České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Kamil Procházka**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Webový framework pro prezentaci dat na klientské straně**

Pokyny pro vypracování:

Analyzujte používané Javascriptové knihovny a frameworky jako je jQuery, AngularJS, Backbone pro použití automatického generování UI pro prezentaci dat aplikace. Navrhněte a implementujte řešení založené na serverové části REST služeb frameworku AspectFaces. Metamodel na serverové straně vhodně rozšiřte, aby odrážel dovednosti moderních prohlížečů z hlediska zobrazení informací, validace nebo rychlosti. Výsledná implementace musí být rozšiřitelná a konfigurovatelná pro další rozšíření jako např. šablonování. Vaši implementaci pokryjte unit testy v některém používaném testovacím frameworku, například Jasmine. Otestujte nasazením v demonstrační aplikaci, na které dále zhodnoťe výhody a možná omezení implementovaného řešení.

Seznam odborné literatury:

Aspect-driven, Data-reflective and Context-aware User Interfaces Design
T Cerny, K Cemus, MJ Donahoo, E Song
Applied Computing Review 13 (4), 53-65

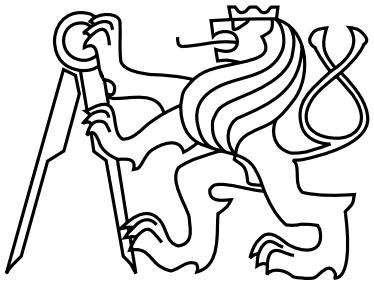
Vedoucí: Ing. Tomáš Černý, MSc.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

V Praze dne 13. 11. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Webový framework pro prezentaci dat na klientské straně

Bc. Kamil Procházka

Vedoucí práce: Ing. Tomáš Černý, MSc.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

11. května 2015

Poděkování

Chtěl bych poděkovat především svým nejbližším, kteří mě během celého mého studia studia podporovali, a bez kterých by nedošlo ani k sepsání této práce. Dále bych chtěl poděkovat svému vedoucímu Ing. Tomáši Černému za jeho ochotu, schovívavost a trpělivost, kterou mi poskytl nejenom během zpracovávání diplomové 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 11. 5. 2015 .....

Abstract

This diploma thesis aims to design and implement extension plugin for AspectFaces library, to be used for generating adaptive, context-aware metamodel, which can be transferred over REST services. Then creating of JavaScript library follows which enables automatic generation of user interface fragments based on the metadata and data available from the server, with a focus on HTML form. At the beginning of the thesis we will discuss techniques of creating user interfaces, compare their advantages and disadvantages. Next section examine existing solutions to generate UI on the client side. The following sections deal with analysis, design and implementation. The last part deals with testing, followed by a summary and discussion of future development possibilities.

Abstrakt

Tato práce si klade za cíl návrh a implementaci rozšíření knihovny AspectFaces, aby byla použitelná pro generování adaptivního, kontextově závislého metamodelu, přenositelného pomocí RESTových služeb. A samotné vytvoření JavaScriptové knihovny, umožňující automatické generování částí uživatelského rozhraní, se zaměřením na formuláře, na základě metadat a dat dostupných ze serverové strany. Na začátku práce probereme techniky tvorby uživatelského rozhraní, porovnáme jejich výhody a nevýhody. Poté prozkoumáme existující řešení pro generování UI na klientské straně. Následující části se zabývají analýzou, návrhem a realizací řešení. Poslední část se zabývá testováním, následovaná shrnutím dosažených výsledků a možným budoucím vývojem.

Obsah

Seznam obrázků

Seznam tabulek

Seznam částí zdrojových kódů

Kapitola 1

Úvod

Webové stránky zaznamenaly od svého počátku dynamický vývoj. Rychlý rozmach internetu posledních dvou desetiletí přinesl vývoj nových zařízení schopných zprostředkovávat interakci s webovými stránkami a rozvoj samotných prohlížečů. Dnes se již nejedná o statické stránky, ale o vysoce interaktivní systémy zpřístupňující uživatelům širokou škálu funkčností zasahující do našeho každodenního života, od objednávání služeb a produktů, po komunikaci, či řízení celé firmy. Zvyšující se nároky na složitost uživatelských rozhraní, variabilita zařízení, dostupnost, či rychlost přenosu dat ke koncovému zařízení vedli k postupnému přechodu od serverem generovaných stránek k hybridním, či čistě prohlížečem generovaným webovým stránkám. Dnes si již svět nedokážeme bez webových stránek představit.

Tato diplomová práce se zabývá vytvořením webového frameworku umožňujícího automatické generování částí uživatelského rozhraní na základě dat a jejich kontextových metadat dostupných ze serverové strany v klientském prohlížeči. Slovo framework je však zavádějící a proto se budeme držet slova knihovna. Metadata obsahují informace o samotných přenášených datech, např. název popisku vstupního políčka pro jméno uživatele, jaký vizuální prvek má být použit, jaké validace se vztahují k políčku, atd. Přenesením těchto a dalších informací, které mohou být specifické pro určité zařízení, či koncového uživatele ze serverové strany na jednotlivé klienty zajistíme snadnější údržbu a rozvoj systémů. Problematika tvorby adaptivních a kontextově závislých uživatelských rozhraní je rozebírána v [14] a [15], kde je představena knihovna AspectFaces, nad kterou bude postavena serverová strana zpřístupňující metadata klientské komponentě starající se o vygenerování UI. Centralizovaná správa struktury a metadat na serverové straně přináší mnoho výhod a vývojáři jsou si jich vědomi a dostává se čím dál více do popředí zájmu.

Práce je členěna do několika kapitol a jejich podčástí. Následující kapitola blíže popíše problematiku a specifikuje jednotlivé dílčí cíle této práce. Třetí kapitola se zabývá obecně problematikou techniky tvorby UI a zasazuje ji do rámce naší práce. Následující čtvrtá kapitola analyzuje dostupné možnosti vhodné k automatickému generování UI na klientské straně a provádí jejich porovnání s přihlédnutím k jejich výhodám a nevýhodám. Pátá kapitola popisuje analýzu a navrh řešení, které navazuje na předchozí zkoumání. Následuje pak samotná realizace navrhovaného řešení, která popisuje charakteristiky a požadavky integrace do stávajících systémů. Sedmá kapitola se zabývá testováním konečného řešení, včetně výkonnostního testu a představení ukázkové aplikace. Poslední kapitola diskutuje budoucí vývoj a shrnuje dosažené poznatky a výsledky práce.

Kapitola 2

Popis problému, specifikace cíle

2.1 Popis problému

Obrovský rozmach a dostupnost webových stránek i samotných zobrazovacích zařízení klade na designery a vývojáře mnoho problémů. Řešení typu jedna velikost padne všem je již dávno pryč. Uživatelské rozhraní již dávno není jen obrazovka počítače. Dnešní svět je plný mobilních zařízení se kterými můžeme přistupovat na webové stránky. Tyto zařízení se od sebe velmi často liší v mnoha ohledech, jak ovládáním, velikostí obrazovky, či rychlostí.

Z pohledu uživatele kvalita uživatelského rozhraní (UI) reprezentuje celkovou kvalitu aplikace a do značné míry ovlivňuje, zda ji uživatelé budou vůbec používat. Uživatelské rozhraní zprostředkovává interakci mezi uživatelem a samotným systémem, jako takové má zasadní vliv na uživatelský prožitek (UX) [1]. Uživatelský prožitek je, jak název napovídá, všechno co uživatel vidí a s čím se potká, když navštíví stránky a chce je použít. Nenáleží sem pouze struktura a obsah stránky, ale také to, jak uživatel stránku najde, zda funguje v jeho prohlížeči, nebo mobilním zařízení, zda fungují standardní funkce na který je zvyklý na svém zařízení, atd. Vše musí fungovat dobře, jinak nebude stránka z uživatelského hlediska úspěšná. Pokud nefunguje, uživatel s největší pravděpodobností navštíví stránku jinou.

Prezentace dat je ovlivněna mnoha aspekty, jak omezeními kladenými samotnými zařízeními, tak koncovými uživateli, kteří zařízení používají, jako např. národnost, geografické umístění, jazyk, či role v jaké vystupují v interakci se samotným systémem. Prezentace dat na webové stránce lze rozlišit na dva pohledy: editační a čistě zobrazovací. Uživatelská rozhraní tudíž představují obousměrný kanál. Data nejsou pouze zobrazována, ale i sbírána a dále zpracovávána. Vstup je zprostředkován webovým formulářem se vstupními políčky představujícími vstupní data. Ta jsou různých typů a v počítači reprezentována různými datovými typy na které je potřeba vstupní textová data z formulářových políček konvertovat. Např. jméno, datum narození, heslo, věk, pohlaví, jsou všechno rozličné typy. Data sbíraná systémem mají učitá konkrétní omezení, jak podle použitých datových typů, tak podle reprezentované vlastnosti v systému. Např. datum narození musí být v minulosti, věk musí být kladné nezáporné číslo, délka hesla musí být minimálně 10 znaků atd. Všechna tato omezení a mnoho jiných musí systém kontrolovat a informovat uživatele o výsledcích kontrol, aby na ně mohl adekvátně reagovat. Tomuto kroku se říká validace a je nutnou navazující součástí

[1] User Experience

zpracování vstupních dat po konverzi, před samotným dalším zpracováním v systému. Z pohledu vývojáře systému se můžeme na uživatelské rozhraní dívat z hlediska reprezentace dat, bezpečnosti, uspořádání komponent a samotném mapování a validace dat na entity, které reprezentují v systému [14]. Některé tyto aspekty se však mohou lišit pro jednotlivá zařízení a uživatele.

Z výše uvedeného výplývá, že abychom zajistili co nejlepší uživatelský prožitek, variabilitu, rychlost a dynamičnost systému je nutné přesouvat čím dál tím více aplikační logiky systému ze serverové na klientskou stranu. Tím jsme schopni okamžitě reagovat na uživatelské akce a generovat odpovědi, bez zbytečně pomalých dotazů na serverovou stranu a překreslení celého rozhraní. Obrácenou stranou mince je duplikovaná logika na klientské a serverové straně. Systém se nemůže spoléhat na přijatá data a musí znovu provést validace a kontroly nutné pro korektní zpracování a uložení dat. Často jsou použity různé programovací jazyky na serverové straně a na klientské straně, zde se nejčastěji jedná o Javascript.

Duplikace logiky není problémem pouze během vývoje, ale hlavně během dalšího rozvoje a údržby, která tvoří většinu života softwarového systému. Po prvotním vývoji přichází často obrovské množství nových požadavků, malých i velkých změn, které musí být v systému realizovány za co nejmenší cenu. Nedílnou součástí těchto zásahů jsou změny databázového modelu, aplikační logiky, validačních pravidel, atd. Všechny tyto změny musí reflektovat UI. Z pohledu vlastníka i vývojáře je proto žádoucí, aby změny byli proveditelné za co nejmenší náklady, ideálně automaticky.

2.2 Cíle práce

Cílem práce je navrhnout a implementovat rozšíření knihovny AspectFaces, aby byla použitelná pro generování adaptivního, kontextově závislého metamodelu, přenositelného pomocí RESTových webových služeb. Inspekce a tvorba metamodelu bude tedy prováděna na serverové straně, který zná objekty se kterými pracuje a klient pouze obdrží jejich definici. Tento přístup umožní klientovi flexibilně reagovat na změny datového formátu, podporované validace, rozložení prvků, lokalizací, možných vstupních hodnot atd. Stávající možnosti metamodelu knihovny budou rozšířeny, aby podporovali moderní webové prohlížeče a standardy.

Druhou částí je realizace klientské části využívající metadata ze serverové strany a poskytující základní stavební kameny pro tvorbu formulářů a vstupních validací, kterým musí komponenta vyhovět, aby bylo možné data odeslat zpět na server. Realizaci předchází rešerše stávajících dostupných technik tvorby uživatelského rozhraní a samotných knihoven k automatickému generování UI v Javascriptu. Zhodnocení jejich výhod a nevýhod s ohledem na použitelnost k řešení našeho problému.

Použití knihovny by mělo být jednoduché a nediktovat architekturu stávajícího systému, jak na klientské, tak serverové straně. Završením práce je netriviální demonstrační aplikace, využívajících implementované serverové a klientské knihovny pro prezentaci dat na které budou zhodnoceny výsledky práce a samotné použití.

2.3 Vymezení práce

Klíčové slovo framework použité v názvu práce je v našem kontextu zavádějící. Cílem není navrhnout plnohodnotný programový model, spíše poskytnout soubor technik a přístupů prostřednictvím samostatných knihoven, které nám umožní dosáhnout lepší znovupoužitelnosti, automatizace a neduplikace informací mezi doménovou a prezentační vrstvou.

Cílem práce také není poskytnout komplexní nástroj pro generování celého uživatelského rozhraní. K tomu samotné doménové objekty, ani jejich metadata nestačí. O těchto technikách pojednává kapitola 3. Záměrem této práce je podpora pro automatické generování pouze částí UI, která lze jednoduše integrovat do stávajícího UI a použitých knihoven. Pro naši práci se zaměříme pouze na formuláře, ovšem metadata přístupná ze serveru jsou obecně využitelná například i k sestavování tabulek, přehledů a mnoha jiných komponent UI.

Kapitola 3

Techniky tvorby UI

Uživatelské rozhraní lze vytvářet manuálně, nebo tento process se můžeme pokusit automatizovat. Automatizace nám pomůže sjednotit vzhled a chování UI, a zároveň sníží náklady na tvorbu UI. Automatizované techniky patří do skupiny automatizovaného programování, které je definováno jako syntéza programu ze specifikace [2]. Pro náš případ z toho plyne, že k vytvoření UI hraje klíčovou roli tato syntéza.

Myšlenka automatického generování uživatelského rozhraní ze specifikace, nebo alespoň jeho částí není nová. První zmínky se objevují již koncem 70 let. Nedlouho poté se začali objevovat první model based systémy [12]. Prvotním cílem bylo zjednodušit a hlavně zrychlit proces vytváření UI. Následně však postupně přicházeli další požadavky na podporu nově vznikajících platforem a zařízení, společně se schopností adaptace na konkrétní kontext [12].

Jednotlivé techniky mohou vytvářet UI staticky, dynamicky, nebo oba přístupy kombinovat. Statické vytváření UI probíhá již ve fázi vývoje, v této fázi však chybí informace o konkrétním kontextu použití, z čehoz plyne nemožnost vytvoření adaptabilního UI, které by se přizpůsobilo na míru aktuálním potřebám uživatele a zařízení. Naopak dynamické vytváření UI probíhá až v runtime fázi systému. Díky tomu mohou být zohledněny samotná runtime data při vytváření UI. Podle [15] můžeme techniky tvorby UI rozdělit na dvě skupiny: *restate-to-extend* a *inspection based*.

Následující podkapitoly se zabývají rozborem jednotlivých technik generování uživatelského rozhraní platných jak pro klientskou, tak serverovou stranu. Hrubě si představíme jednotlivé techniky a zařadíme naši práci s ohledem na serverovou a klientskou stranu. Výsledkem je shrnutí zařazující naši práci do vyjmenovaných technik.

3.1 Restate-to-extend techniky

Restate-to-extend je přístup k tvorbě UI vyžadující duplikaci informací z doménové vrstvy do UI vrstvy a zachování jejich vzájemné integrity. Tento přístup porušuje jeden ze základních principů, a to sice *Don't repeat yourself!* (DRY). Tento přístup značně zvětšuje prostor k zanesení chyb, které jsou často způsobeny nepozorností programátora a jako takový znepříjemňuje správu a budoucí rozvoj. Další nevýhodou je, že není možné jednoduše oddělit znovu se opakující koncerny UI (layout, prezentace, zabezpečení, a další). Následkem je kód,

který není znovupoužitelný, nepřehledný, těžce spravovatelný, vytvořený na jeden konkrétní případ.

Mezi zástupce této techniky pro vytváření UI patří statické generátory kódu, grafické nástroje a doménově specifické modelovací jazyky.

3.1.1 Statické generátory kódu

Statické serverové generátory UI kódu, jako například Spring Roo, nebo Rails static scaffolding, dokáží vytvořit UI velmi rychle s minimálním usilím programátora. Výsledek je však často pouze základní UI (v případě Spring Roo), bez možnosti manuálního zásahu, kterým by se narušilo přegenerovávání. Na klientské straně existuje celá řada automatických generátorů koster celých projektů, uživatelského rozhraní využívající standardní UI komponenty daného HTML&CSS frameworku, avšak tyto scaffolding techniky generují podle statické šablony, která nemá informace o doménových objektech na serverové straně. Jsou však velmi často využívány, protože dokáží zásadně urychlit vývoj webových aplikací vygenerováním koster pro jednotlivé stránek, či poskytují standardní znovupoužitelné knihovny komponent. Nevýhodou je, že v případě pozdějších manuálních změn již nelze tyto části přegenerovat. Mezi tyto generátory patří Twitter Bootstrap[1], ZURB Foundation[2], nebo třeba Polymer[3].

Vytváření UI je plné estetiky a drobných detailů, které nelze zachytit v statickém generátoru. Proto nelze očekávat vygenerování kompletního UI splňujícího veškeré naše představy. Automatizace by měla být používána pouze pro určité části UI a její výsledek by měl vizuálně zapadnout mezi manuálně vytvářené části.

Ideální cestou v případě klientského generování kódu je propojení více technik. Vytvoření hrubé kostry se standardní sadou komponent pomocí dostupného řešení a provedení manuálních úprav této kostry, aby odpovídala našemu záměru. Pomocí dostupných komponent generovat pak jednotlivé části UI již dynamicky. Například z metadat přicházejících ze serverové strany. Tento přístup čím dál tím více nabývá na oblibě mezi vývojáři i businessem, díky ušetřenému času a mnoha vyřešených problémů již samotnou knihovnou.

3.1.2 Grafické nástroje

Grafické nástroje představují techniku tzv. vizuálního programování, kdy vytváření UI probíhá způsobem drag-and-drop. Uživatel vybere požadovaný widget a umístí ho na správné místo v editoru. Výsledný uživatelský vstup je nakonec zpracován a pomocí statického generátoru kódu transformován do výsledného kódu. Tato technika poskytuje uživateli dobrý přehled o vytvářeném UI, jelikož UI v editoru odpovídá konečné podobě vygenerované UI. Toto bývá označováno zkratkou WYSIWYG (What You See Is What You Get). Ovládání těchto nástrojů je velmi snadné a intuitivní. Celkový proces vytváření je však velmi pracný a k dosažení ideálního a konzistentního výsledku je potřeba vynaložit značné úsilí.

Díky značné jednoduchosti této techniky existuje velké množství GUI Builderů pro HTML a existující HTML%CSS frameworky. Níže je krátký seznam:

[1] http://getbootstrap.com/
[2] http://foundation.zurb.com/
[3] https://www.polymer-project.org/0.5/

- **codeCanvas()**[4] - HTML & jQuery UI Builder
- **Glimmer**[5] - HTML & jQuery Design Tool
- **LayoutIt!**[6] - GUI Builder stavějící na Bootstrap komponentách
- **Sencha Architect**[7] - GUI builder pro vývoj platformě nezávislých HTML5 aplikací postavených na frameworku Ext JS

3.1.3 Modelovací jazyky

Většina UI webových aplikací je vytvářena pomocí doménově specifického jazyka. Tento přístup poskytuje největší kontrolu nad vytvářeným UI. Můžeme přesně vyjádřit rozložení prvků (HTML), jejich vzhled (CSS) i validační pravidla a interaktivní chování (JavaScript). Nevýhodou je požadovaná znalost hned několik doménově specifických jazyků a frameworků, kterých je dnešní webový svět nasycen a samotná časová náročnost takovéhoto přístupu. Nevýhody v podobě časové náročnosti tvorby rozhraní lze však velmi zjednodušit používáním statických generátorů a customizací. Poté již stavět na prověřených komponentách některého z frameworků.

