České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačové grafiky a interakce

Diplomová práce

Aplikace pro organizaci uživatelských dat

Bc. Vrchlavský Petr

Vedoucí práce: **Ing. Ondřej Macek**

Studijní program: Otevřená informatika, magisterský

Obor: Softwarové inženýrství a interakce

9.5.2014

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne …… ……..

Podpis

Zde bude oficiální zadání

Abstrakt

Práce se zabývá otázkou organizace a spravování uživatelských dat. Problematika je podrobena analýze společně s průzkumem trhu zaměřeným na existující řešení s podobnou tématikou. Úsilí je kladeno především na návrh a implementaci webové aplikace, která pomocí uživateli definovanými datovými strukturami s daty manipuluje. Součástí práce je také pohled na vývojový cyklus webové aplikace a podrobný popis jednotlivých částí a problematik, které je nutné při realizaci řešit. Na závěr je představen seznam věcí, kterými lze v budoucnu stávající implementaci obohatit a rozšířit tak možnosti aplikace.

Abstract

This thesis deals with a problematics of organizing and management of user data. At the begining the analysis together with market research for existing solutions with similar theme is made. Effort is put on design and implementation of web application, which manipulates with data by user-defined data structures. This thesis also includes an overview over the development cycle of the web application and detailed description of individual parts and problematics, that must be dealt with. At the end the list of possible features, which could enhance the usability of the web application is introduced.

Poděkování

Rád bych na tomto místě poděkoval své rodině a blízkým přátelům za jejich podporu v průběhu celého mého studia. Velké díky také patří vedoucímu práce panu Ing. Ondřeji Mackovi za jeho pozitivní, věcné a vždy přínosné příspěvky k této diplomové práci.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Svět kolem nás se vyvíjí neuvěřitelně rychlým tempem. Ohlednutím do historie si můžeme všimnout, že úspěšné technologie, které měli fatální dopad na lidské bytí, procházely velmi podobnými životními cykly. Vždy zde bylo období objevení a bádání, které bylo následováno fází masové a rychlé expanze do všech zákoutí lidské činnosti. Stalo se tak s objevením kola, pravděpodobně jedním z nejdůležitějších objevů, jaké lidstvo kdy učinilo. Jinak tomu nebylo ani při vynálezu parního stroje, objevení elektřiny ani penicilínu.

Našim novodobým fenoménem, který už nyní zasahuje do všech sfér lidské činnosti je výpočetní technika. Vytvořili jsme ideální ekosystém pro tok informací. Ve středu tohoto systému má své kořeny počin jménem Internet. Lidstvo nikdy dříve nevytvořilo nic tak komplexního, rychle se vyvíjejícího a pro každého dostupného jako je neuvěřitelně hustá sít všemožných elektronických zařízení, které jsou mezi sebou propojené a umožňuji sdílet a konzumovat informace. Pozoruhodné na tomto fenoménu je jeho schopnost sám sebe plnit daty, čili informacemi.

Tato diplomová práce se zaměřuje na cílené shromaždování a organizování dat. Na trhu se vyskytuje nepřeberné množství aplikací, které pomáhají uživatelům shromaždovat a organizovat data. Proč tedy vytvářet další aplikaci? Je zde nezaplněné místo na trhu, kde se informace pohybují chaoticky a neorganizovaně?

Je poměrně jednoduché se zaměřit na konkrétní podmnožinu dat a vytvořit pro ni aplikaci, která intuitivním způsobem umožní s daty manipulovat. Však také všichni známe emailové klienty, kalendáře s možností vkládání poznámek a událostí, či portály na sdílení mediálního obsahu. Každý z nás má zajisté svojí oblíbenou podmnožinu.

Cílem a výstupem této práce je, umožnit uživateli nadefinovat si takovou to podmnožinu svých zájmu. Výstupem mojí diplomové práce je portál pro organizování uživatelských dat – Portal for organizing user data, ve zkratce POUD.

Druhotným výstupem práce je náhled na vývojový cyklus takovéto aplikace, který bude podrobně popsán a kde se budu snažit zúročit svou několikaletou praxi, co by vývojáře tzv. enterprise aplikací.

