. Využít lze však kterýkoli jiný protokol založený na architektuře server-klient. [19]

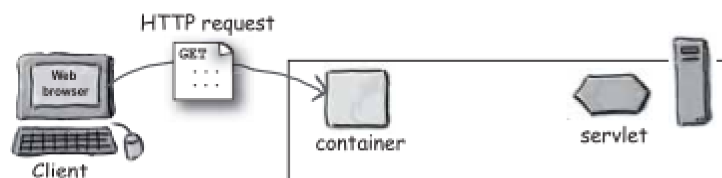
Servlety jsou spravovány servletovým kontejnerem (také znám jako webový kontejner), který zajišťuje následující služby [19]:

- **Řízení životního cyklu** – kontejner řídí životní cyklus servletu. Provádí jeho inicializaci a spravuje garbage collecting.
- **Komunikační podporu** – kontejner zprostředkovává komunikaci servletu s webovým serverem. Vývojář tedy nemusí programovat sockety, datové proudy apod.
- **Vícevláknová podporu** – kontejner automaticky vytvoří nové vlákno servletu pro obsloužení příchozího požadavku a taktéž zajišťuje synchronizaci vláken.
- **Zajišťuje podporu JSP**.

Vyřízení HTTP požadavku servlet kontejnerem

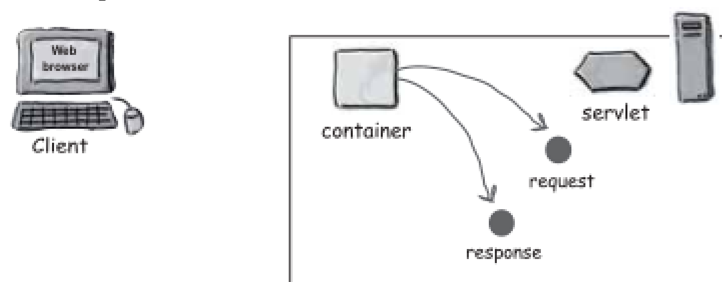
Hlavním úkolem servletu je vyřizovat klientské požadavky. Pro lepší pochopení fungování kontejneru a servletů je třeba vysvělit, jakým způsobem je na klientský požadavek reagováno [19]:

1. Klient odešle požadavek s daným URL servletovému kontejneru (viz obrázek 7).



Obrázek 7 – Zpracování HTTP požadavku, krok 1. Zdroj: [19]

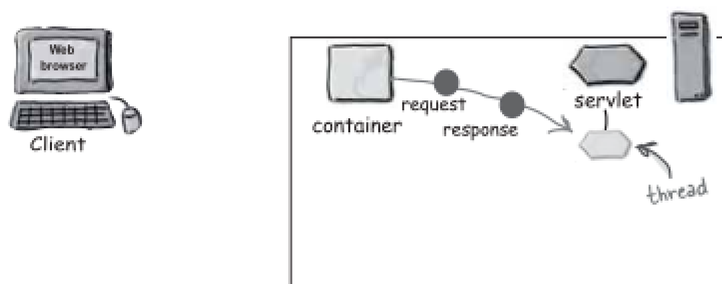
2. Kontejner zachytí požadavek a vytvoří dvojici objektů typu *HttpServletRequest* a *HttpServletResponse* (viz obrázek 8).



Obrázek 8– Zpracování HTTP požadavku, krok 2. Zdroj: [19]

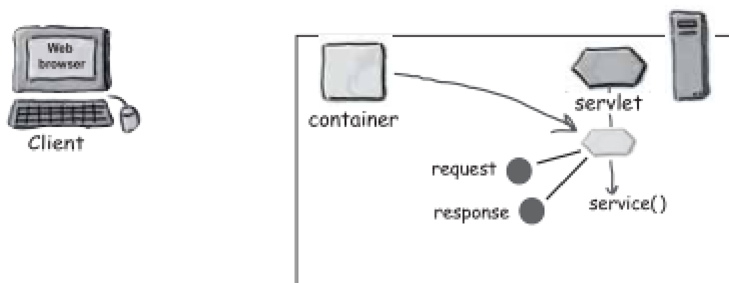
3. Kontejner na bázi porovnání URL se záznamy v deskriptoru *web.xml* zjistí, který servlet má požadavek obsloužit. Pro požadavek alokuje samostatné vlákno. Je zavolána servletová metoda *init()* a vláknu jsou předány výše zmíněné objekty

(viz obrázek 9).



Obrázek 9 – Zpracování HTTP požadavku, krok 2. Zdroj: [19]

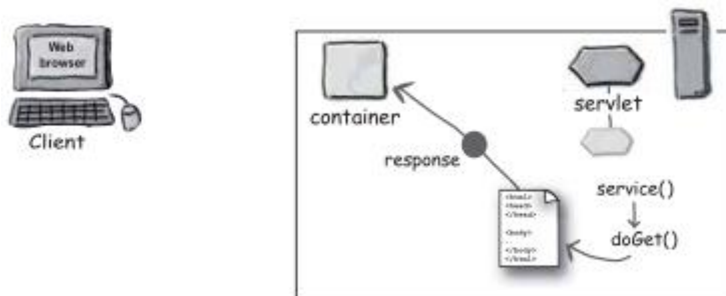
4. Kontejner pro obsluhující vlákno spustí metodu *service*, která ve svém těle v závislosti na použité metodě HTTP(S) protokolu zavolá buď metodu *doGet()*, *doPost(), doPut() nebo doDelete()* (viz obrázek 10).



5.

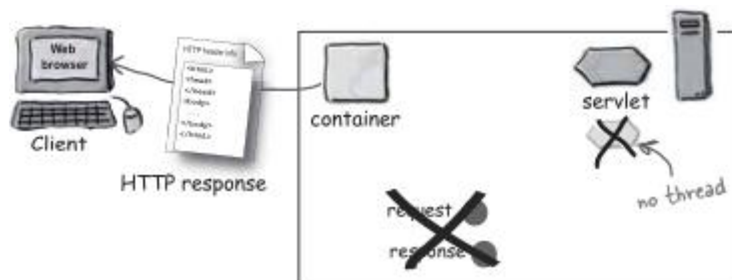
Obrázek 10 – Zpracování HTTP požadavku, krok 2. Zdroj: [19]

6. Vybraná metoda vygeneruje obsah pro frontendovou část aplikace a zabalí jej do objektu *HttpServletResponse* (viz obrázek 11).



Obrázek 11– Zpracování HTTP požadavku, krok 2. Zdroj: [19]

7. Vlákno dokončí svou činnost - zkonvertuje objekt *HttpServletResponse* na vlastní odpověď HTTP protokolu a tu pošle zpět klientovi. Následně odstraní oba vytvořené objekty (viz obrázek 12).



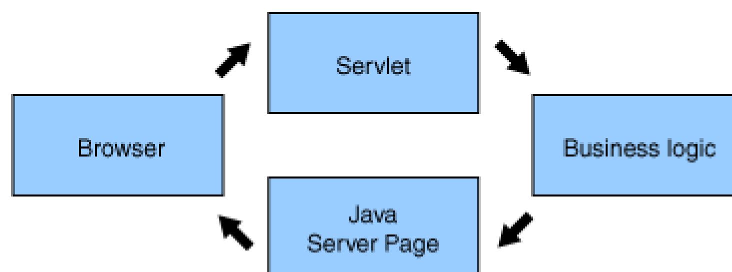
Obrázek 12 – Zpracování HTTP požadavku, krok 2. Zdroj: [19]

3.3 Java Servlet Pages (JSP)

JSP lze s trochou nadsázky označit za syntaktickou zkratku pro kód servletu. Pokud totiž klient přistupuje k JSP stránce, provede se následující posloupnost kroků [19]:

1. Kontejner detekuje požadavak na JSP stránku.
2. Kontejner provede překlad JSP stránky do zdrojového kódu servletu.
3. Zdrojový kód servletu je zkompilován.
4. Dále již vše probíhá způsobem popsaným v předchozí kapitole o servletech.

Je-li v návrhu J2EE aplikace užit architektonický vzor MVC (Model View Controller), pak bude JSP představovat pohledovou část. Komunikace typicky probíhá podle schématu na obrázku 13.



Obrázek 13 – Komunikace při užití architektonického vzoru MVC n platformě J2EE. Zdroj: [20]

Z obrázku 13 je zřejmé, že aplikační logika se do JSP nezapisuje přímo, byť je to možné prostřednictvím skriptletů. Typické je pro vývoj v J2EE použití aplikačních rámců a šablonovacích systémů. JSP je tak v podstatě odstaveno od jakéhokoli formátování výstupu. O všechno se starají příslušné tagy zvoleného aplikačního rámce. Standardní řešení JSP pro formátování výstupu, jako např. tagy knihovny JSTL (JSP Standard Tag Library), se pak stává zbytečným. [19]

3.4 Perzistence dat

Pro přístup k perzistovaným datům a jejich ukládání do databáze lze na platformě J2EE využít hned několik řešení. Představeny jsou specifikace Java Data Objects (JDO) a Java

Persistence Api (JPA). Tyto jsou pak implementovány konkrétními technologiemi. Větší prostor je v kapitole věnován standardu JDO z důvodu jeho nasazení ve vývoji SVK.

3.4.1 Java Data Objects (JDO)

Java Data Objects (JDO) je standard pro abstraktní perzistenci dat. Implementují jej technologie jako DataNucleus, JPOX nebo Xcalia. JDO využívá k reprezentaci perzistovaných dat POJO (Plain Old Java Object). Aby mohly být objekty perzistovány, je nejprve nutné definovat, které třídy budou perzistované (neboli které třídy se stanou entitními). Perzistovanost tříd je určena jejich metadaty. V JDO existují tři možnosti, jak metadata k POJO přidružit [21]:

- XML konfigurace,
- anotace,
- Metadata API.

XML konfigurace se používala u starších verzí JDO (každá POJO třída byla zároveň popsána svým XML dokumentem), než byl v JDK 1.5 představen koncept anotací. Anotace lze vkládat k jednotlivým atributům, metodám a třídám samotným. Tabulka 3 obsahuje stručný přehled nejpoužívanějších anotací JDO v podání implementace DataNucleus, kterou využívá Google Application Engine (GAE) popsaný v kapitole 4. Anotace pro objektově relační mapování nejsou uvedeny (tuto techniku prakticka část diplomové práce nevyužívá, protože datové uložiště GAE není postaveno na relační databázi). [21], [22]

Tabulka 3 – Vybrané JDO anotace. Zdroj: [22]

Anotace	Přidružitelná k	Význam
`@PersistentCapable`	třídě	Definuje, zda je třída perzistovaná.
`@Persistent`	atributu/metodě	Definuje, zda jsou atribut nebo metoda perzistované.
`@NotPersistent`	atributu/metodě	Anotovaný atribut či metodu nelze perzistovat. Toto explicitní označení je pro složky, které si vývojář nepřeje perzistovat, povinné.
`@Query`	třídě	Přiřadí pojmenovaný dotaz k POJO třídě. Název pak slouží jako syntaktická zkratka dotazu. Případně lze definovat více pojmenovaných dotazů najednou prostřednictvím anotace `@Queries`.
`@Serialized`	atributu/metodě	Definuje, zda lze danou složku perzistovat.
`@PrimaryKey`	atributu/metodě	Definuje, zda je daná složka primárním klíčem nebo jeho částí v případě kompozitního primárního klíče.

`@Key`	atributu/metodě	Definuje informace o klíčích v případě, že atribut představuje mapu (tabulku), případně metoda mapu (tabulku) vrací.
`@Value`	atributu/metodě	Definuje informace o hodnotách v případě, že atribut představuje mapu (tabulku), případně metoda mapu (tabulku) vrací.

JDO implementace typicky využívají *obohacení byte kódu*. Samotná anotace totiž představuje pouhá metadata, která sama o sobě perzistovanost POJO nezaručí. K dané POJO třídě je třeba na základě metadat přidružit aplikační kód, který umožní spolupráci s datovým uložištěm a nestatické[8] detekování objektů podle typu. Obohacení byte kódu je krok, který alteruje byte kód zkompilované POJO třídy na základě jejích anotací tak, aby výše uvedenou funkcionalitu obsahovala. Jedná se tedy o jakýsi post-kompilační proces. Alternativou k tomuto přístupu je tzv. *reflexe*, kterou uplatňují převážně některé implementace Java Perstistence Api (viz dále v kapitole). Reflexe umožňuje analyzovat třídu za běhu a volat její metody. Dodatečný kód je zapsán ve formě proxy třídy nebo potřídy, která je po kompilaci *Classloaderem* zavedena do aplikace. Výhoda obohacení byte kódu tedy spočívá v zajištění perzistovanosti POJO tříd v době překladu, což se ve srovnání s reflexí projeví nižší režii. [21], [22]

Posledním nástrojem pro přidružení metadat je Metadata API umožňující definovat perzistované třídy za běhu aplikace. Tuto vlastnost lze s výhodou uplatnit v případě, že aplikace využívá *Classloader*, který jí umožní dynamicky zavádět třídy do běžící aplikace. Další variantou je nasazení pokročilých technologií pro manipulaci byte kódu, jako např. BCEL nebo ASM[9] (používá DataNucleus). [21], [22]

Entitní třída

Entitní třída je POJO třídou opatřenou metadaty nejčastěji v podobě anotací. Ukázka entitní třídy *Kniha* bez metod pro JDO implementaci DataNucleus je obsažena v kódu 3. [22]

```
@PersistenceCapable
@XmlRootElement(name = "book")
@XmlType(name = "bookType", propOrder = { "nazev" ,"autor", "popis",
"ISBN", "jazykCode", "nakladatelstvi", "formattedDatumVytvoreni" })
public class Kniha extends Produkt {

    @Persistent
    private String autor;
    @Persistent
    private Integer rokVydani;
    @Persistent
    private String Nakladatelstvi;
    @Persistent(nullValue = NullValue.EXCEPTION)
```

[8] Za běhu aplikace.

[9] Zkratka ASM je referencí ke klíčovému slovu *__asm__* v jazyku C, díky kterému lze psát kód pro assembler v běžném C programu. Sama o sobě nic neznamená.

```
    private List<Zanr> zanrs;
    @Persistent(serialized = "true", defaultFetchGroup = "true")
    private Soubor fotka;
    @Persistent
    private String ISBN;
    @Persistent
    Key jazyk;
    @NotPersistent
    private Jazyk jazykEntity;
    @NotPersistent
    private JazykTranslation jazykTranslation;
    @NotPersistent
    private Uzivatel vlastnikEntity;
    @NotPersistent
    private List<Vypujcka> registrovane = new ArrayList<Vypujcka>();
    @NotPersistent
    private List<Vypujcka> vypujcene = new ArrayList<Vypujcka>();
...
}
```

Kód 3 – Ukázka entitní třídy Kniha v JDO DataNucleus. Zdroj: vlastní

PersistenceManager

Stěžejním prostředkem v JDO je objekt typu *PersistenceManager*, pomocí kterého lze manipulovat s perzistovanými objekty. Tento lze získat zavoláním metody *getPersistenceManager()* z instance třídy *PersistenceManagerFactory*, která reprezentuje spojení s databází. Režijně se tedy jedná o velmi „drahý“ objekt. Každá aplikace vyžadující připojení k databázi musí disponovat alespoň jedním objektem typu *PersistenceManagerFactory* (více, pokud je napojena na několik databází). [21]

PersistenceManagerFactory je v rámci aplikace indentifikován unikátním názvem a konfigurován nejčastěji prostřednictvím XML souboru (lze však využít i tzv. *properties* některého z aplikačních rámců). Kód 4 zachycuje XML konfiguraci *PersistenceManagerFactory* s názvem *PMFSVK* pro JDO DataNucleus na platformě GAE. [21], [23]

```
<?xml version="1.0" encoding="utf-8"?>
<jdoconfig xmlns="http://java.sun.com/xml/ns/jdo/jdoconfig"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://java.sun.com/xml/ns/jdo/jdoconfig">

<persistence-manager-factory name="PMFSVK">
   <property name="javax.jdo.PersistenceManagerFactoryClass"
value="org.datanucleus.store.appengine.jdo.DatastoreJDOPersistenceManager
Factory"/>
   <property name="javax.jdo.option.ConnectionURL" value="appengine"/>
   <property name="javax.jdo.option.NontransactionalRead" value="true"/>
   <property name="javax.jdo.option.NontransactionalWrite" value="true"/>
   <property name="javax.jdo.option.RetainValues" value="true"/>
   <property name="datanucleus.appengine.autoCreateDatastoreTxns"
        value="true"/>
   <property name="datanucleus.appengine.datastoreEnableXGTransactions"
        value="true"/>
   <property name="datanucleus.CopyOnAttach" value="false"/>
    </persistence-manager-factory>
```

```
</jdoconfig>
```

Kód 4 – XML Konfigurace PersistenceManagerFactory v JDO DataNucleus. Zdroj: vlastní

Object *PersistenceManagerFactory* lze pak získat přes statickou metodu *getPersistenceManagerFactory()* třídy *JDOHelper*, jak ukazuje kód 5. Vhodné je použít návrhový vzor *Factory* implementovaný pomocí *Singletonu*. Po ukončení práce s perzistovanými objekty je *PersistenceManager* nutné zavřít voláním metody *close()*. Automaticky pak dojde k zápisu do databáze. [21]

```
...
private static PersistenceManagerFactory persistenceManagerFactory;
...
public static synchronized PersistenceManagerFactory getInstance() {
        if (persistenceManagerFactory == null){
            persistenceManagerFactory =
                JDOHelper.getPersistenceManagerFactory("PMFSVK");
        }
        return persistenceManagerFactory;
}
...
```

Kód 5 – Získání objektu typu PersistenceManagerFactory. Zdroj: vlastní

V momentě, kdy je zajištěno zprostředkování *PersistenceManager* objektů, má vývojář k dispozici nástroj pro získávání, perzistování, editování a mazání objektů z databáze.

