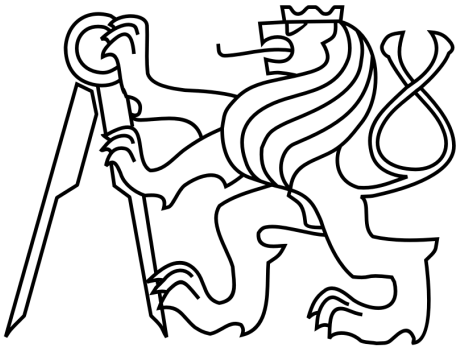
České vysoké učení technické v Praze
Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE



Bc. Jan Bobisud

Automatické generování RESTful rozhraní

Katedra počítačové grafiky a interakce

Vedoucí diplomové práce: Ing. Karel Čemus
Studijní program: Otevřená informatika (magisterský)
Studijní obor: Softwarové inženýrství

Praha 2014

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Abstrakt:

Stávající implementace pro tvorbu webových služeb umožňuje definovat zdroje a jejich logiku při zpracování HTTP dotazů. Některé implementace sice obsahují výchozí logiku pro práci se zdroji, které jsou mapovány na objekty z aplikačního modelu, ale definice zdroje je explicitně nutná pro každý zdroj zvlášť. Případné rozdíly ve výchozí implementaci zpracování zdroje je ale i tak nutné zanášet přímo do kódu. Oba zmíněné problémy tak znepříjemňují programátorovi život.

Alternativní řešení by mohlo generovat zdroje automaticky, pokud by se našla závislost mezi zdroji a objekty z domény aplikace, resp. aplikačního modelu, který tuto doménu reprezentuje. A s univerzální logikou pro zpracování dotazů, které by ovšem bylo schopné reagovat na rozdílné požadavky zdrojů, by mohlo vyřešit zmíněné problémy a usnadnit tak programátorovi jeho práci.

Klíčová slova: Model Driven Development, REST, HTTP, adaptivní rozhraní

Abstract:

The current implementations of building Web Services allow you to define resources and their logic for processing HTTP requests. Although some implementations contain a default logic to work with the resources that are mapped to objects in the application model, but the definition of resources is explicitly required for each single resource. Any differences in the default implementation of the processing of resources are necessary to be put directly into the code. Both of these problems can ruin the programmer's life.

An alternative solution could thus generate resources automatically, if the dependence between resources and objects of the application's domain will be found, respectively application's model that represents this domain. A universal logic for processing requests, which, however, will be able to respond to the different requirements of resources, could solve the aforementioned problems and to simplify the programmer's work.

Keywords: Model Driven Development, REST, HTTP, adaptive interface

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

1. Úvod

Problém automaticky generovaného RESTful rozhraní naráží na omezení implementace existujích řešení, která vyžadují explicitní definici zdrojů. Implementace se pak liší v míře podporované logiky pro zpracování HTTP dotazů. Tuto logiku je možné upravovat v závislosti na zdroji a požadavků na něj kladených dle specifikace aplikace. Nevýhodou při implementaci specifických rozdílů je nutnost zasahovat do kódu logiky, mixování těchto prvků se může projevit zhoršenou přehledností a údržbou kódu, případně i vyšším výskytem chyb.

Proto, abychom byli schopni automaticky generovat RESTful rozhraní, musíme vycházet z nějaké předlohy. Každá aplikace má svoji doménu, tedy jakousi oblast zájmu, ve které jsou řešeny dané problémy. Ta by tak mohla být vhodným kandidátem stát se předlohou pro generování rozhraní, ale je nutné napřed nalézt společné rysy mezi těmito elementy.

Pro tvorbu webové služby je důležité pochopit její stavební kameny v prostředí internetu, tedy protokol HTTP, který zajišťuje komunikaci mezi klientem a serverem, a architekturu orientovanou na zdroje (ROA), která aplikuje architektonický styl - REST s využitím protokolu HTTP. Pro vhodný návrh bude nutné nastudovat základní principy modelem řízeného vývoje a aplikace generativního programování, které řeší některé neduhy objektově-orientovaného programování.

Budeme-li mít definovanou předlohu pro zdroje a znalost postupů při tvorbě webových služeb, můžeme přejít k identifikaci logiky, kterou budeme aplikovat při zpracování dotazů. Logika bude vycházet z požadavků uživatelů při interakci s webovými službami. Pro inspiraci bude vhodné prozkoumat také stávající knihovny a vymezit jejich silné a slabé stránky. Univerzálnost řešení bude klíčová - zpracovávat dotazy jednotným způsobem totiž zjednodušší implementaci služby. Zde bude důležité umožnit upravovat a ohýbat výchozí chování podle potřeb jednotlivých zdrojů takovým způsobem, který nebude mixovat logiku s nastavením pro odlišné chování. Tím, že bude logika oddělena od nastavení, dojde ke zpřehlednění implementace logiky a nastavení bude možné dodávat např. pomocí konfiguračních souborů, které budou specifikovat danou aplikaci (službu).

S existující výchozí logikou je možné implementovat zpracování dotazu v konkrétním, i když ukázkovém, kontextu. Cílem práce totiž není kompletní implementace všech možností, ale vybrat takové řešení, aby mohla vzniknout jednoduchá, ale zároveň plně funkční aplikace, kterou budeme moci ověřit navržený způsob zpracování dotazů pro automaticky generované rozhraní.

2. Analýza

2.1 Webová služba

Touto službou rozumíme způsob komunikace mezi počítači v síťovém prostředí [8]. Bereme v potaz to, že počítače samy nemyslí, ale dělají pouze to, co jim přikáží lidé. V této komunikaci mohou lidé vystupovat jako jednotlivci, či organizace. Na jedné straně jsou tak poskytovatelé služeb a na straně druhé její konzumenti. Poskytovatelé nabízejí své služby pomocí webového serveru, a naopak konzumenti přistupují k těmto službám s využitím webového klienta.

Web [35] je definován jako sada protokolů, která umožňuje komunikaci mezi klientem a serverem v počítačové síti Internet [20]. Mezi nejdůležitější stavební kameny patří (1) hypertextové dokumenty, které umožňují klientovi procházet Web pomocí odkazů, (2) jednoznačné identifikátory dokumentů na serveru - URL, a (3) komunikační protokol HTTP, který zajišťuje přenos dokumentů ze serveru na klienta. Komunikace začíná na straně klienta, který iniciuje dotaz na server pomocí URL, server odpoví klientovi odpovídajícím dokumentem.

V dnešní době nabízí Web nepřeberné množství služeb - od „obyčejných“ statických stránek až po sofistikované webové aplikace - kdy např. banka poskytuje přístup klientům ke svým účtům pomocí webového klienta, klienti banky tak nemusejí chodit kvůli běžným operacím, jako je převod peněz na jiný účet, fyzicky na pobočku. Každá služba tak plní svůj účel, pro který byla vytvořena. Oblast, do které tento účel spadá, se nazývá „doména“ a objekty z této domény (např, bankovní účet, klient, transakce) tvoří tzv. model služby (aplikace).

2.1.1 Vlastnosti služeb

Bez ohledu na zvolenou architekturu potřebují služby disponovat určitou sadou vlastností. Tyto vlastnosti vycházejí z principu komunikace (zaměření dané služby), ale většina vlastností je společná pro všechny typy komunikace. Níže uvedené vlastnosti jsou identifikovány na konkrétním případě webové služby mezi bankou a jejím klientem.

Naše fiktivní bankovní služba by nemohla fungovat bez zabezpečeného přístupu k osobnímu účtu klienta. Data (např. zůstatek na účtě) svázaná s účtem jsou citlivými údaji určenými výhradně pro majitele onoho účtu a banku, ke kterým nesmí mít přístup třetí osoby. To samé platí pro operace, které smí provádět pouze klient, např. zmíněný převod peněz na cizí účet. Ale v bance existují i operace, které naopak jsou dostupné třetím osobám, např. příjem peněz z cizího účtu je naprosto v pořádku, pokud osoba získá číslo účtu klienta banky, na který mají být peníze zaslány.

První vlastností webových služeb je jejich bezpečnost.

Má-li náš klient zajištěn bezpečný přístup ke svým datům, může tak s nimi nějak pracovat, např. prohlížením historie transakcí. Tyto transakce mohou být

zapsané v tabulce, či reprezentovány grafem s časovou osou apod. Klient požádá banku o historii transakcí v určitém formátu dat, ve kterém chce s daty dále pracovat. Je na straně serveru, jestli požadavku vyhoví - může se totiž i stát to, že danému formátu nebude rozumět. Pokud však rozumí, převede data do požadovaného formátu a zašle klientovi. Data je nutné převádět z tohoto důvodu, že server pracuje se svým interním formátem daným jeho architekturou a požadavky na systém, které se nemusí nutně shodovat se stranou klienta.

Proces převodu dat do určitého formátu se nazývá serializace.

Z principu komunikace mezi klientem a bankou je zřejmé, že jsou data uložena na straně serveru. Klient se nestará o to, jakým způsobem jsou data uložena, on pouze očekává, že data dostane na požádání - v případě historie transakcí to může být kompletní historie od zavedení účtu. Na straně serveru se tak postupem času „hromadí“ různě velké objemy dat v závislosti na jejich povaze, ale ať už jsou data malá či velká, je nutné zajistit uživateli jejich persistenci. Data v operační paměti jsou pouze dočasná s přístupem el. energie, je proto nutné odkládát data na trvalé médium (např. pevné disky), je pak na serveru, jaké technologie k tomu využije (relační databáze, apod.).

Jelikož uživatelé mohou přistupovat ze zařízení, která mají odlišně kvalitativní připojení (např. rychlost, odezva), zejména pak mobilní telefony mohou trpět zhoršeným spojením, je snahou snížit komunikaci mezi klientem a serverem na minimum. V dané službě mohou existovat neměnná (statická) data, která po prvním stažení klientem není nutné opětovně stahovat. Jednou staženou historii transakcí za měsíc určitě není nutné stahovat znovu. Tomuto způsobu snížení požadavků na server se říká kešování.

Podobný princip kešování je použit i na straně serveru, kdy příprava dat pro klienta může být časově (ale i pamětově) náročná operace, která zatěžuje server a mohlo by tak dojít ke zpomalení služby v závislosti na počtu dotazujících se klientů. Je tedy žádoucí si výsledná data ukládat do mezipaměti, ze které budou zasílána klientovi.

Toto byla určitá podmnožina vlastností služeb, které nejsou specifické k jedné dané službě, ale bývají požadovány napříč službami. Je však nutné vzít v potaz to, že tyto vlastnosti se mohou navzájem kombinovat, a mohou tak vznikat různá nastavení pro jednotlivé objekty. Nyní se podívejme, jakými způsoby je možné implementovat služby.

2.1.2 Možnosti implementace

Implementace webových služeb je značně odlišná v závislosti na použité technologii, ale v [1] jsou identifikovány celkem tři typické architektury:

1. RESTful architektura (orientovaná na zdroje),
2. RPC stylem inspirované,
3. kombinace (1) a (2).

RESTful implementace splňuje požadavky definované architektonickým stylem v [5]: (1) klient-server, (2) bezestavovost, (3) kešování dat, (4) uniformní rozhraní, (5) vrstvený systém, (6) kód na vyžádání (nepovinné omezení), a v co nejvyšší míře využívá možností protokolu HTTP. V [1] je zmíněna služba implementující tento architektonický styl S3 od Amazonu [36].

RPC inspirované služby odpovídají způsobu komunikace definované specifikací pro vzdálené volání metod [26]. Využívají tak URL pro nalezení dané metody a tělo HTTP dotazu pro vlastní obálku, která nese potřebné parametry. Existují implementace jako např. XML-RPC [27], či SOAP [18] (do verze 1.1).

Kombinovaným způsobem označuje takové webové služby, které se při získávání dat tváří jako RESTful, ale přitom zanášejí jak informaci o použité metodě do samotné URL. Ukázkovou službou využívající tento způsob je např. `https://delicious.com` [37].

2.2 Služby orientované na zdroje

Jelikož je REST pouze architektonický styl nezávislý na konkrétní platformě, je nutné dodat architekturu na konkrétní platformu, v prostředí Webu je touto architektrurou 'Resource-Oriented Architecture' (ROA) definovaná v [1].

Jak už z názvu vyplývá, základem této architektury je zdroj. Zdroje představuje objekty z oblasti zaměření dané služby. V bance by tak zdrojem mohl být účet, transakce za rok 2010, trvalé příkazy, apod. Tyto zdroje jsou přístupné přes jejich identifikátory - URL, formát adres není pevně dán, ale pro zmíněné zdroje by mohly adresy vypadat následovně:

- 'https://example.com/accounts/123',
- 'https://example.com/accounts/123/transactions/2010',
- 'https://example.com/accounts/123/standing-payments'.

Každý zdroj má kromě svého identifikátoru i sadu vlastností, které ho reprezentují - např. zdroj účet má vlastnosti jako číslo, datum založení, aktuální zůstatek, majitele, transakce, apod.

Nyní, když víme, co je zdrojem, můžeme se přesunout na požadavky kladené touto architekturou.

2.2.1 Adresovatelnost

Proč chceme, aby byly služby adresovatelné? Představme si, že naše bankovní služba není adresovatelná, naši klienti mají k dispozici pouze adresu serveru, např. 'https://example.com'. Chtějí-li se dostat ke svému účtu, museli by na stránce vyplnit formulář s číslem k účtu a tento formulář odeslat na server, který by vrátil stránku pro daný účet. Ale díky tomu, že banka zpřístupnila účty pod adresou 'https://example.com/accounts/XXX', kde 'XXX' je číslo účtu, klient banky může rovnou přistupovat ke svému účtu díky této adrese zdroje.

2.2.2 Bezstavovost

Bezstavový systém zabraňuje zmatení klienta při dotazování se na server. Máli např. klient za dobu svého používání účtu několik stovek či tisíc transakcí, bude pro něj výhodné získávat seznam těchto transakcí stránkovaně. Se stavovým systémem by si musel server pamatovat aktuální stránku transakcí a dotaz na server 'GET/accounts/123/transactions' by tak vracel transakce ze stránky, na které byl klient posledně, z tohoto dotazu není jasné, na které stránce ale skončil minule, nebo na kterou se chce dostat. Pokud ale využijeme bezstavový systém, musíme veškeré potřebné informace připojit k dotazu. V našem příkladě by se tak dotaz mohl upravit takto: 'GET/accounts/123/transaction/?page=1&limit=10' - což znamená, že klient chce získat první stránku s deseti transakcemi. Dalším důvod pro použití bezstavového systému je ten, že snadněji umožňuje škálovatelnost serveru a rozdělit tak zátěž na více strojů, tím, že si server nedrží stav aplikace, odpadá tak režie spojená se správou stavu.

2.2.3 Propojitelnost

Objekty v reálném světě jsou propojené, stejně tak jako jejich modely ve službách - k bankovnímu účtu klienta jsou svázané transakce, trvalé příkazy, apod. Propojitelnost umožňuje jednoduché procházení těchto souvisejících zdrojů díky odkazům obsaženým v reprezentaci daného zdroje. Získáme-li např. reprezentaci zdroje účtu, měl by obsahovat odkazy na související zdroje, tedy např. proběhlé transakce, apod. Reprezentace, která obsahuje odkazy (tzv. 'hyperlinky'), se nazývá 'hypertext' [38], na Webu existuje několik jazyků pro popis těchto reprezentací, mezi nejpoužívanější patří jazyky z rodiny SGML (HTML, XML a XHTML).

