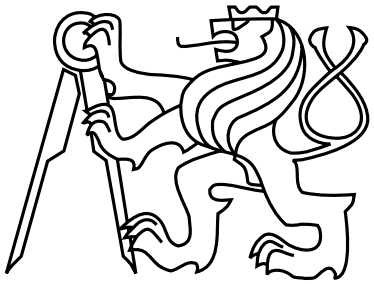
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Aspektově orientovaný vývoj uživatelských rozhraní pro Java SE aplikace

Bc. Martin Tomášek

Vedoucí práce: Ing. Tomáš Černý, MSc.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

1. ledna 2015

Poděkování

Tímto bych rád poděkoval celé svojí rodině za podporu během studia. Dále bych rád poděkoval vedoucímu mé diplomové práce panu Ing. Tomáši Černému za ochotu, pomoc, čas, zkušenosti a příležitosti, které mi poskytoval během celého mého studia.

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).

Ve Strakonicích 28. 12. 2014 .....

Abstract

Present-day applications use graphical user interface in order to interact with the user. We can find the input fields in each application together with the output fields and other components that are used to operate the application. To generate the forms that contain data that we want to insert, edit or view is time-consuming. In addition to generating the correct input fields, it is also necessary to consider the production of the user interface in terms of validation, distribution of components and security. In these cases, it would be appropriate that the parts of the interface were generated based on a model. In the case of client server types of applications, the model and the data can be provided by a server. This solution will provide a centralized data management and its definition and will allow the client to respond flexibly to any changes in the data model.

Abstrakt

Současné aplikace využívají grafické uživatelské rozhraní k interakci s uživatelem. V každé aplikaci můžeme najít vstupní pole, výstupní pole a další komponenty, které slouží k ovládání aplikace. Generování formulářů, která obsahují data, jenž chceme vkládat, editovat nebo prohlížet je časově náročné. Kromě generování správných vstupních polí je také potřeba nahlížet na tvorbu uživatelského rozhraní z pohledu validací, rozložení komponent a bezpečnosti. V těchto případech by bylo vhodné, aby se části rozhraní generovala na základě modelu. V případě aplikací typu klient server může model a data poskytovat server. Toto řešení zajistí centralizovanou správu dat a jejich definicí a umožní klientovi pružně reagovat na změnu datového modelu.

Obsah

Seznam obrázků

Seznam tabulek

Seznam částí zdrojových kódů

Kapitola 1

Úvod

Tato diplomová práce se zabývá analýzou, návrhem a otestováním generovaného uživatelského rozhraní na základě aspektů, které by bylo využitelné na platformě Java SE. Práce se zaměřuje zejména na generování rozhraní v tlustých klientech neboť trendem moderní doby je využívat webové API serverů, z kterých jsou získávána data. V první části práce jsou popsány problémy, jenž jsou s tímto procesem spojeny. Druhá část analyzuje způsob, jakým lze proces zjednodušit a navrhuje framework, který by tohoto cíle dosáhl. Třetí část diplomové práce popisuje vlastní implementaci a celkovou architekturu frameworku. Poslední část se zabývá testováním a vytvořením ukázkového projektu, v kterém je framework použit.

Práce obsahuje seznam použitých zkratek viz. příloha A, instalační a uživatelskou příručku viz. příloha B, použité UML diagramy a obrázky viz. příloha C, ukázky zdrojového kódu a XML souborů viz. příloha D a samozřejmě zdrojové kódy, které jsou přiloženy na CD. Obsah tohoto CD je v příloze E.

1.1 Motivace

Vytváření uživatelských rozhraní je součástí téměř každé aplikace. Obvykle jsou zobrazovány formuláře, do kterých se zadávají data, která mohou být následně zobrazována v tabulkách. Vývoj uživatelských rozhraní je časové náročná věc, která obvykle podléhá testování jak funkčnosti, tak použitelnosti. Kromě toho lze předpokládat, že se bude navržené rozhraní měnit. Tyto změny jsou obvykle iniciovány zákazníkem. V případě, že jsou získávána data ze serveru, tak musí klientská aplikace znát strukturu dat, na jejichž základě vytváří komponenty. Pokud je struktura změněna, pak je potřeba upravit i klienta. Generování uživatelského rozhraní za běhu aplikace tyto problémy odstraní, neboť umožní klientovi dynamicky reagovat na změnu dat. Na uživatelské rozhraní lze nahlížet i z dalších aspektů, jako je rozvržení komponent, bezpečnost a validace. Tyto aspekty vyžadují další čas na vývoj aplikace. Pokud bychom vytvořili framework, který automatizuje zadané procesy a správně interpretuje uživatelské rozhraní, pak bychom ušetřili čas a snížili náklady potřebné na vývoj této aplikace. Toto téma mi přišlo velmi zajímavé, a když mi bylo nabídnuto vytvořit koncept, který by výše uvedené věci automatizoval, tak jsem neváhal a zpracoval téma jako diplomovou práci.

Kapitola 2

Popis problému a specifikace cíle

2.1 Popis problematiky

Softwarové systémy jsou určeny k tomu, aby méně či více úspěšně poskytovaly uživateli nástroj, který mu pomůže s řešením problémů. Systém tedy musí komunikovat s uživatelem. K tomuto účelu se využívá uživatelské rozhraní. Vývoj uživatelského rozhraní zabere přibližně 50 % času [17], který je určen na vývoj konkrétního systému. Tento údaj se samozřejmě může lišit v závislosti na účelu a velikosti systému. Při tvorbě uživatelského rozhraní se obvykle zaměřujeme na použitelnost. V tomto případě provádíme testy použitelnosti na cílové skupině, na jejichž základě jsme schopni určit, zdali je návrh použitelný či nikoliv. Důvodem tohoto testování je fakt, že obvykle systém vytváříme pro uživatele a ne obráceně. Z výše uvedených skutečností vyplývá, že je potřeba uživatelské rozhraní důkladně testovat, aby bylo pro cílovou skupinu správně použitelné. Na uživatelské rozhraní lze nahlížet z aspektu, jakým ho bude používat koncový uživatel. Rozhraní musí mít prvky umožňující vložit data, také musí vložená data validovat a jistě musí zobrazovat uživateli data v závislosti na jeho roli. Na uživatelské rozhraní se proto můžeme dívat z hlediska reprezentace dat, bezpečnosti, uspořádání komponent a mapování dat na entitu, kterou reprezentují [20].

Bohužel, když se hovoří o uživatelském rozhraní, často se zapomíná na to, že toto rozhraní se musí nejen vytvořit, ale také udržovat. Softwarový systém tráví většinu svého života v udržovacím režimu, kterému se říká support nebo li podpora. V této fázi přichází na systém mnoho požadavků, které musí být proveditelné a to za přijatelné náklady. Nedílnou součástí jsou změny, které se týkají databázového modelu a obvykle tyto změny musí reflektovat UI. Podívejme se proto na systém z pohledu vývojáře. Systém pro něj musí být snadno udržovatelný, změny lehce proveditelné a bez větších dopadů na systém. V tomto případě by bylo vhodné reflektovat tyto změny v UI. Provedené změny v uživatelském rozhraní na klientské straně musí být v souladu s modelem, který rozrhraní zobrazuje. Jedná se především o typovou kontrolu. Při změně může dojít ze strany vývojáře k chybě, která může mít za následek nefunkčnost aplikace [19]. Nedílnou součástí každého uživatelského rozhraní jsou validace, které by měli reflektovat například změny v databázovém modelu ale i změny v business modelu.

2.1.1 Typy uživatelských rozhraní

Jak již bylo zmíněno, uživatelské rozhraní se testuje na základě typu aplikace a jejím použití. Je také důležité vzít v potaz zařízení, na kterém je aplikace provozována. Může se jednat o desktopovou, mobilní či serverou aplikaci. V každém z výše uvedených případů bude návrh uživatelského rozhraní podmíněn jinými faktory, které jsou specifické pro dané zařízení. Těmito faktory jsou způsoby, jakými se aplikace ovládá, prostředí, v kterém se uživatel právě nachází a účel, ke kterému je aplikace určena. Například aplikace na mobilních zařízení nemusí podporovat klávesové zkratky, ale mohla by podporovat gesta. Obdobně aplikace použitá na desktopu může počítat s použití myše, touchpadu, klávesnice - jak standardní tak dotykové, či jiného externího zařízení. Je tedy zřejmé, že uživatelské rozhraní je kromě jeho účelu podmíněno i zařízením, na kterém je používáno.

Základními ovládacími a vizuálními prvky téměř každé aplikace jsou tlačítka, vstupní pole, přepínače, tabulky, menu a statické texty. Vstupní pole můžeme shrnout do jedné kategorie, která se nazývá formulář. Formulář obvykle osahuje 1 až N prvků, s tím, že zde má každý prvek svojí funkčnost a účel. Účelem je poskytnout uživateli možnost vložení dat, či možnost volby chování aplikace. Funkčností je tato data správně interpretovat a na jejich základě provést specifické akce. Ve formuláři také mohou být pouze statická data, která slouží k reprezentaci aktuálního stavu, který slouží uživateli k tomu, aby pochopil aktuální stav ve kterém se aplikace či jeho část nachází a na základě tohoto stavu mohl rozhodnout o další akci, pokud je toto rozhodnutí vyžadování a umožněno.

2.1.2 Získávání a vkládání dat

Aplikace používá vizuální prvky k tomu, aby uživateli reprezentovala data či umožnila uživateli tato data vytvořit. Moderním způsobem je dnes využívat k získávání a vkládání dat webové služby. Výhodou je, že se klient může připojit na různé zdroje a z těchto dat si vytvářet tzv.: mashup. Obvyklé použití je takové, že server získá data z více zdrojů a ty pak interpretuje klientům. Klient tedy nezná originální původ informací. Jedním z dalších způsobů je vlastní databáze na klientovi. V tomto případě již ale nemůžeme hovořit o klientovi, neboť se jedná o soběstačnou aplikaci, v případě, že nezískává data z jiných dalších zdrojů. Samozřejmě existují i kombinace těchto možností. Volba závisí vždy na konkrétním zadání a účelu, pro který je aplikace navržena.

V případě reprezentace je potřeba data získat. Jak již bylo zmíněno výše, existuje mnoho způsobů, kde a jak data získat. Zaměříme se nyní na získání dat z jiných zdrojů. Pokud žádáme o data jiný zdroj, tak jsme obvykle schopni zjistit formát a způsob, jakým o data požádat, ale strukturu dat předem neznáme. Nejčastěji jsou data přenášena jako JSON [7] nebo XML. To nám umožní data serializovat do objektu, pokud známe definici objektu. Definice objektu lze získat obvykle v dokumentaci k dané službě, takže námi navržená aplikace očekává data specifického typu. Uvažujme o následujícím příkladu, ve kterém jsme vytvořili klienta, jenž zobrazuje jména a příjmení uživatelů v systému. Po určité době je však potřeba kromě jména zobrazovat i jejich uživatelské jméno. Do dat, která získáváme od služby, tedy přibude sloupeček s uživatelským jménem. Nyní musíme naší klientskou aplikaci upravit tak, aby byla schopná zobrazovat i tyto informace. Provedli jsme tady poměrně triviální úpravu. Přidali jsme pole k zobrazení uživatelského jména, upravili jsme objekt, do kterého se data serializovala, a v další verzi vydání aplikace se tato změna projeví. Změna tedy není

klientům dostupná ihned. Po určité době se rozhodlo, že se kolonka uživatelského jména odstraní. Tento případ je tedy mnohem horší. Neboť po serializace bude v poli reprezentující uživatelské jméno hodnota null. Pokud jsme jméno pouze vypisovali je vše v pořádku, avšak pokud jsme nad ním prováděli nějaké operace, můžeme obdržet výjimku a aplikace nemusí být schopna pokračovat v běhu.

V případě vkládání dat do jiného zdroje platí chování a nastavení z odstavce uvedeného výše. Je tedy potřeba znát zdroj, na který lze data odeslat, formát dat, metodu a popřípadě další dodatečná nastavení služby. Budeme uvažovat o stejném příkladu, který byl již rozebrán v předchozím odstavci s tím rozdílem, že nyní data vkládáme. Na server tedy nejprve odesíláme pouze jméno a příjmení. Předpokládejme, že tyto dvě hodnoty jsou serverem vyžadovány. Je tedy potřeba mít validaci, která zkontroluje, zdali jsou data před odesláním v pořádku nebo musíme správně interpretovat odpověď serveru, který sdělí, že data nejsou validní, pokud touto validací disponuje. Při přidání nového pole, u kterého server vyžaduje jeho vyplnění, klient přestane správně fungovat a server by měl data odmítat. Musíme tedy udělat změnu na klientovi. Přidat vstupní pole, přidat proměnou, která bude zastupovat uživatelské jméno a novou verzi nasadit a distribuovat. Po určité době, stejně jako v prvním případě se definice dat změní a uživatelské jméno již nebude klientům zasíláno a nebude ani možnost ho vyplňovat. Klient se opět stane nevalidním, neboť při serializaci na serverové straně dojde k chybě, protože klient posílá proměnou, kterou již nyní server nezná. Formulář se tedy opět stane nefunkčním.

2.1.3 Aspektový přístup

Z dosavadního testu je zřejmé, že aplikace obsahuje vizuální prvky, na které lze nahlížet z několika aspektů. Jedním z důležitých aspektů je bezpečnost. Funkce, které může konkrétní uživatel využívat, se přidělují na základě uživatelských rolí. V této souvislosti je mnoho přístupů jak role přidělovat a spravovat, ale všechny způsoby mají jedno společné. Ověřují, zdali má uživatel právo akci provádět. Souvislost mezi rolí a uživatelským rozhraním je patrná. Mějme uživatele v role administrátora. Tato role bude mít práva na úpravu uživatelských dat jiných uživatelů. Dále mějme roli hosta, která si může data uživatelů pouze zobrazit. Při detailnějším prozkoumání zjistíme, že existuje množina zobrazovaných dat, která je pro obě role stejná, nicméně pro roli hosta by měla být všechna tato data needitovatelná. Dosáhnout této možnosti lze několika způsoby a záleží na platformě a volbě řešení. Například v Java SE aplikaci využívající Swing to znamená, že buď musí být data zobrazována jakou label nebo musí být komponenty vypnuty. V obou případech musí vývojář na základě uživatelské role zvolit jeden z přístupů a nastavovat data na konkrétní komponentě. Pokud zvolí způsob labelů, tak vytváří duplicitní formulář pouze s jiným aktivním prvkem. Data získávána ze serveru, jsou již poskytovány na základě bezpečnostní politiky a postačilo by jejich výsledky propagovat do klientské aplikace. Server by tedy sám rozhodl, jaká data zobrazit.

2.2 Existující řešení

V současné době existuje několik řešení, které se snaží zjednodušit tvorbu uživatelského rozhraní. Níže zmíněné technologie jsou frameworky [2], které nabízí předpřipravené komponenty

či usnadňují způsob, jakým lze data přenášet spravovat či reprezentovat s cílem zjednodušit tvorbu uživatelských rozhraní Jsou jimi například RichFaces [13], PrimeFaces [12], JSF, JSP, Swing, Struts, Vaadin. Tyto řešení se prozatím nezaměřují na dynamické generování uživatelského rozhraní, ale pouze poskytují komponenty či nástroje, které vývoj urychlí. Zjednodušení je například v tom, že vývojář nemusí definovat tabulku pomocí HTML tagu table, ale lze využít předpřipravenou komponentu. Poskytovaná řešení obvykle využívají návrhových vzorů. Například ve Swingu je využívají komponenty vzoru MVC [22], stejně tak JSF [21]. Žádná z výše uvedených technologií však neumožňuje generovat dynamická uživatelská rozhraní na základě obecných definic, či tyto definice vytvářet.

2.2.1 SwiXml

Tento framework se zaměřuje pouze na generování uživatelského rozhraní ve Swingu. Základem je specifikace rozhraní pomocí XML, což je velmi velkou výhodou, neboť specifikace v tomto formátu má jasně dané možnosti a je na první pohled zřejmé jak bude daná komponenta fungovat. Knihovna implementuje téměř všechny možnosti, které lze nastavit standardní cestou vývoje Swing aplikace [15]. ActionListenery a dodatečné nastavení si vývojář může specifkovat po vygenerování komponent. Hlavní nevýhodou je, že všechny komponenty, které se generují, je potřeba mít specifikované i ve výsledné aplikaci. Není zde žádné zapouzdření komponent a při detailnější specifikaci se stává XML definice až příliš rozsáhlá.

2.2.2 Metawidget

Projekt Metawidget [11] se zaměřuje na vytváření uživatelského rozhraní na základě inspekce tříd. Použít ji lze s mnoha populárními frameworky jako jsou Spring, Struts, JSF, JSP a další. Generování uživatelského rozhraní probíhá na základě inspekce již existující třídy a konfigurace konkrétní aplikace. Aktuální verze je 4.0 a byla vydána 1. listopadu 2014. Je k dispozici pod licencí LGPL/EPL. Mezi hlavní výhody této knihovny patří zejména široká škála podporovaných frameworků a validátorů. Data lze získat i pomocí REST, nicméně nativní a jednoduchá podpora zde chybí. Framework bohužel neumožňuje generování tabulek. Avšak lze vytvořit vlastní implementaci widget builderu, která bude umět vytvořit jakoukoliv komponentu.

2.2.3 AspectFaces

Jedná se o framework, který umožňuje inspekci na základě tříd [1]. Framework umožňuje použití různých layoutů a inspekčních pravidel. Výsledné vygenerované uživatelské rozhraní se může pro stejné objekty lišit na základě specifického nastavení kontextu. Framework vychází z myšlenky, že uživatelské rozhraní by mělo být generováno na základě modelu. [19]. Inspekce je tedy prováděna vůči statické třídě, ale výsledek závisí na aktualních použitých šablonách a kontextu. V současné době je stabilní verze 1.4.0 a je distribuován pod licencí LGPL v3 na verzi 1.5.0 se v současnosti pracuje. Vývojář si může své vlastní nastavení pro generování uživatelského rozhraní upravit. Tato nastavení jsou v XML formátu a lze je tedy snadno modifikovat. Framework podporuje velkou škálu anotací z JPA, Hibernate a uživatel si v případě nutnosti může vytvořit vlastní anotaci, která se promítne do inspekce. Framework je

prozatím bohužel jednostranně orientovaný na JSF. Tomu odpovídá i způsob generování dat a způsob, jakým je prováděna složitější inspekce.

2.3 Cíle projektu

Existující řešení poskytují mnoho různorodých funkcí. Jejich hlavní výhody jsou zkrácení času, který je potřeba k vývoji a úpravě uživatelského rozhraní. Trendem současné doby jsou webové služby, proto se i v této práci budu soustředit na získávání definice dat z webových služeb a jejích interpretaci na klienta stejně tak jako na plnění této reprezentace skutečnými daty. Inspekce tedy bude prováděna na straně serveru, který zná objekty, s kterými pracuje a klient pouze obdrží jejich definici. Tento přístup umožní klientovi pružně a ihned reagovat na změny v datovém formátu, který diktuje server. Dalším pozitivním vlivem, bude to, že server bude klientovi poskytovat i seznam validací, jimiž musí jednotlivá komponenta vyhovět, aby bylo možné data odeslat zpět na server a ten je správně zpracoval. Celý tento proces by měl být pro klienta zapouzdřen, aby aplikace nevyžadovala od klienta více informací než je nutné. Mezi nutnou informaci patří specifikace připojení a formát dat. Například JSON, XML. Použití frameworku by mělo být velmi jednoduché a v případě, že bude chtít klient postavit formulář, mělo by mu stačit pouze několik řádků kódu. Dalším důležitým aspektem jsou již existující zdroje na serveru, které by měli po přidání frameworku zůstat stejné.

Kapitola 3

Analýza

3.1 Funkční specifikace

Framework [2] musí uživateli umožňovat vytvářet a dále pracovat s vytvořenými komponentami. Komponenty budou procházet určitým životním cyklem. Kromě samotných komponent musí framework poskytovat i dodatečné funkcionality, které jsou spojeny se získáváním dat, jejich propagací na klienta, zabezpečení, organizací komponent, lokalizací a skinováním komponent.

3.1.1 Funkční požadavky

Z dosavadního popisu problému byly vytvořeny následující požadavky na systém.

- Framework bude umožňovat generovat metadata objektů, na základě kterých budou generovány komponenty.
- Framework bude umožňovat vygenerovat formulář nebo tabulku na základě dat získaných ze serveru.
- Framework bude generovat metadata, která nebudou závislá na platformě.
- Framework bude umožňovat získat data ze serveru.
- Framework bude umožňovat naplnit formulář i tabulku daty.
- Framework bude umožňovat odeslat data z formuláře zpět na server.
- Framework bude umožňovat používat lokalizační resource bundly.
- Framework bude umožňovat validaci dat na základě metadat, která obdržel od serveru.
- Framework bude umožňovat klientovi překrýt chybové validační hlášky.
- Framework bude umožňovat skinovatelnost.
- Framework bude umožňovat specifikovat zdroje ve formátu XML.

- Framework bude umožňovat vytvářet vstupní pole, combo boxy, výstupní pole, textarea, checkboxy, option buttony.
- Framework bude umožňovat vkládat do formulářových polí texty, čísla a datum.
- Framework bude umožňovat generování komponent určených pouze pro čtení.

Z požadavků vyplívá, že framework bude umožňovat získání definici dat na serveru a poté je distribuuje koncovému uživateli. Koncový uživatel tedy nebude potřebovat znát objekty, s kterými pracuje. Toto zaručí pružnou reakci na změnu dat a generování aktuálních formulářů či tabulek.

3.2 Popis architektury a komunikace

Jak již bylo uvedeno, definice objektů, na základě které se budou vytvářet komponenty, je generována na serverové straně. Klient tedy komunikuje se serverem a vyžádá si tyto definice. Dále je potřeba definice na klientovi zpracovat. Definice nebudou závislé na platformě. Budou tedy popisovat data v obecné formě, což umožní generovat formuláře a tabulky nezávisle na platformě, jazyku a technologii. Referenční implementace bude napsána v jazyce Java s využitím komponentového frameworku Swing [14]. Definice dat, která jsou zasílána ze serveru na klienta by neměla ovlivňovat ostatní klienty, kteří framework nepoužívají.