Kapitola 2

Rešerše

V této kapitole popíši několik služeb, které nějakým způsobem řeší organizování uživatelských dat či jejich manipulaci/propagaci nebo získáváni. Společnou vlastností je tedy způsob, jakým služby s daty nakládají, a jaké funkce poskytují pro jejich správu. Cílem kapitoly je identifikovat různá řešení dostupná na trhu a porovnat je s POUD aplikací.

2.1 IFTTT

If This Then That služba [1] nabízí prostředí, pomocí kterého můžete definovat a řídit tok dat mezi množinou podporovaných služeb třetích stran. Tato služba je volně dostupná na adrese IFTTT. Pro využití její služeb je třeba se zaregistrovat.

Princip IFTTT spočívá v párování webových služeb (Facebook, Gmail, LinkedIn, Evernote a další…) a definování akcí a reakcí na události v daných službách. Prostředí IFTTT staví na několika základních prvcích. Pojďme se na ně podívat více zblízka.

Obrázek 1: Základní stavební kameny IFTTT služby

Na obrázku 1 je zobrazena základní filozofie služby *IFTTT*. Objekt *Recipe* je základ, pomocí kterého definujete spínače (*Trigger*) a akci (*Action*) pro vámi aktivovanou službu třetích stran (Facebook, Gmail,…). Jedná se v podstatě o výrazový programovací jazyk na vysoké úrovni abstrakce.

Jak to tedy funguje? Při vytváření *Recipe* jste zprvu vyzváni pro výběr služby, na jejíž akce se bude reagovat (*Trigger*). Pro názornost zde použiju službu *Gmail* od společnosti *Google*. Tato služba je široce známá, a hlavně se jedná o emailového klienta, jehož principy jsou v povědomí většiny uživatelů internetu. Vybereme tedy službu Gmail. Je jasné, že pro bezproblémový chod je třeba nějakým způsobem umožnit *IFTTT* přístup do našeho emailového klienta.

Zde na chvíli odbočím. Protože *IFTTT* je služba, nad kterou nemáme kontrolu, respektive nemáme kontrolu ve smyslu, jak je s našimi daty nakládáno a co se s nimi ve skutečnosti na pozadí děje. Je tedy na místě se na chvíli zastavit a uvědomit si, jaká rizika to přináší. Mluvím

zde o ochraně soukromí a citlivých dat. Proto, aby mohla služba *IFTTT* vykonávat to, co vykonávat má, tj. přistupovat k našim službám a reagovat na akce, musí ji být přidělena určitá úroveň přístupu. Tím prakticky zpřístupňujeme naše data neznámo kam. Toto je bohužel daň, kterou musíme zaplatit za využívání služeb tohoto typu. Je tedy na každém z nás, jakou důvěru vkládáme službě, kterou chceme využívat.

Vraťme se zpět k naší ukázce. Vybrali jsme tedy službu *Gmail*. Po odsouhlasení přístupu k našemu účtu můžeme přejít k definování spouštěče, tedy *Triggeru*. Množina *Triggeru* je pevně daná a pro každou službu unikátní, respektive specifická. Pro *Gmail* máme na výběr spouštěče typu, "přišel nový email", "emailu byl přidán štítek" (label), "obdrželi jsme email od specifického odesílatele" a další. Nastavíme spouštěč na „přišel nový email".

Nyní máme nastavenou službu, kterou chceme monitorovat a událost, na kterou chceme reagovat. Dalším krokem je nastavení akce, co se má stát, když… Opět pro názornost zde využiju službu Facebook pro její všeobecnou známost jako „konzumenta". Nastavíme si tedy náš Facebook účet jako protistranu Gmailu, povolíme všechna vyžadovaná přístupová opatření a vybereme akci, která nastane v momentě zachycení události na straně Gmailu. Pro Facebook jsou nabízený akce „vytvoř status", „nahraj fotografii z dané url" a další. Vybereme akci „vytvoř status".