2.2.4 Jednotné rozhraní

Jednotné rozhraní umožňuje klientům pracovat se zdroji stejným způsobem, i když význam jednotlivých operací se může lehce lišit. Co může klient s takovým zdrojem jako je 'trvalý příkaz' dělat? Může si vypsat seznam zadaných trvalých příkazů, prohlédnout podrobnosti jednotlivých příkazů, vytvořit nový příkaz, upravit, či ho celý smazat. A co třeba s transakcemi? Banka mu zřejmě nedovolí upravovat či mazat proběhlé transakce, ale vypsat jejich seznam či zadat (vytvořit) novou transakci jistě ano. A takto bychom mohli pokračovat s každým zdrojem. Můžeme si tak všimnout čtyř základních operací, které chceme se zdroji provádět: (1) získat zdroje, (2) vytvořit nový zdroj, (3) upravit existující zdroj, a (4) smazat zdroj. Tato čtveřice — označována zkratkou CRUD — tak tvoří ono jednotné rozhraní, díky kterému tak můžeme pracovat se zdroji jednotným způsobem a zautomatizovat tak tento proces. V prostředí Webu slouží k určení operace metody protokolu HTTP: (1) GET, (2) POST, (3) PUT, a (DELETE).

2.3 Shrnutí

Nyní, když víme, co jsou to webové služby, a jak je lidé používají, se můžeme zaměřit na to, jak se vytvářejí, protože zde spatřujeme nevyhovující implementace

existujících řešení. Platforma Java EE určuje jednotný způsob definice zdrojů pomocí specifikace JAX-RS [23]. Tato specifikace vyžaduje po tvůrci aplikace, aby každý zdroj a jeho logiku explicitně definoval třídou a anotacemi. Věříme totiž, že když lze se zdroji pracovat pomocí jednotného rozhraní, lze je i jednotným způsobem definovat. O pomyslný krok dále jsou implementace na jiných platformách (Python, Ruby), které díky napojení na model aplikace disponují univerzální logikou, přičemž zůstává možnost přízpůsobení se rozdílným požadavkům, avšak nutná definice jednotlivých zdrojů stále zůstává.

Naším cílem tak je zautomatizovat proces vytváření zdrojů generickým způsobem a zajištění univerzální logiky práce se zdroji — avšak přizpůsobitelnou rozdílným požadavkům na jednotlivé zdroje —, která bude vycházet z modelu aplikace. Základní logika by měla vycházet z uvedených vlastností webových služeb.

3. Rešerše

Pro návrh automaticky generovaného RESTful rozhraní z modelu aplikace je třeba správně pochopit a zanalyzovat následující části:

- REST a jeho použití v prostředí Webu (protokol HTTP),
- modelem řízený vývoj aplikace,
- generativní programování.

Nakonec je vhodné prozkoumat stávající RESTové implementace, konkrétně se zaměřit na:

- interní architekturu knihovny,
- použití uživatelem (tedy programátorem),
- identifikovat silné a slabé stránky.

Vybrané implmentace vycházejí přímo ze zadání, tedy pro jazyk Java jsou to: Jersey [39] a RESTEasy [40]. Pro porovnání s jinou platformou, konkrétně Python, se jedná o tyto knihovny: Django REST [41] a Django Piston [42].

3.1 HTTP

Jedná se o aplikační protokol pro distribuované, hypermedia informační systémy, aktuálně používaný ve versi 1.1 z roku 1999. Tento protokol implementuje RESTový styl definovaný R. Fieldingem v jeho disertační práci.

Principem protokolu je zasílání požadavků ('HTTP Request') od klienta k serveru, na kterém dojde k jejich zpracování a zaslání odpovědi ('HTTP Response') zpět ke klientovi.

Jak požadavek, tak i odpověď jsou určitým typem obálky, která nese informace potřebné ke zpracování dotazu, respektive k odeslání odpovědi, jsou to tedy ony samopopisné zprávy definované v REST.

3.1.1 Metoda

Metoda určuje akci, kterou by měl server provést se zdrojem. Zavedení obecných metod slouží pro jednotnou komunikaci mezi různými systémy - tedy splnění jednoho z požadavků kladených na RESTové rozhraní. Každý HTTP dotaz na server je specifikován jednou z těchto metod (HTTP verse 1.1): 'GET', 'HEAD', 'POST', 'PUT', 'DELETE', 'OPTIONS', 'TRACE', 'CONNECT', 'PATCH'.

GET vrací informace o zdroji. Obsahuje jeho reprezentaci, pokud nebyl zdroj kešován či nenalezen. Tato metoda je idempotentní, to znamená, že několikanásobné použití dotazu nemění stav aplikace, a zároveň je označena za bezpečnou, tedy na serveru by nemělo docházet k postranním efektům, které by mohly ovlivňovat stav aplikace.

HEAD má stejný význam jako ‘GET‘, ale nevrací žádná data v tělu, tedy reprezentaci zdroje. Slouží k získání metadat o zdroji obsažených v hlavičkách HTTP odpovědi. Tato metoda je bezpečná a zároveň idempotentní.

POST metodou je možné vytvářet nové zdroje z reprezentace obsažené v těle dotazu. Metoda není bezpečná, protože s každým dotazem se mění stav aplikace (vytvářejí se nové zdroje), a není ani idempotentní, opět s každým dotazem vznikne nový zdroj - toto chování nemusí být na závadu, ale je třeba, aby s tím autor služby počítal.

PUT aktualizuje konkrétní zdroj - informace z reprezentace obsažené v těle dotazu nahradí informace uložené na serveru. Pokud zdroj neexistuje, tak ho server může vytvořit. Metoda není bezpečná, ale je idempotentní.

DELETE odstraní konkrétní zdroj. Tato metoda není bezpečná, ale je idempotentní.

TRACE vrací dotaz zpět ke klientovi - umožňuje mu tak sledovat, co se děje s dotazem, který putuje sítí přes jednotlivé kompomenty systému. Tato metoda je bezpečná i idempotentní.

OPTIONS vrací seznam metod akceptovaných pro danou URL. Lze tedy využít k inspekci akcí, které může klient s daným zdrojem provést. Tato metoda je bezpečná i idempotentní.

CONNECT je speciální metoda, která transformuje spojení požadavku na transparentní TCP\IP tunel - hlavně se používá při šifrovaném spojení přes protokol HTTPS a nemá tak samotný význam pro práci se zdroji.

PATCH částečně aktualizuje daný zdroj. Na rozdíl od metody ‘PUT‘ tato metoda nenahrazuje všechny informace, ale pouze částečně upravuje zdroj. Metoda není bezpečná ani idempotentní.

3.1.2 Hlavičky

Hlavičky specifikují požadavky zdroje na server (v dotazu), či upřesňují informace o zdroji odeslaném klientovi (v odpovědi). Jsou to metadata o zdroji, na který se dotazujeme, nebo který dostáváme v odpovědi.

Některé hlavičky jsou společné pro oba dva typy obálek (např. ‘Content-Type‘), jiné jsou specifické pouze pro dotaz (např. ‘Accept‘), a pak jsou ty, které jsou určené pouze pro použití v odpovědi (např. ‘Last-Modified‘).

Zajímavou vlastností hlaviček je tzv. content negotiation, tedy způsob určení reprezentace zdroje. Pokud chceme, aby reprezentace zdroje byla v českém jazyce, pak stačí uvést tuto hlavičku při dotazu 3.1.

Jsme-li ochotni přijmout i reprezentaci v jiném jazyce, např. anglickém, ale s nižší prioritou, pak stačí upravit hlavičku podle 3.2.

Algoritmus 3.1: Hlavička pro požadovaný jazyk

```
Accept-Language: cs
```

Algoritmus 3.2: 'Content negotiation' u hlavičky specifikující jazyk

```
Accept-Language: cs, en; q=0.5
```

Server se pokusí splnit požadavek na český jazyk, který má nejvyšší prioritu ('q=1.0'), ale pokud nebude daná reprezentace k dispozici, pokusí se najít tu s anglickým jazykem.

3.1.3 Požadavek

Požadavek je první z dvojice samopopisných zpráv, kterou zasílá klient na server. Tato zpráva (HTTP request) je jakousi obálkou, která obsahuje povinně versi protokolu, metodu, hlavičky a nepovinné tělo zprávy, označované také jako dokument, entita, etc., reprezentující daný zdroj.

Z hlaviček je povinná pouze jediná, a to ta, která určuje adresu námi dotazovaného serveru. Příklad nejjednoduššího dotazu na zdroj zaměstnanců ve firmě by vypadal následovně: 3.3.

Algoritmus 3.3: Jednoduchý dotaz v protokolu HTTP

```
GET /employees HTTP/1.1
Host: example.com
```

Spojení HTTP metody a cesty ke zdroji splňuje požadavky na jednotné rozhraní a s využitím potřebných hlaviček se tak dotaz stává samopopisným.

3.1.4 Odpověď

Odpověď je druhou samopopisnou zprávou z protokolu HTTP, kterou posílá server zpět klientovi na jeho dotaz. Zpráva se skládá z kódu, hlaviček a těla.

Kód informuje klienta o výsledku zpracování dotazu. Podle této informace se klient rozhoduje, jak bude pokračovat v průchodu aplikací.

Kódy odpovědi jsou rozděleny do pěti tříd:

Tabulka 3.1: Rozdělení kódů odpovědí do tříd

Třída	Popis
1xx	Informují klienta o přijetí dotazu, ale proces zpracování stále probíhá.
2xx	Označují úspěšné přijetí dotazu a jeho zpracování pro server.
3xx	Pro dokončení dotazu je třeba aktivního zapojení na straně klienta.
4xx	Klient poslal špatně zformulovaný dotaz.
5xx	Server nedokázal zpracovat validní dotaz.

Každá třída obsahuje jednotlivé kódy, které konkretizují daný výsledek zpracování dotazu, a jejich krátké textové reprezentace, které mají informativní charakter pro uživatele (lidskou bytost). Seznam kódů a jejich význam je uveden ve specifikaci protokolu HTTP.

Hlavičky obsahují nezbytné informace o zdroji a jeho reprezentaci. Klient je díky těmto informacím schopen zpracovat tělo odpovědi a zobrazit tak zdroj v dané reprezentaci uživateli, či provést dodatečné operace.

Tělo opdovědi může být prázdné, ale většinou po serveru něco chceme, takže odpověd obsahuje nějakou reprezentaci zdroje - např. dotaz na zaměstance by mohl vracet v těle odpvědi seznam v XML podobě 3.4.

Algoritmus 3.4: Reprezentace zdroje ve formátu XML

```
<employees>
  <employee>
    <id>1</id>
    <name>Jan Novak</name>
  </employee>
  <employee>
    <id>2</id>
    <name>Jan Novak</name>
  </employee>
  ...
</employees>
```

3.2 REST

REST je architektonický styl určený pro „distribuované hypermedia systémy“, který uvedl na světlo světa Roy T. Fielding ve své disertační práci v roce 2000. Jeho práce vychází z vývoje webového protokolu HTTP vydaného v roce 1999.

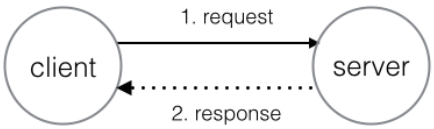
Tento styl je definován ze spojení několika architektur a architektonických stylů používaných především v síťově orientovaných aplikacích.

Fielding definoval celkem pět povinných omezení a jedno nepovinné, která je třeba dodržovat, aby se jednalo o plnohodnotný RESTový architektonický styl:

1. klient-server,
2. bezestavovost,
3. kešování dat,
4. uniformní rozhraní,
5. vrstvený systém,
6. kód na vyžádání (nepovinné omezení).

3.2.1 Klient-server

Styl typu klient-server je jedním ze základních architektonických stylů použitých v síťově orientovaných aplikacích. Server nabízí své služby. Klient posílá dotazy na server, ten je buď odmítne, nebo zpracuje a odešle odpovídající odpovědi zpět klientovi. Schéma komunikace je znázorněno na obrázku 3.1. Použitím tohoto stylu zjednodušujeme implementaci na straně serveru, a děláme ho tak snadněji škálovatelným. Typickým příkladem zjednodušení je implementace uživatelského rozhraní na straně klienta a datového skladiště na serverové části. Tento styl nijak neurčuje, jak je rozdělen stav aplikace mezi klienta a server.



Obrázek 3.1: Komunikace klient-server

3.2.2 Bezestavový systém

Každý dotaz mezi klientem a serverem obsahuje veškeré potřebné informace proto, aby mohl být dotaz na serveru přijat, zpracován, a odpověď zaslána zpět ke klientovi. Je tedy na klientovi, aby si spravoval stav se serverem.

Toto omezení umožňuje, aby byl systém škálovatelný, tj. uvolnit použité zdroje po vykonání dotazu, dále aby byl spolehlivý - systém se může vypořádat s případným selháním, a viditelný - není třeba dohledávat další informace k dotazu někde jinde.

Jeho nevýhodou je naopak nutnost opakovaně posílat potřebná data, což může mít vliv na výkon sítě. A také tím, že je stav aplikace uchováván na straně klienta, je potřebná správná implementace všech klientů zapojených v systému.

3.2.3 Kešování dat

Data v každé odpovědi serveru na dotaz klienta musejí být implicitně, či explicitně označena, jestli jsou, či nejsou určená k uložení (a znovupoužití) na straně klienta. Toto omezení má dopad na síťový výkon, kdy klient zvažuje poslání dotazu na server, nebo použije data ze své paměti.

Keš může být použita jak na straně klienta, tak na straně serveru. Použití na straně klienta snižuje množství dotazů na server, tedy zatížení sítě, protože se klient zbytečně neptá při každém dotazu uživatele, pokud k tomu má potřebné informace, např. je-li obrázek statický a určen ke kešování HTTP hlavičkou ‘Expires‘ nastavenou na určité datum, klient se nebude dotazovat serveru na daný obrázek do uvedeného data.

Pokud server potřebuje vysoké prostředky (výpočetní výkon, paměť) pro zpracování dotazu, je vhodné, aby si daný výsledek uložil do keše, pro případné stejné

dotazy, které mohou chodit na server. Je třeba, aby měl server podchycenou invalidaci nakešovaných dat, aby nedocházelo k jejich zamrznutí. Kromě toho, že tato keš sníží zátěž na samotném serveru, zrychlí také odezvu ke klientovi.

3.2.4 Jednotné rozhraní

RESTový architektonický styl vyžaduje po systémech jednotné komunikační rozhraní mezi komponentami. Jedná se sice o svazující omezení napříč různými aplikacemi, ale na druhou stranu přináší do celého Webu pravidla, která umožní jednotlivým komponentám se spolu dorozumět.

Jednotné rozhraní je v RESTovém architektonickém stylu zajištěno těmito prostředky:

Zdroj představuje základní stavební kámen RESTového stylu. Jedná se o jakoukoliv informaci, které můžeme přiřadit nějaké jméno, např.: dokument, video, aktuální stav nějaké služby („stav akcií firmy na burze“), odkazy na další zdroje, či virtuální objekty. Fielding definuje zdroj R jako funkci MR(t), která daný čas t mapuje na množinu reprezentací zdroje, a nebo na množinu jeho identifikátorů. Tato množina může být i prázdná, to znamená, že zdroj sice existuje, takže na něj můžeme odkazovat, ale prozatím nemá žádnou svoji reprezentaci. Představme si to na příkladu služby hodnoty akcií firmy na burze v daný den: máme zdroj „akcie firmy dne 1. 1. 2020“, v současné době nemá daný zdroj reprezentaci (hodnotu akcie), ale mohu na tento zdroj odkazovat např. na svých stránkách.

Zdroje mohou být statické, nebo naopak variabilní - v našem příkladě služby může být takový statický zdroj např. „akcie firmy 12. 6. 2002 v 10:00“, naopak proměnný zdroj „akcie firmy právě teď“, je zřejmé, že tato hodnota se může v průběhu času měnit. Je tedy možné, aby dva různé zdroje odkazovaly na stejnou hodnotou v daný čas, či po určitou dobu.

Identifikátor slouží k jednoznačnému určení konkrétního zdroje, se kterým pracují jednotlivé komponenty v systému. V RESTové architektuře je ponecháno na autorovi, aby určil, jak bude identifikovat své zdroje. Kvalita identifikátorů je závislá na prostředcích (autor, peníze, čas), použitých při jejich tvorbě. Je tedy běžné, že v prostředí Webu existují nefunkční odkazy ne proto, že by zdroj přestal existovat, ale proto, že autor změnil jeho identifikátor, tedy URL. Jednou zveřejněný identifikátor by měl zůstat neměnný, nebo pokud je nutné ho změnit, potom poskytnout automatické přesměrování na zdroj pod novým identifikátorem.