Transakce

Při inicializaci objektu typu *PersistenceManager* se implicitně vytvoří transakce. Aktuální transakci lze získat metodou *currentTransaction()* a začít voláním *begin()*. Konkrétní transakce je v každém okamžiku spojena pouze s jedním aktivním *PersistenceManagerem*. Dojde-li k potvrzení transakce metodou *commit*(), veškeré změny učiněné v kontextu perzistence se uloží do databáze. Vyskytne-li se během provádění transakce chyba, lze standardně provést *rollback*. Většina metod využívajících JDO bude mít kostru dle kódu 6. [21], [22]

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction transaction = pm.currentTransaction();

try
{
    transaction.begin(); // Začátek transakce
    // Operace s perzistovanými objekty
    transaction.commit(); // Potvrzení transakce
}
finally
{
    if (transaction != null && transaction.isActive())
    {
        transaction.rollback(); // V případě výskytu chyby návrat
    }
}
```

Kód 6 – Transakce v JDO. Zdroj: vlastní

JDO podporuje čtyři úrovně izolace transakcí tak, jak jsou známy z relačních databází (viz tabulka 4). Úroveň izolace lze nastavit pro jednotlivé transakce metodou *setIsolationLevel*(). Globální nastavení lze provést v XML konfiguraci pro konkrétní *PersistenceManagerFactory*. [21], [22]

Tabulka 4 – Urovně izolace transakcí. Zdroj: [21]

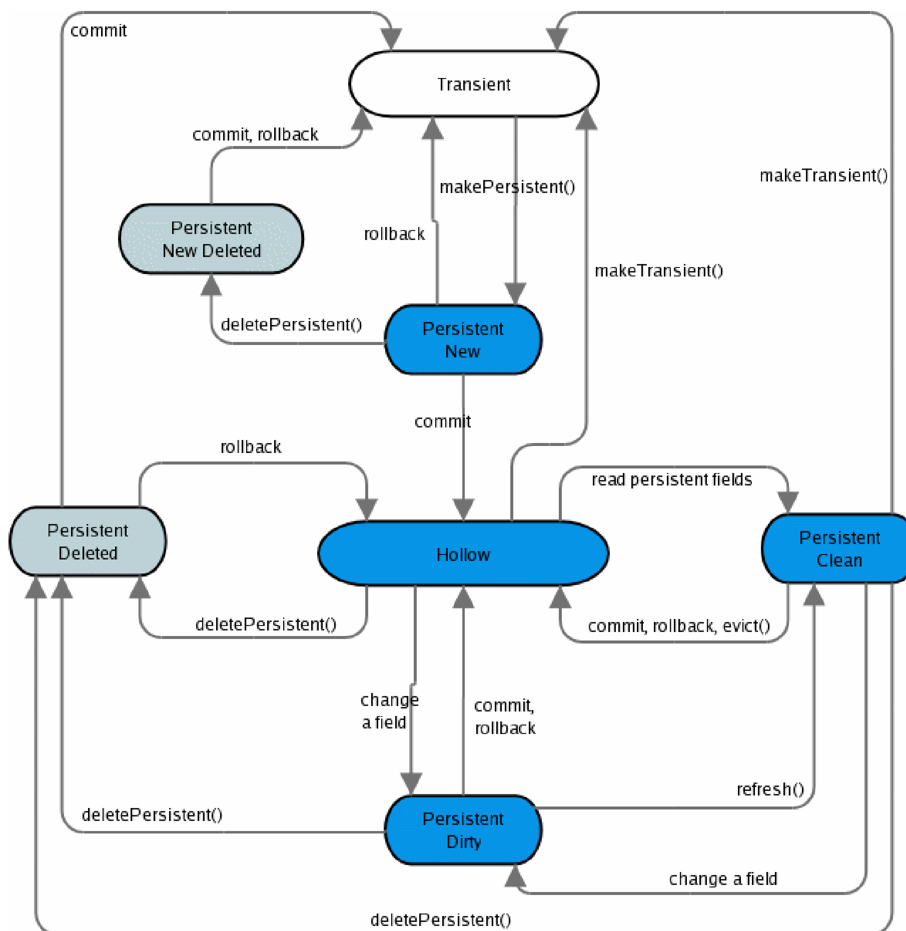
Úroveň izolace	**Vysvětlení**
Read uncommited	Transakce může vidět nepotvrzené změny jiných transakcí. Tento problém je znám jako *dirty reads*.
Read commited	Je zaručeno, že transakce čte pouze potvrzené záznamy. Chrání tedy proti *dirty reads*. *Fuzzy reads* [10] a *phantom reads* [11] jsou stále problémem.
Repeatable reads	Přidává navíc zámek pro zápis. *Fuzzy reads* je tedy tímto vyřešeno. Stále však hrozí *phantom reads*.
Serializable	K záznamům může přistupovat pouze jedna transakce. Řeší všechny uvedené problémy s transakcemi za cenu snížení propustnosti databáze.

Stavy objektů v JDO

Než bude přikročeno k vysvětlení jednotlivých operací nad perzistovanými objekty, je vhodné zmínit hlavní stavy (viz obrázek 14), ve kterých se může objekt v JDO nacházet. JDO spravuje životní cyklus perzistovaných objektů od jejich vytvoření, zařazení do kontextu perzistence až po uložení do databáze. Přechody mezi stavy se realizují voláním metod *PersistenceManageru*. Následující výklad neuvažuje možnost netransakčního zpracování, které je nedoporučováno z důvodu možnosti narušení konzistence dat. Netransakční zpracování nalézá uplatnění pouze ve specifických typech aplikací. [21], [22]

[10] Transakce znovu čte jednou přečtená data a shledává, že byla změněna nebo smazána jinou transakcí.

[11] Transakce znovu spouští dotaz s vyhledávacím kritériem a shledává, že jsou vráceny další záznamy vyhovující danému kritériu, které přidala jiná potvrzená transakce.



Obrázek 14 – Přechody mezi hlavními stavy objektů v JDO. Zdroj: [21]

JDO od verze 2 navíc zavádí koncept *attach*/*detach*, který umožňuje vyjmout objekt z kontextu perzistence a posléze jej do něj zase vrátit. S výhodou ho lze využít tehdy, je-li třeba s objektem v aplikaci pracovat a není žádoucí hodnoty jeho průběžně měnících datových složek ukládat do databáze při každém potvrzení transakce. V prvních verzích JDO bylo nutné vytvořit hlubokou kopii a hodnoty složek poté nakopírovat zpátky do původního objektu. Aby mohl být objekt dané entitní třídy vyjmutelný z kontextu perzistence (*detachable*), je třeba ke třídě přidat odpovídající atribut v podobě `@PersistenceCapable(detachable="true")`. [21], [22]

Výchozím stavem objektu je *Transient*. V tomto stavu se nachází objekt po jeho vzniku nebo v případě navrácení transakce (*rollback*). Takový objekt ještě nemá přiřazenou **JDO identitu**. JDO identita zajišťuje, že objekt, který se nachází v kontextu perzistence, je jednoznačně odlišitelný od ostatních objektů. Identitu může spravovat buď samotná implementace JDO (pak se hovoří o tzv. *datastore identity*), anebo přímo vývojář (*application identity*). Obvykle se používá *datastore identity*. V případě, že jsou na generování identity kladeny zvláštní požadavky, je vhodné využít druhou nabízenou možnost. [21], [22]

Objekt je zařazen do kontextu perzistence zavoláním metody *makePersistent()*. Následně je mu přiřazena JDO identita a zároveň je uložen do databáze. Jedná se o stav

Persistent New. Pokud se objekt již v kontextu perzistence nachází, pak metoda *makePersistent()* provede aktualizaci jeho databázového obrazu. [21], [22]

Do stavu *Hollow* se objekt dostává po potvrzení aktuální transakce. Objektu jsou nastaveny perzistentní datové složky na výchozí hodnoty kromě těch reprezentujících klíč. JDO identita objektu je zachována. Pokud by mělo dojít ke čtení perzistentní datové složky *Hollow* objektu, provede se aktualizace hodnot z databáze a objekt přechází do stavu *Persistent Clean*. Naopak, uvažuje-li se zápis do perzistentní složky, proběhne přechod do stavu *Persistent Dirty*. [21], [22]

Perzistování objektu

Jak již bylo řečeno, perzistování objektu se provádí metodou *makePersistent()* třídy *PersistenceManager*. Do databáze se uloží celý objektový graf. Není tedy nutné tuto metodu volat pro každý referencovaný objekt zvlášť. Použití je velmi jednoduché (viz kód 7).

```
pm.makePersistent(kniha);
```

Kód 7 – Perzistence objektu typu Kniha. Zdroj: vlastní

Objekt (zároveň s objekty, které referencuje) lze z kontextu perzistence vyjmout voláním metody *detachCopy()* třídy *PersistenceManager*. Pro zpětné zařazení do kontextu perzistence (*attach*) stačí zavolat známou metodu *makePersistent()*. V implementaci DataNucleus lze navíc v konfiguraci *PersistenceManagerFactory* nastavit booleovský atribut *datanucleus.DetachAllOnCommit*. Tento zajistí, že všechny objekty budou po potvrzení transakce automaticky vyjmuty z kontextu perzistence. [21], [22]

Může existovat podezření, že se databázový obraz perzistovaného objektu změnil (do databáze například přistupuje jiná aplikace). Pro tento případ nabízí *PersistenceManager* metodu *refresh()*, která vynutí aktualizaci datových složek objektu hodnotami z databáze. [21]

Získání perzistovaného objektu

JDO poskytuje několik možností pro získání perzistovaného objektu z databáze [21]:

- na základě znalosti JDO identity,
- *extentem*,
- vykonáním dotazu.

JDO získává objekty transparentně pokaždé, když je použita reference na ně. Pokud tedy aktuální perzistovaný objekt obsahuje reference na jiné objekty, jsou tyto přečteny z databáze až v momentě jejich zavolání. Jedná se tedy v podstatě o druh *lazy loadingu*, který je v prostředí JDO nazýván *transparent navigation*. Za zmínku stojí, že klasická serializace oproti tomu načítá do paměti celý objektový graf. JDO řešení je tedy úspornější z hlediska obsazení paměti i nižší dodatečné režie pro prvotní získání objektu. [21], [22]

Pro získání objektu prostřednictvím JDO identity lze využít metodu *getObjectById()* třídy *PersistenceManager*, které se jako argument předá identita hledaného objektu. Identita však nemusí být vždy známa, a tak se používá spíše přetížená verze s parametry entitní třídy a primárního klíče objektu. Fragment kódu 8 ukazuje získání objektu typu *Jazyk*. [21]

```
Jazyk jazykLocal =
        pm.getObjectById(Jazyk.class, jazykKey);
```

Kód 8 – Získání objektu typu jazyk prostřednictvím metody getObjectById. Zdroj: vlastní

Extentem se rozumí kolekce všech objektů dané entitní třídy. Pro jeho obdržení je třeba zavolat metodu *getExtent()* třídy *PersistenceManager*. První parametr metody je třída, jejíž objekty se mají získat. Druhý parametr představuje booleovskou hodnotu určující, zda má výsledek zahrnout také všechny potřídy hledané entitní třídy. Vrácena je kolekce implementující rozhraní *Extent*, které dědí z rozhraní *Iterable*. Přes objekty v *extentu* tedy lze iterovat. Po ukončení práce s *extentem* je nutné tento uzavřít voláním metody *closeAll()*. Ukázka použití *extentu* je ke zhlednutí v kódu 9. [21], [22], [23]

```
Extent extent = pm.getExtent(Book.class, true);
for (Book b : extent) {
  // ...
}
extent.closeAll();
```

Kód 9 – Získání všech objektů typu Kniha s použitím extentu a následnou iterací. Zdroj: vlastní

JDO definuje dotazovací jazyk JDOQL, pomocí kterého lze provádět kriteriální výběr objektů z databáze. JDOQL je v základě podobné SQL, jen je přizpůsobené objektově orientovanému přístupu. Umožňuje kromě jiného i typovou kontrolu parametrů a výsledku. Konkrétní dotaz lze zapsat jako řetězec podléhající dotazovací syntaxi, anebo je možné využít objektu typu *Query*. Více o problematice dotazování v JDO pojednává kapitola 4.3.2. [21], [22]

Editování objektu

Existují dvě možnosti, jak lze v JDO přistupovat k editaci perzistovaných objektů. První je editace objektu, který byl získán v rámci aktuálního *PersistenceManageru*. Změny se projeví ihned po zavolání metody *close()*. [21]

Druhou variantou je vyjmutí objektu z kontextu perzistence voláním *detachCopy()* (entitní třída musí být *detachable*, jak bylo vysvětleno výše). Po editaci datových složek lze změny uložit zavoláním metody *makePersistent()* libovolného *PersistenceManageru*. [21]

Smazání objektu

Smazání objektu z databáze zajišťuje metoda *deletePersistent()* třídy *PersistenceManager*. Je-li třeba smazat více objektů, je vhodnější zavolat metodu *deletePersistentAll()*, které se předá kolekce objektů. Stejně jako u perzistování i zde se uvažuje celý objektový graf. [21]

3.4.2 Java Persistence Api (JPA)

Na rozdíl od JDO je JPA standardem zaměřeným výhradně na spolupráci s relačními databázemi. Realizuje objektově relační mapování. Mezi známé implementace JPA se řadí OpenJPA, EclipseLink, Oracle Toplink nebo s výhradami Hibernate. Použití nerelační databáze je možné pouze s výraznými změnami specifikace převážně v oblasti implementace dotazování. Na trhu se aktuálně nachází jen minimum JPA řešení pro nerelační databáze. Za všechny lze jmenovat např. ObjectDb nebo řešení společnosti Datanucleus pro platformu GAE.

Princip práce s JPA je v základě podobný jako u JDO. *PersisitenceManager* a *PeristenceManagerFactory* jsou nahrazeny třídami *EntityManager* a *EntityManagerFactory*. Metadata entitních tříd jsou opět vyjádřena anotacemi apod.

4 Google App Engine (GAE)

Tato kapitola představuje teoretický základ pro praktickou část diplomové práce, a sice vytvoření systému veřejné knihovny (viz kapitola 5). Zaměřuje se na technické aspekty GAE, jeho architekturu a důležité součásti. Veškeré příklady zde uvedené předpokládají využití GAE pro vývoj a hostování Java aplikace.

4.1 Představení GAE

Google App Engine (GAE) je cloud computingová platforma pro hostování webových aplikací společnosti Google. Taxonomicky se jedná o servisní model PaaS. Od typického webového aplikačního serveru se jeho struktura liší hned v několika aspektech. Aplikace vytvořené na GAE nemají přístup k fyzické infrastruktuře. Nemohou tedy otevírat sockety nebo spouštět procesy na pozadí (vývojář však pro tento účel může standardně použít *backend* [12]nebo *cron*[13]). Aplikace vyvinutá v GAE může využívat některé z Google služeb, jako např. URL Fetch[14]. [24], [25]

Obchodní model GAE je příznivý pro hostování aplikací s nízkým trafficem. Platí se totiž za používáné prostředky (využití CPU, datového uložiště, příchozí a odchozí datový traffic) s tím, že do určité míry je hostování zdarma. Google odhaduje, že pro nenáročné aplikace se lze dostat až na 5 milionů impresí za rok bez toho, aniž by klient musel za cokoli platit. [24], [25]

4.2 Architektura GAE

Architektura GAE je stejně jako J2EE založena na kontejnerech. V kontextu GAE se ovšem touto skutečností rozumí použití vyšší úrovně abstrakce. [24], [25]

GAE si lze představit jako kontejner a aplikaci pro něj vyvinutou jako komponentu. Když uživatel přistupuje k dané aplikaci, GAE ji automaticky nahraje a obslouží jeho požadavek. V případě, že po určitou dobu k aplikaci nepřistupují žádní uživatelé, GAE tuto uloží ke spánku, čímž šetří cenné cykly procesoru. Stejně tak je pak aplikace probuzena v případě příchodu nového požadavku. Probuzení instance obvykle trvá kolem deseti sekund. S toutou prodlevou je třeba počítat hlavně u aplikací s velmi nízkým trafficem, kde by mohla potenciální návštěvníky odradit (hrozí, že netrpělivý uživatel zavře aplikaci ještě před probuzením instance). [24], [25]

---

[12] *Backend* umožňuje spouštět výpočetně náročné úlohy na pozadí. Na rozdíl od *cronu* a běžných požadavků není provádění úloh časově limitováno.

[13] Úkoly pro cron lze pro Javu definovat v souboru cron.xml, pro Python a Go pak v souboru cron.yaml (používá se alternativní formát k XML – YAML, který je podle jeho tvůrců přívětivější pro čtení člověkem). Cron je na platformě GAE významně omezen skutečností, že veškeré úlohy musejí být ukončeny do 60 sekund (stejně jako běžné požadavky).

[14] GAE aplikace tuto službu může využít pro odeslání HTTP(S) požadavku jiné vzdálené aplikaci a obdržet odpověď. Vzhledem k nemožnosti naprogramování vlastního řešení pro otevírání socketů na GAE je služba URL Fetch nutností v případě požadavku na komunikaci s externími aplikacemi.

Výhodou architektury GAE je především snadná škálovatelnost. V případě velkého počtu požadavků je instance aplikace replikována na několik dalších serverů, na něž jsou pak požadavky rovnoměrně distribuovány. Při následném poklesu požadavků naopak dochází k rušení takto vytvořených instancí. [24], [25]

Každá aplikace běží ve svém vlastním prostředí zvaném *sandbox*. Technika zvaná *sandboxing* umožňuje GAE provozovat více aplikací na tom samém serveru bez jejich vzájemného ovlivňování. Pokud v danou chvíli existuje více instancí aplikace než jedna (z důvodu škálování), požadavek vždy vyřizuje server, u kterého je nejnižší předpokládaná doba jeho zpracování. Zároveň to znamená, že další požadavek od stejného klienta již nemusí být nutně zprácován tím samým serverem. [24], [25]

4.2.1 Podporovaná prostředí

V rámci GAE je v době psaní diplomové práce možné zvolit ze čtyř prostředí pro programovací jazyky Java, Python, Go a PHP. Ke každému jazyku je k dispozici vlastní GAE SDK. Podpora pro poslední dva jmenované je v současné době v experimentální fázi. GAE API pro Go a PHP tedy mohou být poskytovatelem kdykoli změněna, přičemž hrozí zpětná nekompatibilita. [23]

Podpora pro Javu je samostatně popsána v následující kapitole 4.2.2 (Javu na GAE využívá praktická část diplomové práce). Následuje alespoň stručný přehled podpory pro zbývající jmenované jazyky [23]:

- **Python** – aplikace napsaná v Pythonu komunikuje s GAE prostřednictvím WSGI protokolu[15]. K dispozici je SDK pro Python 2.7 a 2.5. SDK není kompatibilní s Python verze 3. Součástí SDK je také jednoduchý framework webapp2 (případně starší webapp pro Python 2.5). Ostatní frameworky, jako např. oblíbené Django, Bottle nebo Flask jsou taktéž podporovány. Python extenze pro jazyk C není v prostředí GAE povolena.
- **Go** – tento jazyk byl vytvořen přímo společností Google a jedná se o mix Javy a C. Je zaměřen na konkurenční programování. V experimentálním režimu je podporována verze 1.
- **PHP** – podpora tohoto masově rozšířeného skriptovacího jazyka byla do GAE začleněna teprve nedávno. SDK je stále v rané verzi a v tzv. *limited preview*. SDK je tedy uvolněno pro testování a vývoj aplikací v lokálním prostředí, ovšem přímo na GAE servery lze zatím nahrát jen aplikace explicitně povolené společností Google.

Ne všechna funkcionalita ze standardních knihoven těchto jazyků je dostupná (viz 4.2.4). Google pro každý jazyk zveřejňuje *white list* tříd, které je možné použít.

[15] WSGI je zkratka pro Web Server Gateway Interface, jednoduché univerzální rozhraní umožňující vzájemnou komunikaci mezi webovým serverem a webovou aplikací napsanou v Pythonu.

4.2.2 Podpora jazyka Java pro J2EE na GAE

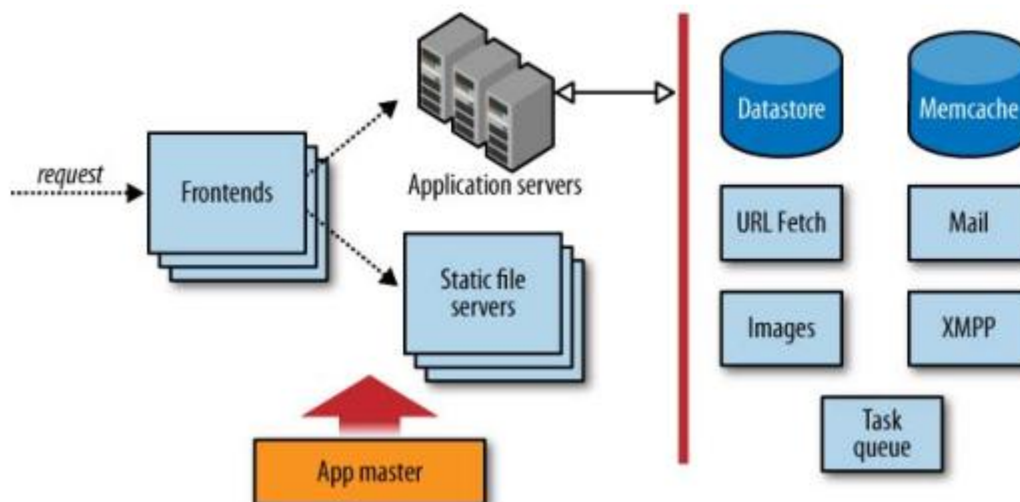
V době psaní práce GAE Java SDK podporuje verze 6 a 7. Google však oznámil, že končí s podporou Javy 6 v budoucích aktualizacích. Uživatelům provozujícím aplikace pro starší verzi Javy doporučil migraci na novější verzi. [23]

GAE používá servletový standard. Aplikace se tedy obvykle skládá z přeložených servletových tříd, JSP stránek, statických a datových souborů a deskriptoru v podobě souboru *web.xml*. GAE obsluhuje požadavky tak, že podle *web.xml* volá příslušné servlety, jak bylo vysvětleno v předchozí kapitole 4.2.

Pro perzistování dat je možné zvolit mezi JDO a JPA. Zatímco implementace JDO je na velmi dobré úrovni a poskytuje převážnou většinu funkcí popsaných ve standardu, JPA je doporučeno se spíše vyhýbat. Podpora JPA se nachází v experimentálním stádiu. Protože se navíc přistupuje k nerelační databázi, některé metody předpokládající využití objektově relačního mapování při svém zavolání vyvrhne výjimku *UnsupportedOperationException*.

4.2.3 Vyřizování požadavků

Z hlediska architektury GAE je důležité popsat způsob vyřizování klientských požadavků na aplikaci. Následující popis je ilustrován schématem na obrázku 15.



Obrázek 15 – Obsluha požadavku na GAE. Zdroj: [25]

Příchozí požadavek je nejprve zachycen GAE frontendem. *Load balancer*, dedikovaný systém zodpovědný za optimální distribuci požadavků, směruje požadavek na jeden z frontendových serverů. Frontendový server má za úkol určit, pro kterou aplikaci je požadavek určen. Tuto informaci extrahuje z doménového jména obsaženého v hlavičce požadavku. Dále se podívá do konfigurace příslušné aplikace, aby určil nadcházející krok. [25]

Fronted může být dále nakonfigurován tak, aby spolupracoval se službou Google Accounts. Lze tak např. omezit přístup k různým URL podle úrovně autorizace.

Administrátor aplikace si také může zvolit z jedné z nabízených výkonových variant frontendu (viz obrázek 16). Vyšší vykonové třídy frontendu pak samozřejmě spotřebovávají více prostředků, což se projeví na ceně služby. [25]

Obrázek 16 – Nastavení výkonové třídy frontendu na GAE. Zdroj: vlastní

Pokud URL požadavku cílí na statický soubor (např. obrázek nebo JavaScriptový soubor), frontendový server předá požadavek serverům pro statické soubory. Tyto servery mají síťovou topologii a konfiguraci optimalizovanou pro rychlé doručování obsahu, který se mění pouze výjimečně. Které soubory jsou statické, je možné specifikovat v souboru *appengine-web.xml* dané aplikace. [25]

V případě, že URL vyhovuje vzoru, na který je namapován *request handler* aplikace, frontendový server předá požadavek poolu aplikačních serverů. Mapování je zaznamenáno v deskriptoru[16] aplikace (ukázka viz kód 2). Počet namapovaných handlerů je omezen na 100 (včetně mapování interních požadavků, tedy např. filtrů u Javy). Při testování Java aplikace s již 50 záznamy mapování se však vyskytly technické problémy, kdy aplikace odmítala naběhnout. Tento údaj je tedy nutné brát s rezervou. [25], [23]

