České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Tomáš Jiříček**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Nástroj pro generování dokumentace vzdálených rozhraní (API) webových služeb aplikací Java Enterprise Edition**

Pokyny pro vypracování:

Cílem práce je vyvinout nástroj pro generování dokumentace vzálených rozhraní (API) webových služeb aplikací Java Enterprise Edition.
Hlavními úkoly práce jsou:

1. Seznamte se a nastudujte kontext webových služeb REST a WS API, založených na specifikacích JAX-RS a JAX-WS používaných v jazyce Java.
2. Seznamte se s Apache Maven, zejména s vývojem zásuvných modulů.
3. Analýzujte existující řešení a požadavky na nástroj.
4. Navrhněte, implementujte a otestujte na open-source projektu nástroj pro generování dokumentace vzálených rozhraní (API) webových služeb jako zásuvný modul pro Maven.
5. Zásuvný modul bude umožňovat zaintegrování do aplikace bez změny jejího zdrojového kódu.
6. Výstupem modulu bude dokumentace služeb s ukázkovým klientem v HTML s CSS a JavaScript.
7. Porovnejte a vyhodnoťte nástroj s existujícími řešeními.
8. Výsledkem práce bude vlastní nástroj pro generování dokumentace společně s vygenerovanou dokumentací existujícího open-source systému.

Seznam odborné literatury:

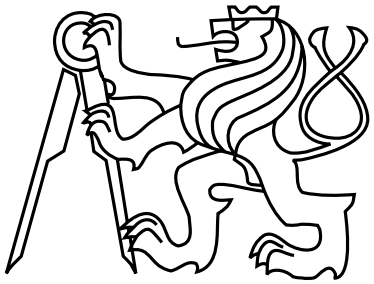
Sonatype Company. 2008. Maven: The Definitive Guide, 1st Edition (First ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN 978-0-596-51733-5.
Bill Burke. 2013. RESTful Java with JAX-RS 2.0, 2nd Edition. O'Reilly Media. ISBN 14-493-6134-X.
Brett McLaughlin. 2002. Java and XML Data Binding. O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN 978-0596002787.
Alessio Soldano. 2014. Advanced JAX-WS Web Services: Practical guide for creating SOAP Web Services using opensource solutions. ITBuzzPress. ISBN 9788894038903.
Bloch, Joshua. 2008. Effective java, 2nd Edition. Upper Saddle River: Addison-Wesley, ISBN 03-213-5668-3.

Vedoucí: Ing. Tomáš Černý, MSc.

Platnost zadání: do konce letního semestru 2015/2016

V Praze dne 24. 3. 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Nástroj pro generování dokumentace vzdálených rozhraní (API) webových služeb aplikací Java Enterprise Edition

Bc. Tomáš Jiříček

Vedoucí práce: Ing. Tomáš Černý, MSc.

Studijní program: Otevřená informatika, strukturovaný, Navazující magisterský

Obor: Softwarové inženýrství

11. května 2015

Poděkování

Velmi rád bych poděkoval svým rodičům za podporu při studiích a důvěru ve mě. Velký dík posílám také přítelkyni Veronice.

Dále bych rád poděkoval Mgr. Ivu Bekovi a Ing. Tomáši Černému, MSc. za připomínky a odborné rady při tvorbě této práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 10. 5. 2015 .....

Abstract

With the increasing number of software products, which have a need to be integrated in between themselves, a big pressure developed with interface unification. Therefore a row of protocols and technologies was developed, which brings rules into system integration and aproaches, which designate how the remote communication between system should work and how to create providers and clients of such interfaces. If there is a required documentation for an interface, which enables the consumer to obtain all needed information about the interface, it is fairly easy to realize the integration. In reversed case incomplete, nonactual or otherwise insufficient documentation can lead to growth of needed funds for the realization of integration.

The aim of the thesis is to analyze the requirements of the documentation and to develop a tool for automatic generation of actual documentation of remote interfaces. With the usage of the tool there will be a decrease of needed funds for the development of the documentation and for the realization of the system integration.

Abstrakt

S narůstajícím počtem softwarových produktů, které mají potřebu být mezi sebou vzájemně integrovány vznikl velký tlak na unifikaci rozhraní. Byla proto vytvořena řada protokolů a technologií, které přinášejí do systémové integrace pravidla a postupy určující, jak probíhá vzdálená komunikace mezi systémy a jak tvořit poskytovatele a klienty takových rozhraní. Pokud existuje k vystavenému rozhraní odpovídající dokumentace, která umožní konzumentovi poskytnout všechny potřebné informace o rozhraní, lze poměrně rychle a jednoduše provést integraci. V opačném případě nekompletní, neaktuální či jinak nedostačující dokumentace může způsobit nárust nákladů na realizování integrace.

Cílem práce je analyzovat požadavky dokumentace a vytvořit nástroj pro automatické generování aktuální dokumentace vzdálených rozhraní. S využitím nástroje dojde ke snížení nákladů potřebných pro tvorbu dokumentace a tím i realizaci systémové integrace.

Obsah

Seznam obrázků

Seznam tabulek

Listings

Kapitola 1

Úvod

Systémová integrace je velmi často skloňovaný pojem. Drtivá většina systémů v dnešní době není bez integrace schopna fungovat. Schopnost integrace s jinými systémy je žádoucí, protože čím více se umí software integrovat, tím je flexibilnější. Integrace znamená spojení menších komponent do vetšího celku. Takový celek potom dokáže efektivně pracovat pomocí definovaných pravidel pro komunikaci mezi subsytémy. Integraci lze vytvořit na různých systémových vrstvách.

První počátky integrace sahají do roku 1980, kdy vznikla myšlenka tzv. digitální výroby, která byla aplikována v továrnách. V devadesátých letech společnosti nakupovaly softwarová řešení jako je SAP, Oracle ERP, Siebel a další. Tato řešení fungovala dobře jako samostatný software, vytvořily se kolem nich takzvané informační ostrovy. Ve většině případů každé řešení produkovalo redundantní data. Ve výsledku, při vzniklé změně dat, musela být redundantní data ručně změněna i v ostatních systémech. Takové řešení bylo těžkopádné a musela přijít změna. Tyto problémy daly vzniknout rozmachu integrace mezi systémy.

Komunikace mezi komponentami byla často platformově závislá a používaly se pro ni proprietární protokoly. Postupem času, jak šly informační technologie dopředu a sílil tlak na vznik technologií umožňujících platformovou nezávislost s otevřenými protokoly pro komunikaci, jsme se dostali do stavu, kdy máme technologie a standardizované protokoly. S jejich pomocí vznikají unifikovaná rozhraní systémů, která zjednodušují realizování integrace mezi softwarovými komponentami.

Protokoly definují, jak probíhá komunikace a technologie definují, jak vytvářet rozhraní a klienty k nim, opomíjí se však tvorba, obsah a umístění dokumentace. Přestože poskytovatelé rozhraní používají jednotné protokoly a technologie, dokumentaci má každý mírně odlišnou. Momentálně se můžeme setkat nejčastěji s dokumentací v HTML, dokumentech kancelářských balíků či PDF dokumentech. Taková dokumentace je tvořena ručně a má řadu nevýhod. Při její tvorbě může být zanesena chyba, dokumentace nemusí být aktuální a nemusí odpovídat popisovanému rozhraní. Dále musí existovat člověk, který bude za dokumentaci zodpovědný. Provádět integraci se systémem mající neodpovídající dokumentaci může prodloužit dobu integrace a tím vzrostou náklady.

Cílem diplomové práce je navrhnout, implementovat a otestovat nástroj pro automatické generování dokumentace vzdálených rozhraní. Nástroj by měl odstranit problémy s tvorbou dokumentace, protože proces její tvorby nebude zahrnovat lidský element a bude zajištěno, aby se dokumentace aktualizovala vždy se změnou rozhraní.

Kapitola 2

Analýza

2.1 Specifikace technologií pro vzdálená rozhraní

Způsobů, jak na platformě Java EE vzdáleně komunikovat mezi systémy přes počítačovou síť, je několik. V současné době se pro integraci nejvíce používají webové služby založené na SOAP zprávách a také služby založené na architektonickém stylu REST. Dalšími způsoby vzdálené komunikace jsou například JMS, RMI nebo CORBA. Tato práce se zabývá prvními dvěmi zmíněnými.

2.1.1 SOAP-based webová služba

SOAP-based webová služba [28] je způsob komunikace mezi počítači. Komunikace, struktura zpráv, popis služeb a registr těchto služeb jsou řízeny otevřenými protokoly, které se neustále vyvýjejí. Díky otevřenosti jsou SOAP-based webové služby platformově nezávislé. Technologie užívané v SOAP-based webových službách v této práci podléhají následujícím verzím:

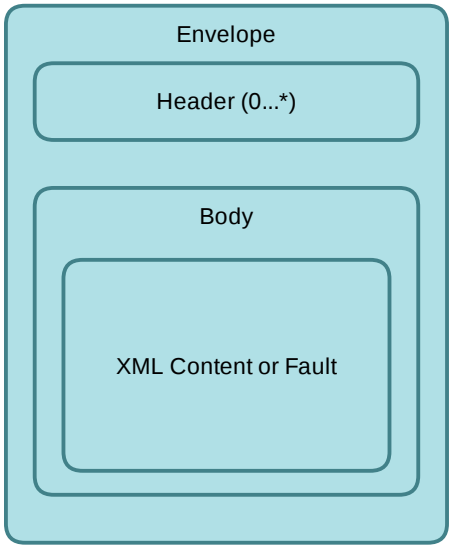
- XML 1.0
- WSDL 1.1
- SOAP 1.2

První verze protokolu SOAP pochází z roku 1999 od společností Microsoft, IBM a dalších. Pro SOAP-based webové služby jsou definovány čtyři hlavní protokoly.

2.1.1.1 Transportní protokol (Transport Protocol)

Přes transportní protokol jsou posílány zprávy po síti. Jako transportní protokol se nejčastěji využívá HTTP, SMTP a FTP. V prostředí jazyka Java lze použít i JMS, což je API pro posílání zpráv. Toto API nemá pevně definován WP, který je zodpovědný za přenos zpráv, ale je možné si zvolit mezi různými protokoly od různých implementací JMS. JMS nařizuje, aby různé implementace dokázaly mezi sebou komunikovat. Praxe však ukazuje, že ne všechny jsou mezi sebou plně kompatibilní.

2.1.1.2 Protokol zpráv (Messaging Protocol)

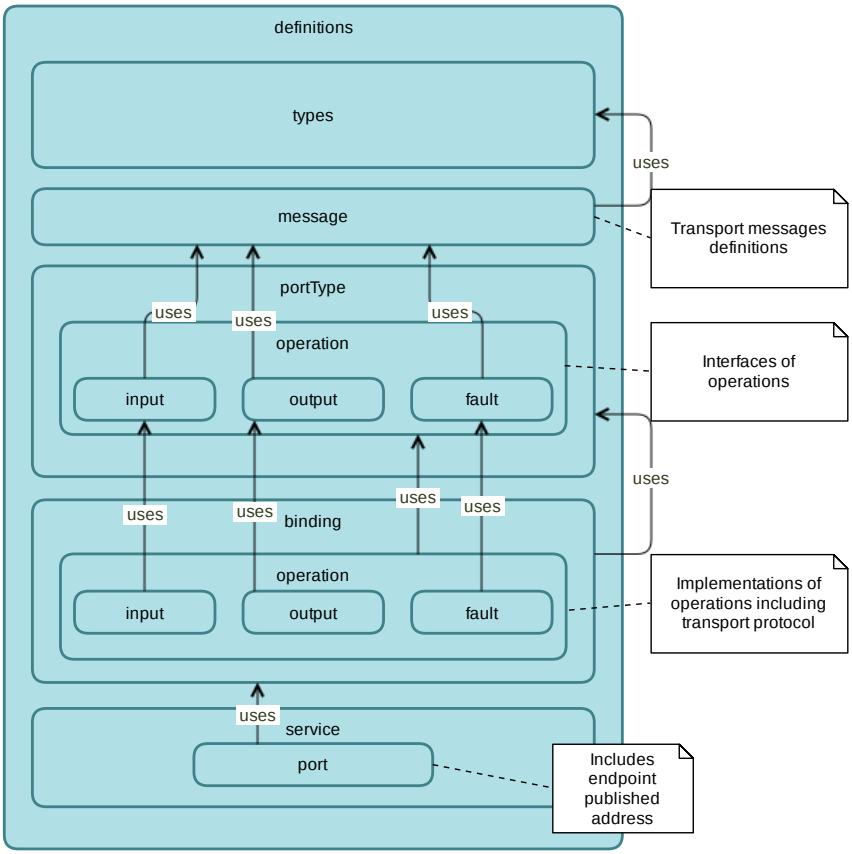


Obrázek 2.1: SOAP zpráva

Protokol zpráv je zodpovědný za zakódování a dekódování zprávy do, resp. z XML-based formátu. Nejčastěji se užívá protokol SOAP [11] [20]. Na 2.1 je vyobrazena struktura SOAP zprávy. SOAP protokol udává, jak bude vypadat přenášená zpráva, která se skládá ze čtyř hlavních částí:

- ***Envelope*** – kořenový element SOAP zprávy, který je povinný a definuje, že jeho obsahem je SOAP zpráva.
- ***Header*** – volitelný element, který se používá pro přidání nové funkcionality a vlastností, jakou může být například autentizace. Ve zprávě je povoleno použít více těchto elementů. Pokud je header ve zprávě přítomen, musí být umístěn jako první přímý potomek elementu envelope.
- ***Body*** – povinný element, který obsahuje přenášená data nazývána „payload“. Body musí být uvnitř elementu envelope a následuje až za elementy header. Sémantika obsahu v elementu body je definována v XML Schema.
- ***Fault*** – tento element je umístěn v elementu body. Ve zprávě se objevuje, pokud je nutné poslat informaci o vzniklé chybě. Fault má definováno, jak má informace o chybě vypadat a obsahuje následující subelementy:
  - *faultCode*,
  - *faultString*,
  - *faultActor*,
  - *detail*.

2.1.1.3 Protokol popisující službu (Description Protocol)



Obrázek 2.2: Struktura WSDL

Tento protokol popisuje veřejné rozhraní dané SOAP-based webové služby. Typicky se používá WSDL, což je XML-based protokol. WSDL definuje, jak ke službě přistupovat a jaké metody můžeme na službě volat. V tomto protokolu jsou dále zaneseny například informace, jaké má metoda parametry, jaké návratové hodnoty a jaké mohou nastat faulty při volání metody.

Kořenový element WSDL dokumentu se jmenuje *definitions*. Schéma elementu je na vidět 2.2. Definitons formuluje název webové služby a deklaruje jmenné prostory, takzvané „namespace“. Uvnitř elementu definitions je několik přímých potomků:

- ***Types*** - element obsahující všechny datové typy, které se mohou přenášet. Datové typy jsou zapsány v XML Schema.
- ***Message*** – značí abstrakci přenášené zprávy. Těchto elementů bývá zpravidla více, protože pro jednu metodu, která má mít vstup, výstup a fault, musí existovat tři tyto elementy. Části elementu message se budou skládat z datových typů definovaných v elementu types.

- ***PortType*** – jedná se o abstraktní množinu operací k jednomu nebo více endpointům[1], které mohou být volány. Těmto operacím se potom v elementu *binding* nastavuje transportní protokol.

  – *Operation* - přímý potomek elementu portType, který lze přirovnat k metodě v klasických programovacích jazycích, jako je například Java či C. V tomto elementu je definováno jméno metody, input message, output message a fault. Části input, output a fault se odkazují na definice v elementech message.

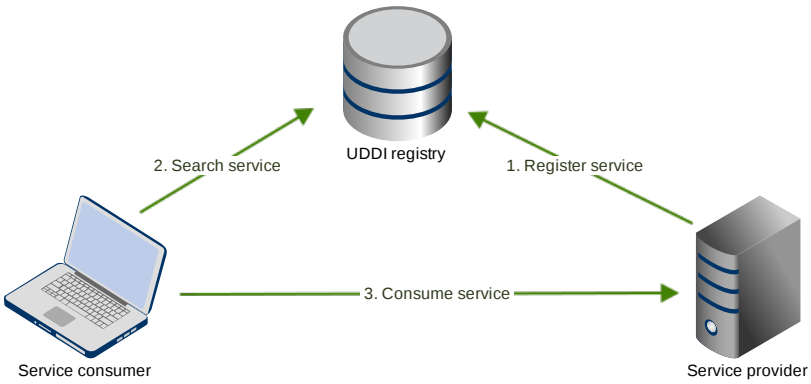
- ***Binding*** – tento element definuje pro každý portType konkrétní datové typy, operace a protokol.

- ***Service*** - poslední přímý potomek kořenového elementu. Uvnitř něj je definován seznam endpointů. Binding služby je zde mapován na *port*.

  – *Port* - přímý potomek elementu service. Port je kombinací bindingu a síťové adresy, na které je služba dostupná.

2.1.1.4 Protokol pro objevování služeb (Discovery Protocol)

Protokol pro objevování služeb je XML-based registr, díky němuž mohou být po celém internetu shromažďovány informace o webových službách. Umožňuje zaregistrování webové služby a jejich procházení v daném registru, který nazýváme UDDI. V tomto registru je možné získat dokument popisující službu (například WSDL) a její metadata.



Obrázek 2.3: Discovery protocol

Informace o určité službě jsou obsaženy napříč třemi komponentami, které UDDI obsahuje - v každé jsou informace jiného charakteru:

- ***Bílé stránky*** - obsahují informace o poskytovateli služby, jako je například jméno firmy, podrobnosti o firmě či kontakt do firmy.

[1] Rozhraní publikované služby

- ***Žluté stránky*** - obsahují zařazení služby či poskytovatele do taxonomie, která může být například geografická. Dalšími typy taxonomie jsou Standard Industrial Classification [16] či United Nations Standard Products and Services Code [19].