Špatnou ukázkou tvorby identifikátorů jsou krkolomné, nic neříkající adresy zdrojů. Například tato adresa ‘http://www.autokelly.cz/#|WP2N0moO+EvNJ6ym96nEeMpdQszkBzWrzIwDiyacNCWX27dFmq2NhkrXNkcjQwPjzmZ5aJ4Bw26pPXxNGkqdQI12p4AD842khimLLQsFij351VDHIsjO2ZVEkeOxTQr9tNEiHGVzetpxPSdF2emL75hvAB7z2t2g3Yp7Fd3oXtf/IIhN1+v/it6nHABmUs5m‘ odkazuje na konkrétní typ čtyřkolky ‘Jinling 250‘. S takto vypadající adresou toho člověk moc nenadělá: psát si ji na papír nebude, natož aby si ji mohl zapamatovat. Přitom by adresa tohoto zdroje mohla vypadat třeba nějak takto: ‘http://www.autokelly.cz/jinling/250‘.

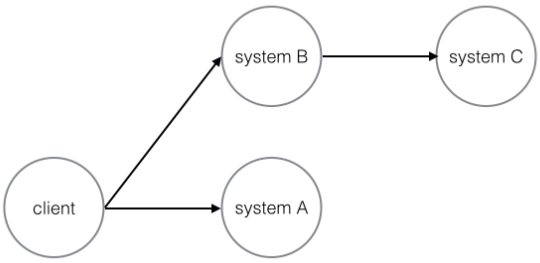
Reprezentace zdroje se používá k vykonávání akcí v RESTových aplikacích. Reprezentace je sekvence bytů a informace, která říká, co tato sekvence znamená (např. HTML, JSON, JPEG, ...). Zdroj může mít více reprezentací, např. zaměstnanec s osobním číslem 324 ve firmě může být reprezentován jak stránkou s kontaktními údaji, tak i svojí fotkou. Tento zaměstnanec je identifikován adresou 'http://example.com/employees/324'. Zde však vzniká problém, jak rozlišit jednotlivé reprezentace. V prostředí Webu máme k dispozici HTTP hlavičky, viz sekce x.y.z, které jsou přítomny při dotazu na službu, konkrétní hlavička pro HTML stránku by tak byla: 'Accept: text/html', a pro fotku: 'Accept: image/jpeg'. Tento standardní způsob funguje dobře, pokud má uživatel kontrolu nad vytvářením dotazu, což však selhává při používání protokolu HTTP a běžných klientů (webových prohlížečů). V URL adrese není totiž možné specifikovat jednotlivé HTTP hlavičky, je proto třeba využít část URL, konkrétně 'query string'. Query string by tak obsahoval řetězec pro HTML stránku: '?accept=text%2Fhtml' (http://example.com/employees/324?accept=text%2Fhtml) a pro fotku: '?accept=image%2Fjpg' (http://example.com/employees/324?accept=image%2Fjpeg). Toto už ale není standardizovaný přístup a je nutná souhra klienta a serveru. Také lze namísto 'query string' použít přímo 'path' v URL, opět tedy pro stránku http://example.com/employees/324.html a pro fotku: http://example.com/employees/324.jpeg, toto řešení však vytváří pro každou reprezentaci zdroje nový identifikátor, který by měl být jednotný. Proto nesouhlasím s doporučením autora ROA, používat specifikaci reprezentace v URL.

Samopopisné zprávy jsou dotazy klienta a odpovědi serveru, které si mezi sebou navzájem posílají. Tyto zprávy korespondují s podmínkou bezestavového systému, tedy že zprávy obsahují veškeré potřebné informace ke zpracování dotazu, resp. odpovědi.

Hypermedia slouží jako hnací síla k procházení mezi stavy aplikace - posíláním samopopisných zpráv, které mohou obsahovat reprezentaci, identifikovaným zdrojům může docházet ke změně stavu aplikace. Hypermedia jsou rozšířením pro hypertext, tedy kromě textů a odkazů mohou obsahovat zvuk, grafiku a video. V RESTovém prostředí umožňují hypermedia nelineární procházení aplikace (jejího stavu) pomocí reprezentací zdrojů. Odkazy tak slouží klientovi k dynamickému reagování na odpovědi ze serveru, jak může dále pokračovat v průchodu aplikací (jejím stavem).

3.2.5 Vrstvený systém

Jednotlivé komponenty systému je možné na sebe napojovat 3.2, ale komunikace mezi komponentami probíhá pouze o jednu úroveň dále. Tímto oddělením je možné např. zajistit zpětnou kompatibilitu pro zastaralé komponenty a přitom poskytovat rozhraní pro nové systémy. Nevýhodou je zvýšený tok v síti, ale při správné implementaci kešování může být tento problém potlačen.



Obrázek 3.2: Vrstvený systém

3.2.6 Kód na vyžádání

Klientům je umožněno stahovat a vykonávat kód ve formě appletů, nebo skriptů, odpadá tak nutnost implementace na straně klienta. Na druhou stranu snižuje viditelnost, kdy se stává jedna komponenta (klient) závislá ná jiné (server), proto se Fielding rozhodl ponechat toto omezení jako volitelné.

3.3 Srovnání existujících řešení

Na následujích stránkách se budeme věnovat průzkumu a vyhodnocení několika stávajících implementecí RESTového rozhraní. Cílem je pochopit jak fungují a jak se používají progamátory, dále určit jejich výhody a nevýhody. Tyto informace budou sloužit pro návrh vlastního způsobu práce s RESTovým rozhraním.

Pro Java Enterprise aplikace je výchozím bodem JAX-RS: Java API for RESTful Web Services. Jedná se o specifikaci API mapování zdrojů na obyčejné třídy v Javě (POJO). V prostředí Pythonu není podobná specifikace, ale existují nezávislé knihovny s podporou pro RESTové rozhraní.

3.3.1 Jersey

Tato knihovna je jednou ze základních implementací specifikace JAX-RS, konkrétně ve versi JSR 311 (1.1) a i novější JSR 339 (2.0), která umožňuje vývoj webových služeb na platformě Java podle RESTového stylu. Pro popis webových zdrojů používá kombinaci POJO tříd a upřesňujících anotací, které určují životní cyklus objektů a jejich rozsah. Knihovna předpokládá použití výhradně HTTP protokolu jakožto síťovou vrstvu. Dále si klade za cíl, aby zdroj byl nezávislý na konkrétní reprezentaci, proto dává možnost jednoduchého rozšíření o nové reprezentace. Specifikace také určuje, jak mají být artefakty nasazeny v Servlet kontejneru a jako JAX-WS Provider. Nakonec specifikuje začlenění API do prostředí Java EE.

Koncept zdroje je velice jednoduchý - udělat z obyčejné třídy zdroj, který je schopný zpracovat HTTP dotazy a vracet odpovědi. Pro definici zdrojů mohu využít dle specifikace POJO a anotace, nebo programovatelné API.

Přístup pomocí anotací je přímočarý a jednoduchý: (1) vytvoříme třídu s anotací ‘@Path‘, která mapuje URL na daný zdroj. (2) Přidáme metody, které budou

namapovány na HTTP metody. (3) Určíme, jaké reprezentace mají být pro daný zdroj dostupné anotacemi ‘@Consumes‘ a ‘@Produces‘, tyto anotace můžeme libovolně kombinovat. (4) Implementujeme funkcionalitu pro danou metodu.

Algoritmus 3.5: POJO třída s JAX-RS anotacemi

```
package com.example;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Path("resource")
public class Resource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String get() {
        return "Resource";
    }
}
```

3.5 je velice jednoduchá ukázka, která nebere žádné parametry a pouze vrací textový řetězec. Pro podporu parametrů slouží anotace ‘@PathParam‘ 3.6, ‘@QueryParam‘, ‘@FormParam‘, ‘@HeaderParam‘, atd., které mapují HTTP parametry na vstupní parametry dané metody, tímto způsobem si tedy lze vyžádat konkrétní parametry a jejich hodnoty, které uživatele zajímají a chce s nimi dále pracovat.

Algoritmus 3.6: Získání parametru ‘ID‘ z URL cesty

```
@GET @Path("/employees/{id}")
@Produces({"application/json"})
public Response get(@PathParam("id") String id) { /* return employee
    for id */ }
```

Pro malý počet vstupních parametrů je tento způsob vhodný díky jednoduchosti a přehlednosti. K složitější logice s parametry je možné získat celý dotaz pomocí anotace ‘@Context‘ takto ‘public Response get(@Context Request request) ... ‘ a pracovat přímo s ní. To samé platí pro odpověď, kdy je možné vracet instance třídy ‘Response‘, která umožňuje vytvořit libovolnou odpověď. Knihovna ale také podporuje automatickou serializaci a deserializaci reprezentací na POJO pomocí anotací a zpět pomocí specifikace JAXB, pokud by ani tento přístup nevyhovoval, tak je možné použít dvojici tříd ‘MessageBodyReader<T>‘ a ‘MessageBodyWriter<T>‘, respektive jejich metody ‘readFrom()‘ a ‘writeTo()‘. Následuje ukázka 3.7 automatické (de)serializace třídy a její použití jako vzor pro reprezentaci zdroje.

Tento přístup pro mapování tříd z aplikačního modelu na zdroje však explicitině vyžaduje to, aby pro každou třídu, kterou bychom chtěli zpřístupnit přes RESTové rozhraní, měli odpovídající definici zdroje, vedlo by to tedy k duplicitám v kódu, což zbytečně navyšuje velikost aplikace a může i zanášet chyby do kódu, které mohou být lehce přehlédnuty. Proto je možné použít programovatelné API, které umožňuje definovat zdroj bez existence konkrétní třídy 3.8. Toto

Algoritmus 3.7: Serializace pomocí specifikace JAXB

```
@XmlRootElement
public class Employee {
    public String name;
    public int age;

    public Employee() {  }
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

@Path("/employees")
public class EmployeeResource {

    @GET @Path("/{id}")
    @Produces({"application/json"})
    public Employee find(@PathParam("id") String id) { /* return
       employee for id */ }

    @POST @Path("/")
    @Consumes({"application/json"})
    public void create(Employee employee) { /* persist employee */ }
}
```

řešení není sice součástí specifikace JAX-RS, ale rozšířením knihovny Jersey, ale i tak už by bylo vhodnějším pro automatické generování zdrojů z modelu, chybí zde ale nějaký konvertor, který by tuto funkcionalitu umožňoval, takže uživatel by musel stejně logiku implementovat sám.

Tím, že je definice zdroje přenechána na uživateli, je na něm, jak bude zpracovávat dotazy. Je však možné toto zpracování doplnit o společnou logiku, k tomu slouží (a) filtry a (b) interceptory. Filtry slouží k úpravě dotazu, resp. odpovědi co se týče meta informací, tedy URL, HTTP metody a hlaviček, tak interceptory slouží k úpravám těla dotazů, resp. odpovědí, tedy reprezentace zdroje. Filtry i interceptory jsou rozhraní, které se dělí na vstupní a výstupní: 'ContainerRequestFilter', 'ContainerResponseFilter', 'ReaderInterceptor' a 'WriterInterceptor'. Uživatel může určit pořadí daného filtru, nebo interceptoru celočíselnou hodnotou, knihovna poté sama poskládá jednotlivé filtry, resp. interceptory v uvedeném pořadí. Ukázka použití vlastního filtru 3.9.

Bezpečnost REST rozhraní je zajištěna několika způsoby: (1) HTTP Authentication Basic a Digest, (2) OAuth protokolem a (3) při nasazení aplikace v Servlet kontejneru je použito standardního mechanismu pro Java EE webové aplikace. Zabezpečení je možné definovat standardně pro Java EE aplikace pomocí elementu '<security-constraint>' v konfiguračním souboru 'web.xml', což ale vyžaduje manuální synchronizaci s definovanými zdroji. Proto knihovna nabízí použití anotací přímo ve třídách jednotlivých zdrojů, viz následující ukázka 3.10.

Knihovna přidává širokou podporu pro různé reprezentace zdrojů (XML, Atom, JSON, atd.) a možnosti zabezpečení, rozšíření pomocí filtrů a interceptorů. Samotná specifikace JAX-RS umožňuje jednoduchou tvorbu webové služby pomocí tříd a anotací. Tento způsob definice zdrojů však není vhodný pro správu vel-

Algoritmus 3.8: Definice zdroje pomocí programovateleného API

```
package com.example;

import ...

public static class EmployeeConfig extends ResourceConfig {

    public EmployeeConfig() {
        final Resource.Builder resourceBuilder = Resource.builder();
        resourceBuilder.path("employees/{id}");

        final ResourceMethod.Builder methodBuilder =
            resourceBuilder.addMethod("GET");

        methodBuilder.produces(MediaType.APPLICATON_JSON)
            .handledBy(new Inflector<ContainerRequestContext,
                Employee>() {

            @Override
            public Employee apply(ContainerRequestContext context) {
                String id = /* get ID from request  */;
                return findEmployeeById(id);
            }
        });

        final Resource resource = resourceBuilder.build();
        registerResources(resource);
    }
}
```

kého množství zdrojů - znamenalo by to pro každou modelovou třídu definovat zdroj. Bylo by možné mít jeden „univerzální“ zdroj, který by zpracovával všechny dotazy, ale to by fungovalo pěkně s jednotnou logikou napříč celým modelem aplikace - s každou odchylkou by bylo třeba upravovat tento univerální zdroj, což by ho učinilo postupem času nepřehledným a náchylným k chybám. Díky programovatelnému API přidává knihovna Jersey možnost automatického provázávání s modelem aplikace, je to však pouze možnost, protože chybí jakékoliv napojení na model, ze kterého by šlo vybudovat RESTové rozhraní programově.

3.3.2 RESTEasy

RESTEasy je konkurentem knihovny Jersey pro tvorbu RESTových webových služeb na platformě Java. Implementuje stejnou specifikaci JAX-RS jak novější JSR 339 (2.0), tak i starší JSR 311 (1.1). Díky stejné specifikaci umožňuje knihovna tu samou funkcionalitu jako předešlá knihovna, tedy definování zdrojů pomocí POJO a anotací. Většina funkcionalit je taktéž stejná - např. serializace do/z JSON reprezentace pomocí knihovny Jackson, či podpora specifikace JAXB. Bezpečnost aplikace je zajištěna na úrovni Java EE webové aplikace, takže se opět nabízí možnost využít anotace z balíčku ‘javax.annotation.security‘, a nechybí podpora pro protokol OAuth [43]. Na rozdíl od Jersey nemá alternativu pro filtry, ale interceptory pro transormaci těla dotazu (odpovědi) jsou zde přítomny.

Algoritmus 3.9: Filtr upravující HTTP hlavičku odpovědi

```
package com.example;

import ...

@Priority(Priorities.HEADER_DECORATOR)
public class PoweredByResponseFilter implements
    ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
                       ContainerResponseContext responseContext)
                       throws IOException {
        responseContext.getHeaders().add("X-Powered-By", "com.
            example");
    }
}
```

Algoritmus 3.10: Zabezpečení zdroje podle rolí

```
@Path("employees")
@PermitAll
public class Employee {

    @RolesAllowed("user")
    @GET
    public Response findAll() { /* return list of employees */ }

    @RolesAllowed("administrator")
    @POST
    public String create(Employee employee) {
      /* create a new employee record */
    }
}
```

Výrazným rozdílem oproti knihovně Jersey je podpora pro kešování na straně serveru. Používá produkt zvaný Infinispan, který kešuje data v paměti. Knihovna automaticky kontroluje HTTP hlavičky kešování a načítá (ukládá) data podle URL, a invaliduje je podle volání HTTP metod 'POST', 'PUT' a 'DELETE'. Pokud uživatel potřebuje pracovat s keší manuálně, může si ji vytáhnout z kontextu, viz ukázka 3.11.