```
...
<servlet>
   <servlet-name>jersey-serlvet</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
               <param-name>
                   com.sun.jersey.config.property.packages
               </param-name>
            <param-value>webservice.rest</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
        <servlet-name>jersey-serlvet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
...
```

Kód 10 – Ukázka mapování deskriptoru web.xml pro Java aplikaci. Zdroj: vlastní

[16] Pro Javu je to soubor *web.xml*, pro Go a Python pak *app.yaml*.

Jak bylo řečeno dříve v této kapitole, z poolu aplikačních serverů je pro obsloužení požadavku vybrán ten, u kterého se předpokládá nejrychlejší vyřízení požadavku. Cílem je maximalizovat propustnost. Aplikační server pak vyvolá příslušný *request handler*, který přípraví odpověď, vrátí ji a ukončí se. Aplikační server odešle odpověď klientovi, až když *request handler* terminoval. Z toho vyplývá, že není možné odpověď streamovat nebo nechat spojení otevřené po delší dobu. [25]

4.2.4 Restrikce pro aplikace

Aplikace hostované na GAE musejí za benefity automatické škálovatelnosti či možnosti využívat Google API platit daň v podobě různých restrikcí, které standardní webhosting obvykle nevynucuje. Mezi ně se řadí [24], [25]:

- Časový limit na vyřízení požadavku, který je v současné době 60 sekund. Za zmínku stojí, že GAE je optimalizováno pro aplikace, jejichž střední doba vyřízení požadavku je méně než jedna sekunda.
- Pokud je aplikace náročná na procesor, GAE jí omezí procesorový čas, aby nezhoršovala odezvu instancí aplikací sdílejících stejný server.
- GAE může obsloužit pouze požadavky přícházející z protokolu HTTP či jeho zabezpečené varianty HTTPS.
- Aplikace nemá oprávnění zapisovat do souborového systému. Toto omezení je obzvlášť nepříjemné pro aplikace využívající logování. Jako náhradu lze zvolit GAE datastore (viz kapitola 4.5) s omezením velikosti entity na 1 MB. Při velikosti entity vetší jak uvedený limit je nutné využít Blobstore (službu pro ukládání velkých souborů).
- Existují různá další omezení (pro Datastore či JRE), která jsou popsána dále.

4.3 GAE služby

Jednou z velkých výhod vývoje aplikací na GAE je snadná integrace s Google službami. Uvedeny jsou služby, které jsou zároveň používány v praktické části diplomové práce. Jsou-li zahrnuty konfigurační a implementační detaily služby, je tak učiněno z hlediska Java prostředí GAE.

Users

GAE umožňuje autentizovat uživatele třemi způsoby [23]:

- přes Google Accounts,
- účtem ve vlastní Google Apps doméně,
- prostřednictvím Federated Login (využivá OpenID[17] identifikátor).

[17] Decentralizovaný způsob přihlašování uživatelů založený na otevřeném standardu. Pro provozovatele služby odpadá nutnost implementovat vlastní autentizační mechanismus. Benefitem pro uživatele je zase možnost použít stejné přihlašovací údaje pro více služeb.

Aplikace pak může detekovat, zda je uživatel přihlášený, případně jej přesměrovat na relevantní stránku, kde přihlášení provede (nebo vytvoří nový účet, pokud je použita autentizace přes Google Accounts). Aplikace má poté přístup k e-mailové adrese uživatele (nebo OpenID identifikátoru). Může také ověřit, zda uživatel disponuje administrátorskými právy. [23]

Získání aktuálního uživatele se provede dle kódu 11.

```
UserService userService = UserServiceFactory.getUserService();
User user = userService.getCurrentUser();
```

Kód 11 – Získání aktuálního uživatele pomocí služby Users. Zdroj: vlastní

Pokud je vráceno *null*, uživatel ještě není přihlášený a je třeba ho přesměrovat na přihlašovací stránku dané autentifikační služby. Typ *User* nabízí metody jako *getAuthDomain()*, *getNickname()*, *getEmail()* či *getFederatedIdentity()*, jejichž názvy jsou samovysvětlující.

4.3.1 Task Queues

Aplikace může přidávat úlohy do fronty, která je monitoruje a postupně zpracovává. Pro aplikaci lze v současné době vytvořit až deset front s různou konfigurací. Pro zařazené úlohy fronta zajišťuje, že budou spuštěny minimálně jednou a v případě selhání opakovány do doby, než proběhne jejich úspěšné vykonání. [25], [23]

Fronta funguje na bázi algoritmu *token bucket*. Nádrž s určitou kapacitou (*bucket*) se danou rychlostí doplňuje *tokeny*. Pro zpracování jedné úlohy je třeba z nádrže odebrat jeden *token*. GAE tedy nevyužívá žádné sofistikované metody pro odhad složitosti úloh, kdy by náročnější úlohy byly ohodnoceny vyšším počtem *tokenů*.

Fronta může být dvojího typu [23]:

- **Push fronta** – výchozí typ fronty. Zpracovává úlohy na základě konfiguračního souboru. Po úspěšném provedení je úloha z fronty odstraněna. GAE procesní kapacity fronty automaticky škáluje podle počtu a výpočetní náročnosti úloh.
- **Pull fronta** – umožňuje zpracovávat úlohy ve specifickém časovém rámci. Pro zpracování úloh je možné využít aplikace třetích stran prostřednictvím zatím ještě experimentálního Task Queue REST API. Tento typ fronty dává vývojáři větší kontrolu, avšak škálování a odstraňování úloh z fronty (úlohy lze opakovat) si musí řešit sám.

Konfigurace pro fronty se ukládá do souboru *queue.xml* (pokud tento soubor neexistuje, je k dispozici výchozí fronta pojmenovaná *default*). V případě fronty typu *push* může obsah souboru vypadat tak, jak znázorňuje kód 12.

```
<queue-entries>
    <queue>
        <name>send-email-queue</name>
        <rate>1/s</rate>
```

```
        <bucket-size>10</bucket-size>
        <retry-parameters>
            <min-backoff-seconds>10</min-backoff-seconds>
            <max-backoff-seconds>40</max-backoff-seconds>
            <max-doublings>2</max-doublings>
        </retry-parameters>
    </queue>
</queue-entries>
```

Kód 12 – Ukázka konfigurace fronty typu push v souboru queue.xml. Zdroj: vlastní

Jak bylo řečeno, element `<queue-entries>` může obsahovat až 10 vnořených front, kde každá z nich je reprezentována elementem `<queue>`. Element `<name>` určuje jedinečný název fronty, dle kterého je identifikována. `<rate>` definuje rychlost přidávání *tokenů* do nádrže. `<bucket-size>` určuje kapacitu nádrže. `<retry-parameters>` se uplatňuje pro případ neúspěšného provedení úlohy. `<min-backoff-seconds>` udává výchozí časový bod, za kolik sekund se má neúspěšná úloha opakovat. Tento čas je po každém neúspěšném pokusu násoben dvěma, a to tolikrát, jaká je hodnota v `<max-doublings>` (pokud je hodnota nula, pravidlo není aplikováno). V případě, že se po neúspěchu úlohy neaplikuje zdvojnásobení času, je k aktuální periodě přičtena hodnota `<min-backoff-seconds>` až do dosažení hodnoty v `<max-backoff-seconds>`. [23]

Pokud by tedy určitá úloha z fronty *send-email-queue* z kódu 4 neustále selhávala, nabývala by pro ni perioda opakování hodnot 10, 20, 40, 40, …, 40, …

Kromě uvedených existují pro fronty i další konfigurační elementy, které lze najít v oficiální dokumentaci GAE [23].

Zařazení úlohy do fronty je jednoduché (viz kód 13).

```
Queue queue = QueueFactory.getQueue("send-email-queue");
            Uzivatel loggedUzivatel = (Uzivatel) session.get("user");
...
queue.add(TaskOptions.Builder.withUrl("/Task_sendEmail.do").method(Method
.POST).param("senderKeyString",loggedUzivatel.getGoogleId())
.param("recipientKeyString", uzivatel.getGoogleId()).param("body", telo).
param("subject", predmet));
```

Kód 13 – Ukázka zařazení úlohy do GAE fronty v Javě. Zdroj: vlastní

4.3.2 Search

API služby Search umožňuje prohledávat strukturovaná data různých formátů. Tato služba je dostupná pouze pro aplikace využívající High Replication Datastore (viz kapitola 4.6). Search API je do dnešního dne stále ve stavu *preview release*. Služba tedy není pokryta SLA a samotné API se může kdykoli změnit. [23]

Ve svém jádru Seach API představuje jednoduchý model pro indexování a vyhledávání dat, stejně jako organizování a prezentaci nalezených záznamů. Vyhledávat lze jakákoli data, která jsou **popsána dokumentem**. Dokument sestává z libovolného počtu polí, která mohou obsahovat různý obsah (integrální hodnoty, datum, prostý text, HTML, …).

Pro vytváření dokumentů slouží třída *Document.Builder*. Pole lze vytvářet prostřednictvím metod třídy *Field.Builder*. Kód 14 ukazuje vytvoření dokumentu pro objekt *k* typu *Kniha*. [23]

```
Document document = Document.newBuilder().setId(k.getKeyString())
.addField(Field.newBuilder().setName("name").setText(k.getNazev()))
.addField(Field.newBuilder().setName("author").setText(k.getAutor()))
.addField(Field.newBuilder().setName("publisher").setText(k.getNakladatel
stvi()))
.addField(Field.newBuilder().setName("isbn").setText(k.getISBN()))
.addField(Field.newBuilder().setName("owner").setText(k.getVlastnikEntity
().getPrezdivka()))
.addField(Field.newBuilder().setName("location").setText(k.getVlastnikEnt
ity().getLokalita()))
        .build();
```

Kód 14 – Vytvoření dokumentu pro objekt typu Kniha. Zdroj: vlastní

Jakmile jsou data popsána, lze dokumenty dále strukturovat do indexů. K tomuto účelu se využívá třída *Index*. Tato umožňuje vkládat dokumenty do konkrétního indexu, mazat je a především se nad nimi dotazovat pomocí speciálního dotazovacího jazyka[18] Search API. Index se definuje vytvořením objektu typu *IndexSpec*. Kód 15 znázorňuje metodu pro získání indexu *bookSearch*. V případě, že služba Search index s takovýmto názvem neregistruje, je automaticky vytvořen. [23]

```
private Index getIndex() {
IndexSpec indexSpec =
        IndexSpec.newBuilder().setName("bookSearch").build();
return
        SearchServiceFactory.getSearchService().getIndex(indexSpec);
}
```

Kód 15 – Vytvoření dokumentu pro objekt typu Kniha. Zdroj: vlastní

Na závěr je třeba ukázat, jak provést samotné vyhledávání. Vyhledávání se spouští metodou *search()* pro daný index. Parametrem metody je řetězec představující dotaz nebo objekt typu *Query*. Druhá varianta je sofistikovanější možností pro vytvoření dotazu, kde pro jeho konstrukci lze využít metod tříd *Query.Builder*, *QueryOptions.Builder* a *SortOptions.Builder*. Tutu variantu je vhodnější aplikovat u složitějších dotazů. [23]

Výsledek vyhledávání představuje objekt typu *ScoredDocument*. Objekt obsahuje informace o tom, zda vyhledávací dotaz proběhl korektně, a seznam dokumentů vyhovujících vyhledávacímu kritériu. Podporuje také práci s kurzory. Metoda vracející množinu id knih na základě dotazu jako jejího parametru je ukázána v kódu 16.

```
public Set<Key> searchBook(String searchText) {
   Set<Key> bookIds = new HashSet<Key>();
   Results<ScoredDocument> results = getIndex().search(searchText);
```

[18] Kromě standardních operátorů pro porovnávání a nerovnost lze využít např. operátor tilda ~, který pro zadané podstatné jméno v singuláru prohledá i jeho varianty v plurálu. Napíšeme-li do dotazu např. ~"book", bude vrácen výsledek pro "book" i "books".