3.2 Inspection-based techniky

Inspection-based techniky využívají k tvorbě UI, či jeho částí existující metada z vrstvy doménových objektů. K získání těchto metadat staví na pod-disciplíně reverzního inženýrství nazývaného jako software mining. Konkrétně přístup dynamické analýzy, reprezentováného reflekcí. Řeší tak problém duplikace informací, kterým jsou zatíženy restate-to-extend techniky. I přes široké možnosti reflekce, převážně na serverové straně, nelze všechny informace získat pomocí inspekce kódu. Příkladem nám může být pořadí atributů, datový typ kolekce a mnoho dalších. Z tohoto plyne, že hlavní tíha leží na informačním zdroji, nebo-li doménových entitách, které musí zachytit všechny informace nutné k odvození UI a jednotlivých koncernů UI (layout, prezentace, zabezpečení, a další). Konkrétní způsobů inspekce a reprezentace metadat je závislý na konkrétním použitém programovacím jazyku.

Tento přístup využívá inspekci zaznamenaných metadat o doménových třídách, dynamicky sestavuje ad-hoc strukturální model a provádí transformaci na konkrétní UI. Díky tomu zásadně zjednodušuje vývoj a údržbu, protože informace nutné k vykreslení UI (či jeho částí) je pouze jednou v doménových entitách. Nevýhodou je, že informace nutné k vykreslení UI musí být obsažené v doménových třídách. Zkušenost z praxe ukazuje že standardy, jako Java EE ve svých entitách velké množství v podobě anotací již obsahují, od mapování na databázi, datových typů, po validační pravidla.

Klientská strana v podobě Javascriptu jakožto netypového jazyka však tyto možnosti nemá a v aktuální době neexistuje jednoduchý způsob, jak je zaznamenat bez konkrétního vyjmenování. Inspection-based techniky však obecně nedávají smysl na klientské straně v

[4] http://www.codecanvas.org/
[5] http://visitmix.com/work/glimmer/
[6] http://www.layoutit.com/
[7] http://www.sencha.com/products/architect/

prohlížečí. Ten funguje pouze jako zobrazovací médium pro data, která jsou uložena na serverové straně a jako takový nemá informace o samotné persistenci dat, datových typech, či serverových validacích. Způsobem zpřístupnění výsledků inspekce back-end architektury a jejich mapování na widgety v UI se zabývá podkapitola 3.2.2 Object/User Interface Mapping.

3.2.1 Model Driven Development

Model driven development (MDD) je postaven na myšlence vytvoření pouze modelu aplikace, ze kterého je poté automaticky vygenerována celá aplikace, včetně UI. Model může mít jak grafickou, tak textovou podobu. Model Driven Architecture (MDA) je představitelem využívající grafickou reprezentací modelu pomocí modelovacího jazyka UML[8]. Model může být vyjádřen několika vrstvami abstrakce umožňující popsat i velmi složité závislosti mezi komponentami UI. Na každou vrstvu lze navíc aplikovat mnoho různých typů adaptací.

Myers et al. [8] zmiňuje dva hlavní problémy související s automatickým vytvářením UI na základě modelů. Prvním je, že tato technika neposkytuje dobrý přehled o vytvářeném UI. Druhý problém představuje složitost v podobě nutnosti se naučit složité modelovací jazyky a postupy. Navíc generování kompletního UI se neobejde bez značného omezení flexibility vinou použitých koncových prezentačních knihoven a vzorů integrace.

Přístup lze rozdělit na podskupiny:

- **Static modelling** - statický přístup k vytváření UI.
- **Generative runtime modelling** - dynamický přístup k vytváření UI, umožňující jednoduché adaptace. Provádí se počáteční vygenerování zobrazovacího kódu.
- **Interpreted runtime modelling** - kompletně dynamický přístup vytváření UI neprovádějící generování kódu. Namísto toho interpretuje UI během běhu aplikace pomocí enginu. Umožňuje tak pokročilejší adaptace a provedení změn nevyžaduje opakovanou rekompilaci aplikace. S tím přichází samozřejmě také dopad na celkový výkon aplikace.

3.2.1.1 Externí modely

Speciálním případem MDD je využívání modelu pouze k vytvoření částí aplikace, například pouze UI. V tomto případě se však nelze vyhnout duplikaci informací z vrstvy doménových objektů, proto můžeme tento přístup zařadit mezi restate-to-extend techniky.

3.2.1.2 Domain Driven Design

Domain Driven Design (DDD) spadá také do kategorie MDD technik. Jedná se o zjednodušený případ MDD, protože model je tvořen pouze jedinou vrstvou abstrakce a to vrstvou doménových objektů vyjádřených pomocí objektové jazyka. Obrovskou výhodou tohoto přístupu je možnost soustředit se na danou reálnou problematickou doménu a zjednodušení komunikace mezi vývojáři systému a znalci problematické domény v reálném světě. Technika je hojně využívána v nástrojích pro rychlý vývoj (rapid development) a určuje jednotné názvosloví doménových objektů v kódu, čímž zlepšuje čitelnost a srozumitelnost.

[8] Unified Modeling Language

Jak však zmiňuje Raneburger[13], bez explicitní specifikace informací o rozložení, nebo designu je výsledné UI velmi těžko vyhovující reálným požadavkům. Zčehož lze usuzovat, že tento přístup je použitelný pouze v případě, že si vystačíme s generickým UI. V takovém případě jsme schopni ušetřit velmi mnoho času a prostředků. Technika Rich entity aspect/audit design popisována později se snaží toto omezení eliminovat.

3.2.2 Object/User Interface Mapping

Object/User Interface Mapping (OIM) je technika, která na základě inspekce existujících back-end struktur, zprostředkovávající všechna metadata nutná pro mapování objektů na widgety v UI v programátorem zvoleném frameworku.

Úkolem OIM není vytvořit kompletní UI, ale pouze automatizovat vytvoření těch částí (widgetů), které by za použití restate-to-extend přístupu obsahovaly duplikaci informací z doménové vrstvy, viz. projekt Metawidget [11]. Automatizace procesu vytvářeních těchto částí UI přináší mnoho ušetřeného času a napomáhá předcházet možným chybám, které mohou vzniknout díky duplikaci informací. Důležitým kritériem je, aby technika provádějící automatizaci dokázala vytvářet UI na základě reálných požadavků a byla zachována co největší flexibilita manuálních technik.

OIM je technika založená na podpoře široké škály back-end a front-end technologií a jednoduché rozšiřitelnosti v podobě implementace vlastní podpory dosud nepodporovaných technologií. Tímto přístupem se snaží omezit závislost na konkrétních technologiích a umožnit její využití co největšímu počtu vývojářů. Prakticky obsahuje to nejlepší z obou světů. OIM umožňuje generovat UI jak staticky, tak dynamicky.

3.2.3 Rich entity aspect/audit design

Rich entity aspect/audit design (READ) je inspection-based technika založená na aspektově orientovaném programování (AOP) a detailně popsána v [15]. Samotnou implementací této techniky je již zmiňovaná knihovna AspectFaces [15][1]. V generalized procedure (GP) jazycích jsme omezeni pouze jednou dekompozicí problému a to funkcionální. AOP přináší další prvek - *aspekt*. Aspekt představuje vlastnost, která se může prolínat mnoha komponentami, které nelze zapouzdřit do samostatného celku. V případě UI se dá na aspekty nahlížet jako na rozložení prvků UI, prezentace a pořadí formulářových polí, nebo zabezpečení. AOP umožňuje tyto aspekty izolovat, znovu skládat a opětovně využívat. O propojení aspektů a komponent se v AOP stará *aspect weaver*, během procesu nazývaného také jako *weaving*. Aspekty jsou definovány a udržovány jednotlivě, tím se zlepšuje jejich čitelnost, znovupoužitelnost a celková údržba kódu [15]. Předmětem inspekce jsou doménové objekty rozšířené o další informace, v knihovně AspectFaces v jazyce Java formou anotací, dohromady spolu tvoří *rich entity*. Samotné generování UI může být provedeno staticky, tak dynamicky na základě získaných metadat.

Knihovna AspectFaces je reprezentantem této techniky a je použita v navrhované knihovně pro serverovou stranu řešení, proto podrobnější popis jednotlivých kroků inspekce a generování rozhraní bude rozebrán v pozdějších kapitolách.

3.3 Vyhodnocení technik

Tato podkapitola obsahuje shrnutí technik k tvorbě uživatelského rozhraní a obsahuje shrnutí a doporučení k volbě technik/y vhodných pro náš problém.

3.3.1 Manuální techniky

V manuálních jazycích se nelze vyhnout duplikaci existujících informací z doménové vrstvy. Neumožňují separtovat a aplikovat cross-cutting koncerny UI a tím zlepšit znovupoužitelnost komponent. Problémem jsou také omezené možnosti adaptace UI. Obrovskou výhodou je však velká flexibilita, která často většinu nevýhod v komplikovaných projektech předčí.

3.3.2 Poloautomatizované techniky

Grafické nástroje spadající do této části nevyužívají automatizaci k zefektivnění procesu vytváření UI. Zásadně však zjednodušují a zpřehledňují vytváření UI z pohledu uživatele. Nesou sebou stejné problémy jako manuální techniky a navíc přidávají mnohem menší flexibilitu.

3.3.3 Automatizované restate-to-extend techniky

Do této skupiny zahrnujeme MDD externí modely a statické generátory. Tyto techniky jsou v porovnání s manuálními a poloautomatizovanými technikami efektivnější, ale díky nutné duplikaci informací z doménové vrstvy stále velmi omezené. Mezi velké výhody MDD externích modelů patří možnost aplikovat adaptace na kteroukoliv vrstvu abstracke modelu a také dobrá znovupoužitelnost modelů. Hlavní překážkou je však pomalé generování UI a nutnost naučit se specifický modelovací jazyk. Statické generátory zase nemají přístup k runtime metadatům a navíc znesnadňují následnou údržbu. V případě provedení změn je velmi obtížné provést přegenerování UI při zachováním těchto změn.

3.3.4 Automatizované inspection-based techniky

Automatické inspection-based techniky (DDD, OIM, READ) jsou nejefektivnější, zohledňující širokou škálu vlastností při generování samotného UI. Domain Driven Design neumožňují zachytit tak širokou problematiku, jako je UI v jediné vrstvě a proto umožňuje vytváření pouze generického UI, což pro některé aplikace nemusí být problém. Pokud však jsou tyto vlastnosti vyžadované, je vhodnější použít OIM, nebo READ k runtime generování částí UI, které nejlépe řeší problém duplikace informací nutných k tvorbě UI v doménových objektech.

3.3.5 Výběr přístupu k tvorbě UI

Z předchozího rozdělení vyplývá, že ideálním řešením pro naši práci je použití *Object/User Interface Mapping* techniky, získávající metadata z back-endových doménových objektů pomocí inspekce kódu a jejich zpřístupnění klientské straně, která se postará o samotné vygenerování UI widgetů. Klientská část musí být schopna dodat serverové straně všechny kontextové informace, které jsou nutné pro správnou konstrukci a vyhodnocení metadat, které budou přeneseny zpět na klienta.

Serverová strana inspekce kódu bude realizována pomocí knihovny AspectFaces, která reprezentuje techniku *Rich entity aspect/audit design* a umožňuje nám získat potřebná metadata pro klienta, s přihlédnutím k dostupnému kontextu. Klientská strana bude obsahovat konfigurovatelný a rozšiřitelný runtime generátor komponent generující části UI.

Následující kapitola se zabývá rozborem existujících řešeních vhodných pro generování UI na klientské straně a výběr platformy pro generování klientského UI.

Kapitola 4

Existující řešení pro klientskou stranu

Z předchozí kapitoly vyplývá, že existují dvě techniky generování uživatelského rozhraní: *restate-to-extend* a *inspection-based*. Poslední podkapitola Výběr přístupu k tvorbě UI shrnuje dosavadní poznatky a navrhuje použití techniky *Object/User Interface Mapping*. Samotná inspection-based technika pouze na klientské straně není vhodná z mnoha důvodů. Tím nejdůležitějším z nich je, že klientská strana reprezentuje pouze pohled na data, které jsou ve skutečnosti uložena na serverové straně a jako taková není vlastníkem informací o jednotlivých entitách. Z toho plyne nutnost kombinace více přístupů k dosažení automatického vygenerování UI reflektující změny a dostupná metadata ze serverové straně. V takovém případě back-end poskytuje veškeré informace nutné k vygenerování částí UI. A klientská strana poskytuje serverové potřebný kontext pro jejich vygenerování a stará se o interpretaci a samotné vygenerování UI z metadat a dat s možností konfigurace a zachováním flexibility.

Dominatní roli dostupného programovacího jazyku ve světě prohlížečů převzal JavaScript. Jedná se o multiplatformní, objektově orientovaný skriptovací jazyk, jehož autorem je Brendan Eich z tehdejší společnosti Netscape [16]. Jeho syntaxe patří do rodiny jazyků C/C++/Java, jeho název byl zvolen z marketingových důvodů a nemá, kromě syntaxe mnoho společného s jazykem Java. Javascript byl v červenci 1997 standardizován asociací ECMA[1] a v srpnu 1998 ISO. Standardizovaná verze JavaScriptu je pojmenována jako EcmaScript, v aktuální nejrošířenější verzi EcmaScript 5. Jedná se o interpretovaný slabě typovaný funkcionální jazyk s objektově orientovanými rysy, který se obvykle spouští až po stažení celé WWW stránky. V posledních letech se stále více používá i na jiných místech, například na serverové straně na platformě Node.js[2]. S jeho oblibou je spojena široká škála knihoven a frameworků usnadňující práci vývojářům při tvorbě vysoce interaktivních stránek.

Samotný JavaScript pracuje s reprezentací HTML v podobě Document Object Model (DOM)[3] v prohlížeči a nemá žádné pokročilejší funkce usnadňující práci s DOMem. Proto vývojáři přišli s celou řadou knihoven a frameworků, které tyto vlastností nabízejí, případně i diktují celkovou architekturu systému. Proto se následující podkapitoly zabývají jednotlivými dostupnými knihovnami vhodnými pro generování částí uživatelského rozhraní na základě schémat, která mohou být jak statického, tak dynamického charakteru dostupná ze serverové

[1] http://www.ecma-international.org/
[2] https://nodejs.org/
[3] http://www.w3.org/DOM/

strany. Tím se vyhneme znovu vynalézání kola a použijeme standardní přístupy používané na klientské straně a zajistíme lepší přijetí komunitou našeho řešení.

4.1 JSON Schema

V následujících kapitolách se bude velice často vyskytovat pojem JSON Schema[3], jehož dokumentace je dostupná na adrese http://json-schema.org/documentation.html. Jedná se o draft specifikace, jejíž cílem je definovat strukturu JSON dat a poskytovat kontrakt k čemu data slouží v dané aplikaci a jak s nimi interagovat. JSON Schema umožňuje definovat strukturu, validace, dokumentaci, navigaci a možnou interakci.

Nejjednoduší ukázka definice je zobrazena v ukázce 4.1. Samotná ukázka je dostatečně samopopisná. Bohužel samotná specifikace je relativně obtížně uchopitelná a jednotlivé nástroje, které ji využívají si vybírají často pouze část specifikace, kterou implementují, velmi často také s vlastními rozšířeními. Samotné schéma bývá dost často velmi „ukecané“. Například v případě hyperlinků, či možných hodnot jednotlivých properties. Asi největší nevýhodou však je, že se nejedná o standardizovaný protokol, ale pouze o draft a jednotlivé knihovny, které nad ním staví ho přesně nedodržují.

```
{
    "title": "Example Schema",
    "type": "object",
    "properties": {
        "firstName": {"type": "string"},
        "lastName": {"type": "string"},
        "age": { "description": "Age in years", "type": "integer", "minimum": 0 }
    },
    "required": [ "firstName", "lastName" ]
}
```

Část zdrojového kódu 4.1: Jednoduché JSON Schema

4.2 jQuery

jQuery je malá svobodná a otevřená JavaScriptová knihovna s širokou podporou prohlížečů, která klade důraz na interakci mezi JavaScriptem a DOMem. Byla vydána Johnem Resigem v lednu 2006 [7]. Dnes je jedna z nejrozšířenějších a nejpoužívanějších knihoven na webových stránkách. Knihovna byla designována aby usnadnila obtížně proveditelné úlohy v samotném JavaScriptu (navigace v DOMu a jeho snadná manipulace, animace, zpracování událostí a vývoj AJAX aplikací). Knihovna je snadno rozšiřitelná formou pluginů, kterých existuje obrovské množství a řeší velké množství nekonzistentního chování mezi prohlížeči, tzv. quirks[4].

JQuery se těší obrovské oblibě, ačkoliv samotná knihovna v sobě nemá přímou podporu pro generování UI, existuje řada pluginů, které se snaží tento problém řešit. Seznam pluginu lze najít na stránkách jQuery plugins.

[4] http://www.quirksmode.org/

4.2.1 Alpaca

Asi nejpokročilejším pluginem pro jQuery je Alpaca[5]. Alpaca poskytuje jednoduchý způsob, jak generovat HTML5 formuláře pomocí definice zapsané v JSON Schema[3]. Pro generování UI fragmentů využívá šablonovací knihovny Handlebars[6] s předpřipravenou sadou šablon používající frameworky jako Twitter Bootstrap, jQuery UI a jiné. Knihovna je na velmi vysoké úrovní s možností konfigurace prakticky všech částí vstupujících do generovaní formulářů, od šablon pro jednotlivé vstupní fieldy, jejich lokalizaci, validace, až po celkový layout formuláře.

K vytvoření jednoduchého formuláře nám stačí přidání HTML *DIV* elementu, kam se výsledný formulář vygeneruje a připojit nutné knihovny dle dokumentace. Jednoduchá úkázka je zobrazena v ukázce 4.2. Proměnná *schema* obsahuje definici dle JSON schéma reprezentující předpis pro jednotlivé datové položky, jejich validace a použití. Proměnná *options* potom umožňuje přepsat přijaté schéma, doplnit validace, či překlady a definovat celkové chování včetně použitého layout formuláře a to i včetně všech ovládacích prvků. Konečně stačí zavolat jQuery *alpaca* plugin, který celý formulář vykreslí do stránky.

```
1  ...
2  <body>
3  <div id="form"></div>
4  <script type="text/javascript">
5      $(document).ready(function () {
6          var data = {"name": "Inigo Montoya", "age": 29, "country": "usa"};
7          var schema = {
8              "type": "object",
9              "properties": {"name": {"type": "string"}, "age": {"type": "number", "minimum":
                   0, "maximum": 50}, "country": {"type": "string", "required": true}}
10         };
11         var options = {
12             "fields": {
13                 "name": {"type": "text", "label": "Name"}, "age": {"type": "number", "label":
                       "Age"}, "country": {"type": "country", "label": "Country"}
14             }, "form": {
15                 "attributes": {"method": "POST", "action": "http://httpbin.org/post", "
                       enctype": "multipart/form-data"},
16                 "buttons": {"submit": {"value": "Submit the Form"}}
17             }
18         };
19         $("#form").alpaca({
20             "data": data, "schema": schema, "options": options, "postRender":
                   postRenderCallback
21         });
22     });
23 </script>
24 </body>
```

Část zdrojového kódu 4.2: Alpaca použití

Alpaca je velice mocný nástroj, mezi jehož hlavní výhody patří vysoká flexibilita a v

[5] http://www.alpacajs.org/
[6] http://handlebarsjs.com/

podstatě automatické vygenerování formuláře, dle deklarativní zápisu, včetně odeslání na nastavenou adresu. Nevýhodou, jako u ostatních řešení používající kompletně deklarativní přístup je nutnost pochopit celou knihovnu a nemožnost míchat deklarativní s programovým vytvářením podčástí formuláře. Dynamické přidávání komponent s ohledem na deklarativní způsob je také velmi obtížné a často vyžaduje překreslení celého formuláře. Další problematickou částí je konverze datových typů během serializace, kterou zatím nemá nejlépe vyřešenu.