Algoritmus 3.11: Vytažení keše z aplikačního kontextu

```
@GET
public String get(@Context ServerCache cache) {
    /* do something with cache */
}
```

Knihovna nedisponuje programovatelným API jako Jersey, proto bavit se o nějakém automatickém mapování modelu aplikace na zdroje bohužel nelze.

3.3.3 Django REST

Třetí ze zkoumaných knihoven je určena pro tvorbu RESTových aplikací ve webovém frameworku Django v jazyce Python. Na rozdíl od platformy Java EE nenabízí Django standardizované rozhraní pro zpřístupnění webových služeb. Poskytuje však možnosti, které jsou shodné s touto platformou, oproti předchozím knihovnám nabízí některá výchozí rozšíření, jako např. filtrování či stránkování.

Níže uvedený postup 3.12 vychází z ukázkového kódu na stránkách dokumentace ke knihovně. Mám dvě modelové entity 'User' and 'Group', které chci zpřístupnit pomocí RESTového rozhraní. Na začátku mohu nastavit výchozí Django REST chování pro celý model apliakce, např. serializaci, oprávnění přístupu k modelu, či autentizaci. Uvedený serializer automaticky generuje reprezentaci pro daný model, vztahy mezi entitami (1:N a M:N) jsou řešeny hyperlinky.

Algoritmus 3.12: Globální nastavení knihovny

```
REST_FRAMEWORK = {
    # Use hyperlinked styles by default.
    # Only used if the `serializer_class` attribute is not set on a
        view.
    'DEFAULT_MODEL_SERIALIZER_CLASS':
        'rest_framework.serializers.HyperlinkedModelSerializer',

    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.
            DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Druhým krokem je definice propojení mezi HTTP dotazy a jejich vykonáním nad zdrojem. K tomuto slouží třída 'ViewSet' a její potomci, obzvláště třída 'ModelViewSet', která přidává výchozí implementaci CRUD operací nad modelem aplikace. Třída 'ViewSet' odpovídá definici zdroje obdodně jako v Javě EE definuji POJO třídu s anotacemi '@Path', '@GET', atp. - rozdíl je tu ten, že není nutné psát implementaci, pokud pracujeme s modelem aplikace, a nepotřebujeme upravovat výchozí chování. Nakonec je nutná registrace URL, pod kterou bude daný zdroj dostupný. Výsledná definice rozhraní je znázorněna v 3.13.

Toto stačí pro zpřístupnění modelových entit přes RESTové rozhraní - postup je jednoduchý a z kódu pochopitelný. Opět zde ale narážíme na to, že musíme pro každou entitu explicitně definovat 'view set' a zaregistrovat URL, pod kterou bude daný zdroj přístupný.

Rozšířit či upravit výchozí implementaci zpracování dotazu lze děděním některých z výchozích tříd, např. 'ModelSerializer' pro serializaci modelu, či 'ModelViewSet' pro úpravu logiky zpracování dotazu. Když chceme přidat atribut navíc oproti modelu uživatele, např. `days_since_joined`, zdědíme serializer a přidáme atribut, který se dostane voláním metody `get_days_since_joined`. 'SerializerMethodField' (podtřída třídy 'Field') serializuje hodnoty vrácené metodou a je

Algoritmus 3.13: Definice zdrojů a registrace URL

```
from django.conf.urls.defaults import url, patterns, include
from django.contrib.auth.models import User, Group
from rest_framework import viewsets, routers

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    model = User

class GroupViewSet(viewsets.ModelViewSet):
    model = Group

# Routers provide an easy way of automatically determining the URL
    conf
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)
router.register(r'groups', GroupViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browseable API.
urlpatterns = patterns('',
    url(r'^', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='
        rest_framework'))
)
```

pouze pro čtení 3.14. Typů třídy Field je více, kromě zmíněné 'SerializerMethodField', máme k dispozici 'WritabeField', 'ModelField', nebo můžeme implementovat vlastní.

Algoritmus 3.14: Přidání vlastního atributu do zdroje

```
from django.contrib.auth.models import User
from django.utils.timezone import now
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    days_since_joined = serializers.SerializerMethodField('
        get_days_since_joined')

    class Meta:
        model = User

    def get_days_since_joined(self, obj):
        return (now() - obj.date_joined).days
```

Další možností je specifikovat pouze určitou množinu atributů modelu, které chceme dostat do výsledné reprezentace. K tomuto slouží pole se jmény atributů 'fields' 3.15, druhou možností by bylo určit atributy, které naopak chceme vyloučit, k tomu slouží pole atributů 'exclude'.

Pokud by bylo zcela nutné změnit pro daný zdroj proces serializace, určíme přímo třídu pro serializaci atributem `serializer_class` třídy zdroje 3.16.

Můžeme také zpřístupnit zdroj pouze pro čtení, tedy že bude odpovídat pouze

Algoritmus 3.15: Specifikace množiny atributů

```
class Meta:
        model = User
        fields = ('name', 'email')
```

Algoritmus 3.16: Vlastní serializer pro daný zdroj

```
class UserViewSet(viewsets.ModelViewSet):
    model = User
    serializer_class = SomeSpecialUserSerializer
```

na HTTP volání 'GET' metody, a ostatní metody budou vracet chybový stav. K tomu stačí určit rodičovskou třídu 'ReadOnlyModelViewSet', viz ukázka níže 3.17.

Algoritmus 3.17: Zdroj určený pouze pro čtení

```
class UserViewSet(viewsets.ReadOnlyModelViewSet):
    model = User
```

Knihovna nabízí programátorovi jednoduché napojení modelu aplikace na RESTové rozhraní, umožňuje i široké možnosti rozšíření a úprav podle specifických potřeb, na druhou stranu ale zde chybí automatizace definice tohoto rozhraní pro aplikační model. Takže se opět nevyhneme opakující se práci, oproti knihovnám na platformě Java však již tuto práci uživateli výrazně zjednodušuje tím, že nabízí výchozí implementaci pro napojení na CRUD operace nad modelem.

3.3.4 Django Piston

Poslední ze zkoumaných knihoven je Django Piston - relativně malá a jednoduchá sada tříd pro tvorbu RESTových rozhraní na platformě Django. Jako všechny předešlé knihovny, i tato má pro reprezentaci zdroje svoji specifickou třídu ('Resource'), která má na starosti zpracování HTTP dotazu a zaslání odpovědi zpět klientovi. Celá logika zpracování dotazu je pouze v této třídě, sama pouze deleguje některé dílčí akce na jiné třídy, těmito akcemi jsou: (1) serializace reprezentace zdroje, (2) autentizace dotazu, resp. uživatele, a (3) samotné vykonání akce pro danou HTTP metodu.

Pro nejjednodušší zpřístupnění zdroje 3.18 (jakožto modelovou třídu) pomocí této knihovny je třeba (1) mít definovaný model, (2) podědit 'BaseHandler' a nastavit atribut 'model', (3) vytvořit instanci třídy 'Resource', (4) zaregistrovat URL, pod kterou bude zdroj dostupný.

Serializace dat je zajištěna třídou 'Emmiter' a jejími potomky, které upravují metodu 'render' pro podporu daného formátu. Knihovna nativně podporuje reprezentaci dat v XML, JSON, YAML, Python Pickle, a Django formátu. Pro přidání podpory nového formátu, či úpravy stávajícho způsobu serializace pro konkrétní formát je třeba podědit danou třídu a doplnit kód do metody render. Bohužel logika serializace pro datové typy (QuerySet, dict, Model, function,

Algoritmus 3.18: Zpřístupnění modelu v Django Piston

```
from django.db import models

class Article(models.Model):
    title = models.CharField(max_length=255)
    text = models.TextField()

from piston.resource import Resource
from piston.handler import BaseHandler

class ArticleHandler(BaseHandler):
    model = Article

article = Resource(handler=ArticleHandler)
urlpatterns += patterns('',
    url(r'^articles/(?P<id>[^/]+)/$', article)
)
```

atd.) je přímo ve třídě ‘Emitter‘, z toho vyplývá špatná rozšiřitelnost pro specifické (vlastní) datové typy, která by byla možná pouze zděděním třídy ‘Emitter‘ a úpravou potomků pro konkrétní formáty tak, aby dědili od upravené třídy. Ukázka implementace emitenta pro JSON formát 3.19.

Algoritmus 3.19: Implementace a registrace JSON emitenta

```
class JSONEmitter(Emitter):
    def render(self, request):
        cb = request.GET.get('callback', None)
        seria = simplejson.dumps(self.construct(), cls=
            DateTimeAwareJSONEncoder, ensure_ascii=False, indent=4)

        # Callback
        if cb and is_valid_jsonp_callback_value(cb):
            return '%s(%s)' % (cb, seria)

        return seria

Emitter.register('json', JSONEmitter, 'application/json; charset=utf
    -8')
```

Deserializace neprobíhá ve třídě ‘Emitter‘, ale ve třídě ‘Mimer‘. Formulářový formát dat (‘application/x-www-form-urlencoded‘) je zpracován na úrovni samotného frameworku Django, formát ‘multipart‘ je podporován nativně v této třídě, pro ostatní formáty je třeba se zaregistrovat s daným typem a metodou, která dokáže deserializovat obsah dotazu 3.20.

Algoritmus 3.20: Registrace deserializéru pro MIME typ

```
from django.utils import simplejson

Mimer.register(simplejson.loads, ('application/json',))
```

Takto lze snadno přidat podporu pro nový formát, má to však nevýhodu v tom, že při deserializaci nedochází k žádné validaci struktury ani hodnot dat, stačí aby data byla validní pro daný formát, ale nemusí být validní v kontextu aplikace. Tento nedostatek je možné částečně vyřešit v třetím kroku zpracování dotazu (viz 'BaseHandler').

Bezpečnost je zajištěna na úrovni autentizace dotazů pomocí protokolu OAuth [43], HTTP Authentication (Basic a Digest) [11] a nebo je možné napsat vlastní logiku autentizace. Následuje ukázka použití HTTP Authentication: Basic 3.21.

Algoritmus 3.21: Nastavení HTTP autentizace pro daný zdroj

```
from piston.resource import Resource
from piston.authentication import HttpBasicAuthentication
from piston.handler import BaseHandler

basic = HttpBasicAuthentication(realm="example.com")
resource = Resource(ResourceBaseHandler, authentication=basic)
urlpatterns = patterns('',
    url(r'^resource/', resource),
)
```

BaseHandler slouží k vykonání základní CRUD logiky nad daným zdrojem. Tato třída mapuje HTTP metodu 'GET' na funkci 'read', 'POST' na 'create', 'PUT' na 'update' a 'DELETE' na 'delete'. Třída nativně pracuje s logikou, kterou nabízí modul 'django.db.models', tedy operace pro práci s databázemi pomocí ORM. BaseHandler počítá s tím, že bude mít jeho instance definovaný atribut 'model' s modelovou třídou. Pokud je třeba přidat, či upravit existující funkcionalitu, je nutné třídu podědit a doplnit implementaci pro jednu ze zmíněných metod.

Jak už bylo zmíněno, při deserializaci nedochází k validaci dat. Je k tomu však možné využít nativní podporu z frameworku Django ('modul django.forms'), která umožňuje validaci formulářových dat z explicitní definice (nadefinovaná uživatelem), či podle definice modelu. I když je validace vstupních dat podporována, je potřeba, aby si ji uživatel vyžádal explicitně. Je tedy nutné pro každý model, který chci validovat, podědit třídu a přidat anotaci 3.22.

Algoritmus 3.22: Validace vstupních dat podle modelu

```
from django import forms
from piston.utils import validate
from com.example.models import Article