```
    for (ScoredDocument document : results) {
        bookIds.add(KeyFactory.stringToKey(document.getId()));
    }
    return bookIds;
}
```

Kód 16 – Vytvoření dokumentu pro objekt typu Kniha. Zdroj: vlastní

4.3.3 App Identity

Služba App Identity řeší problém získání identifikátoru prostředí, ve kterém je kód spouštěn. Identifikátor je obvykle nutné získat v případě generování URL či pro větvení kódu na základě aktuálního prostředí. Lze jej jednoduše získat zavoláním `ApiProxy.Environment.getAppId()`. App Identity také nabízí možnost identifikovat zdroj požadavku, avšak tato funkcionalita je stále ještě v experimentálním stádiu. [23]

4.4 GAE datastore

GAE datastore představuje bezschémovou objektovou databázi založenou na technologiích **Google File System (GFS), Bigtable** a některých dalších. Než bude přikročeno k vysvětlení hlavní funkcionality GAE datastore, je na místě nejprve stručné seznámení s těmito technologiemi.

4.4.1 Google File System (GFS)

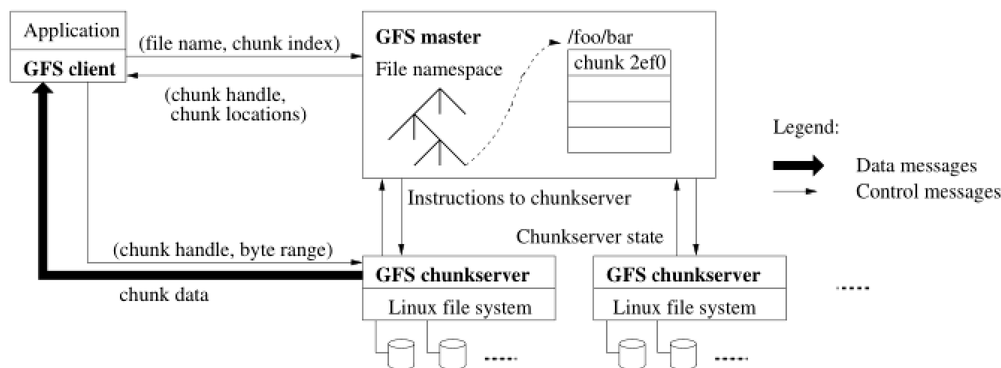
GFS je škálovatelný distribuovaný souborový systém vytvořený speciálně pro potřeby společnosti Google. GFS sestává z tisíců datových uložišť vybudovaných z běžně dostupných hardwarových komponent. Vzhledem k míře zátěže, které takový systém musí odolávat, je zřejmé, že hardwarové poruchy se stávají spíše normou než výjimečným úkazem. GFS proto neustále monitoruje sám sebe, detekuje chyby a snaží se z nich rychle zotavit. [26]

Typické jsou soubory o velikosti 100 MB a také větší počet souborů o několika GB, do kterých jsou ukládány veškeré objekty, se kterými Google aplikace pracují. Vzhledem k objemu přenášených dat by správa enormního množství souborů o velikosti několika KB nebyla udržitelná. To však neznamená, že menší soubory nejsou podporovány. Jen pro ně GFS není optimalizován. [26]

Nízká doba odezvy pro GFS není tak důležitá jako udržitelná datová propustnost. Většina Google aplikací je totiž zaměřena na zpracovávání velkého objemu dat. Jen u několika málo aplikací jsou vynucována časová omezení s ohledem na individuální operace čtení a zápisu. [26]

Architektura GFS

GFS cluster se skládá z jednoho **master serveru** a libovolného množství tzv. **chunk serverů** (viz obrázek 8). Soubory jsou rozděleny na kousky, tzv. *chunks*. Při vytvoření *chunku* je mu master serverem přiřazen jedinečný a neměnitelný identifikátor, tzv. *chunk handle*. Jednotlivé *chunky* jsou replikovány napříč *chunk servery* pro zajištění vyšší spolehlivosti a odolnosti GFS proti chybám. [26]



Obrázek 17 – Architektura GFS. Zdroj: [26]

Chunks jsou na *chunk serverech* uloženy jako běžné linuxové soubory, přičemž zápis a čtění z nich je realizováno za pomocí zmiňovaného *chunk handle* a bytového rozsahu. GFS stanovuje velikost *chunku* na 64 MB, což je nepoměrně víc než velikost bloku volená u běžných souborových systémů. Hlavní výhodou tohoto přístupu je fakt, že vzhledem k velikosti bude klient s větší pravděpodobností provádět více operací na daném *chunku* za daný časový rámec, což znamená méně dotazování na master server. Master server je zřejmě nutné kontaktovat za účelem získání *chunk handle*. [26]

GFS Metadata

Master server ukládá tři typy metadat [26]:

- jmenné prostory pro *chunky* a linuxové soubory,
- mapování souborů na *chunky*,
- umístění všech replik daného *chunku*.

První dva uvedené se ukládají do *operačního logu*, který se nachází na lokálním disku master serveru a je navíc ještě replikován na vzdálená uložiště. Master server informace o umístění *chunků* neperzistuje, ale ptá se na ně každého *chunk serveru* při svém nastartování. Dále se zeptá v případě, že se do clusteru připojí nový *chunk server*. [26]

4.4.2 Bigtable

Bigtable je distribuovaný úložný systém pro spravování strukturovaných dat zaměřený na škálovatelnost. Bigtable využívají aplikace jako Gmail, Google Earth, Google Analytics nebo Google Finance. Především je však na této technologii založeno celé GAE datastore společně s GFS, který ukládá datové soubory a logy Bigtable. [27]

Co se datového modelu týče, je Bigtable řidkou, persistentní multidimenzionální datovou mapou. Tato mapa je indexována podle **klíče řádku**, **sloupce** a **časového razítka**. Lze tedy říci, že data v Bigtable jsou organizována do třírozměrné kostky. [27]

Řádek

Řádek je identifikován klíčem tvořící náhodný řetězec až do velikosti 64 KB. Běžná hodnota je však typicky 10 až 100 B. Data jsou v Bigtable seřazena abecedně podle hodnoty klíče řádku. Každá operace čtení nebo zápisu do příslušného řádku je atomická. Znamená to, že daná operace buď proběhne nad všemi zvolenými sloupci řádku, nebo vůbec. [27]

Každá tabulka je pro GFS optimalizována tak, že její řádky jsou dynamicky rozdělovány na rozsahy zvané *tablety*. To odlehčuje zátěži při požadavku vrácení „rozumného“ počtu řádků (je třeba kontaktovat jen omezený počet strojů). V případě, že by velikost daného *tabletu* překročila povolené meze, je na něj automaticky aplikován zvolený komprimační algoritmus. [27]

Sloupec

Sloupec je identifikován klíčem sloupce. Klíče sloupce se sdružují do rodin, tzv. *column families*. Rodina tvoří základní jednotku řízení přístupu (aplikace např. může data z vybrané rodiny jen číst, nikoli do rodiny zapisovat). Data spadající pod konkrétní rodinu jsou obvykle stejného typu a jsou také společně komprimovaná. [27]

Časové razítko

Časové razítko umožňuje, aby každá buňka Bigtable obsahovala různé verze dat. Jedná se o integrální hodnotu o velikosti 64 b. Při ukládání dat může časové razítko explicitně definovat uživatel, anebo je určeno Bigtable jako aktuální časová hodnota. Data jsou v buňce seřazena dle hodnoty časového razítka sestupně. Pokud je jeho určení ponecháno na Bigtable, může dojít k výskytu kolizí (stejný časový záznam pro různá data). Kolizím se dá předejít pouze tak, že hodnotu razítka explicitně stanoví uživatel. [27]

Data podléhají *garbage collectoru*, který automaticky vyřazuje staré verze. Uživatel může pro každou *column family* definovat, zda se ponechá posledních *n* verzí, anebo data „dostatečně čerstvá“. Obvykle se jedná o data ne starší než posledních sedm dní (vždy bude samozřejmě zachován alespoň jeden nejnovější zápis). [27]

4.4.3 Typy GAE datastore

Existují dva typy GAE datastore, a sice **High Replication** (HR) a **Master/Slave**. Druhý jmenovaný byl k datu 4.4. 2012 označen jako zastaralý. Google doporučuje migraci na novější HR. [23]

Master/Slave

Master/Slave je založen na principu určení jednoho z datových serverů jako master. Tento si drží veškerý obsah databáze. Data jsou zapisována výhradně na master server, který je asynchronně replikuje na své slave servery. Jelikož v kterémkoli okamžiku má master

server k dispozici všechna data, je zaručena konzistence pro čtení. Nevýhodou je skutečnost, že v případě servisního výpadku master serveru mohou být data dočasně nedostupná (nebyly ještě replikovány na slave servery). [23]

High Replication (HR)

HR zapisuje data synchronně do několika různých datacenter. To také znamená jistou prodlevu v dostupnosti dat napříč stanovenými datacentry. Nehrozí však problém s možnou nedostupností dat v případě výpadku jediného serveru jako u Master/Slave. V dalším textu se bude vždy předpokládat použití HR. [23]

4.4.4 Porovnání GAE datastore s tradičními relačními databázemi

Kromě distribuované architektury se GAE datastore od tradičních relačních databází liší zejména způsobem, jakým popisuje vztahy mezi datovými objekty. Místo cizích klíčů se používají reference na ostatní objekty. Vztah mezi perzistentními objekty může být buď *owned*, nebo *unowned*. Rozdíl mezi nimi je ten, že objekty ve vztahu *owned* nemohou existovat jeden bez druhého, zatímco *unowned* toto omezení nevynucuje. [23]

Zápis a čtení jsou škálovatelné. Při ukládání velkého množství entit obvykle dochází k distribuovanému zápisu do různých datových center. Rychlost čtení z datastore je zajištěna podporou pouze těch operací, které výkonnostně škálují s velikostí množiny vrácených entit (nezáleží tedy na velikosti prohledávané množiny). Omezená podpora dotazovacích operací znamená, že není možné použít [23]:

- operaci spojení,
- poddotazy,
- u restrikce nelze použít operátor nerovnosti pro dvě a více vlastnosti entity.

Využití technologie Bigtable s sebou přináší taktéž řadu odlišností. Některými z nich jsou [23], [27]:

- Rozdíl v indexování dat (viz kapitola 4.5.3).
- Každý řádek v Bigtable může (a nemusí) mít jiný počet sloupců než řádek předcházející.
- Uživatel má kontrolu nad tím, v jaké lokalitě budou jeho data ukládána.
- Uživatel může zvolit, zda budou data vrácena z paměti (pokud se tam nacházejí) či čtena přímo z disku.

4.5 Základní konstrukty GAE datastore

V této podkapitole jsou vysvětleny základní konstrukty GAE datastore z pohledu implementace JDO DataNucleus. Vychází se z poznatků uvedených v kapitole 3.4.1.

4.5.1 Entita

Objekt uložený v datastore se nazývá **entita**. Entity jsou ukládány do jedné ze šesti Bigtables, ze kterých se GAE datastore skládá. V tabulce jsou uloženy **všechny entity ze všech aplikací** využívajících datastore. Entita je v tabulce jednoznačně

identifikovatelná podle hodnoty svého **klíče**. Klíč může být datovou součástí entity a stát se tak **referencí** na jinou entitu. [25], [23]

Entitní klíč sestává ze dvou částí, které po vytvoření entity již nelze měnit [25]:

- **Identifikátor aplikace** – zaručuje unikátní jmenný prostor z důvodu předejití kolize s entitami z jiných aplikací. Tento je přiřazován automaticky na základě názvu, který vývojář při vytvoření nové aplikace v admin konzoli zvolil.
- **Cesta** – v GAE datastore lze vytvářet tzv. *ancestor* entity. Jedná se o hierarchickou vazbu mezi entitami stejného nebo různého druhu[19] známou jako *entity group*[20]. Cesta je spojení *n* dvojic ve tvaru *druh*:*identifikátor*[21]. Tato dvojice tvoří unikátní identifikátor entity. Cesta vždy začíná entitním klíčem kořene. Pokud daná entita není kořenovou, k cestě se připojí klíče nižších entit směrem od shora dolu, přičemž posledním (nejpravějším) klíčem bude ten dané entity. Hierarchická vazba zaručuje, že je možné v GAE kdykoli získat přístup k předkovi. Celá hierarchická skupina je uložena na stejném datovém centru, což významně urychluje práci s jejími členy. Nevýhodou je, že entita může být členem právě jedné *entity group* v určitém časovém okamžiku. Protože entitní klíč nelze měnit, nelze měnit ani cestu. Členství entity ve skupině je tedy permanentní. Smazání entity ze skupiny však nepředstavuje problém (nedochází k rozbití hiarchie, protože je uchovávána vždy celá cesta).

Datové složky entity jsou uloženy v jedné nebo více **vlastnostech** (*properties*). Každá vlastnost má svůj unikátní název a datový typ. Přehled dostupných datových typů pro jazyk Java uvádí tabulka 5. Pokud je u názvu typu uvedeno „GD“, jedná se o součást Google Data protokolu. Většina z nich je implementována jako textový řetězec s výjimkou geografického bodu (uspořádaná dvojice hodnot *float*) a hodnocení uživatele (celé číslo mezi 1 až 100, používá se např. pro hodnocení pluginů do internetového prohlížeče Chrome). Entity stejného druhu mohou mít různé vlastnosti, zatímco entity nestejného druhu zase stejnojmenné vlastnosti rozdílných datových typů [25]

Tabulka 5 – Java datové typy pro GAE datastore. Zdroj: [25]

Název	**Datový typ v Javě**
Text ve formátu Unicode (do 500 B, podpora indexace)	`java.lang.String`
Text ve formátu Unicode (neindexovaný)	`datastore.Text`
Bajtový řetězec (do 500 B, indexovaný)	`datastore.ShortBlob`
Bajtový řetězec (neindexovaný)	`datastore.Blob`
Boolean	`boolean`
Integer (64 b)	`byte`, `short`, `int`, nebo `long`

[19] Druh entity kategorizuje entitu z hlediska dotazování. Je specifikován při vytvoření entity a odvozen z nekvalifikovaného[19] jména entitní třídy. Příkladem může být druh *Book*.

[20] Každá entita je členem *entity group,* ať už je něčím předkem či potomkem nebo ne.

[21] Jedná se buďto o řetězec, který určí sama aplikace, nebo jeho hodnotu vygeneruje sám GAE datastore. GAE mezi těmito případy rozlišuje a na identifikátor entity se pak odvolává jako na *key name*, resp. *ID*.

Float	`float` nebo `double`
Datum a čas	`java.util.Date`
Nulová hodnota	`null`
Entitní klíč	`datastore.Key`
Google účet	`api.users.User`
Kategorie (GD)	`datastore.Category`
URL (GD)	`datastore.Link`
E-mailová adresa (GD)	`datastore.Email`
Geografický bod (GD)	`datastore.GeoPt`
Identifikátor pro Instant Messaging (GD)	`datastore.IMHandle`
Telefonní číslo (GD)	`datastore.PhoneNumber`
Poštovní adresa (GD)	`datastore.PostalAddress`
Hodnocení uživatele (GD)	`datastore.Rating`

4.5.2 Dotazy v GAE datastore

Dotaz vrací z datastore entitity, vlastnosti (jedná se pak o tzv. *projection queries*) nebo entitní klíče, které splňují množinu definovaných podmínek. Dotaz je vždy prováděn pouze nad zvoleným druhem entity, přičemž lze filtrovat dle [23]:

- vlastností,
- entitních klíčů,
- předků entity.

Výsledkem dotazu je nula a více entit, které lze navíc řadit podle jejich vlastností.

Objekt Query

Základem dotazování v GAE datastore (stále uvažujme JDO DataNucleus) je objekt implementující rozhraní *Query*. Tento objekt je vytvářen metodou *newQuery() PersistenceManageru*. *PersistenceManager* může mít v kterémkoli okamžiku přidružen libovolný počet *Query* objektů. Dotazy lze napsat celé v JDOQL nebo je možné využít pomocné metody *Query*, jako např. *setFilter()*, *setOrdering()* apod. Těm lze jako argumenty předat jednotlivé JDOQL řetězce. Na rozdíl od psaní celého dotazu je tato možnost přehlednější a lépe se v ní hledají případné chyby. [23]

Dotaz se spouští voláním metody *execute()*, případně *executeWithArray()*. Rozdíl je pouze v počtu parametrů, kdy pro *execute()* existují čtyři přetížené varianty pro nula až tři parametry v dotazu (viz níže), zatímco *executeWithArray()* přijímá jejich pole. [23]

Parametry

Často je třeba předat dotazům nějaké hodnoty. K tomuto účelu slouží metoda *declareParameters()*, v níž se deklarují typy a názvy parametrů oddělené čárkou. Metoda *getKniha*() v kódu 17 přečte entitu typu *Kniha* na základě jejího entitního klíče, který je deklarovaný jako parametr. V kódu je dále použita metoda *setUnique()*, která říká, zda se má očekávat vrácení jediného objektu. Pokud by jí byla předána hodnota *false*, výsledkem by byla kolekce daného entitního druhu (v tomto případě *Kniha*). [23]

```
public Kniha getKniha(Key key) {
```

```
    Kniha kniha = null;
       try {
             Query q = pm.newQuery(Kniha.class);
             q.setFilter("platny == true && key == keyParam");
             q.declareParameters("com.google.appengine.api.datastore.Key
keyParam");
             q.setUnique(true);
             kniha = (Kniha) q.execute(key);
       } catch (JDOObjectNotFoundException e) { }
         return kniha;
     }
```

Kód 17 – Ukázka použití parametru v JDO. Zdroj: vlastní

Filtry

Filtry se přidávají buď prostřednictvím metody *setFilter()*, nebo přímo do konstruktoru *Query*. Je třeba připomenout významné omezení GAE datastore, kdy v jednom dotazu nelze filtrovat dle nerovnosti na více jak jedné vlastnosti. Kromě základních porovnávacích operátorů je k dispozici i operátor *contains()* ne nepodobný *IN* ve standardním SQL. Ukázku použití filtru společně s operátorem *contains()* v konstruktoru *Query* obsahuje kód 18. Metoda v ukázce vrátí kolekci entit druhu *Hodnoceni* na základě množiny entitních klíčů. [23]

```
public List<Hodnoceni> getList(Set<Key> keys) {
    if (keys.isEmpty()) {
          return new ArrayList<Hodnoceni>();
    }
    Query q = pm.newQuery(Hodnoceni.class, ":p.contains(key)");
    List<Hodnoceni> hodnoceni = (List<Hodnoceni>) q.execute(keys);

    return hodnoceni;
}
```

Kód 18 – Ukázka použití filtru v JDO. Zdroj: vlastní

Rozsahy

Pro vymezení množiny výsledků lze využít rozsahy definované prostřednictvím metody *setRange()*. Jejími parametry jsou počáteční a koncový index. Pokud např. zavoláme *setRange(10, 20)*, GAE datastore nejprve získá 20 výsledků, prvních 10 zahodí a zbytek vrátí aplikaci. Google upozorňuje, že použití rozsahů pro velký počet záznamů může negativně ovlivnit výkon aplikace. Pro tyto případy se doporučovalo použití metody *fetch()*, která však přešla do statutu *obsolete/deprecated* a nahradily ji kurzory. Příklad použití rozsahu ukazuje kód 19. Metoda *getFilteredList()* vrátí kolekci knih v rozsahu *startItem* až *endItem* seřazenou podle zadaných kritérií. [23]

```
public List<Kniha> getFilteredList(String sortColumn, String sortOrder,
int startItem, int endItem) {
       List<Kniha> knihy = new ArrayList<Kniha>();
        try {
            Query q = pm.newQuery(Kniha.class);
            if (sortColumn != null) {
                q.setOrdering(sortColumn + " " + sortOrder);
            } else {
                q.setOrdering("datumVytvoreni desc");
```

```
            }
            q.setFilter("platny == true");
            q.setRange(startItem, endItem);
            knihys = (List<Kniha>) q.execute();
        } catch (JDOObjectNotFoundException e) {
        }
        return knihy;
}
```

Kód 19 – Ukázka použití rozsahu v JDO. Zdroj: vlastní

Řazení

Řazení zprostředkovává metoda *addSort()*. Jejími parametry jsou vlastnosti, podle kterých se bude řadit, a výčtový typ *SortDirection* s literály *ASCENDING* a *DESCENDING* definující směr řazení. Alternativně má také přetíženou variantu čistě pro JDOQL řetězce. Metoda *getList()* v kódu 20 seřádí vrácené knihy podle data vytvoření sestupně užitím právě JDOQL řetězce. [23]