4.2.2 Další knihovny

Existuje další řada pluginů, jmenovitě například jsForm[7], jsonform[8], json-schema-form-js[9], který lze integrovat do Backbone frameworku při generování formulářů z JSON Schema. Většina těchto pluginů se však již stará pouze o dílčí části životního cyklu formuláře a neřeší ho celý, tak jako Alpaca.

4.3 Backbone.js

Backbone.js je minimalistický MVP[10] framework, který vznikl v roce 2010 a rychle nabyl velké popularity jako lean alternativa k plnohodnotným MV* frameworkům jako je ExtJS, AngularJS, EmberJS a další. Backbone je malý, rychlý s malým memory footprintem obsahující několik jednoduchých konceptů (Model/Collections, Views, Routes). K dispozici je obsáhlá dokumentace plná příkladů. Autoři knihovny píší, že první co je potřeba se odnaučit je snaha spojovat data s DOMem. Pokud má být klientská aplikace opravdu „rich“, pomůže strukturovanější přístup. S Backbone popisujete svá data pomocí modelů, které mohou být vytvářeny, ověřovány, rušeny či ukládány na server. Když požaduje UI změnu některého atributu, spustí model událost „change“, což mimo jiné způsobí, že všechny pohledy, které zobrazují data daného modelu, budou vědět o změně a mohou se překreslit. Nemusíte psát spojovací kód, který bude sledovat DOM, v něm najde konkrétní element se zadaným *ID* a upraví ručně HTML: když se změní model, pohledy se jednoduše překreslí samy.

Na druhou stranu Backbone neřeší mnoho problému, například šablonování a mnoho dalších. Umožňuje vývojaři, aby si sám zvolil knihovnu, která mu s daným problémem pomůže a zapadá do jeho stávající architektury. Backbone ve skutečnosti obsahuje určitou podporu pro šablonování v podobě knihovny Underscore.js[11], která je jedinou povinnou závislostí frameworku, avšak možnost Underscore pro šablonování jsou často nedostatečné. Backbone postrádá řešení na mnoho problémů, které plnohodnotné MV* frameworky řeší. Nediktuje aplikační architekturu, strukturu layoutu, vnořené modely, data-binding mezi view a modelem, validace, šablonování a několik dalších. Na druhou stranu dává vývojáři flexibilitu vybrat si vhodnou knihovnu, který tento problém řeší a integrovat ji do Backbone. Jako takový je vhodný například jako základ pro vlastní framework, či pokud vyžadujeme velkou

[7] https://github.com/corinis/jsForm
[8] https://github.com/joshfire/jsonform
[9] https://www.npmjs.com/package/json-schema-form-js
[10] Model View Presenter
[11] http://underscorejs.org/

flexibilitu ve výběru knihoven, které chceme na řešení konkrétního problému použít a plně vybavené frameworky jako AngularJS, EmberJS, nebo ExtJS nám nevyhovují.

V předchozí kapitole jsem se zmínil o frameworku json-schema-form-js[12], který lze do Backbone integrovat. Pro Backbone je však dostupný mnohem pokročilejší plugin backbone-forms[13]. Jedná se o velmi flexibilní a přizpůsobitelný formulářový framework starající se o veškeré aspekty životního cyklu formuláře, včetně validací, layoutu a bindingu na Backbone model. Součástí je široká škála předpřipravených šablon pro vstupní políčka.

Jednoduchý layout je zobrazen v ukázce 4.3. Layout je definován jako šablona ve speciálním *typu* script tagu, podle kterého je potom vygenerovaný výsledný formulář, jak ukazuje ukázka 4.4. Vytvoření samotného formuláře, tak jak to ukazuje ukázka 4.4 je velmi intuitivní a snadná. Pro schéma popisující jednotlivé komponenty je použit vlastní formát. Backbone Forms poskytují build-in podporu pro několik view frameworků, v základní stavu staví nad Twitter Bootstrapu.

```
1  <script id="formTemplate" type="text/html">
2      <form>
3          <h1>New User</h1>
4
5          <h2>Main Info</h2>
6          <div data-fields="title,name,birthday"></div>
7
8          <h2>Account Info</h2>
9          <h3>Email</h3>
10         <div data-fields="email"></div>
11
12         <h3>Password</h3>
13         <p>Must be at least 8 characters long</p>
14         <div data-editors="password"></div>
15     </form>
16 </script>
```

Část zdrojového kódu 4.3: Jednoduchý layout pro Backbone Forms

Backbone.js je velice zajímavý a rozšiřitelný framework umožňující ponechat velké množství rozhodnutí na samotných vývojářích. Pomocí pluginů lze snadno chybějící vlastnosti doplnit. Backbone Forms poskytují silnou podporu pro vytváření formulářů, jistou vadou je nepříliš flexibilní možnost přidání nových fieldů do stávajícího schématu, což většinou znamená přegenerování celé definice a formuláře ve stránce. Přílišná variabilita může být problémem pro začínající vývojáře, pro které je výhodnější mít jasně definovaný programovací model a sadu nástrojů, které poskytují plnohodnotné MV* frameworky.

[12] https://www.npmjs.com/package/json-schema-form-js
[13] https://github.com/powmedia/backbone-forms

```
var UserForm = Backbone.Form.extend({
    template: _.template($('#formTemplate').html()),

    schema: {
        title: { type: 'Select', options: ['Mr', 'Mrs', 'Ms'] },
        name: 'Text',
        email: { validators: ['required', 'email'] },
        password: 'Password'
    }
});

var form = new UserForm({
    model: new User()
}).render();

$('body').append(form.el);
```

Část zdrojového kódu 4.4: JavaScript generující Backbone formulář do layout

4.4 AngularJS

AngularJS společně s dalšími frameworky jako EmberJS[14], Sencha Ext JS[15] patří do skupiny frameworků, které dávájí vývojářům kompletní framework se svým programovým modelem, který je nutné dodržovat. Všechny tyto frameworky poskytují více či méně podobné vlastnosti a proto si zde představíme pouze Angular.

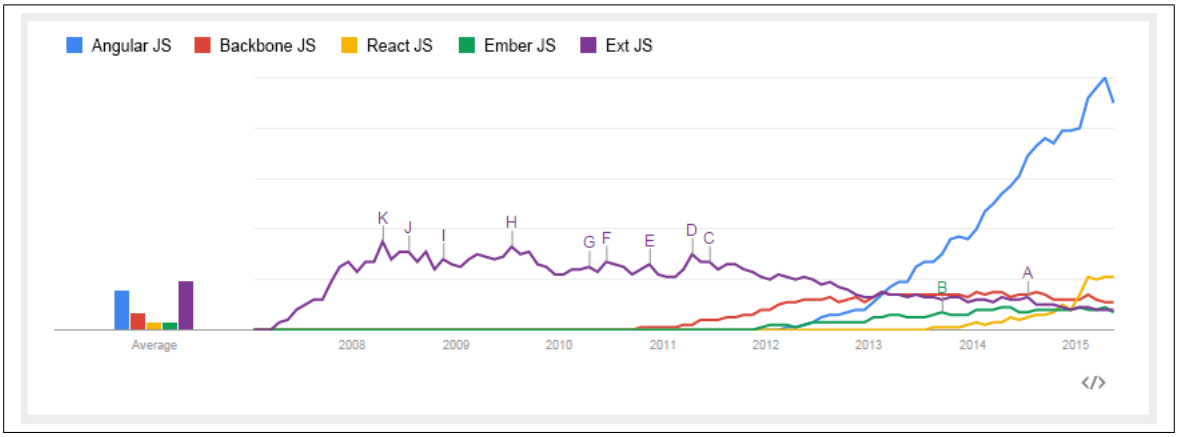
Google Trends[16] jsou dobrým zdrojem pro zjištění popularity jednotlivých porovnávaných knihoven, viz. obrázek 4.1. Můžeme se snadno podívat jak populární je dané téma aktuálně, jak se vyvíjí v čase, případně i na předpověd do budoucna. Samozřejmě jednotlivá klíčová slova mohou vést k jiným výsledkům, avšak do značné míry jsou tyto výsledky reprezentativní pro představu o popularitě.

AngularJS samotný je open source framework vyvíjený a spravovaný Googlem a komunitou a snaží se řešit problémy při tvorbě tzv. single page application[17]. Snaží se zjednodušit vývoj a testování těchto aplikací svou MVC architekturou, společně s komponentami běžně používanými ve webových aplikacích.

Aktuálně se vyvíjí Angular 2, o kterém se toho příliš zatím mnoho neví, kromě toho, že bude prakticky úplně jiný[18]. V této kapitole se tedy budeme zabývat Angularem 1.x.

AngularJS staví na filozofii, že pro sestavování UI, by mělo být využito deklarativního přístupu, k čemuž framework rozšiřuje možnosti tradičního HTML, aby obsahoval sémantické značky představující dynamický obsah, který skrze two-way data binding[19] umožňuje

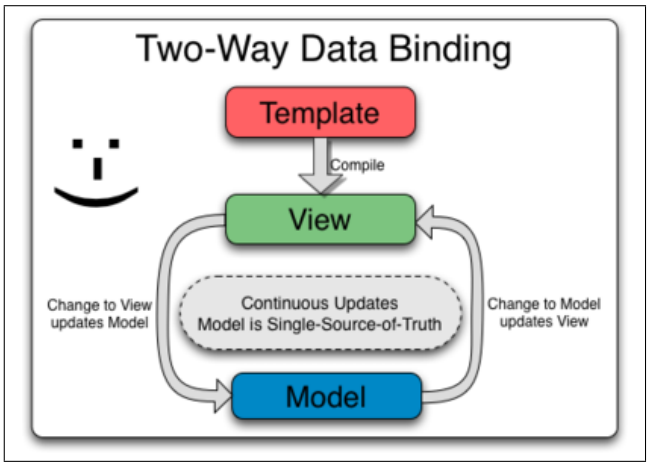
[14] <http://emberjs.com/>
[15] <http://www.sencha.com/products/extjs/>
[16] <http://www.google.com/trends/explore?hl=en-US#q=angular%20js%2C%20backbone%20js%2C%20react%20js%2C%20ember%20js%2C%20ext%20js&date=1%2F2007%20102m&cmpt=q&tz=>
[17] Aplikací, které jsou kompletně vykreslovány na klientské straně na jediné stránce s cíle zajistit co nejplynulejší uživatelský prožitek
[18] <http://jaxenter.com/angular-2-0-112094.html>
[19] Obousměrné propojení modelu a UI



Obrázek 4.1: Google Trends provnávaných knihoven

automatickou synchronizaci modelu a view. Jako výsledek Angular se snaží kompletně elimitovat jakkoukoliv přímou manipulaci DOMu, která znesnadňuje testování a výkon. Angular implementuje MVC umožňující oddělit prezentační část, data a logiku komponent. Používá dependency injection (DI) k pospojování jednotlivých modulů, služeb a komponent ke zvýšení testovatelnosti a přehlednosti kódu.

Největší výhoda a zároveň možná do značné míry nevýhoda je two-way databinding. Ten umožňuje automatickou synchronizaci view dat mezi modelem a zobrazovanými komponentami. Způsob, jakým Angular implementuje data-binding umožňuje se chovat k modelu, jako k jedinému zdroji pravdy v aplikaci. View je pouze projekce dat z modelu v jakémkoliv časovém okamžiku. Nevýhodou two-way data bindingu je jeho výpočetní náročnost, kdy prakticky jakkákoliv operace způsobí kontrolu a synchronizaci view a modelu. Dříve, či později vývojáři narazí na výkonností problémy způsobené právě data bindingem, k jejichž vyřešení je potřeba detailní znalost samotného frameworku i DOMu.



Obrázek 4.2: AngularJS Two-Way Data Binding

Angular samotný obsahuje direktivy a služby dostačující k sestavení HTML formulářů. Jednoduchá ukázka potřebného kódu k sestavení jednoduchého formuláře je zobrazena v

ukázce 4.5. Jak je vidět Angular využívá samotného HTML pro deklarativní definování komponent a propojení s aplikační logikou. Tyto direktivy jsou po načtení stránky prohlížečem interpretovány Angularem a sestaveno view a propojeno s modelem.

```
<div ng-controller="ExampleController">
  <form novalidate class="simple-form">
    Name: <input type="text" ng-model="user.name" /><br />
    E-mail: <input type="email" ng-model="user.email" /><br />
    Gender: <input type="radio" ng-model="user.gender" value="male" />male
    <input type="radio" ng-model="user.gender" value="female" />female<br />
    <input type="button" ng-click="reset()" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>
</div>
```

Část zdrojového kódu 4.5: Standardní AngularJS Form direktivy

Deklarativní přístup k vytváření view má však svá omezení chceme-li se bavit o dynamickém generování UI na základě schématu dostupného ze serveru. S tímto problémem úspěšně bojuje modul angular-dynamic-forms[20]. Tento modul umožňuje podobně jako dříve zmíněné knihovny vygenerovat formulářové vstupní prvky podle definice zapsané v JavaScriptu, či JSON objektu. Schéma používá vlastní proprietární formát. Modul umožňuje deklaraci šablony, která bude použita k rozložení jednotlivých prvků formuláře. Direktivu potřebnou k vytvoření formuláře si můžeme prohlédnou v ukázce 4.6. Zápis je velice snadný a srozumitelný, atributy odkazují na konkrétní šablonu, zobrazovaná data a akci, která má být provedena po stisknutí tlačítka. Ukázka 4.7 ukazuje samotné propojení direktivy s Angularem. Pro aktuální kontroler a jeho $*scope* se nadefinuje pomocná proměnná *formData*, která je nutná pro lokální uchovávání dat z formuláře a *formTemplate* obsahující rozložení, vstupní fieldy, jejich validace, popisky atd. a konečně samotný binding na formData model.

```
<dynamic-form template="formTemplate"
       ng-model="formData"
       ng-submit="processForm()">
</dynamic-form>
```

Část zdrojového kódu 4.6: Angular Dynamic Forms Direktiva

Angular je možné integrovat i s jQuery pluginem JSON Form, zmíněném v předchozím textu, či přímo jeho přeimplentací do čistém Angularu modulem Angular Schema Form[21].

AngularJS i ostatní komplexní MV* frameworky jsou velmi zajímavým řešením, odstiňujícím vývojáře od DOMu a poskytující řešení pro většinu problémů, které by jinak musel řešit manuálně. Poskytují plnohodnotný programový model schopný řešit většinu problémů, včetně generování formulářového UI. Jejich samotná komplexita často přináší problémy novým vývojářům, kteří se nejdříve musí naučit framework správně používat.

[20] https://github.com/danhunsaker/angular-dynamic-forms
[21] https://github.com/Textalk/angular-schema-form

```
// JavaScript needs an object to put our form's models into.
$scope.formData = {};

$scope.formTemplate = [
    {"type": "text", "label": "First Name", "model": "name.first"},
    {"type": "text", "label": "Last Name", "model": "name.last"},
    {"type": "email", "label": "Email Address", "model": "email"},
    {"type": "submit", "model": "submit"},
];

$scope.processForm = function () {
    /* Handle the form submission... */
};
```

Část zdrojového kódu 4.7: Angular Dynamic Forms JS

4.5 ReactJS

React.js[5] je open source JavaScriptová knihovna vyvinutá a udržovaná Facebookem, Instagramem a komunitou jednotlivých vývojářů. React není framework jako AngularJS a jemu podobné popsané v předchozí kapitole. React reprezentuje **View** v aplikaci, velmi rychlé view. Toto není však jediný rozdíl, čím se liší od předchozích knihoven, v zásadě s nemá nic společného s knihovnami diskutovanými v předchozí kapitole v čele s Angularem. Ačkoliv React není framework, existují koncepty, knihovny a principy které nám pomohou vytvořit rychlou, kompaktní, snadno udržovatelnou, rozšiřitelnou a pochopitelnou aplikaci jak na serverové, tak klientské straně.

Základním stavebním kamenem Reactu jsou komponenty. Vlastně celý React je o tvorbě znovupoužitelných komponent. Ty obsahují dobře zapouzdřenou část programu, umožňují snadnou znovupoužitelnost, testování a separation of concerns. React se nesnaží vymyslet nové způsoby šablonování UI, místo toho využívá jazyku, který již máme, JavaScriptu. React díky syntaktickému cukru v podobě JSX, který je Reactem zkompilován do standardního JavaScriptu zpřístupňuje vlastnosti zatím nestandardizované v samotném JavaScriptu a díky tomu zjednodušuje vývoj. V React.js aplikaci bychom měli rozložit naši stránku, jednotlivé stránky i jednotlivé vlastností, které chceme zobrazovat na sadu komponent s jasně definovanými parametry a zodpovědnostmi, které lze snadno znovupoužít na jiných místech a libovolně zanořovat.

4.5.1 Virtual DOM

Abychom zachovali naše data v synchronizaci s DOMem musíme si být vědomi dvou důležitých věcí:

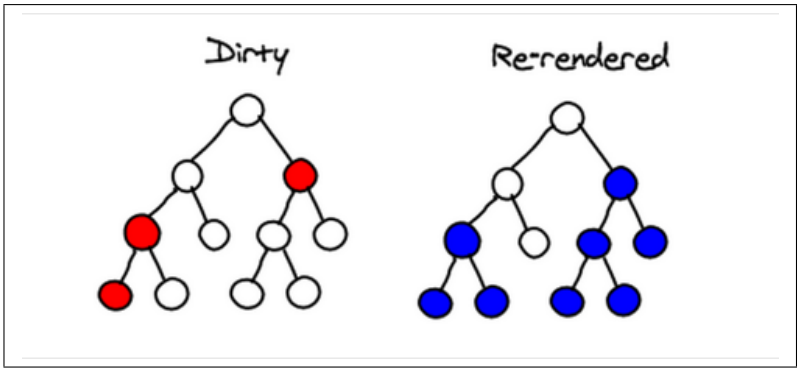
- Kdy se data změnila.
- Které elementy DOMu by měli být updatovány.

K detekci změn React využívá Observer pattern[10], místo dirty checkingu[22]. To je důvod, proč nemusí kontrolovat co se změnilo, ale hned ví co se změnilo a může reagovat. To

[22] Kontinuální kontrola modelu na změnu

umožňuje výrazně redukovat kontrolu dat a celá aplikace je mnohem plynulejší. React implementuje one-way reactive data flow narozdíl od two-way data bindingu implementovaným Angularem. Tím redukuje znovuopakující kód a hlavně umožňuje velice snadno rozhodovat odkud a kam data tečou, což se u tradičního data bindingu často velice špatně odhaluje.

Mnohem více zajímavější věcí je samotná modifikace DOMu v případě změn v modelu. K zajištění překreslení DOMu React postavil vlastní stromovou reprezentaci DOMu v paměti a vypočítává nejmenší možný počet DOM elementů nutných k překreslení v prohlížečí na základě změn v modelu. Jedná se o relativně jednoduchý výpočet. Algoritmus využívá stromové reprezentace DOMu a vyhledává všechny podstromy rodiče, který se změnil (byl nastaven jako dirty) a ty překreslí, jak je znázorněno na obrázku 4.3. Vývojář si musí být vědom změn v modelu, protože ty zapříčiňují překreslení DOMu. Tento algoritmus jde do značné míry ovlivnit, například implementací speciální lifecycle metody komponenty *shouldComponentUpdate()*, která je schopna rozhodnout, zda má být komponenta překreslena. Reálně je algoritmus mnohem složitější, avšak pro jednoduchou představu tento pohled stačí.