class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article

@validate(ArticleForm)
def create(...
```

Knihovna potěší svým jednoduchým použitím, které umožní snadné zpřístupnění modelu aplikace přes RESTové rozhraní. Na druhou stranu každá modelová třída musí být explicitně označena uživatelem pro její zpřístupnění. Tento přístup opět zanáší do aplikace opakování kódu při definování zdrojů a zvyšuje tak možnost výskytu chyb. Rozšiřitelnost funkcionality by znamenala zásah do již existujících tříd (‘Resource‘, ‘Emitter‘), resp. dědit od třídy ‘BaseHandler‘, což není dobré z koncepčního hlediska, resp. tato třída je určená pro manipulaci s daty. Třídy ‘Resource‘ či ‘Emitter‘ by měly být upraveny tak, aby bylo možné je rozšiřovat takovým způsobem, aby nedocházelo k opakování kódu, který byl už jednou napsán.

3.3.5 Zhodnocení knihoven

Na platformě Java má uživatel k dispozici specifikaci, která dává určitý standard pro všechny implementace, proto je použití obou knihoven téměř stejné. Tento standard je velice abstraktní, dává tak neomezené možnosti při definici zdrojů. Bohužel způsob, jakým jsou definovány zdroje, tedy použití anotací na třídě reprezentující zdroj, vede k opakujícímu se kódu, který se v logice víceméně neliší. Pracuje-li totiž uživatel s modelem definovaným např. pomocí JPA, je logika CRUD operací stejná. V příkladu je vidět rozdíl, který dělá tento přístup hůře použitelný při větším množství zdrojů. V JPA definuji entity, ale práce s nimi je stejná, nemusím tak psát pro každou z nich kód pro práci s databází.

Knihovny na platformě Python jdou o tento krok dál, a jsou napojeny na svůj standardní aplikační model (tedy ORM nad databází), usnadňují tak uživateli práci s definicí zdroje. Přitom umožňují snadné začlenění zdrojů, které nepocházejí z tohoto modelu. Stále je však nutné explicitně propojit tento zdroj s daným obsluhovačem, a zaregistrovat URL, pod kterou bude zdroj dostupný. Ale i tak je to oproti řešení v Javě veliký posun.

Pro návrh vlastního řešení lze vyjmout určité části, které by bylo dobré použít či podporovat. (1) Předzpracování dotazu a pozpracování odpovědi pomocí filtrů v knihovně Jersey. (2) Napojení na persistentní model aplikace v knihovnách Django REST a Django Piston. Pokud aplikace zrovna nepředstavuje spojení více služeb (‘mashup‘), potřebuje pro svoje data úložiště, kterým ve většině případů bývá databáze, či souborový systém na serveru. (3) Nakonec je to programovatelné rozhraní z knihovny Jersey, které je však potřeba napojit na model, abychom mohli hovořit o nějaké automatizaci.

3.4 Vývoj řízený modelem

Při vývoji software tímto způsobem se soustředíme na specifikaci modelu a nikoliv na počítačový program jako takový. To nám umožňuje využít koncepty, které jsou méně závislé na určité platformě, ale s jazyky jako např. UML jsou blíže k doméně daného problému. Základním předpokladem této metody je pak automatické generování programu podle daného modelu. Kód tohoto programu pak již není uživatelem upravován.

Využití uvedeného přístupu pro potřeby této práce je zjevné, jenom je nutné zanést význam do kontextu RESTového rozhraní. Modelem rozumíme aplikační model, nejedná se tedy o abstraktní specifikaci v dané doméně, definovaný pomocí UML, ale jedná se o model definovaný již na úrovni kódu. Z toho vyplývá, že jsme sice platformově závislí na dané technologii, ale v našem případě to není problém, protože už samotný model, ze kterého se vychází, je již platformově závislý. RESTové rozhraní představuje onen automaticky generovaný program, to znamená, že uživatel definuje pouze model a pravidla pro generování. Uživatel pak nemění způsob zpracování dotazu, vše potřebné bylo vygenerováno z modelu aplikace.

3.5 Generativní programování

Toto paradigma navrhuje takové programovací postupy, které vedou k znovupoužitelnosti a přizpůsobitelnosti, poslední zmíněná vlastnost je pro nás velice důležitá, protože přizpůsobitelnost je jedním ze základních požadavků na naše RESTové rozhraní.

K zajištění výše uvedených vlastností definuje generativní programování následující principy: (1) rozdělení zodpovědnosti, (2) parametrizace rozdílů, (3) analýza a modelování závislostí a interakcí, (4) oddělení problému od jeho řešení, a (5) snížení režie a provádění doménově závislé optimalizace.

Princip rozdělení zodpovědnosti nám říká, abychom rozdělili program na samostatné části, kde každá část řeší právě a pouze jednu funkcionalitu. Tyto části jsou poskládány do výsledné komponenty.

Parametrizace rozdílů nám umožňuje reprezentovat skupiny kompoment, které mají mnoho společného.

Analýzou parametrů můžeme zjistit, že některé kombinace jejich hodnot nemusejí být vždy validní, a tak mohou vznikat závislosti mezi jednotlivými hodnotami, které se navzájem ovlivňují. Tyto závislosti jsou označovány jako znalost horizontální konfigurace, protože se objevují mezi parametry z jedné úrovně abstrakce.

Oddělení problému od jeho řešení zajišťuje to, že tvůrce aplikace pracuje s abstrakcemi pro danou doménu, kdežto řešení obsahuje implementace jednotlivých komponent. Jak problém tak i jeho řešení používá různé struktury, které je nutné na sebe mapovat s pomocí tzv. znalosti vertikální konfigurace, kde interakce mezi parametry probíhá z různých úrovní abstrakce. Horizontální i vertikální znalost konfigurace umožňují její automatizaci.

Snížení režie spojené s kontrolou za běhu programu, či nevyužitým kódem, je možné, jsou-li komponenty generovány staticky, tedy v době kompilace. V této době mohou být provedeny i některé složité doménově závislé optimalizace.

3.6 Shrnutí

V této kapitole jsme provedli rešerši protokolu HTTP, který umožňuje vybudovat webové služby odpovídající architektonickému stylu zvanému REST. Dodržením zásad definovaných tímto stylem jsme schopni poskytovat takové služby, které pracují se zdroji pomocí jednotného rozhraní, jsou snadno škálovatelné díky své bezstavovosti, dále snižují nároky na síťovou komunikaci i prostředky na straně serveru pomocí kešování.

Poté jsme prozkoumali stávající řešení na platformách Java a Python, abychom byli schopni poučit se z jejich silných, ale i slabých stránek a použít ty dobré vlastnosti ve vlastním návrhu a těm horším se mohli vyhnout.

Pomocí konceptů použitých při modelem řízeném vývoji a aplikování základních principů generativního programování budeme schopni navrhnout vlastní řešení pro plně automatizané generování RESTového rozhraní.

4. Návrh

Následující kapitola patří návrhu vlastního řešení pro zpracování HTTP dotazů a jejich přeměnu na odpovědi. Návrh bude vycházet ze znalostí získaných při zkoumání existujících řešení.

4.1 Proces

Zpracování dotazu a jeho transformaci na odpověď si lze představit jako funkci s jedním vstupním parametrem (dotazem) a návratovou hodnotou (odpovědí):

- y = f(x).

Toto je vlastně definice vycházející ze zkoumaných řešení, kdy např. podle specifikace JAX-RS definuji třídu (zdroj), která implementuje logiku, jak dotaz přetransformovat na odpověď. Takto definovaný vztah mezi dotazem a odpovědí zanáší veškerou logiku do jedné funkce, což není žádané kvůli rozšiřitelnosti. Je sice možné rozšířit funkcionalitu pomocí pre a post filtrů

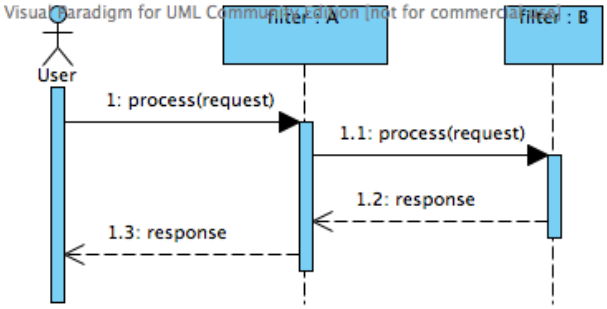
- x' = pre2(pre1(x)),
- y = f(x'),
- y' = post2(post1(y)),

ale toto slouží pouze pro úpravu dotazu, resp. odpovědi, samotná logika stále zůstává ve funkci 'f'. Využijme tento přístup použitý v knihovně Jersey, jelikož ale nechceme po uživateli, aby explicitně definoval každý zdroj zvlášť, umožníme tedy zapojení všech filtrů do série:

- ctx' = f3(f2(f1(ctx))),

Je však třeba zavést jednotnou proměnnou (rozhraní) - kontext, tak aby bylo možné filtry na sebe napojovat. Kontext bude procházet jednotlivými filtry, které budou moci měnit jeho stav podle dané logiky filtru. V tuto chvíli ponecháme definici kontextu na později a vrátíme se k samotnému zpracování dotazů.

Tento způsob řešení problému není ničím novým, odpovídá návrhovému vzoru nazývajícímu se 'Řetěz odpovědnosti' ('Chain of Responsibility') [6], který je znázorněn na obrázku 4.1.



Obrázek 4.1: Řetěz odpovědnosti filtrů

4.2 Filtry

V tuto chvíli, kdy je definován proces zpracování dotazu, je třeba přiřadit filtrům jejich logiku. Současné knihovny nabízejí více či méně podobnou sadu funkcí, které lze při tvorbě RESTového rozhraní použít. Následující seznam vycházející z požadavků definovaných v analýze 2.1.1, uvádí základní sadu filtrů, které budou využity v návrhu:

1. zabezpečení,
2. kešování,
3. (de)serializaci,
4. persistenci.

Tento seznam rozhodně není konečný, v budoucnu se může vyskytnout potřeba nového typu filtru, který v tuto chvíli není znám, avšak díky navržené architektuře nebude velký problém tento nový filtr zakomponovat do systému. Pořadí filtrů není zvoleno náhodně a má svoji logiku. Zabezpečení zamezí jakékoliv následující akci, pokud by nebyl dotaz oprávněný. Kešování může odlehčit dalším filtrům, kdyby byl daný zdroj již uložen v keši. Serializace poté převede reprezentaci zdroje na konkrétní objekt v aplikaci a persistence zajistí ukládání dat a jejich načítání např. z databáze.

4.2.1 Zabezpečení

Zabezpečení zdrojů proti neoprávněnému použití, či v horším případě zneužití, je důležitou součástí síťových aplikací. Definuji obecný filtr pro ověřování identity, tzv. autentizaci, kde možností na ověření identity uživatele je hned několik: HTTP Authentication Basic, nebo Digest [11], či využití nových protokolů jako je OAuth. Druhý filtr bude sloužit k ověřování oprávnění dané identity, tzv. autorizaci, která specifikuje oprávnění k vykonání akce uživatelem nad zdrojem.

4.2.2 Kešování

Jeden ze základních požadavků na RESTové rozhraní definované p. Fieldingem. Jak už bylo zmíněno, správně implementované kešování zdrojů snižuje (1) nároky na vlastní server, (2) velikost přenášených dat po síti. Knihovna proto nabídne základní rozhraní pro implementaci této funkcionality. Na začátku zpracování dotazu bude možnost načíst nakešovanou odpověď a rovnou ji zaslat klientovi. Obdobně bude na konci zpracování možné vytvořenou odpověď uložit do paměti, pro pozdější využití.

4.2.3 Serializace

Serializace a její opačná funkce deserializace zajistí převod objektů z modelu aplikace do reprezentace zdroje (JSON, XML, atd.), resp. z reprezentace zdroje na objekt z modelu aplikace. Z logiky věci vyplývá, že první akcí bude deserializace reprezentace na objekt z modelu, bude-li přítomný v dotazu, při opouštění filtru bude pak použita serializace.

4.2.4 Persistence

Tento filtr slouží pro práci s daty - zajišťuje jednotné rozhraní pro zpracování HTTP metod. To, co bylo nutné explicitně definovat v třídě zdroje při použití specifikace JAX-RS, bude zajištěno právě tímto filtrem. Způsob uložení dat bude záležet na konkrétní implementaci, jestli se použije databáze, souborový systém, či volání na jiný server.

4.3 Kontext

Pro zpracování HTTP dotazů je nutné zasadit filtry do určitého kontextu, který umožní filtrům se na jeho základě rozhodovat při svém vykonávání. Kontext předávaný mezi jednotlivými filtry se skládá celkem ze čtyř částí:

- HTTP dotaz,
- HTTP odpověď,
- model,
- konfigurace.

Dotaz a odpověď pojmenujme jako HTTP kontext, který se mění s každým dotazem na server. Model a konfigurace pak tvoří tzv. aplikační kontext, ten je neměnný mezi jednotlivými dotazy, tedy za běhu aplikace.

4.3.1 HTTP dotaz

První část reprezentuje samotný HTTP dotaz. Pro zpracování dotazu jsou tak nezbytné informace uvedené ve specifikaci protokolu HTTP: metoda, hlavičky, a tělo dotazu. Při zpracování dotazu budou filtry tyto informace primárně číst, ale je vhodné umožnit i jejich změnu při zajištění (ne)očekávaných problémů s kompatibilitou, např. pokud je klient schopen používat pouze metody ‘GET‘ a ‘POST‘ (třeba blokujícími firewally [1]), je nutné použít hlavičku ‘X-HTTP-Method-Override‘, ve které dojde k „přepsání“ HTTP metody, filtr by pak upravil metodu podle hodnoty uvedené v hlavičce.

4.3.2 HTTP odpověď

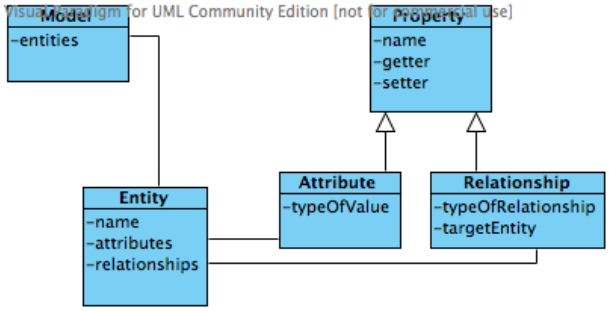
Filtr vytváří odpověď, která se pošle ke klientovi na jeho dotaz. Každý filtr má možnost (1) zpracování dotazu zastavit a okamžitě vrátit odpověď, (2) předat dotaz dále ke zpracování a vrátit odpověď na základě dalšího zpracování, (3) a nebo vůbec s odpovědí nic nedělá a jenom předá kontext zpět. Dle specifikace HTTP odpovědi je možné nastavit kód, upřesňující hlavičky a tělo odpvědi.

4.3.3 Model

Každá aplikace je tvořena jejím modelem (tzv. aplikační model 2.1), který reprezentuje specifickou doménu informací - např. ve firmě udržují seznam zaměstnanců a projekty, na kterých pracují. Znalost aplikačního modelu umožní zpracování

dotazů (např. deserializaci reprezentace zdroje nad modelem definovanou třídou) a vykonání akcí nad daným zdrojem. Akce obvykle odpovídají CRUD operacím 2.2.4, ale rozhodně to není konečný seznam akcí, které lze s objekty provádět.

Abychom byli schopni automaticky vytvářet zdroje 2.3, potřebujeme znát model aplikace (objekty a jejich vlastnosti). K popisu modelu slouží metamodel, který definujeme jako množinu entit reprezentující objekty. Každá entita má sadu vlastností, které dělíme na atributy a vztahy. Atributy slouží k popisu entity, např. entita 'Employee' by mohla mít tyto atributy: 'firstName', 'lastName', 'job'. Vztahy reprezentují spojení mezi entitami a umožňují tak propojitelnost 2.2.3 mezi jednotlivými zdroji, např. entita 'Employee' by měla vztah 'activeProjects' k entitě 'Project', a naopak 'Project' by měl definován vztah 'assignedEmployees'. Kompletní struktura metamodelu je znázorněna na obrázku 4.2.



Obrázek 4.2: Diagram metamodelu

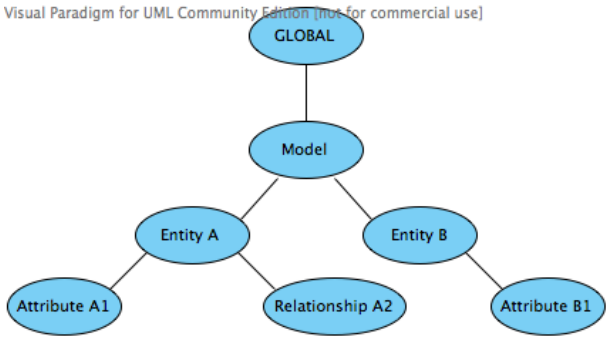
4.3.4 Konfigurace

Proto, abychom mohli měnit chování filtrů při zpracování dotazů nad jednotlivými zdroji, aniž bychom museli zasahovat do jejich implementace, definujeme konfiguraci modelu. Konfigurace nám tak zajistí přizpůsobitelnost různým nastavením 2.1.1 požadovanou v analýze.

Kombinací bezpečnosti 2.1.1 a jednotného rozhraní 2.2.4 vznikají různé možnosti zpracování dotazu. Potřebujeme-li např. dovolit zpracování dotazu 'GET' nad zdroji 'Employee' a 'Project' všem a ostatní metody pouze určité skupině lidí (např. administrátorovi), použijeme konfiguraci. Jelikož si uživatelé mohou vybírat, v jakém formátu chtějí se zdroji pracovat 2.1.1 - chceme-li tak přijímat data ve formátu XML a JSON, ale odesílat pouze v XML, musíme opět použít konfiguraci. Aby takto konfigurace fungovala, je nutné zohlednit tuto vlastnost v implementaci jednotlivých filtrů, které se musejí přízpůsobovat danému nastavení. Každý filtr tak musí zveřejnit svoje možnosti nastavení, kterými je možné měnit jeho chování.

Konfigurace je reprezentována hierarchickou strukturou, která odpovídá struktuře metamodelu. Tato hierarchie umožňuje mít jednu globální konfiguraci platnou pro celý aplikační model, a poté upravovat konfiguraci pro jednotlivé entity, či její atributy, resp. vztahy. Opět zde využijeme návrhový vzor Řetěz odpovědnosti, který nám zajistí hodnoty konfigurace od nejvíc specifických (u vlastností

entity) až po jejich obecné (univerzální) platné pro všechny. Uživatel se pak sám rozhodne podle požadavků aplikace, jaké hodnoty budou specifické a jiné globální. Na obrázku 4.3 je znázorněna hierachie konfigurace.



Obrázek 4.3: Hierarchie konfigurace

4.4 Adaptivita

Pojmem adaptivita rozumíme přizpůsobení se změnám v prostředí. V kontextu RESTového rozhraní to znamená, že filtry se přizpůsobují uživatelským požadavkům na model a jeho konfiguraci, viz 2.1.1. Základním požadavkem je model sám o sobě - uživatel se rozhoduje, které entity chce zpřístupnit, a které nikoliv, to samé platí pro atributy entit a vztahy mezi nimi. Z rozdílných požadavků na model, entity, vlastnosti, se sestaví konfigurace modelu, která umožňuje filtrům se přizpůsobovat daným dotazům. Pro zjištění uživatelských požadavků slouží inspektor.

Inspektor je odpovědný za vytvoření metamodelu a konfigurace modelu, jelikož je ale možné tuto akci provést různými způsoby (např. explicitně v kódu, konfigurační soubory, reflexe, anotace v Javě, apod.), je nutné, aby byl inspektor přizpůsobitelný daným požadavkům uživatele. To je zajištěno delegováním určitých činností na posluchače při inspekci modelu, resp. konfigurace. Tento způsob je definován jako návrhový vzor posluchač (‘Listener‘, či ‘Observer‘) [6]. V tomto případě je ale vhodné pojmenovat tento vzor jako delegát (‘Delegate‘), protože není jenom posluchačem inspektora, ale je na něj delegována určitá část práce.

4.4.1 Inspekce modelu

Při tvorbě metamodelu vycházíme z existujícího aplikačního modelu, který pomocí inspkektora na tento metamodel převádíme, což v podstatě odpovídá transformaci jednoho modelu na druhý definovaný v [2]. Základem inspekce modelu je tedy procházení tříd a jejich metod inspektorem, který vytváří entity, jejich atributy a vztahy. Inspektor nemá na starosti samotnou tvorbu entit a jejich vlastností, ale tato funkcionalita je delegována na posluchače. Inspektor pouze sestaví daný model podle získaných entit a jejich vlastností od posluchače. Tento způsob delegace umožňuje rozšiřitelnost různými způsoby inspekce pro danou platformu, např. v Javě je možné využít reflexi a anotace.

4.4.2 Inspekce konfigurace

Proces inspekce konfigurace opět deleguje vytváření konfigurací na svého posluchače, je zde tedy možnost různých implementací získání konfigurace podle použité platformy (např. externí soubory, anotace v Javě, apod.). Inspekce je závislá na vytvořeném metamodelu z předchozího textu a probíhá ve čtyřech krocích. (1) Inspektor získá od posluchače globální konfiguraci. (2) Poté následuje konfigurace daného modelu. (3) Inspektor projde všechny entity a ke každé si vyžádá od posluchače konfiguraci. (4) Nakonec projde v entitě atributy a vztahy, ke kterým si opět vyžádá danou konfiguraci.

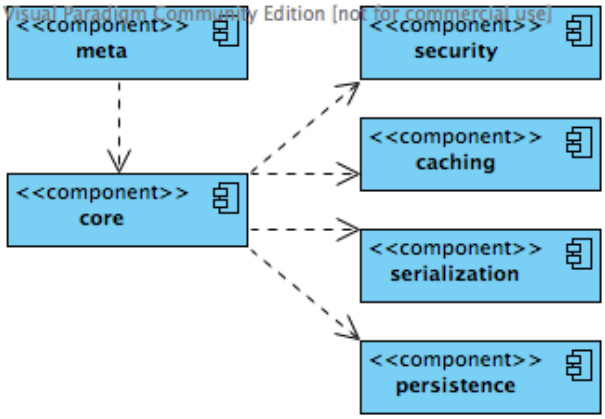
4.5 Shrnutí

Tato kapitola navrhuje vlastní řešení zpracování dotazů, a to pomocí řetězce filtrů, kde každý filtr má svoji roli - logiku, kterou aplikuje na dotaz s využitím znalostí o modelu a jeho konfigurace. Kromě modelu a konfigurace tvoří kontext filtrů ještě samotný HTTP dotaz a odpověď. Dále jsme navrhli sadu základních filtrů, vycházející z požadavků analýzy.

5. Implementace

Projekt je implementován v jazyce Java verse 1.7 [21] s využitím správce projektů Maven [30], který určuje strukturu projektu a mimo jiné se stará o sestavení aplikace, spouštění testů, správě závislostí a další. Zdrojové kódy jsou pod kontrolou distribuované správě versí, systémem Git [31]. Knihovnu je tak snadné otevřít v jakémkoliv IDE, které pracuje s projekty založenými na Mavenu, a přispívat do kódu de facto standardním způsobem. Kód projektu je uvolněn pod svobodnou licencí LGPL [19]. Veškerý zdrojový kód je obsažen v přízoze 4.

Knihovna je rozdělena do samostatných modulů 5.1, kde základem jsou moduly ‘meta‘ (tvorba metamodelu a konfigurace) a ‘core‘ (definice filtru a kontextu). Každá logická oblast definovaná v návrhu má svůj modul, který staví na tomto základu. Všechny moduly přidávají svůj konkrétní filtr, který upřesňuje implementaci s využitím kontextu (HTTP, modelu i konfigurace) pro danou oblast, která vychází z potřeb analýzy 2.1.1.



Obrázek 5.1: Rozdělení knihovny na moduly

5.1 Meta

Modul ‘meta‘ je použit pro stavbu metamodelu s entitami a jejich atributy a vztahy pro popis aplikačního modelu, a také pro vytvoření konfigurace pro daný model. Za proces vytvoření modelu a konfigurace je zodpovědný inspektor (‘Inspector‘). Jelikož je konfigurace budována z modelu, je nutné napřed provést inspekci modelu a poté konfigurace, než se ale pustíme do vysvětlení samotného procesu inspekce modelu, resp. konfigurace, je třeba popsat jednotlivé třídy pro popis modelu, resp. konfigurace.

Model je reprezentován třídou ‘model.Model‘, která obsahuje jméno modelu a kolekci jeho entit. Tato i zbývající třídy v balíčku ‘model‘ jsou implementovány jako neměnné, to znamená, že po vytvoření instance je celý model neměnný. Hlavní výhodou neměnnosti objektů je jejich bezpečnost při vícevláknovém použití, protože jiné vlákno nemůže tento objekt změnit za běhu aplikace.

Entita definovaná třídou ‘model.Entity‘, představuje šablonu pro zdroje. Zdroj je pak v Javě reprezentován instancí konkrétní třídy. Máme-li tedy např. modelovou třídu ‘com.examole.Employee‘, pak její instance jsou zdroje, se kterými pracuji pomocí šablony - entity. Každá entita má své jednoznačné jméno v modelu, odkaz na třídu, kterou reprezentuje a kolekci vlastností.

Vlastnosti entity (‘model.Property‘) představují její atributy (‘model.Attribute‘) a vztahy (‘model.Relationship‘). Každá vlastnost je reprezentována svým jménem, metodou ‘getter‘, která slouží k získání hodnoty dané vlastnosti, a metodou ‘setter‘, kterou naopak používáme pro nastavení hodnoty. U atributů rozlišuji jejich datový typ (typicky ‘String‘, ‘Integer‘, ‘Date‘, ...) a je-li atribut entity označen za primární, který umožňuje jednoznačnou identifikaci zdroje.

Vztahy se od atributů liší tím, že neodkazují na „primitivní“ datové typy (nemusí se výslovně jednat o primitivní datový typ v Javě, např. ‘java.lang.Date‘ je brán jako atribut, ne jako vztah, i když se jedná o objekt), ale na ostatní entity. Vztah si tedy udržuje informaci o cílové entitě, tzv. ‘target entity‘. Důležitý pro vztahy je význam entity k druhé, zde mohou nastat dvě možnosti: (1) entita obsahuje odkaz na instance jedné entity (‘RelationshipType.ToOne‘), (2) odkazuje na kolekci instancí dané entity (‘RelationshipType.ToMany‘).

Inspekci modelu provádí inspektor nad konkrétním balíčkem určeným uživatelem. Pomocí reflexe zjistí všechny třídy, které nemají žádné potomky. Poté projde každou třídu a požádá svého posluchače (objekt implementující rozhraní ‘ModelInspectionListener‘) o entitu k dané třídě.

Když zná inspektor všechny entity modelu, přistoupí ke zjištění vlastností. Projde všechny entity a opět pomocí reflexe vytvoří ke každé entitě seznam kandidátů na její vlastnosti. Tento kandidát je trojicí (‘reflection.Triplet): (1) fieldu, (2) getteru a (3) setteru. Povinná je přítomnost getteru a, nebo setteru, field je nepovinný. Poté požádá posluchače o atribut, případně vztah ke kandidátovi, nakonec vrátí uživateli kompletní model.

Tento postup byl zvolen proto, aby inspektor nemusel rozlišovat mezi atributem a vztahem, kdy k této činnosti je potřeba složitější práce s reflexí a i tak nemusí být 100%, jelikož může být typem atributu i třída. Inspektor je tak nezávislý na konktrétní reprezentaci modelu - lze použít např. JPA, tak i vlastní řešení tím, že uživatel dodá svého posluchače.

Konfigurace jakožto rozhraní ‘configuration.Configuration‘ umožňuje získat hodnotu konfigurace od dané úrovně. Podle definice modelu rozlišuji tyto úrovně konfigurace: (1) globální, (2) modelovou, (3) entitní, (4) atributovou a (5) vztahovou. Záleží pak na konkrétní implementaci filtru, od které úrovně vyžaduje konfigurovat svoji logiku. Rozhraní je implementováno třídou ‘configuration.Pack‘, která ukládá klíče a jejich hodnoty pro úrovně do mapy. Konfigurace je opět koncipována jako neměnný objekt.

Inspekce konfigurace probíhá tak, že se inspektor zeptá posluchače na seznam klíčů a hodnot (‘configuration.Variable‘) (1) globální úrovně, (2) modelu, (3) všech entit, (4) atributů a (5) vztahů těchto entit.

5.2 Core

Kód obsažený v tomto modulu pojmenovaný (‘core‘) obsahuje základní a velice abstraktní logiku pro zpracování HTTP dotazu filtrem. Samotný filtr je reprezentován abstraktní třídou ‘Filter‘, která umožňuje, aby byl dotaz zpracován konkrétním filtrem v metodě ‘process(...)‘. Dále je možné zřetězit filtry do seznamu a předávat tak zpracování dotazu na další filtr, k tomu slouží metoda ‘resign(...)‘. Metoda ‘process‘ je deklarována tak, že je schopná vyhodit výjimku ‘FilterException‘, tímto způsobem může uživatel zpracovat jakýkoliv chybový stav během zpracování dotazu.

Kontext zpracování dotazu, tak jak je definován v návrhu, obsahuje čtveřici: HTTP dotaz a odpověď, model, konfiguraci. Model a konfiguraci jsme již popsal dříve, zbývá tedy definovat HTTP dotaz a odpověď. Tato dvojice je zabalena do HTTP kontextu (‘HttpContext‘), který obsahuje jak informace o dotazu, tak slouží i k zapsání odpovědi. Informace o dotazu jsou neměnné, ale zapsání odpovědi je možné navícekrát, ovšem pouze kompletní, tedy uvedení návratového kódu, hlaviček a těla - pouze jeden filtr tedy může nastavit odpověď na dotaz. Pro předávání dočasného obsahu mezi filtry slouží proměnná ‘content‘. Pokud dojde např. k deserializaci zdroje reprezentovaného JSON objektem na instanci POJO, filtr uloží tuto instanci do zmíněné proměnné a předá řízení následujícímu filtru.

Algoritmus 5.1: Deklarace metody pro zpracování HTTP dotazu filtrem