```
public List<Kniha> getList() {
    List<Kniha> knihy = new ArrayList<Kniha>();
      try {
         Query q = pm.newQuery(Kniha.class);
         q.setFilter("platny == true");
         q.setOrdering("datumVytvoreni desc");
         knihy = (List<Kniha>) q.execute();
      } catch (JDOObjectNotFoundException e) {
      }
    return knihy;
}
```

Kód 20 – Ukázka použití řazení v JDO. Zdroj: vlastní

4.5.3 Indexy v GAE datastore

GAE datastore na indexy nahlíží zcela jinak, než např. databáze Oracle. Pro každou vlastnost je při vytvoření entity implicitně definován jednoduchý index. Dále v případě, že perzistovanou datovou složkou je kolekce, vytvoří se kombinace indexů určena kartézským součinem prvků v kolekci. Pro velké kolekce může snadno dojít k **indexové explozi**. Počet indexů je již tak velký, že nastane vyvržení výjimky a uložení objektu do datastore se nezdaří. Indexy jsou nezbytné pro získávání objektů z datastore. Dotaz nemůže vrátit objekt, pokud pro porovnávané vlastnosti nebo druh entity neexistuje index. [23]

Struktura indexu

Index je definován na jedné či více vlastnostech entity určitého druhu. Indexy se rozdělují do čtyř typů na *EntitiesByKind*, *EntitiesByProperty ASC*, *EntitiesByProperty DESC* a *EntitiesByCompositeProperty*. První tři typy jsou vytvářeny automaticky, jakmile dojde k zápisu entity do datastore. Stejně tak jsou automaticky spravovány GAE enginem. Index typu *EntitiesByCompositeProperty* je nutné explicitně definovat (viz níže). Pro každý typ indexu je vyhrazena samostatná Bigtable. [23]

Index *EntitiesByKind* umožňuje získat všechny entity určitého druhu. Pokaždé, když dojde k zápisu nové entity, přidá GAE do indexové tabulky záznam. Strukturu indexové tabulky *EntitiesByKind* zachycuje tabulka 6. [23]

Tabulka 6 – Strukturu indexové tabulky EntitiesByKind. Zdroj: [23]

Název sloupce	**Typ**	**Popis**
Identifikátor aplikace	`string`	ID aplikace, které entita patří.
Druh entity	`string`	Název entitní třídy.
Entitní klíč	`Key`	Zpřístupňuje entitu.

Záznamy do indexové tabulky *EntitiesByProperty ASC* a *EntitiesByProperty DESC* se přidávají pro každou vlastnost vytvořené entity kromě datových typů *Text* a *Blob* (tyto nejsou indexovatelné). Jak názvy typů indexů napovídají, způsob řazení v dotazu pro danou vlastnost předurčuje tabulku, ze které bude index získán. Struktura je podobná jako v předchozím případě. Přibývají jen dva sloupce pro vlastnosti (viz tabulka 7). [23]

Tabulka 7 – Strukturu indexové tabulky EntitiesByProperty. Zdroj: [23]

Název sloupce	**Typ**	**Popis**
Identifikátor aplikace	`string`	ID aplikace, které entita patří.
Druh entity	`string`	Název entitní třídy.
Entitní klíč	`Key`	Zpřístupňuje entitu.
Název vlastnosti	`string`	Na název se odkazuje v dotazu, pokud tento pracuje pouze s jednou vlastností.
Hodnota vlastnosti	datový typ vlastnosti	Obsahuje hodnotu vlastnosti. Její velikost záleží na datovém typu.

Poslední jmenovaný typ indexu, *EntitiesByCompositeProperty*, se používá pro *komplexní dotazy*. Zde je třeba jasně definovat, co se rozumí pod pojmem *komplexní dotaz*. Je to takový dotaz, který splňuje alespoň jednu z následujících podmínek [23]:

- používá filtrování podle nerovnosti (větší, menší, …) a předka,
- používá filtrování podle rovnosti pro jednu nebo více vlastností.

Tyto indexy se již nevytvářejí automaticky. Vývojář je musí pro každý dotaz vytvořit sám. Neučíní-li tak, bude při pokusu o provedení daného dotazu vyvržena výjimka. Struktura tabulky *EntitiesByCompositeProperty* viz tabulka 8. [23]

Tabulka 8 – Strukturu indexové tabulky EntitiesByCompositeProperty. Zdroj: [23]

Název sloupce	**Typ**	**Popis**
Identifikátor indexu	`int64`	ID indexu.
Identifikátor aplikace	`string`	ID aplikace, které entita patří.
Druh entity	`string`	Název entitní třídy.
Předek	`string`	Pokud dotaz používá filtrování dle předků, uloží se pro každého předka entity vlastní

		záznam.
Hodnoty vlastností	různé	Pro každou použitou vlastnost se přidá sloupec s její hodnotou.
Entitní klíč	`Key`	Zpřístupňuje entitu.
Stav indexu	`int32`	

Konfigurace vlastních indexů

Vlastní indexy se pro Javu definují v souboru *datastore-indexes.xml*, který se ukládá do adresáře *WEB-INF*. Ukázku souboru obsahuje kód 21. Význam jednotlivých tagů a atributů by měl být z předchozího výkladu zřejmý.

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes autoGenerate="true">
    <datastore-index kind="Hodnoceni" ancestor="true" source="manual">
        <property name="hodnocenis_INTEGER_IDX" direction="asc"/>
    </datastore-index>

    <datastore-index kind="Kniha" ancestor="false" source="manual">
        <property name="vlastnik" direction="asc"/>
        <property name="datumVytvoreni" direction="desc"/>
    </datastore-index>

    <datastore-index kind="Vypujcka" ancestor="true" source="manual">
        <property name="vypujckas_INTEGER_IDX" direction="asc"/>
    </datastore-index>

    <datastore-index kind="Vypujcka" ancestor="false" source="manual">
        <property name="produkt" direction="asc"/>
        <property name="stav" direction="asc"/>
        <property name="datumDo" direction="desc"/>
    </datastore-index>
    ...
</datastore-indexes>
```

Kód 21 – Obsah souboru datastore-indexes.xml. Zdroj: vlastní

Sekvence pro generování ID

Sekvence se používají pro generování integrálních hodnot pro entity a vlastní indexy. Jsou uloženy ve vlastní Bigtable, která obsahuje minimálně dva řádky pro každou aplikaci. Jeden řádek pro všechny kořenové entity a druhý pro vlastní indexy. Pokud tedy vývojář nepoužívá hierarchické vazby mezi entitami, vystačí si pouze s těmito dvěma řádky. V opačném případě se jejich počet může navýšit. Tabulka 9 zachycuje strukturu Bigtable pro sekvence. [23]

Tabulka 9 – Struktura indexové tabulky pro ukládání sekvenci. Zdroj: [23]

Název sloupce	Typ	Popis
Identifikátor aplikace	`string`	ID aplikace, které sekvence patří.
Entitní klíč	`Key`	Klíč kořenové entity dané *entity group*, pokud je použita.
Další ID	`int64`	Integrální hodnota, která bude přiřazena jako další v pořadí.

Perfect index

V GAE dokumentaci se hovoří o tzv. *perfect indexu*. Je to takový index, který umožní vykonat daný dotaz nejefektivněji. Je definován pro (se zachováním pořadí) [23]:

1. vlastnoti porovnávané podle rovnosti,
2. vlastnosti porovnávané podle nerovnosti (ne více než jedna),
3. vlastnosti použité v řazení.

Perfect index pro dotaz v kódu 22 bude představovat tabulku entitních klíčů pro druh entity *Book* se sloupci pro hodnoty vlastností *nakladatelstvi, rokVydani* a *autor*. Index je nejprve seřazen vzestupně podle *nakladatelstvi* a posléze v tom samém pořadí pro vlastnost *autor*.

```
Query q = new Query("Book")
        .addFilter("nakladatelstvi", Query.FilterOperator.EQUAL, "Tiata")
        .addFilter("rokVydani", Query.FilterOperator.GREATER_THAN, 1970)
        .addSort("autor", Query.SortDirection.ASCENDING);
```

Kód 22 – Dotaz pro perfect index. Zdroj: vlastní

4.5.4 Transakce v GAE datastore

GAE datastore podporuje transakce, jejichž použití je čistě volitelné a není vynucováno. Transakce smí trvat maximálně 60 sekund s tím, že po uplynutí poloviny této doby se automaticky spustí 10 sekundový *idle* časovač (je-li transakce v této době nečinná, terminuje). [23]

Datstore vyvrhne výjimku, pokud v průběhu zpracování transakce nastane některý z těchto případů [23]:

- Entity ve stejné *entity group* jsou vystaveny příliš velkému počtu konkurenčních zápisů.
- Transakce vyčerpá svůj časový limit.
- GAE datastore registruje interní chybu.

Použití transakce

Vývojář může využít standardní JDO transakce (viz 3.4.1) nebo GAE datastore transakce. V druhém případě je klíčový objekt *DatastoreService*, který je nejprve nutné získat z *DatastoreServiceFactory*. Ukázku uplatnění GAE přístupu k transakcím zachycuje kód 23. V kódu lze dále vidět vytvoření vlastního entitního klíče metodou *createKey()* a explicitní nastavení vlastnosti pomocí metody *setProperty()*. [23]

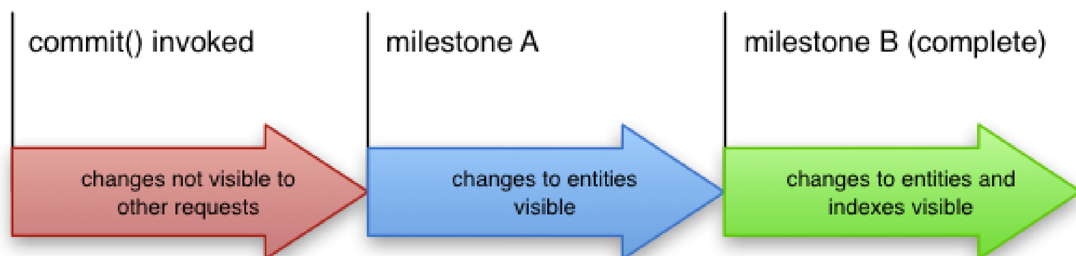
```
DatastoreService datastore =
        DatastoreServiceFactory.getDatastoreService()
Transaction txn = datastore.beginTransaction();
try {
    Key employeeKey = KeyFactory.createKey("Employee", "Joe");
    Entity employee = datastore.get(employeeKey);
    employee.setProperty("vacationDays", 10);

    datastore.put(employee);
```

```
        txn.commit();
} finally {
    if (txn.isActive()) {
        txn.rollback();
    }
}
```

Kód 23 – Využití DatastoreService v GAE transakci. Zdroj: [23]

Izolace transakcí v GAE datastore a commit

GAE datastore uplatňuje uvnitř transakcí izolační přístup *serializable*, zatímco vně transakcí *read commited.* Úspěšný *commit* znamená, že změny provedené v transakci budou aplikovány. Neznamená to však, že změny budou ihned viditelné pro čtení. Potvrzení transakce má dva milníky (viz obrázek 18). [23]



Obrázek 18 – Milníky potvrzení transakce. Zdroj: [23]

V High Replication datastore jsou transakční změny aplikovány v řádu stovek milisekund (zastaralý Master/Slave se obešel prakticky bez prodlevy). Pokud je registrován konkurenční požadavek pro získání modifikované entitity po milníku A, dostane se mu skutečně aktuální verze entity. Problém nastává při vykonání konkurenčního dotazu s predikátem (filtry či klauzule *WHERE*). Pokud daná entita nevyhovuje predikátu před její samotnou modifikací, ale vyhovuje mu po ní, pak bude součástí kolekce výsledků pouze po dosažení milníku B. [23]

5 Systém veřejné knihovny

Tato kapitola se zabývá popisem vývoje systému veřejné knihovny (dále jen SVK). Je zde shrnuta analytická, návrhová a implementační část.

Motivace pro vytvoření SVK

Vietnamská komunita v České republice poptává systém, přes který by bylo možné domluvit výpůjčky knižních titulů mezi jeho uživateli jako náhradu za stávající řešení. Stávající řešení je velmi primitivní. Skládá se v podstatě z jedné statické HTML stránky s nutností manuální úpravy zdrojového kódu v případě požadavku na změnu dat. Z uvedeného je zřejmé, že podrobnější srovnání tohoto systému s novým by nebylo smysluplné. Nový systém by kromě nezbytné funkcionality měl nabízet také běžné komunitní funkce, jako např. hodnocení knižních titulů a spolehlivosti uživatelů.

Zvolené technologie

Představitel vietnamské komunity udává jako nejvyšší prioritu nízkou provozní cenu, od čehož se odvíjí také volba technologií pro vývoj a hostování aplikace. Zvolena je platforma J2EE na GAE. Shrnutí domluvených technologií je následující:

- Platforma J2EE s aplikačním rámcem Struts 2.
- Šablonovací systém Tiles společně s JSP na frontendu.
- Integrace a hostování na GAE.
- JDO pro přístup ke GAE datastore.
- Využití vybraných GAE služeb.
- Google Maps a Google Street View pro upřesnění místa výpůjčky.

5.1 Analýza SVK

U SVK se předpokládá dlouhá životnost a patrně časem i obměna na postu vývojáře. Detailní model celé aplikace je tedy nezbytným základem. V rámci analytické části je proveden sběr požadavků, vytvořeny případy užití společně se scénáři a taktéž provedena jejich realizace. Použit je modelovací jazyk Unified Modeling Language (UML) v CASE nástroji Enterprise Architect (EA).

Analytická část používá jmenné konvence zapsané regulárními výrazy dle tabulky 10.

Tabulka 10 – Jmenné konvence artefaktů analýzy SVK. Zdroj: vlastní

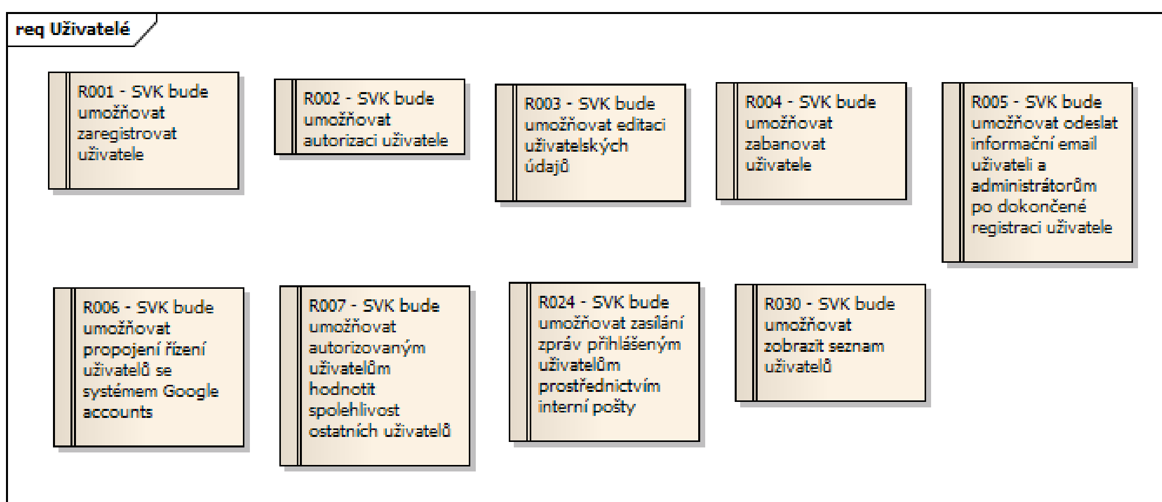
Artefakt	Jmenná konvence
Funkční požadavek	R[0-9]{3}
Nefunkční požadavek	NR[0-9]{3}
Případ užití	UC[0-9]{3}

5.1.1 Požadavky

Na základě konzultací s kontaktní osobou jsou sesbírány a kategorizovány požadavky na funkcionalitu SVK.

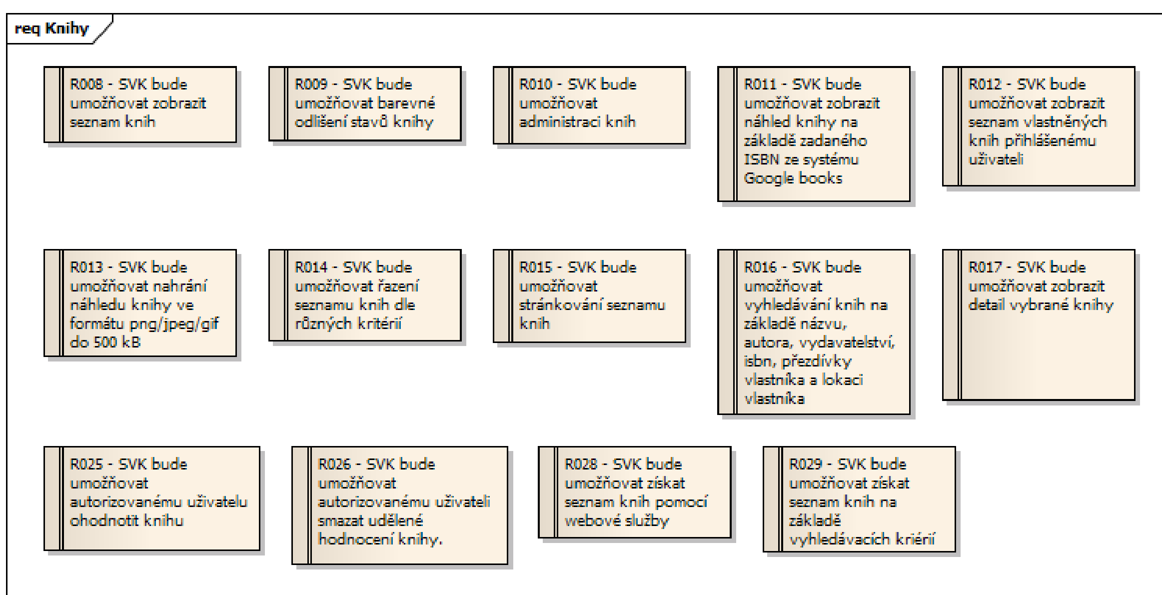
Funkční požadavky

Jako první jsou uvedeny požadavky na podporu uživatelských funkcí (viz obrázek 19). Kategorie obsahuje standardní požadavky na komunitní systém.



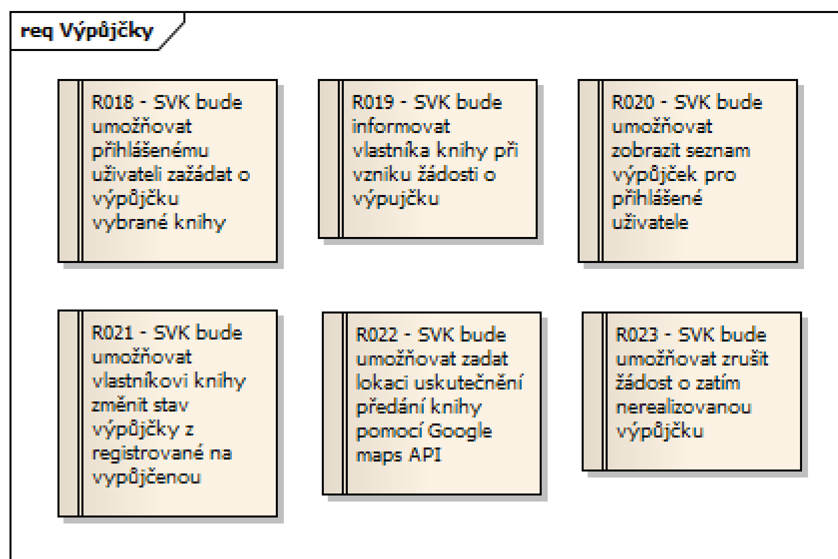
Obrázek 19 – Funkční požadavky kategorie Uživatelé. Zdroj: vlastní

Další kategorie se zaměřuje na funkcionalitu spojenou s knihami (viz obrázek 20).



Obrázek 20 – Funkční požadavky kategorie Knihy. Zdroj: vlastní

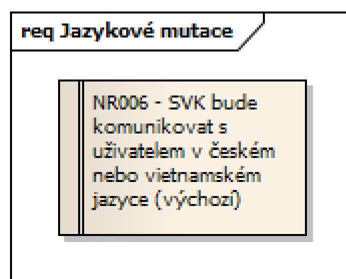
Konečně je třeba sumarizovat požadavky na výpůjčky (viz obrázek 21).