Obrázek 4.3: React.js Virtual DOM

4.5.2 Komponentově orientovaný přístup

Jedna z nejhůře uchopitelných věcí pro vývojáře přicházejících z frameworku jako je Angular k Reactu je komponentový návrh. Nelze jednoduše vidět celou stránku v jedné šabloně. Naopak na stránku se díváme jako na sadu samostatných, znovupoužitelných komponent s jasně definovanými zodpovědnostmi. Ty jsou díky tomu mnohem snadněji pochopitelné a udržovatelné, protože se nemusíme zabývat celým systémem, ale pouze danou komponentou.

Pro lepší představu, jak vypadá komponentový návrh obrázek 4.4 zobrazuje komponentu složenou z dalších komponent. Každý odlišný druh komponenty je rozlišen jinou barvou rámečku. Podle tohoto rozdělení se díváme na komponentovou strukturu následovně:

- FilterableProductTable
  - Search Bar
  - ProductTable
    - ProductCategoryRow
    - ProductRow

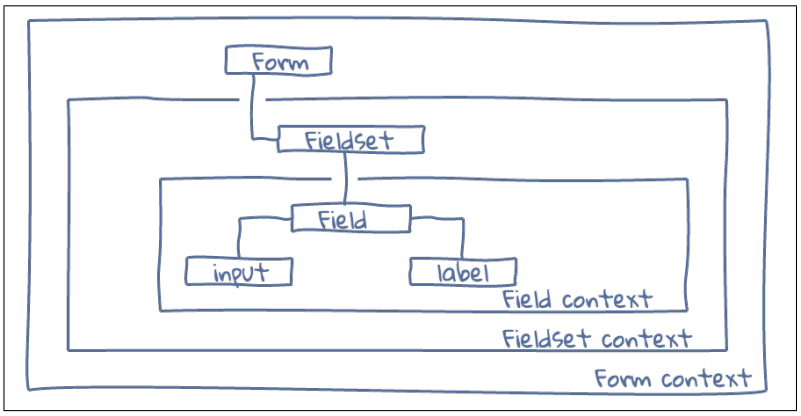
Obrázek 4.4: React.js Komponentová struktura

Thinking in React je oficiální část dokumentace React.js pokrývající problematiku návrhu aplikace komponentovým přístupem. Design jednotlivých komponent by se měl držet Single Responsibility principu, pokud se nám zdá že komponenta dělá příliš mnoho měli bychom ji rozdělit na menší komponenty. Jednoduchá komponenta zapsaná v JSX a ECMAScriptu 6 je zobrazená v ukázce 4.8. Zde je názorně vidět, že komunikace mezi komponentami je ve **POUZE** směru rodič-potomek. Rodičovská komponenta předává všechny parametry prostřednictvím parametrů, zde například komponenta User předává komponentě UserName parameter name a samotná komponenta User přijímá parametr user představující JS proměnou s objektem reprezentujícího uživatele. Obě tyto komponenty jsou beze stavové, komponenty mohou mít svůj interní stav, který používají. Detailnější popis knihovny lze nalézt v oficiální dokumentaci[5], včetně celého životního cyklu a metod komponent.

```
1  class UserName extends React.Component {
2    render() {
3      return <div>name: {this.props.name}</div>;
4    }
5  }
6  class User extends React.Component {
7    render() {
8      return <div>
9          <h1>City: {this.props.user.city}</h1>
10         <UserName name={this.props.user.name} />
11       </div>;
12   }
13 }
14 var user = { name: 'John', city: 'San Francisco' };
15 React.render(<User user={user} />, mountNode);
```

Část zdrojového kódu 4.8: Jednoduchá React.js komponenta

React.js narozdíl od předchozích frameworků neobsahuje přímou podporu tvorby formulářů, ani validací. Existuje však několik rozšíření například wingspan-forms[23], tcomb-form[24], nebo react-forms[25] které usnadňují vytváření a validaci formulářů. Poslední jmenovaný využívá stejně jako dříve představené knihovny vlastního deklarativního schématu k popisu formuláře. Formulář je poté složen z schématem naplněných komponent. Obrázek 4.5 zobrazuje strukturu a vztahy jednotlivých komponent tvořící výsledný formulář.



Obrázek 4.5: React Forms komponenty

Na ukázce 4.9 je jednoduchý příklad schématu nutného pro vygenerování formuláře a jednotlivých prvků. Schéma může obsahovat validace, rozložení komponent a mnoho dalších aspektů. Samotná knihovna se stará o vygenerování UI na základě schématu a dostupných validací a zpřístupňuje zvalidovaná data pomocí callback funkcí.

React.js zažívá velký boom v poslední době, zatím neexistuje tolik standardních přístupů k některým problémům, avšak knihovna je produkčně používána samotným Facebookem, Instagramem, Netflixem, či firmou Sony. Pro generování UI se zdá být jako velmi vhodný kandidát s ohledem na komponentovou architekturu, která do značné míry odpovídá Objektově Orientovanému přístupu a lze mapovat na doménové objekty. Zmíněná knihovna však má jistá omezení. Schéma pro generování musí být ve specifickém formátu. Nelze jednoduše změnit zobrazení jednotlivých *Property*, či přepsat za použití jiného CSS frameworku. Definice rozložení prvků přes schéma je do značné míry komplikovaná.

4.6 Shrnutí

Z předchozího rozdělení plyne, že plnohodnotné MV* frameworky fixují uživatele k použití svého konkrétního programovacího modelu, čímž ho do značné míry fixují k používání pouze daného frameworku. Na druhé straně existuje řada knihoven, které řeší pouze část problematiky UI, například jQuery manipulaci DOMem, React.js na druhé straně představuje pouze View, reprezentované hierarchickou strukturou komponent.

[23] https://github.com/wingspan/wingspan-forms
[24] http://react-components.com/component/tcomb-form
[25] http://prometheusresearch.github.io/react-forms/

```
function Person(props) {
  props = props || {}
  return (
    <Schema name={props.name} label={props.label}>
      <Property name="first" label="First name" />
      <Property name="last" label="Last name" />
    </Schema>
  )
}

var family = (
  <Schema>
    <Person name="mother" label="Mother" />
    <Person name="father" label="Father" />
    <List name="children" label="Children">
      <Person />
    </List>
  </Schema>
);

React.renderComponent(
  <Form schema={family} />,
  document.getElementById('example'));
```

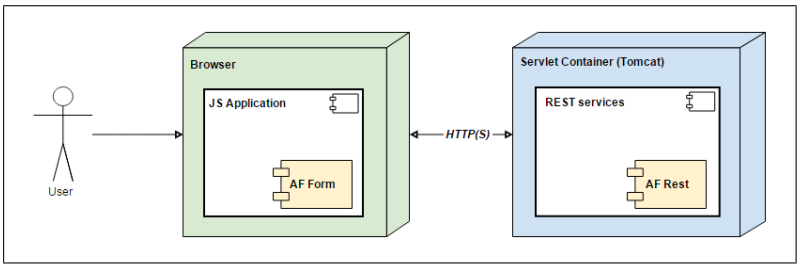
Část zdrojového kódu 4.9: React Forms použití

Soudobé JavaScriptové knihovny z nichž některé byly vyjmenovány v této kapitole umožňují generování formulářů i jiných částí UI na základě deklarativního schématu a to samostatně, či s pomocí rozšíření. Problematické je samotné schéma, neexistuje jednotný formát, jakým popsat jednotlivé prvky a vlastnosti, existuje draft specifikace JSON Schema, který je však relativně složitý a často není přesně dodržován. Většina těchto řešení musí znát samotné schéma hned na počátku generování formuláře. Možnosti dynamické modifikace schématu je často omezená a je nutné přegenerovat celé formuláře znovu. Zachycení celého rozložení formuláře ve schématu do značné míry snižuje flexibilitu. Častá je také fixace konkrétní prezentace formulářových prvků na konkrétní HTML&CSS knihovny, což snižuje jejich použitelnost.

Kapitola 5

Analýza a návrh řešení

Předchozí dvě kapitoly byly důležitým vstupem pro samotnou analýzu a návrh v této práci. Kapitola 3 představila techniky tvorby uživatelského rozhraní, včetně technik automatických a porovnala jejich použitelnost pro náš konkrétní případ a to jak pro klientskou, tak serverovou stranu. Výsledkem porovnání jednotlivých technik jsme dospěli k technice 3.2.2, která umožňuje inspection-based přístup na serverové straně a zpřístupnění metamodelu k sestavení UI klientovi, který se stará o samotnou interpretaci metamodelu do widgetů. Architekturu si tedy lze z pohledu vývojáře, který bude naši knihovnu využívat představit tak, jak ji znázorňuje diagram 5.1.



Obrázek 5.1: Diagram nasazení navrhovaných komponent z pohledu vývojáře

Protože již ze zadání plyne, že pro serverovou inspekci doménových tříd využijeme knihovnu AspectFaces, kapitola 4 analyzovala stávající dostupná řešení pro klientskou stranu vhodná k interpretaci metamodelu. Představila několik nejpoužívanějších JavaScriptových knihoven. Prakticky každá z nich obsahuje podporu pro generování formulářů ze schématu, to většinou bývá proprietárního formátu. Po dohodě s vedoucím práce jsme se rozhodli, že pro implementaci klientské části využijeme knihovnu React.js. Jedním z důvodem, který nás k tomuto rozhodnutí vedl, byla rychle získávaná obliba v komunitě vývojářů, jednosměrný tok dat a komponentový přístup. Proti AngularJS do značné míry hrála také nejistá budoucnost kompatibility s příchodem Angularu 2. To by do značné míry degradovalo naimplementované řešení, jako zastaralé. Navíc protože React.js není framework, ale pouze View technologie, lze ji velice snadno integrovat do stávajících systému, což v případě plnohodnotných frameworků nelze.

Následující podkapitoly jsou rozděleny na serverovou stranu, která nás seznámí se stávajícími možnostmi knihovny AspectFaces a návrhem řešení stavějícím nad touto knihovnou

zpřístupňující nám metamodel pro RESTové webové služby. Druhá kapitola nás zavede na klientskou stranu a představí nám požadavky kladené na klientskou stranu a návrh řešení ve vybrané knihovně React.js.

5.1 Server

Serverová strana je zodpovědná za získání metamodelu a jeho zpřístupnění pro RESTové služby. Neměla by však omezovat vývojáře v samotném výběru REST frameworku, či diktovat architekturu.

5.1.1 Funkční a Nefunkcionální požadavky

Funkční i nefunkční specifikace je do značné míry spojena s použitou knihovnou AspectFaces. Navrhované řešení však musí poskytovat „běhové prostředí“ pro knihovnu AspectFaces nahrazující JSF se kterým je jeho aktuální verze těsně svázaná a závislá.

5.1.1.1 Funkční požadavky

Z předchozích kapitol a dostupných informací byly vyvozeny tyto požadavky:

1. Knihovna bude umožňovat generovat metadata z doménových tříd na základě kterých lze vygenerovat části UI.
2. Knihovna bude generovat metadata s přihlédnutím k aktuálně dostupnému kontextu, tento kontext bude snadno modifikovatelný k ovlivnění samotného generování metadat.
3. Knihovna bude umožňovat přijmout informace s klientské strany, které vloží do kontextu a mohou být následně využity během konstrukce metadat.
4. Knihovna bude umožňovat generovat metadata společně s konkrétními daty, které mohou ovlivnit samotné sestavení metadat.
5. Knihovna bude umožňovat snadnou I18N[1] a lokalizaci textových popisů dat.

5.1.1.2 Nefunkcionální požadavky

Mimo funkčních požadavků, popisujících co systém dělá, aby dosáhl svých cílů je ještě potřeba doplnit druhou skupinu požadavků, takzvané nefunkcionální, někdy také nefunkční požadavky. Ty jsou důležité k dosažení kvalitní a stabilní aplikace, někdy je nazýváme jako Service-level requirements. Popisují co musí systém splňovat z pohledu designu, výkonu, bezpečnosti, a mnoha dalších.

1. Knihovna bude navržena jako rozšiřující plugin pro knihovnu AspectFaces umožňující generovat metamodel.

[1] Internacionalizaci

2. Knihovna bude snadno použitelná a integrovatelná do standardního Java Servlet kontejneru a samotné aplikace, bez architektonických omezení na strukturu projektu, či výběr knihoven.

3. Knihovna zachová p výkon? podobnost s AF ?

5.1.2 AspectFaces

AspectFaces je knihovna implementovaná v jazyce Java, schopná kontextově podmíněné inspekce javovských tříd obsahující metadata v podobě anotací nad jednotlivými atributy. Ty obsahují jednotlivé aspekty (prezentační, validační, bezpečností, ...), které jsou poté transformovatelné do textového výstupu například v podobě XML, či JSF fragmentů. Zpracování a generování výstupu je založeno na textové manipulaci a vkládání jednotlivých aspektů do textu pomocí Unified Expression Language (EL) z inspektovaných tříd. EL umožňuje vkládat jednoduchou podmiňovací logiku do samotné transformace a tím značně ovlivnit výsledek transformace a vylepšit flexibilitu. AspectFaces je možné použít, jak pro statickou, tak dynamickou integraci aspektů, avšak v aktuální době je pouze podporována technologie Java Server Faces (JSF).

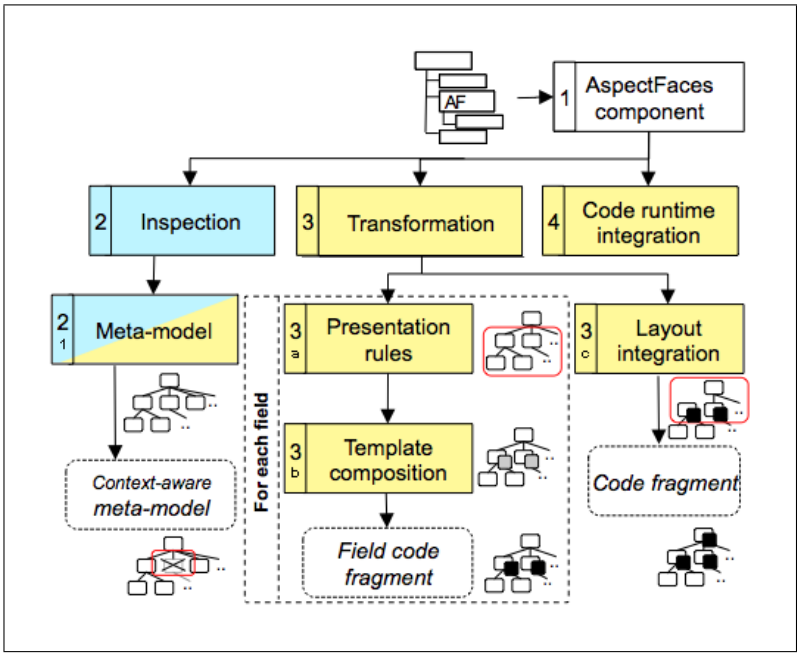
Z výše uvedeného vyplývá, že AspectFaces obsahují přímou integraci pro dynamické generování UI pouze pro framework JSF. Schématicky celý proces zobrazuje obrázek 5.2. Modře označené kroky jsou nezávislé na JSF, zbylé žluté jsou v aktuální verzi specifické pro JSF a nelze je znovupoužít. Tyto žluté části je potřeba nahradit, abychom byli schopni generovat fragmety mimo JSF. Nyní si popíšeme aktuální stav životního cyklu vygenerování JSF fragmentu pomocí AspectFaces a poté ukážeme jakým způsobem je můžeme znovu použít.

První krok znázorňuje definici UI fragmentu, který má být automaticky vygenerován v JSF stránce pomocí značky 5.1.

Druhý krok **Inspekce** má přístup k aplikaci, JSF kontextu a samotné předané instanci. Ta se stává předmětem inspekce, ze které se vytěží všechny metadata (JPA anotace, validace, securita, či vlastní podporované anotace). Z těchto metadat se po vyhodnocení sestaví metamodel, nebo-li struktura reprezentující data. Poté se provede vyhodnocení tohoto metamodelu proti kontextu, který může některé první označit, jako nevalidní, jak je znázorněno na obrázku 5.2. Příkladem takovéto doménové entity nám může být ukázka 5.2.

Dalším krokem jsou prezentační pravidla, krok **3a** na obrázku 5.2. Ten říká, jakým způsobem se budou jednotlivé fieldy mapovat do UI. Jinými slovy, jaký widget má být použit, aby reprezentoval datový field. Pravidlo je mapováno staticky oproti datovému typu, avšak s flexibilní možností definice výrazu v EL, který se na základě kontextu může rozhodnout použít jiné mapování. Také můžeme do XML elementu přidat další atributy, které se budou automaticky propagovat do kontextu při generování samotného fieldu. Toto nám umožňuje použít různé šablony pro jednotlivé typy na základě informace z kontextu, což může být například geo lokace, jazyková mutace, velikost zařízení, atd... Ukázka 5.3 ukazuje takovouto jednoduchou konfiguraci.

Nasledující krok **3b** načte odkazovanou šablonu, kterou si můžeme prohlédnout na obrázku 5.4. Samotná šablona je v Doménově Specifickém Jazyce (DSL) cílového UI frameworku, zde JSF. Samotná šablona může obsahovat dodatečný markup v podobě proměnných mající přístup do kontextu zpracovávaného fieldu a celého kontextu pomocí EL



Obrázek 5.2: AspectFaces: Dynamické generování UI fragmentů

```
1 <af:ui instance="#{controller.personInstance}"/>
```

Část zdrojového kódu 5.1: AspectFaces komponenta zapsaná v JSF Facelet stránce

uzavřeným z obou stran znakem **$**. Ten nám dává značnou flexibilitu v nahrazení generických částí specifickými dle zpracovávaného fieldu. Například jeho název, který se později použije k lokalizaci. Tato šablona je poté v prvním kroku vyhodnocena proti AspectFaces kontextu a poté předána kroku **3c**, který poskytuje dodatečný markup pro vytvoření šablony, kam se budou jednotlivé fragmenty podle názvu fieldu přídávat.

Poslední krok **4. Code runtime integration** z obrázku 5.2 již předá textovou reprezentaci celé AspectFaces komponenty do cílového frameworku. Pro JSF je to definice zapsaná v ukázce 5.1, která je potom zpracována jako standardní stránka Facelet stránka a dynamicky vygenerované widgety jsou zasazeny do stránky.