```
abstract HttpContext process(HttpContext httpContext, Model model,
    Configuration configuration) throws FilterException
```

Důvod, proč jsme spojili dotaz a odpověď do jednoho HTTP kontextu, viz deklarace metody ‘process‘, je ten, že metoda v Javě má pouze jednu návratovou hodnotu. Tím, že každý filtr může předat zpracování dotazu na další, ale poté, co se zpracování vrátí opět k danému filtru, může aktuální filtr reagovat na vrácený kontext a změnit tak odpověď, či jinak reagovat. Dotaz v kontextu je sice neměnný, ale to neznamená, že nemůže dojít k nahrazení celé instance kontextu, který bude mít změněné hodnoty dotazu, o které by daný filtr přišel, pokud by se vracela samotná HTTP odpověď.

Pro ukázku následuje jednoduchá implementace metody ‘process‘ fiktivního filtru, který vytvoří instanci třídy ‘Hello‘ s řetězcem obsaženým v těle dotazu a předá zpracování na další filtr.

Aplikační kontext (‘ApplicationContext‘) je obálka pro uchování instancí modelu a konfigurace během života aplikace. K inicializaci modelu a konfigurace dochází zpravidla při jejím spuštění, ale přístup k nim je potřebný až v době zpracování dotazu. Jelikož způsobů zpracování HTTP dotazu je více (např. Servlet, či Netty), je k dispozici toto jednotné rozhraní, které umožní implementaci pro jednotlivá rozšíření bez zásahu uživatele, který pouze provede inicializaci kontextu.

Algoritmus 5.2: Ukázková implementace filtru

```
public class HelloFilter extends Filter {

    @Override
    public final HttpContext process(HttpContext httpContext, Model
        model, Configuration configuration) throws FilterException {
         String requestContent = httpContext.getRequestContent();

         if (requestContent.startsWith("Hello,")) {
             String name = requestContent.substring("Hello,".length()
                 );

             // set the content for next filter
             httpContext.setContent(new Hello(name));

             // resign the process to the next filter
             return this.resign(httpContext, model, configuration);

         } else {
             throw new FilterException(HttpStatus.S_400); // bad
                 request
         }
    }

}
```

5.3 Security

Podle návrhu modul poskytuje dva abstraktní filtry pro zabezpečení RESTového rozhraní: (1) ‘Authentication‘ a (2) ‘Authorization‘. Při zpracování dotazu tyto filtry volají své abstraktní metody ‘authenticate‘, resp. ‘authorize‘, které v případě nepovoleného přístupu musejí vyhodit výjimku ‘AuthenticationException‘, resp. ‘AuthorizationException‘, jinak předají zpracování dotazu následujícímu filtru.

5.4 Caching

Modul rozšiřuje filtr o kešování na straně serveru. Jelikož možností kešování je více (In-Memory, Memcahced, atd.), je implementace ponechána na konkrétnější filtry, ale filtr ‘Cache‘ přidává základní logiku. Na začátku zpracování dotazu zavolá filtr metodu ‘load(HttpContext httpContext, Model model, Configuration configuration)‘, pokud vrátí ‘true‘, tak dojde k vrácení kontextu, pokud vrátí ‘false‘, tak předá filtr zpracování dále. Získaný kontext od následujícího filtru se pokusí uložit, závoláním metody ‘save(HttpContext httpContext, Model model, Configuration configuration)‘, a nakonec vrátí kontext.

5.5 Serialization

Filtr ‘Resolver‘ je odpovědný za vyhledání konkrétní třídy z konfigurace podle hodnot hlaviček (‘Accept‘ a ‘Content-Type‘) dostupných v dotazu. Tato třída musí implementovat rozhraní ‘Serializer‘ pro (de)serializaci reprezentace zdroje

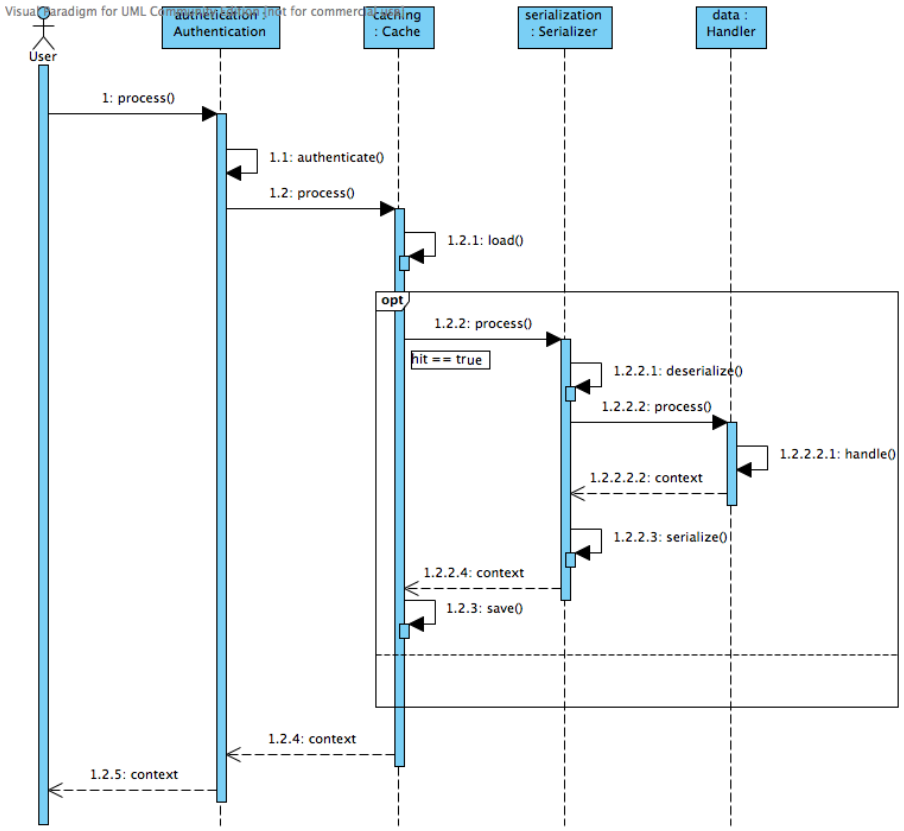
(entity), tedy její dvě metody: serialize(HttpContext httpContext, Model model, Configuration configuration), a deserialize(HttpContext httpContext, Model model, Configuration configuration)‘. Proces zpracování dotazu ve filtru je (1) deserializovat obsah těla dotazu do obsahu kontextu, (2) předat zpracování dotazu dalšímu filtru, a (3) serializovat obsah kontextu do těla odpovědi.

5.6 Data

Datový modul přidává rozhraní pro obsluhu akcí nad zdrojem. Obsluha akce je definovaná rozhraním ‘Handler‘ s metodou ‘handle(Entity entity, HttpContext context, Configuration configuration)‘, k dispozici jsou abstraktní obsluhy HTTP metod: ‘GetHandler‘, ‘PostHandler‘, ‘PutHandler‘ a ‘DeleteHandler‘. Filtr, který se stará o volání obsluhy pro danou akci, je implementován třídou ‘Dispatcher‘. Jeho úkolem je (1) určit z HTTP kontextu, o kterou entitu se jedná, (2) najít v konfiguraci obsluhu pro danou entitu a HTTP metodu a (3) předat zpracování akce nad entitou obsluze. Pokud se nepovedlo určit entitu, či nalézt obsluhu, je vyhozena výjimka. Opět je více způsobů, jak pracovat se zdroji, proto je potřeba dodat konkrétní implementaci obsluhy, např. pro práci se specifikací JPA.

5.7 Shrnutí

V tuto chvíli máme připravenu kostru knihovny - metamodel popisující aplikační model, schránku pro hierarchicky definovanou konfiguraci, jejich inspekce, a základní filtry navržené v předchozí kapitole, které tvoří řetěz pro zpracování dotazu 5.2. Tyto filtry upřesňují použití v dané oblasti, ale stále jsou abstraktní na konkrétní využití, pro funkční ukázku je nutné doplnit tyto filtry o finální implementace.



Obrázek 5.2: Zřetězené filtry

6. Testování

V závěrečné kapitole se budeme věnovat ověření funkčnosti navrženého způsobu zpracování dotazů. K tomu nám pomůže ukázková aplikace, která slouží jako test použitelnosti. Pro otestování kódu byla použita knihovna TestNG [34], umožňující krom jiného psát jednotkové a integrační testy.

6.1 Jednotkové testování

Základní moduly ‘core‘ a ‘meta‘ obsahují jednotkové testy (‘unit tests‘), tedy takové testy, které ověřují funkčnost jednotlivých částí kódu - objektů. Jednotkové testy jsou na sobě nezávislé, pokud je nutné dodat určité objekty, bez kterých se testovaná jednotka neobejde, používají se pomocné objekty (‘mock objects‘).

Následující ukázka 6.1 jednotkového testu pro inicializaci aplikačního kontexu, který je inicializován, pokud byly použity objekty modelu a konfigurace.

Algoritmus 6.1: Jednotkový test inicializace aplikačního kontextu