- ***Zelené stránky*** - v této komponentě nalezneme technické informace o službě. V zelených stránkách lze zjistit, na jaké adrese je služba dostupná, service binding nebo parametry či odkazy na specifikaci rozhraní. Mohou zde být informace s kontaktními údaji na službu s určitým bindingem. Pokud má totiž služba více bindingů, bude mít i více záznamů v zelených stránkách.

2.1.2 RESTful Web Services

Na rozdíl od SOAP-based webových služeb není pro RESTful webové služby definovaný žádný protokol. RESTful služby vycházejí z architektonického stylu REST, který ve své dizertaci [24] popsal Roy Fielding. RESTful služby jsou orientovány na zdroje „resources“ a jsou aplikací architektury webu na architekturu webových aplikací.

Aplikaci můžeme nazvat RESTful, pokud bude splňovat REST styl, pro který jsou definovány následující podmínky [22]:

- ***Klient-server***

  Aplikace je postavena na modelu, kde je klient oddělen od serveru. Server si nepamatuje stavy, ve kterých se klienti nachází. Klienti a servery mohou být vyvíjeni paralelně, protože mají definované rozhraní mezi sebou. Další výhodou je, že mohou být nahrazeni za jiné klienty nebo servery.

- ***Bezstavová komunikace***

  Veškerý stav, ve kterém se aplikace nachází, si uchovává klient. Bezstavová aplikace se velmi dobře škáluje. Na druhou stranu musí každý dotaz obsahovat veškeré informace o stavu aplikace vůči klientovi. Pokud klient poslal požadavek, který ještě nebyl vyřízen, nachází se klient v tzv. stavu přechodu.

- ***Cache***

  Stejně jako World Wide Web mohou klienti kešovat odpovědi, které musí být definovány jako kešovatelné nebo nekešovatelné. Dobře spravované kešování aplikace pomáhá k lepšímu výkonu, protože se v mnoha případech nemusí posílat požadavek až do aplikace na serveru.

- ***Vrstvený systém***

  Vrstvený systém dovolí mezi server a klienta vložit systém jiný, kterým může být cache nebo loadbalancer, aniž by klient pocítil změnu. Klient nemá k dispozici informaci, se kterým systémem komunikuje.

- ***Jednotné rozhraní***

  Jednotným rozhraním je myšleno použití malé množiny dobře definovaných metod k manipulaci se zdroji. K tomuto rozhraní se vztahují následující omezení:

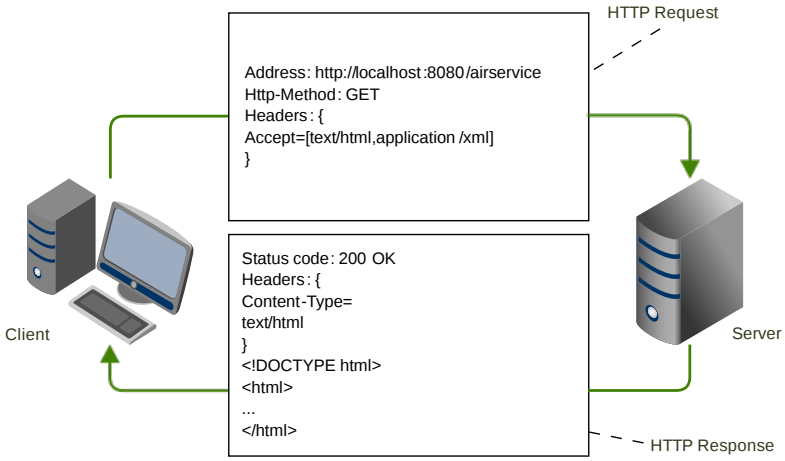
- *HATEOAS (hypermedia as the engine of application state)* – klienti mohou přejít dynamicky do jiného stavu v aplikaci přes hypermedia poslané serverem v předchozí odpovědi. Jinými slovy, odpověď od serveru obsahuje hyperlinky[2], na které může klient přejít, aby se dostal do jiného stavu v aplikaci.
- *Adresovatelné zdroje* – každý zdroj poskytující informace a data musí být jednoznačně identifikovaný svou URI[3].
- *Manipulace se zdrojem a jejich reprezentacemi* – pokud se klient nachází na určitém zdroji, může s ním provádět operace modifikace či smazání.
- *Orientace na reprezentaci* – u jedné URI lze nastavit, aby dokázala použít odlišné formáty dat, protože ne všechny platformy užívají stejný formát. Například webový prohlížeč používá HTML a JavaScript, který potřebuje JSON. Java aplikace může vyžadovat XML.

- ***Kód na vyžádání (volitelné)***

  Server může odeslat klientovi spustitelný kód, který rozšiřuje funkčnost. Takový kód může být ve formě JavaScriptu nebo třeba Java appletu.

2.1.2.1 Svázání RESTu s protokolem HTTP

REST nemá definovaný žádný protokol, avšak ve většině případů, kdy se zmiňujeme o RESTu, máme na mysli REST přes HTTP. Webové aplikace ve webových prohlížečích využívají pouze nepatrnou část vlastností HTTP. Aplikace založené na SOAP-based webových službách používají HTTP protokol striktně jako transportní vrstvu, především kvůli průchodu přes firewally, a využívají tak také jen malou část HTTP.



Obrázek 2.4: Ukázka komunikace přes HTTP

[2] Adresa zdroje, na který je možno přejít
[3] Textový řetězec s definovanou strukturou sloužící pro přesnou specifikaci zdroje informací

HTTP je aktuálně velmi bohatý aplikační protokol poskytující množství zajímavých a užitečných schopností. Na 2.4 je zobrazen princip komunikace přes HTTP. Podmínkou pro psaní RESTful aplikací je velmi dobrá znalost HTTP protokolu. HTTP je protokol s určitými vlastnostmi (synchronní, požadavek/odpověď, aplikační, síťový) používaný pro distribuované spolupracující systémy. Jedná se o primární protokol používaný na webu, který je velmi jednoduchý - klient posílá požadavek složený z názvu HTTP metody, lokaci zdroje, hlavičky a volitelně z těla požadavku, které může obsahovat cokoliv. Nejčastěji se však pro tělo požadavku používá reprezentace dat v JSON, XML, HTML nebo prostém textu „plain text“.

2.2 Rešerše generování vzdáleného API na platformě Java EE

2.2.1 Specifikace JAX-WS 2.0

Specifikace JAX-WS [23] je standardizovaná technologie pro tvorbu SOAP-based webových služeb na platformě Java. JAX-WS API skrývá aplikačnímu vývojáři svou komplexitu. Na serverové straně je třeba definovat interface[4] a v něm metody, které se budou mapovat na operace v endpointu.

Specifikace nenutí, aby byly SOAP zprávy generovány a parsovány programově, protože JAX-WS runtime systém převádí zprávy na Java objekty a obráceně za programátora. V některých situacích se však může stát, že je nutné zprávu generovat či parsovat ručně, což ale JAX-WS umí, a to na různých úrovních abstrakce.

2.2.1.1 Implementace endpointu webové služby

Výchozím bodem pro tvorbu JAX-WS webové služby je třída mající anotaci *@WebService*, která je definována jako endpoint webové služby. SEI je Java rozhraní či třída deklarující metody, které může klient konzumovat na webové službě. Při tvorbě endpointu není povinné mít rozhraní, protože třída implicitně definuje SEI. Pokud přeci jenom chceme oddělit funkcionalitu od rozhraní, je nutné, kromě toho, aby třída implementovala *interface*, nastavit v anotaci *@WebService* v elementu *endpointInterface* celý název rozhraní, tedy i s balíčkem. Název služby (angl. *service name*) lze získat z elementu *serviceName* anotace *@WebService* a pokud je hodnota elementu prázdná, použije se název implementující třídy s příponou „Service“.

Pokud třída implicitně definuje SEI, jako webové operace budou namapovány metody s modifikátorem *public*, které splňují následující:

- Metoda je anotována s *@WebMethod*, jejíž element *exclude* je roven hodnotě *false*. Defaultní hodnota je *false*, tudíž není povinné element specifikovat.
- Metoda není anotována s *@WebMethod* anotací, ale její deklarující třída má anotaci *@WebService*.

Pro mapovací účely u implicitního SEI jsou mapovány stejné anotace jako u implementace webové služby a jejích metod. Pokud nastane situace, že u třídy anotované s *@WebService*

[4] Publikovaná kompenenta v aplikaci definující zacházení s aplikací

chceme metodu s modifikátorem *public* vynechat z mapování na operaci webové služby a máme SEI definované implicitně, musíme přidat k metodě anotaci *@WebMethod* a element *exclude* nastavit na *true*. Pro mapovací účely musí být třída nejvyšší úrovně (angl. *top-level class*) nebo statická vnitřní třída. Třída anotovaná *@WebService* musí mít defaultní veřejný konstruktor.

2.2.1.2 Rozhraní endpointu webové služby

SEI je mapováno na *wsdl:portType* element a jedná se o Java rozhraní, které splňuje následující omezení:

- Musí být anotován s *@WebService*.
- V *@WebService* smí být vyplněn pouze element *targetNamespace*.
- Jakákoliv metoda může být anotována s *@WebMethod*.
- Pokud je u metody použita anotace *@WebMethod*, nesmí obsahovat element *exclude* s hodnotou *true*.
- Všechny parametry a návratové typy musí být kompatibilní s JAXB 2.0 specifikací [30].

WSDL 1.1 nedefinuje standardní formu dědičnosti pro *wsdl:portType* elementy. Jediná povolená dědičnost je dědění Java metod s JAX-WS anotacemi. I když jsou metody definovány v jiném rozhraní než v SEI, je s nimi zacházeno, jako kdyby byly definovány uvnitř SEI.

2.2.1.3 Metoda

Každá veřejná metoda v SEI nemající anotaci *@WebMethod* s elementem *exclude* nastaven na *true*, je mapována na *wsdl:operation* element v korespondujícím *wsdl:portType*.

Jméno operace webové služby je mapováno z elementu *name* v anotaci *@WebMethod* a pokud je element nespecifikován, použije se jméno Java metody. Element *name* se používá k řešení jmenných konfliktů, pokud existuje více potencionálních operací webové služby se stejným názvem.

Webové služby definují dva druhy metod - jednosměrné (angl. *one-way*) a obousměrné (angl. *two-way*). Jednosměrné mají vstup, ale neprodukují žádný výstup. Obousměrné mají vstup a produkují výstup. Defaultně jsou všechny metody mapovány na obousměrné operace. Pro definování jednosměrné operace stačí metodu anotovat s *@OneWay*, ve WSDL mají definován element *wsdl:input* definující vstupní parametry. Pro obousměrné operace je navíc definován element *wsdl:output* za předpokladu, že Java metoda nevrací typ *void*. Jsou u nich definovány *wsdl:fault* elementy v rozsahu nula až vícekrát. Element *wsdl:fault* je mapován z výjimek definovaných v metodě.

Jednosměrná metoda musí splňovat určitá kritéria. Pouze metody s návratovým typem *void*, které nemají parametr implementující *Holder*, a které nevyhazují kontrolované výjimky, mohou, nikoliv musí, být jednosměrné.

Elementy *wsdl:input*, *wsdl:output* a *wsdl:fault* jsou asociovány s elementy *wsdl:message*, které definují přenášené zprávy. Hodnota atributu *name* elementu *wsdl:message* není nijak důležitá a proto se používá hodnota jména operace pro příchozí zprávy a hodnota jména operace zřetězená s „Response“ pro odchozí zprávy. U zpráv typu *fault* se použije pro hodnotu atributu název Java výjimky, pokud není specifikováno jinak v anotaci *@WebFault* u metody.

Pro každý element *wsdl:message* lze definovat *Document* styl nebo RPC styl. V Javě lze styl zpráv nakonfigurovat jak pro celý SEI, tak jednotlivě pro každou metodu. Slouží k tomu anotace *@SOAPBinding* a element *style*.

2.2.1.4 Parametry a návratové typy

Parametry a návratový typ Java metody jsou mapovány buď na zprávy, nebo na deklarace globálních elementů. Pro přizpůsobení mapování parametru na zprávu je určena anotace *@WebParam* a pro stejný účel slouží anotace *@WebResult* u návratového typu. Parametry mohou být mapovány na komponenty pro příchozí zprávy, odchozí zprávy nebo pro oboje. Mapování záleží na klasifikaci parametru, který může být namapován jako SOAP hlavička ve zprávě nastavením elementu *header* na *true* v anotaci *@WebParam*. Element *header* v anotaci *@WebResult* může být použit pro vložení výsledku do SOAP hlavičky.

Parametry a návratové typy jsou rozděleny na tři módy podle toho, ve kterých zprávách se vyskytují.

- ***In*** – hodnota je přenášena ve zprávě z klienta na SEI, ale není přenášena zpět.
- ***Out*** – hodnota je přenášena ve zprávě ze SEI ke klientovi, ale není přenášena zpět.
- ***In/Out*** – hodnota je přenášena z klienta na SEI a zpět ke klientovi.

Návratový typ je vždy mód *out*. U parametrů, jejichž typ je obalen třídou *Holder<T>*, je možno nastavit mód *in/out* nebo *out*. U ostatních parametrů je vždy použit mód *in*. Při obalení parametrů do Holder můžeme docílit funkcionality, kterou Java neumí, a to vracet více hodnot z jedné metody. V anotaci *@WebParam* se elementem *mode* nastavuje mód.

2.2.1.5 Serializace dat

Pro definici serializace Java typů do SOAP zprávy je použita anotace *@SOAPBinding*, kterou je možno použít u třídy nebo metody. Při užití na obou místech je upřednostněna ta u metody. Anotace *@SOAPBinding* má tři elementy s možnými výčty hodnot popsanými v 2.1.

Element	Výčet hodnot
Style	DOCUMENT, RPC
Use	LITERAL, ENCODING
ParameterStyle	WRAPPER, BARE

Tabulka 2.1: Výčet možných hodnot pro SOAP binding

Kombinováním těchto tří elementů lze vytvářet odlišné struktury zpráv se stejnými nosnými daty. Struktura zprávy je určena na základě zvolených hodnot v elementech *@SOAPBinding*.

Style

(a) *DOCUMENT* - obsah SOAP body bude XML dokument, který bude možno validovat vůči předdefinovanému XML Schema dokumentu.

(b) *RPC* - SOAP body bude obsahovat XML reprezentaci volání metody. Jako hlavní element se použije jméno metody a jako potomci parametry metody a návratová hodnota.

Use

(a) *ENCODING* – udává, jak jsou hodnoty zakódovány do XML formátu. SOAP ENCODING je rozšíření SOAP Frameworku a nabízí pravidla, jak konvertovat jakákoliv data ze SOAP datového modelu do XML formátu.

(b) *LITERAL* - definuje, že data jsou serializována podle XML Schema.

ParameterStyle

(a) *WRAPPED* - říká, že zpráva bude zabalena ještě do jednoho elementu.

(b) *BARE* - udává neobalování hodnot do jednoho hlavního předka.

Přístup *DOCUMENT/LITERAL* je jednodušší přístup, protože se jednoduše spoléháme na validaci zprávy vůči XML Schema.

Tvorba XML Schema z Java typů je definována pomocí JAXB. Pomocí tohoto mapování JAX-WS generuje do WSDL datové typy, které jsou užívány v elementech *message*. Jsou podporovány tři styly mapování: *document wrapped*, *document bare* a RPC. Styly se liší v XML Schema konstrukcích.

Document wrapped

Document wrapped je definován v anotaci *@SOAPBinding* s následujícími vlastnostmi:

- ***style*** – DOCUMENT,
- ***use*** – LITERAL,
- ***parameterStyle*** – WRAPPED.

Pro parametry se vytvoří wrapper[5], do kterého budou vygenerované typy umístěny a pro návratový typ se vytvoří druhý wrapper. Jejich vlastnosti lze upravit v anotacích *@RequestWrapper* a *@ResponseWrapper*. Zde přicházejí do popředí módy (*in*, *out*, *in/out*):

[5]Obálka, do které jsou vloženy elementy jako vnitřní prvky

- Parametr s módem *in* je mapován jako potomek globálního elementu request beany[6].
- Parametr s definovaným *out* nebo návratová hodnota jsou mapovány jako potomci response beany[7].
- Parametr s módem *in/out* je mapován jako potomek request beany i response beany.

Document bare

Styl *document bare* je definován v anotaci *@SOAPBinding* s následujícími vlastnostmi:

- ***style*** – DOCUMENT,
- ***use*** – LITERAL,
- ***parameterStyle*** – BARE.

Při aplikaci *document bare* jsou na metodu kladeny tyto požadavky:

- Metoda musí mít maximálně jeden parametr *in* nebo *in/out*, který není mapován na hlavičku.
- Pokud má návratový typ jiný než *void*, nesmí mít *in/out* a *out* parametry, které nejsou mapovány na hlavičky.
- Pokud má návratový typ *void*, může mít maximálně jeden parametr *in/out* nebo *out*, který není mapován na hlavičku.

Globální deklarace elementu je vygenerována pro vstupní typ metody a analogicky pro výstupní typ metody.

Styl RPC

Styl RPC je definován v anotaci *@SOAPBinding* s následujícími vlastnostmi:

- ***style*** – RPC,
- ***use*** – LITERAL,
- ***parameterStyle*** – WRAPPED.

Java typy pro parametry *in*, *in/out*, *out* a návratové typy jsou mapovány na XML Schema typy s použitím JAXB. Pro každý parametr metody a návratový typ je vytvořen ve WSDL dokumentu element *part*, který je použit v elementu *message*. Při použití RPC stylu nesmí být posílány *null* hodnoty.

[6] Objekt vytvořený pomocí JAXB ze vstupu metody
[7] Objekt vytvořený pomocí JAXB z výstupu metody

2.2.2 Specifikace JAX-RS 2.0

Specifikace JAX-RS [29] byla vytvořena za účelem zjednodušení vývoje RESTful služeb. Použitím JAX-RS je webový zdroj (angl. *resource*) implementován jako resource třída[8] a požadavky jsou obsluhovány resource metodami[9].