5.1.3 Návrh řešení

Obrázek 5.2 rozdělil jednotlivé fáze standardního zpracování AspectFaces komponenty. Tento proces se osvědčil při používání knihovny[15]. Žlutě označené části procesu jsou však příliš fixované na JSF. Potřebujeme vytvořit API, které by umožnilo předat k inspekci instanci doménového objektu a na jejím základě by vygenerovalo metamodel vhodný k přenosu na klientskou stranu. Ten musí obsahovat všechny informace nutné k sestavení samotného UI. Velmi zjednodušeně můžeme celý cyklus znázornit tak, jako ho ukazuje obrázek 5.3

Architektura znázorněná na obrázku 5.3 je typická pro mnoho *request-based* knihoven. *AFRestApplication* představuje objekt spravující konfiguraci celé knihovny, která je zkon-

```
1  @Entity
2  @Table(name = "Person")
3  public class Person extends EntityObject implements java.io.Serializable {
4      //..
5      @Column(name = "firstName", nullable = false, length = 100)
6      @NotNull
7      @Length(max = 100)
8      @UiOrder(1)
9      public String getFirstName() { return this.firstName; }
10     //..
11 }
```

Část zdrojového kódu 5.2: Doménová entita s anotacemi

struována při startu aplikace. Při volání aplikace se provede zpracování Servlet Filtru[2], který při každém requestu vytvoří nový request scoped instanci třídy *AFRestContext*. Ta je zodpovědná za udržování všech informací, které se vztahují ke zpracování v tomto jednom requestu, tzn. obsahuje informace, které přijdou v samotném requestu, zpřístupňuje data z servlet requestu, session i celé aplikace. Celý tento proces se děje automaticky pro namapované URL adresy. Až poté je zavolám samotný REST endpoint aplikace. Zde klient může získat *AFEntityBuilder*, jež je zodpovědný za vytvoření metamodelu pomocí *AFRestContextu*, jak se můžeme podívat na ukázce 5.5. Builder klient získá voláním factory metody na AFRestApplication instanci. Samotnému builderu pak již stačí přidat instanci/e, nebo pouze třídu/y, ze kterých má být metamodel vygenerován a voláním *build()* metamodel získat.

5.1.3.1 Metamodel

Doménový model metamodelu vrácený *AFEntityBuilderem* je znázorněn v ukázce 5.4. Obsahuje dva základní elementy: AFEntity a AFField. Oba implementují rozhraní AFResponseComponent, což umožňuje vracet hierarchickou strukturu objektů, což samotná knihovna AspectFaces neumí a ani naše řešení toto přímým způsobem neintegruje. Lze však použít jako jednoduché API a vygenerovat metamodel i vnořených entit a do rodičů snadno vložit a přenést na klientskou stranu. Každý z obou elementů obsahuje minimálně název, který by měl být unikátní.

- **AFEntity** - AFEntity představuje metamodel doménového objektu, může však obsahovat fieldy z několika doménových objektů, pokud si je přejeme přenést jako jednu logickou skupinu. Skládá se tedy z množin AFFieldů, které obsahují konkrétní metamodel fieldů, případně v nich obsažena i přímo data. Dále obsahuje kolekci serverových validačních zpráv, které mohou být odeslány klientovi při odeslání nevalidního požadavku. Kvůli rozšiřitelnosti obsahuje také mapu (klíč-hodnota), která umožňuje přenést na klienta jakákoli nestrukturovaná data.
- **AFField** - AFField představuje meta informace o konkrétním fieldu inspektované třídy. Nese jeho popisek, datový typ, konkrétní hodnotu inspektované instance, název fieldu třídy, pořadí a tag který má být použit na klientské straně a prefix. Ten může

[2] <http://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html>

```
<configuration>
    <mapping>
        <type>Integer</type> <type>int</type> <type>BigDecimal</type> <type>Long</type>
            <type>long</type>
        <default tag="inputNumberTemplate.xhtml" />
    </mapping>
    <mapping>
        <type>Date</type>
        <default tag="inputDateTemplate.xhtml" />
    </mapping>
    <mapping>
        <type>String</type>
        <default tag="inputTextTemplate.xhtml" maxLength="255" size="30" required="false
            " />
        <var name="username" tag="usernameTemplate.xhtml" />
        <condition expression='${not empty type and type == "readOnly"}' tag="
            outputTextTemplate.xhtml" />
        <condition expression="${not empty email and email == true}" tag="
            emailTemplate.xhtml" />
        <condition expression="${not empty password and password == true}" tag="
            passwordTemplate.xhtml" />
        <condition expression="${not empty maxLength and maxLength > 255}" tag="
            textAreaTemplate.xhtml" />
    </mapping>
</configuration>
```

Část zdrojového kódu 5.3: Jednoduchá mapovací pravidla na šablony

posloužit k odlišení více fieldů stejného názvu z více tříd a primárně slouží při samotném zpětném bindingu na doménové objekty. Na AFField jsou navázány tři mapy, jedna obsahuje množinu hodnot, použitelných pro například select, či radio button v HTML, druhá obsahuje mapu validačních pravidel a třetí, stejně jako AFEntity mapu nestrukturovaných dat, dávající vývojáři flexibilitu.

5.1.3.2 Transformace metamodelu do AFEntity

Zatím jsme si knihovnu ukázali pouze zvenku. K tomu, abychom získali AFEntitu, je potřeba transformovat inspekcí získané aspekty z doménových objektů. Na serverové straně však máme velice často příliš málo informací o samotném rozložení, které bude použito a nelze obsáhnout všechny možnosti. I proto je z logických důvodů krok **3c** obrázku 5.2 ponechán jako zodpovědnost klientské strany.

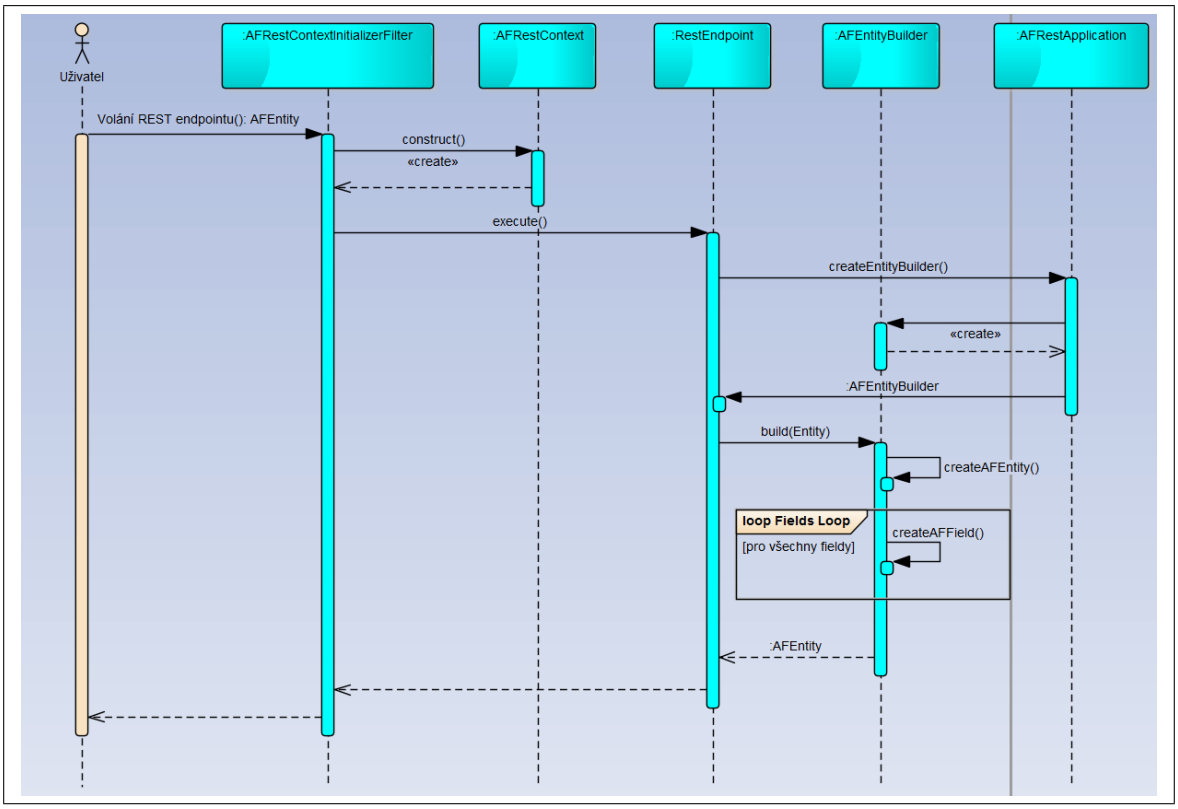
Mapovací pravidla, tak jak je ukazuje ukázka 5.3 je velice mocným a flexibilním nástrojem umožňující zapsat a spravovat mapování jednotlivých fieldů a typů na šablony. Avšak samotné šablony jsou problematickou částí pro svou textovou reprezentaci.

Pokud bychom šablony použili například ke generování XML fragmentů, museli bychom pak fragmenty parsovat zpět do objektové reprezentace. Navíc by tento přístup byl velice neflexibilní, pokud bychom chtěli přidávat neznámá metadata. Částečně by tento problém řešilo použití XSLT transformací. Jejich zápis není zrovna nejjednoduší a zpracování nejrychlejší, stále by pak bylo nutné výsledek transformace zparsovat do objektové reprezentace. Navíc

```
<h:outputLabel
          id="#{prefix}$fieldName$Label"
          for="#{prefix}$fieldName$"
          value="#{text['$className$.$fieldName$']}:" />

<h:inputText
          disabled="#{empty edit$FieldName$ ? not edit : not edit$FieldName$}"
          value="#{$className$.$fieldName$}"
          required="#{empty required$FieldName$ ? '$required$' : required$FieldName$}"
          size="$size$"
          title="#{text['title.$className$.$fieldName$']}"
          maxlength="$maxLength$"
          rendered="#{empty render$FieldName$ ? 'true' : render$FieldName$}"
          id="#{prefix}$fieldName$" />
```

Část zdrojového kódu 5.4: AspectFaces šablona



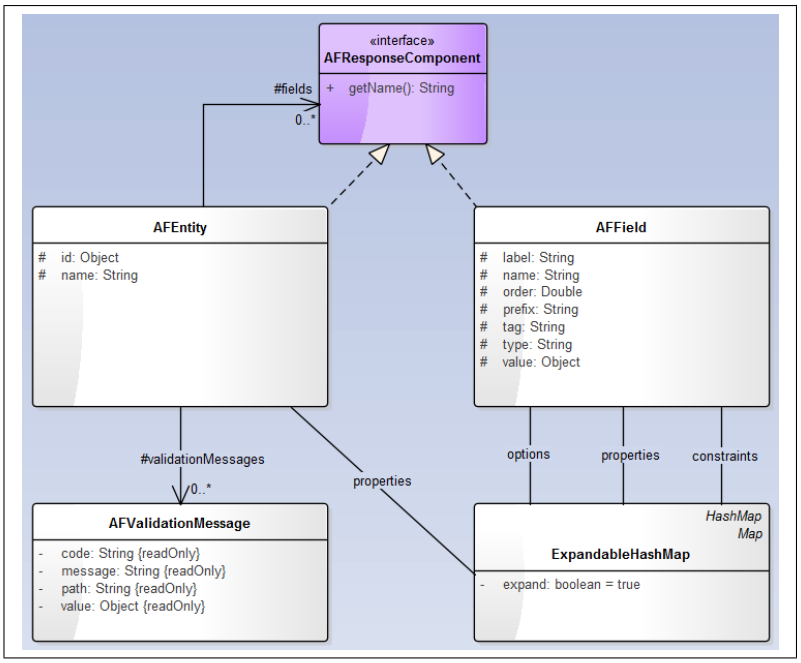
Obrázek 5.3: Schématický sekvenční diagram volání REST služby pro získání metamodelu

obě tyto řešení by znamenali další potřebné knihovny pro zpracování a uzamknuli by nás na použití konkrétního jazyka šablony.

Ještě se nabízela otázka použít šablony přímo jako výstup, který by se vracel REST voláním. Tím by se do značné míry ospravedlnilo použití konkrétní technologie XML, nebo třeba JSON, avšak neměli bychom možnost mimo šablonu jakkoliv manipulovat s jejím obsahem bez parsování. Což by nakonec vedlo k velmi složitému API na straně builderu a

```
1 AFEntityBuilder entityBuilder = restApplication.getEntityBuilder();
2 entityBuilder.id(person.getId()).of(person).of("address", person.getAddress()).collate(
      true);
3 AFEntity entity = entityBuilder.build();
```

Část zdrojového kódu 5.5: Použití AFEntityBuilderu v endpointu



Obrázek 5.4: Analytický model Metamodelu

velké množství manuálního kódu, který by bylo potřeba napsat.

Nakonec se podařilo nalézt řešení zachovávající flexibilitu mapovacích pravidel na šablony. Které pak sami jistým druhem bindingu na představený metamodel a pospojovaný pomocí EL. Vyčerpávající mapování **booleanu** do AFFieldu si můžeme prohlédnout na ukázce 5.6. Jak je vidět z ukázky nápadně tento soubor připomíná standardní javovský *.properties* soubor, obsahující klíč a hodnotu oddělenou rovnítkem. Skutečně se jedná o properties soubor. Levá strana představuje binding kam bude hodnota po vyhodnocení z pravé strany přiřazena. Kontext nám umožňuje přistoupit, jak ke informacím o právě zpracovávaném fieldu, tak k celé aplikaci, dokonce třeba i servisní vrstvě, jak ukazuje mapování *field.options*.

Samotné zpracování je stejně, jako v případě samotného AspectFaces rozděleno na 2 fáze. První krok použije třídu *Context* obsahující pouze meta informace dostupné z inspekce, či z request parametrů, které jsou do tohoto kontextu automaticky přidány. Tento krok předzpracuje jednotlivé řádky z generické formy na konkrétní pro specifické zpracovávané fieldy. Poté přichází na řadu *AFRestContext*, který v porovnání standardního použití AspectFaces v JSF simuluje *FacesContext*. Tento kontext, stejně jako FacesContext má přístup k celé aplikaci, servnisním třídám a co vývojář zná z FacesContextu. Tabulka 5.2 shrnuje defaultní objekty

```
1  # Boolean properties send with AF metamodel. Empty values are ignored and keys are not
       passed !
2  # Special command not to include in render to response
3  rendered=true
4  # Prefix can be used for generating <input name="{prefix}.{name}" /> Useful to
       distinguish between property names from multiple classes.
5  field.prefix=#{prefix}
6  field.label=#{empty '$labelOverride$' ? msg['$ClassName$.$field$'] : msg['
       $labelOverride$']}
7  # By default the value is taken from instance of entity, this is override rule
8  field.value=#{$ClassName$.$field$}
9  # Default value
10 field.name=$field$
11 # Default type is field.getClass().getSimpleName()
12 field.type=$DataType$
13 # Default tag name is name of this .properties file
14 field.tag=checkbox
15 # Default from @UiOrder annotation
16 field.order=#{metaProperty.order}
17 # Constraints
18 field.constraints.required=#{empty afParam.required$FieldName$ ? '$required$' :
       afParam.required$FieldName$}
19 field.constraints.readOnly=$readOnly$
20 # Options
21 field.options=#{facade.listBooleanValues()}
22 # Properties
23 field.properties.custom=Boolean value is great ;) Try 'radio' as a tag
24 field.properties.multiple=false
```

Část zdrojového kódu 5.6: Mapování boolean fieldu

dostupné AFRestContextu. Syntaxe je standardní EL pro *deffered evaluation*[3].

Ukázka 5.6 obsahuje vyčerpávající ukázku použití šablony. Ovšem pravdou je, že až na *field.label*, *field.tag*, *field.options* a oboje *field.properties* které aplikují nějakou logiku lze vše ostatní vynechat, protože se jedná o defaultní chování. Prakticky je možné namapovat všechny proměnné na jednu šablonu a zde použít EL, avšak není to nejlepší způsob udržování kódu.

5.1.3.3 Kontext

Během generování AFEntity ze šablony máme dostupné dva kontexty. Prvním z nich je core AspectFaces *Context* použitý během předzpracování šablony z generického zápisu pro obecný field. Tento kontext má k dispozici standardní množinu proměnných, tak jak ji definuje dokumentace AspectFaces, pro názornost jsou uvedeny v tabulce 5.1. Spousta z nich je pouze syntaktický cukr a lze je získat přímým voláním funkcí v EL na metamodelu získaného inspekcí. Nad rámec defaultních hodnot vývojář může specifikovat jakkékoliv další proměnné, nebo mohou přijít jako součást HTTP requestu. Ty jsou potom automaticky propagovány.

Druhým kontextem, který vstupuje do zpracování je *AFRestContext*. Ten již umožňuje přistoupit předzpracovaným příkazům k celé aplikaci. Jeho možnosti jsou podobné jako Face-

[3] Odložené vyhodnocení, viz. <http://docs.oracle.com/javaee/6/tutorial/doc/bnahr.html>

Proměnná	Hodnota
fieldName	Název aktuálně zpracovávaného fieldu začínající malým písmenem
FieldName	Název aktuálně zpracovávaného fieldu začínající malým písmenem
className	Jednoduchý název třídy do které field patří začínající malým písmenem
ClassName	Jednoduchý název třídy do které field patří začínající velkým písmenem
fullClassName	Celý název třídy do které field patří začínající malým písmenem
FullClassName	Celý název třídy do které field patří začínající velkým písmenem
value	Zkratka pro zápis $Instance$.$fieldName$ představující cestu, která bude přístupná během druhého kroku
dataType	Datový typ fieldu začínající malým písmenem
DataType	Datový typ fieldu začínající velkým písmenem
DataTypeFullClassName	Celý název třídy aktuálně zpracovávaného fieldu
label	Konvertovaný název fieldu, s prvním písmenem velkým, ostatní camel case znaky tvoří oddělovač (mezeru)
instance	Cesta pod kterou je uložena instance třídy, která se zpracovává, která bude přístupná během druhého kroku z AFRestContextu

Tabulka 5.1: AspectFaces core Context: defaultní proměnné

sContext, či JspContext. Tabulka 5.2 shrnuje defaultně dostupné proměnné. Poslední pětice vars, field, metaProperty, index, prefix je dynamická a dostupná pouze během samotného zpracovávání jednotlivých fieldů. Implementací standardního *ELResolveru*[4] lze jednoduchým způsobem rozšířit možnosti kontextu, tak jako jsme viděli v ukázce 5.6 pro *field.options*.

5.1.3.4 Binding a Validace

Data přicházející na server je potřeba na doménové objekty tzv. nabindovat a poté provést validace. Tento proces je nedílnou součástí vývoje webových služeb. Knihovna jako taková pro tento proces nemá přímou podporu. Není to ani její starostí a zbytečně by tím fixovala uživatele knihovna na její proprietární řešení, když je Java EE platforma těchto řešení plná. Pro validaci se standardně používá JSR 303: Bean Validation[9], mimo jiné jeho anotace jsou přímo používány knihovnou AspectFaces při zjišťování validačních pravidel. Použití binding knihovny již závisí na svobodné volbě vývojáře. Častým zástupcem je Spring WebDataBinder, či JAX-RS implementace RESTEasy[6], která také obsahuje podporu pro binding. Alternativním řešením může být použití frameworkově nezávislého řešení Formio[5], které samo

[4] <http://docs.oracle.com/javaee/7/api/javax/el/ELResolver.html>
[5] <http://www.formio.net/>

Proměnná	Hodnota
restContext	AFRestContext
restContextScope	Mapa proměnných uložených v AFRestContextu
afParam	Mapa parametrů dostupných z HTTP request požadavku. Všechny parametry s prefixem *_af_* jsou přidány do této mapy.
application	ServletContext
applicationScope	Mapa atributů uložených v ServletContextu
cookie	Mapa umožňující přístup ke HTTP Cookies
header	Mapa umožňující přístup ke HTTP hlavičce požadavku
headerValues	Mapa umožňující přístup ke HTTP hlavičkám požadavku
initParam	Mapa umožňující přístup ServletContext inicializačním parametrům
param	Mapa umožňující přístup ke HTTP request parametru
paramValues	Mapa umožňující přístup ke HTTP request parametrům pod daným klíčem
request	HttpServletRequest
requestScope	Mapa HttpServletRequest attributů
session	HttpSession
sessionScope	Mapa HttpSession attributů.
response	HttpServletResponse
vars	Mapa pod kterou jsou zpřístupněny všechny proměné obsažené v AspectFaces core Contextu
field	Konstanta pod kterou se během zpracování nachází aktuálně zpracovávaný AFField
metaProperty	Konstanta pod kterou lze nalézt MetaProperty pocházející z inspekce
index	Pořadí aktuálně zpracovávaného fieldu
prefix	Prefix, který má být použit pro tento field, přichází z AFEntityBuilderu

Tabulka 5.2: AspectFaces REST Context: defaultní proměnné

staví nad Spring WebDataBinderem.