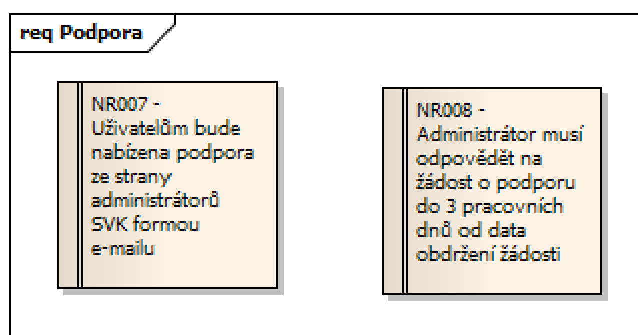
Obrázek 21 – Funkční požadavky kategorie Výpůjčky. Zdroj: vlastní

Nefunkční požadavky

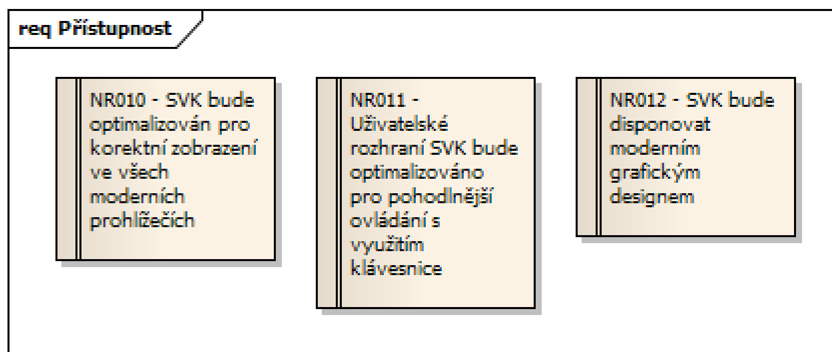
Nefunkční požadavky jsou rozděleny do čtyř kategorií (viz obrázky 22, 23, 24 a 25).



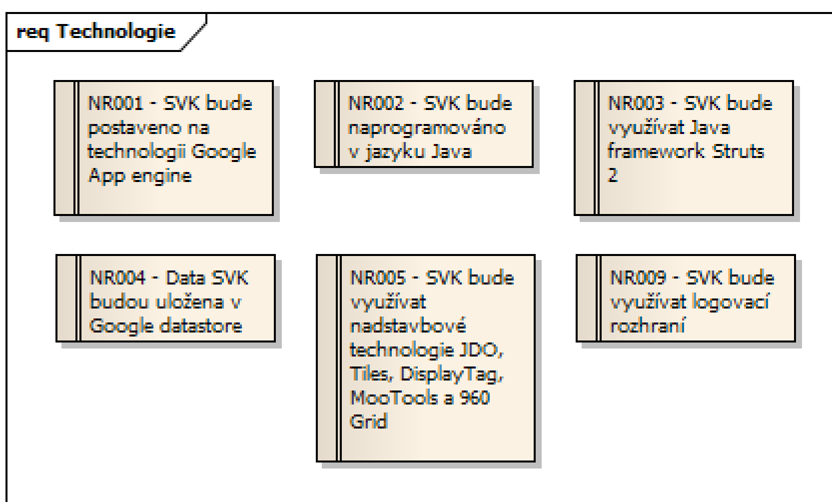
Obrázek 22 – Nefunkční požadavky kategorie Jazykové mutace. Zdroj: vlastní



Obrázek 23 – Nefunkční požadavky kategorie Podpora. Zdroj: vlastní



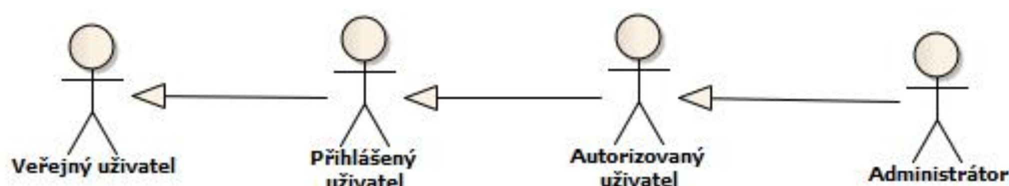
Obrázek 24 – Nefunkční požadavky kategorie Přístupnost. Zdroj: vlastní



Obrázek 25 – Nefunkční požadavky kategorie Technologie. Zdroj: vlastní

5.1.2 Aktéři

Nalezení aktéři jsou rozděleni na interní (viz obrázek 26) a externí (viz obrázek 27). Níže je uveden popis jednotlivých rolí, které aktéři představují.



Obrázek 26 – Interní aktéři. Zdroj: vlastní

- **Veřejný uživatel** – návštěvník SVK, který prozatím nemá vytvořený uživatelský účet. Může si prohlížet seznam knih, ale není mu dovoleno kontaktovat ostatní uživatele, žádat o výpůjčky apod.
- **Přihlášený uživatel** – uživatel, který se přihlásil prostřednictvím svého Google Accounts účtu. V SVK se mu tak automaticky vytvořil vlastní účet, který může editovat. Stále se však ještě nemůže zapojit do běžného provozu SVK.

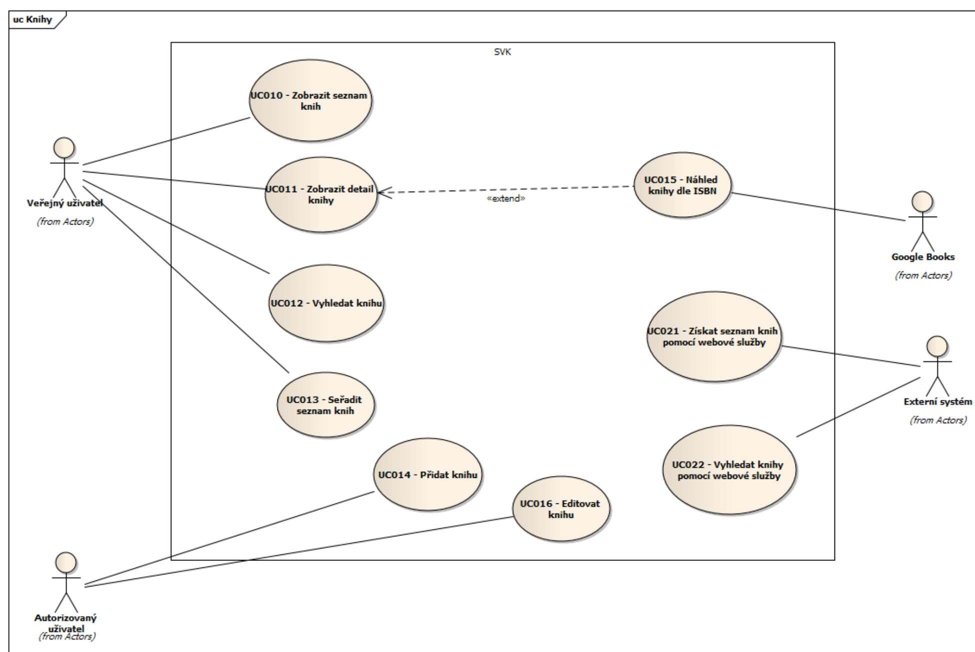
- **Autorizovaný uživatel** – v momentě, kdy administrátor na základě posouzení autorizuje přihlášeného uživatele, je mu přiřazena tato role. Autorizovaný uživatel již může využívat kompletní funkcionalitu SVK.
- **Administrátor** – běžná role pro každý komunitní systém. Administrátor autorizuje nové uživatele, má pravomoc odstavit účet v případě hrubého porušení pravidel SVK apod.

Obrázek 27 – Externí aktéři. Zdroj: vlastní

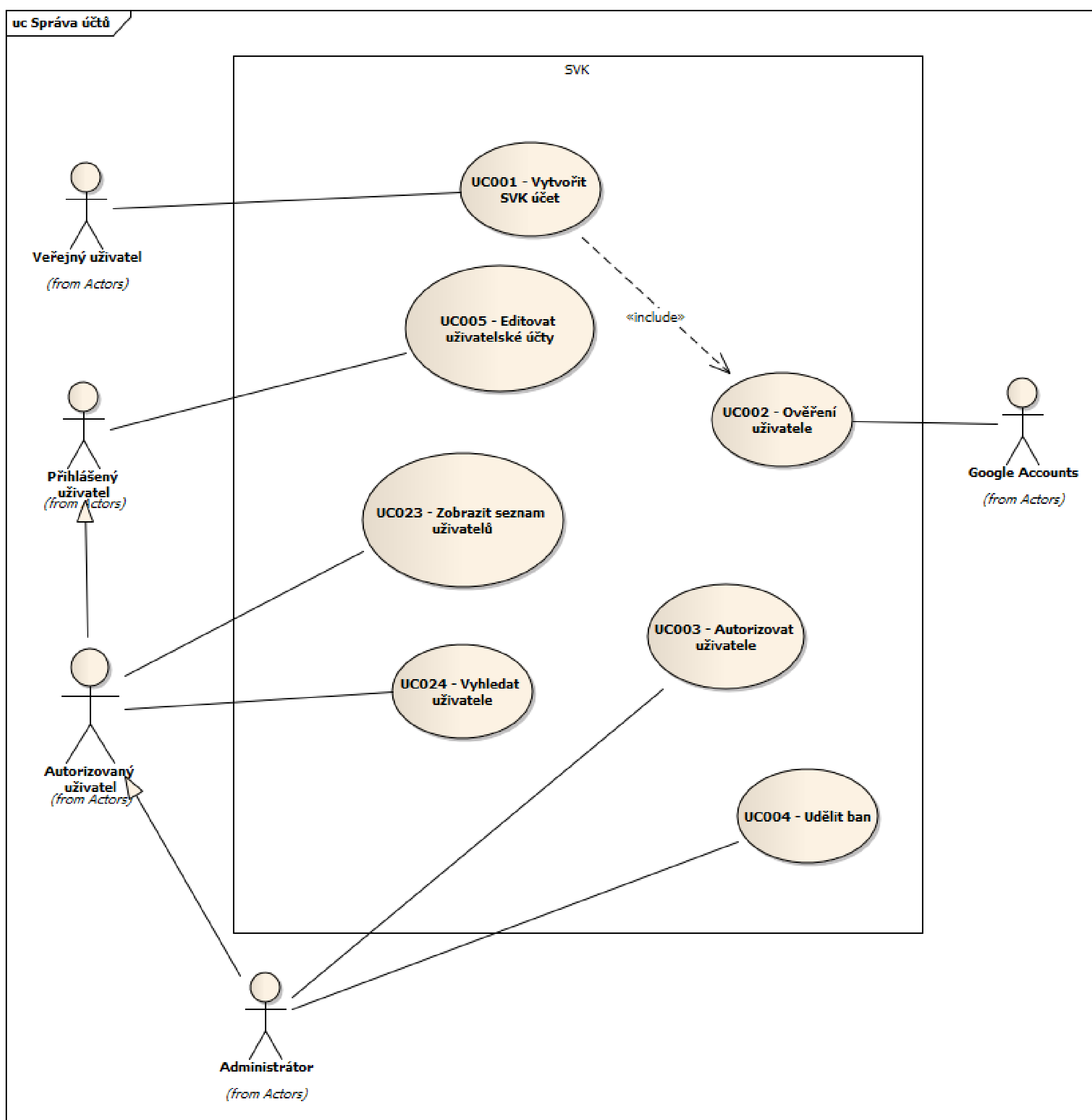
- **Google Accounts** – zprostředkovává přihlašování uživatelů. SVK pak může přečíst informace spojené s Google Accounts účtem.
- **Google Books** – SVK jej využívá pro získávání informací o daném knižním titulu na základě jeho ISBN.
- **Externí systém** – SVK nabízí seznam knih prostřednictvím RESTful webové služby s WADL rozhraním. Externí systém právě této služby využívá. Může se jednat prakticky o jakékoli zařízení od webového prohlížeče až po specializovanou aplikaci pro mobilní telefony a tablety.

5.1.3 Případy užití

Pro přehlednost je vytvořeno hned několik diagramů případů užití ukazujících, jak aktéři interagují s SVK. Obrázek 28 znázorňuje interakci z pohledu knižních titulů. Obrázek 29 pak ukazuje správu účtů. Zbývající diagramy lze nalézt v EA modelu SVK.



Obrázek 28 – Diagram případů užití Knihy. Zdroj: vlastní



Obrázek 29 – Diagram případů užití Správa účtů. Zdroj: vlastní

Dále jsou pro vybrané důležité případy užití sestaveny scénáře. Příklad jednoho ze scénářů uvádí tabulka 11.

Tabulka 11 – Hlavní scénář případu užití UC014. Zdroj: vlastní

Případ užití: UC014 - Přidat knihu	
ID	UC006
Stručný popis	Registrovaný uživatel přidá novou knihu do systému
Hlavní aktéři	Registrovaný uživatel
Vedlejší aktéři	Žádní
Vstupní podmínky	Žádné
Hlavní scénář	1. Scénář začíná po zvolení možnosti „Přidat knihu” 2. DOKUD jsou údaje o knize neplatné 2.1. Systém žádá uživatele, aby zadal všechny údaje o knize včetně Názvu, ISBN a autora

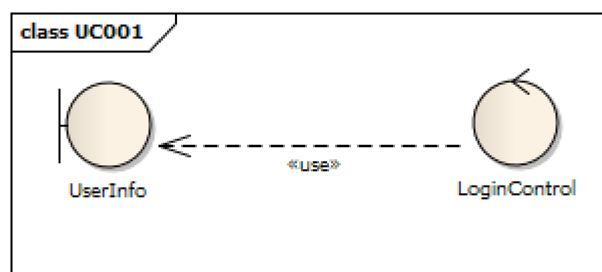
	2.2. Systém ověří údaje o knize 3. Systém vytvoří nový záznam knihy
Alternativní scénáře	**UC006.1 - Přidat knihu k výpůjčce - Neplatné ISBN** **UC006.2 - Přidat knihu k výpůjčce - Neplatný název** **UC006.3 - Přidat knihu k výpůjčce - Neplatný autor** **UC006.4 - Přidat knihu k výpůjčce - Storno**
Výstupní podmínky	Pro knihu byl vytvořen nový záznam

Integrita mezi funkčními požadavky a případy užití je ověřena maticí pokrytí dostupnou ke zhlédnutí v příloze A.

Realizace případů užití

Případy užití jsou realizovány prostřednictvím analytických tříd a sekvenčních diagramů v samostatných balíčcích označených dle identifikátoru případu užití. Analyticke třídy jsou nalezeny na základě znalosti cílové domény vycházející z konzultací se zadavatelem společně s metodou hledání podstatných jmen a sloves (hlavním zdrojem jsou scénáře). Rozděleny jsou do nestandardních UML stereotypů *boundary*, *control*, *entity* dle architektonického vzoru Model View Controller (MVC).

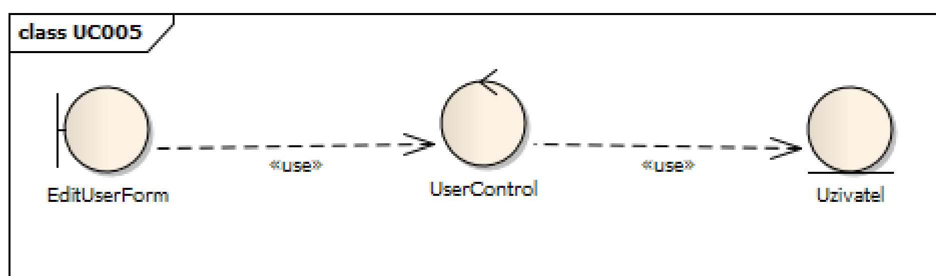
Obrázky 30 a 31 představují realizaci případu užití UC001. Jedná se o specificky řešené přihlašování do systému prostřednictvím služby Google Accounts. Přihlašuje-li se uživatel poprvé, je mu navíc vytvořen separátní účet pro SVK. Tento je napojen na Google Accounts a má přístup k vybraným informacím o uživatelově identitě.



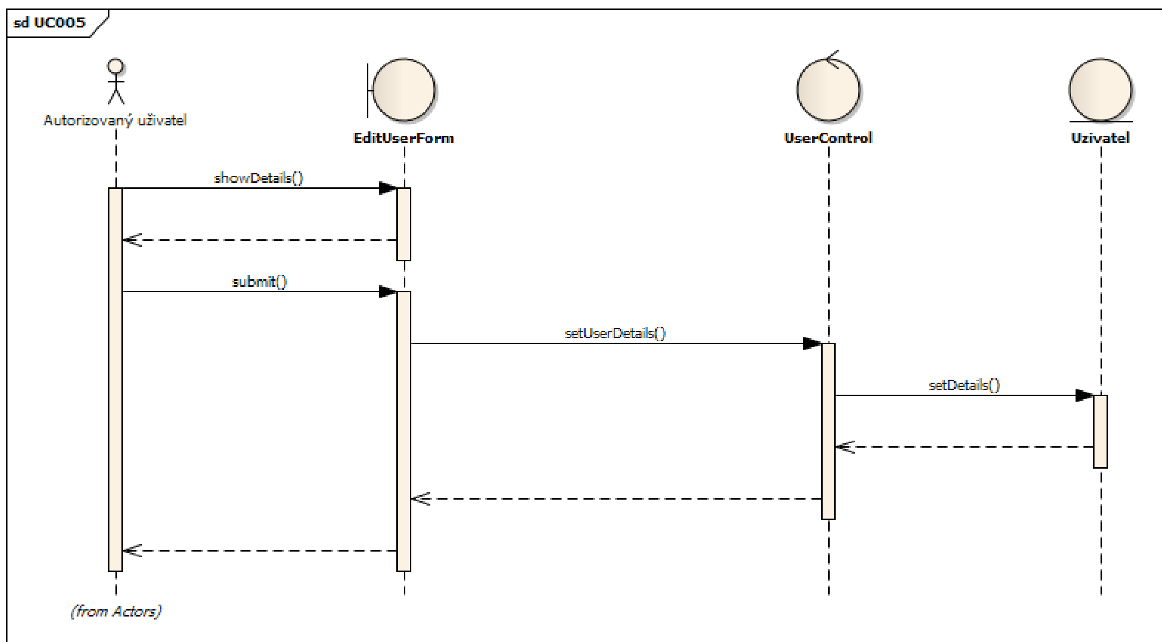
Obrázek 30 – Realizace UC001 (analytické třídy). Zdroj: vlastní

Obrázek 31 – Realizace UC001 (sekvenční diagram). Zdroj: vlastní

Obrázky 32 a 33 znázorňují realizaci případu užití UC005. Jedná se o standardní případ editace uživatelského profilu. Ostatní artefakty této fáze analýzy jsou součástí EA modelu.