Business proces, který zachycuje generování formuláře včetně validace a odeslání je zachycen na obrázku C.2. Uživatel nejprve specifikuje zdroje, které bude klient využívat. Rozeznáváme následující zdroje:

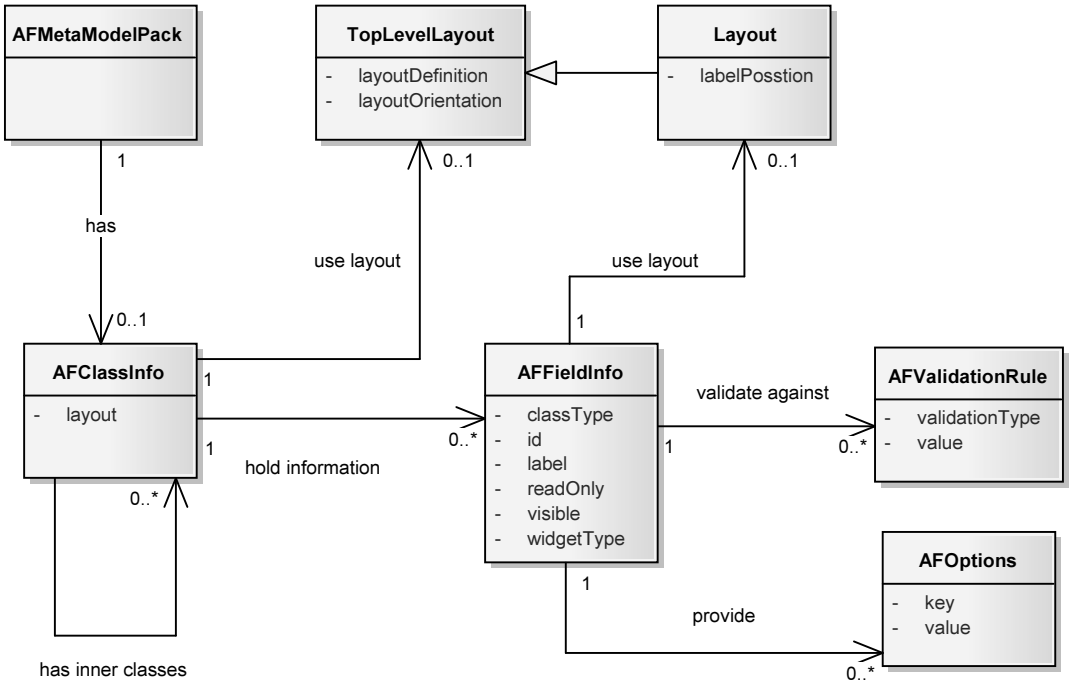
1. Zdroj s metadaty, které definují komponentu.
2. Zdroj s daty, která budou v komponentech zobrazena.
3. Zdroj, na který budou data odeslána.

Následně je vygenerována komponenta na základě metadat. V případě, že byl specifikován zdroj s daty, klientská část aplikace požádá server o tato data a vloží je do předpřipravené komponenty. Pokud datový zdroj specifikován není, zůstane komponenta bez konkrétního obsahu. Komponenta je nyní připravena a uživatel s ní může pracovat. V případě, že uživatel chce odeslat data na server a specifikoval zdroj, na který budou data odeslána, pak framework provede validaci dat. Pokud je validace dat úspěšná, na základě metadat se sestaví objekt, který je naplněn daty z aktuálního formuláře, a je odeslán na server. Server zpracuje request a vrátí klientovi odpověď. Na základě této odpovědi může uživatel dále upravovat formulář či s ním pracovat.

3.2.1 Metadata

Metadata [10] jsou data o datech. Framework, který je popsán v této práci, generuje na straně serveru metadata a zasílá je klientovi, který na jejich základě sestaví komponenty a poté je s nimi schopný pracovat. Metadata jsou generována z modelu, který reprezentuje aktuální uživatelské rozhraní [18]. Výhodou je, že model může reprezentovat databázové entity. Bude

mít specifikované validace a chování. Například pomocí anotací. Generováním metadat, která budou sloužit k sestavení komponent, na základě modelu přináší několik výhod. Generovaná data budou okamžitě reflektovat změny v modelu. Kromě toho mohou reflektovat i změny uživatelských rolí, pokud je generování těmito rolemi podmíněno. Klient musí být schopný s metadaty, které obdržel pracovat. Konkrétně je potřeba aby mohl data zpětně sestavit a odeslat je na server. Klient využívá klientskou část frameworku a server serverovou část. Z hlediska implementace je důležité, aby byly objekty, nesoucí informace o metadatech, stejné a bylo možné provést na klientovi generování na základě těchto dat. Následující doménový model, modelovaný pomocí UML [16], znázorňuje popis definic objektu, který je vytvořen po inspekci zadaného objektu. Inspekce vytvoří XML popis, který je převeden na obecný popis, jenž lze využít ke generování dat na klientovi. Tento obecný popis je zaslán klientovi, který využívá klientskou část frameworku, jenž očekává tyto objekty a na jejich základě je schopná vygenerovat uživatelské rozhraní. Na obrázku 3.1 je zobrazen doménový model [16], který je použit při popisu metadat objektu. Nejedná se o doménový model frameworku, ale pouze jeho části, která je zodpovědná za reprezentaci metadat.



Obrázek 3.1: Doménový model objektů obsahující metadata o objektu, nad kterým byla provedena inspekce

3.2.1.1 AFMetaModelPack

Tato třída zapouzdřuje informace, které popisují objekt, nad kterým byla prováděna inspekce. Třída rovněž slouží jako fasáda a nabízí programátorovi upravení metadat po gene-

rování. V případě serveru je toto návratový typ zdroje, na který klient přistoupí, chce-li znát metadata, která zdroj poskytuje.

3.2.1.2 AFClassInfo

Tato třída udržuje informace o hlavním objektu, z kterého je vytvářena definice. Třída má dále reference na své vnitřní proměnné a své potomky. Na základě generování dat je potřeba udržet pořadí, v kterém byla nad jednotlivými komponenty prováděna inspekce. V případě inspekce dat je i reference na neprimitivní datový typ jeho proměnná. Nicméně je potřeba, aby byl klient schopný určit, že se jedná o složitý datový typ a určit jeho pořadí v aktuálním objektu, aby mohl rozhodnout na jaké pozici objekt zobrazit. Z tohoto důvodu je ve třídě AFFieldInfo proměnná classType, která určuje, zdali se jedná o vnitřní třídu či primitivní datový typ.

3.2.1.3 AFFieldInfo

Tato třída je zodpovědná za poskytování detailních informací o proměnné objektu, nad kterým byla provedena inspekce. Třída udržuje název proměnné, widget, na který bude proměnná převedena, pravidla, která musí být splněna, název, pod kterým bude prezentována uživateli a zdali se jedná o složitý či jednoduchý objekt. Kromě těchto vlastností nese objekt také informace o tom, je-li komponenta viditelná a pouze pro čtení.

3.2.1.4 AFRule

Každá proměnná má souhrn vlastností, které musí být splněny. Typ widgetu ještě vždy nemusí určovat datový typ komponenty a neurčuje, je-li pole povinné či nikoliv. Tento soubor vlastností je popisován v této třídě. Třída využívá ENUM, který specifikuje podporované validace. Důvodem je to, že klient musí být schopný vytvořit tyto validační pravidla a interpretovat je na komponentě svým vlastním způsobem, který je specifický pro technologii, kterou používá. Jednou z dalších výhod je validace XML souboru, z kterého jsou pravidla vytvářena. Framework vytvoří pouze ty validační pravidla, která podporuje. Klient poté musí podporovaná pravidla interpretovat.

3.2.1.5 AFOptions

Některé widgety umožňují, aby si uživatel vybral z několika předem připravených možností. Tyto možnosti musí být klientovi prezentovány. Tato třída udržuje informace o možnostech výběru v dané komponentě. Proměnná klíč je hodnota, která bude odeslána zpět na server a proměnná value je hodnota, která bude zobrazena klientovi. Tímto způsobem lze klientovi zobrazit jakýkoliv text, který bude zpětně mapován na jeho skutečnou hodnotu. Kromě textu lze samozřejmě zobrazit čísla či hodnoty výčtových typů.

3.2.2 Server

Server je zařízení či software, který umožňuje zpracovat požadavky od klientů a na jejich základě vytvořit odpověď. Server tedy poskytuje svým klientům určitý typ obsahu. Způsob

a původ obsahu, který server poskytuje, je pro klienta povětšinou neznámý. V současné době je velmi populární přístup, při kterém server získá data z více zdrojů a poskytne je klientovi. Hovoříme o tzv. mashup [26]. Mashup nemusí být pouze z veřejných zdrojů, lze využít i privátní zdroje, či lze k sestavení odpovědi využít další služby. Klient napojený na server tohoto typu nemusí mít o těchto dalších zdrojích vůbec žádnou povědomost a dotazuje se pouze vůči tomuto serveru, který zpracovává jeho požadavky. Klientů, kteří získávají data z veřejných, či privátních zdrojů serveru může být celá řada. Mohou to být vlastní privátní aplikace, mobilní aplikace, Javascriptoví klienti či další server, který pouze využívá veřejné zdroje serveru k sestavení odpovědi svým vlastním klientům. V těchto případech je potřeba zvážit způsob generování definic dat, které by mohly způsobit stávajícím klientům problémy. V ideálním případě musí být framework integrován takovým způsobem, aby byla zachována stávající funkcionalita a framework ji pouze rozšířil. Ke generování definic objektů jsou potřeba následující věci:

1. Objekt, jehož definice budou generovány.
2. Mapování, na jehož základě bude rozhodnuto, o jaký typ komponenty půjde.
3. Definice komponenty včetně vlastností jako jsou validace, layout a popis chování komponenty.
4. Layout, ve kterém budou komponenty sestaveny.
5. Framework, který provede inspekci.
6. Framework, který bude inspekci řídit a bude interpretovat vygenerovaná data. Tento framework musí zároveň ověřit validitu jednotlivých komponent.

Výše uvedené vlastnosti, nebudou mít vliv na změnu funkcionality. K inspekci a mapování bude využit framework AspectFaces [1], který umožňuje na základě datových typů rozhodnout jakou komponentu využít. Definice komponent a jejich vlastností bude již v plné kompetenci vývojáře, nicméně základní komponenty a jejich chování bude předpřipraveno ve vzorovém projektu, aby se vývojář mohl inspirovat.

3.2.3 Klient

Klientská část frameworku bude vytvářet komponenty na základě metadat, která obdržela od serveru. Klient nebude mít žádnou znalost o objektech, které mu server poskytuje, předtím než obdrží jejich definice. Klient nicméně musí vědět, který zdroj mu poskytne relevantní definice a který ze zdrojů mu poskytne data odpovídající těmto definicím. Zároveň také musí vědět, na který zdroj data zpětně odeslat. Zdroj je obvykle specifikován následujícími parametry:

1. Adresa serveru
2. Port
3. Protokol

4. Metoda (get, post, put, delete)

5. Dodatečné hlavičkové parametry například content-type

Klient tedy bude muset vždy specifikovat tyto parametry. Z hlediska použitelnosti je vhodné mít tyto specifikace v XML souboru, který bude umět klient jednoduše načíst. Pro usnadnění bude načítání provádět framework. Ukázka je na obrázku 3.1. V ukázkovém příkladu je specifikován zdroj s metadaty, který je vždy povinný. Zdroj se nachází na adrese http://localhost:8080/AFServer/rest/users/loginForm. Zdroj s daty není specifikován, což způsobí, že ve formuláři nebudou žádná data. Formulář bude možné odeslat na adresu http://localhost:8080/AFServer/rest/users/login. Zdroj má identifikátor loginForm. V jednom XML dokumentu lze mít více zdrojů. K vložení dat do konkrétního zdroje lze využít EL. V hlavičce může být 0 až N parametrů, přičemž každý parametr musí být uveden ve stejném formátu, jako je znázorněno na obrázku 3.1. Obdobný způsob se využívá v JavaEE aplikací v deskriptoru web.xml. Klient umí sestavit požadavek na základě tohoto popisu a interpretovat odpověď od serveru. Není tedy nutné, aby uživatel implementoval třídy, které se umožní aplikaci připojení na server a získání dat.

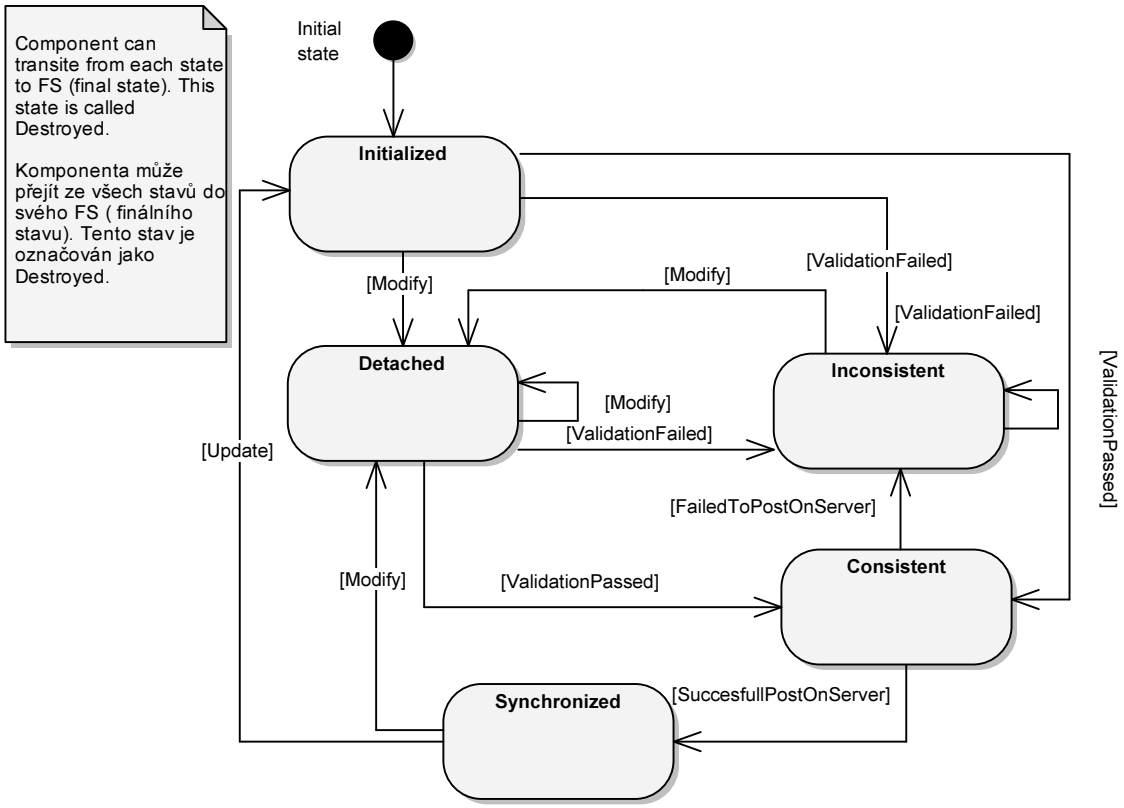
Část zdrojového kódu 3.1: Ukázka XML specifikace zdrojů

```
<?xml version="1.0" encoding="UTF-8"?>
<connectionRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <connection id="loginForm">
                <metaModel>
                        <endPoint>localhost</endPoint>
                        <endPointParameters>/AFServer/rest/users/loginForm</endPointParameters>
                        <protocol>http</protocol>
                        <port>8080</port>
                        <header-param>
                                <param>content-type</param>
                                <value>Application/Json</value>
                        </header-param>
                </metaModel>
                <send>
                        <endPoint>localhost</endPoint>
                        <endPointParameters>/AFServer/rest/users/login</endPointParameters>
                        <protocol>http</protocol>
                        <port>8080</port>
                        <method>post</method>
                        <header-param>
                                <param>content-type</param>
                                <value>Application/Json</value>
                        </header-param>
                </send>
        </connection>
</connectionRoot>
```

Je patrné, že klient je schopný získat definice formulářů či tabulek, naplnit je daty a poté odeslat zpět na server. Důvodem, proč je klient schopný generovat formuláře na základě definice ze serveru je ten, že klient pracuje se stejnými objekty, které popisují metadata, jako server. Prozatím je k dispozici pouze strohý popis dat. Klientská část musí nyní rozhodnout jak data interpretovat, jak s nimi pracovat, jakým způsobem je validovat, jak je znovu sestavit

a odeslat na server. Důležitým prvkem je i způsob uspořádání jednotlivých prvků, jejich velikosti, barvy a texty.

Využití frameworku by obdobně jako v případě serveru nemělo mít vliv na stávající použití aplikace. V případě referenčního řešení ve Swingu generuje klient JPanel, který lze vkládat do dalších panelů a vývojář tak není nikterak omezen, co se týče stávající aplikace.



Obrázek 3.2: Životní cyklus formuláře

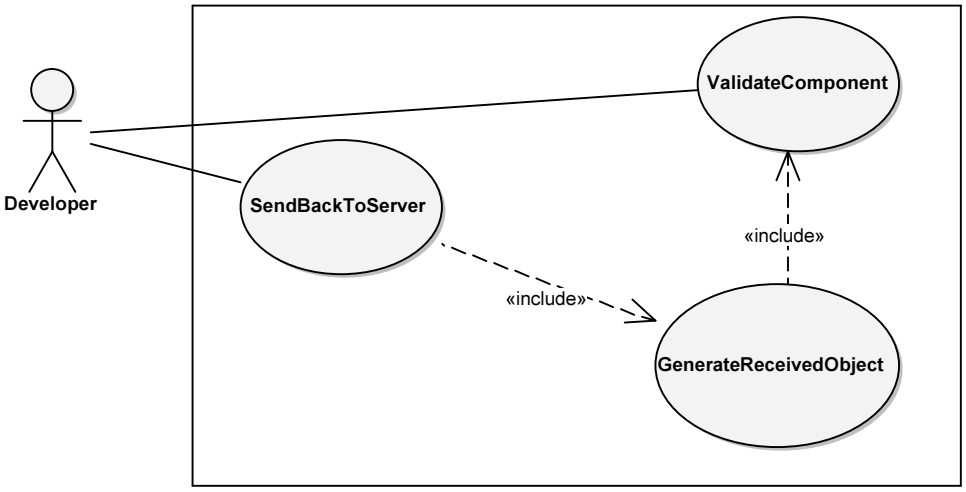
3.2.4 Životní cyklus formuláře

Formulář, jako každá komponenta, má svůj životní cyklus. Jeho stavy jsou znázorněny na obrázku 3.2. Formulář je po vygenerování a po naplnění daty v inicializačním stavu. V tomto stavu může být komponenta nevalidní, neboť data, která obdržela, nemusí splňovat požadované validace. K této situaci může dojít, je-li například přidána nová proměnná do datového modelu. Toto přidání obvykle probíhá tak, že se v koncové databázi, pokud jí software má, přidá políčko a nastaví se mu defaultní hodnota pro již existující data, která může být například null. V definici může být pole označeno jako povinné, avšak ve formuláři nemusí být vyplněno, což způsobí, že jsou data nevalidní, byť uživatel ve formuláři data nezměnil. Komponenta se pak přepne do stavu Inconsistent. V případě modifikace dat se komponenta dostává do stavu Detached. Tento stav značí, že byla data změněná. Pokud uživatel data stále mění, pak komponenta zůstává v tomto stavu. Ze stavu Detached přechází komponenta

v případě úspěšné validace do stavu Consistent. V případě neúspěšné validace se komponenta dostane do stavu Inconsistent. Z tohoto stavu lze přejít pouze do dvou stavů jedním z nich je stav Detached. Do tohoto stavu lze přejít, pokud uživatel změní data ve formuláři. Komponenta může také zůstat ve stavu Inconsistent, pokud uživatel data nezmění a zkusí provést validaci znovu a tato validace opět selže. Konzistentní stav značí, že je komponenta připravená k odeslání dat na server. Pokud odeslání dat selže je komponenta přepnuta do stavu Inconsistent. V případě úspěšného odeslání dat, je komponenta přepnuta do stavu Synchronized. Ze stavu Synchronized lze přejít do stavu Initialized, pokud jsou data znovu načtena ze serveru a do stavu Detached, pokud jsou data upravena uživatelem.

3.3 Případy užití

Případy užití a jejich scénáře [16] specifikují chování systému. V této práci lze nahlížet na případy užití ze dvou stran. První z nich je koncový uživatel, neboli vývojář, který framework využívá. Druhým z nich je samotný framework, který provádí akce, aby splnil úkol, který mu uživatel uložil. V této sekci se zaměříme na případy užití koncového uživatele, které specifikují použití frameworku. Na obrázku C.1 jsou zachyceny všechny tyto případy užití. Pro ukázku detailně rozebereme případ užití na obrázku 3.3. Na tomto případu je znázorněno odeslání dat z vygenerovaného formuláře zpět na server. Součástí je samozřejmě validace zadaných dat a jejich zpětné sestavení, neboť formulář byl vytvořen na základě metadat a klient tedy zná pouze strukturu objektu popsanou těmito metadaty.



Obrázek 3.3: Část případů užití znázorňující odeslání dat na server z vygenerovaného formuláře

3.3.0.1 Validace komponenty

Tento případ užití je znázorněn na obrázku 3.3 a jmenuje se ValidateComponent.
Případ užití: Validace komponenty
ID: 1
Popis: Uživatel využívá framework ke generování formulářů. Hlavním úkolem formuláře je možnost vkládat či upravovat data a odesílat je zpět na server. Před odesláním dat na server musí být provedena validace, aby se zajistilo, že bude formát dat serveru vyhovovat a bude umět s daty pracovat.
Aktér: Uživatel
Vstupní podmínky:

1. Formulář musí být sestaven na základě metadat a framework musí znát formulář, s kterým chce pracovat.

Scénář:

1. Případ užití začíná obdržením požadavku od uživatele žádající validaci dat.
2. Systém vyhledá pole k validaci.
3. Dokud existují pole k validaci pak:
   (a) Systém získá konkrétní pole k validaci.
   (b) Systém určí typ komponenty a požádá builder, který ji sestavil o data.
   (c) Dokud existuje validace, která zatím nebyla na poli vykonána pak:
      i. Systém požádá validátor o validaci.
      ii. Pokud validace selže pak:
         A. Systém ukončí validování tohoto pole a zobrazí u něj chybové validační hlášení.

Tento případ užití je znázorněn na obrázku 3.3 a jmenuje se GenerateReceivedObject.
Případ užití: Sestavení dat
ID: 2
Popis: Uživatel využívá framework ke generování formulářů. Hlavním úkolem formuláře je možnost vkládat či upravovat data a odesílat je zpět na server. Před odesláním dat na server musí být tyto data zpětně sestaveny, aby s nimi server dokázal pracovat.
Aktér: Uživatel
Vstupní podmínky:

1. Formulář musí být sestaven na základě metadat a framework musí znát formulář, se kterým chce pracovat. Framework musí znát formát dat, která server očekává.

Scénář:

1. Případ užití začíná obdržením požadavku od uživatele žádající sestavení dat z formuláře.

2. Zahrnout(Validace komponenty).
3. Pokud validace dopadla úspěšně pak:
   (a) Systém získá pole, která budou odeslána.
   (b) Pokud existuje pole, které ještě nebylo transformováno pak:
      i. Systém určí typ komponenty a požádá builder, který ji sestavil, o data.
      ii. Systém určí název proměnné a třídu, do které patří, a nastaví jí data.
   (c) Systém na základě formátu dat, které server očekává, rozhodne, v jakém formátu data zaslat a převede je na daný formát.

Výstupní podmínka:

1. Data ve formuláři byla převedena na objekt, s kterým umí server pracovat.

Tento případ užití je znázorněn na obrázku 3.3 a jmenuje se SendBackToServer.
Případ užití: Odeslání dat
ID: 3
Popis: Uživatel využívá framework ke generování formulářů. Hlavním úkolem formuláře je možnost vkládat či upravovat data a odesílat je zpět na server. Před odesláním dat na server musí být tato data zpětně sestavena, aby s nimi server dokázal pracovat, a musí splnit validační kritéria.
Aktér: Uživatel
Vstupní podmínky:

1. Formulář musí být sestaven na základě metadat a framework musí znát formulář, se kterým chce pracovat. Framework musí znát zdroj, na který mají být data odeslána, a všechny potřebné informace, které zdroj vyžaduje.