Protože během inspekce i transformace mohou být některé fieldy z metamodelu odstraněny, například kvůli bezpečnosti, či pro daný profil, mohou si binding frameworky z metamodelu získat seznam povolených fieldu, které mohou ze vstupních dat použít.

5.1.4 Omezení řešení

Existují určitá omezení samotné knihovny AspectFaces, která nejsou ani součástí tohoto řešení. Seznam níže je popisuje k aktuální verzi knihovny 1.4.0.

- Knihovna sice podporuje inspekci proměnných typu Array, Collection, avšak již nedokáže na jednotlivých položkách kolekce provést automaticky inspekci.
- Knihovna nepodporuje hierarchickou propagaci inspekce v případě atributů odkazujících na složené datové typy.

Výše zmíněná omezení však lze snadno obejít vyvoláním inspekce pro složené typy zvlášť a přidáním do AFEntity programově. Inspekce objektových proměnných třídy může nahradit lepší šablona schopná, která může získat všechny potřebné informace. Příklad obejití tohoto omezení najdeme v kapitole Realizace, konkrétně pro seznam uživatelských rolí.

5.2 Klientská strana

Tato podkapitola se zabývá analýzou a návrhem klientské strany, schopné automaticky generovat formulář podle metamodelu ze serveru v podobě serializované AFEntity v JSON formátu. První část se zabývá požadavky kladenými na toto řešení a následující pak rozborem a návrhem. Samotné řešení bude provedeno ve zvolené technologii React.js.

5.2.1 Funkční a Nefunkcionální požadavky

Funkční a nefunkční specifikace vychází ze serverového metamodelu a ze stanovených cílů této práce. Do jisté míry do ní také zasahuje vybraná knihovna React.js, kterou jsme zvolili pro referenční řešení generování UI.

5.2.1.1 Funkční požadavky

Z předchozích kapitol a dostupných informací byly vyvozeny tyto požadavky:

1. Knihovna bude umožňovat automaticky generovat formulář z metamodelu ze serveru
2. Knihovna bude umožňovat takto vygenerovaný formulář jednoduše naplnit daty, nezávisle na metamodelu
3. Knihovna bude schopna interpretovat základní validace na datech, včetně validací na datové typy (number a date), samotná validační logika bude snadno rozšiřitelná a lokalizovatelná, podporované validace obsahují:
   - min, max - povolená velikost číslice
   - minLength, maxLength - povolená délka vstupního řetězce
   - required - povinné pole
   - email - vstupní pole reprezentuje emailovou adresu
   - regex - povolený vstup podle regulárního výrazu
4. Knihovnna bude umožňovat dynamické vkládání a odebírání vstupních políček
5. Knihovna bude podporovat vykreslování jak v editačním, tak prezentačním módu (inputy jsou disabled)
6. Knihovna bude umožňovat vkládat externě přijaté validační zprávy ze serveru

5.2.1.2 Nefunkcionální požadavky

Mimo funkčních požadavků, popisujících co systém dělá, aby dosáhl svých cílů je ještě potřeba doplnit druhou skupinu požadavků, takzvané nefunkcionální, někdy také nefunkční požadavky. Ty jsou důležité k dosažení kvalitní a stabilní aplikace, někdy je nazýváme jako Service-level requirements. Popisují co musí systém splňovat z pohledu designu, výkonu, bezpečnosti, a mnoha dalších.

1. Knihovna bude implementována v knihovně React.js
2. Knihovna nabídne základní sadu widgetů v některém ze standardních HTML$CSS frameworku, jmenovitě HTML inputy typů:
   - text, number, password, email, hidden, date, datetime-local
   - checkbox (pouze v single módu)
   - radio
   - select (včetně multiple selectu)
3. Knihovna bude snadno rozšiřitelná o nové implementace vstupních prvků, které půjde snadno integrovat do stávajícího API formuláře
4. Knihovna bude umožňovat snadno upravit rozložení prvků v formuláři

5.2.2 Komponentový návrh

React.js[5] je komponentový UI framework. Komponenty mají svůj životní cyklus, případně stav, který mohou udržovat. Kapitola 4 se obecně zabývala možnostmi generování UI a shrnula dosažené poznatky a konkrétní výhody a nevýhody jednotlivých knihoven. Z požadavků výše a z podstaty samotné knihovny React.js je jasné, že naši problematiku budeme reprezentovat pomocí komponent.

Obrázek 5.5 nám ukazuje zjednodušený diagram tříd klientského řešení. Celý návrh je založen na použití tzv. mixinů[6], jež nám umožňují simulovat dědičnost v React komponentách. To nám umožnilo definovat předka všech vstupních inputů a jednotlivé konkrétní implementace se již starají o jejich rozdíly. Jednotlivé konkrétní inputy budou založené na frameworku Twitter Bootstrap, respektive přímo na knihovně komponent pro React.js *react-bootstrap*[7].

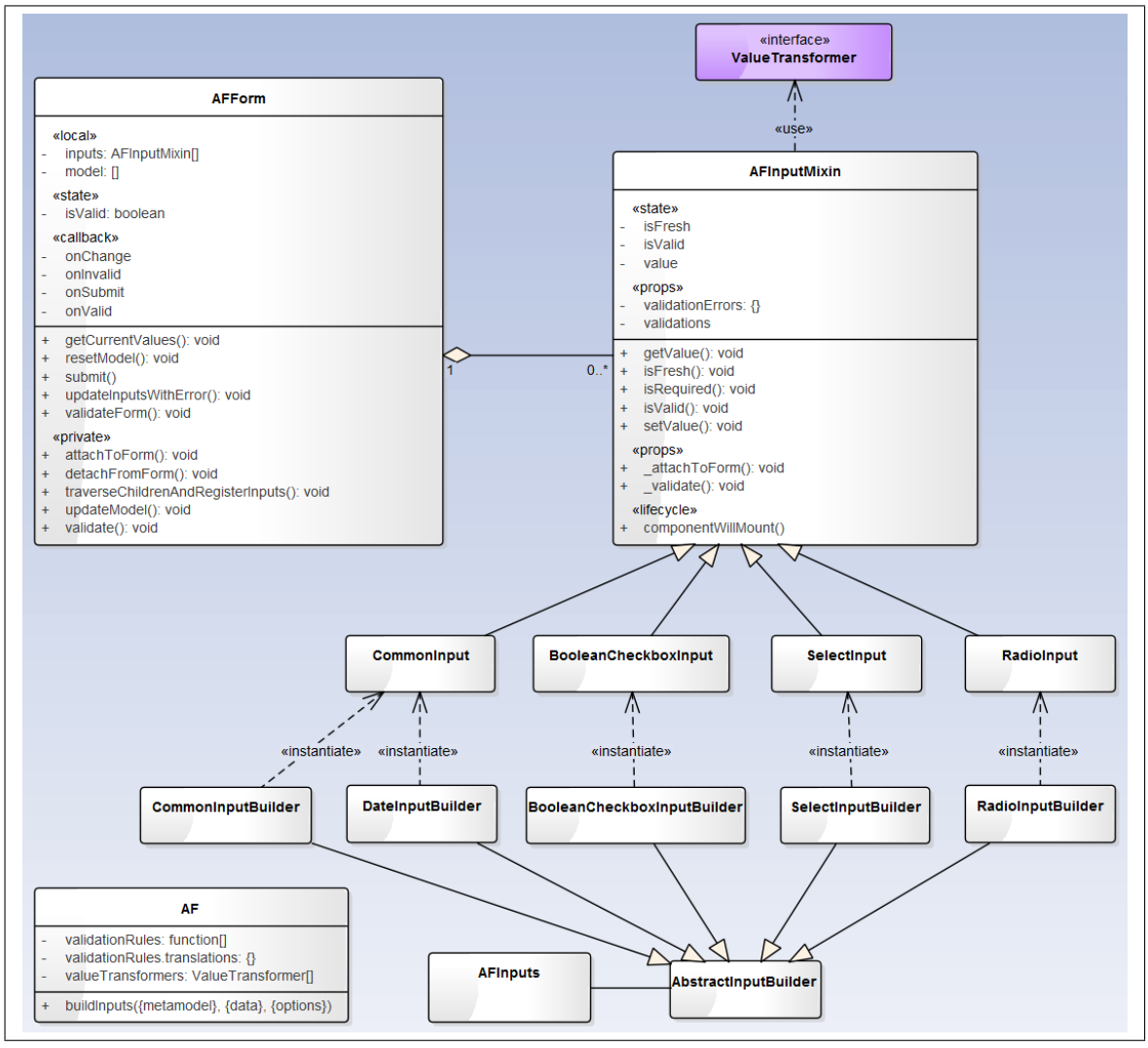
K vygenerování inputů pro formulář z metamodelu použijeme třídy AF, která představuje kořenový vstup do knihovny, přesněji voláním metody *buildInputs()*. Ta přijímá metamodel, data, která má použít jako hodnoty vstupních polí a volitelné options. Knihovna se postará o provolání konkrétních InputBuilderů, kteří již sestaví samotné React componenty.

Atribut **name** formulářových inputů je klíčovým prvkem v celé integraci. Jeho tvar je odvozen z metamodelu spojením volitelného **prefixu** a samotné **nazvu** fieldu z metamodelu. Tím by, pokud serverová strana dodržuje unikátnost názvů měla vzniknout unikátní sada názvů ve tvaru **prefix.name**. Tento tvar nebyl zvolen náhodně. Většina serverový

[6] <https://facebook.github.io/react/docs/reusable-components.html#mixins>
[7] <http://react-bootstrap.github.io/>

bindingovacích knihoven používá právě „tečku“, jako zanořovací znak při procházení objektovou reprezentací. Tímto způsobem lze velice snadno dosáhnout automatického bindingu na serverové straně. V dalším textu se bude všeobecně používat atribut name jako unikátní identifikátor inputů, máme jím na mysli právě tento sdružený formát z prefixu a samotného názvu z metamodelu.

Samotný formulář o jednotlivých inputech nic neví, do té doby, než se mu sami zaregistrují zavoláním callbacku *_attachToForm()*, který formulář jednoduše předá atributem všem potomkům do libovolné hloubky, kteří obsahují atribut **name**. AFInputMixin volá předaný callback *_attachToForm* ve chvíli kdy ho React navěsí na DOMu. Kdykoliv, kdy je formulář překreslen je všem potomkům, kteří mají atribut **name** předán callback, na který AFInputMixin reaguje ve chvíli, kdy je napojen na DOM. Tím je zajištěna automatická registrace nově vzniklých inputů odvozených od AFInputMixin mixinu. Existuje i opačný callback *_detachFromForm()*, který input volá ve chvíli, kdy je odstraňován z DOMu Reactem.



Obrázek 5.5: Zjednodušený diagram tříd klientské knihovny

Samotný AFInputMixin se stará, jak o vykreslení input widgetu, tak získání, transformaci a uložení hodnoty z inputu v DOMu. K tomu mu slouží ValueTransformer, který představuje konvertor mezi datovými typy. V případě změny hodnoty volá callback *_validate()*, který byl stejně jako registrační callback předán jako atribut AFFormem.

Centrální komponentou a mozkem celé správy formulářových prvků je **AFForm**. Stará se o registraci a deregistraci potomků, alias AFInputMixinů a o kompletní životní cyklus formuláře. Při jakkékoliv změně kterékoliv z managovaných inputů je zavolána funkce *validate(component)*, které je předána komponenta, která akci způsobila. AFForm provede validaci této komponenty na základě validačních pravidel, které mu komponenta poskytne proti hodnotě, kterou získá voláním metody *getValue()* na inputu představující aktuální hodnotu inputu. Po dokončení validace komponenty spustí kaskádovitě validaci na všech ostatních managovaných inputech. To se děje proto, že změna hodnoty v jednom inputu ovlivňuje výsledek validace v jiném, například potvrzovací input pro heslo. AFForm automaticky reaguje na akci *onSubmit* a provede validace všech prvků, získá si aktuální hodnoty modelu ze všech inputů a formou callbacku předaného klientem AFFormu umožní reagovat na tuto událost. Ten poté může updatovat model, resetovat ho, či předat serverem vrácené validační chyby pro jednotlivé komponenty. To se děje voláním funkce *updateInputsWithError()*, která přijíma JS objekt, kde klíč objektu reprezentuje name atribut inputu a hodnota je lokalizovaná validační zpráva ze serveru.

AFForm je možné zobrazit ve dvou stavech: editačním a zobrazovacím. Tabulka 5.3 shrnuje možná nastavení konfiguračního objektu *options* předávaného při generování inputů a který zásadní mírou může ovlivnit, zda bude konečný widget zobrazen a jakými vlastnostmi bude naplněna React Bootstrap komponenta widget představující.

Proměnná	**Hodnota**
fieldCallback	Callback, který je zavolán konkrétním AbstractInputBuilderem poté co sestaví seznam atributů pro React Bootstrap komponentu, avšak před tím, než je skutečně předá. Můžeme zde například nastavit vlastní CSS třídu, změnit popisek, či jakkoliv ovlivnit předáváné atributy komponentě.
ignore	Pole názvů inputů, které se mají ignorovat, užitečné pokud nechceme nějaký input podmíněně vykreslit.
readOnly	Boolean hodnota, nebo pole názvů inputů, které mají byt vykresleny jako readOnly (disabled). Boolean hodnota se používá pro vykreslení formuláře v editačním módu.

Tabulka 5.3: Konfigurační proměnné JS knihovny

5.2.3 Omezení knihovny

Stejně jako serverová strana knihovny, i klientská má jistá omezení vyjmenovaná dále.

- Knihovna jako taková se nestará o automatické propojení se serverovou stranou, ale dává uživateli sadu callbacku, kterými sám řídí chování při vyvolání nejrůznějších akcí, sama je však pouze zobrazovací vrstvou dat, které spravuje.

- Knihovna je postavená na vlastnostech HTML5 a knihovně React Bootstrap, některé vlastnosti HTML5 nejsou podporovány ve všech prohlížečích, hlavně se jedná o vstupní políčka typu *number* a *date* s *datetime-local*, které jsou podporované pouze v Chrome[8].
- Knihovna nepodporuje checkbox grupy.
- Knihovna z podstaty svého návrhu není funkční, pokud prohlížeč nepodporuje JavaScript, nebo je vypnutý.

[8] <http://www.google.com/chrome/>

Kapitola 6

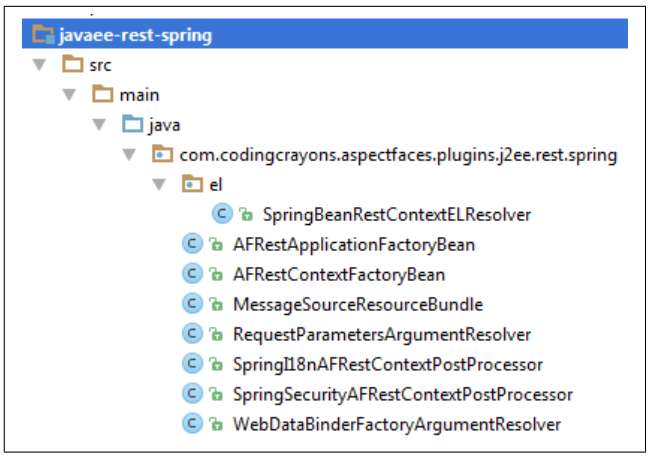
Realizace

Tato kapitola představí nejzajímavější a dosud nepředstavené implementační detaily. Všechny zdrojové kódy jsou vystaveny ve veřejném Git repozitáři.

Ukázkový projekt, ze kterého některé kousky kódu jsou vyňaty, byl realizován jako součást projektu pro představení technologie a testování. Jeho nasazení je blíže popsáno v kapitole Testování. Jedná se o netriviálně rozsáhlý projekt řešící standardní problémy, které se vyskytují v běžném projektu. Samotný projekt obsahuje jak serverou, tak klientskou stranu a využívá vyvinuté knihovny pro automatické generování formulářů a jejich validaci.

6.1 Serverová strana

Samotná implementace serverové strany je rozdělena do dvou pluginů do AspectFaces knihovny: **javaee-rest** a **javaee-rest-spring**. Druhý z dvojice je pouze specifickým rozšířením usnadňujícím použití samotné knihovny v aplikacích založených na Spring Frameworku[1]. Adresářovou strukturu pluginu si můžeme prohlédnout na obrázku 6.1



Obrázek 6.1: Adresářová struktura pluginu javaee-rest-spring

[1] <http://projects.spring.io/spring-framework/>

V rámci návrhu obou maven modulů byly přesunuty závislosti na JSF frameworku z obecného AspectFaces modulu *javaee-connector* do modulu *javaee-jsf2*. Díky tomu bylo možné znovupoužít již naprogramovanou logikou získávání a zpracovávání XML mapovacích souborů a zachování standardní adresářové struktury, na kterou jsou vývojáři zvyklí z používání AspectFaces pro JSF. Oba pluginy jsou narozdíl od stávajících částí knihovny AspectFaces kompilovány Javou 7. Tento posun bude čekat v dohledné době i samotnou knihovnu AspectFaces.

Standardní adresářová struktura při použití Mavenu[4] pak vypadá v případě ukázkového projektu následovně:

```
|-- src/main - kořenový adresář zdrojových kódů Maven projektu
| |-- java - zdrojové kódy v jazyce Java
| |-- resources - nezdrojové soubory přidané do artefaktu během kompilace
| | \-- aspectfaces.properties - základní konfigurační soubor
| |-- webapp - kořenový adresář pro soubory webové aplikace
| | |-- WEB-INF - chráněný, zvenčí neviditelný adresář dle Servlet specifikace
| | | |-- af - adresář, kde jsou uloženy jednotlivé šablony
| | | | \-- rest.config.xml - konkrétní mapovací XML soubor na šablony
| | | \-- aspectfaces-config.xml - hlavní konfigurační soubor
```

Třída *SpringBeanRestContextELResolver* je implementací EL ELResolveru, který zpřístupňuje Springem managované beany z ApplicationContextu při zpracování mapování fieldů. Za zmínku také stojí *SpringI18nAFRestContextPostProcessor* a *SpringSecurityAFRestContextPostProcessor*, což je implementace SPI rozhraní pro post processing AFRestContextu po jeho vytvoření v Servlet Filtru. První z jmenovaných nastavuje lokalizaci do kontextu a druhý role přihlášeného uživatele ze Spring Security[2] kontextu. Hezkou implementací je *MessageSourceResourceBundle*, která zpřístupňuje lokalizovaný resource bundle prostřednictvím springovského kontextu, ten pak může být použit pro lokalizaci popisků vstupních políček.

Implementace pluginu *javaee-rest-spring* je zobrazena na balíčkovém diagramu 6.2. Balíčky jsou rozděleny do několika logických celků starající se o konkrétní problém.