2.2.2.1 Resource třída

Resource třída je Java třída používající JAX-RS anotace k implementaci konkrétního webového zdroje. Musí splňovat požadavek na POJO třídu a zároveň musí mít alespoň jednu metodu s anotací *@Path* nebo s anotací obsahující request method designator[10]. Pokud není explicitně změněno, nová instance resource třídy je vytvořena pokaždé, když přijde požadavek na webový zdroj, jenž třída implementuje. Implementace JAX-RS může kromě životního cyklu *per-request* poskytovat jeho další volby. Speciální resource třídou je kořenová resource třída[11]. Aby se resource třída mohla stát kořenovou, musí obsahovat anotaci *@Path* s URI, na které bude tato root resource třída zaregistrována a kde bude vyřizovat požadavky. [29]

2.2.2.2 Konstruktor

Root resource třída je instanciovaná JAX-RS runtimem a musí mít konstruktor s modifikátorem *public*, který může být bezparametrický nebo parametrický. Může obsahovat takové parametry, aby je JAX-RS runtime byl schopen předat konstruktoru. Jako parametry může konstruktor obsahovat ty, které jsou anotovány *@Context*, *@HeaderParam*, *@CookieParam*, *@MatrixParam*, *@QueryParam* a *@PathParam*. Pokud je obsaženo více veřejných konstruktorů, použije se ten, který obsahuje nejvíce parametrů. Při shodě počtu parametrů dvou konstruktorů není pořadí jednoznačně určeno a implementace JAX-RS si mohou zvolit, který z nich použijí, zároveň by však měly generovat varování o této víceznačnosti. Resource třída, která není kořenová, je instanciovaná aplikačním kódem a nepotřebuje výše popisovaný veřejný konstruktor.

2.2.2.3 Atributy

Při vytváření instance resource třídy lze naplnit její atributy konstruktorem nebo setter metodami. Pokud má atribut modifikátor *public*, lze přidat anotace přímo k atributu. Anotace jsou stejné u všech tří zmíněných přístupů plnění hodnot - *@Context*, *@HeaderParam*, *@CookieParam*, *@MatrixParam*, *@QueryParam* a *@PathParam*. Protože injektování probíhá při tvorbě objektu, je použití těchto anotací (s výjimkou *@Context*) v resource třídě u konstruktoru a při nastavování atributů podporováno pouze u *per-request* životního cyklu. Objekty vrácené pomocí sub-resource lokátoru[12] se inicializují programově přímo v kódu, nejsou tedy ve správě JAX-RS runtime.

[8] Java třída využívající JAX-RS anotace pomocí, kterých implementuje korespondující webový zdroj

[9] Java metoda resource třídy anotovaná JAX-RS anotacemi, která obsluhuje HTTP požadavky

[10] Runtime anotace anotovaná s *@HttpMethod*. Používá se pro identifikaci jaký typ HTTP metody může resource metoda obsloužit.

[11] Je to resource třída anotovaná *@Path*. Kořenová resource třída je výchozím bodem pro adresy subzdrojů.

[12] Metoda resource třídy používaná k lokaci subzdrojů korespondujícího zdroje.

Typy, pro které mohou být použity anotace *@HeaderParam*, *@CookieParam*, *@MatrixParam*, *@QueryParam* a *@PathParam*, musí splňovat jednu z následujících podmínek:

1. Typy, pro které je *ParamConverter* definován pomocí registrace *ParamConverterProvider*.
2. Primitivní typy.
3. Typy mající konstruktor, který obsahuje jeden parametr typu *String*.
4. Typy mající statickou metodu pojmenovanou *valueOf* nebo *fromString* s jedním parametrem typu *String* a návratovou hodnotou vracející instanci daného typu. Pokud typ není *enum*, musí být při výskytu obou metod současně upřednostněna *valeuOf*. V případě *enum* má přednost *fromString* metoda.
5. *List<T>*, *Set<T>* nebo *SortedSet<T>*, kde T vyhovuje bodu 3 nebo 4.

Anotace *@DefaultValue* může být použita pro případ, kdy atribut nedostane žádnou hodnotu. Anotace přijímá hodnotu jako *String* a ten je poté převáděn do požadované podoby pomocí kroků 1 - 5 výše.

2.2.2.4 Resource metoda

Resource metoda je metoda obsluhující požadavek, který přišel na resource třídu. Tato metoda musí mít anotaci request method designatoru. Taková anotace má anotaci *@HttpMethod* a JAX-RS definuje množinu request method designátorů pro obecně definované HTTP metody. Jejich anotace jsou následující *@GET*, *@POST*, *@PUT*, *@DELETE*, *@HEAD*, *@OPTIONS*. Lze si zadefinovat designator pro vlastní HTTP metodu tím, že se vytvoří anotace mající anotaci *@HttpMethod*. Pouze metody, které mají modifikátor přístupu *public*, mohou být považovány za resource metody.

Po invokaci resource metody jsou parametry anotované *@Context*, *@HeaderParam*, *@CookieParam*, *@MatrixParam*, *@QueryParam*, *@PathParam* a *@FormParam* mapovány z požadavku podle sémantiky anotace. Podobně jako u atributů je zde možno použít anotaci *@DefaultValue*. Parametr nemající výše uvedenou anotaci je označován jako entity parametr[13] a je mapován z těla požadavku. Za převody mezi tělem požadavku a Java typem jsou zodpovědní entity provideři2.2.2.6. Resource metoda může mít maximálně jeden entity parametr.

Resource metoda může vracet návratový typ *void*, *Response*, *GenericEntity* nebo jiný Java typ. Tyto typy jsou mapovány do těla odpovědi následovně:

- ***Void*** – prázdné tělo odpovědi se status kódem 204.
- ***Response*** – tělo odpovědi je naplněno z entity property objektu typu *Response* se status kódem specifikovaným ve status property objektu typu *Response*. Pokud je návratová hodnota *null*, status kód bude 204. Při *non-null* entity property v *Response* a nespecifikovaném statusu je status nastaven na 200. Při *null* entity property je status nastaven na 204.

[13] Parametr metody, který není anotován request method designatorem. Je mapován z těla požadavku.

Návratový typ	Návratová instance	Raw typ	Generický typ
GenericEntity	GenericEntity nebo podtřída	RawType property	Type property
Response	GenericEntity nebo podtřída	RawType property	Type property
Response	Object nebo podtřída	Třída instance	Třída instance
Jiný	Návratový typ nebo podtřída	Třída instance	Generický typ vráceného typu

Tabulka 2.2: Určení generických a raw typů návratových hodnot

- ***GenericEntity*** – tělo odpovědi je naplněno z entity property objektu typu *GenericEntity*. Při *null* těle odpovědi je status 204, při *non-null* 200.
- ***Ostatní*** – tělo odpovědi je přímo mapováno z instance, která se vrací. Pokud je vrácena instance anonymní vnitřní třídy, je použit její předek. Při *null* je status 204, jinak 200.

Metody, které potřebují posílat více metadat než jen tělo odpovědi a status kód, by měly vracet instanci třídy *Response*.

Sub-resource metody a sub-resource lokátor metody mohou vyhazovat očekávané nebo neočekávané výjimky. Tyto výjimky musí být runtimem namapovány na HTTP odpověď.

Očekávanou výjimku lze přebalit do instance *WebApplicationException* a jejích podtříd nebo je možné implementovat pro výjimku exception mapper[14], kterému bude jako parametr předána výjimka a na základě ní bude vytvořena požadovaná odpověď.

Root resource třída je ukotvena v URI prostoru použitím anotace *@Path*. Hodnota v anotaci je relativní URI šablona cesty, jejíž základní URI je postavena z deployment kontextu a cesty aplikace. Šablona URI cesty je textový řetězec se žádným nebo více vloženými parametry. Po dosazení konkrétních hodnot za parametry se ze šablony stane validní URI cesta. Parametr v šabloně cesty je umístěn ve složených závorkách {}. Parametr v URI šabloně může specifikovat i regulární výraz. Například *@Path(„test/id:.+“)* říká, že resource třída či metoda bude vyhovovat, pokud URI bude začínat na test a poté bude následovat alespoň jeden segment cesty. V tomto případě lze parametr *id* nainjektovat jako atribut třídy nebo parametr metody pomocí anotace *@PathParam*.

Metody v resource třídě mající anotaci *@Path* mohou být buď sub-resource metodami nebo sub-resource lokátory. Sub-resource metody odchytí a zpracují HTTP požadavek přímo uvnitř metody, zatímco sub-resource lokátor metoda vytvoří a vrátí instanci, která obslouží HTTP požadavek. Zda-li se jedná o první nebo druhý typ, je řízeno přítomností request method designatoru. Pokud je designator přítomný, jedná se o sub-resource metodu a ta přímo obslouží požadavek a vrátí například instanci *Response*. Při absenci designatoru se dynamicky rozhodne o objektu, na který bude delegováno obsloužení požadavku. Při vyřizování požadavku je objekt považován za resource třídu a může mít stejné anotace jako resource třída. Sub-resource lokátor metoda může mít stejné parametry jako normální resource metoda, ale nesmí obsahovat entity body parametr.

U resource metod a resource tříd lze definovat formát akceptovatelný pro požadavek i formát, jenž bude navrácen v odpovědi. Pokud je nadefinován formát u třídy i metody,

[14] Java třída provádění mapování výjimky na HTTP odpověď.

má přednost formát u metody. Při nedefinovaném formátu u metody se akceptují ty, které jsou definovány u třídy. Pokud není formát definovaný ani ve třídě, použije se formát „*/*“. V HTTP protokolu se pro práci s formáty používají dvě hlavičky, jsou jimi *Content-Type* a *Accept*. *Content-Type* obsahuje formát přenášeného obsahu s touto hlavičkou a *Accept* říká, jaký formát chce posílající přijmout. V JAX-RS těmto hlavičkám odpovídají anotace *@Consumes* a *@Produces*. *@Consumes* odpovídá hlavičce *Accept* a *@Produces* hlavičce *Content-Type*.

2.2.2.5 Dědičnost resource tříd

JAX-RS anotace mohou být užity nejen přímo v resource třídě, ale také v nadtřídách a rozhraních, jež implementují. Proto je nutné specifikovat pořadí, v jakém budou mít anotace přednost. Největší přednost mají anotace přímo v resource třídě. Pokud u resource třídy nebo u resource metody nějaké jsou, jsou ostatní třídy a rozhraní ignorovány. Dalším pravidlem je, že má vždy přednost nadtřída před rozhraním a nezáleží na tom, jak je třída vysoko. V řetězu tříd dědičnosti seřazeném od nejobecnější po nejkonkrétnější třídu má vždy větší prioritu ta, která je konkrétnější. Jelikož jedna třída může mít maximálně jednoho přímého předka, nemůže zde nastat kolize. Pokud by každá třída implementovala pouze jeden interface, kolize by také nemohla nastat a přednost by měly anotace z interfacu, který je nejblíže třídě. Pokud je na jedné úrovni více rozhraní, není jejich přednost specifikována a každý implementátor JAX-RS si může zvolit vlastní pořadí.

2.2.2.6 Entity provideři

V JAX-RS existují entity provideři, kteří se starají o mapování přenášených dat na Java objekty a obráceně. Jedním typem entity providerů je *MessageBodyReader* a druhým je *MessageBodyWriter*. Rozhraní *MessageBodyReader* definuje kontrakt mezi JAX-RS runtime a komponentami mapující data na Java objekty, rozhraní *MessageBodyWriter* definuje obrácený směr mapování.

MessageBodyReader

Java třídy poskytující tuto funkcionalitu musí implementovat rozhraní *MessageBodyReader* a mohou být anotovány s *@Provider* pro automatické zaregistrování.

Následující body popisují kroky provedené JAX-RS implementací při mapování entity body na parametr Java metody:

1. Získání media typu požadavku. Pokud požadavek neobsahuje HTTP hlavičku *Content-Type*, pak je použit *application/octet-stream*.
2. Identifikace Java typu parametru metody.
3. Provedení výběru množiny možných *MessageBodyReader* providerů, kteří podporují media typ požadavku.
4. Iterování přes vybrané *MessageBodyReader* třídy a zavolání metody *isReadable*. Provede se výběr nejvhodnějšího *MessageBodyReader* providera, který podporuje již zmíněný Java typ.

5. Pokud předchozí krok vybral vhodného *MessageBodyReader* providera, pak je použita jeho metoda *readFrom* pro namapování entity body na požadovaný Java typ.

6. V opačném případě musí server runtime vygenerovat výjimku *NotSupportedException* se statusem 415.

MessageBodyWriter

Zaregistrování *MessageBodyWriter* providera je stejné jako pro *MessageBodyReader* (viz kapitola 2.2.2.6).

Následující body popisují kroky provedené JAX-RS implementací při mapování návratového hodnoty na entity body:

1. Získání objektu, který bude mapován na entity body. Pro návratový typ *Response* a jeho podtříd je objekt získán z property entity.

2. Dojde k určení media typu pro odpověď.

3. Vybere se množina *MessageBodyWriter* providerů, kteří podporují typ vráceného objektu a media typ.

4. Seřadí se vybraní *MessageBodyWriter* prideři primárně podle podporovaného generického typu od toho, který je nejblíže vrácenému objektu, a sekundárně podle podporovaného media typu.

5. Iterace přes všechny seřazené *MessageBodyWriter* providery a zavolání metody *isWritable*. Následně dojde k vybrání nejvhodnějšího *MessageBodyWriter*.

6. Pokud v předchozím kroku došlo k vybrání vhodného kandidáta, použije se metoda *writeTo* k namapování objektu na entity body.

7. V opačném případě musí server runtime vygenerovat *InternalServerErrorException*, což je podtřída *WebApplicationException* se statusem 500.

2.2.3 Apache Maven

Velká většina Maven [21] uživatelů označuje Maven za sestrojovací nástroj, což není zcela správně, protože Maven umí více než jen tvorbu artefaktů ze zdrojových kódů. Typickým představitelem kategorie sestavovacích nástrojů je Apache Ant [1]. Při srovnání těchto dvou nástrojů je patrné, jaké funkcionality má Maven oproti klasickému sestavovacímu nástroji navíc. Klasický sestavovací nástroj provádí preprocessing, kompilaci, zabalení, testování a distribuci. Maven může navíc tvořit reporty, generovat webové stránky a usnadňuje komunikaci mezi ostatními členy týmu. Správné označení pro Maven je tedy project management tool.

2.2.3.1 Konvence a jednotné rozhraní

Nástroj Maven jako první zavedl konvence a jednotné rozhraní. Díky němu se vyvarujeme tomu, aby člověk, který je u projektu nový, strávil hodiny studováním sestavení. Maven defaultně definuje, ve kterých adresářích má být uložen zdrojový kód (*$basedir/src/main/java*), ostatní zdroje (*$basedir/src/main/resources*) nebo zdrojové kódy pro testy (*$basedir/src/test*). Dále jsou také definovány defaultní umístění pro zkompilované třídy (*$basedir/target/classes*) a zabalený artefakt (*$basedir/target*). S použitím Mavenu se již nemusí ručně stahovat všechny závislosti pro knihovnu, kterou jsme se rozhodli použít, jelikož jsou tyto závislosti automaticky získány z repozitáře. Maven definuje také životní cyklus sestavení, který se skládá z po sobě jdoucích fází. Pokud se člověk dostane k projektu používající Maven, nejenže se hned orientuje v adresářové struktuře, ale pro spuštění celého sestavení stačí zadat příkaz *mvn install*.

Listing 2.1: Defaultní adresářová struktura Maven projektu

```
1  src/
2    main/
3      java/
4      resources/
5    test/
6      java/
7      resources/
8  target/
9    classes/
10   deployable artifact (JAR, EAR, WAR, ...)
```

Zde je popsána struktura z 2.1. Soubor *pom.xml* je Maven konfigurační soubor daného projektu. Adresář *java* obsahuje zdrojové kódy, adresář *resources* pak ostatní zdroje mimo zdrojové kódy (konfigurační soubory, descriptoty, obrázky atd...). Adresář *classes* obsahuje zkompilovaný kód. Deployable artifact může být výsledný soubor jar, war či ear.

2.2.3.2 Project Object Model (POM)

Apache Maven udržuje model projektu, nestará se pouze o kompilování kódu do bajtkódu. Udržuje název projektu, licenci pod kterou je software vyvíjen, kdo stojí za vývojem projektu, kdo přispívá do projektu a na kterých projektech je tento projekt závislý. Maven drží informaci, jakým SCM je projekt verzován a na jaké adrese lze nalézt repozitář. Všechny tyto informace jsou konfigurovány v souboru pom.xml, který je povinný pro každý projekt. Každý projekt v Mavenu je jednoznačně identifikovatelný, protože má Maven coordinates, které se skládají z *groupId*, *artifactId* a *version*. *GroupId* uvozuje skupinu, pod kterou projekt patří (například *org.apache.maven*), *artifactId* blíže identifikuje projekt v rámci *groupId* (například *maven-plugin-api*) a *version* udává verzi projektu (například *1.0.0*).

2.2.3.3 Maven repozitář

Při prvním spuštění Mavenu začne Maven stahovat závislosti z místa nazývaného Maven repozitář. Repozitář je zdroj v síti, kde jsou uchovávány artefakty (jar, war, ear, maven pluginy atd...), které jsou uloženy v adresářové struktuře podle Maven coordinates. Centrální

repozitář se nachází na *http://repo1.maven.org/maven2*. Knihovny, které potřebuje projekt, se nacházejí právě v takových repozitářích. Potřebnou knihovnu není třeba stahovat ručně, ale pouze se v pom.xml do elementu *dependencies* přidají coordinates požadované knihovny, která se při spuštění sestavení automaticky získá.