Scénář:

1. Případ užití začíná obdržením požadavku od uživatele žádající odeslání formuláře na server.
2. Zahrnout(Validace komponenty).
3. Zahrnout(Sestavení dat)
4. Dokud je validace či sestavení dat neúspěšné pak:
   (a) Systém zobrazí chybové hlášení a určí, u kterých polí nastala chyba.
   (b) Uživatel chybu opraví a požádá systém o opětovnou validaci a sestavení dat.
5. Systém vytvoří připojení na specifikovaný zdroj a odešle data.
6. Pokud odeslání selhalo pak:
   (a) Systém zobrazí chybové hlášení, že nebylo možné data odeslat včetně odpovědi od serveru, je-li nějaká.

Výstupní podmínka:

1. Data byla odeslána.

3.4 Omezení frameworku

Existují určité možnosti, které nebudou ve frameworku podporovány. V následujícím přehledu budou představeny nepodporované vlastnosti frameworku v aktuální verzi.

1. Inspekce datových proměnných typu List a Array
2. Získávání dat ve formátu XML. Framework plně podporuje JSON.
3. Customizace jednotlivých polí. Framework podporuje customizaci formuláře a všech jeho polí, nicméně nedisponuje možností přizpůsobovat jednotlivá pole.
4. Framework neumožňuje v jednom poli reprezentovat složený datový typ. Například třídu.
5. Odesílání dat a jejich validace z tabulky. Tabulka je v této verzi pouze readonly.
6. Framework vyžaduje ke své funkčnosti jak serverovou tak klientskou stranu. V případě, že klientské straně chybí serverová strana, tak je framework nefunkční, pokud však serverové straně chybí klientská strana, pak je serverová strana stále schopná vytvářet definice dat.
7. Klientská strana zobrazuje pouze ta data, která obdržela od serveru, nelze vytvářet automatický Mashup na klientovi. Nicméně klient může generovaný formulář umístit mezi jiné komponenty.
8. Klientská strana nedokáže sestavit objekt, který získala z metadat do takové míry aby z něj byla schopná vytvořit instanci. Neboli klientská strana si neudržuje konkrétní objekt, který obdržela, ale pouze jeho popis.
9. Serverová část využívá k automatické inspekci framework. Bez tohoto frameworku není možné inspekci provést, nicméně uživatel si může definovat svou vlastní definici bez nutnosti inspekce dat.

3.5 Uživatelé a zabezpečení

Téměř každá aplikace využívá způsob, při kterém se uživatel autentizuje a aplikace mu na základě jeho rolí přidělí oprávnění. V případě využití bez stavového protokolu, jakým REST je, lze posílat informace o uživateli v hlavičce požadavku. Tyto informace mohou být samozřejmě zašifrované. Framework podporuje vkládání libovolných informací do hlavičky requestu. Klient si také může zvolit, zdali bude využívat http či https protokol. Velmi rozšířenou možností je využitý Oauth. Jednou z možností je vložení parametrů do hlavičky, či do adresy. Vkládání dynamických adres či proměnných do hlavičky requestu framework podporuje.

Druhou částí jsou uživatelské role, na základě kterých se generuje uživatelské rozhraní. Serverová část využívá framework AspectFaces [1], který podporuje uživatelské role v systému. Je jen na programátorovi, jaký framework na autentizaci a autorizaci na straně serveru použije. Jednou z možností je například napsat si vlastni interceptor, který určí, o jakého uživatele se jedná, přiřadí mu roli v systému, na základě které se mu zobrazí konkrétní obsah. Server při generování metadat může využít různé mapovací soubory na základě uživatelské role. Specifikace tohoto chování je opět v plné kompetenci uživatele.

3.6 Použité technologie

V následující sekci jsou rozebrány použité technologie. Kromě samotného frameworku je součástí práce i ukázkový projekt na platformě JavaEE a JavaSE.

3.6.1 Java SE - Swing

Klientská část frameworku je schopná vygenerovat formuláře nebo tabulky, naplnit je daty a data odeslat. Klientská část je přizpůsobena frameworku Swing [14]. Důvodem je, že vývoj Swingové aplikace je rychlý a Swing zná velké množství vývojářů, kteří si framework mohou jednoduše otestovat. Nicméně metadata, která server generuje lze interpretovat v jakékoliv technologii. Swing je knihovna grafických a uživatelských prvků. Poskytuje komponenty, layouty, actionListenery, okna, dialogová okna a další prvky, pomocí kterých lze vytvářet interaktivní aplikace.

3.6.2 Java EE

Java EE je platforma sloužící k vývoji enterprise aplikací. V současné době je oblíbená jak u velkých tak u menších korporací. Java EE přináší podporu pro Restové služby, JFS, JSP, EJB, databázové frameworky, anotace a další komponenty. Aplikace v JavaEE se obvykle nasazuje na aplikační server. Aplikační servery mohou být v cloudu a podílet se společně na zpracování requestů. Důvodem využití této platformy je fakt, že klient vytváří requesty vůči serveru a server tato data zpracovává. Je vhodné mít na serveru platformu, která je ověřená a má potenciál ke zpracování těchto requestů. V této práci generujeme data pomocí restového rozhraní a Java EE splňuje specifikaci, které se tohoto rozhraní týká.

3.6.3 AspectFaces

AspectFaces je framework, který umí provádět inspekci nad zadanými objekty a na základě datových typů a dalších parametrů rozhodovat o tom, jaká komponenta se použije pro konkrétní datový typ. Framework využívá AspectFaces k tomu, aby provedl toto mapování a sestavil popis uživatelského rozhraní. Hlavním důvodem využití tohoto frameworku je fakt, že je distribuován jako open source pod licencí LGPL v3 a lze využít jeho funkcionalitu ke statické inspekci dat. Tato inspekce je již odladěna a není tedy důvod psát znovu již vynalezenou věc.

3.6.4 Ukázkový projekt

Ukázkový projekt demonstruje použití frameworku. Skládá se z klientské a serverové části. Klientská část využívá pouze Swing. Serverová část je mnohem sofistikovanější a využívá aktuální technologie. Ukázkový projekt zde znázorňuje použití frameworku a jeho omezení. Ukázkový projekt je koncipován tak, aby ho bylo možné nasadit bez nutnosti dodatečného nastavení.

3.6.4.1 GlassFish

GlassFish [3] je open source aplikační server. Jedná se o certifikovaný server JavaEE, umožňující clustering, monitoring, podporu EJB, REST a JDBC. Architektura jádra je založena na frameworku OSGI, který umožňuje vzdáleně přidávat, startovat či ukončovat komponenty bez nutnosti restartování celého serveru. Důvodem, proč je Glassfish využit na tomto projektu je čistě demonstrativní. Nicméně každý aplikační server má svá specifika, a proto je ukázková aplikace odladěna právě pro GlassFish. Jak již bylo zmíněno, Glassfish distribuje vestavěnou podporou pro REST a umožňuje použití Derby DB v režimu in-memory bez nutnosti speciálního nastavení.

3.6.4.2 RestEasy

Jedná se o framework, pomocí kterého lze vytvářet RESTful aplikace. Tento framework může běžet v libovolném servletovém kontejneru. Framework podporuje například JSON, XML serializace objektů, EJB a je splňuje JAX-RS implementaci. Tvorba aplikací s restovým rozhranním je tak díky tomuto frameworku mnohem jednodušší a vývoj je rychlejší.

3.6.4.3 EJB

Enterprise JavaBeans [21] jsou serverově orientované komponenty, které zapouzdřují business logiku a přístup do databáze. Jsou spravovány v rámci serverového kontejneru, který zajišťuje jejich vytvoření i odstranění z paměti. EJB mohou být různých typů.

- Stateless
- Statefull
- Singleton

Jak již bylo zmíněno, o jejich správu se stará serverový kontejner, nemusíme tedy řešit problémy spojené s vytvořením a destrukcí singletonu [23]. Mezi hlavní výhody EJB patří transakční zpracování, zajištění systémových služeb a bezpečnostní autorizace. Abychom definovali či získali přístup k těmto třídám, používáme anotace, které jsou velmi dobře čitelné a srozumitelné.

3.6.5 Derby DB

V ukázkovém projektu je potřeba data ukládat do databáze. Při vytváření byl kladen důraz na to, aby noví uživatelé nemuseli v konfiguračních souborech specifikovat nastavení, tudíž by mohli ukázkový projekt ihned nasadit a vyzkoušet. Z tohoto důvodu je využita light databáze Derby. Ukázkový projekt ji využívá v in-memory módu, což znamená, že budou data po zastavení serveru ztracena. Spolu s Derby DB využívá ukázkový projekt ORM s defaultním nastavením na create. Po startu aplikace jsou v čisté databázi vytvořeny požadované tabulky, které odráží definice objektů, jenž jsou anotované jako entity. Další výhodou je možnost využít anotací k nastavení validací přímo na databázi. Tyto validace pak mohou být využity při inspekci dat a na jejich základě mohou být vytvořeny validace, či konkrétní komponenty.

Kapitola 4

Implementace

4.1 Architektura

V tomto frameworku [2] rozlišujeme klientskou a serverovou část. Serverová část generuje data pro klienta a tímto způsobem ovlivňuje ovládací prvky, které klientská část aplikace zobrazuje uživateli. Diagram nasazení na obrázku C.3 zachycuje použití frameworku. Serverová část frameworku je nasazena na serveru a je schopná generovat definice formulářů s použitím frameworku AspectFaces [1]. Tyto definice jsou převedeny na model, který je možné upravit a odeslat klientovi. Aby byla tato část plně funkční, je potřeba nasadit aplikaci na aplikační server na platformě Java EE. Nicméně v případě využití pouze staticky generovaných definic lze aplikaci nasadit na libovolný aplikační server, který bude poskytovat klientům definice kompatibilní s definici poskytovanými při dynamickém generování. Specifikace komponenty, je poté zaslána na klienta, který ji interpretuje za použití klientské části zvané AFSwinx. Tato část využívá i serverovou část, a to z důvodu kompatibilnosti objektů a jejich vlastností. Přidání do projektu lze provést tak, že se do adresáře s knihovnami, obvykle bývá nazván lib, vloží přeložený jar soubor nebo se přidá projekt jako Maven závislost. Maven [9] je framework pro správu projektů, sloužící k buildu aplikací, dokumentací a spouštění testů. Pomocí Mavenu lze do projektu vkládat závislosti na dalších knihovnách. V současné době není framework k dispozici v centrálním repositáři, a proto je potřeba stáhnout aktuální verzi a zkompilovat ji do lokálního repositáře.

4.1.1 Server

Jak již bylo zmíněno, server využívá ke generování serverovou část frameworku nazvanou AFRest. Na obrázku C.4 jsou zobrazeny třídy a balíčky, které tato část využívá. Jsou zde výčtové typy, které určují podporované komponenty a jejich vlastnosti, dále objekty zodpovědné za informace o volbě layoutu a objekty nesoucí informace o definicích, na jejichž základě budou sestaveny formuláře či tabulky klientem a samozřejmě třídy zodpovědné za inspekci dat. Framework doplňuje do AspectFaces několik anotací, které lze využít při generování definic. Jsou to následující anotace:

1. @UIWidgeType - tato anotace určuje typ widgetu, který bude využit. Do XML šablon, které se používají při generování definic, je propagována jako proměnná s názvem widgetType

2. @UILayout - tato anotace definuje layout na dané proměnné. Lze specifikovat typ layoutu, jeho orientace a pozice popisu prvku. Do XML šablon jsou tyto hodnoty propagovány jako proměnné layout, layoutOrientation a labelPossition.

Výše zmíněné anotace akceptují pouze hodnoty z výčtových typů v balíčku common. V případě typu komponenty, neboli widgetType, přijímá anotace hodnoty ze třídy SupportedWidgets. V případě anotace určující layout lze vložit pouze hodnoty z výčtových typů LayoutDefinitions, LayoutOrientation a LabelPosition. Hlavní výhodou tohoto řešení, je typová kontrola a jistota, že klient obdrží od serveru pouze takové hodnoty, s kterými je schopný pracovat. Stejným principem jsou řešeny validace a proměnné, které definují vlastnosti jednotlivých komponent.

4.1.1.1 Generování modelu

Výsledkem inspekce objektu je model, který nese informace potřebné k tomu, aby klient mohl sestavit formulář či tabulku a byl do těchto komponent schopný vložit data získaná ze serveru. Na obrázku C.5 je konečná podoba modelu vytvořeného k tomuto účelu. Model je výsledkem hledání analytických tříd z doménového modelu na obrázku 3.1. Tento model byl popsán již v analytické části, nicméně v této části je model kompletní, a proto zde budou uvedeny pouze změny oproti původnímu modelu. Proměnné, které nesou informace o layoutu, typu komponenty validátorech a jejich typech jsou výčtové typy. Jak již bylo zmíněno výhodou je typová bezpečnost a jednoznačnost vlastností, které framework podporuje. Model slouží také jako fasáda k nastavení dodatečných atributů. Jedním z těchto atributů je proměnná options ve třídě AFFieldInfo. Tato proměnná drží informace o možných hodnotách, kterých může komponenta nabývat. V současné verzi je tento atribut využit u komponent výběrového typu, mezi které patří například zaškrtávací políčka či výběrová menu. Programátor specifikuje množinu těchto hodnot, ve které klíč určuje hodnotu, jež bude odeslána na server. Text, který bude zobrazen uživateli je určen proměnnou value. Tyto možnosti nejsou generovány automaticky a v případě potřeby je musí programátor specifikovat ručně. A to tak, že určí množinu dat a pole, ke kterému je přiřazeno. Třída AFMetaModelPack zapouzdřuje způsob jakým se množina dat nastaví na konkrétní políčko a nabízí uživateli funkci, která je schopná nastavení provést na základě dat, zadaných uživatelem.

K dynamickému generování definic se využívá framework AspectFaces [1], který umožňuje na základě mapování rozhodnout, jaká komponenta bude použita pro konkrétní proměnnou dané třídy. Dále nabízí určení layoutu, jež bude použit, a samozřejmě určení mapovacího souboru. Tímto lze docílit mnoha různých transformací. Tento framework je potřeba nejprve nastavit, nicméně toto nastavení provede za vývojáře serverová část frameworku AFRest. Rozhraní AFRest z obrázku C.5 a jeho implementace AFRestGenerator provedou kompletní nastavení a spustí generování dat. Rozhraní umožňuje uživateli určit mapovací soubor a šablonu, která bude použita. Mapování lze použít na všechny proměnné objektu, či může vývojář určit, které mapování se použije na konkrétní proměnnou. Ukázka mapování z frameworku AspectFaces je znázorněna v ukázce zdrojových kódu 4.1. Proměnná typu String se bude mapovat na vstupní textové pole, které je definováno v structure/inputField.xml. V případě, že se bude jednat o typ password, bude se proměnná typu String mapovat na vstupní textové pole typu, které místo vepsaných znaků zobrazuje zástupné znaky. Komponenta je defino-

vána v structure/inputPassword.xml. Typ Address, což je neprimitivní datový typ se bude mapovat na entitní typ, jehož definice je v structure/entity.xml.

Část zdrojového kódu 4.1: Ukázka mapování proměnných na komponenty

```
<mapping>
        <type>String</type>
        <default tag="structure/inputField.xml" maxLength="255"/>
        <condition expression="${type == 'password'}" tag="structure/inputPassword.xml" />
</mapping>
<mapping>
        <type>Address</type>
        <default tag="structure/entity.xml" />
</mapping>
```

Mapování tedy určí soubor s komponentou, který bude reprezentovat aktuální proměnnou. Soubor s definicí komponenty je pak dále využit k finální definici proměnné. Ukázka vstupního textového pole je v ukázce zdrojových kódu 4.2. Komponenta je v kořenovém elementu widget. Jelikož se jedná pouze o fragment xml, který je použit ke složení celé definice, jenž je uvedena v příloze v ukázce zdrojových kódů D.1, není zde uvedena deklarace XML [25]. Ve výsledném XML již však deklarace uvedena je. Popis jednotlivých uzlů je v tabulce 4.1.

Část zdrojového kódu 4.2: Ukázka definice komponenty

```
<widget>
        <widgetType>textField</widgetType>
        <fieldName>$field$</fieldName>
        <label>$label$</label>
        <validations>
                <required>$required$</required>
                <minLength>$minLength$</minLength>
                <maxLength>$maxLength$</maxLength>
        </validations>
        <fieldLayout>
                <layoutOrientation>$layoutOrientation$</layoutOrientation>
                <labelPossition>$labelPossition$</labelPossition>
                <layout>$layout$</layout>
        </fieldLayout>
</widget>
```

Knihovna AspectFaces umožňuje určovat způsob, jakým bude prováděna inspekce. Tento způsob se určuje v šablonách. K optimálnímu využití je nejvýhodnější použít způsob, při kterém je provedena inspekce všech proměnných, které mají definováno mapování. V případě jednoduchých datových typů je vše v pořádku. Nicméně knihovna neobsahovala nativní podporu pro neprimitivních datových typů a v případě, že byla použita inspekce, která by nevyužívala JSF, nebyla by provedena inspekce složitých datových typů. Z tohoto důvodu je důležité, aby se všechny neprimitivní datové typy mapovali na entity.xml, která je znázorněna v části zdrojového kódu 4.3. Framework totiž pro všechny tyto entity provede inspekci znovu, následně části sestaví a tím vznikne kompletní definice. V tomto bodě lze určit mapování a šablony, které se mají při rekurzivní inspekci použít. Framework AspectFaces byl proto doplněn o proměnné, které umí vrátit kanonický název třídy a na základě tohoto názvu lze provést nad touto třídou inspekci. Jak je patrné z výsledné definice, každý uzel má svého

rodiče. Na základě rodiče lze jednoznačně určit, kam uzel patří. Tato vlastnost umožňuje provádět inspekci i nad třídami, které mají více proměnných stejného datového typu. Klient totiž potřebuje znát strukturu objektu, aby ho mohl zpětně sestavit a odeslat zpět na server, který objekt přijme. Znalost struktury klient taktéž vyžaduje v případě získávání dat.

Část zdrojového kódu 4.3: Ukázka definice neprimitivního datového typu

```
<entityClass>
        <entityFieldType>$DataTypeFullClassName$</entityFieldType>
        <fieldName>$fieldName$</fieldName>
</entityClass>
```

Tabulka 4.1: Uzly XML, které definují strukturu dat

Uzel	**Popis**
widget	Typ komponenty. Určuje, jak komponentu bude klient interpretovat.
fieldName	Název aktuální proměnné, kterou komponenta zastupuje.
Label	Popis komponenty, který bude zobrazen uživateli.
validations	Validace, které budou umět komponenta ověřit.
fieldLayout	Popis layoutu, který bude na komponentě použit.

4.1.1.2 Použití

Aby byl klient schopný získat definice dat, musí serverová strana poskytovat zdroj těchto definic. V tomto zdroji server využije serverovou část frameworku ke generování dat. Použití je přímočaré a ukázka zdroje je zobrazena v části zdrojového kódu D.2. Nejprve je vytvořena instance třídy AFrestGenerator, která umožňuje generování dat, jenž jsou následně odeslána klientovi. Generátor nastaví framework AspectFaces automaticky, nicméně očekává, že bude framework AspectFaces použit. Ke správnému použití je potřeba, aby ve WEB-INF byly konfigurační soubory a aby existovali mapovací soubory a definice komponent. V tomto případě využije implicitního nastavení pro mapování i šablony. Bude použito mapování v souboru structure.config.xml a šablona v template/structure.xml. Tyto ukázkové soubory jsou poskytovány spolu s frameworkem.

Zdroj poskytuje definice dat. V případě, že klient požaduje data do vygenerované definice, je potřeba poskytnout mu objekt stejného typu, nad kterým byla prováděna definice, nebo objekt se stejnými proměnnými a datovými typy. V tomto případě třídu Country. Klientská strana nerozlišuje datový typ obdrženého objektu, avšak očekává, že objekt bude mít určité proměnné, ke kterým se budou vázat specifické validace. V některých případech je žádoucí, aby se na úrovni business a view nepracovalo s databázovou entitou, ale s jejím mapovacím objektem. V části zdrojových kódů 4.4 je příklad získání dat do již vygenerovaného formuláře či tabulky. Zdroj využije EJB [7] managera CountryManager k získání

konkrétní instance třídy Country z databáze. Tuto instanci vrátí klientovi. Tento zdroj nemá žádnou vazbu na předchozí zdroj, který generoval definice. Vývojář tedy v případě použití frameworku nemusí měnit stávající implementaci, pokud již nějaká existuje. Stejně tak nemá použití frameworku dopad na klienty, kteří již používají webové API serveru. Zodpovědnost za správnou interpretaci dat je na klientské straně.