- **Root** - kořenový balíček obsahuje základní stavební kámen a továrnu pro všechny ostatní třídy, třídu AFRestApplication. Ta definuje SPI, kterým lze jednoduše knihovnu rozšířit o nové schopnosti, jednoduše implementací několika rozhraní.
- **Servlet** - balíček obsahuje integrační třídy knihovny do Servlet kontejneru a třídu *AFRequestParameters*, která jak sám název napovídá nese informace odeslané klientem na server. Pro extrakci těchto parametrů zde existuje rozhraní AFRequestParameterFactory se svou defaultní implementací, která všechny request parametry začínající řetězcem **_af_** přidá mezi AF parametry.
- **Context** - balíček obsahuje třídy usnadňující přístup ke kontextovým informacím z mnoha objektů. Centrální třídou celého balíčku a i celé knihovny je *AFRestContext*,

[2] <http://projects.spring.io/spring-security/>

který představuje request scoped context nesoucí a zpřístupňující veškeré informace v ostatních částech systému. Za zmínku stojí třída AFRestContextPostProcessor, která funguje jako SPI, které může vývojář naimplementovat a zaregistrovat do AFRestApplication instance. AFRestContextInitializerFilter se poté postará o zavolání tohoto zaregistrovaného post processoru a můžeme zde libovolně měnit nově vzniklý kontext.

- **EL** - balíček nese implementace EL Resolveru a tříd nutných pro snadnou přenositelnost mezi jednotlivými implementacemi EL. Knihovna není závislá na žádné konkrétní implementaci a lze využít, jak JUEL[3], tak referenční implementaci. Součástí je základní sada ELResolverů, kteří jsou schopny zpřístupnit objekty a jejich obalující mapy dostupné v balíčku *context*.
- **Annotation** - balíčky annotation a annotations obsahují rozšíření stávajícího frameworku AspectFaces o nové anotace. Anotace UiPlaceholder slouží k zaznamenání textu použitelného jako placeholder na klientské straně. Druhé dvě pak během procesingu možných options hodnot fieldu na jejich textovou reprezentaci a budou rozebrány v textu později.
- **Entity** - balíček je složen ze tříd, které tvoří public API metamodelu generovaného knihovnou.
- **Builder** - konečně balíček builder obsahuje třídy vystupující během vytváření AFEntity z inspektovaného metamodelu.

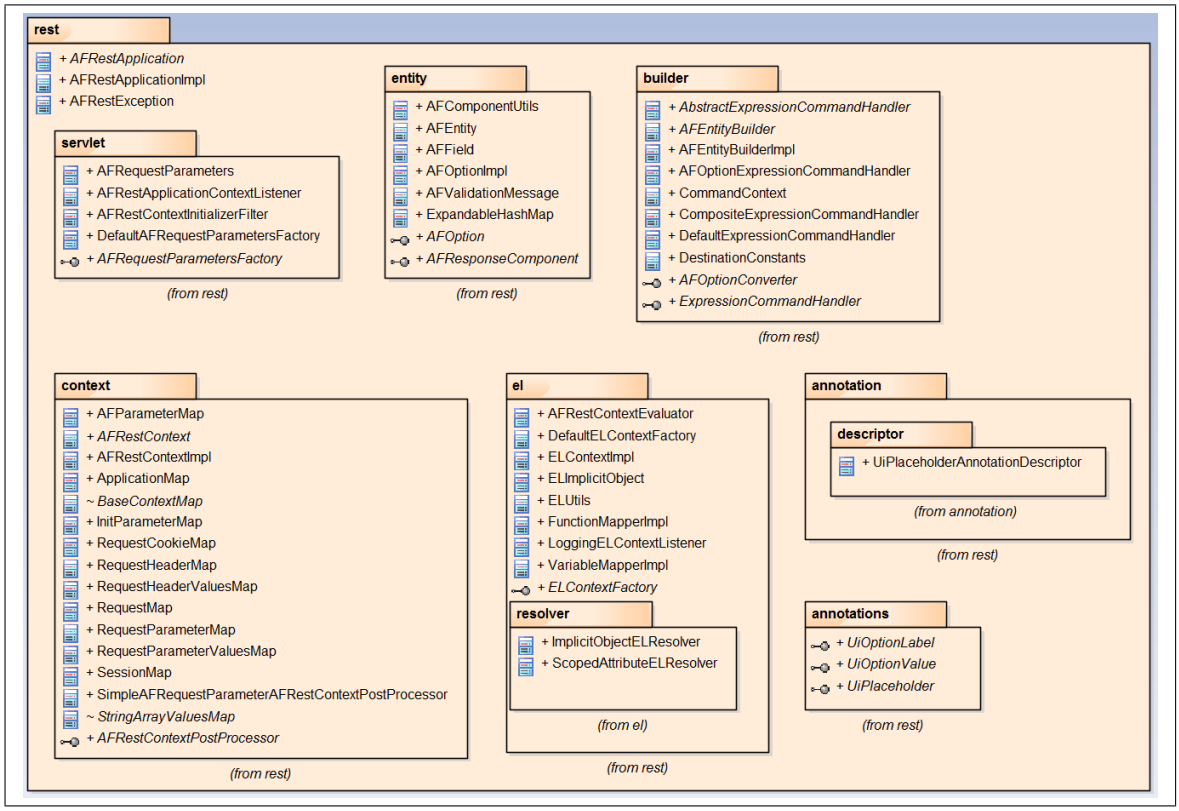
6.1.1 Mapování kolekcí objektů na field.options

Jeden z nejzajímavějších a nejsložitějších problémů je získávání možných hodnot, které se mají zobrazovat například v selectu na klientovi. K reprezentaci možnosti nám na klientské straně v podstatě stačí jedna hodnota, případně dvě pokud se liší popisek a samotná hodnota. V našem modelu v případě třídy AFField je tento problém reprezentován mapou, tzn. klíč:hodnota, neboli popisek:hodnota. Problémem zůstává, že v rámci použití EL, nelze jednoduše vytvořit a přiřadit dvojici do mapy.

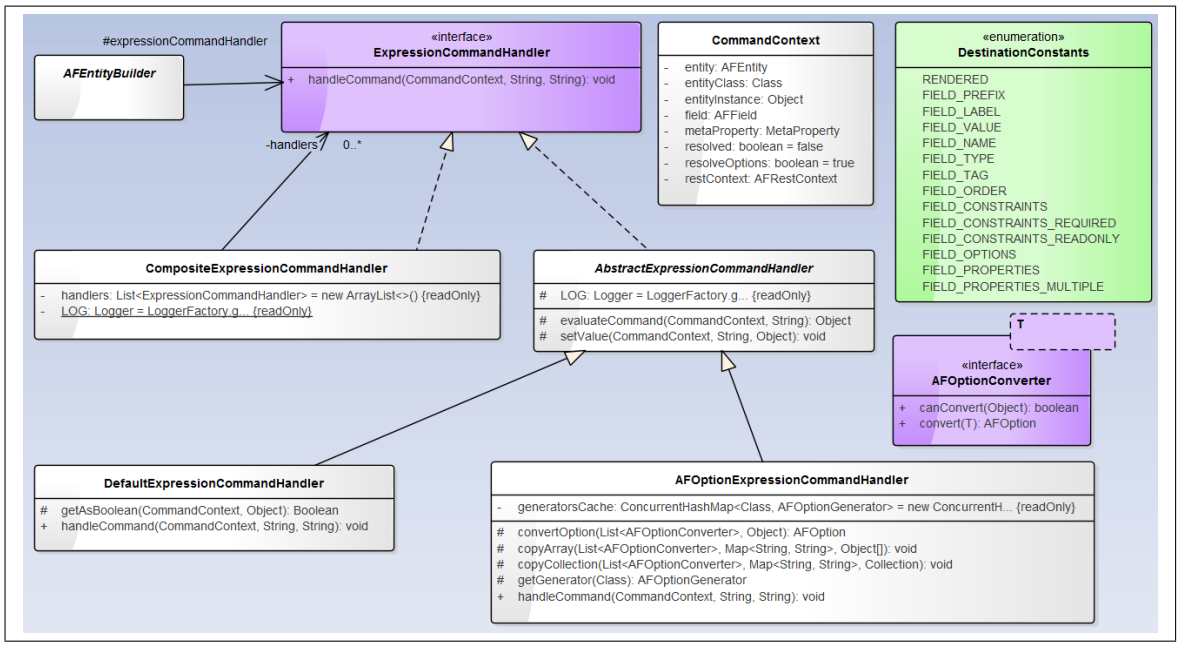
Diagram tříd, jak je zobrazen na obrázku 6.3 hraje ústřední roli během sestavování AFField instance ze šablony. Ta je obohacena o EL expressions, které jsou dynamicky vyhodnoceny proti AFRestContextu a poté přiřazeny jednotlivým atributům AFField instance. Podíváme-li se zpět na ukázku 5.6, můžeme si tuto šablonu znovu prohlédnout. Knihovna se dívá na šablonu jako na seznam po sobě jdoucím příkazů (command). Kde značka = představuje přiřazení dynamicky vyhodnocené hodnoty z pravé strany rovnítka do levé.

Z pohledu diagramu 6.3 knihovna řeší tento úkol vytvořením CommandContextu obsahujícím všechen potřebný kontext nutný pro zpracování ExpressionCommandHandlerem. Ten implementuje návrhový vzor Chain of Responsibility[10]. Obsahuje dvě konkrétní a jednu obalovací implementací, která konstruuje a tvoří vyhodnocovací řetězec[10], tak že obsahuje opět kolekci ExpressionCommandHandlerů, tentokráte již konkrétních. Ty poté sekvenčně volá, dokud některý z nich nenastaví atribut **resolved** v CommandContextu. Třída DefaultExpressionCommandHandler se chová nejjednoduším možným způsobem. Pomocí EL

[3] <http://juel.sourceforge.net/>



Obrázek 6.2: Struktura balíčků pluginu javaee-rest



Obrázek 6.3: ExpressionCommandBuilder class diagram

vyhodnotí pravou stranu příkazu. Výsledek vyhodnocení se pokusí přiřadit do levé strany výrazu. Sám nekontroluje, zda je přiřazení platné a proveditelné.

Druhá implementace AFOptionExpressionCommandHandler je zajímavější. Ta zpracovává pouze ty expressions, jejichž cílem přiřazení je konkrétně **field.options**. V takovém případě se aplikuje a nenechá vyhodnocení na DefaultExpressionCommandHandleru. Handler vyhodnotí pravou stranu a podle vráceného datového typu se rozhodne se s daty udělá.

1. Collection - vráceným typem je třída Collection, nebo její potomek. Potom se pokusí převést každý její prvek na pár klíč hodnota.
2. Pole (Java Array) - vráceným typem je pole, pokračuje stejně jako v případě Collection.
3. Map - vráceným typem je mapa, pak použije přímo tuto mapu, jako možné options pro field a provede přiřazení.

V 1. a 2. případě se Handler pokusí každou položku zvlášť převést na Map.Entry, který by se vrátil klientovi jako reprezentace popisku a hodnoty možné hodnoty, kterou lze vybrat. To se opět děje několika možnostmi podle typu objektu:

1. AFOption - instance je třídy AFOption, pak se z této instance jednoduše vytvoří pár klíč hodnota.
2. Existuje AFOptionGenerator pro danou třídu - AFOptionGenerator je speciální třídou, která cachuje informace a přístup k entitě, která obsahuje anotace *UiOptionLabel* a *UiOptionValue*. Pak tyto hodnoty atributů instance budou použity. Zde se jedná vlastně o dynamické použití metadat k transformaci pro UI.
3. Existuje AFOptionConverter registrovaný v AFRestApplication, či přímo do AFEntityBuilderu, který umí tuto instanci převést. Tato třída představuje jednoduché SPI v případě, že nám jednoduché anotace nestačí a potřebujeme sloučit více atributů dohromady, například jméno a příjmení.
4. Enum - pokud je instance enum, pak se použije jeho metoda *toString()*, která bude pro popisek transformována do malých písmen s počátečním prvním velkým písmenem a hodnota zachována dle volání metody.

Jak je vidět na příkladu AFOptionExpressionCommandHandleru, SPI pro vyhodnocování jednotlivých expressions je velmi silné a lze nad ním stavět velmi zajímavé věci.

6.1.2 Integrace pluginu do aplikace

Integrace pluginu do projektu je snadná a standardní. Ukázka 6.1 zobrazuje úsek souboru web.xml potřebného k integraci pluginu do aplikace, jména balíčků jsou pro kratší zápis zkrácena pouze na první znak. Sestává z listeneru, který vytvoří instanci třídy AFRestContext, která je potom uložena jako atribut ServletContextu. Druhá definice se týká Servlet Filtru zodpovědného za vytvoření AFRestContextu pro každý request, který je namapovaný na daný servlet, v ukázce na Spring Dispatcher Servlet.

```
...
<listener>
    <listener-class>
      c.c.a.p.j.r.s.AFRestApplicationContextListener
    </listener-class>
</listener>
<filter>
    <filter-name>AFRestContextInitializer</filter-name>
    <filter-class>c.c.a.p.j.r.s.AFRestContextInitializerFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>AFRestContextInitializer</filter-name>
    <servlet-name>dispatcher-servlet</servlet-name>
</filter-mapping>
```

Část zdrojového kódu 6.1: Výsek web.xml konfigurace javaee-rest pluginu

Ukázka 6.2 obsahuje kompletní ukázku inicializace aplikace používající javaee-rest plugin s rozšířením rest-spring. Vidíme přidání Spring EL REsolveru, který nám zpřístupňuje Spring beany, listener který je notifikován pokaždé co je vytvořena instance třídy AFRestContext. Nastavení defaultního locale a časové zóny. Přidán jednoduchá implementace AFRestContext post procesoru, který umí vyčíst z AFRequestParametru parametry (config, layout, collate, ignore, profile), které automaticky propaguje do AFRestContextu. Dále pak post processor, který ze Spring Security získá role aktuálně přihlášeného uživatele a nastaví je do kontextu a defaultní resource bundle. Všechno toto se děje pouze jednou po startu aplikace a potom již vše funguje automaticky na pozadí.

Nyní si ukážeme samotné použití knihovny pro získání metamodelu včetně dat na klientskou stranu. Ukázka 6.3 zobrazuje REST endpoint k dynamicky generujícího metamodel třídy Person a jeho vnořené třídy Address. Pro fieldy třídy Address je použit prefix „address“, čímž se zachová unikátnost názvů fieldů, hlavně nám to však pomůže se zpětným bindingem.

Ukázka 6.4 ukazuje částečně zjednodušený příklad bindingu příchozích dat z klienta a jejich validace pomocí standardně nakonfigurovaného Beans Validation frameworku v rámci Springu. O samotný binding @RequestBody mapy obsahující data z těla HTTP požadavku se již stará automaticky Spring WebDateBinder, kterého pouze inicializujeme sadou povolených a zakázaných fieldů, pomocí pomocné třídy BindingHelper. Seznam povolených fieldů získáme z metamodelu třídy Person a Address, které si buď můžeme cachovat, nebo dynamicky znova vypočítat. V případě, že validace objeví nějaké chyby, seznam těchto chyb nejprve zkonvertujeme do seznamu *AFValidationMessage* objektů, vytvoříme přepravku AFEntity, kam je přidáme a vrátíme zpět klientovi. Ač se může zdát samotná implementace složitá, faktem je, že tento kód je pro všechny kontroly již prakticky totožný a nijak se neliší od standardního způsobu zpracování na serverové straně. Obrovská výhoda metamodelu zde přichází v seznamu povolených a zakázaných fieldů vhodných k bindingu. Představme si, že by se útočník snažil si změnit uživatelské role, na což nemá právo. Zapomněli bychom dát kontrolu do správné vrstvy naší abstrakce a otevřeli bychom tak dveře útočníkovi. V našem případě stačí pouze jednou anotací viz. 6.6 na úrovni doménové třídy specifikovat, kdo smí a kdo nesmí tento field vidět. I z toho plyne, že metamodel je velmi závislý na konkrétním kontextu a nelze ho použít pro každého uživatele, respektive zařízení, nebo jejich kombinace.

```
@Component
class RestAppInitializer {
    @Autowired
    private AFRestApplication application;
    @Autowired
    private MessageSource messageSource;
    @PostConstruct
    void setUp() throws Exception {
        if (application == null || application instanceof AFRestApplicationImpl) {
            AFRestApplicationImpl app = (AFRestApplicationImpl) application;
            app.setELContextFactory(new DefaultELContextFactory(new
                SpringBeanRestContextELResolver()));
            app.getELContextListeners().add(new LoggingELContextListener());
            app.setDefaultLocale(new Locale("cs", "CZ"));
            app.setDefaultTimeZone(TimeZone.getTimeZone("Europe/Prague"));
            app.getContextPostProcessors().add(
                new SimpleAFRequestParameterAFRestContextPostProcessor(Collections.<
                    String>singleton("rest"), null)
            );
            app.getContextPostProcessors().add(new
                SpringSecurityAFRestContextPostProcessor());
            ResourceBundle rb = new MessageSourceResourceBundle(messageSource);
            app.getResourceBundles().put("msg", rb);
        } else {
            throw new RuntimeException("AFRestApplicationImpl not present!");
        }
    }
}
```

Část zdrojového kódu 6.2: Inicializace AFRestApplication instance

```
    @RequestMapping(method = RequestMethod.GET, value = "/{id}")
    public HttpEntity get(@PathVariable Long id) {
        Person person = facade.findPerson(id);
        if (person == null) {
            return ResponseEntity.notFound().build();
        }
        AFEntityBuilder entityBuilder = restApplication.getEntityBuilder();
        entityBuilder.id(person.getId()).of(person).of("address", person.getAddress() !=
            null ? person.getAddress() : new Address()).collate(true);
        AFEntity entity = entityBuilder.build();
        return ResponseEntity.ok(entity);
    }
```

Část zdrojového kódu 6.3: Ukázka získání metamodelu třídy s daty

```
@RequestMapping(method = RequestMethod.PUT, value = "/{id}", consumes =
    MediaType.APPLICATION_JSON_VALUE)
public HttpEntity update(@PathVariable Long id, ServletWebRequest request,
    WebDataBinderFactory binderFactory,
                    @RequestBody Map<String, Object> personInput) throws
                        Exception {
    WebDataBinder binder = binderFactory.createBinder(request, person, "");
    BindingHelper bindingHelper = new BindingHelper(binder, messageSource,
        restContext.getLocale());
    bindingHelper
        .allowedFields(AFComponentUtils.getFieldNames(afEntity, false))
        .allowedFields(AFComponentUtils.getFieldNames(address, true))
        .disallowedFields("id").bindAndValidate(personInput);
    if (bindingHelper.hasErrors()) {
        List<AFValidationMessage> validationMessages =
            bindingHelper.getValidationMessages();
        AFEntity entity = new AFEntity();
        entity.setId(id);
        entity.setName(Person.class.getSimpleName());
        entity.setValidationMessages(validationMessages);
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(entity);
    }
    person = facade.saveOrUpdatePerson(person);
    return get(person.getId());
}
```

Část zdrojového kódu 6.4: Ukázka zpracování a validace JSON dat

```
@UiUserRoles("ROLE_ADMIN")
@Enumerated(EnumType.STRING)
public List<UserRole> getUserRoles() { return userRoles; }
```

Část zdrojového kódu 6.5: Ukázka užití role based security v doménovém modelu

6.2 Klient

Samotná interpretace metamodelu na klientské straně se děje tak, jak byla popsána v kapitole Komponentový návrh. Vývojář klientské aplikace získá metamodel ze serverové strany, který poté předá do volání AF.js funkce *buildInputs()*, ta již vrátí kolekci vygenerovaných komponent pomocí knihovny React Bootstrap. Ty poté mohou být jednoduše vnořeny, nebo podle pořadavků vývojáře vloženy, do jeho připraveného layoutu v rámci komponenty AFForm, která se pak již postará o registraci fieldů.