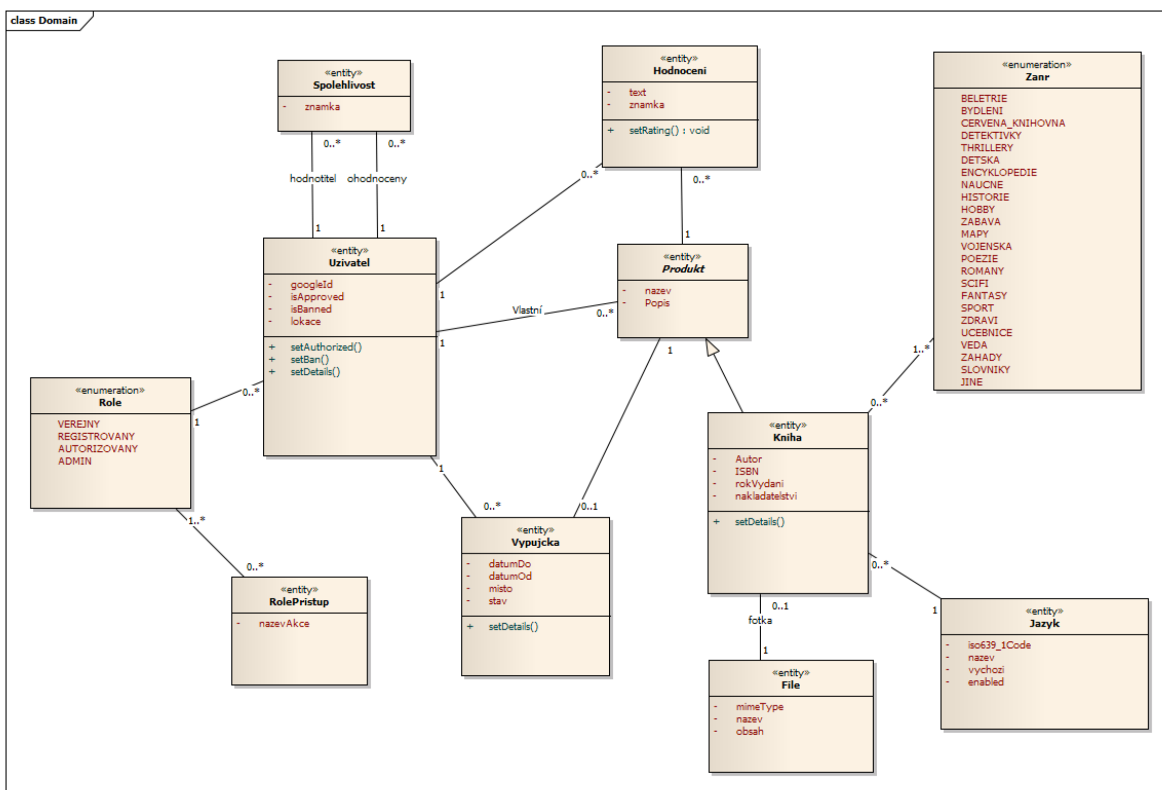
Obrázek 32 – Realizace UC005 (analytické třídy). Zdroj: vlastní



Obrázek 33 – Realizace UC005 (sekvenční diagram). Zdroj: vlastní

5.1.4 Doménový model

Z doménového modelu (v UML jej standardně reprezentuje diagram tříd) se po upřesnění v návrhové části stane základ pro entitní třídy JDO. Doménový model reprezentuje obrázek 34.

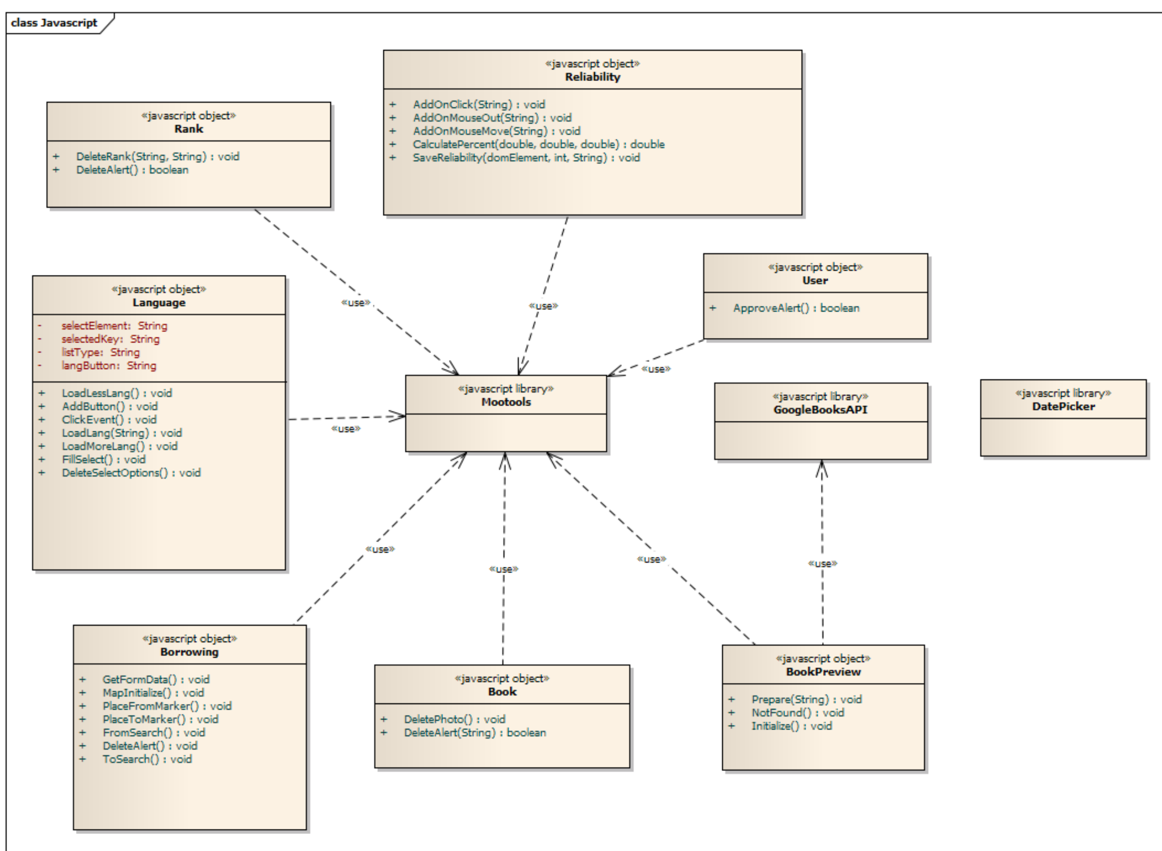


Obrázek 34 – Doménový model SVK. Zdroj: vlastní

5.2 Návrh SVK

Návrh webové aplikace v UML je problematickou záležitostí. EA v tomto případě nedokáže vygenerovat kostru frontendu pro JSP a controllery pro aplikační rámec Struts 2. Znázornění webových stránek formou diagramu tříd lze prakticky realizovat jen prostřednictvím extenzivního užívání stereotypů nebo nestandardních datových typů (např. *Select*, RadioButton). Stejně tak ani po zpřesnění doménového modelu (viz příloha A) není možné provést generování entitních tříd v JDO se všemi potřebnými metadaty. Přínos návrhové části modelu je tedy s ohledem na zvolené technologie velmi nízký. Návrhový model slouží pouze pro hrubou orientaci ve vytvořeném systému.

Obrázek 35 obahuje diagram tříd pro Javascriptovou část frontendu. Ostatní návrhové diagramy jsou k nalezení v EA modelu aplikace.



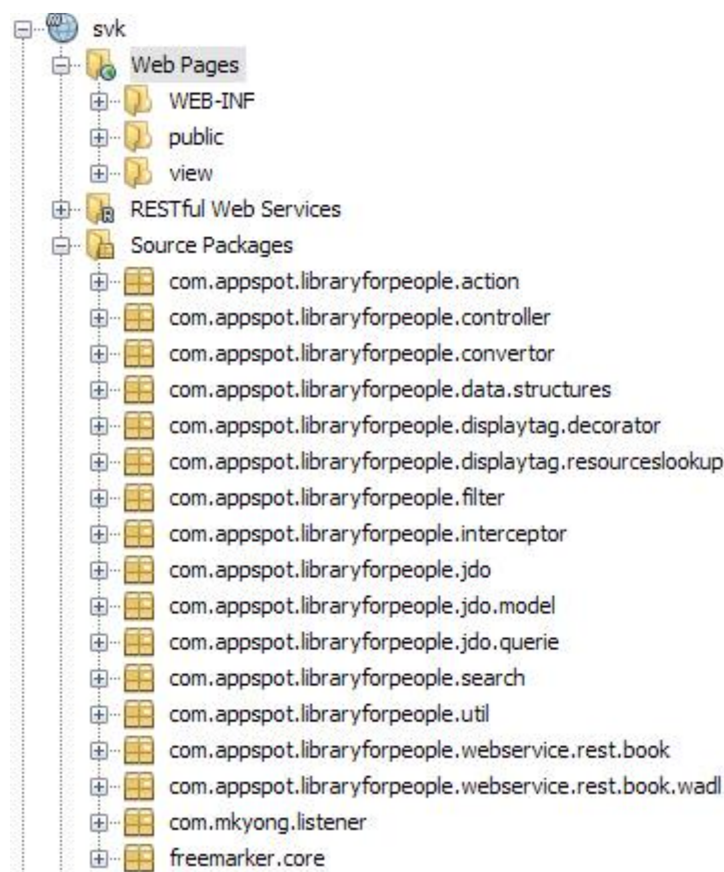
Obrázek 35 – Javascriptová část frontendu. Zdroj: vlastní

5.3 Implementace SVK

Jako vývojové prostředí je zvoleno NetBeans 7.3 s pluginem pro aplikační rámec Struts 2. NetBeans od verze 6.7 také plně podporuje Maven pro správu projektu a jeho snadné nasazení. Maven je při vývoji aplikace využíván společně se sadou skriptů pro GAE

5.3.1 Struktura projektu

Struktura projektu se nachází na obrázku 36. Následuje popis vybraných adresářů a balíčků.



Obrázek 36 – Struktura projektu SVK v NetBeans. Zdroj: vlastní

- *WEB-INF* – každá webová aplikace v Javě obsahuje tento adresář. Kromě standardního *web.xml* pro routing požadavků obsahuje v tomto případě některé další soubory:
    - *appengine.xml* – konfigurace aplikace pro GAE.
    - *queue.xml* – konfigurace front pro GAE Queue službu.
    - *tiles.xml* – konfigurace šablonovacího systému Tiles.
    - *urlrewrite.xml* – konfigurace UrlRewriteFilter pluginu pro přepisování URL adres.
- *public* – obsahuje veřejně přístupné zdroje aplikace, jako např. Javascriptové soubory, CSS, XML schémata apod.
- *view* – prezentační vrstva aplikace. Obsahuje JSP stránky.
- balíček *controller* – obsahuje Struts 2 controllery dědící třídu *SvkActionSupport*. Tato je rozšířením původní Struts třídy *ActionSupport*.
- balíček *filter* – obsahuje dvojici servlet filtrů pro nastavení kódování UTF8 v objektu typu *request* a kontrolu přihlášení uživatelů.

- balíček *interceptor* – obsahuje vybrané třídy implementující návrhový vzor *Interceptor* napojené na aplikační rámec Struts 2. *Interceptor* umožňuje vykonání kódu před nebo po zavolání Struts 2 akce.
- balíček *jdo.model* – v tomto balíčku se nacházejí entitní třídy pro JDO.
- balíček *webservice.rest.book* – obsahuje RESTful webovou službu vracející XML seznam dostupných knih k zapůjčení.

5.3.2 Aplikační rámec Struts 2

SVK využívá webový aplikační rámec Struts 2 spravovaný společností Apache Software Foundation. Struts 2 je snadno rozšiřitelný framework, který je v současné době poněkud zastíněn komplexnějším rivalem Spring. Pro aplikace malého až středního rozsahu je však vhodným řešením. V základní výbavě Struts 2 příliš neexceluje, proto jsou do něj integrovány následující pluginy:

- **Tiles** – jde o jednoduchý šablonovací systém vycházející z enginu Freemarker. Nabízí klasické funkce, které lze od šablonovacího systému očekávat. V Tiles se definují jednotlivé fragmenty webových stránek, které lze pak dle příchozího klientského požadavku skládat v celek. Výřez z konfiguračního souboru *tiles.xml* pro ukázku uvádí kód 24. Definice každého fragmentu je uvozena tagem `<definition>`. Tento může disponovat ještě atributy *template* s cestou k dané JSP stránce, případně atributem *extends* rozšiřující již definovaný fragment o nové vlastnosti.

```
<tiles-definitions>
  <definition name="layout" template="/view/layout.jsp">
      <put-attribute name="header" value="/view/header.jsp" />
      <put-attribute name="menu" value="/view/menu.jsp" />
      <put-attribute name="userBox" value="/view/user/info.jsp" />
      <put-attribute name="leftContent" value="/view/book/list.jsp" />
      <put-attribute name="rightMenu" value="/view/user-menu.jsp" />
      <put-attribute name="userInfo" value="/view/user/info-caller.jsp"/>
    </definition>

    <definition name="Book_list" extends="layout">
        <put-attribute name="leftContent" value="/view/book/list.jsp" />
    </definition>

    <definition name="Book_search" extends="layout">
        <put-attribute name="leftContent" value="/view/book/search-
        list.jsp" />
    </definition>

    <definition name="User_search" extends="layout">
        <put-attribute name="leftContent" value="/view/user/search-
        list.jsp" />
    </definition>
    ...
</tiles-definitions>
```

Kód 24 – Výřez z konfiguračního souboru tiles.xml. Zdroj: vlastní

- **Tuckey.org UrlRewriteFilter** – filtr pro přepisování URL adres podobný známému modulu mod_rewrite pro aplikační server Apache. Na základě shody s regulárním výrazem v tagu `<from>` je provedeno přesměrování (přepis) URL na tvar v tagu `<to>`. Pro integraci UrlRewriteFilteru do Struts 2 je nutné přidat dodatečné mapování do souboru *web.xml*.

```
<urlrewrite>
    <rule>
        <from>/Borrowing/pujcene/upravit/([a-zA-Z0-9\-
        _\.~:\/#\[\]@\!\$'\(\)\*\+,;=]+)[/]?$</from>
        <to
type="forward">/Borrowing_borrowEditForm.do?registerId=$1</to>
    </rule>
    <rule>
        <from>/Borrowing/pujceno/upravit/([a-zA-Z0-9\-
        _\.~:\/#\[\]@\!\$'\(\)\*\+,;=]+)[/]?$</from>
        <to
type="forward">/Borrowing_borrowedToEditForm.do?registerId=$1</to>
    </rule>
    <rule>
        <from>/Borrowing/smazat/([a-zA-Z0-9\-
        _\.~:\/#\[\]@\!\$'\(\)\*\+,;=]+)[/]?$</from>
        <to type="forward">/Borrowing_delete.do?registerId=$1</to>
    </rule>
    ...
</urlrewrite>
```

Kód 25 – Výřez z konfiguračního souboru urlrewrite.xml. Zdroj: vlastní

- **Struts 2 I18n** – plugin pro podporu jazykových mutací. Do příslušných konfiguračních souborů (každá jazyková mutace má vlastní) se zapisují dvojice *klíč=hodnota*, dle kterých lze pak v aplikaci získat příslušný řetězec zvolené jazykové mutace.

Napojení Struts 2 na GAE

Aby bylo možné aplikační rámec Struts 2 využít na platformě GAE, je třeba podnikout jistá opatření týkající se zabezpečení. GAE kontejner totiž omezuje mnohé podpůrné knihovny Struts 2. Řešením je přidání servlet listeneru, který v inicializační fázi provede volání kódu `OgnlRuntime.setSecurityManager(null)`. Tento výraz zakáže provádění kontroly oprávnění ze strany OGNL[22]. GAE má totiž vlastní bezpečnostní manager a docházelo by tak ke konfliktům.

GAE konterjer dále zakazuje použití třídy *javax.swing.tree.TreeNode*, kterou využívá šablonovací engine Freemarker, na němž je postaven systém Tiles. Řešením je v tomto případě nahrazení zmíněné třídy její neoficiální mutací od společnosti The Visigoth Software Society.

[22] OGNL (Object Graph Navigation Library) je open sourcový výrazový jazyk pro J2EE. Operuje na prezentační vrstvě a dovoluje např. volání metod Java tříd v JSP stránkách.

Struts 2 akce

Akce jsou základem aplikačního frameworku Struts 2. Obsahují aplikační logiku, starají se o validaci vstupních dat a určují, která stránka (nebo lépe šablona) zobrazí výsledky provedení akce uživateli.

Výchozí metodou v akci je *execute()*. SVK využívá volání metod dle masky nastavené v konfiguračním souboru *struts.xml*, jak ukazuje kód 26.

```
<action name="*_*"
class="com.appspot.libraryforpeople.controller.{1}Action" method="{2}">
    <interceptor-ref name="gaeFileUploadInterceptor" />
    <interceptor-ref name="basicStack" />
    <interceptor-ref name="ACLInterceptor" />
    <interceptor-ref name="userBannedInterceptor" />
    <interceptor-ref name="i18n" />
    <interceptor-ref name="redirectMessage" />
    <interceptor-ref name="validation">
        <param name="excludeMethods">list,input</param>
    </interceptor-ref>
    <interceptor-ref name="workflow">
        <param name="excludeMethods">list,input</param>
    </interceptor-ref>
    <result name="success" type="tiles">{1}_{2}</result>
    <result name="redirect" type="redirect">${redirectUrl}</result>
    <result name="input" type="tiles">{1}_{2}</result>
    <result name="empty" type="httpheader">
        <param name="status">200</param>
    </result>
    <param name="bufferSize">1024</param>
</action>
```

Kód 26 – Výřez z konfiguračního souboru urlrewrite.xml. Zdroj: vlastní

Akce se definuje tagem `<action>`. Atribut *name* obsahuje buď doslovný název akce podle URL, nebo lze využít tzv. *wildcard*. Ve Struts 2 *wildcard* je možné psát pouze zástupné znaky '*' a '?'. Namapování na vybranou metodu akční třídy se provádí dle parametrů ve složených závorkách.

V příkladu v kódu 26 je dále uvedena sada interceptorů, z nichž některé se nacházejí již v základu Struts 2 (např. interceptor *validation* pro validaci vstupních dat). Ostatní interceptory (*ACLInterceptor*, *GetFileUploadInterceptor*, *RedirectMessage* a *userBannedInterceptor*) jsou doprogramovány a zajišťují různou funkcionalitu od řízení práv uživatelů po nahrání obrázku na server. Příklad zdrojového kódu jednoho z interceptorů se nachází v příloze B.

Do definice akce lze dále zanořit libovolný počet tagů `<result>`. Jedná se o výsledek, který vrací metoda akční třídy po jejím dokočení. Na základě typu výsledku je pak provedeno předání kontroly. Je-li jako typ uveden *tiles*, řízení je předáno šablonovacímu systému, který provede zobrazení výsledku. Typ *redirect* zajistí přesměrování na cílové URL (lze tak např. docílit řetězení akcí) a konečně *httpheader* slouží pro nastavení hodnot v HTTP hlavičce dle příslušného parametru.

5.3.3 Databázová vrstva

Implementace databázové vrstvy je zajištěna vytvořením entitních tříd v balíčku *com.appspot.libraryforpeople.jdo.model*. Metadata entitních tříd jsou obsažena v anotacích, jejichž význam je popsán v kapitole 3.4.1. Vývojové prostředí NetBeans obsahuje nástroj pro generování entitních tříd ze snapshotu databáze. Vzhledem k využití GAE datastore však tento přístup nemohl být aplikován, protože NetBeans neumožňují sejmutí snapshotu z tohoto typu datového uložiště. Entitní třídy jsou proto naprogramovány manuálně.

5.3.4 Výsledná podoba aplikace

Výslednou podobu vyvinuté aplikace zachycuje ukázka na obrázku 37.

Obrázek 37 – Ukázka z vyvinuté aplikace SVK. Zdroj: vlastní

5.3.5 Nasazení aplikace do prostředí GAE

Aplikace určená pro GAE musí v adresáři *WEB-INF* kromě standardního *web.xml* obsahovat také konfigurační soubor *appengine-web.xml*. Tento soubor povinně obsahuje:

- identifikátor aplikace včetně označení poslední verze zdrojového kódu,
- cestu ke statickým souborům.