Část zdrojového kódu 4.4: Zdroj poskytující konkrétní instanci třídy Country

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
public Response getCountry(@PathParam("id") int id) {
        try {
                CountryManager<Country> countryManager = getCountryManager();
                Country country = countryManager.findById(id);
                return Response.status(Response.Status.OK).entity(country).build();
        } catch (BusinessException e) {
                return Response.status(Response.Status.BAD_REQUEST).build();
        } catch (NamingException e) {
                return Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
        }
}
```

4.1.2 Klient

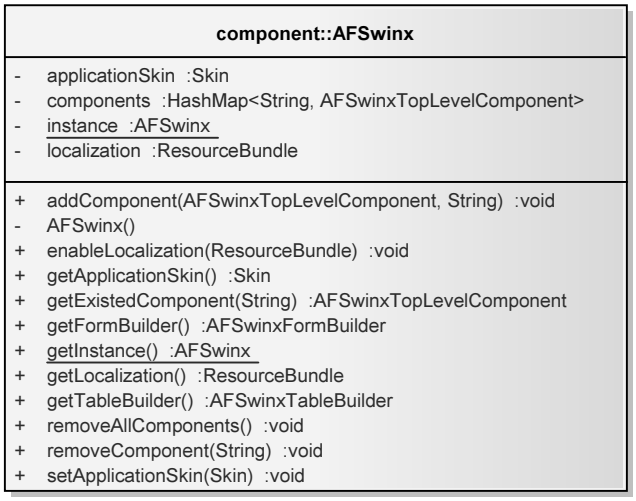
Klientská část aplikace, využívá klientskou část frameworku ke generování formulářů či tabulek. Definice a data získává ze serveru. Referenční implementace je napsána pro aplikace na platformě JavaSE, které získávají data ze serveru a k vizualizaci využívají technologie Swing. Integrace frameworku do klientské aplikace je možná dvěma způsoby. Prvním z nich je vložení knihovny do složky lib a druhým je přidání Maven [9] závislosti.

4.1.2.1 Komponenty

Klientská část umožňuje generovat tabulky nebo formuláře. Tyto celky označujeme jako komponenty. V případě formuláře se tato komponenta skládá z dalších aktivních ovládacích prvků. Komponenty jsou děděny z třídy AFSwinxTopLevelComponent, která implementuje rozhraní AFSwinxInteraction, který je zobrazen na obrázku C.9, jež vynucuje implementovat metody k získání modelu dat a k jejich odeslání zpět na server. Součástí je také validace dat. Mimo výše zmíněné rozhraní implementuje třída ještě rozhraní ComponentResealization. Toto rozhraní je využito k zpětnému získání dat z komponent. Aby bylo možné přidávat komponenty do již existující aplikace, tak tato komponenta ještě dědí od třídy JPanel, což zajistí, že výslednou komponentu lze přidat na jakékoliv místo ve stávající Swingové aplikaci. Vývojář může nad takto generovanými komponentami provádět operace. V případě odeslání dat na server lze tuto akci vyvolat metodou sendData. Komponenta již sama provede validaci dat, sestavení dat a jejich odeslání.

Při návrhu jsem se zaměřil i na použitelnost, neboť je potřeba aby framework umožňoval dodatečná nastavení. Nicméně pokud se vývojář bude s frameworkem učit, je velice pravděpodobné, že bude chtít vytvořit první prototyp, aby si vyzkoušel funkčnost. Z tohoto

důvodu byla zavedena třída AFSwinx, která slouží jako správce komponent. Je znázorněna na obrázku 4.1. Třída je singleton [23] a slouží jako fasáda [23]pro ovládání frameworku. Umožňuje komponenty vytvářet, přidávat, mazat a nastavovat globální skin a lokalizace. Důležitou součástí je i získání již sestavené komponenty. Každá komponenta je jednoznačně určena svým identifikátorem, který si vývojář zvolí. Na základě tohoto identifikátoru je zaregistrována a lze k ní získat přístup a provádět nad ní operace. Vzhled jednotlivých prvků v komponentě již není možné po vygenerování měnit. Skiny a lokalizace musí být tedy nastaveny před samotným vygenerováním. Také je potřeba určit způsoby připojení ke zdrojům a jejich URI. Proces vytváření komponent vyžaduje několik operací, které na sebe navazují. V případě, že by byl tento proces ponechán na vývojáři, byl by framework nepoužitelný. Z tohoto důvodu poskytuje třída AFSwinx buildery [23] pro tabulky a formuláře, které komponenty sestaví, vloží do nich data a vývojáři vrátí výsledný JPanel. Typ komponenty určuje vývojář a buildery umí vytvořit formulář či tabulku na základě jedné definice. V případě tabulky je možné ještě provést dodatečné nastavení. Jedná se o automatické nastavení šířky sloupečků a nastavení velikosti tabulky.



Obrázek 4.1: Třída AFSwinx, která uchovává všechny aktuálně vygenerované komponenty.

Ukázka vytvoření formuláře je zobrazena v části zdrojových kódů 4.5. Nejprve je potřeba získat instanci builderu, který bude použit. Typ builderu určí, zdali bude vytvořena tabulka či formulář. V tomto konkrétním případě bude vytvořen formulář. Metamodel získává klient ze serveru. Framework zapouzdřuje způsob získání dat, vývojář tedy musí definovat pouze zdroje. Jednou z možností je specifikovat zdroje jako samostatné objekty, druhou možností je využít XML. V případě použití XML souboru musí být uveden soubor a identifikátor připojení. Tyto vlastnosti jsou nastaveny builderu pomocí metody initBuilder, která očekává identifikátor formuláře, soubor se specifikací připojení a identifikátor připojení. Builder má samozřejmě několik přetížených metod initBuilder. Tímto lze docílit různých způsobů počátečního nastavení. Metoda buildComponent již vygeneruje výsledný formulář. Pokud se při generování vyskytne chyba, pak je vyhozena výjimka AFSwinxBuildException. Je na vývo-

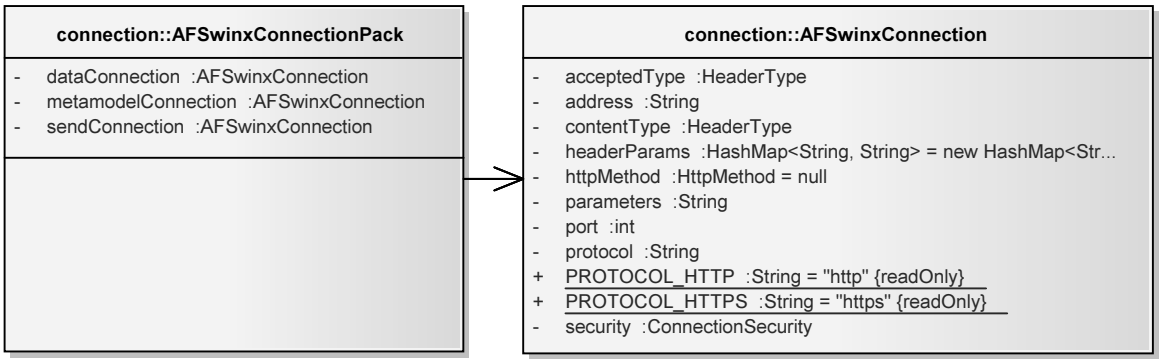
jáři jak výjimku zpracuje. V ukázce 4.5 je zobrazen dialog s chybovým hlášením.

Část zdrojového kódu 4.5: Generování formuláře na klientovi

```
InputStream connectionResrouce = getClass().getClassLoader().getResourceAsStream("connection.xml");
try {
        AFSwinxForm form =
                AFSwinx.getInstance().getFormBuilder()
                .initBuilder(loginFormName, connectionResrouce, "loginForm")
                .buildComponent();
} catch (AFSwinxBuildException e) {
        getDialogs().failed("afswinx.build.title.failed", "afswinx.build.text.failed", e.getMessage());
}
```

4.2 Přenos modelu server klient a generování komponent

Model, na jehož základě jsou generovány komponenty, je přenášen ze serveru na klienta. Klient musí tento model správně zpracovat a interpretovat. Nejprve je však potřeba model získat. Pro přenos modelu je použit protokol HTTP či HTTPS. Klientská strana poskytuje vývojáři nativní podporu k získání dat ze serveru. K tomuto účelu je využit framework HttpComponents [24], který poskytuje předpřipravené komponenty, jež lze využít k vytváření HTTP či HTTPS požadavků. Využitím této knihovny se zjednoduší použití našeho frameworku, neboť vývojář nemusí ztrácet čas vytvářením tříd, které by byly schopné získat data ze serveru. Bohužel použití má i své nevýhody. Vývojář nemůže ovlivnit implementaci získání dat, pouze může ovlivnit způsob a to změnou ve specifikaci zdrojů a způsobů připojení. Z tohoto důvodu lze specifikovat zdroje ve formátu XML s podporou EL. K získání modelu je potřeba mít definovaný zdroj v uzlu metamodel. Uzel se specifikací konkrétních dat, a umístění, kam data odeslat je nepovinný. Zpracování pomocí DOM parseru je však provedeno nad všemi uzly konkrétního připojení a výsledkem je třída AFSwinxConnectionPack, která má reference na konkrétní připojení reprzentovanou třídou AFSwinxConnection. Ukázka je na obrázku 4.2. Mimo adresy, portu a protokolu lze specifikovat i hlavičku a v případě zabezpečení zdroje autorizaci k tomuto zdroji.



Obrázek 4.2: Třídy zodpovědné za specifikaci zdrojů a způsobu připojení.

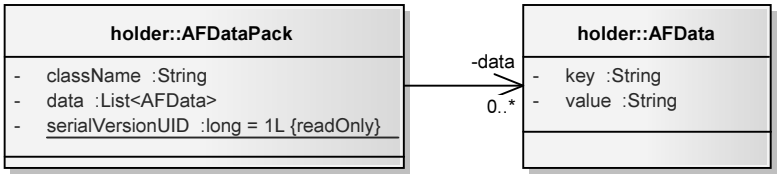
4.2.1 Generování komponent

Vývojář na základě builderu určí, jaká komponenta se bude generovat. Sekvenční diagram je na obrázku C.6. Z diagramu je patrné, že builder nejprve získá model ze serveru na základě specifikace zdroje, který mu byl předán během inicializace. Od serveru získá klient třídy reprezentující metamodel. Tento metamodel byl již popsán a je na obrázku C.5. Klient nyní začne rekurzivně vytvářet již konkrétní aktivní prvky. Pro každou proměnnou objektu bude vygenerován widget. Pokud se jedná o neprimitivní datový typ, pak se k příslušnému typu vyhledá jeho reprezentace v metamodelu a generování bude pokračovat tímto objektem. Takovýto přístup zajišťuje zachování pořadí proměnných, které lze měnit na serverové straně, nikoliv však na klientovi. Typ aktivního prvku určuje atribut widgetType, který je součástí každého popisu konkrétní proměnné. Formulářový builder si nechá vytvořit od tovární třídy WidgetBuilderFactory, která je zodpovědná za vytváření konkrétního builderu, builder jenž je schopný vytvořit požadovaný aktivní prvek. Tento builder vrátí již konkrétní komponenty, jako jsou například vstupní textová pole, zaškrtávací políčka a další. Před kompletním generováním aktivního prvku lze nastavit jazykové lokalizace a skin. Tyto aktivní komponenty jsou zapouzdřeny v objektu AFSwinxPanel. Důvodem je, to že tento panel již zohledňuje layout dané komponenty a mimo aktivní prvek obsahuje i popis, placeholder určený k zobrazení validačního hlášení, a všechny validace, které musí být nad tímto prvkem vykonány. Panel si také udržuje jednoznačný identifikátor v rámci formuláře, na základě kterého lze určit, jakou proměnnou prvek reprezentuje a její umístění v hierarchii tříd. Panel je následně přidán do dalšího panelu, který udržuje všechny prvky formuláře. V rámci tohoto formulářového panelu jsou také zohledněny layouty a uspořádání komponent. V tomto bodě je formulář sestaven a lze již nad ním provádět validace, či je možné formulář odeslat zpět na server. Nyní je potřeba rozhodnout, zdali by měli být ve formuláři zobrazeny data, či nikoliv. Formulářový builder vyhodnotí, má-li naplnit formulář daty a to na základě specifikovaných zdrojů. Pokud byl zdroj s daty specifikován, pak jsou data získána a automaticky vložena do komponenty, jinak se formulář nemodifikuje a práce builderu je ukončena.

4.2.1.1 Vkládání dat do komponenty

Data vkládá do komponenty builder, který tuto komponentu vytvořil. Komponenta, kterou builder vytváří disponuje funkcionalitou, umožňující získat data ze serveru. O datovém objektu, který server poskytuje, nemá komponenta předem žádné informace. Proto je tento objekt po obdržení převeden na třídu AFDataPack. Hierarchie je zobrazena na obrázku 4.3. Klíč určuje umístění proměnné v hierarchii a je v něm použita standardní tečková notace. Například mějme třídu Person, která má referenci na třídu Address přes proměnou myAdress a ve třídě Address je textová proměnná city. Pokud je třída Person první v hierarchii, je nahrazena zástupnou hodnotou root. Klíč k proměnné city je pak následující: root.myAdress.city. Stejným způsobem byly vygenerovány klíče pro konkrétní komponenty formuláře, jenž byly sestaveny na základě metadat. Formulář či tabulka disponují komponentami, které mají klíče kompatibilní s klíči vygenerovanými z obdržených dat. Lze je tedy spolu spárovat. Problémem však je, že každá z komponent je jiná a byla sestavena specifickým builderem. Nicméně builder zná tento způsob a proto mu byla přidána funkcionalita, na základě které lze upravit současný model a vložit data do již existující komponenty. Typ

builderu je určen na základě widgetType. Toto je proměnná, kterou mají komponenty, jenž byly vytvořeny buildery. Data jsou vkládána do každé komponenty, která byla vytvořena.



Obrázek 4.3: Třídy, na které jsou převedena všechna data, jenž klient obdrží.

4.2.1.2 Widget builder

V tabulce 4.2 jsou popsány všechny widget buildery, které je možné použít. Všechny buildery mají společného předka. Abstraktní reprezentace buildera je znázorněna na obrázku C.7. Konkrétní instance pak využívá společných metod, jež jsou implementovány v jeho předkovi. Předek umí sestavit placeholder určený k zobrazení výsledku validací, popis ke každé komponentě, nastavit lokalizace, skiny včetně jejich aplikace a přidat validátory na pole. Tato abstraktní třída také umí vygenerovat dummy field, který je vytvořen pouze pokud komponenta dostala list hodnot jichž může nabývat a současně není volba z tohoto listu povinná. Pak je zapotřebí udržovat informaci o tom, že uživatel volbu neučinil. Konkrétní prvky builderu z tabulky 4.2 pak vytváří každý builder sám.

Část zdrojového kódu 4.6: Vytváření vstupního pole builderem.

```
public AFSwinxPanel buildComponent(AFFieldInfo field)
throws IllegalArgumentException, AFSwinxBuildException {
        super.buildBase(field);
        // And input text field
        JTextField textField = new JTextField();
        customizeComponent(textField,field);
        layoutBuilder.addComponent(textField);
        coreComponent = textField;
        // Create panel which holds all necessary informations
        AFSwinxPanel afPanel =
                new AFSwinxPanel(field.getId(), field.getWidgetType(), textField, fieldLabel,
                message);
        // Build layout on that panel
        layoutBuilder.buildLayout(afPanel);
        // Add validations
        super.crateValidators(afPanel, field);
        return afPanel;
}
```

Ukázka metody, která vygeneruje vstupní textové pole je znázorněna v části zdrojových kódu 4.6. Nejprve jsou vytvořeny společné vlastnosti pro všechny buildery. Tyto vlastnosti vytváří abstraktní předek. Poté je vytvořeno vstupné pole a na toto pole je aplikován skin. Komponenta je poté přidána do layout builderu. Následně je vytvořen AFSwinxPanel, jenž

nese všechny nezbytné informace o aktuální komponentě. Tento panel je také přidán do layout, který poté vytvoří konečné uspořádání komponent. Nakonec jsou v panelu registrovány všechny validátory, které se postupně spustí v případě odeslání dat, či v případě žádosti o zjištění validnosti formuláře.

Část zdrojového kódu 4.7: Vložení dat do vstupního pole vytvořeného builderem.