Samotná JavaScriptová knihovna je složená, ze 4 souborů, všechny jsou spuštěny jako lokální funkce a jedinou zvenku dostupnou proměnou je **AF**, která zpřístupňuje jednotlivé public API metody. Rozložení souborů je následující:

- **AF.js** - definuje globální namespace AF, obsahuje sadu Util metod, ValueTransformery pro konverzi mezi různými datovými typy a validační funkce mapující se na klíčová slova jako „maxLength“, včetně překladů. Ty lze snadno globálně změnit.
- **AFForm.js** - AFForm je React komponenta starající se o celý životní cyklus formuláře. Udržuje model, což jsou zkonvertovaná data z inputů a public API pro správu formuláře, nastavení dat, error hlášek, provedení validací atd.
- **AFInputMixin.js** - AFInputMixin je React.js mixin umožňující snadnou implementaci vlastního druhu inputu, který bude automaticky rozpoznán AFFormem během skenování potomků. Obsahuje základní sadu metod nutných pro správné fungování celé knihovny.
- **AFInputs.js** - je sada předimplementovaných input komponent, stavějících na AFInputMixinu a knihovně React Bootstrap. Součástí je sada builderu a defaultních transformátorů metamodelu do seznamu properties předávaných Bootstrap komponentám.

```
1  render: function () {
2      var cfg = {ignore: ["id"]};
3      this.inputs = AF.buildInputs(this.state.metamodel, {}, cfg);
4      var left, right;
5      if (this.inputs && this.inputs.length > 0) {
6          left = this.inputs.splice(0, 10); right = this.inputs;
7      }
8      return (<ReactBootstrap.Panel header="My Profile" bsStyle="info">
9          <AF.AFForm ref="afForm" onSubmit={this.submit}>
10             <div className="row">
11                 <div className="col-lg-6">{left}</div>
12                 <div className="col-lg-6">{right}</div>
13             </div>
14             <ReactBootstrap.Button bsStyle='primary' type="submit" {...disabled}>Save</
                   ReactBootstrap.Button>
15         </AF.AFForm>
16     </ReactBootstrap.Panel>);
17 }
```

Část zdrojového kódu 6.6: Ukázka užití role based security v doménovém modelu

Kapitola 7

Testování

Testování je nedílnou součástí během i po ukončení vývoje softwaru. Mimo ověření správné funkčnosti aplikace se také kontroluje, zda výsledné dílo odpovídá specifikaci. Testování rozdělujeme na manuální a automatické. Manuální uživatelské testování provádějí vždy lidé. Volba těchto „testerů“ by měl odpovídat cílové skupině, pro kterou byla aplikace navrhována a vyvíjena. V této kapitole budou popsány různé typy testování, které byly na knihovně provedeny a představen ukázkový projekt, který byl vytvořen jako součást této práce.

7.1 Jednotkové (Unit) testování

Jednotkové testování (unit testing) je metodika, která testuje kód po malých nezávislých částech. Testy jsou stavěny k tomu, aby byli spouštěny často a běžely rychle. U objektového kódu jsou tyto části často reprezentovány metodami. Jednotlivé testy by měli být mezi sebou pokud možno nezávislé a měly by testovat a sledovat pouze požadovanou vlastnost. Při návrhu testů je potřeba zvážit, zda test přinese užitek a zdali může odhalit potencionální chybu. Existují dokonce metodiky, které jdou ještě dále, jako TDD[1], které říkají že prvně bychom měli psát testy a až potom jejich samotnou implementaci. Avšak například testování „geterů“ a „setterů“ je zbytečné, pokud neobsahují aplikační logiku. Absolutní pokrytí všech možností není možné z časové a prostorové náročnosti vývoje a údržby těchto testů. Proto je vhodné testovat krajní případy a několik standardních případů použití. Pokud má objekt nějaké netriviální závislosti obsahující svou logiku, pak se typicky využívá mockování.

Mockování je metodika, kdy na místo opravdového objektu vytvoříme objekt falešný. Tento falešný objekt typicky žádnou logiku nemá, nebo pouze předem definované chování a slouží pouze k ověření, že náš kód tyto systémy volá v předem definovaném pořadí a parametry. Mockování se používá pokud nás nezajímá konkrétní integrace s jinými objekty, nebo tyto objekty jsou nedostupné, či časové náročné na zkonstruování.

K testování AspectFaces pluginu byl využit framework jUnit[2], alternativou je TestNG, avšak s ohledem na existující zkušenosti byl zvolen jUnit. Pro mockování bylo vybráno Mockito. Protože plugin je vytvořen jako Maven modul, lze jeho testy spustit standardně z pří-

[1] <http://en.wikipedia.org/wiki/Test-driven_development>
[2] <http://junit.org/>

kazové řádky zavoláním příkazu *mvn test*. Pro testování vybraných částí klientské knihovny byla zvolena knihovna Jasmine[3].

7.2 Výkonové testování

Protože testování kódu není jenom kvalita samotného kódu, ale i jeho rychlost, která zásadní mírou ovlivňuje uživatelskou přívětivost, je na místě otestovat generování metamodelu i z hlediska výkonu. Cílem tohoto měření je změřit zatížení, které samotná knihovna AspectFaces společně s nově vzniklým pluginem generuje. Je zřejmé, že s ohledem na použitý flexibilní přístup ke generování metamodelu ze šablony za použití EL bude mít viditelný dopad, avšak knihovna by se ho měla pokusit svým návrhem minimalizovat.

Pro testování byl vybrán endpoint **/person/1** z ukázkové aplikace, který představuje nadprůměrně složitou entitou, která se sestavuje hned ze dvou tříd: Person a Address. Měřený čas budeme brát od odeslání požadavku k přijetí výsledku zpět. Díky tomu, že ukázkový projekt obsahuje data přímo v paměti, nebude test tolik zkreslen.

Jako nástroj pro test výkonosti byl zvolen Apache Jmeter[4]. Jedná se o Javovskou desktopovou aplikaci vhodnou jak pro jednoduché, tak komplexní měření a provádění load testů. Pro naše účely bohatě postačí konfigurace s těmito parametry:

- 20 současných vláken
- 20 opakování

Konfigurace testovacího stroje na kterém poběží jak aplikace JMetter, tak instance defaultně nastaveného Tomcatu 8.0.20x64 běžícího na Oracle JDK 1.7.0_71 64bit, na které poběží ukázková aplikace je:

- Dell Precision 4620
- procesor: Intel Core i5 2520M, 2,5Ghz
- paměť: 8GB 667 Mhz DDR3

Výkonostní testování proběhlo ve dvou fázích: s použitím naimplementovaného modulu javaee-rest a poté pouze staticky vrácením obsahu stejné velikosti. Cílem bylo porovnat jak velké zatížení přinese knihovna a jak moc zpomalý běh aplikace s ohledem na stejně velká přenesená data.

Měření bylo opakováno 10x a hodnoty potom zprůměrovány. Medián doby odpovědi s použitím knihovny byl 15ms, maximální doba odezvy byla 34ms. Medián doby odezvy bez použití pluginu se staticky vráceným textem o stejné velikosti byl 5ms a maximální doba odezvy byla 28ms.

Z testování vyplynulo několik zajímavých poznatků. Knihovna umožňuje přepnutí do development módu, které necachuje konfiguraci, mapování, ani šablony. V takovém případě

[3] <http://jasmine.github.io/>
[4] <http://jmeter.apache.org/>

jsou doby odezvy větší. Druhým aspektem, který velice ovlivňuje výkon je nastavená úroveň logování. Pro vývoj je vhodné si nechat vypisovat jednotlivé řádky šablon, jejich transformace a výsledné přenesené hodnoty na úrovni DEBUG, ovšem v produkci to znamená zpomalení v řádu desítek až stovek procent.

Z testování jasně vyplynulo, že využití knihovny nenese prakticky žádné zásadní zpomalení a toto zpomalení je lineární co do počtu transformovaných objektů.

7.3 Ukázkový projekt

Součástí práce je i relativně rozsáhlý ukázkový projekt, jehož některé fragmenty kódu již byly ukázány. Tento projekt obsahuje, jak klientskou, tak serverovou část. Projekt slouží jako demonstrační aplikace a k testovacím účelům během vývoje i k akceptaci celého řešení proti seznamu nadefinovaných požadavkům.

7.3.1 Popis projektu

Ukázkový projekt je zjednodušená aplikace sloužící k zadávání dovolených. Aplikace umožňuje vkládat žádosti o dovolenou. Administrátor má možnost dovolené spravovat, také může spravovat země, ve kterých se uživatelé nachází, a ke kterým se váží konkrétní typy pracovní nepřítomnosti. Serverová strana je postavena kompletně pouze v paměti počítače. RESTové API je postaveno nad Springem. Zabezpečení aplikace je realizováno pomocí Spring Security, který na základě basic autorizace a anotací určí, zda uživatel má právo přístupu ke konkrétnímu zdroji, či nikoliv.

Klientská část využívá realizované JavaScriptové knihovny k dynamickému generování formulářů z metamodelu ze serveru. V aplikaci existuje jednoduchá implementace security kontextu uloženého v sessionstorage prohlížeče pamatující si aktuálně přihlášeného uživatele. Autentizace je prováděna proti serveru, který vrací informace o přihlášeném uživateli zpět na klienta.

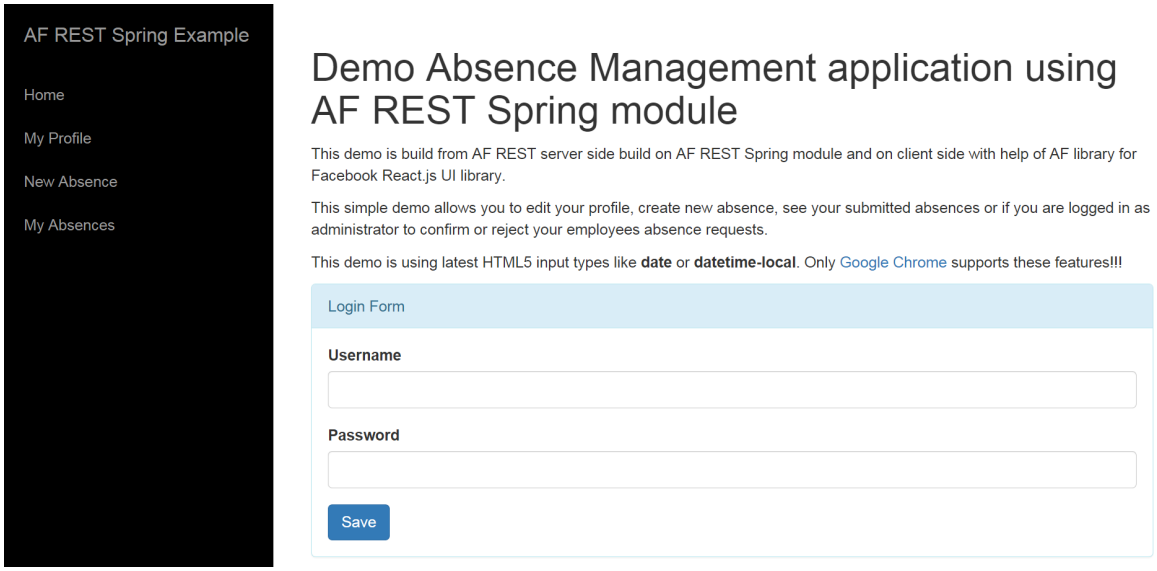
Dostupní uživatelé a jejich role shrnuje tabulka 7.1

Uživatelské jméno	**Heslo**	**Role**
admin	admin	USER, ADMIN
tom	tom	USER
user	user	USER

Tabulka 7.1: Dostupní uživatelé ukázkové aplikace

7.3.2 Nasazení

Aplikaci lze velice snadno spustit rozbalením archivu javaee-rest-spring-example.zip a spuštěním skriptu *startup.bat* v adresáři */bin* rozbaleného Apache Tomcat aplikačního serveru. Poté již stačí přejít na URL adresu http://localhost:8080/index.html kam server defaultně startuje. **Pozor na uvedení stránky index.html** v URL adrese.



Obrázek 7.1: Ukázkový projekt - klientská část

7.3.3 Zhodnocení

Ukázkový projekt dokazuje velice snadnou použitelnost AspectFaces pluginu na serverové straně a velkou flexibilitu mapovacích pravidel. Dynamická inspekce a šablonování nám pomáhá řešit problém s podmíněně zobrazovnými fieldy entit. Díky navrženému prefixování fieldu, které mimo jiné zachovává unikátnost mezi názvy, můžeme snadno bindovat data zpět na doménové entity pomocí dostupných knihoven. Klientská strana potvrdila počáteční myšlenky o neduplikaci kódu a velmi zjednodušené udržbě v případě změn provedených na serveru. Samotná JavaScriptová knihovna umožnila velmi flexibilní přístup k vytváření vlastního rozložení formuláře, což s porovnáním s některými představenými knihovnami je velká výhoda. Obecně tento přístup na ukázkovém projektu potvrdil svou flexibilitu a užitečnost a bude dále rozvíjen.

Kapitola 8

Závěr

Diplomová práce splnila cíle, které si stanovila. Byli zanalyzovány existující přístupy k vytváření UI a to jak na klientské, tak serverové straně a porovnány jejich výhody a nevýhody. Poté byla provedena rešerše existujících JavaScriptových knihoven vhodných pro automatické generování částí UI, specificky formulářů. Porovnáním dostupných knihoven bylo zjištěno, že generování formulářů na základě schématu není novým přístupem, avšak zatím neexistuje standardní přístup k definici tohoto schématu. Všechny porovnávané knihovny obsahovaly způsob řešení s jemnými rozdíly.

V rámci této práce byla vytvořena JavaScriptová knihovna pro generování dynamických formulářů na klientské straně, postavená na knihovně React.js. Pro serverovou stranu byl vytvořen plugin do existující knihovny AspectFaces, který umožňuje dynamické generování metamodelu na základě dostupného kontextu z klientské i serverové strany. Použití pluginu pro dynamické generování metamodelu a jeho využití k automatickému vygenerování editovatelných i needitovatelných formulářů zásadně urychluje tvorbu UI a snižuje duplikaci kódu. Avšak nejenom tom, díky jedinému zdroji pravdy, který je obsažen v metamodelu, jež generuje server, jsou jakékoliv změny provedené na serverové straně automaticky propagované i na klientské straně. Společně s přenášenými a interpretovanými validacemi knihovna výrazně zlepšuje uživatelskou interakci se systémem.

Výsledná práce byla otestována na netriviálním ukázkovém projektu demonstrujícím použití obou knihoven. Na ukázkové aplikaci bylo provedeno výkonostní testování, které odhalilo očekávané lineární zpomalení systému v případě generování metamodelu pro kolekci objektů. V případě standardního použití, či cachování metamodelu je čas konstantní a prakticky zanedbatelný.

Knihovna je dostupná z veřejného GIT repozitáře[1] a čeká na merge do hlavní větvě knihovny AspectFaces a zveřejnění komunitě.

8.1 Budoucí práce

Následující kroky povedou hlavně k vylepšení dokumentace a příkladů použití. Dalším krokem bude vytvoření dalších pluginu stavějící nad základním pluginem *javaee-rest* a usnadňující jeho integraci do dalších frameworků, jako například RESTEasy. Podobnou cestou

[1] <https://bitbucket.org/prochka/aspectfaces>

se bude ubírat i práce na klientské části. Je potřeba doplnit aktuálně podporované vstupní komponenty o další běžně používané, jako například date picker, či autocomplete.

Literatura

[1] *AspectFaces Documentation* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <http://wiki.codingcrayons.com/display/af/AspectFaces>.

[2] *Automatic Programming* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <http://www.cs.utexas.edu/users/novak/autop.html>.

[3] *JSON Schema* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <http://json-schema.org/>.

[4] *Apache Maven* [online]. 2012. [cit. 17. 5. 2012]. Dostupné z: <http://maven.apache.org/>.

[5] *React.js* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <https://facebook.github.io/react/i>.

[6] *RESTEasy - JAX-RS implementace* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <http://www.jboss.org/resteasy/>.

[7] *jQuery* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <https://jquery.com/>.

[8] B. MYERS, R. P. S. E. H. Past, present and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*. 2000, 7, 1, s. 3–28.

[9] BERNARD, E. [online].

[10] AL., G. E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[11] KENNARD, R. *Metawidget* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <http://metawidget.org/>.

[12] P. A. AKIKI, Y. Y. A. K. B. Adaptive model-driven user interface development systems. *ACM Computing Surveys (CSUR)*. 2014, 47, 9.

[13] RANEBURGER, D. Interactive model driven graphical user interface generation. *In Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. 2010, s. 321–324.

[14] T. CERNY, E. S. M. J. D. Toward Effective Adaptive User Interfaces Design. *In Proceedings of the 2013 Research in Adaptive and Convergent Systems*. 2013, 13, s. 373–380.

[15] T. CERNY, M. J. D. E. S. K. C. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. *Applied Computing Review 13.* 2013, 4, s. 53–65.

[16] WIKIPEDIE, P. *JavaScript* [online]. 2015. [cit. 8. 5. 2015]. Dostupné z: <http://cs.wikipedia.org/wiki/JavaScript>.

Příloha A

Seznam použitých zkratek

UI User Interface

UX User Experience

API Application Programming Interface

REST Representational State Transfer

XML Extensible Markup Language

JSON JavaScript Object Notation

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

WSDL Web Service Definition Language

W3C World Wide Web Consortium

WWW World Wide Web

URI Uniform Resource Identifier

URL Uniform Resource Locator

JSR Java Specification Request

HTML HyperText Markup Language

UML Unified Modeling Language

JAX-RS Java API for RESTful Web Services

JAXB Java Architecture for XML Binding

DTO Data Transfer Object

JDK Java Development Kit

JRE Java Runtime Environment

Příloha B

Instalační příručka

Knihovna byla vytvořena jako Maven[4] projekt, jeho přidání do již existující aplikace lze provést buď jako knihovnu (JAR), nebo lépe jako Maven závislost, viz B.1. K tomu je potřeba nastavit cestu k Maven repozitáři, kde lze artefakt najít. Kompilaci zdrojových kódů lze provést standardním způsobem *mvn install*. Klientská strana nevyžaduje žádné speciální nástroje, stačí pouze zkopírovat do projektu a přidat jako script tagy.

```
1 <dependency>
2   <groupId>com.codingcrayons.aspectfaces</groupId>
3   <artifactId>javaee-rest-spring</artifactId>
4   <version>1.5.0-SNAPSHOT</version>
5 </dependency>
```

Část zdrojového kódu B.1: Ukázka Maven závislosti na pluginu javaee-rest-spring

B.1 Softwarové požadavky

- Serverový plugin do AspectFaces vyžaduje Javu 7 a vyšší.
- Klientská knihovna s ohledem na používané HTML5 typy inputů korektně funguje pouze v Google Chrome.

B.2 Ukázkový projekt

Ukázkový projekt, včetně klientské části je již připraven a přiložen na CD. Ukázkový projekt je rovnou distribuován s aplikačním serverem Apache Tomcat[1]. Ke spuštění již stačí pouze rozbalit archiv *javaee-rest-spring-example.jar* a spustit startovací skript serveru, který nalezneme v podadresáři *bin* serveru.

[1] <http://tomcat.apache.org/>

Příloha C

Obsah přiloženého CD

Na CD jsou umístěny tyto položky:

- *javaee-rest* - složka obsahující zdrojové kódy AspectFaces pluginu
- *javaee-rest-spring* - složka obsahující rozšíření pro pro plugin javaee-rest s podporou Spring Frameworku
- *javaee-rest-spring-example* - zdrojové kódy ukázkového projektu, včetně klientské knihovny
- *javaee-rest-spring-example.zip* - zazipovaný archiv obsahující Apache Tomcat 8.0.22 se spustitelný ukázkovým projektem na adrese localhost:8080/index.html
- *dipl* - složka obsahující text diplomové práce