Aby se při každém sestavení nemuseli všechny závislosti opakovaně stahovat, na lokální stanici se vytváří lokální repozitář na cestě */home/USERNAME/.m2/repository*, kde se ukládají artefakty. Maven vždy hledá artefakt v lokálním repozitáři a až poté prohledává ty vzdálené. Při spuštění *mvn install* se v poslední fázi sestavení nainstaluje nově vytvořený artefakt do lokálního repozitáře pro použití v dalších projektech. Pokud projekt obsahuje závislost mající další závislosti, jsou automaticky získány všechny. Strom závislostí lze zobrazit spuštěním *mvn dependency:tree*, což lze vysvětlit jako spuštění pluginu *dependency* s goalem *tree*.

2.2.3.4 Životní cyklus

Životní cyklus sestavení je seřazená sekvence fází (angl. *phases*) zahrnutá v sestavování projektu. Maven může podporovat různé životní cykly, ale defaultní životní cyklus začíná validací základní integrity projektu a končí fází deployování do produkčního prostředí.

Seznam fází defautního životního cyklu:

- ***validate*** – ověření, že projekt nemá žádné chyby a všechny potřebné informace jsou v něm dostupné,
- ***compile*** – kompilování zdrojových kódů,
- ***test*** – testování zkompilovaného kódu s použitím testovacího frameworku. Tyto testy nevyžadují, aby byl kód zabalen nebo deployován,
- ***package*** – vezme zkompilovaný kód a zabalí ho do distribuovatelného formátu, například jako jar,
- ***integration-test*** – distribuce balíčku na prostředí, kde mohou proběhnout integrační testy, provedení testů,
- ***verify*** – kontroly k ověření validnosti balíčku a splnění kvality,
- ***install*** – instalace balíčku do lokálního repozitáře k použití v jiných projektech,
- ***deploy*** – v integračním nebo release prostředí zkopíruje balíček do vzdáleného repozitáře pro sdílení s ostatními projekty a vývojáři.

Seznam fází není kompletní a může se rozrůst. Pokud chceme projekt dostat do určité fáze (nemusí se nutně jednat o poslední fázi), stačí zadat jméno poslední fáze, která se má spustit a Maven automaticky začne spouštět fáze od první až po explicitně vyvolanou. Co se děje v jednotlivých fázích, je popsáno v kapitole 2.2.3.5.

2.2.3.5 Pluginy a goaly

Goal je specifická úloha obsažená v Maven pluginu, kterou je možné spustit. Vhodným příkladem může být jednoduchý *maven-jar-plugin*, který má *pluginId* jar a goal jar. Plugin může být spuštěn příkazem *mvn pluginId:goal* nebo může být připojen k určité fázi životního cyklu. Vyvoláním fáze se spustí plugin s určitým goalem. Celý životní cyklus sestavení je v podstatě postupné spouštění různých goalů z různých pluginů. Goaly je možno konfigurovat v pom.xml.

Listing 2.2: Ukázka konfigurace v pluginu

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
    <debug>true</debug>
  </configuration>
</plugin>
```

Konfiguraci goalu lze umístit na různá místa v konfiguračních pom.xml souborech, čímž je myšleno, do kterého XML elementu se má umístit element *configuration*. Element *configuration* uvnitř sebe obsahuje další elementy, které reprezentují danou konfiguraci. Při globální konfiguraci pluginu je konfigurace umístěna do elementu *plugin*. Tato konfigurace bude společná pro všechny goaly uvnitř pluginu. Další možností je umístit konfiguraci pouze ke konkrétnímu goalu, což se provede vložením konfigurace do elementu *execution*, který obsahuje informaci, ke kterému goalu se vztahuje. Pokud plugin obsahuje konfiguraci na obou místech, ke konfiguraci goalu, která má přednost, se aplikuje konfigurace pluginu. Aplikovány budou pouze ty konfigurační elementy pluginu, které nepřepisují konfigurační elementy goalu. Situace se trochu zkomplikuje při multimodulových projektech, kde je plugin nakonfigurován v rodičovském modulu i v potomkovi. Jelikož Maven tvoří efektivní pom, přidá konfiguraci pluginu, resp. goalu z rodiče do konfigurace pluginu, resp. goalu potomka. Seřazená konfigurace od nejobecnější k nejkonkrétnější vypadá takto: konfigurace pluginu v rodiči, konfigurace pluginu v potomkovi, konfigurace goalu v rodiči a konfigurace goalu v potomkovi.

2.2.3.6 Multimodulový projekt

Multimodulový projekt je definován rodičovským pomem, uvnitř kterého jsou definovány jeho submoduly. Rodičovský pom bývá též nazývaný top-level pom.

Listing 2.3: Ukázka konfigurace multimodulového projektu

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.jrapidoc</groupId>
    <artifactId>jrapidoc-pom</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>pom</packaging>

    <modules>
        <module>jrapidoc-rest-plugin</module>
        <module>jrapidoc-soap-plugin</module>
        <module>jrapidoc-annotation</module>
        <module>jrapidoc-gui</module>
        <module>jrapidoc-model</module>
        <module>jrapidoc-logger</module>
        <module>jrapidoc-plugin-base</module>
    </modules>

    <build>
        <pluginManagement>
            <plugins...>
        </pluginManagement>
    </build>

    <dependencyManagement>
        <dependencies...>
    </dependencyManagement>
</project>
```

Rodič definuje Maven coordinates (*groupId*, *artifactId* a *version*). Top-level projekt nevytváří artefakty JAR, WAR a podobné, o to se starají submoduly. Top-level projekt definuje submoduly a provádí globální konfiguraci, která je společná submodulům. Například mohou být definovány společné závislosti pomocí coordinates knihoven, poté pracují submoduly se stejnou verzí knihovny. Pokud je provedeno sestavení v top-level projektu, všechny submoduly budou také sestaveny.

2.2.3.7 Vlastní plugin

Maven plugin se skládá z plugin descriptoru a jednoho nebo více MOJO. Maven běží uvnitř IoC kontejneru Plexus [12], který se stará o injektování komponent, které jsou definovány v MOJO třídě. MOJO třída je třída implementující rozhraní *Mojo*, který obsahuje výchozí metodu *execute*, jež je výchozím bodem pro každý plugin. Komponenty mohou být v MOJO třídě buď Java primitivní typy, soubory, pole, MOJO třídy nebo reference na jiný projekt, než ve kterém je plugin spuštěn.

Plugin descriptor, který je povinný v každém modulu, obsahuje konfiguraci pluginu. Je

umístěn v *META-INF/maven/plugin.xml* Tento descriptor není nutné psát ručně, ale lze jej generovat pluginem *maven-plugin-plugin* a jeho goalem *descriptor*. Descriptor je generován na základě introspekce kódu.

Listing 2.4: Ukázka plugin descriptoru

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
  <name>jrapidoc-rest-plugin</name>
  <description></description>
  <groupId>org.jrapidoc</groupId>
  <artifactId>jrapidoc-rest-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <goalPrefix>jrapidoc-rest</goalPrefix>
  <isolatedRealm>false</isolatedRealm>
  <inheritedByDefault>true</inheritedByDefault>
  <mojos>
    <mojo>
      <goal>run</goal>
      <description>Created by papa on 14.3.15.</description>
      <requiresDirectInvocation>false</requiresDirectInvocation>
      <requiresProject>true</requiresProject>
      <requiresReports>false</requiresReports>
      <aggregator>false</aggregator>
      <requiresOnline>false</requiresOnline>
      <inheritedByDefault>true</inheritedByDefault>
      <phase>process-classes</phase>
      <implementation>org.jrapidoc.plugin.RestMojo</implementation>
      <language>java</language>
      <instantiationStrategy>per-lookup</instantiationStrategy>
      <executionStrategy>once-per-session</executionStrategy>
      <threadSafe>false</threadSafe>
      <parameters>
        <parameter>
          <name>baseUrl</name>
          <type>java.lang.String</type>
          <required>false</required>
          <editable>true</editable>
          <description></description>
        </parameter>
        <parameter>
          <name>custom</name>
          <type>java.util.Map</type>
          <required>false</required>
          <editable>true</editable>
          <description></description>
        </parameter>
    ...dalsi parametry
      </parameters>
      <configuration>
        <basedir implementation="java.io.File"
        default-value="${project.basedir}"/>
        <target implementation="java.io.File"
        default-value="${project.build.directory}"/>
      </configuration>
    </mojo>
  </mojos>
  <dependencies...>
</plugin>
```

Postup při tvorbě pluginu spočívá v založení Maven projektu s *packaging* nastaveným na *maven-plugin*. Dalším krokem je nakonfigurování potřebných pluginů a závislostí v pom.xml. Po splnění těchno prerekvizit je možné tvořit MOJO třídy.

Anotace v MOJO

Ke třídě MOJO se váže nejdůležitější anotace *@Mojo*, některé její elementy zde budou vysvětleny. Element *name* udává jméno goalu, který bude reprezentován MOJO třídou. Element *defaultPhase* specifikuje defaultní fázi, ke které bude goal napojen, pokud nebude explicitně definováno v konfiguraci jinak. Element *requiresDependencyResolution* definuje, jaké závislosti bude plugin mít na classpath při spuštění. Možné volby jsou: *NONE*, *COMPILE*, *COMPILE_PLUS_RUNTIME*, *RUNTIME*, *RUNTIME_PLUS_SYSTEM* a *TEST*.

Parametry v MOJO

Další důležitou částí vedle metody execute() a anotací jsou v MOJO třídě parametry. Parametry se do MOJO dostanou z konfigurace pluginu v pom.xml pomocí IoC kontejneru Plexus.

Listing 2.5: Definice parametru v Mojo

```
1 @Parameter(defaultValue = "${project.basedir}", readonly = true)
2 File basedir;
```

V ukázce 2.5 je vidět, jak se definuje instanciování objektu typu *File* reprezentujícího kořenový adresář projektu. Hodnota elementu *defaultValue* obsahuje ve složených závorkách jméno maven property, ve které je uložena cesta k tomuto adresáři. Parametry mohou být i vícehodnotové, což je užitečné v případě, kdy je třeba do konfigurace přidat parametr typu pole nebo *List* a podobné. Co je pro to nutné udělat, je vidět na ukázkách 2.6 a na 2.7.

Listing 2.6: Definice vícehodnotového parametru v Mojo

```
1 @Parameter(name = "includes")
2 List<String> includes;
```

Listing 2.7: Konfigurace vícehodnotového parametru v pom.xml