```
public void setData(AFSwinxPanel panel, AFData data) {
        if (panel.getDataHolder() != null && !panel.getDataHolder().isEmpty()) {
                JTextComponent textField = (JTextComponent) panel.getDataHolder().get(0);
                textField.setText(data.getValue());
        }
}
```

Jak již bylo zmíněno, builder zná způsob, jakým byly komponenty vytvořeny. Z toho vyplývá, že zná i způsob, jakým jsou reprezentovýny. V případě potřeby vložení dat do textového pole, je potřeba získat builder, který toto pole vytvořil a požádat ho o vložení dat. Ukázka je v části zdrojových kódů 4.7. Builder nejprve ověří, zdali existují v panelu komponenty. Pokud ano přetypuje je na konkrétní instance, které vytvářel. V tomto případě JTextComponent. Poté jim nastaví data specifickým způsobem pro danou komponentu.

Tabulka 4.2: Widget buildery, kterými disponuje klient

Builder	**Typ widgetu**	**Popis**
DateBuilder	Calendar	Používá se při reprezentaci datového typu. Umožní uživateli zobrazit date picker, pomocí kterého lze vybrat datum.
DropDownMenuBuilder	dropDownMenu	Menu, ze kterého lze vybrat jednu z několika voleb.
CheckBoxBuilder	checkBox	Zaškrtávací políčk či několik zaškrtávacích políček. Záleží, zdali jsou uvedeny možnosti. V případě, že uvedeny nejsou, vytvoří se jedno a pokud je zaškrtnuto je převedeno na hodnotu true.
InputBuilder	textField	Builder pro textové pole. Nemá žádné výchozí omezení.
LabelBuider	label	Vypíše pouze textovou hodnotu. Do této komponenty nelze vkládat data ani ji jinak upravovat.
NumberInputBuilder	numberField	Vytvoří vstupní pole a přidá mu číselnou validaci.
OptionBuilder	option	Vytvoří skupinu radiobuttonů, z které lze vybrat jednu hodnotu.
PasswordBuilder	password	Vytvoří vstupní pole, v kterém jsou znaky nahrazeny zástupnými znaky.
TextAreaBuilder	textArea	Vytvoří vstupní pole pro zadání velkého množství znaků.

4.2.1.3 Skin

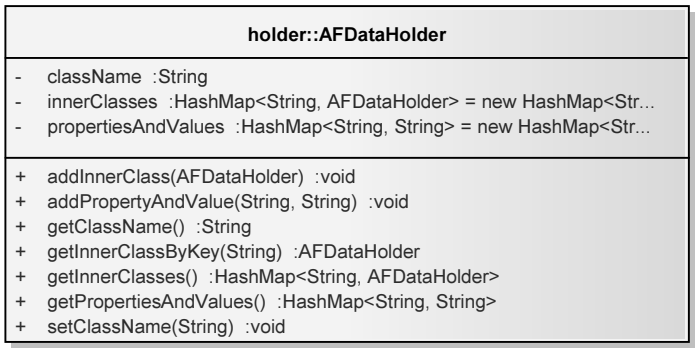
Widget builder aplikuje na vygenerované komponenty skin. Skin lze nastavit již při získávání formulářového builderu. Skin určuje vzhled konkrétní komponenty. Pomocí skinu lze určit následující vlastnosti.

1. Barvu, typ fontu, výšku a šířku popisu, který je zobrazen u komponenty.
2. Barvu a typ fontu komponent.
3. Barvu a typ fontu validačních hlášení.
4. Šířku komponent. V případě textových polí i jejich výšku.

Pokud není skin nastaven, potom je použita výchozí implementace, která je součástí frameworku. Vývojář si může definovat vlastní skin a to tak, že buď implementuje rozhraní Skin, nebo využije dědičnost a překryje metody z třídy BaseSkin. Výhodou druhého přístupu je fakt, že vývojář může upravit pouze některé metody a nemusí implementovat všechny, které vyžaduje rozhraní.

4.3 Přenos a generování dat klient server

Formuláře a tabulky jsou vytvářeny k tomu, aby reprezentovali uživateli data v systému. Hlavním úkolem formulářů je také odeslání dat na server. Z předchozích sekcí je již zřejmé, že klientská část aplikace nedisponuje stejnými datovými objekty jako server, ale pouze popisem struktury daného objektu. Tato informace je však dostačující a na jejím základě lze vygenerovat data, která je server schopný přijmout. Přenos probíhá v několika krocích. Tyto kroky zachycuje sekvenční diagram na obrázku C.8. Nejprve je zjištěno, zdali byl při vytváření komponenty specifikován zdroj, na který se mají data odeslat. Před vygenerováním dat, která budou odeslána, je provedena validace. V případě, že je validace úspěšná začnou se generovat data, která budou odeslána. K tomuto účelu slouží třída JSON builder, v případě že server očekává data ve formátu typu JSON. Framework nyní podporuje pouze JSON, nicméně návrh počítá s přidáním dalších datových builderů. Tyto buildery již neparsují data, která jsou uložena v komponentě, neboť za tuto činnost je zodpovědná konkrétní komponenta sama. Komponenta data parsuje z panelů, které si udržuje. Panel má jasně daný klíč, kterým lze určit umístění proměnné v původním objektu. Na základě tohoto klíče je vytvořen nový objekt. Klíč tedy určuje cestu. Pokud je v klíči znak tečky, znamená to, že je potřeba vyhledávat v již existující struktuře další potomky. Pokud v klíči znak tečky není, znamená to, že jsme již na správném místě a objektu, který je reprezentován třídou AFDataHolder, jejíž ukázka je na obrázku 4.4, bude přidána do jeho mapy další proměnná s hodnotou. Kromě klíče je potřeba znát i aktuální data v komponentě. Obdobně jako při vkládání dat do komponenty je i při získávání dat využit konkrétní widget builder, který komponentu sestavil, neboť zná strukturu a způsob, jakým data z komponenty získat. JSON builder tedy dostane objekt, ze kterého může data sestavit. K sestavení dat je využit framework GSON [5]. Když jsou již data sestavena, postačí je odeslat na konkrétní zdroj, který byl specifikován při vytváření komponenty. Framework toto odeslání provede automaticky.



Obrázek 4.4: Třída, která reprezentuje data ve formuláři, na jejím základě je sestaven objekt, který je odeslán na server.

Pokud chce vývojář data odeslat například při kliknutí na tlačítko, musí provést pouze dva kroky. Nejprve musí získat konkrétní formulář, který chce odeslat a poté nad ním zavolat akci sendData. Formulář lze získat z hlavní třídy AFSwinx, na základě jeho identifikátoru, kdekoliv v aplikaci. Žádné další akce nejsou potřeba. Mezi hlavní výhody patří snadná použitelnost formulářů a skutečnost, že je vývojář odstíněn od způsobu jakým se data odesílají. Nevýhodou je, že vývojář nemůže plně kontrolovat odeslání dat, či měnit implementaci odesílání a musí používat pouze metody, které mu framework nabízí.

4.4 Lokalizace

Aplikace mohou mít různé jazykové mutace. Klientská strana frameworku nabízí podporu pro lokalizace. Soubor s lokalizací lze nastavit buď builderu, který staví formulář, nebo třídě, zodpovědné za správu všech vygenerovaných komponent. Pokud je lokalizace nastavena této třídě, pak ji implicitně využijí všechny komponenty, pokud nebude toto nastavení překryto v builderu. Framework disponuje vlastním lokalizačním souborem, ve kterém jsou všechná validační hlášení. Pokud nebyl lokalizační soubor určen, využije se výchozí nastavení, kterým framework disponuje. Toto nastavení se používá pouze pro validační hlášení a pro objekt, který indikuje, že nebyla zvolena žádná hodnota. Hlavním důvodem tohoto chování je fakt, že nelze předpokládat, jaké texty bude chtít klient zobrazit. Pro popisy komponent, které jsou generovány serverem, se používá lokalizační soubor, který specifikoval uživatel. Pokud v něm není hodnota nalezena, použije se hodnota, kterou klient od serveru obdržel. Nemůže se tedy stát, že by byl text během překladu zahozen.

4.5 Validace dat a vlastní validátory

Při odesílání formuláře, je potřeba nejprve provést validace. Důvodem je, že validace omezují model buď na základě datových omezení, nebo na základě business omezení. Pokud server očekává v poli typu integer číslo a obdrží od klienta řetězec, pak dojde k chybě. Úkolem

validací je tyto chyby odhalit a poskytnout typovou kontrolu. Framework nenabízí business validace na serverové straně. To znaměná, že se data musí odeslat a pokud dojde k chybě na serveru během zpracování, je o této chybě klient informován. Nicméně z podstaty funkčnosti a návrhu je toto omezení nepřekonatelné, protože klient nemá reference na objekty, na základě kterých byly vygenerovány formuláře či tabulky a také nezná způsob, jakým se s objektem dále pracuje.

4.5.1 Podporované validace

V současné verzi jsou podporovány validace z tabulky 4.3. Všechny validátory jsou registrovány na komponentě až v poslední fázi generování. Výjimku tvoří retype validace, která ke své funkčnosti vyžaduje, aby byl zobrazen druhý identický prvek k widgetu, jenž bude tímto validátorem validován. Tento klon musí být taktéž přidán do aktuálního layoutu a zaregistrován jako komponenta, aby s ním bylo možné pracovat. V panelu je také nastaven speciální klíč a příznak, že se jedná o klon, neboť při generování dat, která budou odeslána na server, nechceme generovat data i z naklonovaného panelu.

Tabulka 4.3: Validátory, kterými disponuje klientská část

Název validátoru	**Priorita**	**Popis funčknosti**
Required	> 2miliony	Validátor zjistí, je-li je pole vyplněno, či je vybrána hodnota. Pokud není, je vyhozena výjimka.
NumberValidator	100	Validátor na základě typu widgetu určí, zdali má být v poli integer, double či long. Implicitní hodnota je integer. Poté získá aktuální hodnotu. Pokud hodnota nevyhovuje datovému typu, je vyhozena výjimka.
MinAndMaxValue	50	Validátor porovná aktuální hodnotu s minimální a maximální hodnotou. Pokud aktuální hodnota nevyhovuje, je vyhozena výjimka.
MinAndMaxLength	70	Validátor porovná počet znaků aktuální hodnoty s minimem a maximem, pokud aktuální hodnota nevyhovuje, je vyhozena výjimka.
Contains	30	Validátor porovná, zdali se v aktuální hodnotě vyskytuje řetězec, který je požadován. Pokud aktuální hodnota nevyhovuje, je vyhozena výjimka.
Retype	10	Validátor zjistí, zdali se hodnota v aktuálním poli shoduje s hodnotou v jiném poli. Pokud hodnoty nevyhovují, je vyhozena výjimka.

4.5.2 Vyhodnocení validací

Validace musí být vyhodnocovány v určitém pořadí. Nejprve je třeba vyhodnotit, má-li být pole vyplněné a až poté vyhodnocovat, jsou-li v poli pouze číselné hodnoty. Validátory jsou proto registrovány do prioritní fronty, kterou panel disponuje. U každého validátoru je nastavena jeho priorita, která určuje pořadí, v jakém budou validace ověřovány. Validaci vyvolává klient, pokud chce znát validitu formuláře, či framework automaticky v případě, že je požadováno odeslání dat na server. Validátory jsou řízeny výjimkami. Pokud validace není splněna, je vyhozena výjimka. V případě výjimky zareaguje framework tak, že ji odchytí a zobrazí její textovou reprezentaci u příslušného políčka, poté pokračuje s vyhodnocováním. Formulář, je tedy vždy celý validován. Klient si může doimplementovat vlastní validátor a následně ho zaregistorvat ke konkrétnímu panelu, který lze získat z vygenerovaného formuláře. Opět má klient dvě možnosti jak validátor vytvořit. Doporučeným způsobem je dědit od třídy AFBaseValidator a překrýt metodu validate. Tyto třídy jsou znázorněny na obrázku 4.5.

4.6 Layouty

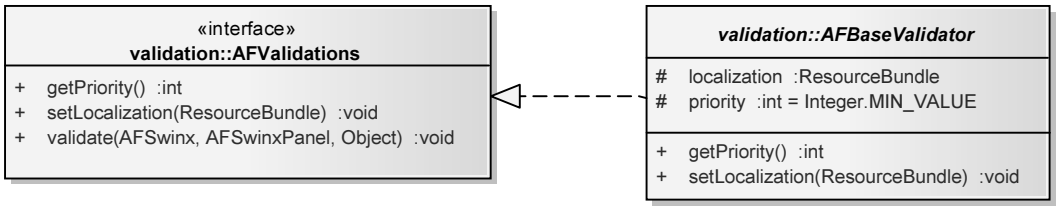
Komponenty v rámci formuláře musí být uspořádany tak, aby byl formulář dobře ovladatelný. Způsob jakým toho lze dosáhnout je specifický pro danou technologii. Nicméně server generuje stále stejné definice bez ohledu na technologii, kterou používá klient. Uspořádání komponent je definováno pomocí osy, ve které jsou prvky zobrazovány počtem sloupců a pozicí popisu komponenty v případě, že jde o konkrétní aktivní prvky formuláře. V případě, že jde o layout formuláře, není popis komponenty uveden. Definice layoutu může nabývat těchto hodnot:

1. layoutOrientation - orientace layoutu. Nabývá těchto hodnot osa X či Y (AxisX, AxisY)
2. layout - typ layoutu. Nabývá těchto hodnot: jednosloupcový či dvousloupcový (OneColumnLayout, TwoColumnsLayout)
3. labelPossiton - pozice popisu komponenty. Nabývá těchto hodnot před, za, není. (Before, After, None)

Layout je tedy potřeba na klientovi správně interpretovat a na základě jeho specifikace zobrazit komponenty v definované mřížce. Určení absolutní pozice komponenty ve formuláři není možné. Nicméně lze ovlivňovat pořadí, ve kterém jsou komponenty zobrazovány. Toto pořadí určuje server a klient zobrazuje komponenty v pořadí, v jakém je obdržel.

4.6.1 Layout builder

Ve Swingu je několik layoutů, které jsou jeho součástí a pak samozřejmě existují knihovny, které poskytují speciálně vytvořené a upravené layouty. Jedním z nejvíc komplexních a flexibilních layoutů je GridBagLayout [4]. Výhodou je, že ne všechny komponenty musí mít stejnou výšku, lze tedy vynutit specifické nastavení a docílit tak požadovaného vzhledu. Nevýhodou je složitost tohoto nastavení. Aby bylo možné dosáhnout požadovaného vzhledu, je potřeba kombinovat Swingové layouty. Framework disponuje svým vlastním implicitním



Obrázek 4.5: Abstraktní třídy validátorů, které lze použít k implementaci vlastního validátoru.

builderem, který vytváří GridBagLayout složený z BoxLayoutu. Tento builder umí vytvářet layouty pro jednotlivé komponenty formuláře, ale i pro celý formulář. Nejprve je builder inicializován, poté jsou definovány komponenty, ze kterých je potřeba layout poskládat a následně je layout vytvořen. Tento objekt je reprezentován panelem, jenž lze zobrazit. V tomto panelu jsou již všechny komponenty vykresleny a uspořádány v požadovaném pořadí.

Layout builder, určený k uspořádání prvků v rámci komponenty je využit v konkrétním widget builderu, který má znalost o tom, jaké prvky je potřeba přidat. Například vstupní textové pole má pouze tři prvky. Jsou jimi popis, samotné vstupní pole a placeholder určený k zobrazení validačního hlášení. Oproti tomu zaškrtávací políčka mají více komponent, neboť lze mít více zaškrtávacích políček, které mohou mít svoje popisy. Výhodou následujícího využití je v budoucnu možnost specifické implementace layoutu v závislosti na typu komponenty. Layout builder, určený k sestavení formuláře je využit ve formulářovém builderu. Tento builder má znalost, obdobně jako widget builder, o všech komponentách formuláře a jejich umístění.

4.7 Bezpečnost

Součástí každé aplikace je i zabezpečení. Na oblast bezpečnosti lze nahlížet z několika aspektů. Jedním z těchto aspektů je šifrování dat a ověření, komunikuje-li klient opravdu se správným serverem. Druhým aspektem je bezpečnost v rámci aplikace. S tím je spojená autentizace a autorizace zdrojů. Framework neposkytuje komplexní zabezpečení, ale v některých ohledech nabízí nástroje, kterými toho lze dosáhnout.

4.7.1 Přenos dat

Klient může specifikovat protokol, s jehož pomocí, klient komunikuje se serverem. Framework nabízí protokoly HTTP nebo HTTPS [6]. Výhodou HTTPS je HTTP protokol, který využívá SSL. Princip přenosu data je obdobný jako v případě HTTP. Nejprve klient vytvoří připojení na server a vyžádá si připojení přes SSL a poté pošle HTTP request pomocí SSL. HTTPS připojení vyžaduje svůj vlastní port.

Pokud chce uživatel využít protokolu HTTPS, tak je potřeba framework nastavit. Nastavení musí být provedeno na konkrétním datovém zdroji, který je specifikován pomocí XML. S

ohledem na uživatele byla přidána funkcionalita, která umí request pro HTTPS sestavit automaticky. Uživatel tedy pouze, v XML souboru se zdroji, uvede jako typ protokolu HTTPS a port, na kterém server umožňuje HTTPS požadavek přijmout. Pokud je certifikát serveru nevalidní, pak request pomocí HTTPS nelze provést.

4.7.2 Autentizace a autorizace

V netriviálních aplikacích je potřeba ověři,t zná-li systém uživatele, a zdali má uživatel právo provádět specifické akce. Ověření uživatele je autentizace. Výsledkem procesu autentizace je informace, jestli uživatele známe, tedy může-li se uživatel přihlásit do aplikace. Autorizace již ověřuje, jestli má přihlášený uživatel práva provádět specifické akce. Obvykle existuje tzv: Security Context, který umí ověřit, je-li uživatel v určité roli. Vzhledem k tomu, že framework je navržen tak, aby komunikoval se serverem pomocí HTTP či HTTPS, tak jsou možnosti autentizace a autorizace omezenější. Je potřeba umět na server odeslat uživatelské jméno a heslo. K tomuto účelu lze použít autorizaci typu basic. Klientská strana disponuje podporou pro tento typ autorizace. Obdobným způsobem jako v případě HTTPS, lze určit metodu, uživatelské jméno a heslo v XML souboru, který specifikuje připojení. Aby se hodnoty daly měnit za chodu aplikace, lze využít EL a konkrétní hodnoty nastavit až při generování definice zdrojů v klientské části. Na serverové straně lze vyřešit zabezpečení způsobem, jaký si vývojář sám zvolí. Framework AspectFaces [1], který je použit ke generování dat, disponuje anotací @UiUserRoles, pomocí které lze určit, při jakých uživatelských rolích bude proměná inspektována. Konkrétní roli uživatele je pak potřeba určit v kontextu. Zabezpečení pomocí oAuth není v současné verzi možné, ale framework umožňuje v hlavičce requestu odestlat jakýkoliv parameter. Například synchronizační token.

4.8 Porovnání přístupů

Při vytváření prezentační vrstvy dochází opakování kódů, což je v rozporu s jedním z principů vývoje softwaru. Tento prinicip se nazývá DRY. Do not repeat yourself. Bohužel nelze tento princip dodržovat striktně, neboť pokud vývojář chce zobrazit aktivní prvek, vždy musí vytvořit instanci určité komponenty či komponentu specifikovat. Formuláře obvykle mají několik komponent a každá z nich bude mít popis, aktivní prvek, validační hlášení a validaci jejíž logika musí být napsána. Dále je potřeba takto vygenerovaný formulář naplit a poté z něj získat data. Toto všechno musí vývojář naprogramovat. Kromě toho je potřeba také umístit komponenty do vhodného layoutu a zohlednit bezpečnost. I přes použití různých návrhových zdrojů, jako je například MVC [22] má výsledný kód mnoho řádků, které se v případě změny musí revidovat. Pokud formulář získává data ze serveru, musí být doimplementovány připojení na server, způsob získání dat, jejich uložení, reprezentace těchto dat a zpětné odeslání na server. Výhodou však zůstává, že má vývojář plnou kontrolu nad generovanou prezentační vrstvou a implementuje klienta vůči API na serveru, které se nemusí upravovat.

Pokud jsou komponenty generovány frameworkem, nemá nad nimi vývojář plnou kontrolu. Musí pracovat s API, které framework poskytuje a nemůže si komponenty nastavovat jinak, než framework dovolujte a je potřeba na server umístit zdroj, který bude generovat definice. Výhody jsou však mnohem větší. Není potřeba implementovat připojení na server

a základní autorizace. O reprezentaci dat se framework také postará, stejně tak jako o získání dat a vygenerování aktivních komponent dle serverové specifikace. Validátory a jejich logika je také součástí takto vygenerovaných komponent. Prezentační vrstva reaguje pružně na změny datových typů a uživatelských rolí. Důvodem je přístup k vytváření uživatelského rozhraní, při kterém se vždy zohledňují aspekty. Jedním z aspektů jsou komponenty, které jsou uživateli zobrazeny, dalším jsou způsoby, jak je sestavit, získat z nich data či do komponent data vložit. Způsoby validace a způsob jakým klient se serverem komunikuje. Jednou z největších výhod je ale fakt, že lze měnit prezentační vrstvu klienta přímou změnou na serveru. Ve většině případů neznamená změnový požadavek na klientskou prezentační vrstvu nutnost distribuovat novou verzi.

Kapitola 5

Testování

Testování je nedílnou součástí při vývoji softwaru. Mimo ověření správné funkčnosti aplikace se také testuje, zdali bylo vytvořeno přesně to, co bylo v zadání. Rozeznáváme automatické a manuální testování. Manuální testování dělají lidé. Výběr těchto testerů by měl odpovídat cílové skupině, pro kterou je aplikace designována. Na toto testování lze nahlížet z několika aspektů. Prvním z nich je, zdali aplikace opravdu dělá to, co má a druhým aspektem je, jak se uživateli s aplikací pracuje. V této kapitole budou popsány různé typy testování, které byly na frameworku provedeny a testovací projekt, který byl v této souvislosti vytvořen. Software disponuje několika Unit testy, které pokrývají chování systému, které provádí složitější generování či parsování. Uživatelský test prověřil použitelnost frameworku a poskytl informace o tom, jak uživatelé pracují s aktuální verzí. Showcase projekt demonstruje možnosti a svým způsobem také testuje software. V tomto případě se jednalo o Alfa test.

5.1 Unit testy

Unit testy testují určitou část softwaru, která by měla být co nejmenší. Od toho pochází název unit, neboť se jedná o malou jednotku, jenž test pokrývá. Unit testy pokrývají většinou metody či spolupráci metod a neměli by testovat celkové chování systému. Při návrhu testů je potřeba zvážit, zdali test přinese užitek a zdali může odhalit potencionální chybu. Například testování getterů a setterů je zbytečné. Absolutní pokrytí testů není možné z důvodů časové a prostorové náročnosti, vývoje testů, udržování a možností, které by se museli vygenerovat. Proto je vhodné testovat krajní případy a několik standardních případů použití.

K testování byl využit framework JUnit [8]. JUnit testy se spouští při každém buildu aplikace pomocí Mavenu [9], pokud není určeno jinak. Framework disponuje mnoha vlastnostmi, které lze použít. Integrace do existujících projektů je jednoduchá a framework podporuje anotace, pomocí kterých lze přizpůsobovat test a určovat chování testu či specifikovat výjimky, které test očekává. Testy mohou být spouštěny ve skupinách. Automatické testování pokrývají následující vlastnosti:

1. Test správné propagace proměnných do XML souboru s definicemi zdrojů - Do XML souborů je možné vložit hodnoty z hash mapy. Tyto hodnoty mohou za běhu ovlivňovat definici zdrojů a způsob jejich připojení.

2. Test správné serializace dat získaných z formuláře na JSON - Z formuláře jsou získána data, která mají pouze textovou reprezentaci a na jejich základě je třeba sestavit JSON. Sestavení JSONu je potřeba ověřit.

3. Test správného vytvoření modelu na základě XML - Model je vytvářen na základě definic. Statické definice jsou součástí zdrojů a testuje se, je-li na jejich základě možné vygenerovat model. Netestují se tedy definice, ale generátor.

4. Test správného získání možností z určeného výčtového typu - V mnoha případech je třeba získat data z výčtových typů. V tomto testu se testuje zdali algoritmus, který data získává, funguje korektně.

5.2 Uživatelský test

Návrh frameworku zohledňoval cílovou skupinu uživatelů a snažil se přizpůsobit jejich potřebám. Při návrhu jsem čerpal ze svých vlastních zkušeností, které jsem získal při vývoji software a při své stáži v RedHatu. Firma RedHat vyvíjí v rámci jejich projektu WFK produkt RichFaces [13]. Jedná se o open source komponentový framework pro webové aplikace využívající technologii JSF. Tento projekt má početný tým testerů, který je napojen na upstream. K testování se využívají JUnit testy a Arquillian testy. I přes komplexnost projektu a stabilnost frameworku, bylo rozhodnuto o jeho ukončení. Jedním z důvodů je fakt, že projekt dostatečně rychle nepřinesl podporu mobilních zařízení, dalším důvodem je skutečnost, že na rozdíl od svého přímého konkurenta PrimeFaces [12], není tak často používaný. Tento fakt lze přikládat použitelnosti framewoku. Při použití komponent je potřeba udělat několik operací, které nemusí být pro začínající uživatele intuitivní.

5.2.1 Cílová skupina

V mém návrhu jsem se proto soustředil na použitelnost. Cílovou skupinou jsou vývojáři, kteří chtějí knihovnu použít v nové či v stávající aplikaci. Tito vývojáři by měli mít základní zkušenost s webovými službami a v případě využití Swingové části samozřejmě s knihovnou Swing. Architekturu a způsob použití jsem se koncipoval tak, aby bylo přímočaré, nicméně bylo potřeba ověřit tuto skutečnost s uživateli během testu. Účastníky testu jsem vybíral ze svých kolegů studentů, o nichž jsem věděl, že patří do cílové skupiny. Také jsem zvolil jako dva z testerů osoby, které již pracují jako softwarový vývojáři. Tímto výběrem jsem chtěl zajistit různorodost informací, které jsem mohl získat během testování. Test jsem provedl se čtyřmi participanty.

5.2.2 Test setup

Vzhledem k tomu, že má framework dvě části, bylo potřeba vytvořit prostředí pro serverovou i klientskou část. Serverová část je mnohem komplikovanější, neboť vytvoření aplikace na platformě Java EE s restovým rozhraním a vrstvou, jež bude poskytovat data, není triviální věc a samotné nastavení serveru a tvorba tříd, které jsou zodpovědné za tuto funkcionalitu, by zabrala více času než test samotný. Takto koncipovaný test by neměl vypovídající hodnotu o využití testovaného frameworku. Testujícímu byla tedy připravena aplikace, která

již tyto vlastnosti splňovala a obsahovala výchozí data. Uvažoval jsem o případu, kdy chce vývojář rozšířit aktuální aplikaci o tento framework. Myslím si, že toto bude v praxi nejčastější použití, neboť zatím nepředpokládáme, že by vývojáři tvořili aplikace na míru našemu frameworku. Co se týče klientské strany, byla připravena Swingová aplikace, kterou mohl participant upravovat. Klientská aplikace nedisponovala, žádnými knihovnami třetích stran. Test probíhal na mém notebooku, který měl předpřipravené prostředí. Tester mohl používat nástroj RestClient pro ověření funkčnosti serverové strany, JBDS vývojové prostředí a byla mu dodána uživatelská příručka. Před testem mu byla vysvětlena základní idea frameworku a byl mu představen framework AspectFaces, který je využit pro generování definic dat. Po testu bylo prostředí resetováno a test se mohl opakovat.

5.2.3 Test case

Při sestavování úkolů pro testera bylo potřeba vzít v úvahu náročnost jednotlivých úkolů. Z tohoto důvodu byly koncipovány úkoly tak, aby na sebe navazovaly. Participantovi byly zadány následující úkoly v níže uvedeném pořadí.

1. Seznamte se se serverovou a klientskou částí. Prostudujte uživatelskou příručku.
2. Vytvořte v serverové části zdroj, který bude vracet definici dat objektu Country. Předveďte funkčnost pomocí RestClienta
3. Vytvořte v klientské části prázdný formulář, který bude sestaven na základě definice z předchozího bodu.
4. Naplňte formulář daty ze serveru. Konkrétně zobrazte ve formuláři zemi s identifikačním číslem 1.
5. Vytvořte prázdný formulář stejný jako v bodu 3, který lze odeslat zpět na server a po jeho odeslání bude přidána nová země do databáze.
6. Vytvořte tabulky se všemi zeměmi v databázi a přesvědčte se, zdali byla země z předchozího bodu vložena.
7. Upravte tabulku z předchozího bodu tak, aby byla její velikost závislá na velikosti textů, které zobrazuje.
8. Vytvořte formulář z bodu 3, který bude moci upravovat zemi s identifikačním číslem 1 a po jeho odeslání se změna projeví na serveru v databázi.
9. Vytvořte formulář z bodu 3, který zobrazí zemi s identifikačním číslem 1 a po jeho odeslání bude země ze serverové databáze smazána.
10. Vytvořte formulář z bodu 3. Změňte komponentu tak, aby u proměnné active bylo místo checkboxu dropDownMenu.

Cílem bylo zjistit, zdali jsou uživatelé schopni s frameworkem pracovat, jakým způsobem ho využívají a jsou-li schopni pracovat s vygenerovanými komponentami.

5.2.4 Vyhodnocení

Všichni uživatelé byli schopni splnit všechny zadané úkoly. V některých případech bylo potřeba testera trochu navést. Tyto případy zde budou rozvedeny. Po skončení testů dostali uživatelé otázku, jak se jim s frameworkem pracovalo. Participanti hodnotili práci kladně, ale měli i pár výtek. Konkrétně měli dojem, že je framework moc chytrý. Jako příklad zde uvedu bod 3. Při plnění tohoto bodu postupovali podle uživatelské příručky, ve které jsou zobrazeny příklady definic zdrojů. Uživatelé vždy zkopírovali zdroj celý a upravili cesty ke zdrojům. To způsobilo, že byl vygenerován formulář, který již obsahoval data. Testeři jako první zkusili vyhledat na formuláři metodu, která data odstraní, či se snažili najít metodu, která zakáže zobrazování dat. Až po nějaké době zkusili vymazat definici zdroje. Tento problém se vyskytl téměř u všech participantů. Výjimkou byl pouze jeden z nich, který nezkopíroval celou definici zdroje, ale jen jeho část. Tento participant však strávil mnohem více času definicí zdroje než jeho kolegové.

Tabulka 5.1: Problémy se splněním jednotlivých bodů

Popis problému	**Návrh řešení**
Pokud je specifikován zdroj, jsou zobrazena data ve formuláři, pokud není pak nejsou. Framework si formulář automaticky přizpůsobí.	Uživatelé by měli rádi větší kontrolu nad tím, jak se formulář chová. Je vhodné přidat metodu, která zapne či vypne plnění dat do formuláře.
Určení zdroje na základě XML. Problém byl s identifikátorem. Uživatel sice správně identifikátor zvolil, důvodem byl fakt, že ve třídě, která načítá data je identifikátor značen jako connectionKey a v konkrétních XML je reprezentován jako id.	Předělat název proměnné ve třídě, tak aby odpovídala XML.
Při vytváření tabulky chtěl uživatel provádět akce navíc. Uživatel nebyl přesvědčený o tom, zdali výměna builderu při zachování zdrojů umožní sestavit tabulku.	Je potřeba upravit uživatelskou příručku, aby tuto informaci přesně specifikovala.
Získání přístupu k již existující komponentě. Uživatel chvilku tápal jak získat komponentu, kterou vygeneroval. Důvodem bylo to, že uživatel chtěl komponentu získávat z panelu do kterého ji vložil. Použití správce komponent AFSwinx mu došlo až po té co si znovu přečetl uživatelskou příručku.	Možná by bylo vhodné zvážit, zdali nepřejmenovat třídu na ComponentManager či použít jiný název, který by více odrážel její funkci. Dále je potřeba upravit uživatelskou příručku a věnovat této komponentě vlastní sekci.
Odesílání dat na severu. Uživatel zde chvilku hledal metodu, jakou lze data odeslat. Problémem bylo to, že výsledná komponenta dědí od JPanel a nabízí tedy velké množství metod.	Řešením by bylo přidat metodu k odeslání do AFSwinx a předat jí formulář, metodu k posílání dat na komponentě je ale potřeba rozhodně zachovat.

Uživatelé byli s frameworkem spokojeni. Jeden z uživatelů si poté vytvořil vlastní panel, ve kterém zobrazil formulář pro vložení země do databáze, vedle něj umístil tabulku se zeměmi a sledoval, jak se data mění. Další z nich při vytváření tabulky použil již existující kód, který mu vygeneruje formulář. Pouze změnil slovo Form na Table, prohlásil, že by to takto mohlo fungovat a program spustil. Tabulka se mu sestavila, ale obsahovala pouze jeden prvek. Změnu zdroje, který získá všechny země, pak uživatel zvládl bez problémů. Test byl pro uživatele i časově náročný neboť se museli seznámit s frameworkem, začít ho správně používat a chybějící metody a chování doprogramovat pomocí frameworku. Během testování bylo sledováno, jak participanti ovládají vygenerované komponenty. S ovládáním komponent neměl žádný participant problém. Vygenerované formuláře a tabulky jim přišli dostatečně velké a přehledné. Při odesílání dat na server byli uživatelé schopni opravit vstupní data, pokud byla validace neúspěšná.

5.3 Ukázkový projekt

Součástí práce je i poměrně rozsáhlý ukázkový projekt. Tento projekt je rozdělen na dvě části. Serverovou a klientskou část. Obě tyto části demonstrují použití frameworku. Serverová část je odladěna pro aplikační server GlassFish V3 a v4 [3]. Ukázkové projekty se využívají k tomu, aby uživateli ukázaly jak framework funguje a v některých případech i k testování, nejčastěji pokud jde vývoj knihoven. Projekt se pak používá k odladění během fáze implementace, poté k testování během fáze testování a k smoke testům před závěrečným releasem. Pokud je ukázkový projekt dostatečně rozsáhlý, zabere mnoho času rozšiřování tohoto projektu. V našem případě bylo potřeba implementovat jak serverovou, tak klientskou část.

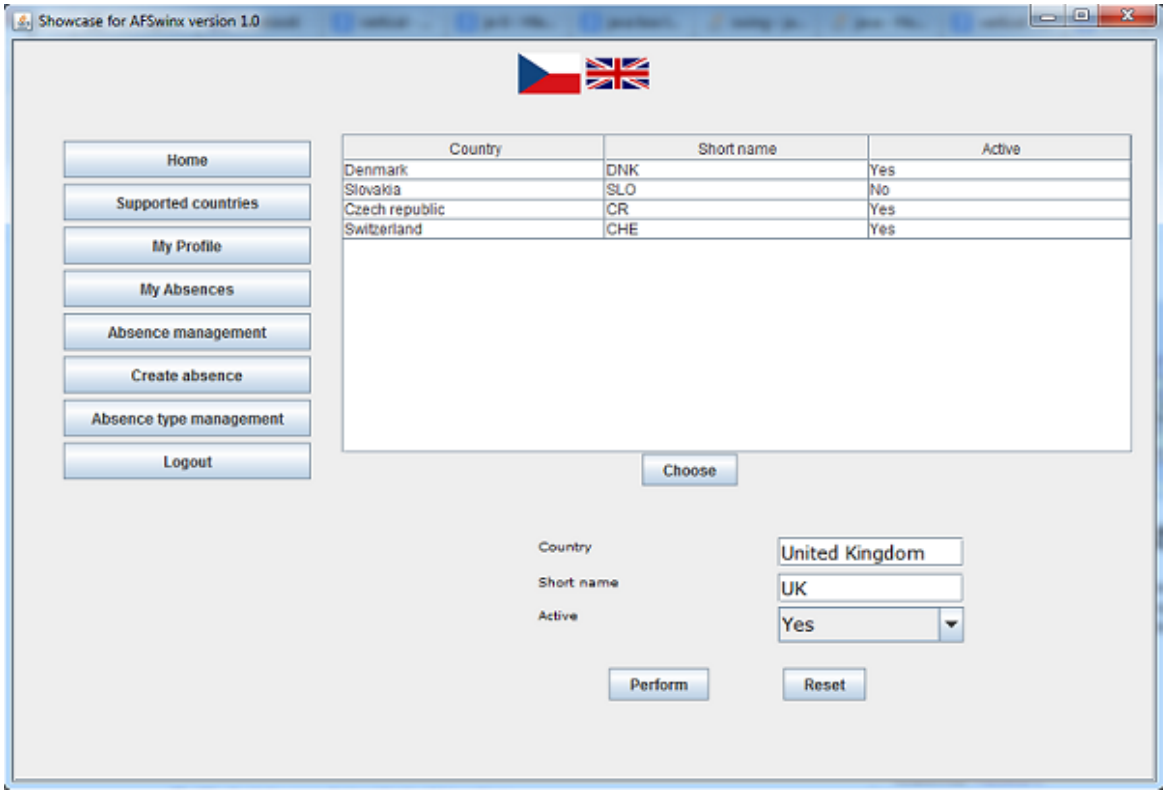
5.3.1 Popis projektu

Ukázkový projekt je zjednodušená aplikace sloužící k zadávání dovolených. Aplikace umožňuje žádat o dovolenou a rušit tyto žádosti. Umí spravovat uživatele, také umí vytvářet země, ve kterých se uživatel nachází, a ke kterým se váží konkrétní typy pracovní nepřítomnosti. Serverová strana využívá in-memory DerbyDB. Databázová a business vrstva je v ukázkové aplikaci spojená a manažeři, kteří jsou zodpovědní za správu dat, jsou stateless EJB [7]. Webové API je poskytováno pomocí knihovny RestEasy. Zabezpečení je realizováno pomocí interceptoru, který na základě basic autorizace a anotací z javax.annotation.security určí, zdali má uživatel právo k přístupu ke zdroji či nikoliv.

Klientská část aplikace využívá frameworku AFSwinx, získává data ze serverových zdrojů a ty prezentuje uživatelům. Také nabízí vytvoření formulářů a vkládání, mazání či úpravy dat. V aplikaci existuje tzv. security context, který udržuje informace o přihlášeném uživateli a v případě potřeby je odesílá na server.

5.3.2 Správa zemí

Na obrázku 5.1 je zobrazen náhled obrazovky, zobrazující komponenty potřebné ke správě zemí. Aby si uživatel zobrazil tyto země, je potřeba se nejprve přihlásit. Přihlášení je realizováno pomocí metody basic. Obrazovka, na které je zobrazen přihlašovací formulář, je na obrázku C.10. Aplikace zobrazí všechny země, které jsou k dispozici v zadané v tabulce.



Obrázek 5.1: Ukázkový projekt - klientská část

Uživatel si může zemi zobrazit v detailním náhledu a popřípadě zemi upravit. Zobrazení detailních informací provede uživatel tak, že klikne na konkrétní zemi, a poté na tlačítko choose. Data jsou do formuláře propagována pomocí kontroleru bez nutnosti stahovat data ze serveru. Tato obrazovka také demonstruje možnost použití zabezpečení. V případě, že je uživatel v roli administrátora, je mu umožněna editace těchto polí. Pokud uživatel v této roli není, pak jsou políčka needitovatelná. První vrstva zabezpečení je interpretována na straně klienta pomocí vygenerovaného uživatelského rozhraní. Druhá vrstva je na serveru. V případě obdržení požadavku na změnu se ověří, zdali má uživatel právo měnit zadaná data.

5.3.3 Správa typů nepřítomností

Každá země má svoje vlastní důvody nepřítomnosti a počet dní, které lze na tuto nepřítomnost čerpat. Na obrázku C.11 je zobrazen náhled na správu absenčních typů. Nejprve je potřeba vybrat zemi pomocí výběrového menu ve vrchní části stránky. Toto menu je generovaný jednoprvkový formulář, jehož vnitřní struktura reprezentuje identifikátor konkrétní země a uživateli je zobrazen název dané země. Po výběru konkrétní hodnoty je kontrolerem inicializována akce, při které se začne vytvářet tabulka a formulář, kterým je předán identifikátor vybrané země. Do tabulky jsou vložena data a je možné je opět měnit pomocí formuláře pod tabulkou.

5.3.4 Správa nepřítomností

Každý uživatel může žádat o schválení nepřítomnosti. Při vytváření zvolí datum začátku, datum konce a typ nepřítomnosti. Uživatel může zadat pouze typ, který odpovídá jeho zemi. Po vytvoření nepřítomnosti se mu tato nepřítomnost zobrazí v přehledu a ve správě nepřítomností ji lze měnit. Správa nepřítomností je na obrázku C.13. Upravit nepřítomnost lze pouze pokud je ve stavu Requested nebo Cancelled. V případě, že si správu zobrazí správce, může měnit nejen své nepřítomnosti, ale také nepřítomnosti ostatních uživatelů. Tuto skutečnost zachycuje obrázek C.13. Správce má k dispozici také mnohem více stavů. Data do tabulky jsou poskytována serverem na základě uživatelských rolí. Definice tabulky je pro obě role společná, nicméně definice formuláře je opět závislá na typu role.

5.3.5 Nasazení

Aplikaci lze bez nutnosti konfigurace nasadit na aplikační server Glassfish verze 3 a 4. K využití lze použít asaadmin, což je nástroj, který umožňuje ovládat aplikační server pomocí příkazové řádky. Soubor s WAR je součástí přiloženého CD či ho lze získat po spuštění mvn clean install v kořenovém adresáři. Poté je potřeba provést následující:

1. Rozbalte aplikační server GlassFish, který je přiložen na CD nabo si stáhnětě verzi 3 z http://www.oracle.com/technetwork/middleware/glassfish/downloads/java-archive-downloads-glassfish-419424.html Verzi 4 lze stáhnout z http://dlc.sun.com.edgesuite.net/glassfish/4.1/release/glassfish-4.1.zip
2. Rozbalte soubor, ve složce bin spusttě utilitu asadmin napsáním asadmin
3. Vložte následující příkaz: start-domain domain1
4. Vložte následující příkaz deploy PATHTOFILE/AFServer.war
5. Jděte na http://localhost:8080/AFServer - zobrazí se text: I am alive. Serverová strana nedisponuje grafickým uživatelským rozhranním. Funkčnost můžete otestovat rest clienta například na adrese http://localhost:8080/AFServer/rest/country/list - content-type: application/json metoda GET.

Klientskou část aplikace lze spustit. Spustitelný JAR soubor je na přiloženém CD či ho lze vytvořit spuštěním následujícího příkazu ve složce examples/Showaces. Příkaz je: mvn clean package assembly:single . Do složky target bude vygenerován soubor Showcase.jar, který lze spustit java –jar Showcase.jar.

Kapitola 6

Závěr

6.1 Budoucí vývoj

Testování s uživateli dopadlo dobře a na základě toho bychom chtěli framework dále rozvíjet. Serverová část frameworku umožňuje vytvářet definice komponent. Tyto informace lze využít ke stavbě komponent na libovolné platformě. V další iteraci, bychom rádi přidali možnost využívat tyto definice na mobilní platformě a v dynamicky se rozvíjejícím frameworku AngularJS. S tím se bude pojit vyjmutí specifických modulů z části AFSwinx a jejich propagace do jiného projektu, který pak bude využit na těchto platformách. Z hlediska bezpečnosti lze zatím využít autorizace typu basic, v tomto ohledu bychom rádi projekt rozšířili o další možnosti například o oAuth. Framework nyní podporuje množinu validací, kterou budeme dále rozšiřovat. Budou provedeny další UX testy, na základě kterých může být upraven způsob, jakým framework funguje.

Iterace a rozvoj frameworku by měl vyústit v plně nasaditelnou technologii, kterou bychom chtěli distribuovat v rámci AspectFaces. Tím docílíme toho, že vývojář při využití tohoto projektu definuje definice pouze jednou a klienti je budou moci používat na různých platformách. Výhodou bude velmi rychlé prototypování a vytváření aktivních prvků, které budou reflektovat změny v serverové konfiguraci a změny v modelu, který reprezentují.

6.2 Zhodnocení práce

Cílem práce bylo navrhnout způsob, jakým lze aspektově generovat uživatelská rozhraní na platformě Java SE. Práce se zaměřuje především na tlusté klienty k serverovým aplikacím. Framework byl navržen, sestaven a otestován pomocí ukázkové aplikace. Použitelnost frameworku byla testována při testech použitelnosti s cílovou skupinou uživatelů.

Nejprve bylo potřeba zpracovat již existující řešení a zhodnotit jejich výhody a nevýhody. Již v této fázi bylo zřejmé, že pokud budou objekty dat centralizovány, bude zjednodušena jejich správa a možné úpravy. Způsob má i jednu nevýhodu, pokud si klient od serveru vyžádá data, potom je struktura, kterou obdrží, závislá na datech. V případě že se mění struktura dat, musí se přizpůsobovat i klient. Tuto nevýhodu lze odstranit dynamickou inspekcí dat, která vytvoří definice, jež se využijí k sestavení komponenty a poté ji nastaví na základě

obdržených dat. Toto nastavení je dynamické. Klient se tedy nemusí přizpůsobovat, pokud server změní strukturu dat, neboť mu tuto skutečnost server sdělí ihned při generování definic.

Byl vytvořen návrh definic dat, pomocí kterých umí klient vytvořit formuláře a tabulky. Tyto definice jsou generovány na serverové straně s využitím frameworku AspectFaces a poté odeslány na klienta, kde jsou interpretovány. Klient umí na základě těchto definic vytvářet tabulky či formuláře, které sestaví podle určeného layoutu a nastaví všem aktivním prvkům validátory. Při obdržení dat umí tyto data správně vložit do komponent a následně datový objekt znovu sestavit pouze ze znalosti definice. Klientská strana předem nezná objekt, jenž obdrží a může tedy pružně reagovat na změny datových definic. Z hlediska bezpečnosti podporuje klient komunikaci pomocí HTTPS a autorizaci typu basic. Framework, je tedy rozdělen na klientskou a serverovou část, s tím, že klientovi stačí vložení závislosti na klientskou stranu a serveru na serverovou stranu a v případě využití AspectFaces i závislost na tento projekt. Ke generování dat se využívá knihovna třetí strany. Výhodou je její stabilnost a fakt, že disponuje velkou škálou možností, jak uživatelské rozhraní generovat. Framework zohledňuje uživatelské role a nabízí anotace, které lze využít při generování. Šablony byly upraveny tak, aby vytvářely jednotnou definici použitelnou na více platformách. Definice je tedy platformě nezávislá.

Při vývoji frameworku jsem se musel na problém zaměřit jak z hlediska funkčnosti, tak z hlediska použitelnosti, a zohlednit budoucí vývoj frameworku. Bylo potřeba navrhnout způsob, jakým se bude model generovat a následně přenášet do klientské části aplikace. V klientské části bylo třeba model zpracovat a na jeho základě sestavit konkrétní komponenty, se kterými může klient dále pracovat, a ve kterých bude zohledněna bezpečnost, layout a vnitřní reprezentace dat. Tyto vlastnosti byly v analytické části rozpracovány a v implementační části vytvořeny. Výsledný framework byl otestován s uživateli při testech použitelnosti, ve kterých jsem se zaměřil na způsob, jakým uživatelé framework využívají. Některé části pokrývají unit testy a také byla vytvořena ukázková serverová a klientská aplikace. Tato ukázková aplikace sloužila k alfa testování a do budoucnosti bude sloužit pro smoke testy. Framework byl úspěšně navržen, vytvořen a otestován.

Literatura

[1] *AspectFaces framework* [online]. [cit. 10. 12. 2014]. Dostupné z: <http://www.aspectfaces.com/overview>.

[2] *Framework definition* [online]. [cit. 23. 12. 2014]. Dostupné z: <http://cs.wikipedia.org/wiki/Framework>.

[3] *GlassFish Server Open Source Edition* [online]. [cit. 20. 12. 2014]. Dostupné z: <https://glassfish.java.net/docs/4.0/quick-start-guide.pdf>.

[4] *How to Use GridBagLayout* [online]. [cit. 21 . 12. 2014]. Dostupné z: <http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html/>.

[5] *Google-GSON* [online]. [cit. 20. 12. 2014]. Dostupné z: <https://code.google.com/p/google-gson/>.

[6] *SSL and TLS*, s. 292 – 307. Addison-Wesley, Boston, vyd. 1. edition, 2001.

[7] *Java EE at a Glance* [online]. [cit. 10. 12. 2014]. Dostupné z: <http://www.oracle.com/technetwork/java/javaee/overview/index.html>.

[8] *JUnit* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://junit.org/>.

[9] *Maven* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://maven.apache.org/what-is-maven.html>.

[10] *Metadata* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://cs.wikipedia.org/wiki/Metadata>.

[11] *Metawidget* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://metawidget.sourceforge.net/>.

[12] *PrimeFaces* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://primefaces.org/>.

[13] *RichFaces* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://richfaces.org/>.

[14] *Creating a GUI With JFC/Swing* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://docs.oracle.com/javase/tutorial/uiswing/index.html>.

[15] *SwiXml* [online]. [cit. 22. 12. 2014]. Dostupné z: <http://www.swixml.org/>.

[16] ARLOW, J. – NEUSTADT, I. *UML 2 a unifikovaný proces vývoje aplikací: Objektově orientovaná analýza a návrh prakticky.* Computer Press a.s., Brno, 2nd edition. ISBN 978-90-251-1503-9.

[17] CERNY, T. – CHALUPA, V. – DONAHOO, M. Towards Smart User Interface Design. In *Information Science and Applications (ICISA), 2012 International Conference on*, s. 1–6, May 2012. doi: 10.1109/ICISA.2012.6220929.

[18] CERNY, T. – SONG, E. UML-based Enhanced Rich Form Generation. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, s. 192–199, New York, NY, USA, 2011. ACM. doi: 10.1145/2103380.2103420. Dostupné z: <http://doi.acm.org/10.1145/2103380.2103420>. ISBN 978-1-4503-1087-1.

[19] CERNY, T. et al. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. *SIGAPP Appl. Comput. Rev.* December 2013, 13, 4, s. 53–66. ISSN 1559-6915. doi: 10.1145/2577554.2577561. Dostupné z: <http://doi.acm.org/10.1145/2577554.2577561>.

[20] CERNY, T. – DONAHOO, M. J. – SONG, E. Towards Effective Adaptive User Interfaces Design. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, s. 373–380, New York, NY, USA, 2013. ACM. doi: 10.1145/2513228.2513278. Dostupné z: <http://doi.acm.org/10.1145/2513228.2513278>. ISBN 978-1-4503-2348-2.

[21] ERIC JENDROCK, I. E. a. a. *The Java EE 7 Tutorial* [online]. [cit. 10. 12. 2014]. Dostupné z: <http://docs.oracle.com/javaee/7/tutorial/doc/>.

[22] FOWLER, M. *Patterns of Enterprise Application Architecture.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321127420.

[23] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA : Addison-Wesley, 1995. ISBN 978-0-201-63361-0.

[24] KALNICHEVSKI, O. *HttpCore Tutorial* [online]. [cit. 20. 12. 2014]. Dostupné z: <http://hc.apache.org/httpcomponents-core-ga/tutorial/pdf/httpcore-tutorial.pdf>.

[25] TIM BRAY, J. P. a. a. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. [cit. 20. 12. 2014]. Dostupné z: <http://www.w3.org/TR/REC-xml/1>.

[26] TUCHINDA, R. et al. Building Mashups by example. *Proceedings of the 13th international conference on Intelligent user interfaces - IUI '08.* 2008, s. 15–28. Dostupné z: <http://www.isi.edu/integration/papers/tuchinda08-iui.pdf>.

Příloha A

Seznam použitých zkratek

GNU GPL GNU General Public License

UML Unified Modeling Language

MVC Model-view-controller

REST Representational State Transfer

JDBC Java Database Connectivity

OSGI Open Services Gateway Initiative

HTTP Hypertext Transfer Protocol

ORM Object relational mapping

EJB Enterprise JavaBean

JPA Java Persistence API

API Application programming interface

EL Expression language

HTML HyperText Markup Language

XML Extensible Markup Language

JSF JavaServer Faces

LGPL EPL Lesser GPL Eclipse Public Licence

SSL Secure Sockets Layer

HTTPS Hypertext Transfer Protocol Secure

WAR Web Application Archive

JAR Java Archive

Příloha B

Instalační a uživatelská příručka

Framework byl vytvořen jako Maven projekt. Do aplikace ho lze přidat buďto jako knihovnu nebo jako Maven závislost. Způsob, jakým lze integrovat projekt a jak ho používat je detailně popsán v uživatelské příručce na přiloženém CD.

B.1 Maven závislosti

Nejprve je potřeba provést build frameworku. Zdrojové kódy jsou na přiloženém CD. Framework zatím není v žádném z veřejně dostupných repositářích. Poté lze na serveru stranu přidat následující závislosti:

Část zdrojového kódu B.1: Závislosti na serveru