```
@Test
public void testInitializeValid() throws Exception {
    Model model = new Model("model", new LinkedList<Entity>());
    Configuration conf = new Pack();

    ApplicationContext.initialize(model, conf);

    ApplicationContext context = ApplicationContext.getInstance();
    assert (context.isInitialized() == true) : "Application context
        should be initialized.";
}
```

6.2 Integrační testování

Integračního testování bylo použito na ověření inspekce metamodelu a konfigurace, protože proces inspekce vyžaduje ke své práci definovaný model a posluchače zajišťující sestavení metamodelu konfigurace. Byly definovány kombinace modelu, které využívají základní možnosti objektově-orientovaného programování: třídy a metody, abstraktní třídy a metody, rozhraní.

TestNG umožňuje vytvořit testovací sadu, která poskytuje data k testování. V našem případě inspekce se tak jedná o název balíčku, ve kterém jsou umístěny třídy modelu, a dvojice posluchačů pro model a konfiguraci. Testy slouží ke kontrole vytvořeného modelu a konfigurace.

6.3 Testovací aplikace

Pro otestování použitelnosti byl zvolen scénář sestavení ukázkové aplikace. Aby mohla být tato aplikace sestavena, bylo nutné implementovat sadu jednoduchých

Algoritmus 6.2: Poskytovatel dat pro testování

```
@DataProvider(name = "packages")
public static Object[][] packages() {
    TestInspectionListener listener = new TestInspectionListener();

    return new Object[][] {
            { "cz.cvut.fel.adaptiverestfulapi.meta.data.simple",
              Object.class, listener, listener},
            { "cz.cvut.fel.adaptiverestfulapi.meta.data.base", Item.
              class, listener, listener },
            { "cz.cvut.fel.adaptiverestfulapi.meta.data.abstracts",
              Object.class, listener, listener },
            { "cz.cvut.fel.adaptiverestfulapi.meta.data.interfaces",
               Object.class, listener, listener }
    };
}
```

filtrů, posluchače inspekce modelu a konfigurace.

6.3.1 Zabezpečení

Autentizace uživatelů probíhá pomocí „basic“ schématu definovaného v protokolu RFC-2617 [?]. Toto schéma vyžaduje, aby se uživatel ověřoval svým identifikátorem a heslem pro každou aplikaci (tzv. realm) zvlášť (realm slouží pouze pro rozlišení více aplikací na jednom serveru). Na každý neautorizovaný dotaz vrací server odpověď s kódem 401 (Unathorized) a hlavičkou: ‘WWW-Authenticate: Basic realm="com.example„. Pro autentizaci zasílá uživatel dotaz na server s hlavičkou ‘Authorization: Basic hash‘, kde ‘hash‘ je zakódovaný řetězec se jménem a heslem odděleným ‘:‘ (např. ‘admin:1234‘) pomocí ‘base64‘ [?].

Toto zabezpečení je implementováno filtrem ‘basic.BasicAuthentication‘ z modulu ‘security‘, který z dotazu určí uživatelské jméno a heslo a porovná je s uloženými údaji. Uživatelská jména a hesla jsou pro jednoduchost načtena do paměti při startu aplikace do mapy, je však možné načítat data např. z databáze apod.

6.3.2 Serializace

Pro reprezentaci zdrojů byla implementována serializaci do formátu JSON, což je textový formát pro strukturovaná data, který je odvozen od zápisu objektů v ECMAScript [28] jazyce. Pracuje s jednoduchými datovými typy: řetězce, čísla, boolean, a null a složenými typy: pole a objekt.

Třída ‘application.json.JsonSerializer‘ používá externí knihovnu Gson [29], která automaticky serializuje základní datové typy a pro vlastní objekty je potřeba implementovat rozhraní ‘com.google.gson.JsonSerializer‘ a ‘com.google.gson.JsonDeserializer‘. Pro každou entitu je tak zaregistrována dvojice tříd ‘application.json.EntitySerialiser‘ a ‘application.json.EntityDeserializer‘ zajišťující spolupráci s knihovnou Gson.

Při serializaci se vytvoří prázdný JSON objekt, který se naplní atributy a vztahy entity. Jelikož jsou atributy „primitivní“ datové typy, lze je snadno přidat do JSON objektu jako ‘klíč:hodnota‘, kde klíč je jméno atributu. Hodnota se zís-

ká zavoláním metody ‘getter‘ nad instancí objektu z modelu. Pro vztahy ‘toOne‘ je použit primární atribut (identifikátor), v JSON objektu bude reprezentován jako ‘klíč:identifikátor‘. Vztah ‘toMany‘ projde všechny instance a uloží si jejich identifikátory do pole, které následně vloží do JSON objektu jako ‘klíč:[pole identifikátorů]‘.

Algoritmus 6.3: Ukázka reprezentace entity ve formátu JSON

```
{
    "id": 1,
    "name": "Project",
    "started": false,
    "startedAt": null,
    "manager": 4,
    "tasks": [ 1, 2, 3 ]
}
```

Druhou, speciální reprezentací entity, je její popis z metamodelu v textové podobě. Tato serializace převede jméno entity, reprezentující třídu, její atributy a vztahy s jejich parametry do textu.

6.3.3 Kešování

Na straně serveru je kešování implementováno s využitím HTTP hlavičky ‘If-Modified-Since‘. Klient posílá dotazy ‘GET‘ s uvedenou hlavičkou, která obsahuje datum, kdy došlo k poslednímu dotazu na daný zdroj. Server buď vrací odpověď ‘304 Not Modified‘ s prázdným tělem, nebo běžnou odpověď (‘200 OK‘) s požadovanou reprezentací zdroje. Třída ‘IfModifiedSinceCache‘ udržuje v paměti mapu ‘klíč:hodnota‘, kde klíčem je jméno entity a identifikátor a hodnotou je časová známka, která udává vložení klíče do mapy. Z toho vyplývá, že keš je omezena pouze na dotazy s konkrétním identifikátorem, např. ‘GET /users/123‘. Při každém dotazu se filtr podívá do mapy na hodnotu daného klíče, mohou tak nastat celkem tři situace: (1) hodnota pro klíč v mapě není -> zpracování filtru pokračuje normálně (2) časová známka je v mapě, ale hodnota je novější než je uvedena hlavičce -> stejné jako případ (1), (3) časová známka je v mapě, ale hodnota je starší než je uvedena v hlavičce -> dojde k zastavení zpracování dotazu a vrátí se odpověď ‘304‘.

Při prvním dotazu ‘GET‘ na zdroj s identifikátorem, nebo při dotazu ‘POST‘ se uloží časová známka, která se později revaliduje s každým dotazem ‘PUT‘, nebo odstraní při dotazu ‘DELETE‘.

6.3.4 Práce s daty

Persistence dat je v Javě řešena standardizovaným způsobem pomocí JPA. Tato specifikace popisuje práci s relačními daty a mapování na objekty jazyka (ORM). Pro vykonávání základních CRUD operací nad JPA slouží ‘EntityManager‘, v následující tabulce 6.1 je uvedeno mapování HTTP metod na akce prováděné ‘EntityManager‘em.

Tabulka 6.1: Mapování HTTP metod na persistentní akce

HTTP metoda	URL	Akce
GET	/resource	createQuery(Object clazz)
GET	/resource/123	find(Object clazz, Object id)
POST	/resource	persist(Object obj)
PUT	/resource/123	merge(Object obj)
DELETE	/resource/123	remove(Object obj)

Tuto obsluhu implementují třídy ‘Get,Post,Put,DeleteHandler‘ v balíčku ‘persistence‘ z datového modulu.

6.3.5 Servlet

Tento modul umožňuje použití RESTové knihovny v prostředí Java EE [22], které disponuje třídou ‘javax.servlet.http.HttpServlet‘ zajišťující zpracování HTTP dotazu. Třída ‘FilteredServlet‘ dědí od zmíněné třídy a přepisuje metodu ‘service(HttpServletRequest req, HttpServletResponse resp)‘ pro zpracování dotazů. Při každém HTTP dotazu dojde k zavolání této metody, která (1) vytvoří instanci HTTP kontextu podle ‘HttpServletRequest‘. (2) Zavolá metodu ‘process‘ nad filtrem s HTTP kontextem, modelem a konfigurací z aplikačního kontextu, tím dojde ke zpracování dotazu knihovnou. (4) Naplní ‘HttpServletResponse‘ odpověď podle HTTP kontextu. Pokud dojde k vyhození výjimky ‘FilterException‘ při zpracováné, je tato výjimka převedena na odpověď a zaslána klientovi. Tato třída se nestará o vytvoření řetězu filtrů, ale je nutné ho třídě dodat.

6.3.6 Model

Inspekci modelu zajišťuje posluchač ‘ModelListener‘, který používá jak reflexi, tak i anotace ze specifikace JPA. Vzhledem k tomu, že ukázková aplikace použije tuto specifikaci pro definici svého modelu, tak to není na závadu, naopak tím dojde ke zjednodušení implementace jednotlivých částí posluchače.

Jméno entity odpovídá jménu třídy - ‘getName()‘, a jméno vlastnosti je vytaženo ze trojice <field, getter, setter> (také je použito ‘getName()‘), podle toho, které hodnoty jsou přítomny.

Datové typy atributů jsou určeny reflexí v závislosti, jedná-li se o ‘field‘ - ‘getGenericType()‘, ‘getter‘ - ‘getGenericReturnType()‘, nebo ‘setter‘ - ‘getGenericParameterTypes()[0]‘.

Primární atribut je detekován pomocí anotace ‘javax.persistence.Id‘. Tato anotace je povinná při definici entity v JPA. Jako ukázka jiného přístupu je implementováno pomocí rovnosti jména atributu s hodnotou řetězce ‘id‘.

Typ vztahu se v ukázkové aplikaci opět určí anotacemi specifikujícími vztahy v prostředí JPA. V následující tabulce 6.2 je uvedeno mapování vztahů JPA na vztahy metamodelu.

Tabulka 6.2: Mapování JPA vztahů na vztahy metamodelu

JPA	Meta
javax.persistence.OneToOne	meta.model.RelationshipType.ToOne
javax.persistence.ManyToOne	meta.model.RelationshipType.ToOne
javax.persistence.OneToMany	meta.model.RelationshipType.ToMany
javax.persistence.ManyToMany	meta.model.RelationshipType.ToMany

Toto zjednodušení si můžeme dovolit, protože vztahy JPA jsou důležité z hlediska implementace databáze, kde se rozlišuje, která tabulka bude obsahovat odkaz na jinou tabulku - použití cizího klíče. Reprezentace zdroje, v našem případě JSON, určuje vztah s pomocí jednoho primárního klíče (‘toOne‘), nebo pole primárních klíčů (‘toMany‘). Pro ukázku je uvedena i implementace pomocí reflexe, která využívá typu ‘trojice‘ obdobně jako při určení typu atributu.

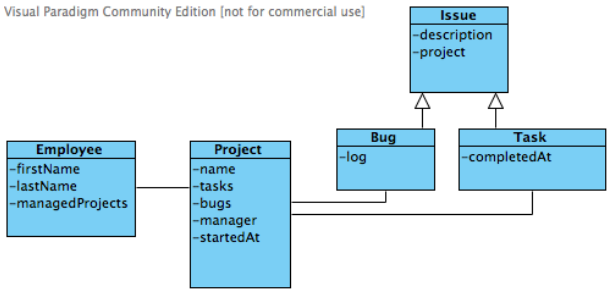
Cílová entita je určena opět pomocí anotací JPA pro vztahy. Tyto anotace mají parametr ‘targetEntity()‘ vracející třídu entity, podle které je dohledána entita v metamodelu. Je možné využít i reflexi, ale ta má své omezení - cílové třídy musejí být konkrétní třídy, nelze použít rozhraní (‘interface‘), abstraktní třídu (‘abstract class‘) a ani rodičovskou třídu, protože typ cílové třídy musí být instancovatelný (nutné např. při deserializaci) a jednoznačný, což v případě rodičovské třídy, která může mít více potomků, není možné.

6.3.7 Konfigurace

Kvůli jednoduchosti aplikace je konfigurace definována přímo ve třídě posluchače ‘ConfigurationListener‘. Na globální úrovni jsou tak nastaveny obsluhy pro zpracování dat pomocí JPA a serializace do JSON a textu. Pro serializaci do formátu JSON jsou entity a jejich vlastnosti autorizovány - to znamená, že je možné pro určité uživatele skrývat entity, resp. vlastnosti entit v jejich reprezentaci.

6.3.8 Ukázková aplikace

Nyní máme všechny potřebné stavební kameny připravené a můžeme tak zkonstruovat naši ukázkovou aplikaci. Tato aplikace simuluje jednoduchý systém na správu projektů, aplikační model znázorněný na obrázku **??** tak tvoří třídy ‘Project‘, ‘Issue‘ -> ‘Task‘, ‘Bug‘ a ‘Employee‘ a vztahy mezi nimi, které jsou znázorněny na obrázku X.Y. Entity metamodelu odpovídají svým třídám, vynecháme pouze třídu ‘Issue‘, jelikož se jedná o rodičovskou třídu se dvěma potomky. K persistence modelu je použito Hibernate [32], tedy konkrétní implementace JPA, a databáze Derby [33], která je pro jednoduchost aplikace umístěna v paměti aplikace.



Obrázek 6.1: Model ukázkové aplikace

Pro zabezpečení aplikace je použita HTTP Basic autentizace se dvěma uživateli: administrátor (‘admin‘) a uživatel (‘client‘). Autorizace je řešena na úrovni HTTP metod, kdy jsou klientovi dovoleny pouze dotazy ‘GET‘, administrátor smí vše. Při serializaci dojde k vynechání atributu ‘startedAt‘ u všech entit, které ho obsahují, dotazuje-li se uživatel. Definice uživatelů a autorizačních pravidel je uvedena ve třídě ‘Users‘.

Inicializace aplikačního kontextu je zajištěna při inicializaci servletového kontextu. Třída ‘ApplicationContextListener‘ implementuje potřebné rozhraní ‘ServletContextListener‘ a při startu aplikace tak načítá persistentní kontext s předpřipravenými daty a provádí inspekci modelu s konfigurací. Registrace posluchače je standardně uvedena v konfiguračním souboru ‘web.xml‘.

Posledním krokem je sestavení posloupnosti filtrů pro zpracování dotazů. V inicializaci třídy ‘ExampleServlet‘, která dědí od ‘FilteredServlet‘, dojde ke spojení celkem pěti filtrů pro: (1) autentizaci, (2) autorizaci, (3) kešování, (4) serializaci, a (5) persistenci dat. Tento servlet je zaregistrován v konfiguračním souboru ‘web.xml‘ webové aplikace.

Nezbývá než spustit aplikaci na libovolném aplikačním serveru, který splňuje specifikaci Java EE a začít s dotazováním. Následuje jedno ukázkové volání na server, kdy se jako administrátor zajímáme o zdroj ‘Project‘ s identifikátorem ‘2‘ v JSON reprezentaci.

Algoritmus 6.4: Odpověď aplikace na HTTP dotaz