```
1 <includes>
2     <include>airservice.resources</include>
3     <include>airservice.otherresources</include>
4 </includes>
```

V definici parametru pro MOJO je podporován i typ *Map<String, String>*. Konfigurace takového parametru vypadá podobně jako konfigurace pro pole, ale místo názvů elementů include z ukázky 2.7 se použijí klíče pro mapu.

2.3 Rešerše existujících generátorů dokumentace

Cílem rešerše existujících nástrojů je prozkoumat ty nástroje, které umí dokumentovat RESTful rozhraní, SOAP-based webové služby. Analyzovány budou takové nástroje, které umí provádět generování ze specifikací JAX-WS a JAX-RS. Jelikož popis některých částí rozhraní není možné staticky vyčíst z analýzy kódu, je nutné zjistit, jak jsou v jednotlivých nástrojí tyto situace řešeny.

Na základě pozitiv a negativ, které budou u jednotlivých nástrojů zjištěny, bude stanoveno, jaké funkce bude mít nově vznikající nástroj.

2.3.1 Swagger

Swagger [18] je nástroj pro tvorbu API dokumentace a testovacího klienta a pracuje pouze s RESTful službami. V současné době je zřejmě nejoblíbenějším nástrojem pro API dokumentaci. Do zdrojového kódu přidává vlastní anotace, které pomáhají vytvořit kvalitnější API dokumentaci. Swagger při generování artefaktů vytváří soubory podobným principem, jako je vytvořeno WSDL u SOAP-based webových služeb. Vygenerované soubory jsou ve formátu JSON. Pro každou resource třídu je jeden soubor, na který odkazuje hlavní soubor service.json, jež slouží jako výchozí bod se seznamem zdrojů. Vygenerované artefakty tvoří model rozhraní[15], který je načítán klientem, jež umožňuje zobrazit API dokumentaci. Klient je grafické rozhraní v HTML, CSS a JavaScriptu. Pokud chceme provolat metody, musíme vygenerovaný model načíst v Swagger UI. Swagger UI je grafické rozhraní podobné klientovi.

Swagger bude pro svou oblibu analyzován mnohem podrobněji než ostatní nástroje.

Výhody

- Pod Apache License, Version 2.0 [2].
- Dokáže provolat metody z API.
- Tvorba modelu rozhraní z různých jazyků (Java, Scala, .NET, Go, JavaScript, PHP, Python, ...).
- Tvorba klienta v různých jazycích (Java, Scala, .NET, Go, JavaScript, PHP, Python, ...).
- Integrace s Apache Maven.

Nevýhody

- Podpora pouze RESTful služeb.
- Složitá konfigurace.
- Nepodporuje dostatečně JAX-RS.

[15] Datová struktura reprezentující popis vzdáleného rozhraní

- Nepodporuje dědičnost u resource tříd. Pokud dědičnost existuje, skončí chybou a nezobrazí v GUI u zdroje žádné metody.
- Nepodporuje sub-resource lokátor. Pokud sub-resource lokátor existuje, skončí chybou a nezobrazí v GUI u resource žádné metody.
- V resource třídě u atributů nepodporuje typy anotované *@MatrixParam*, *@CookieParam*, *@FormParam*.
- Testovací klient nepodporuje jako parametr resource metody typ *@MatrixParam*, *QueryParam*, *CookieParam*.
- Přestože dokumentace uvádí, že *Content-Type* je zobrazen v GUI, není to pravda.
- Pokud je návratový typ *Map<K, V>*, nelze tuto skutečnost zdokumentovat. Stejně tak u entity body parametru.
- Pokud se jedná o asynchronní metodu, ignoruje to.
- Nepodporuje více anotací pro typ HTTP metody (*@GET*, *@POST*, ...) u jedné Java metody.

2.3.2 MireDot

Miredot [10] je další z nástrojů generující API dokumentaci. Dokumentaci generuje pouze k RESTful službám a u dokumentace není možnost provést testovací volání. MireDot má dva druhy licencování. Pro používání free verze je nutné zaregistrovat projekt, který bude využívat MireDot. V konfiguraci maven projektu je zapotřebí vložit licenční kód, který se vztahuje ke konkrétnímu modulu s *groupId* a *artifactId*. Free verzi lze uplatnit jen na projekty spadající do kategorie open-source projektů, které nejsou výdělečné. Komerční verze stojí na 1 rok 200€pro jeden projekt. Placená verze má navíc například nastavitelný HTML výstup, export do formátu docx nebo next business day support. Nástroj generuje výstup i bez nutnosti konfigurace, dokáže upozornit na potencionální problémy s navrženým API. Například pokud metoda vrací data, ale není přítomna anotace *@Produces* definující, v jakém formátu budou data. Dokáže pracovat s Java generickými typy za pomocí vlastní anotace, kde zadefinování generického typu u návratového typu bude vypadat následovně: *@ReturnType("java.util.List<com.company.package.car.parts.CarPart>")*. Kromě anotací dokáže zpracovávat také JavaDoc.

Výhody

- Jednoduchá integrace do projektu, defaultně lze bez konfigurace.
- Vyhledávání v dokumentaci podle filtrů (část URL, typ metody, ...).
- Upozornění na potencionální problémy v API.
- Podpora Jackson pro přesnou JSON payload dokumentaci.
- Podpara Java generických typů.
- Integrace v Apache Maven.

Nevýhody

- Neobsahuje testovacího klienta.
- Nelze použít free verzi na jakékoliv projekty kvůli licenčním podmínkám.
- Nutnost registrace projektu.
- Nepodporuje jiné transportní technologie kromě REST.
- Není pod open-source licencí.

2.3.3 SoapUI

Soap UI [17] je desktopový nástroj určený pro testování transportních technologií jako jsou SOAP-based webové služby, RESTful služby, JMS a jiné. Nástroj je pod open-source licencí European Union Public Licence [4]. SoapUI neslouží pro generování dokumentace ale k testování systémů vystavujících API pro vzdálená volání. Projekt je postavený na platformě Java a jeho uživatelské rozhraní je tvořeno ve Swingu. Typický příklad užití aplikace je naimportování souboru s popisem služby (WSDL, WADL). Při importu se vytvoří pro každou metodu formulář, který obsahuje nastavení, jak danou metodu volat včetně payloadu, adresy služby atd. SoapUI je velmi intuitivní a zároveň obsahuje veškeré potřebné nastavení pro různé volání služeb. Díky těmto vlastnostem se stal velmi oblíbeným a používá se od malých projektů až po projekty ve velkých korporacích.

Výhody

- Pracuje se SOAP-based webovými službami, RESTful službami, JMS a jinými.
- Request se dá editovat pomocí formuláře (urychluje práci).
- Lze vidět přenos dat až na úrovni HTTP protokolu.
- Intuitivní a zároveň široce přizpůsobitelné.

Nevýhody

- Nejedná se o generátor API dokumentace.
- V testované verzi (4.6.4) při načtení WADL přestala aplikace reagovat, přestože byl WADL validní vůči svému XML Schema.

2.3.4 Shrnutí

Při provádění rešerše se podařilo nalézt nástroje, které požadavkům vyhovují pouze z části. Některé nástroje vyhovovaly více a jiné méně. Je překvapující, že na tak populární Web Services (JAX-WS) neexistuje téměř žádný nástroj pro tvorbu dokumentace vzdáleného rozhraní. Pro REST služby je již situace lepší a v nabízených nástrojích se vyskytují poměrně schopná a uspokojující řešení, ze kterých bude moci nový nástroj brát inspiraci. Dle mého názoru není nabídka nástrojů pro tvorbu API dokumentace s klientem pro SOAP-based webové služby i REST příliš bohatá, a proto zde vidím potenciál pro nově vznikající nástroje.

2.4 Analýza požadavků

V této kapitole jsou analyzovány požadavky na vznikající nástroj. Analýza vychází ze specifikací popsaných v kapitolách 2.2.1 a 2.2.2, které se zabývaly možnostmi definice vzdáleného rozhraní na platformě Java EE. Dále analýza vychází z poznatků získaných v kapitole 2.3, kde byly odhaleny způsoby generování spolu se silnými a slabými stránkami alternativních nástrojů.

Nástroj bude nést název JRAPIDoc. Bude jej možné připojit do jakéhokoliv Maven projektu, aniž by bylo nutné měnit jeho zdrojový kód, který obsahuje vzdálené rozhraní. Nástroj bude možné připojit k již existujícímu projektu například až v závěrečné fázi vývoje a vygenerovat tak API dokumentaci. Umožněno bude také volitelné použití doplňujících anotací nástroje. Tyto anotace jsou určeny pro rozšíření a zkvalitnění API dokumentace, protože ne všechna data lze vyčíst na základě introspekce kódu rozhraní.

2.4.1 Požadavky na RESTful dokumentaci

Kapitola REST požadavky obsahuje popis toho, co bude nutné uchovávat v modelu vytvořeného ze vzdáleného rozhraní a jakým způsobem bude tato skutečnost v modelu reprezentována. Jako základní kámen bude v modelu seznam kořenových resourců. Každý resource bude jednoznačně identifikován svou URI, volitelně bude obsahovat popis resource a seznam metod.

2.4.1.1 RESTful model rozhraní

Cílem je vytvořit model, který v sobě nebude nést informace, jak je vzdálené rozhraní navrženo v Javě (například nebude znát, zda je atribut definován v konstruktoru nebo třeba jako parametr metod). Nebude rozpoznatelné, zda-li je vzdálené rozhraní vytvořeno s pomocí JAX-RS nebo jiné technologie (například servletů). Model bude zachovávat REST pohled na rozhraní.

2.4.1.2 Dědičnost a sub-resource lokátory

Na rozdíl od jiných bude nástroj korektně podporovat JAX-RS dědičnost resourců (popsanou v kapitole 2.2.2.5) a sub-resource lokátory (popsané v kapitole 2.2.2.4). Obě tyto vlastnosti JAX-RS jsou velmi důležité a jejich podpora pomůže nástroj odlišit od konkurence.

2.4.1.3 Atributy

Plugin bude podporovat atributy (popsané v kapitole 2.2.2.3) a díky tomu bude zvýhodněn oproti jiným generátorům, ve kterých je podpora atributů jen částečná. Atributy pro API dokumentaci budou podporovány plně podle specifikace JAX-RS, tzn. všechny typy atributů nesoucí anotace *@FormParam*, *@HeaderParam*, *@CookieParam*, *@MatrixParam*, *@QueryParam* a *@PathParam* a zároveň budou brány ze všech specifikovaných míst v kódu. Těmi jsou parametry konstruktoru, atributy resource třídy, setter metody resource třídy a parametry metod. Podporovány budou také bean properties. K atributům bude možné volitelně doplnit popis a indikaci, zda je atribut povinný či nikoliv.

Nejdůležitějšími částmi v modelu budou:

- metody obsahující URI,
- typ HTTP metody,
- atributy získané nejen z parametrů metody,
- popis,
- příznak, zda je metoda synchronní či asynchronní,
- typ entity body parametru,
- seznam možností, které může metoda vrátit jako odpověď klientovi včetně výjimek.

Seznam odpovědí

V pluginu bude nutné vyřešit několik problémů. Jedním z nich je, že metoda může díky dynamickému rozhodování vracet více možností až v runtime, což je spjato s návratovým typem *Response*. Pro tyto případy budou vytvořeny anotace, které budou přidány k metodě. Anotace budou tyto informace uchovávat a nástroj na jejich základě vytvoří dokumentaci.

Pokud dojde v běžícím systému k výjimce, která je propagována až do JAX-RS runtime, je taková výjimka přebalena na *WebApplicationException* nebo na své potomky. Vyhozená výjimka je také druh odpovědi. Pro zanesení výjimek do dokumentace bude sloužit nástrojem definovaná anotace pro dokumentování standardních odpovědí.

2.4.2 Požadavky na SOAP-based dokumentaci

Tato kapitola obsahuje požadavky na informace získané z introspekce kódu pro SOAP-based webové služby. Stejně jako v kapitole 2.4.1 je i zde cílem tvorba seznamu služeb (v RESTful seznam resourců). Každá SOAP-based webová služba bude nést informaci o jednoznačném identifikátoru v podobě názvu služby (*service name*), volitelný popis služby a nakonec seznam metod, které služba definuje.

2.4.2.1 SOAP-based model rozhraní

Model rozhraní bude vytvářen z pohledu SOAP-based webových služeb a nebude mít do sebe zanesen pohled návrhu v Javě. To znamená, že návrh z Javy bude transformován na obecný model SOAP-based webových služeb a nebude možné rozeznat, jakou technologií a přístupem bylo rozhraní navrženo. V modelu například nebudou následující informace:

- zda je služba definována implicitním či explicitním SEI,
- zda je návratový typ definován jako parametr Java metody uvnitř typu *Holder*<*T*> s módem *OUT* či jako návratový typ Java metody.

Takové informace by byly v dokumentaci nežádoucí, protože by se nejednalo o pohled na SOAP-based webové služby, ale o pohled návrhu rozhraní na úrovni Javy.

Anotace budou třeba pro přidávání popisu a podobně. Na rozdíl od JAX-RS, kde je možné využít návratový typ *Response* a je žádoucí přidat pomocnou anotaci pro zkvalitnění dokumentace, lze v JAX-WS většinu informací vyčíst na základě statické analýzy kódu.

2.4.2.2 Metody

Metody budou v modelu nést následující informace:

- o svém názvu,
- volitelně o svém popisu,
- příznaku asynchronosti,
- seznam vstupních objektů,
- seznam vstupních soap hlaviček,
- seznam návratových objektů,
- seznam návratových soap hlaviček,
- zda je metoda jednosměrná,
- jaká je použita serializace/deserializace dat mezi SOAP zprávami a Java objekty.

2.4.3 Jednotný model vzdáleného rozhraní

Model vzdáleného rozhraní je dokument obsahující popis služeb obdobně jako WSDL nebo WADL. Takový univerzální model bude možné použít jak pro popis služeb navržené jako RESTful, tak pro SOAP-based webové služby. Pro jednotlivé služby nebude nutné implementovat odlišný model. Přestože jsou služby rozdílné, mají několik společných principů - přijímání zpráv, zasílání odpovědí, využívání stejného protokolu HTTP (v drtivé většině), dostupnost na určité URI atd. V budoucnu by jako rozšíření nástroje mohl k těmto typům služeb přibýt další typ. Tento univerzální model by měl být schopný pojmout nový typ s minimem rozšíření, kterým může být například podpora JMS.

Pro uspokojení požadavků na model byly navrženy dílčí části, ze kterých se bude výsledný model agregovat:

API model

- základní URI pro skupinu služeb - povinné
- popis skupiny služeb
- seznam služeb

Služba

- identifikátor služby (*path* pro RESTful webové služby, *service name* pro SOAP-based webové služby) - povinné
- popis služby
- seznam metod

Metoda

- *path* metody (pro RESTful webové služby) - povinné
- *operation name* (pro SOAP-based webové služby) - povinné
- popis metody
- způsob serializace dat (SOAP binding) (pro SOAP-based webové služby) – povinné
- typ HTTP metody – povinné
- HTTP parametry (path, query, ...) (pro RESTful webové služby)
- příznak asynchronní metody – povinné
- seznam vstupních transportních typů
- seznam vstupních transportních typů soap hlaviček (pro SOAP-based webové služby)
- seznam možností odpovědí

HTTP parametr

- název parametru - povinné
- popis parametru
- příznak pro povinný/volitelný
- datový typ – povinné (s výjimkou pro header parametr, který má možnost místo datového typu výčet povolených hodnot)

Transportní typ

- popis typu
- datový typ - povinné

Odpověď

- HTTP status – povinné
- popis možnosti odpovědi
- seznam HTTP header parametrů
- seznam HTTP cookie parametrů (pro RESTful webové služby)
- seznam návratových transportních typů soap hlaviček (pro SOAP-based webové služby)
- seznam návratových transportních typů

Pokud budou v dokumentované aplikaci oba typy služeb, výsledkem budou dva vygenerované modely (jeden model pro každý typ).

2.4.3.1 Datový formát modelu

Datovým formátem modelu je myšleno, do jakého typu textového formátu bude model generován. Jako nejvhodnější kandidáti se nabízejí XML a JSON. Oba typy formátu jsou vyhovující, ale pokud má být výsledný model načítán klientem vytvořeným jako HTML stránka, je JSON jednoznačně vhodnější volbou. JavaScript ve webovém prohlížeči má totiž zabudovaný parser pro JSON a dokáže velmi elegantně z textu vytvářet JavaScript objekty a obráceně. Pro JSON formát se rozhodly i jiné nástroje generující dokumentaci (popsáno v 2.3) a proto se nabízí myšlenka použít již vytvořené specifikace pro model rozhraní (například Swagger Specification[16]). Existují ale dva hlavní důvody, proč vytvořené specifikace nepoužít:

1. Byly vytvořeny za účelem podpory pouze RESTful webových služeb , nehodí se tedy pro požadavky SOAP-based webových služeb.
2. Nově vznikající nástroj si klade za cíl generování podrobnější dokumentace než jeho konkurence a proto je pro něj stávající specifikace nedostatečná.

Přestože pro nástroj vznikne nová specifikace (model vzdáleného rozhraní), je do určité míry inspirována již existujícími specifikacemi.

2.4.3.2 Analýza modelu

Skupiny služeb

V modelu budou služby shlukovány do skupin podle shodného URI prefixu. Například jedna skupina služeb může mít prefix *http://localhost:8080/exampleapp/rest* a druhá *http://localhost:8080/exampleapp/admin/rest*. V extrémním případě může pro každou službu existovat jedna skupina. Každá služba ve skupině má jednoznačný identifikátor. Pro RESTful webové služby je to *path* definovaná v kořenové resource třídě popsané v 2.2.2.1. Spojením URI prefixu a *path* kořenové resource třídy se získá URI dostupnosti služby. U SOAP-based webových služeb je jednoznačným identifikátorem *service name* popsané v 2.2.1.1.

Popis

Jedná se o různé popisy jako jsou:

- popis skupiny služeb,
- popis služby,
- popis metody,
- popis parametru,
- popis typu,
- popis typu hlavičky,
- popis možnosti odpovědi.

[16] Dokument popisující službu obdobně jako WSDL či WADL

Popis bude moci být volitelně definován pro položku modelu a bude získáván z konfigurace pluginu v pom.xml a z introspekce kódu, kde bude v podobě anotací. Umístění popisu je určeno tím, ke které položce je popis určen.

Metoda

Metoda RESTful webové služby je jednoznačně identifikovaná v rámci služby svou *path* a typem HTTP metody. Pokud se jedná o metodu služby SOAP-based webové služby, je metoda jednoznačně identifikovaná svým názvem.

Co se týče seznamu vstupních a návratových typů, jsou u RESTful webových služeb a SOAP-based webových službeb zásadní rozdíly. RESTful webové služby umožňují pouze jeden vstupní typ a jeden návratový typ. SOAP-based webové služby umožňují v SOAP zprávě posílat neomezený počet vstupních a návratových typů. SOAP-based webová služba dále obsahuje jednosměrné metody a proto je důležité z hlediska dokumentace odlišit prázdný návratový typ v SOAP zprávě a nevrácení žádné SOAP zprávy. Pokud metoda SOAP-based webové služby vrací alespoň prázdnou SOAP zprávu, je dokumentována odpověd s HTTP statusem 200 OK bez transportních typů. Jedná-li se o jednosměrnou metodu, není seznam možností odpovědi vůbec vytvořen. Dalším rozdílem je, že RESTful webová služba může přijímat HTTP parametry (header a cookie navíc lze i odesílat), což SOAP-based webové služby nedefinují.

2.4.3.3 Typový systém

Ač to nemusí být na první pohled patrné, dokumentování typového systému Javy je jednou z nejdůležitějších částí nástroje. Jedná se o zdokumentování datových typů, které jsou určeny pro přenos mezi klientem a serverem. V této práci jsou označovány jako transportní typy.

Problematika v JAX-RS

Listing 2.8: Definice transportního typu

```
1  public class DestinationExample {
2
3      @com.fasterxml.jackson.annotation.JsonIgnore
4      private long id;
5      public String name;
6
7      public DestinationExample() {}
8      //settery...
9      //gettery...
10 }
```

V JAX-RS se pro mapování dat na Java typy a zpět používají entity provideři (podrobněji v 2.2.2.6). Pro analýzu typového systému je důležitý poznatek, že ke každému media typu lze zaregistrovat jiného entity providera. Jelikož není žádný entity provider v JAX-RS specifikaci pro formát JSON (media typ *application/json*) a XML (media typ *application/xml*) pevně definován (ani pro jiné media typy), spoléhají se určité implementace JAX-RS na konkrétní

implementace zabývající se serializací dat. Například Resteasy-3.0.8.Final [13] (implementace JAX-RS) má vytvořeny entity providery pro media typ *application/json* pracující s knihovnami buď Jackson 1.9.x, Jackson 2.2.x a nebo Jettison. Kromě tří zmíněných je možné poměrně snadno vytvořit jiného providera, který bude použit pro mapování s využitím jiné knihovny.

Listing 2.9: Vrácený objekt při použití media typu application/json

```
{
    "name": "Prague"
}
```

Listing 2.10: Vrácený objekt při použití media typu application/xml

```
<destination>
    <id>54</id>
    <name>Prague</name>
</destination>
```

Takováto diverzita providerů neposkytuje jednoznačný způsob tvorby transportních typů v dokumentaci. Na ukázce 2.8 je znázorněna velmi jednoduchá třída definující transportní typ, který se bude serializovat. Třída obsahuje anotaci *@JsonIgnore* způsobující ignorování atributu. Tato anotace má význam, pouze pokud je jako entity provider použita knihovna Jackson (media typ *application/json*). Například při použití entity providera JAXB (media typ *application/xml*) je anotace ignorována. Na ukázkách 2.9 a 2.10 je vidět výsledek volání stejné metody se serializací stejného objektu pomocí dvou různých entity providerů. Je tedy důležité mít možnost volby, k jakému providerovi se bude dokumentace vázat. Nástroj bude mít zabudovanou podporu pro nejběžnější media typy (*application/json* a *application/xml*) a bude umožňovat rozšíření o další a nahrazení zabudované podpory.

Java také obsahuje generika, která jsou v JAX-RS podporována. Nástroj bude podporovat jak klasické Java typy (typy typu *java.lang.Class*), tak i generické typy typu *java.lang.reflect.ParameterizedType* a *java.lang.reflect.TypeVariable*. Generika budou podporovaná jak pro vstupní, tak návratové typy metod. Budou moci být dokumentovány metody s podobnou signaturou jako na ukázce 2.11.

Listing 2.11: Příklad podporovaných signatur resource metod