```
<dependency>
        <groupId>com.tomscz.af</groupId>
        <artifactId>AFRest</artifactId>
        <version>0.0.1-SNAPSHOT</version>
</dependency>
<dependency>
        <groupId>com.codingcrayons.aspectfaces</groupId>
        <artifactId>javaee-connector</artifactId>
        <version>1.5.0-SNAPSHOT</version>
</dependency>
<dependency>
        <groupId>com.codingcrayons.aspectfaces</groupId>
        <artifactId>annotation-descriptors</artifactId>
        <version>1.5.0-SNAPSHOT</version>
</dependency>
```

Repozitář pro aspectFaces je zde:

Část zdrojového kódu B.2: AspectFaces repozitář

```
<repository>
        <id>codingcrayons-repository</id>
        <name>CodingCrayons Maven Repository</name>
        <url>http://maven.codingcrayons.com/content/groups/public/</url>
</repository>
```

Do složky WEB-INF je potřeba rozbalit soubor templates.zip, v kterém je předpřipravená konfigurace a do web.xml je potřeba přidat listener, který provede nastavení AspectFaces během startu.

Část zdrojového kódu B.3: AspectFaces listener

```
<listener>
        <!-- Include Aspect Faces listener to perform proper framework initialization
        during application start -->
        <listener-class>com.codingcrayons.aspectfaces.plugins.j2ee.AspectFacesListener</listener-class>
</listener>
```

Na klientskou stranu je potřeba přidat následující závislost:

Část zdrojového kódu B.4: Závislost na klientské straně