Dále je možné provést celou řadu nastavení od logování po vyjmutí vybraných souborů z *deployingu* aplikace na GAE. Konfigurační soubor *appengine-web.xml* pro SVK je obsažen v kódu 27.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>libraryforpeople</application>
    <version>1</version>

    <precompilation-enabled>true</precompilation-enabled>

    <system-properties>
        <property name="java.util.logging.config.file" value="WEB-
        INF/logging.properties"/>
        <property name="DEFAULT_ENCODING" value="UTF-8" />
    </system-properties>

    <resource-files>
        <include path="/public/**" />
    </resource-files>

    <sessions-enabled>true</sessions-enabled>
    <threadsafe>true</threadsafe>
</appengine-web-app>
```

Kód 27 – Konfigurační soubor appengine-web.xml pro SVK. Zdroj: vlastní

Nahrání aplikace na server probíhá prostřednictvím GAE pluginu pro Maven. Plugin definuje hned několik vlastních akcí (v Maven známé pod termínem *goals*), z nichž nejpoužívanější z nich sumarizuje tabulka 12.

Tabulka 12 – Přehled Maven goals pro GAE. Zdroj: [23]

Goal	Popis
appengine:rollback	Přeruší probíhající aktualizaci a provede návrat k předchozí verzi.
appengine:update	Vytvoří nebo aktualizuje GAE aplikaci.
appengine:update_indexes	Provede aktualizaci GAE datastore indexů dle konfiguračního souboru *datastore-indexes.xml*. Restrukturalizace příslušných indexových tabulek je obvykle časově náročná.
appengine:update_queues	Provede aktualizaci GAE front na základě konfiguračního souboru *queue.xml*.
appengine:vacuum_indexes	Odstraní z aplikace všechny nepoužívané indexy.

5.3.6 Testování aplikace

Pro testování SVK nebyla použita žádná z formálních metod. Funkčnost aplikace byla průběžně testována zadavatelem, který posléze odsouhlasil její finální podobu.

Zvažován byl nástroj Selenium, který dokáže automatizovat činnost webového prohlížeče a je tak vhodným kandidátem pro provádění testů. Z důvodu nedostatku času a ochotě zadavatele testovat vyvíjenou aplikaci bylo však od tohoto přístupu nakonec upuštěno.

6 Závěr

Hlavním přínosem práce je vytvoření systému veřejné knihovny (SVK) na platformě GAE. Výsledná podoba aplikace je schválena zadavatelem a nabízí veškerou požadovanou funkcionalitu. Výstupem je kromě zdrojových kódů aplikace taktéž detailně zpracovaný model v CASE nástroji Enterprise Architect (EA). Tento by měl pomoci při možném budoucím rozšíření SVK.

Vývoj aplikace byl průběžně monitorován ze strany zadavatele, přičemž na obdržené připomínky a nalezené chyby bylo vždy adekvátně reagováno. Využití GAE se ukázalo i přes značné benefity (zejména z hlediska nízkonákladového provozu a možnosti využití nabízených služeb) jako mírně problematické. Hlavním nedostatkem byl experimentální status u řady nabízených služeb a z toho plynoucí časté modifikace API ze strany společnosti Google. Tato skutečnost se promítla v nutnosti častých aktualizací zdrojového kódu v reakci na prováděné změny.

Korektní implementace aplikační logiky využívající GAE datastore byla jednou z časově nejnáročnějších částí projektu. Tato technologie skýtá řadu nepříjemných omezení. Její principy jsou velmi vzdálené od programování pro klasické relační databáze. Jako příklad lze uvést povinné vytváření indexů pro veškeré JDOQL dotazy používané v aplikaci. Byť se tento požadavek může zdát na první pohled přehnaný, nutí alespoň vývojáře přemýšlet o svém kódu na úrovni optimalizace dotazů.

Dále je nutné připomenout, že produktové portfolio společnosti Google je dlouhodobě nestabilní. Z nabídky průběžně mizí řada služeb, jejichž uživatelská základna se podle vyjádření Google rok od roku zmenšuje nebo nesplnily původní očekávání. Časté jsou také změny parametrů služeb. Tento fakt by mohl mít v budoucnu negativní dopad na provoz SVK.

Platforma GAE nemusí být pro vývoj webových aplikací vždy vhodnou volbou. Pro aplikace závislé na rychlém přístupu k datům uložených v množství různých entit není GAE vhodné. Ke spojování entit totiž dochází na úrovni aplikační logiky, což s sebou obvykle přináší vyšší režii. Platforma GAE je nejvhodnější pro aplikace pracující s takovými objemy dat, které nelze efektivně uložit do klasických relačních databází. Daní za to je nutnost přizpůsobit návrhovou část aplikace specifickým potřebám GAE datastore.

Použité zdroje

[1] **Mohamed, Arif.** A history of cloud computing. [Online] 2009. [Citace: 15. 7. 2013.] http://www.computerweekly.com/feature/A-history-of-cloud-computing.

[2] **Salesforce.** Salesforce. [Online] 2013. [Citace: 15. 7. 2013.] http://www.salesforce.com/eu/.

[3] **Mell, Peter a Grance, Timothy.** The NIST Definition of Cloud Computing. [Online] 2011. [Citace: 15. 7. 2013.] http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[4] **Sosinsky, Barrie.** *Cloud Computing Bible.* Indiana : Wiley Publishing, Inc., 2011. ISBN 978-0-470-90356-8.

[5] **Draalin.** IaaS Overview. *What is the Cloud.* [Online] [Citace: 15. 7. 2013.] http://whatisthecloud.ca/iaas/.

[6] **ZOHO Creator.** Platform as a Service. *ZOHO Creator.* [Online] [Citace: 15. 7. 2013.] https://www.zoho.com/creator/paas.html.

[7] **TeamWox.** What Is Software as a Service (SaaS) and What Are Its Advantages? *Online Team Collaboration Software.* [Online] 2010. [Citace: 15. 7. 2013.] http://www.teamwox.com/en/groupware/articles/60/saas-online-collaboration-system.

[8] **Velte, Anthony T., Velte, Toby T. a Elsenpeter, Robert.** *Cloud Computing: Praktický průvodce.* Brno : Computer Press, a.s., 2011. ISBN 978-80-251-3333-0.

[9] **Ames, Josh.** Cloud Computing vs Virtualization: There Is A Difference. *appcore.* [Online] 2012. [Citace: 16. 7 2013.] http://blog.appcore.com/blog/bid/128515/Cloud-Computing-vs-Virtualization-There-Is-A-Difference.

[10] **Churý, Lukáš.** Základy XML webových služeb. [Online] 2008. [Citace: 17. 7. 2013.] http://programujte.com/clanek/2005081704-zaklady-xml-webovych-sluzeb/.

[11] **Kosek, Jiří.** Komunikační infrastruktura. *Využití webových služeb a protokolu SOAP při komunikaci.* [Online] [Citace: 18. 7. 2013.] http://www.kosek.cz/diplomka/html/websluzby.html.

[12] **Jewell, Tyler a Chappell, David.** *Java Web Services.* místo neznámé : O'Relly, 2002. str. 276. ISBN 0-596-00269-6.

[13] **w3schools.com.** *SOAP Example.* [Online] [Citace: 18. 7. 2013.] http://www.w3schools.com/soap/soap_example.asp.

[14] **Malý, Martin.** REST: architektura pro webové API. *Zdroják.cz.* [Online] 2009. [Citace: 19. 7. 2013.] http://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/.

[15] **W3C.** WSDL Documents. *W3Schools.com.* [Online] [Citace: 19. 7. 2013.] http://www.w3schools.com/wsdl/wsdl_documents.asp.

[16] **W3C.** Web Application Description Language. [Online] 2009. [Citace: 2013. 7. 18.] http://www.w3.org/Submission/wadl/.

[17] **Hurwitz, Judith, a další, a další.** *Service Oriented Architecture for dummies.* Indiana : Wiley Publishing, Inc., 2009. ISBN: 978-0-470-52549-4.

[18] **Google.** Google Apps pro firmy. [Online] [Citace: 20. 7. 2013.] http://www.google.cz/intx/cs/enterprise/apps/business/.

[19] **Bates, Bert, Sierra, Kathy a Basham, Bryan.** *Head First Servlets and JSP.* místo neznámé : O'Reilly, 2004. ISBN 0596005407.

[20] **Tost, Andre.** Integrating data at run time with XSLT style sheets. *developerWorks.* [Online] 2002. [Citace: 30. 7. 2013.] http://www.ibm.com/developerworks/library/x-runxslt/index.html.

[21] **apache.org.** Java Data Objects. [Online] [Citace: 31. 7. 2013.] http://db.apache.org/jdo/.

[22] **DataNucleus.org.** DataNucleus. [Online] [Citace: 1. 8. 2013.] http://www.datanucleus.org/.

[23] **Google.** Google App Engine. *Google Developers.* [Online] 2013. [Citace: 20. 7. 2013.] https://developers.google.com/appengine/.

[24] **Roche, Kyle a Douglas, Jeff.** *Beginning Java™ Google App Engine.* New York : Apress, 2009. ISBN 978-1-4302-2553-9.

[25] **Sanderson, Dan.** *Programming Google App Engine.* Sebastopol : O’Reilly Media, Inc., 2010. ISBN: 978-0-596-52272-8.

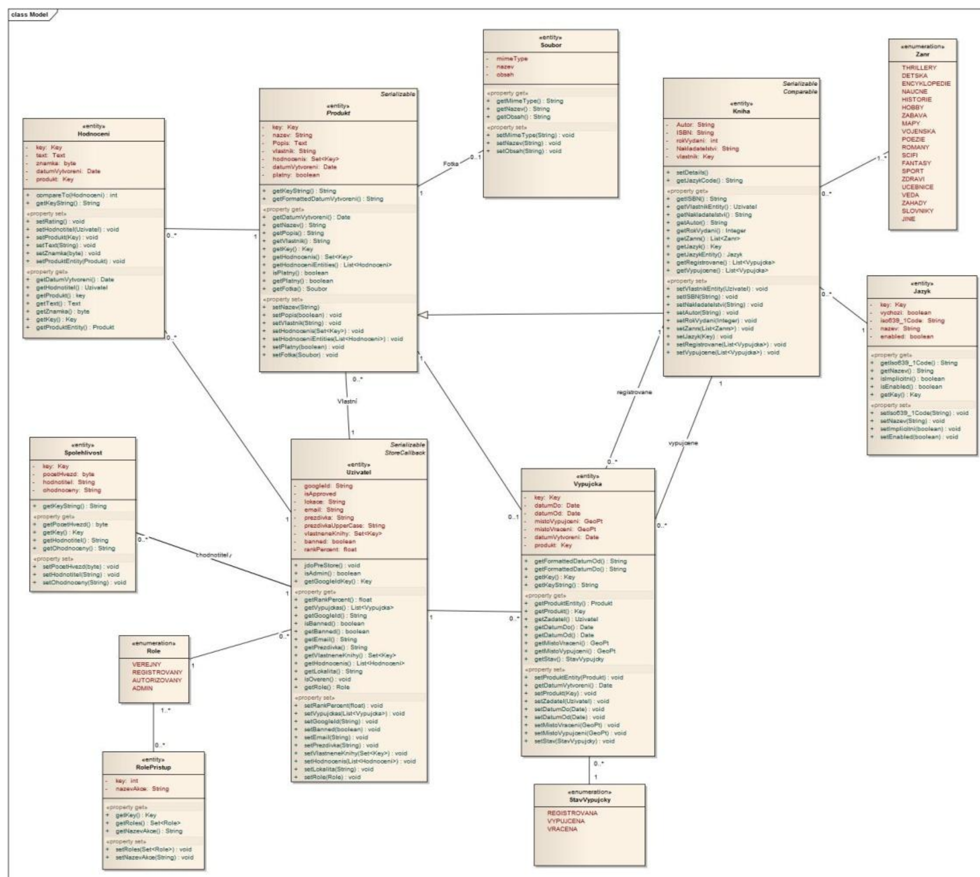
[26] **Ghemawat, Sanjay, Gobioff, Howard a Leung, Shun-Tak.** The Google File System. [Online] 2003. [Citace: 24. 7. 2013.] http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/cs//archive/gfs-sosp2003.pdf.

[27] **Chang, Fay, a další.** Bigtable: A Distributed Storage System for Structured Data. [Online] 2006. [Citace: 24. 7. 2013.] http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/cs//archive/bigtable-osdi06.pdf.

Příloha A – Vybrané artefakty modelu SVK

	Knihy::R008 - SVK	Knihy::R009 - SVK	Knihy::R010 - SVK	Knihy::R011 - SVK	Knihy::R012 - SVK	Knihy::R013 - SVK	Knihy::R014 - SVK	Knihy::R015 - SVK	Knihy::R016 - SVK	Knihy::R017 - SVK	Knihy::R025 - SVK	Knihy::R026 - SVK	Knihy::R028 - SVK	Knihy::R029 - SVK	Uživatelé::R001 - S	Uživatelé::R002 - S	Uživatelé::R003 - S	Uživatelé::R004 - S	Uživatelé::R005 - S	Uživatelé::R006 - S	Uživatelé::R007 - S	Uživatelé::R024 - S	Uživatelé::R027 - S	Uživatelé::R030 - S	Výpůjčka::R018 - S	Výpůjčka::R019 - S	Výpůjčka::R020 - S	Výpůjčka::R021 - S	Výpůjčka::R022 - S	Výpůjčka::R023 - S
Hodnocení::UC007 - Ohodnotit knihu											↑																			
Hodnocení::UC008 - Smazat hodnocení knihy												↑																		
Hodnocení::UC009 - Ohodnotit uživatele																					↑									
Knihy::UC010 - Zobrazit seznam knih	↑	↑			↑			↑																						
Knihy::UC011 - Zobrazit detail knihy										↑																				
Knihy::UC012 - Vyhledat knihu									↑																					
Knihy::UC013 - Seřadit seznam knih							↑																							
Knihy::UC014 - Přidat knihu			↑			↑																								
Knihy::UC015 - Náhled knihy dle ISBN				↑																										
Knihy::UC016 - Editovat knihu			↑			↑																								
Knihy::UC021 - Získat seznam knih pomocí webové služby													↑																	
Knihy::UC022 - Vyhledat knihy pomocí webové služby														↑																
Správa účtů::UC001 - Vytvořit SVK účet															↑				↑	↑										
Správa účtů::UC002 - Ověření uživatele																				↑										
Správa účtů::UC003 - Autorizovat uživatele																↑														
Správa účtů::UC004 - Udělit ban																		↑												
Správa účtů::UC005 - Editovat uživatelské účty																	↑													
Správa účtů::UC023 - Zobrazit seznam uživatelů																								↑						
Správa účtů::UC024 - Vyhledat uživatele																							↑							
Uživatelská komunikace::UC006 - Odeslat zprávu																						↑								
Výpůjčky::UC017 - Zažádat o výpůjčku																									↑	↑			↑	
Výpůjčky::UC018 - Zobrazit seznam výpůjček																											↑			
Výpůjčky::UC019 - Změnit stav výpůjčky																												↑		
Výpůjčky::UC020 - Zrušit žádost o výpůjčku																														↑

Obrázek 38 – Matice pokrytí funkčních požadavků případy užití. Zdroj: vlastní



Obrázek 39 – Datový model SVK. Zdroj: vlastní

Příloha B – Zdrojový kód interceptoru ACLInterceptor.java

```
/**
 * Interceptor zajišťující ověření práv pro vykonání akce Struts2
 *
 * @author Martin Horák
 */
public class ACLInterceptor implements Interceptor {

    private static final Logger log =
        Logger.getLogger(ACLInterceptor.class.getName());
    private static final String IMPORT_ROLES_ACTION =
        "Import_roleAccess";
    private static final String USER_NOT_APPROVED_URL =
        "/Error_notapproved.do";
    private static final String ERROR_PAGE_SUFFIX = "Error";

    @Override
    public void destroy() {
    }

    @Override
    public void init() {
    }

    @Override
    public String intercept(ActionInvocation ai) throws Exception {
        String currentActionName = ai.getInvocationContext().getName();


    Map<String, Object> session =
        ai.getInvocationContext().getSession();
    HttpServletResponse response = (HttpServletResponse)
        ai.getInvocationContext().get(StrutsStatics.HTTP_RESPONSE);
    HttpServletRequest request = (HttpServletRequest)
        ai.getInvocationContext().get(StrutsStatics.HTTP_REQUEST);
    Uzivatel uzivatel = (Uzivatel) session.get("user");


    RolePristup ra = null;
        JdoDBConnector jdoDBConnector = null;
        try {
            jdoDBConnector = new JdoDBConnector();
            RolePristupQuery raq = new
RolePristupQuery(jdoDBConnector.getPm());
            jdoDBConnector.getTransaction().begin();
            List<RolePristup> roleAccesses =
raq.getRoleAccess(currentActionName);
            jdoDBConnector.getTransaction().commit();
            if (roleAccesses.size() > 1) {
                throw new RuntimeException("Akci " + currentActionName +
" odpovídá více než jeden záznam práv v DB!");
            }
            if (roleAccesses.size() > 0) {
                ra = roleAccesses.get(0);
            } else {
                log.log(Level.CONFIG, "Nenalezen z\u00e1znam pr\u00e1v
pro akci: {0}", currentActionName);
            }
```

```
        } catch (Exception e) {
            log.log(Level.SEVERE, "Chyba při získávání rolí k akci: " +
currentActionName, e);
        } finally {
            if (jdoDBConnector != null) {
                jdoDBConnector.disposePM();
            }
        }

        //Pokud neni v db nastaven pristup pro import pristupu, proved
        //import pristupu
        if (IMPORT_ROLES_ACTION.equals(currentActionName) && ra == null)
{
            return ai.invoke();
        }

        //errorove stranky pristupne bez opravneni
        if (ERROR_PAGE_SUFFIX.startsWith(currentActionName)) {
            return ai.invoke();
        }

        //pokud se podarilo ziskat prava k akci
        if (ra != null) {
            //akce je verejna pro vsechny
            if (ra.getRoles().contains(Role.VEREJNY)) {
                return ai.invoke();
            }

            //uzivatel neni nalogovan
            if (uzivatel == null) {
                UserService userService =
UserServiceFactory.getUserService();

response.sendRedirect(userService.createLoginURL(request.getRequestURI())
);
            }
            //uzivatel ma pravo pristupu pro danou akci
            if (ra.getRoles().contains(uzivatel.getRole())) {
                //Pokud je vyzadovano overeni uzivatele adminem
                if (ra.getRoles().contains(Role.AUTORIZOVANY)) {
                    //pokud je uzivatel overen, proved, jinak presmeruj
                    //na error overeni
                    if (uzivatel.isOveren()) {
                        return ai.invoke();
                    } else {
                        response.sendRedirect(USER_NOT_APPROVED_URL);
                    }
                } else {
                    return ai.invoke();
                }
            }
        }
        return null;
    }
}
```

Kód 28 – Interceptor ACLInterceptor.java. Zdroj: vlastní

Příloha C – Adresářová struktura DVD

- praktickaCast
    - dokumentace
        - spusteniProjektu
        - SVKjavadoc
    - EAUmlModel
    - zdrojoveKody
- textovaCast