```
1  public void foo1(List<String> l);
2
3  public List<String> foo2();
4
5  public void foo3(Map<String, String> m);
6
7  public Map<String, String> foo4();
8
9  public void foo5(TypeB<TypeA> d);
10
11 public TypeB<TypeA> foo6();
12
13 public <T extends TypeA> void foo7(T t);
14
15 public <T extends Destination> T foo8();
16
17 public <T extends TypeA> void foo9(List<T> t);
18
19 public <T extends TypeA> List<T> foo10();
20
21 public <T extends List<TypeA>> void foo11(T l);
22
23 public <T extends List<TypeA>> T foo12();
24
25 public <T> void foo13(T t);
26
27 public <T> T foo14();
```

Problematika v JAX-WS

V JAX-WS kontextu je situace jednodušší. Standardně se vše serializuje pouze do formátu XML a používá se pro serializaci JAXB. A jak je popsáno v 2.4.2.1, není nutné přidávat do rozhraní jiné anotace než ty, co jsou v JAX-WS. Entity provider pro JAXB bude zabudován v pluginu, ale možnost nahrazení ho za jiný bude existovat stejně jako pro RESTful webové služby.

2.4.3.4 Rozšiřující operace nad modelem

Aby byl nástroj co nejvíce flexibilní, bude možné mezi fázemi vytvoření modelu v paměti a vygenerováním modelu do souboru rozšířit nástroj o operace nad modelem. Tyto operace budou moci model nejen číst a analyzovat, ale také modifikovat. Poté, co proběhnou operace nad modelem, bude tento modifikovaný model předán komponentě pro vygenerování do souboru. Jednou z ukázek operace může být kontrola, zda všechny metody v modelu mají vytvořen popis a druhou změna názvu datového typu. Tyto operace budou přidávány a odebírány na základě jejich přidávání a odebírání z konfigurace nástroje v pom.xml.

Při běhu operace budou moci vznikat chyby, které budou propagovány ven z operací. Chyby budou rozděleny do čtyř kategorií:

- ***STOP_HANDLER_CURRENT*** – ukončí se právě zpracovávaná operace a přejde se na vykonání další, pokud je nakonfigurována,
- ***STOP_HANDLERS*** – ukončí se právě zpracovávaná operace, ostatní nakonfigurované operace se přeskočí a přejde se ke generování modelu do souboru,
- ***FAILURE_EXCEPTION*** – bude vyhozena výjimka MojoFailureException, její význam popsán v [21]
- ***EXECUTION_EXCEPTION*** – bude vyhozena výjimka MojoExecutionException, její význam popsán v [21]

2.4.4 GUI klient

Klient bude umožňovat zobrazení vygenerovaného modelu v grafické podobě, kde budou zachyceny všechny informace z modelu.

Klient bude tvořen webovými technologiemi HTML, CSS a JavaScript a bude k dispozici na domovských stránkách nástroje, odkud půjde stáhnout a připojit k projektu. Jelikož základní klient bude tvořen webovými technologiemi, bude možné jednoduše upravit vzhled a funkcionalitu přesně podle konkrétních požadavků, jaké bude projekt vyžadovat.

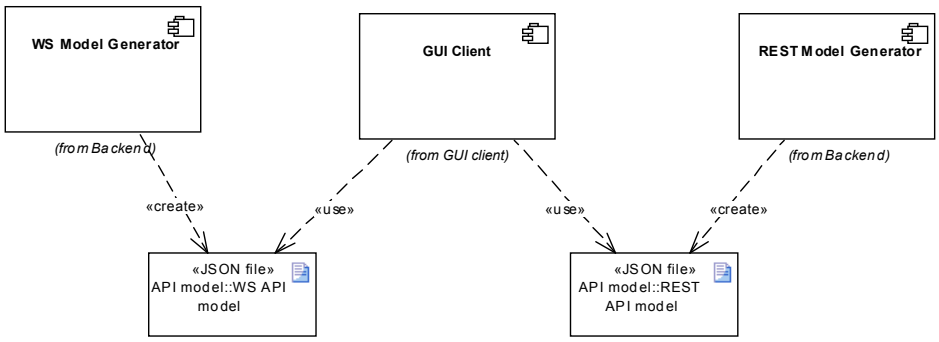
Načítaný model (vygenerovaný pluginem), který je ve formátu JSON, bude v klientu transformovaný na DOM a vložený do stránky. Defaultně bude model načítán z URI nakonfigurované v klientu, ale uživateli bude umožněno zadání jiné URI adresy, odkud dojde ke stažení modelu, transformaci a zobrazení.

Kapitola 3

Návrh

Tato kapitola obsahuje návrh řešení vyplývající z analýzy požadavků. Obsahuje návrh architektury jak nástroje samotného, tak jeho komponent, kterými jsou: Maven plugin a GUI klient. Budou zde popsány výhody a nevýhody zvoleného návrhu.

3.1 Architektura



Obrázek 3.1: Architektura nástroje

Na 3.1 je znázorněna high-level architektura nástroje. Jsou zachyceny tři hlavní komponenty a dva artefakty předávané mezi komponentami. Komponenty SOAP Maven plugin a REST Maven plugin jsou popsány detailněji v 3.1.1, komponenta GUI klienta je podrobněji popsána v 3.1.2. Pluginy jsou na sobě nezávislé a je tedy možné používat v projektu pouze jeden z nich.

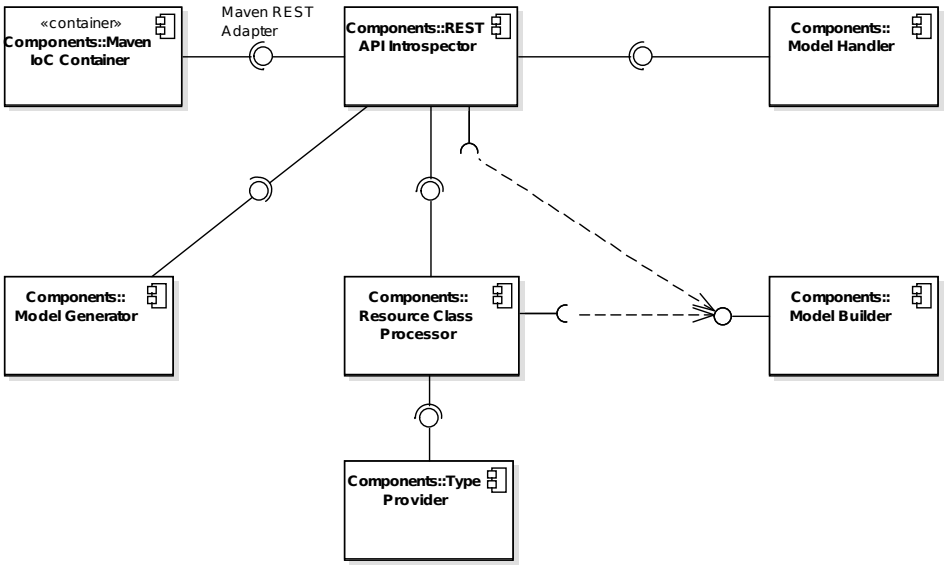
3.1.1 Backend architektura

Jelikož mají pluginy téměř identickou architekturu, je popisován pouze jeden z nich. Na 3.2 je vidět diagram komponent pro REST plugin.

Význam a odpovědnost jednotlivých komponent:

Maven IoC Container

Komponenta reprezentující Maven proces, který pomocí dependency injection [25] nakonfiguruje a vytvoří instanci Maven REST Adapter.



Obrázek 3.2: Architektura pluginu pro RESTful webové služby

Maven REST Adapter

Komponenta sloužící jako wrapper pro předání řízení komponentě REST API Introspector. Jedná se o návrhový vzor adapter, který zde slouží pro oddělení Maven API od funkcionality pluginu. V budoucnu se může přidat adapter pro jiný sestavovací nástroj, než je Maven a nebude nutné provádět žádný zásah do stávajícího zdrojového kódu.

Resource Class Processor

Komponenta provádějící introspekci REST API a JRAPIDoc anotací. Při introspekci jsou za pomocí komponenty Model Builder vytvářeny části modelu, ze kterých se postupně tvoří výsledný model.

Model Builder

Komponenta budující model, která obsahuje doménové třídy reprezentující model a ke každé doménové třídě odpovídající builder třídu, která usnadňuje vytváření. Pro builder třídy je použit návrhový vzor builder.

Type Provider

Tato komponenta se stará o zanesení informací o transportních typech do modelu. Řeší problematiku popsanou v 2.4.3.3. Pomocí tovární metody je prováděna instanciace konkrétního type providera, který se dále stará o samotnou funkcionalitu komponenty. Type provider obsahuje cache, kde uchovává již prozkoumané typy, aby se nemusely opětovně prohledávat. Cache také slouží jako úplný seznam typů, které se nakonec zakomponují do modelu.

Model Handler

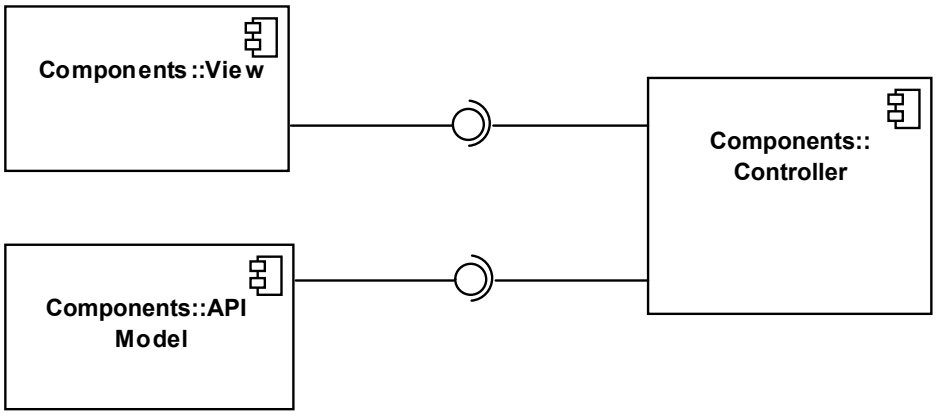
Uvnitř této komponenty se provádějí operace nad modelem definované v 2.4.3.4. Obsahuje factory metodu provádějící instanciování nakonfigurovaných handlerů, kteří obsahují naprogramované chování pro zpracování modelu. V komponentě jsou tyto handlery spouštěny a je zde i zpracování chyb vyskytujících se během provádění handlerů.

Model Generátor

Komponenta zodpovědná za vygenerování předaného modelu do souboru. Obsahuje také konfiguraci způsobu serializace modelu.

3.1.2 Architektura GUI klienta

Pro návrh GUI klienta byl zvolen architektonický styl MVC. Přestože se jedná o jednoduchého klienta, budou z důvodu jednoduché rozšiřitelnosti a přehlednosti vytvářeny komponenty a bude mezi ně rozdělena zodpovědnost.



Obrázek 3.3: Architektura GUI klienta

Význam a odpovědnost jednotlivých komponent:

API Model

Komponenta zodpovědná za načtení modelu a jeho uložení v ní.

View

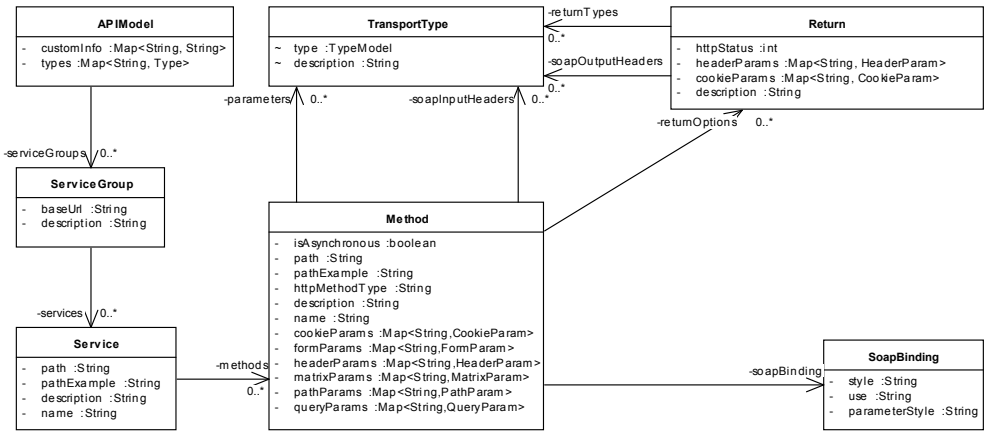
Komponenta zodpovědná za zobrazování dat uživateli na obrazovce. Bude zobrazovat grafickou podobu modelu, vyskytující se chyby (například při načítání modelu) apod.

Controller

Komponenta zodpovědná za inicializaci view i modelu. Obsahuje události vzniklé ve view a funkcionalitu pro zaregistrování těchto událostí.

3.2 Návrh modelu vzdáleného rozhraní

Na 3.4 je znázorněn diagram tříd reprezentující model vzdáleného rozhraní, který vznikl na základě analýzy v 2.4.3. Pro RESTful webové služby i SOAP-based webové služby bude existovat pouze jeden. Zda-li se vytvoří instance reprezentace API k RESTful webovým službám či SOAP-based webovým službám, bude záležet na řídící komponentě tvorby modelu, která jej bude vytvářet.



Obrázek 3.4: Model reprezentující vzdálené rozhraní

Popis tříd reprezentujících model:

APIModel

Hlavní kořenová třída obsahující tři hlavní atributy. První atribut *customInfo* reprezentuje přídavné informace obdržené z konfigurace pluginu. Položky jsou vždy párové textové řetězce klíč-hodnota. Následujícím atributem je seznam služeb sdružených ve skupinách podle společné URI. Posledním atributem je seznam transportních typů ze všech skupin služeb. Typy jsou zde vytvořeny na základě rekurzivního procházení každého typu a jeho atributů.

ServiceGroup

Třída reprezentující skupinu služeb se stejným URI prefixem.

Service

Třída reprezentující kořenovou resource třídu (pro RESTful webové služby) nebo SEI (pro SOAP-based webové služby), která obsahuje seznam svých metod.

Method

V JAX-RS kontextu třída reprezentuje jak resource metody kořenové resource třídy, tak i rekurzivně obsažené metody resource tříd vrácených sub-resource lokátorem popsané v 2.2.2.4. Pro JAX-WS třída reprezentuje webovou operaci a obsahuje seznam vstupních parametrů (JAX-RS může mít vždy jen jeden parametr). Při užití v JAX-WS má metoda seznam vstupních SOAP hlaviček. JAX-WS i JAX-RS obsahují seznam možností odpovědí. Seznam může být prázdný, pokud je metoda JAX-WS jednosměrná.

SoapBinding

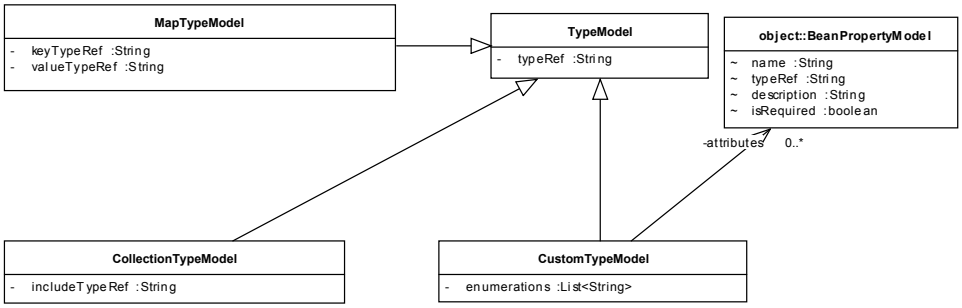
Třída nesoucí informace o způsobu serializování zpráv v kontextu JAX-WS.

Return

Třída nereprezentuje návratový typ metody, jak evokuje její název, ale jednu možnou odpověď, jakou může metoda vrátit. Odpověď se skládá z HTTP statusu, návratových typů, návratových typů SOAP hlaviček, seznamu HTTP hlaviček a seznamu HTTP cookie.

TransportType

Třída reprezentující transportní typ. Jejími atributy jsou popis transportního typu a datový typ vytvořený type providerem (podrobněji v 2.4.3.3).
Návrh datových typů pro model znázorňuje diagram tříd 3.5. Návrh obsahuje tři možné typy:



Obrázek 3.5: Reprezentace datových typů pro transport

CollectionTypeModel

Datový typ vycházející z Javovských kolekcí a polí. Obsahuje atribut nesoucí odkaz na typ, jež je možné do kolekce či pole vložit. Může odkazovat na jakýkoliv typ, který je potomkem třídy *TypeModel*.

MapTypeModel

Třída reprezentující Javovský datový typ *Map<K, V>*. Její atributy odkazují na datové typy, jež reprezentují klíče a hodnoty, které mohou být do mapy vloženy. Mohou odkazovat na jakýkoliv typ, který je potomkem třídy *TypeModel*.

CustomTypeModel

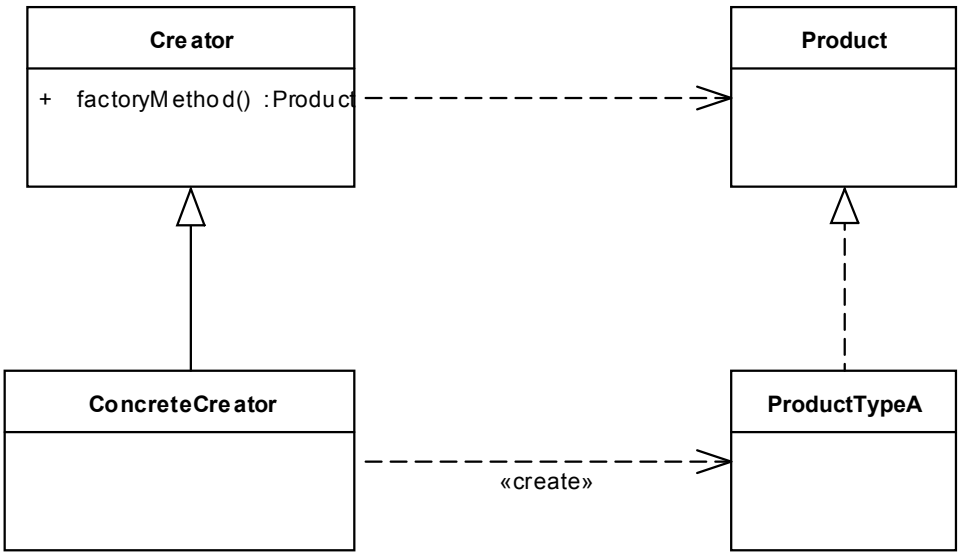
Třída reprezentuje datový typ běžných typů v Javě. Zahrnuje jak primitivní datové, tak i objektové typy. Bude obsahovat všechny ostatní Javovské typy, které nepatří do skupin výše zmíněných.

3.3 Návrhové vzory

Tato kapitola obsahuje popsané návrhové vzory, jenž jsou použity v nástroji.

Factory method

Návrhový vzor patří do skupiny „creational“, která se zabývá principy pro vytváření objektů. Popisuje, jak vytvořit objekt bez použití konkrétní třídy. Tovární metodě jsou předány



Obrázek 3.6: Factory method pattern

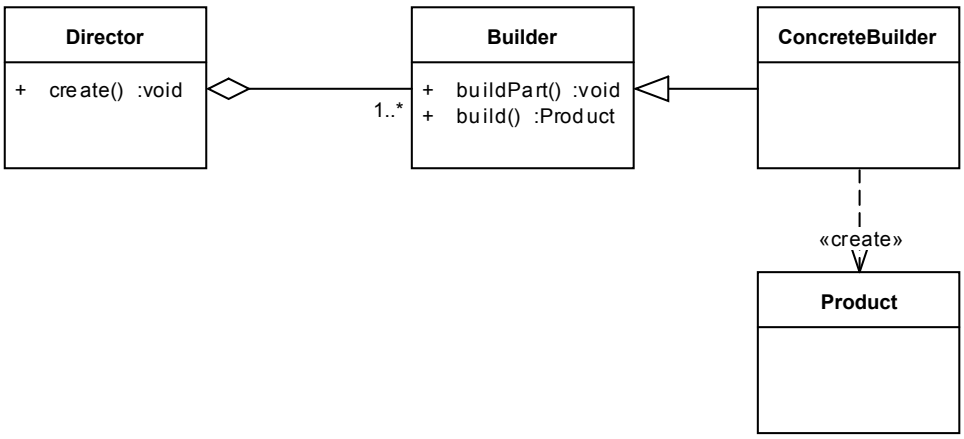
parametry, na základě kterých je rozhodnuto, ze které třídy bude vyrobena a vrácena instance. Je zde využito polymorfizmu, protože tovární metoda vrací instance tříd mající stejný nadtyp. Vzor tovární metoda je v nástroji použit v komponentách Model Handler a Type Provider popsaných v 3.1.1. Návrhový vzor je znázorněn na diagramu 3.6.

Builder

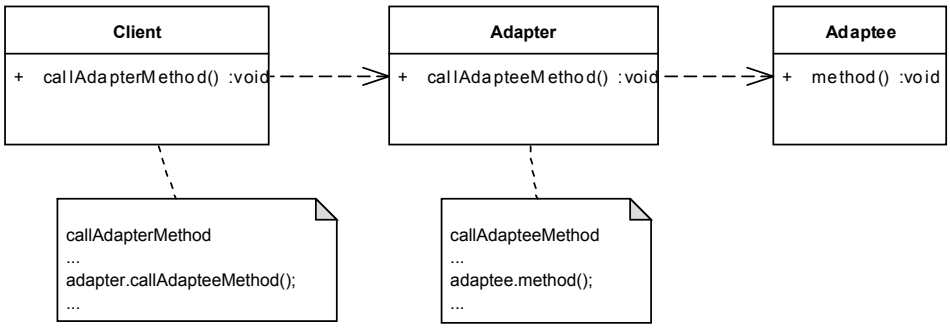
Návrhový vzor patřící také do skupiny „creational“ vzorů a zabývající se principy vytváření objektů. Používá se pro konstrukci složitých objektů, kde se složité vytváření oddělí od reprezentace dat. Hlavní části v builder vzoru jsou vytvářený objekt, builder objekt obsahující funkcionalitu pro tvorbu vytvářeného objektu a director objekt, který řídí objekt builder a tím i celý proces tvorby objektu. Jakmile director ukončí vytváření objektu, zavolá na builder objektu metodu *build*, která vrátí vytvořený objekt. Vzor builder je použit pro tvorbu modelu a rekurzivně pro všechny jeho části. Model se nachází v komponentě Model Builder popsané v 3.1.1. Návrhový vzor je znázorněn na diagramu 3.7.

Adapter

Návrhový vzor adapter patří do skupiny „structural“ návrhových vzorů, jež se zabývají návrhem spojení mezi objekty. Používá se v případech, kdy existují dvě komponenty, ovšem nedokážou spolu komunikovat, protože vystavené rozhraní jedné komponenty neodpovídá požadovanému rozhraní druhé komponenty. Hrají zde roli objekty adaptovaný objekt, klient a adapter. V případech, kdy klient požaduje jiné rozhraní, než vystavuje adaptovaný, vloží se mezi klienta a adaptovaného objekt adapter. Ten bude zodpovědný za vystavení takového rozhraní klientovi a konzumování rozhraní adaptovaného, aby mohly komponenty mezi sebou komunikovat. Návrhový vzor je znázorněn na diagramu 3.8.



Obrázek 3.7: Builder pattern



Obrázek 3.8: Adapter pattern

Kapitola 4

Implementace

4.1 Použité technologie

Pro vývoj nástroje byly použity následující technologie:

Java 6.0 [8] - Vyvíjený nástroj byl vyvíjen a kompilován do Java verze 6.0.

JAX-WS 2.0 [23] - Nástroj při vytváření modelu analyzuje kód JAX-WS specifikace ve verzi 2.0.

JAX-RS 2.0 [29] - Nástroj při vytváření modelu analyzuje kód JAX-RS specifikace ve verzi 2.0.

Apache Maven 3.3.1 [9] - Maven pluginy v nástroji byly vyvíjeny na verzích knihoven 3.3.1.

Reflections 0.9.9-RC2 [15] - Pro analýzu bytecode a vyhledání anotovaných tříd bylo použito Reflections ve verzi 0.9.9- RC2.

Java Reflection API - Při analýze kódu bylo využito rozhraní Java Reflection API, jehož verze je vázaná k verzi Javy.

Jackson 2.5.0 [7] - Knihovna ve verzi 2.5.0 použitá pro tvorbu transportních typů.

JUnit 4.11 [6] - Nástroj byl testován jednotkovými testy s použitím frameworku JUnit ve verzi 4.11.

Json2html [5] - JavaScriptová knihovna zodpovědná za transformaci modelu v JSON formátu do HTML. Transformace je prováděna na základě vytvoření šablony a její aplikace na JSON objekt.

4.2 Vyhledávání tříd rozhraní JAX-WS a JAX-RS

Vyhledávání JAX-WS a JAX-RS spočívá ve vyhledání tříd majících anotace *@Path* a *@WebService*, které jsou na classpath projektu. Zde jsou dvě možnosti jak třídy vyhledat. První

přístup je získání seznamu všech tříd na classpath a jejich postupné nahrávání do paměti instancí třídy *ClassLoader*. Po nahrání třídy do paměti následné dotazování pomocí Java Reflection API, zda třída obsahuje požadovanou anotaci. Tento přístup však není nejlepší volbou, především z hlediska časové náročnosti. V nejhorším případě budou muset být nahrány všechny třídy, což může u většího projektu trvat desítky sekund. Nástroj používá druhou metodu získání tříd, která je rychlejší. Třídy nejsou nahrávány do paměti, ale nejdříve proběhne analýza bytecode a vyberou se pouze ty, které vyhovují a ty se nahrají do paměti. Po jejich nahrání je možné nad nimi používat Java Reflection API. Nástroj pro tento přístup používá knihovnu Reflections, která je nástupcem knihovny Scannotation. Knihovně je třeba předat instanci *ClassLoaderu* patřící k projektu a ne instanci, která patří k pluginu.

4.3 Zpracování JAX-RS a JAX-WS API

Ke zpracování vzdáleného rozhraní je použito Java Reflection API, pomocí kterého probíhá analýza vytvořeného rozhraní. Rozhraní mohou obsahovat oproti klasickým třídám anotace a datové typy z JAX-RS API, JAX-WS API a JRAPIDOC API. Java Reflection API je vlastností jazyka Java, která dokáže vykonávat nebo analyzovat sama sebe.

4.4 Přídavné anotace

Při použití specifikací pro tvorbu vzdáleného rozhraní nelze introspekcí kódu zjistit všechny potřebné informace, které jsou definovány v požadavcích v kapitole 2.4. Jsou pro to vytvořeny přídavné anotace nástroje JRAPIDoc, které mají za cíl pomoci tvořit detailnější dokumentaci. Anotace jsou v samostatném balíku a lze je tedy volitelně připojit do projektu. Připojení probíhá standardním přidáním Maven závislosti do projektu. Jazyk Java nedovoluje pomocí anotací poskytnout takový přístup, který by dokázal popsat všechny možné detaily rozhraní a zároveň, aby bylo anotování efektivní a přehledné. Vznikl proto kompromis, který je efektivní, přehledný, sémantický a zároveň dokáže pokrýt většinu požadavků na tvorbu dokumentace.

Nově vzniklé anotace:

@org.jrapidoc.annotation.DocDescription

Přidává popis k elementům rozhraní.

@org.jrapidoc.annotation.rest.DocIsRequired

Indikuje, zda je parametr či atribut povinný.

@org.jrapidoc.annotation.rest.DocPathExample

Protože v anotaci *@Path* mohou být regulární výrazy, které nemusejí být dobře čitelné, lze v této anotaci uložit ukázku URI vyhovující regulárnímu výrazu.

@org.jrapidoc.annotation.rest.DocReturn

Anotace slouží k tvorbě možnosti odpovědi a je vytvořena především pro situace, kdy je návratovým typem *Response*, ze kterého nelze při statické analýze získat žádné informace.

@org.jrapidoc.annotation.rest.DocReturns
Při více možnostech odpovědí lze možnosti vložit do této anotace.

@org.jrapidoc.annotation.soap.DocReturn
Pomáhá lépe zdokumentovat možnosti odpovědi.

@org.jrapidoc.annotation.soap.DocReturns
Při více možnostech odpovědí lze možnosti vložit do této anotace.

Další podrobnosti o přídavných anotacích lze nalézt v javadoc.

4.5 Tvorba transportních typů

Tvorba transportních typů může být prováděna buď zabudovanými třídami pro jejich tvorbu, nebo mohou být vytvořeny jiné třídy, které se nakonfigurují jako náhrada za původní funkcionalitu. Defaultně se pro část RESTful webových služeb používají třídy využívající knihovnu Jackson a pro SOAP-based webové služby knihovna JAXB. V nástroji jsou ještě dvě třídy, které lze používat pro tvorbu transportních typů. První z nich je kombinace Jackson a JAXB, Jackson je primární a JAXB je sekundární. Druhou možností je také jejich kombinace, ale s obráceným pořadím. Výhodou použití těchto zabudovaných tříd v nástroji je, že při tvorbě transportních typů jsou užity anotace z těchto knihoven. Výsledný typ tedy bude odpovídat přesně tomu, jak jej bude serializovat entity provider.

Kapitola 5

Testování

5.1 Testování vývojářem

Testování probíhalo paralelně s procesem vývoje. Cílem bylo ověřit, zda každá nově přidaná část kódu provádí to, co se od ní čeká. Testování probíhalo připojením nástroje k Maven projektu obsahujícímu JAX-RS a JAX-WS. Proces testování vývojářem probíhal provedením sestavení nástroje a následným spuštěním sestavení testovaného projektu s módem debug. Maven umožňuje mód debug spustit namísto běžného *mvn* příkazem *mvnDebug*, který otevře port a čeká, dokud se debugger na tento port nepřipojí. Poté probíhá standardní debug Java aplikace.

5.2 Jednotkové testy

Jednotkové testy byly zaintegrovány do sestavování nástroje, jejich vykonání a dosažené výsledky tedy byly kontrolovány s každým novým sestavením nástroje. Cílem jednotkového testování bylo provedení regresních testů. Pro měření kvality nástroje bylo vytvořeno přes sto jednotkových testů.

5.3 Integrační testy

Integrační testování bylo realizováno zakomponováním pluginů JRAPIDoc nástroje do sestavování existujících projektů. Neslo si za cíl ověření chování pluginů na reálných projektech a výsledek jejich výstupu.

Integrační testování bylo provedeno jak na testovacím projektu, tak na rozsáhlých existujících projektech používajících pro tvorbu vzdálených rozhraní technologie JAX-RS a JAX-WS.

Prvním z existujících projektů byl projekt jBPM, jehož domovskou stránkou je *http://www.jbpm.org*. jBPM je flexibilní balík pro správu byznys procesů tvořící most mezi byznys analytiky a vývojáři. Produkt je dostupný pod Apache Software License 2.0 a je zaštiťován společností RedHat [14].

Druhým existujícím projektem byl Relationship Registry, jež slouží ke správě informací o používaných nástrojích na projektech uvnitř společnosti CA Technologies [3]. Vzhledem k tomu, že se jedná o interní projekt, bylo integrační testování delegováno na interního vývojáře společnosti.

5.4 Testování použitelnosti

Cílová skupina

Znalost tvorby RESTful webových služeb nebo SOAP-based webových služeb na platformě Java EE?	Ano
Zkušenost s projekty spravovaném pomocí Maven?	Ano
Zkušenost s přidáváním Maven pluginů do životního cyklu sestavování?	Ano

Cíle testování

Intuitivnost

Jedním z cílů testování je zjištění, jak je používání nástroje intuitivní. Uživatelé by neměli být vystaveni neintuitivní konfiguraci, která by je od používání mohla odradit či zdržovat. Naopak nástroj by měl mít možnost být spuštěn s minimální znalostí uživatele. Bude testováno, jak respondent využije pomocných funkcí a jestli mu byly nápomocné. Pomocnými funkcemi jsou goal *help* v pluginu a javadoc rozhraní nástroje.

Přehlednost

Bude testováno, zda uživatel porozumí úkonům, které simulují běžné použití nástroje a bude je moci bez větších potíží vykonat.

Provedení testů

Před provedením testů bude respondent seznámen se základními principy fungování nástroje a poté mu budou předány scénáře pro provedení uživatelských akcí a adresář s klientem. Po jejich vykonání mu budou vysvětleny rozšířené možnosti nástroje a předány další scénáře pro provedení. Od respondenta bude požadováno komentování pozitivních a negativních poznatků a komentování prováděných akcí ze scénářů. Celý průběh testování bude zaznamenán v audio podobě pro pozdější analýzy.

5.4.1 Testovací scénáře

Vygenerování modelu pro JAX-RS

1. Respondent připojí plugin do životního cyklu sestavování.

2. Respondent spustí goal help.
3. Plugin vypíše informace o možnostech konfigurace.
4. Respondent provede konfiguraci pluginu.
5. Respondent spustí sestavení projektu.
6. Plugin vygeneruje do souboru model rozhraní.
7. Respondent nalezne a otevře soubor s modelem.

Načtení modelu v GUI klientovi

1. Respondent nakopíruje do adresáře se zdroji GUI klienta.
2. Respondent otevře ve webovém prohlížeči stránku s klientem.
3. Načte se stránka.
4. Respondent vloží adresu umístění modelu a stiskne „Load model“.
5. Klient načte a zobrazí model.

Alternativní cesta:

5. Pokud byla respondentem vložena neplatná cesta zobrazí se chybová hláška.

Přidání popisu kořenové resource třídy

1. Respondent přidá Maven závislost *org.jrapidoc:jrapidoc-annotation:1.0-SNAPSHOT* do projektu.
2. Respondent přidá do kořenové resource třídy anotaci nacházející se v závislosti z kroku 1.
3. Respondent spustí sestavení projektu.
4. INCLUDE: Načtení modelu v GUI klientovi.
5. Respondent si zobrazí v modelu popis kořenové resource třídy.

5.4.2 Výsledky

Převážně v úvodu testování si respondent stěžoval na neexistující wiki stránku projektu. Návod konfigurace pluginu z goalu *help* se zdál být velmi stručný a pro člověka neznalého principů fungování pluginu téměř nic neříkající. Tento problém by měla vyřešit wiki stránka, která by obsahovala více podrobný popis konfigurace včetně jejích ukázek. Respondent navrhl, aby byl na wiki také ukázkový projekt, na kterém by bylo vidět, jaké vlastnosti plugin má a jak se dá použít.

Ve scénářích není uvedeno, k jakému typu modulu se má plugin připojit. Respondent si tak nebyl jistý, zda v multimodulovém projektu plugin připojit k modulu obsahující vzdálené rozhraní nebo k war modulu, kde je rozhraní přidáno jako závislost. Nakonec se rozhodl správně a připojil ho k war modulu.

Po seznámení se s nástrojem a představení jeho možností respondent ocenil funkce, kterými je plná podpora atributů v resource třídách a podpora subresource lokátorů. Pozitivně hodnotil také podporu SOAP-based webových služeb a možnosti rozšíření nástroje nebo nahrazení komponent za vlastní.

Co se týče GUI klienta, na první pohled působí trochu nepřehledně, obzvlášť když uživatel při procházení modelu neskrývá uzly, které již navštívil. Respondent navrhl lepší grafické zpracování. S výsledky nástroje byl spokojený. Dalším poznatkem bylo umístění tlačítek pro nahrávání modelu a pro přepínání mezi RESTful webovými službami a SOAP-based webovými službami.

Celkově nástroj zanechal působivý pocit z důvodů propracovanosti modelu a informací v něm zanesených. Respondent má zkušenosti s používáním nástroje Swagger, který je popsán v 2.3.1 a po seznámení se s novým nástrojem je ochotný přejít na používání nástroje JRAPIDoc. Většina navržených změn byla provedena.

Ze zpětné vazby byl vytvořen ukázkový projekt[26], na kterém je demonstrováno použití nástroje. Do GUI klienta byly aplikovány navržené změny pro lepší přehlednost a větší komfort načítání modelů. Byla vytvořena wiki stránka na stránkách projektu[27].

Kapitola 6

Výsledky a srovnání

Při srovnání nástrojů generujících dokumentaci ke vzdáleným rozhraním má nově vytvořený nástroj mnohé co nabídnout oproti alternativám. Při tvorbě nástroje bylo myšleno na pozitiva a negativa konkurenčních nástrojů popsaných v 2.3.

Co se týče porovnání tvorby dokumentace RESTful webových služeb na platformě Java EE, byl nástroj především srovnáván s nástrojem Swagger. Ten je v současné době nejpopulárnějším nástrojem na tvorbu dokumentace pro RESTful webové služby. Nástroj se od Swaggeru a dalších odlišuje kombinací plné podpory specifikace JAX-RS a otevřeným kódem. Nástroj se také snaží být jednoduchý na konfiguraci a ovládání na rozdíl od Swagger, který obsahuje poměrně velkou a pro začátečníka nesrozumitelnou konfiguraci.

6.1 Konfigurace

Pokud chceme provést konfiguraci Maven pluginu, je dobré si ji zobrazit pomocí spuštění goalu *help* u konkrétního pluginu. Při spuštění goalu pro Swagger plugin s parametrem *-Ddetail=true* se žádné informace nezobrazí a volání skončí vyhozenou výjimkou *NullPointerException*. Plugin nástroje Miredot goal pro zobrazení konfigurace ani neobsahuje, tudíž nelze získat žádné informace. Nástroj JRAPIDoc oproti ostatním při spuštění zobrazí očekávanou nápovědu konfigurace.

Při srovnání samotné konfigurace se jako nejobtížnější jeví ta ve Swaggeru. Miredot a JRAPIDoc mají konfiguraci stručnou a snadno pochopitelnou. Swagger umožňuje integrovat dva různé GUI klienty, kde každý je v pom.xml konfigurován pomocí jiných elementů. Tato skutečnost působí při prvním kontaktu s nástrojem velmi zmatečně, a to z toho důvodu, že uživatel například neví o existenci druhého GUI klienta a pokud konfiguruje jiného, než si stáhnul, není jasné, proč konfigurace nebyla aplikována.

6.2 Podpora JAX-RS API

Nástroj Swagger po úspěšné konfiguraci a následném pokusu o vygenerování modelu dokumentace nic nevygeneruje. Swagger tvoří model jen u tříd a metod, které jsou anotovány jeho přídavnými anotacemi. U tříd a metod je vždy nutné duplikovat hodnotu z JAX-RS anotace

@Path do anotace Swaggeru. Takové řešení je problematické, pokud není možná úprava zdrojového kódu tříd rozhraní. Tato situace může nastat, pokud jsou třídy do projektu přidány jako knihovny. S touto vlastností Swaggeru je spojena ještě jedna nevýhoda. Tou je nutnost modifikace kódu při změně seznamu dokumentovaných tříd. JRAPIDoc je flexibilnější, po nakonfigurování pluginu automaticky vyhledá třídy rozhraní. Ty jsou vyhledávány v určených Java balíčcích. Může se stát, že v balíčku jsou i třídy, které dokumentovat nechceme. Taková funkčnost je řešena za pomocí konfigurace, ve které se vydefinuje seznam tříd k vynechání. Pokud nastane situace, kdy je nutné v jednom sestavení vygenerovat více než jednu dokumentaci, se Swaggerem se to nepodaří, protože je vždy nutný zásah do zdrojového kódu.

6.2.1 HTTP parametry

Pokud jsou v třídě rozhraní přítomny atributy, které jsou anotovány pomocí *@MatrixParam*, *@CookieParam* a *@FormParam*, Swagger je v dokumentaci nezobrazí. Stejný problém nastane i při použití anotací nad setter metodami atributů. Po spuštění JRAPIDoc jsou v dokumentaci atributy injektované všemi definovanými anotacemi (*@MatrixParam*, *@CookieParam*, *@FormParam*, *@QueryParam*, *@PathParam* a *@HeaderParam*) z JAX-RS. Je také možné injektovat parametry konstruktoru, tuto vlastnost podporuje nástroj JRAPIDoc, Swagger nikoliv.

Výše zmíněnými anotacemi lze injektovat i jiné datové typy než jsou Java primitivní typy a String (viz 2.2.2.3). Runtime provádí převod na tyto typy z datového typu *String*, tudíž v dokumentaci by se měl objevit datový typ *String* a ne ten, který vznikne transformací v runtime. Swagger ovšem do dokumentace vygeneruje datové typy až po převodu ze *String*, což není korektní. JRAPIDoc korektně vygeneruje informaci o datovém typu.

6.2.2 Metody

Ve specifikaci je definováno dynamické rozhodování obsluhy požadavku a slouží pro to takzvané sub-resource lokátory. Podrobnější informace o sub-resource lokátorech jsou popsány v 2.2.2.4. JRAPIDoc takové chování plně podporuje. Ve Swaggeru se situace liší - se sub-resource lokátory je zacházeno, jako by se jednalo o resource metody. Pokud jsou v rozhraní použity sub-resource lokátory, výsledkem Swaggeru je vypsání chyby a není zobrazena dokumentace ani k ostatním metodám a rozhraním.

Specifikace dovoluje užít nad jednou metodou více request method designatorů (anotací *@GET*, *@POST* atd...). V JRAPIDoc se z jedné Java metody vytvoří pro každý její designator jedna metoda v dokumentaci. Vznikne tak vícero totožných metod s rozdílným typem HTTP metody. Při generování pomocí Swagger je vytvořena pouze jedna metoda odpovídající jednomu typu HTTP metody, druhý a další request method designator jsou ignorováni.

V JAX-RS je také možné vydefinovat si vlastní request method designator, který bude reprezentovat nový typ HTTP metody. Tato funkčnost se může využít, pokud ani jedna ze standardních HTTP metod není vyhovující. Na 6.1 je ukázka, jak lze vytvořit vlastní designator, který má úplně stejné zacházení a vlastnosti jako ty zabudované.

Listing 6.1: Vlastní request method designator