```
<dependency>
        <groupId>com.tomscz.af</groupId>
        <artifactId>AFSwinx</artifactId>
        <version>0.0.1-SNAPSHOT</version>
</dependency>
```

B.2 Ukázkový projekt

Ukázkový projekt se svojí klientskou a serverovou částí je již vytvořen a přiložen na CD. Serverovou část lze bez dodatečné konfigurace spustit na serveru GlassFish V3. Ukázkový projekt je možné spustit i na GlassFish V4 nicméně, při deployi aplikace je v konzoli zobrazena chyba, avšak aplikace je plně funkční. Na toto chování byl založen bug, který není prozatím vyřešen. Je potřeba provést následující:

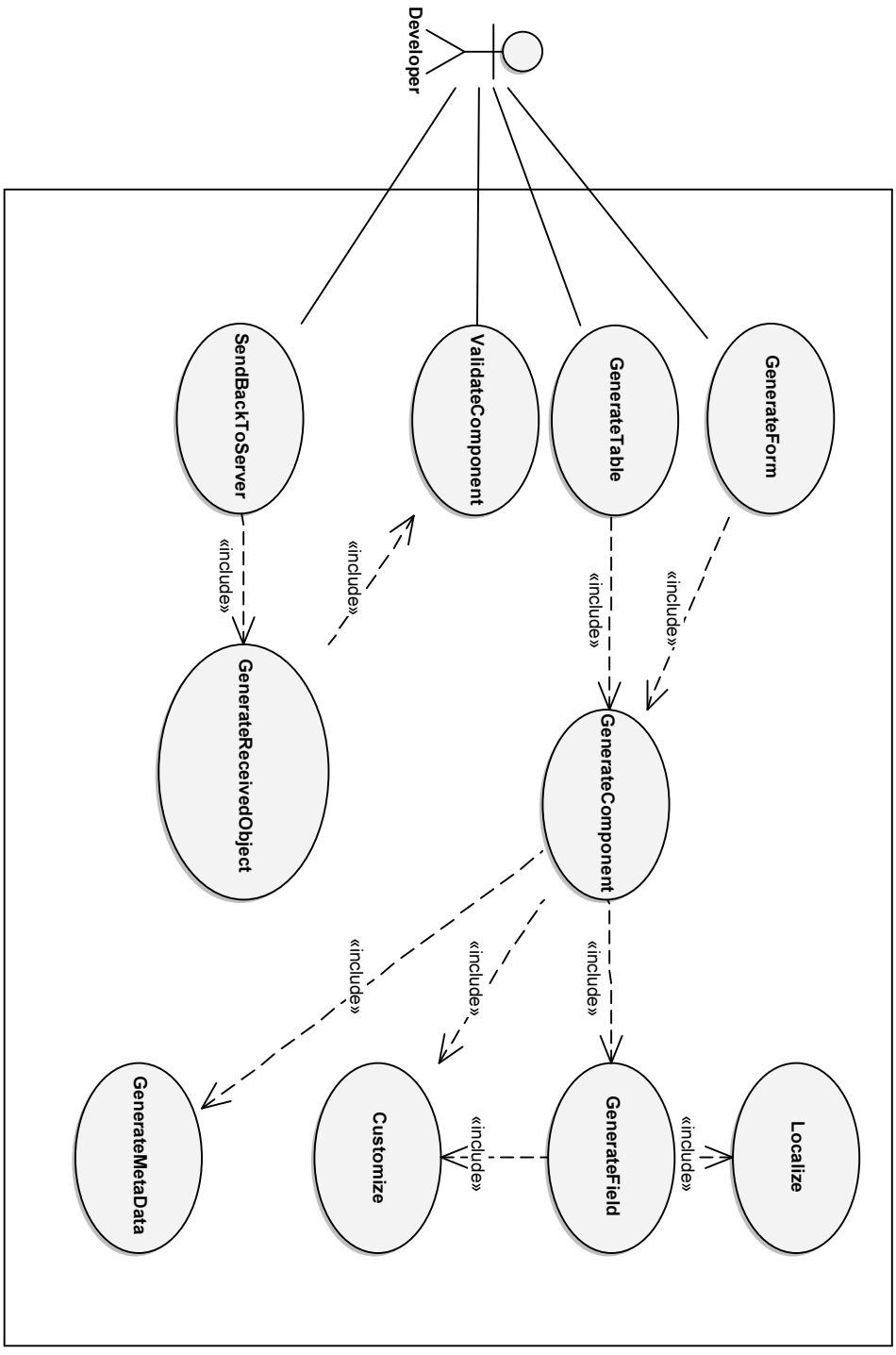
1. Rozbalte aplikační server GlassFish, který je přiložen na CD nebo si stáhněte verzi 3 z http://www.oracle.com/technetwork/middleware/glassfish/downloads/java-archive-downloads-glassfish-419424.html Verzi 4 lze stáhnout z http://dlc.sun.com.edgesuite.net/glassfish/4.1/release/glassfish-4.1.zip
2. Rozbalte soubor, ve složce bin spoustě utilitu asadmin napsáním asadmin
3. Vložte následující příkaz: start-domain domain1
4. Vložte následující příkaz deploy PATHTOFILE/AFServer.war
5. Otevřete webový prohlížeč na adrese http://localhost:8080/AFServer - zobrazí se text: I am alive. Serverová strana nedisponuje grafickým uživatelským rozhraním. Funkčnost můžete otestovat pomocí rest klienta například na adrese http://localhost:8080/AFServer/rest/country/list - content-type: application/json metoda GET.

Nyní je potřeba spustit klientskou část aplikace. Ve složce s Showcase.jar, který je přiložen na CD spusťte java -jar Showcase.jar . Aplikace bude spuštěna.

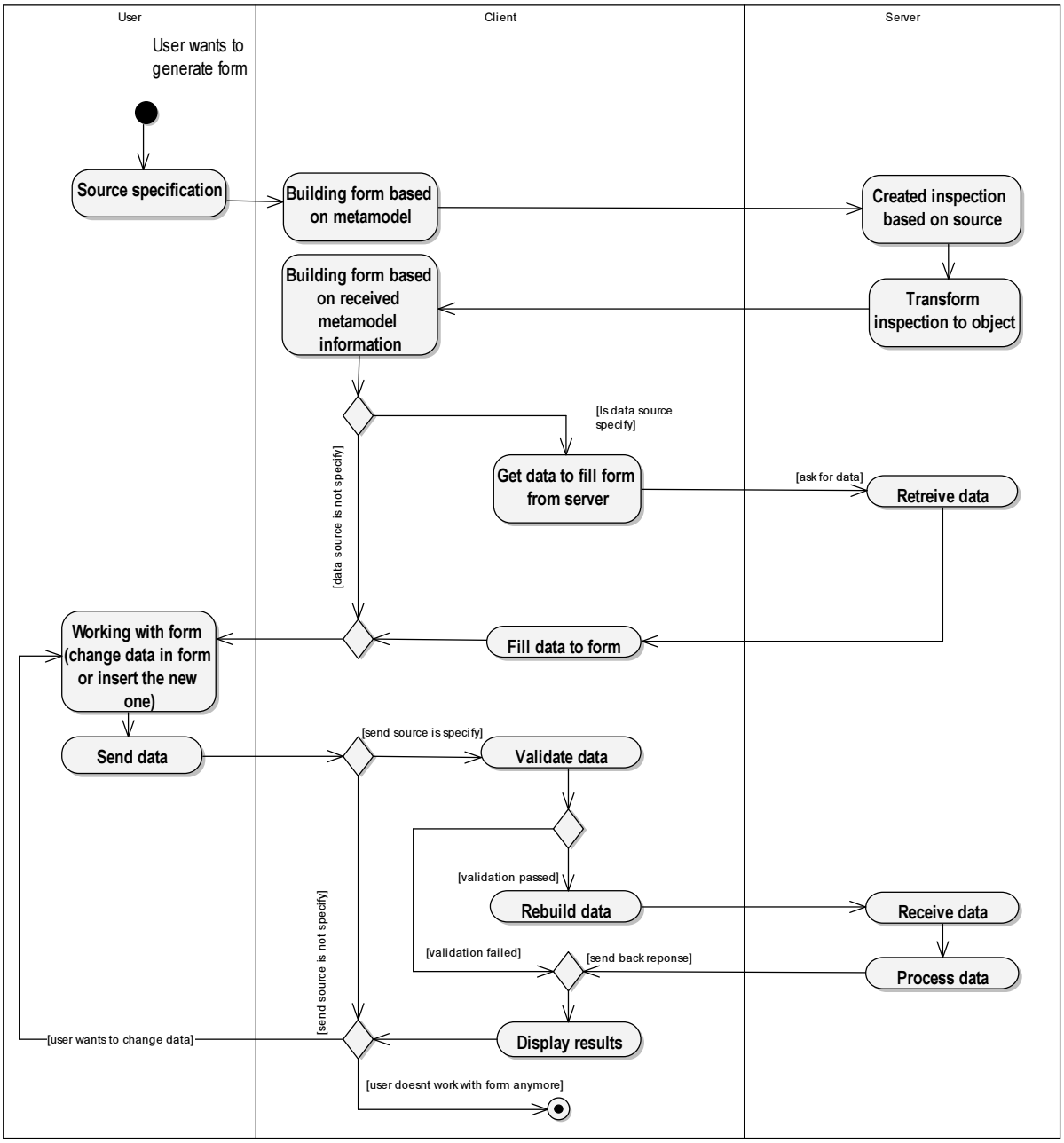
Příloha C

UML diagramy a obrázky

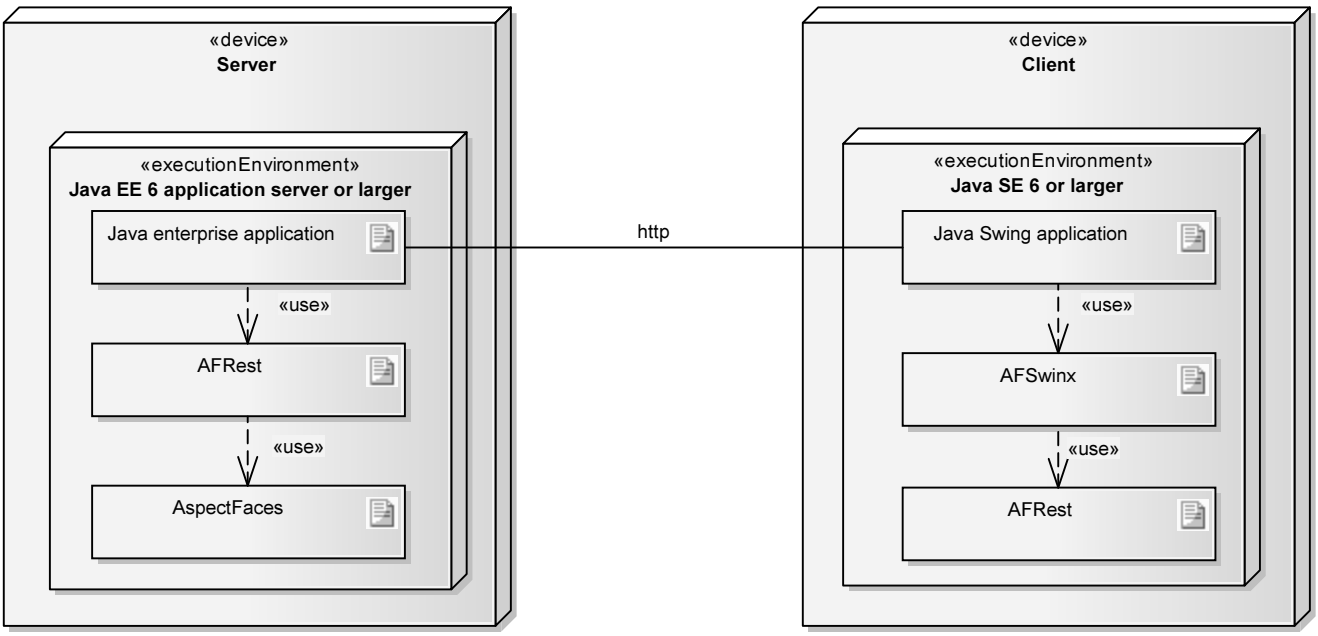
V této sekci naleznete použité UML diagramy a velké obrázky, na které bylo v textu odkazováno.



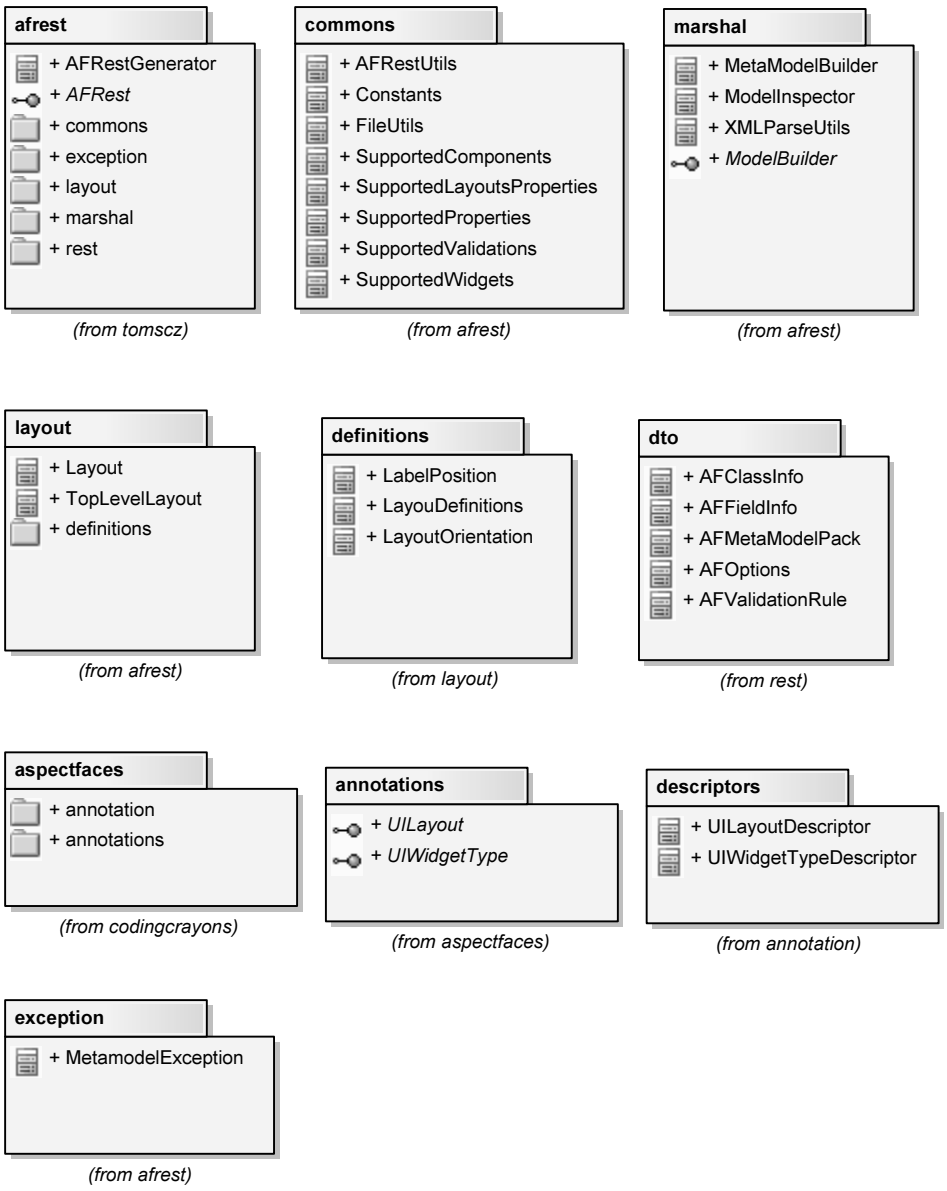
Obrázek C.1: Případy užití frameworku



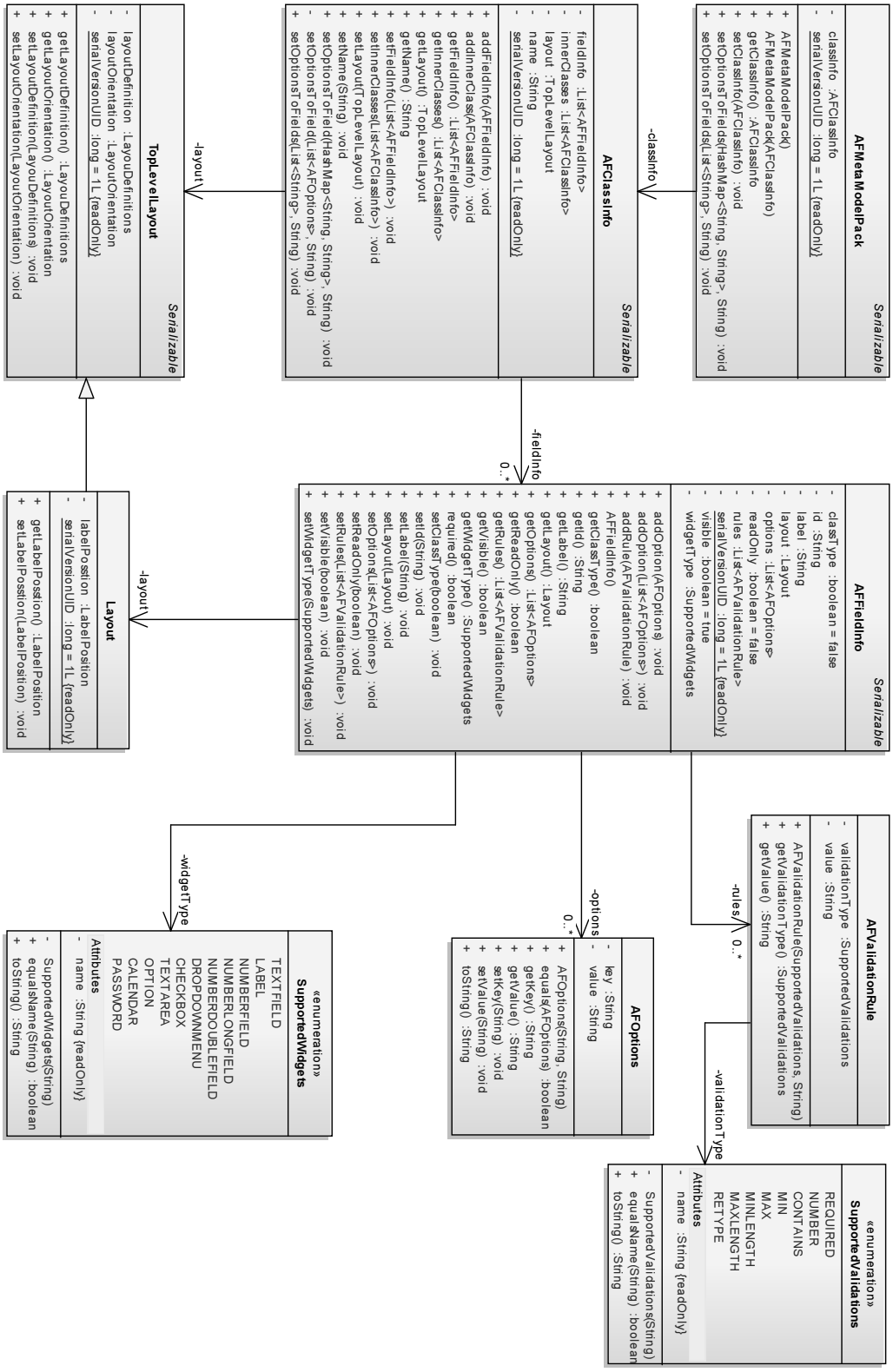
Obrázek C.2: Business model



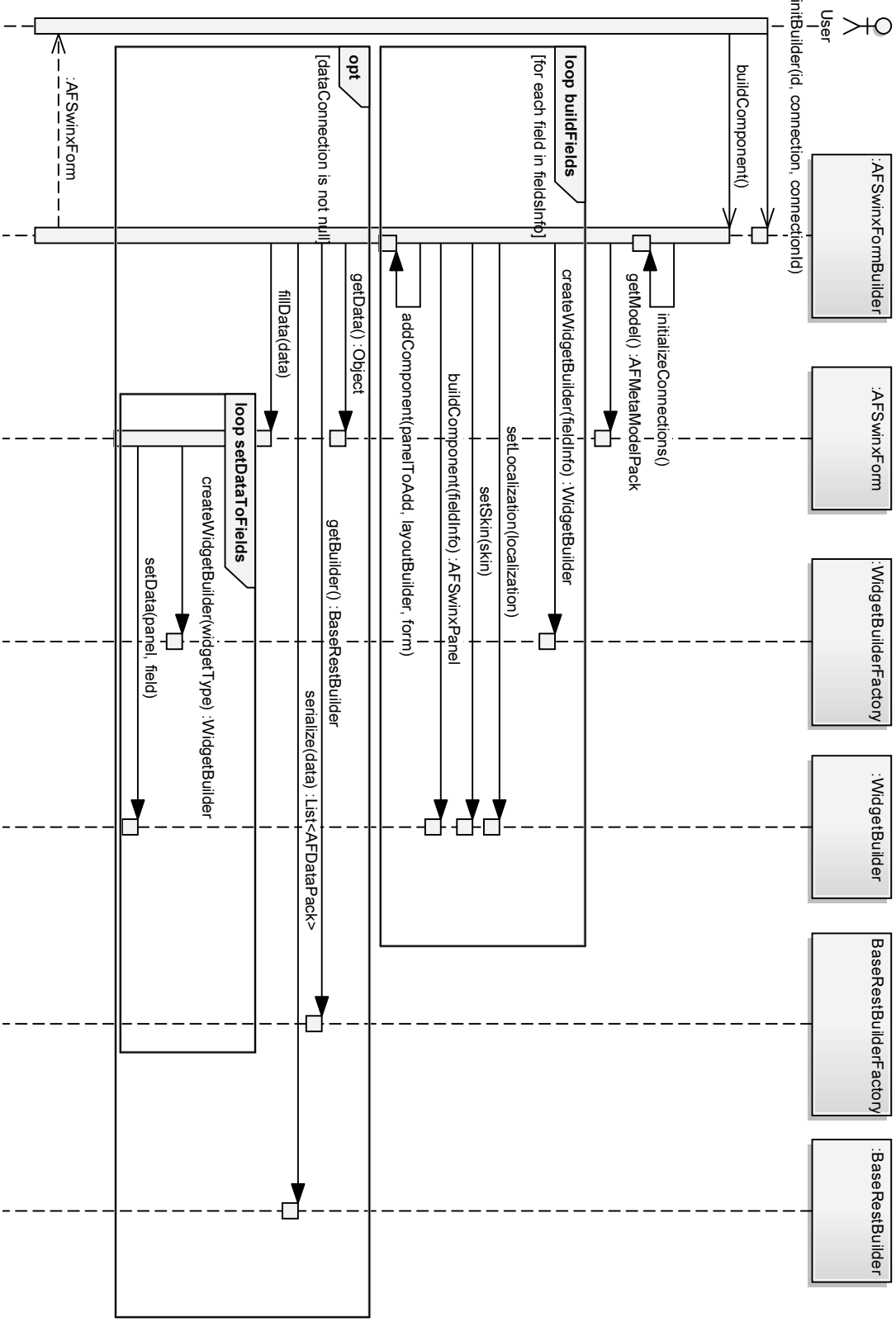
Obrázek C.3: Diagram nasazení



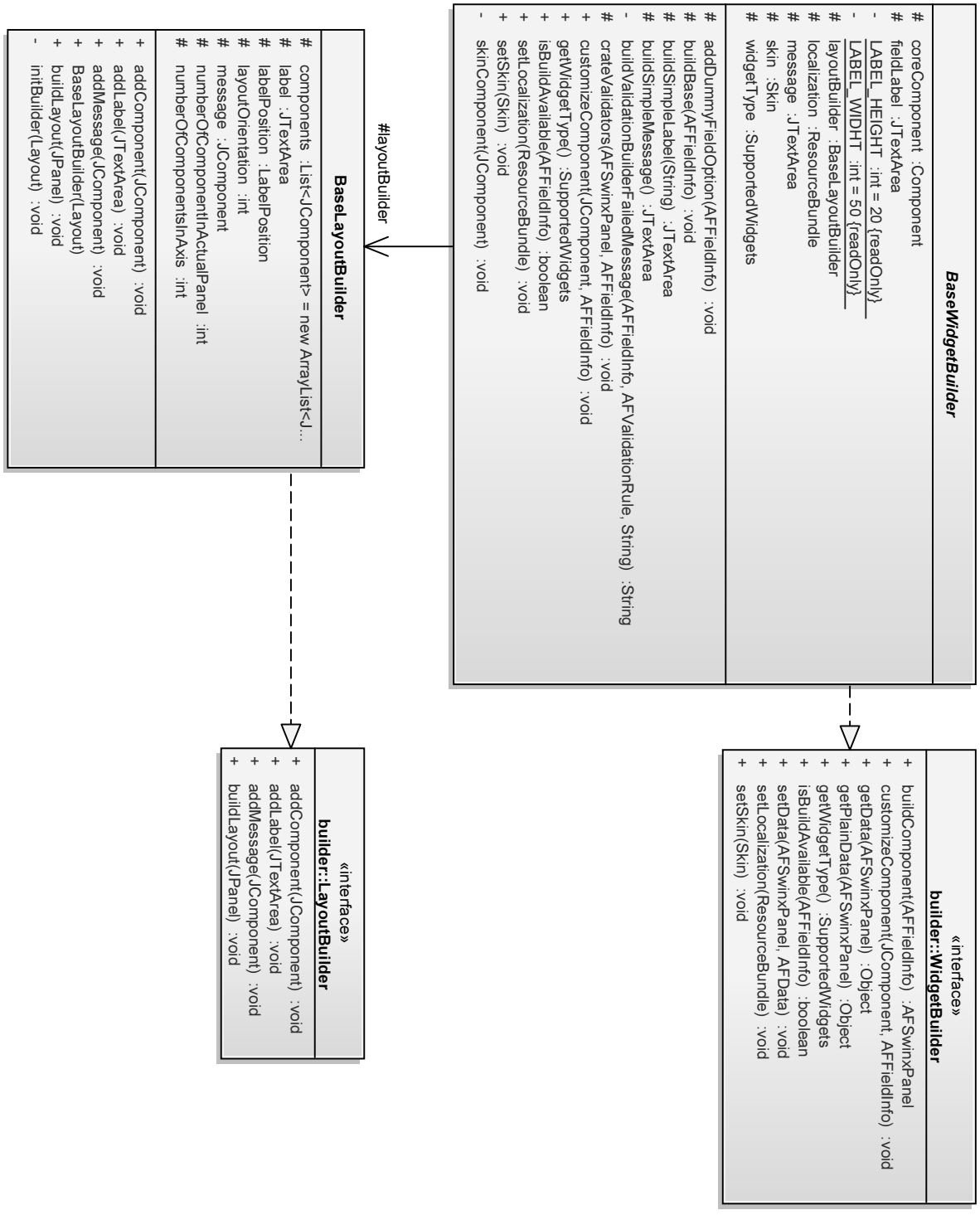
Obrázek C.4: Diagram balíčků a jejich tříd z AFRest



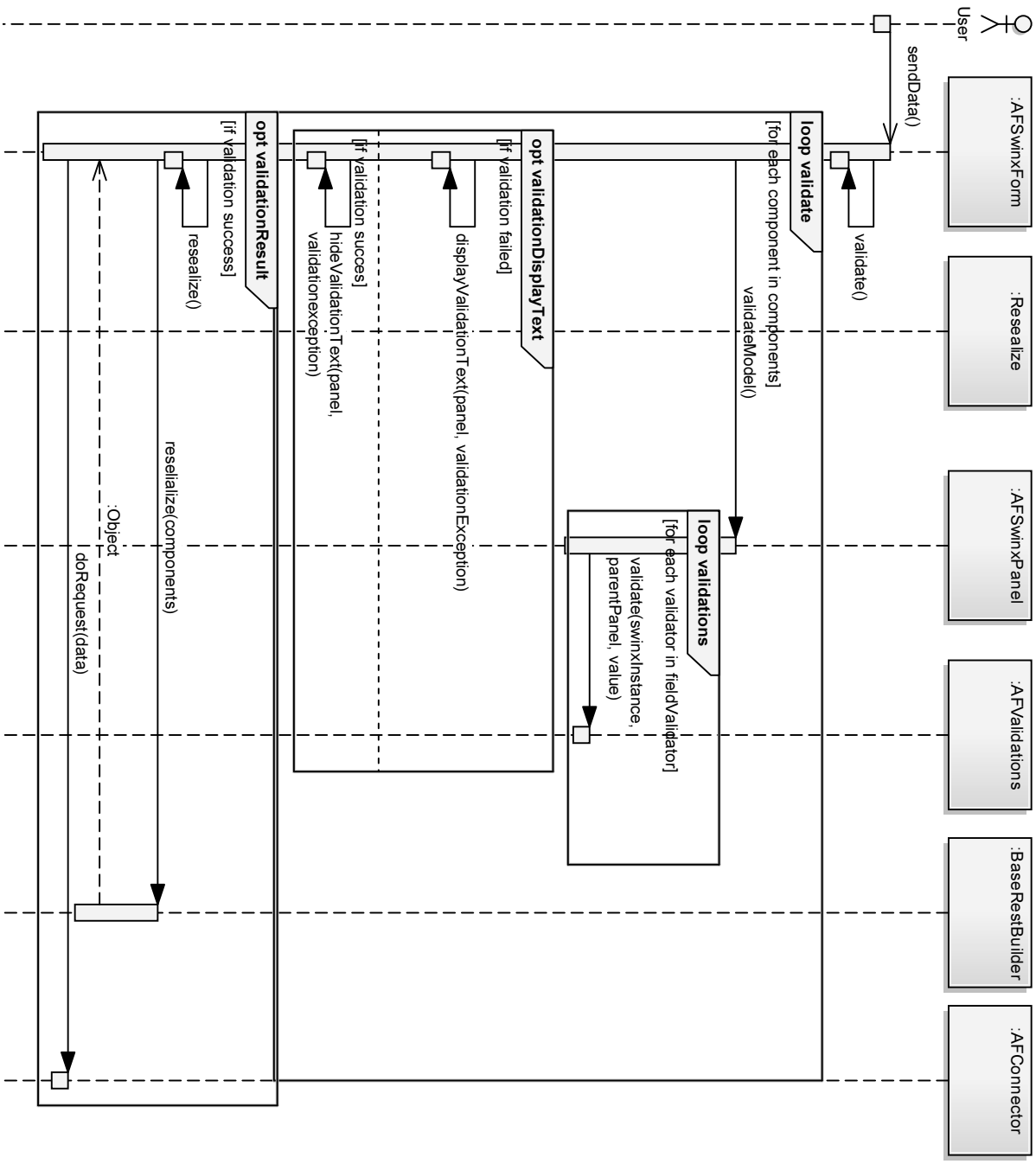
Obrázek C.5: Doménový model obecných definic komponent



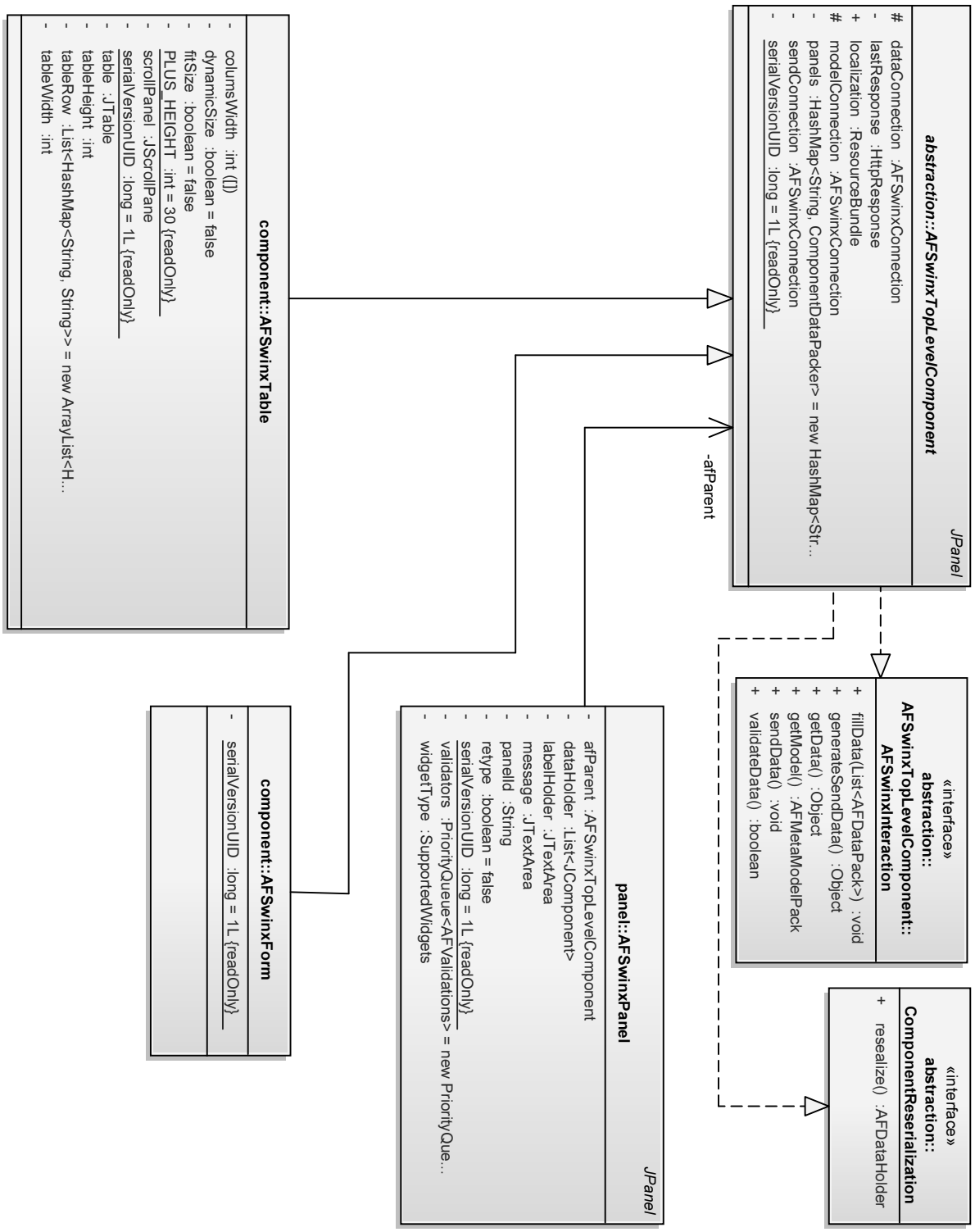
Obrázek C.6: SD diagram sestavení formuláře



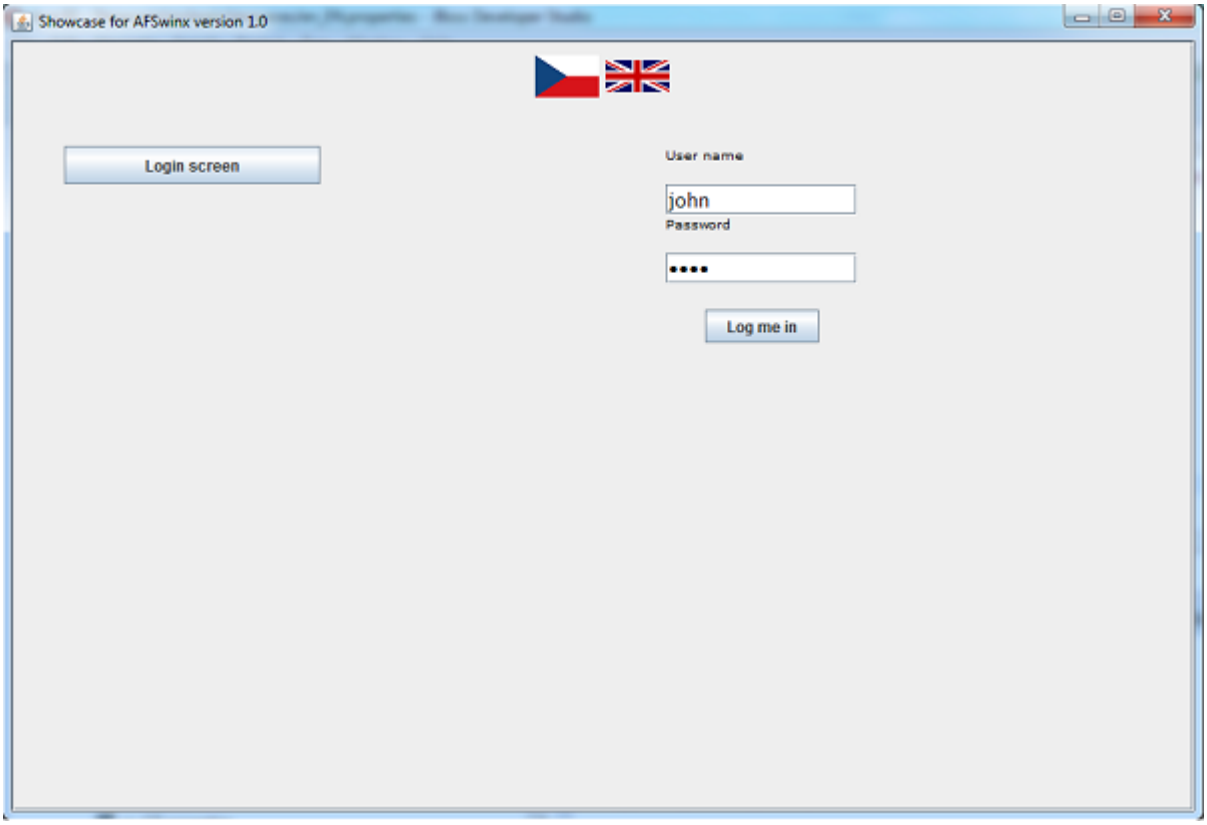
Obrázek C.7: Doménový model znázorňující buildery, které jsou použity při vytváření aktivních prvků



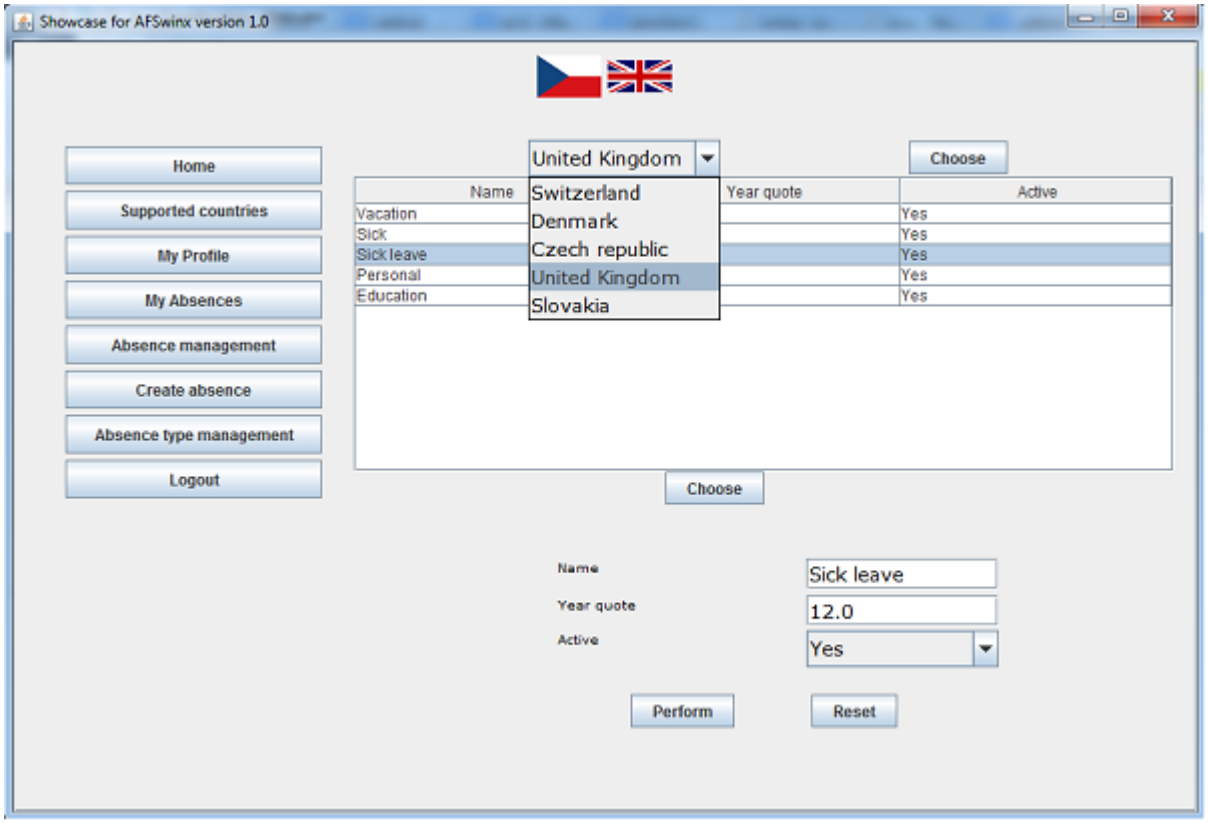
Obrázek C.8: SD diagram odeslání dat na server



Obrázek C.9: Doménový model znázorňující strukturu, které reprzentuje formuláře a tabulky

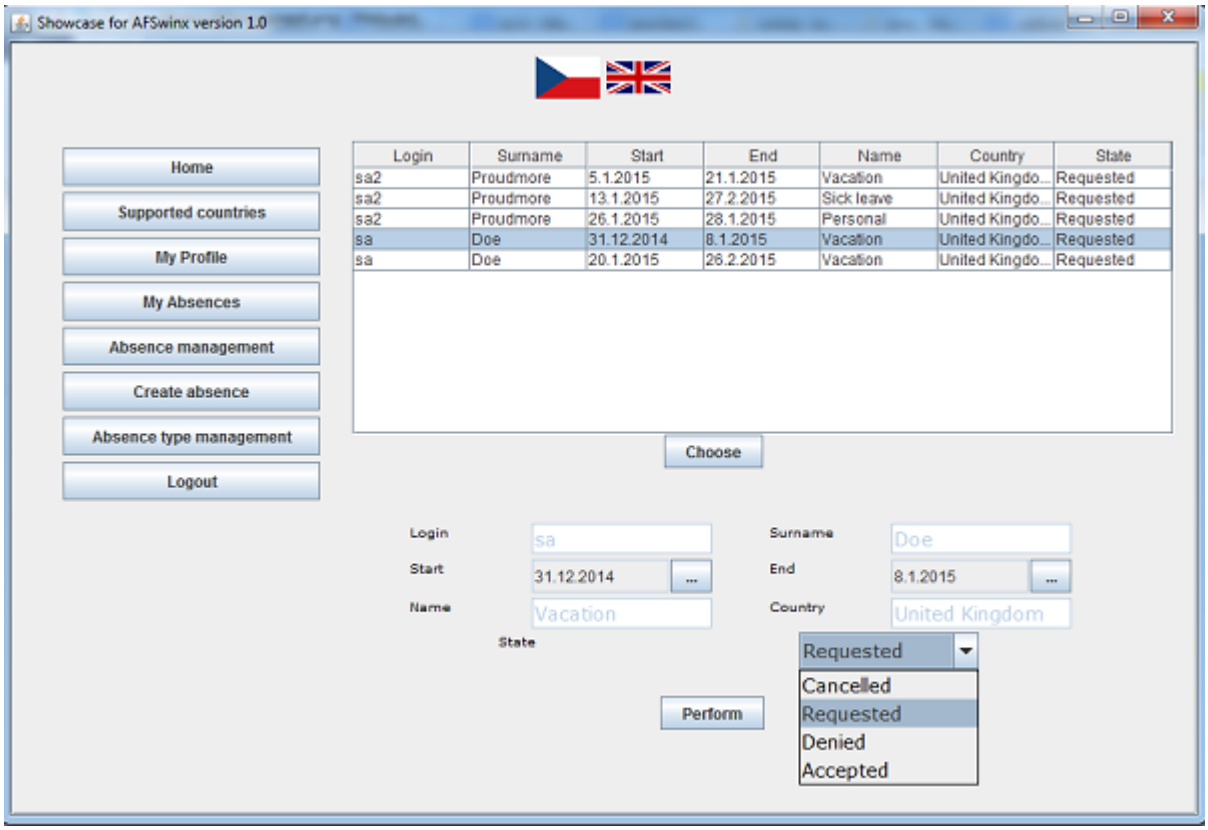


Obrázek C.10: Ukázkový projekt - formulář sloužící k přihlášení.



Obrázek C.11: Ukázkový projekt - správa absenčních typů

Obrázek C.12: Ukázkový projekt - správa absencí z pohledu uživatele



Obrázek C.13: Ukázkový projekt - správa absencí z pohledu administrátora

Příloha D

Ukázky zdrojového kódu a XML souborů

V této sekci naleznete ukázky zdrojového kódu, na které bylo v textu odkazováno.

Část zdrojového kódu D.1: Ukázka definice komponenty

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<afRestEntity xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <entity>
                <entityName>person</entityName>
                <entity>
                        <entityName>address</entityName>
                        <widget>
                                <widgetType />
                                <fieldName>street</fieldName>
                                <label>Street</label>
                                <validations>
                                        <required />
                                </validations>
                                <fieldLayout>
                                        <layoutOrientation />
                                        <labelPossition />
                                        <layout />
                                </fieldLayout>
                        </widget>
                        <fieldName>myAdress</fieldName>
                </entity>
                <widget>
                        <widgetType />
                        <fieldName>firstName</fieldName>
                        <label>person.firstName</label>
                        <validations>
                                <required>true</required>
                        </validations>
                        <fieldLayout>
                                <layoutOrientation />
                                <labelPossition />
                                <layout />
                        </fieldLayout>
```