```
$ curl -u admin:1234 --header "Accept: application/json;charset=UTF
    -8" --header "Content-Type: application/json;charset=UTF-8" http
    ://localhost:8080/Project/2

{
    "started": false,
    "id": 2,
    "startedAt": null,
    "name": "Project B",
    "bugs": [ 2, 3 ],
    "manager": 4,
    "tasks": [ 3 ]
}
```

6.4 Shrnutí

Ukázkovou aplikací bylo ověřeno, že je možné implementovat navržený způsob zpracování HTTP dotazů. S výchozí implementací filtrů a inspekcí modelu bylo nakonfigurováno chování RESTového rozhraní, které je schopné reagovat např. na uživatele, který se identifikuje v dotazu, či nabízet zdroj v různých reprezentacích. V tuto chvíli je tak oddělena logika zpracování dotazu od její konfigurace, což byl jeden z hlavních problémů uvedených v analýze, např. doplnění serializace entit do formátu XML je přidáním jednoho řádku do konfigurace (je-li samozřejmě k dispozici jeho implementace).

Jedním z limitů tohoto přístupu je provázanost modelu s logikou některých filtrů, např. serializací. Jelikož je model strukturovaný, tak je dobře použitelný se strukturovanými reprezentacemi (např. JSON, XML, HTML, apod.). Pokud by ale entita reprezentovala např. obrázek uložený na serveru, mohla by mít atribut ‘path‘, který by obsahoval cestu k fyzickému souboru. To, že je atribut použit jako cesta k souboru, je však informace nad rámec definovaného metamodelu. Toto ovšem není neřešitelná překážka, jen je třeba navrhnout vhodné rozhraní a doplnit její implementaci.

Podobným problémem je i provázanost modelu s URL, na kterou opět navazuje i logika filtrů. V současnosti si implementace vystačí s dvojicí parametrů entita, identifikátor v URL (např. ‘com.example/project/1‘), tyto parametry jsou však definované napevno. Bylo by tak vhodné umožnit uživateli specifikovat vlastní podobu URL pomocí jednoduchého jazyka, přičemž by filtry mohl definovat, co od této URL požadují, např. datový filtr zajímá jméno entity a případně identifikátor instance, definice URL by pak mohla vypadat nějak takto: ‘api/v1/entity/id‘, a obsluha (‘router‘) by zajistila dodání hodnot jednotlivých parametrů filtrů na vyžádání.

Závěr

Navržené řešení bylo implementováno a otestováním ukázkovou aplikací můžeme tedy označit cíl - automaticky generovat RESTful rozhraní - za splněný. Důležitou podmínkou splnění úkolu bylo nalezení mapování aplikačního modelu na rozhraní, resp. entity z modelu na zdroje rozhraní. Vybudování metamodelu a konfigurace umožnilo oddělit kód logiky zpracování dotazů od požadavků kladených na jednotlivé elementy modelu podle specifikace aplikace. Dalším výrazným bodem v návrhu je zřetězení filtrů, aniž by bylo nutné vkládat speciální objekt reprezentující zdroj tak, jak to vyžaduje specifikace JAX-RS.

V tuto chvíli je největším omezením záběr implementace jednotlivých filtrů, přece jenom dohnat vývoj knihoven, který trvá i několik let, je časově náročný úkol, ale zároveň je zde i veliký potenciál pro budoucí rozšíření knihovny. Minimálně se tak nabízí pro studenty možnost doplnit některou dílčí implementaci filtrů dané logiky a obohatit tak projekt. V oblasti zabezpečení je zde prostor pro nové způsoby autentizace pomocí protokolu OpenID [44], či implementovat autorizaci protokolem OAuth [43]. Dále je možné doplnit podporu pro různé formáty serializace, např. XML, a přidat podporu i pro nestrukturované reprezentace jako jsou např. obrázky.

Zajímavějším, ale o to složitějším námětem, může být automatické generování presentační vrstvy (UI) na straně klienta s využitím přístupu AOP, který poskytuje komponenty a aspektově orientované jazyky s různými úrovněmi abstrakce a možnostmi pro sestavení výsledného systému. Komponenty jsou takové části systému, pro které je jednoduché zapouzdřit implementaci v obecné proceduře, kdežto aspekty naopak nejsou, a které představují tzv. business pravidla dané aplikace (např. validace, integritní omezení). Cílem je tak oddělení kódu od těchto pravidel, které nám poté umožní generovat uživatelské rozhraní.

Reference

[1] Richardson, Leonard; Ruby, Sam. *RESTful web services*. O'Reilly, 2008. ISBN 978-059-6554-606.

[2] Kleppe, Anneke G., et al. *The model driven architecture: practice and promise*. Addison Wesley, 2003. ISBN 03-211-9442-X.

[3] Czarnecki, Krzysztof; Eisenecker, Ulrich W. *Generative programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Technical University of Ilmenau, 1998.

[4] Cemus, Karel; Cerny, Tomas. *Aspect-Driven Design of Informations Systems*. Springer International Publishing, 2014. ISBN 978-3-319-04297-8.

[5] Fielding, Roy Thomas *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000. ISBN 0-599-87118-0.

[6] Pecinovský, Rudolf. *Návrhové vzory*. 1. vydání. Brno: Computer Press, a.s., 2007. ISBN 978-80-251-1582-4.

[7] Selic, Bran *The Pragmatics of Model-Driven Development*. IEEE Computer Society, 2003.

[8] Booth, D.; Haas, H.; McCabe, F.; Newcomer, E.; Champion, M.; Ferris, C.; Orchard, D. *Web Services Architecture*. World Wide Web Consortium (W3C®), 2004.

[9] Crockford, D. *RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)*. The Internet Engineering Task Force (IETF®), 2006.

[10] Fielding, R., et al. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*. The Internet Engineering Task Force (IETF®), 1999.

[11] Franks, J., et al. *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication*. The Internet Engineering Task Force (IETF®), 1999.

[12] Berners-Lee, T.; Connolly, D. *RFC 1866: Hypertext Markup Language - 2.0 (HTML)*. The Internet Engineering Task Force (IETF®), 1995.

[13] ISO *"Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)", ISO 8879:1986*. International Organization for Standardization (ISO®), 1986.

[14] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; Maler, E.; Yergeau, F. *Extensible Markup Language (XML) 1.0*. 5. vydání. World Wide Web Consortium (W3C®), 2008.

[15] Pemberton, S.; Austin, D.; Axelsson, J.; Çelik, T.; Dominiak, D.; Elenbaas, H.; Epperson, B.; Ishikawa, M.; Matsui, S.; McCarron, S.; Navarro, A.; Peruvemba, S.; Relyea, R.; Schnitzenbaumer, S.; Stark, P. *XHTML™ 1.0 The Extensible HyperText Markup Language*. 2. vydání. World Wide Web Consortium (W3C®), 2002.

[16] Berners-Lee, T.; Masinter, L.; McCahill, M. *Uniform Resource Locators (URL)*. The Internet Engineering Task Force (IETF®), 1994.

[17] Josefsson, S. *The Base16, Base32, and Base64 Data Encodings*. The Internet Engineering Task Force (IETF®), 2006.

[18] Box, D.; Ehnebuske, D.; Kakivaya, G.; Layman, A.; Mendelsohn, N.; Nielsen, H. F.; Thatte, S.; Winer, D. *Simple Object Access Protocol (SOAP) 1.1*. World Wide Web Consortium (W3C®), 2000.

[19] Stallman, R. *GNU Lesser General Public License v3*. Free Software Foundation, 2007

[20] Information Sciences Institute *Internet Protocol*. California: University of Southern California, 1981.

[21] Reinhold, M. *JSR 336: Java SE 7*. Oracle Corporation, 2011

[22] DeMichiel, L.; Shannon, B. *JSR 342: JavaTM Platform, Enterprise Edition 7 (Java EE 7) Specification*. Oracle Corporation, 2013

[23] Pericas-Geertsen, S.; Potociar, M. *JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services*. Oracle Corporation, 2013

[24] DeMichiel, L. *JSR 338: JavaTM Persistence 2.1*. Oracle Corporation, 2013

[25] Vajjhala, S.; Fialli, J. *JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0*. Oracle Corporation, 2006

[26] Srinivasan, R. *RPC: Remote Procedure Call Protocol Specification Version 2*. The Internet Engineering Task Force (IETF®), 1995.

[27] Winer, D. *XML-RPC Specification* UserLand Software, 1999

[28] Ecma International *ECMAScript® Language Specification v5.1*. Ecma International, 2011.

[29] Google, at al. *Gson:* `https://code.google.com/p/google-gson/` [online]. Google, Inc., 2014.

[30] Apache Software Foundation *Maven:* `http://maven.apache.org` [online]. The Apache Software Foundation. 2014.

[31] Hamano, J.; Torvalds, L., at al. *Git:* `http://git-scm.com` [online]. 2014.

[32] Red Hat *Hibernate:* `http://hibernate.org/orm/` [online]. Red Hat Inc., 2014.

[33] Apache Software Foundation *Derby:* *http://db.apache.org/derby/docs/10.0/manuals/develop/develop13.html* [online]. The Apache Software Foundation, 2014.

[34] Beust, C. at al. *TestNG:* *http://testng.org/doc/index.html* [online]. Cédric Beust, 2014.

[35] Berners-Lee, T.; Cailliau, R. *WorldWideWeb: Proposal for a HyperText Project* European Organization for Nuclear Research (CERN), 1990.

[36] Amazon *S3:* *http://aws.amazon.com/s3* [online]. Amazon Web Services, Inc., 2014.

[37] Karunamurthy, V. *Delicious API:* *https://github.com/vjkaruna/delicious-api* [online]. AVOS Systems, Inc., 2014

[38] W3C; Nelson, T. *What is HyperText:* *http://www.w3.org/WhatIs.html* [online]. World Wide Web Consortium (W3C®), 2014.

[39] Oracle, at al. *Jersey:* *https://jersey.java.net* [online]. Oracle Corporation, 2014.

[40] Red Hat, at al. *RESTEasy:* *http://resteasy.jboss.org* [online]. Red Hat Inc., 2014.

[41] Christie, T., at al. *Django REST framework:* *http://www.django-rest-framework.org* [online]. Tom Christie, 2014.

[42] Noehr, J., at al. *Django Piston:* *https://bitbucket.org/jespern/django-piston/wiki/Home* [online]. Jesper Noehr, 2014.

[43] Hardt, D. *The OAuth 2.0 Authorization Framework.* The Internet Engineering Task Force (IETF®), 2012.

[44] Sakimura, N.; Bradley, J.; Jones, M.; de Medeiros, B.; Mortimore, C. *OpenID Connect Core 1.0.* The OpenID Foundation, 2014.

Seznam použitých zkratek

AOP Aspect-Oriented programmaing. 51

API Application Programming Interface. 19, 20, 22, 23, 53, 54, 56

CERN Conseil Européen pour la Recherche Nucléaire. 54

CRUD Create, Read, Update and Delete. 10, 24, 26, 28, 29, 35, 46

GNU GNU's Not Unix!. 53

HTML Hypertext Markup Language, specifikace [12]. 10, 18, 50, 52

HTTP Hypertext Transform Protocol, specifikace [10]. 1–7, 9, 10, 12–16, 18–21, 23, 24, 26, 28, 31–34, 37, 38, 40, 42, 46, 47, 49, 50, 52

HTTPS Hypertext Transfer Protocol over Secure Socket Layer. 13

IDE Integrated Development Environment. 38

IETF The Internet Engineering Task Force. 52–54

IP Internet Protocol. 13

ISO International Organization for Standardization. 52

JAX-RS Java API for RESTful Web Services. 5, 11, 19–22, 32, 34, 51

JAXB Java Architecture for XML Binding. 5, 20–22, 53

JPA Java Persistence API, specifikace [24]. 4, 29, 39, 42, 46–48

JPEG Joint Photographic Experts Group. 18

JSON Javascript Object Notation, specifikace [9]. 5, 18, 21, 22, 26, 27, 33, 35, 40, 45, 46, 48–50, 52

JSR Java Specification Requests. 53

LGPL Lesser General Public License, [19]. 38

ORM Object-Relational mapping. 28, 29, 46

POJO Plain Old Java Object. 5, 19, 20, 22, 24, 40

REST Representational State Transfer, [5]. 1, 6, 8, 9, 12, 15, 17–22, 24, 26, 29–31, 33, 36, 41, 47, 50, 51, 53, 56

ROA Resource-Oriented Architecture, [1]. 6, 9, 18

RPC Remote Procedure Call, specifikace [26]. 8, 9, 53

S3 Simple Storage Service, [36]. 9, 54

SGML Standard Generalized Markup Language, specifikace [13]. 10, 52

SOAP Simple Object Access Protocol, specifikace [18]. 9, 53

TCP Transmission Control Protocol. 13

UI User Interface. 51

UML Unified Modeling Language. 29, 30

URL Unified Resource Identifier, specifikace [16]. 7, 9, 13, 17–19, 21, 23, 24, 26, 29, 47, 50, 53

W3C World Wide Web Consortium. 52–54

XHTML Extensible HyperText Markup Language, specifikace [15]. 10, 53

XML Extensible Markup Language, specifikace [14]. 9, 10, 15, 21, 26, 33, 35, 50–53

YAML YAML Ain't Markup Language. 26

Přílohy

Přílohy jsou umístěné na přiloženém médiu.

1. Zadání práce (soubor ‘task.png‘)
2. Zdrojový text práce ve formátu LaTeX (složka ‘/thesis‘)
3. Práce ve formátu PDF (soubor ‘/thesis.pdf‘)
4. Zdrojový kód (složka ‘/code‘) s dokumentací (soubor ‘/code/README.md)