```
1 @Target({ElementType.METHOD})
2 @Retention(RetentionPolicy.RUNTIME)
3 @HttpMethod("CUSTOM")
4 public @interface CUSTOM {
5 }
```

Narozdíl od Swagger podporuje JRAPIDoc vlastní request method designatory a zanese vlastní typ HTTP metody korektně do dokumentace. Swagger po vytvoření a načtení dokumentace skončí chybou oznamující, že metoda nemá nastavený typ HTTP metody a k rozhraní nezobrazí žádnou dokumentaci.

6.2.3 Dědičnost tříd vzdáleného rozhraní

JAX-RS specifikuje dědičnost tříd se vzdáleným rozhraním algoritmem popsaným v 2.2.2.5. Swagger tento algoritmus nevyužívá a dědičnost tříd ignoruje. Anotace definované v nadtřídách a rozhraních jsou ignorovány. I přes přidání Swagger anotací do třídy předka a potomka, metoda v dokumentaci zahrnuta není. JRAPIDoc takto definovaný algoritmus pro průchod předky aplikuje při introspekci kódu a výsledkem je model dokumentace, který odpovídá navrženému rozhraní.

6.3 Podpora JAX-WS API

Do JRAPIDoc byla vytvořena podpora pro implicitní i explicitní SEI. Nástroj využívá hodnoty v anotacích z JAX-WS a při jejich absenci se řídí postupem pro nastavení defaultních hodnot (detailní informace o JAX-WS lze nalézt v 2.2.1).

Nástroj negeneruje do dokumentace všechny informace, které definuje JAX-WS. Nejsou například zahrnuty jmenné prostory. Tyto informace by vedly ke znepřehlednění dokumentace. Na tak velký detail slouží dokument WSDL, který je možno generovat při sestavení aplikace a distribuovat s JRAPIDoc dokumentací. Jinou možností jak WSDL získat je přidání ?wsdl na konec URL adresy webové služby. Tato adresa je ve vygenerované dokumentaci obsažena u každé služby, tím pádem by získání WSDL nemělo dělat potíže.

6.3.1 Metody

Tvorba názvů metod je podporována dle postupů specifikace. SOAP-based webové služby neumožňují dynamicky měnit datové typy odpovědí, vše je statické a dá se tak odvodit z introspekce kódu. Není tedy nezbytné používat doplňkové anotace. Jejich použitím lze však zkvalitnit dokumentaci. JRAPIDoc umí korektně zdokumentovat parametry a návratové typy metod s využitím znalostí hodnot elementů z anotací *@WebParam*, *@WebResult* a *@OneWay*. Je také dokumentováno vyhazování výjimek. Na 6.2 je ukázka definice metody.

Listing 6.2: Metoda SOAP-based webové služby s užitím JRAPIDoc anotací