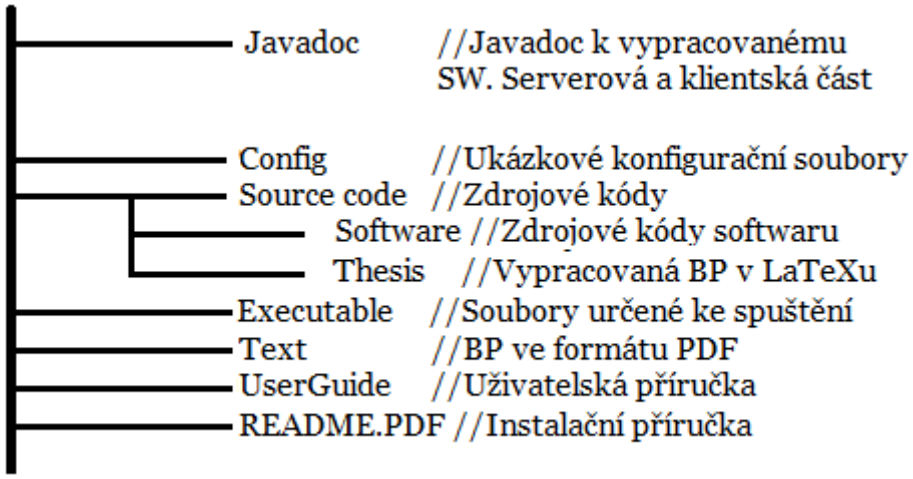
```
                </widget>
                <widget>
                        <widgetType>textArea</widgetType>
                        <fieldName>lastName</fieldName>
                        <label>person.lastName</label>
                        <validations>
                                <required />
                        </validations>
                        <fieldLayout>
                                <layoutOrientation>AxisX</layoutOrientation>
                                <labelPossition>before</labelPossition>
                                <layout>TwoColumnsLayout</layout>
                        </fieldLayout>
                </widget>
                <widget>
                        <widgetType>checkBox</widgetType>
                        <fieldName>confidentialAgreement</fieldName>
                        <label>Confidential Agreement</label>
                        <validations>
                                <required />
                        </validations>
                        <fieldLayout>
                                <layoutOrientation />
                                <labelPossition />
                                <layout />
                        </fieldLayout>
                </widget>
        </entity>
</afRestEntity>
```

Část zdrojového kódu D.2: Ukázka zdroje, sloužícího k vygenerování definice třídy Country

```
@GET
@Path("/definition")
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
@RolesAllowed({"admin"})
public Response getResources(@javax.ws.rs.core.Context HttpServletRequest request) {
        try {
                AFRest afRest = new AFRestGenerator(request.getSession().getServletContext());
                AFMetaModelPack data = afRest.generateSkeleton(Country.class.getCanonicalName());
                return Response.status(Response.Status.OK).entity(data).build();
        } catch (MetamodelException e) {
                return Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
        }
}
```

Příloha E

Obsah přiloženého CD



Obrázek E.1: Obsah přiloženého CD