```
@DocDescription("Example operation")
@DocReturns({
  @DocReturn(
    description = "Expected result of operation",
    typeDescription = "Represents flight destination"),
  @DocReturn(
    http = 500,
    description = "When flight ID is not valid",
    type = AirserviceFault.class,
    typeDescription = "Business logic exception")
})
public Destination foo4b(
  @DocIsRequired
  @DocDescription("ID of flight")
  @WebParam(mode = WebParam.Mode.INOUT, header = true)
  Holder<String> flightId) throws AirserviceFault;
```

Při tvorbě SOAP-based webové služby pomocí JAX-WS je možné použít anotaci *@SOAPBinding*, kterou je možné ovlivnit strukturu přenášených zpráv se zachováním nosných dat. Je možné buď definovat pro celou službu, nebo ke konkrétním metodám. JRAPIDoc zanáší tyto informace do dokumentace podle postupů popsaných v 2.2.1.5.

6.4 Transportní typy

Detailní informace o transportních typech lze nalézt v 2.4.3.3. Zde je popsáno porovnání nástrojů při užití různých datových typů pro parametry a návratové hodnoty metod jako na 2.11. Pokud nejsou k metodám a jejím parametrům přidány přídavné anotace Swaggeru ani JRAPIDoc, výsledky se až na metodu *foo7* liší. Swagger bez přídavných anotací nevygeneruje žádné informace o typech zasílaných v odpovědích. U ostatních metod generuje naprosto nerelevantní informace. Konkrétně pro metodu *public <T extends List<TypeA» void foo11(T l)* je jako parametr uvedeno *List{empty}*, zatímco JRAPIDoc uvádí ve své dokumentaci *array<org.example.TypeA>*, což je správně.

JRAPIDoc podporuje různé type providery, což mu dává flexibilitu při dokumentování transportních typů popsaných v 2.4.3.3). Při tvorbě dokumentace se berou v potaz anotace zanesené v třídách z analyzovaných typů. Tyto anotace mohou být z knihovny Jackson a JAXB. Pomocí konfigurace lze nastavit, které anotace aplikovat. Existuje i možnost přidat vlastního type providera, který bude nahrazen za zabudované a tvořit tak transportní typy jiným způsobem. U Swaggeru se nepodařilo zjistit, zda má podobnou funkčnost jako JRAPIDoc, ale defaultně aplikuje anotace jak Jackson, tak i JAXB současně.

6.5 Rozšiřitelnost

Kromě nahrazení type providera za vlastního poskytuje nástroj rozhraní pro tvorbu vlastních operací nad modelem dokumentace. Tyto operace mohou model číst i modifikovat před tím, než bude vygenerován do souboru. Podrobné možnosti o rozšíření lze nalézt v 2.4.3.4.

Pro vytvoření operace stačí implementovat rozhraní *ModelHandler* a zaregistrovat operaci. Zaregistrování je realizováno zanesením konfigurace v pom.xml k pluginu. Na 6.3 je ukázka operace nad modelem, která kontroluje zda všechny metody v rozhraní mají vyplněn popis. Registrace operace je znázorněna na 6.4.

Listing 6.3: Ukázka použití operace nad modelem

```
public class CheckRestMethodDescriptionHandler implements ModelHandler {
  @Override
  public void handleModel(APIModel model) throws HandlerException {
    for (ServiceGroup serviceGroup : model.getServiceGroups().values()) {
      for (Service service : serviceGroup.getServices().values()) {
        for (Method method : service.getMethods().values()) {
          if (StringUtils.isEmpty(method.getDescription())) {
            Logger.warn("Method {0} has not set description",
              method.getPath());
          }
        }
      }
    }
  }
}
```

Listing 6.4: Registrování handleru pro model

```
<modelHandlers>
    <modelHandler>
        org.example.CheckRestMethodDescriptionHandler
    </modelHandler>
</modelHandlers>
```

Kapitola 7

Budoucí práce

Tato kapitola obsahuje návrhy, jak je možné nástroj v budoucnu rozšířit. První možností je přidat volitelně generování modelu do jiných formátů, než je JSON a do jiných specifikací modelu, než používá nástroj defaultně. Například by mohlo jít o podporu serializace stávající struktury modelu do XML formátu či podporu transformace a vygenerování modelu do WADL. Jelikož nástroj poskytuje rozhraní pro přidání rozšiřujících operací, nebyla by nutná změna v kódu nástroje, ale stačilo by pouze implementovat a přidat operaci do konfigurace pluginu.

Následujícím možným rozšířením by mohla být podpora dalších technologií umožňujících vzdálenou komunikaci v Javě EE. Vedle podpory RESTful webových služeb a SOAP-based webových služeb by tak mohla jednou z takových technologií být JMS.

Momentálně nástroj nepodporuje XML jmenné prostory, které se používají v JAX-WS. Do budoucna by mohla přibýt podpora volitelně je do modelu přidat.

Při návrhu nástroje bylo myšleno i na použití mimo ekosystém Maven. Je proto oddělen vlastní kód nástroje od napojení na Maven. Separování těchto dvou částí je řešeno návrhovým vzorem adapter. V budoucnu je možné vytvořit jiný adapter například pro Apache Ant. Další možností spuštění nástroje je volání přímo z vlastního kódu.

Kapitola 8

Závěr

Diplomová práce si kladla za cíl navržení, implementování a otestování nástroje pro tvorbu dokumentace vzdáleného rozhraní postavených na technologiích JAX-WS a JAX-RS. Důležitou prerekvizitou samotné tvorby nástroje bylo zanalyzování principů tvorby vzdáleného rozhraní, především pak na platformě Java EE. Nástroj byl vyvinut jako zásuvný modul do Apache Maven. Pro tvorbu dokumentace vzdáleného rozhraní je vyžadováno napojení nástroje do sestavování aplikace a provedení minimální konfigurace. Pro kvalitnější a podrobnější dokumentaci je možné do kódu se vzdáleným rozhraním doplnit pomocné prvky ve formě Java anotací. Ty byly vytvořeny jako volitelný doplněk k Maven pluginům. V kontrastu popsaném v 6 vyplývá, že nástroj JRAPIDoc dokáže vytvořit dokumentaci ke vzdálenému rozhraní, aniž by se musely provádět změny v kódu. V tomto ohledu tak předčí nástroj Swagger. Ten téměř nepodporuje specifikaci JAX-RS a je nutné návrh rozhraní přizpůsobovat Swaggeru.

Po závěrečné fázi realizace byly na nástroji provedeny integrační a uživatelské testy. Výsledky těchto testů byly zanalyzovány a zakomponovány do nástroje.

Pro spuštění nástroje je vyžadována jen nejnutnější konfigurace, poté je možné okamžitě generovat dokumentaci. Pluginy nástroje lze spouštět samostatně mimo sestavování, jedinou prerekvizitou je vložení tříd s rozhraním a jejich závislostí na classpath. Nástroj není pevně svázán s Maven, protože poskytuje rozhraní umožňující nahradit stávající napojení za alternativní řešení.

Tvorba dokumentace pro RESTful webové služby i SOAP-based webové služby při použití jednoho nástroje má tu výhodu, že dokumentace je sjednocená jak po grafické, tak po obsahové stránce. Nepodařilo se nalézt nástroj generující dokumentaci k oběma typům služeb současně.

Cíl práce se podařilo splnit podle zadání. Nástroj navíc poskytuje rozhraní, které umožňuje do nástroje doplnit vlastní operace nad modelem rozhraní. Dále je možné některé komponenty nahradit za jiné.

Nástroj a jeho zdrojový kód je uvolněn pod open-source svobodnou licencí a bude integrován do projektu jBPM.

Reference

[1] Apache Ant. http://ant.apache.org/, stav z 5. 2. 2015.

[2] Apache License, 2004. http://www.apache.org/licenses/LICENSE-2.0.txt.

[3] CA Technologies — Business Rewritten by Software. http://www.ca.com, stav z 28. 4. 2015.

[4] European Union Public Licence, 2007. http://joinup.ec.europa.eu/system/files/EN/EUPL%20v.1.1%20-%20Licence.pdf.

[5] JSON2HTML | Transform JSON to HTML. http://json2html.com/, stav z 23. 1. 2015.

[6] JUnit - About. http://junit.org/, stav z 4. 12. 2014.

[7] FasterXML/jackson. https://github.com/FasterXML/jackson, stav z 12. 10. 2014.

[8] Java SE - Downloads | Oracle Technology Network | Oracle. http://www.oracle.com/technetwork/java/javase/downloads, stav z 4. 2. 2015.

[9] Maven - Welcome to Apache Maven. http://maven.apache.org/, stav z 5. 2. 2015.

[10] MireDot | REST API Documentation Generator for Java. http://www.miredot.com/, stav z 7. 11. 2014.

[11] *Oracle® GlassFish Server Message Queue Developer's Guide for Java Client*, Working with SOAP Messages. Oracle Corporation, 2011.

[12] Plexus - Plexus. http://plexus.codehaus.org/, stav z 4. 2. 2015.

[13] RESTEasy - JBoss Community. http://resteasy.jboss.org/, stav z 15. 11. 2014.

[14] Red Hat | The world's open source leader. http://www.redhat.com, stav z 26. 4. 2015.

[15] reflections - java runtime metadata analysis. http://code.google.com/p/reflections/, stav z 5. 10. 2014.

[16] Standard Industrial Classification. http://en.wikipedia.org/wiki/Standard_Industrial_Classification, stav z 22. 1. 2015.

[17] SoapUI - The Home of Functional Testing. http://www.soapui.org/, stav z 7. 11. 2014.

[18] Swagger | The World's Most Popular Framework for APIs. `http://swagger.io/`, stav z 5. 11. 2014.

[19] United Nations Standard Products and Services Code. `http://en.wikipedia.org/wiki/UNSPSC`, stav z 22. 1. 2015.

[20] SOAP. `http://en.wikipedia.org/wiki/SOAP`, stav z 20. 1. 2015.

[21] AND, T. O. *Maven: The Definitive Guide.* O'Reilly Media, 1st edition, 2008.

[22] BURKE, B. *RESTful Java with JAX-RS 2.0.* O'Reilly Media, second edition, 2014.

[23] CHINNICI, R. – HADLEY, M. – MORDANI, R. *The Java API for XML-Based Web Services.* Sun Microsystems, 2.0 final release edition.

[24] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[25] FOWLER, M. Inversion of Control Containers and the Dependency Injection pattern, 1 2004.

[26] JIříčEK, T. sarzwest/jrapidoc-example-app, 5 2015.

[27] JIříčEK, T. sarzwest/jrapidoc, 5 2015.

[28] KALIN, M. *Java Web Services: Up and Running.* O'Reilly Media, second edition, 2013.

[29] PERICAS-GEERTSEN, S. – POTOCIAR, M. *JAX-RS: Java^{TM} API for RESTful Web Services.* Oracle Corporation, 2.0 final release edition.

[30] VAJJHALA, S. – FIALLI, J. *The Java^{TM} Architecture for XML Binding (JAXB) 2.0.* Sun Microsystems, 2.0 final release edition, 2006.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

CORBA Common Object Request Broker Architecture

CSS Cascading Style Sheets

DOM Document Object Model

FTP File Transfer Protocol

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IoC Inversion of Control

Java EE Java Enterprise Edition

JMS Java Message Service

JSON JavaScript Object Notation

MOJO Maven Old Java Object

MVC Model-View-Controller

REST Representational State Transfer

RMI Remote Method Invocation

RPC Remote Procedure Call

SCM Source Code Management

SEI Service Endpoint Interface, Service Endpoint Implementation

SMTP Simple Mail Transfer Protocol

SOAP Simple Object Access Protocol

UDDI Universal Description, Discovery and Integration

URI Uniform Resource Identifier

WP Wire Protocol

WSDL Web Service Description Language

WS Web Services

XML Extensible Markup Language

Příloha B

Obsah přiloženého CD

- *src* - zdrojové kódy
- *text* - text práce
- *ukazka* - GUI Klient a ukázka dokumentace vzorové aplikace