Posledním krokem je nastavení, jak má status vypadat, jaká data má obsahovat. K tomu nám IFTTT nabízí funkcionalitu nazvanou *Ingredient*. *Ingredient* ve svém základu nedělá nic jiného, než že mapuje data z *Triggeru* na výstup, v našem případě zpráva na *zdi Facebooku*. Data, která jsou zde nabízena, opět závisí na službě, kterou jsme si zvolili (*Gmail* v našem případě). Vybráním *Ingredientu* Subject a *FromAddress* a doplněním o libovolný text, docílíme toho, že v momentě, kdy obdržíme email, se nám vytvoří status na *Facebooku* o příchozí poště obsahující informace o odesílateli a předmětu emailu.

Výše popsanými kroky jsme vytvořili jednoduchý Recipe, který se odteď postará o propagaci dat z jedné služby do druhé. To je v podstatě vše, co služba IFTTT nabízí, ještě vedle samozřejmostí jako správa našich Recepie, máme možnost si vybrat z opravdu velkého množství předdefinovaných Recipe, které jsou seřazeny podle různých kritérií. Příkladem může být kritérium oblíbenosti u ostatních uživatelů služby IFTTT. Pro zajímavost, nejvyužívanější Recipe k dubnu 2014 je zálohování kontaktů z iOS (operační systém v mobilních zařízení od společnosti Apple) do tabulkového dokumentu služby Google Drive (Google).

2.2 Evernote

Evernote [2] se vydává trochu jiným směrem než IFTTT služba. Tato služba byla vytvořena za účelem pořizování nejrůznějších poznámek s možností jejich archivace. Je možné vytvořit několik druhů poznámek. Mezi základní druhy patří textové poznámky v podobě formátovaného textu, fotografie či mluvené slovo. Službu naleznete na adrese Evernote.

Hlavním rozdílem mezi IFTTT a Evernote je ve zdroji, odkud čerpají data. Zatímco IFTTT v podstatě bere data již hotová a před propagací do jiné služby je může pozměnit, u Evernote je zapotřebí data ručně dodat. Není tedy potřeba udělovat oprávnění k přístupu k jiným účtům jiných služeb.

Samozřejmostí u Evernote je archivace na serveru poskytovatele služby a synchronizace s různými zařízeními. Mezi těmito zařízeními nechybí mobilní zařízení ať už pro operační systémy Android či iOS, nebo desktopové řešení Windows OS.

Služba je velmi jednoduchá na ovládání a vytvoření poznámky zabere málo času. Pro vytvoření poznámky stačí poklepat na tlačítko New Note a prázdná poznámka je vytvořena. Pak už stačí do poznámky vložit obsah podle potřeby. Tvoření obsahu je velmi podobné psaní textového dokumentu v programu typu Word. Můžeme psát formátovaný text, můžeme přidávat přílohy, fotografie či zvukové záznamy.

Velikou výhodou služby je již zmíněná synchronizace, díky níž můžeme mít své poznámky prakticky vždy při ruce. Synchronizace lze nastavit na automatickou či manuální. Podporován je i offline mód, kdy máme přístupné poznámky od poslední synchronizace bez nutnosti být připojený k internetu. Můžeme k těmto poznámkám přistupovat, měnit je či přidávat další. Nevýhodnou synchronizace je, pokud nemáme přístup k internetu, nemůžeme propagovat naše změny do všech ostatních zařízení, na kterých Evernote používáme a naopak nemůžeme stáhnout aktuální data z jiných zařízení.

2.3 Feedreader

Feedreader představuje zástupce tzv. feed čteček. Věřím, že není potřeba představovat tuto technologii, a proto přejdu rovnou k představení implementace Feedreader. Služba je zdarma k dispozici na adrese Feedreader.

Čecky feedů, jsou obecně jednoduché služby a jinak tomu není ani v případě Feedreaderu. Služba nabízí standardní sadu operací, mezi které patří vytváření feedů, tedy definování zdrojů, odkud se mají čerpat data, kategorizování feedů, mazání feedů či jednoduché filtrování. Jakmile definujete feed, okamžitě se vám načtou data z daného zdroje. Tyto data jsou zobrazeny jako položky v seznamu přijatých feedů. Položky lze rozbalit pro získání detailu a po rozbalení je možné přejít přímo ke zdroji a zobrazit si kompletní informace.

2.4 Shrnutí

Představili jsme si 3 služby, které nějakým způsobem nakládají s daty uživatele, či data uživateli poskytují. Tyto 3 služby jsem vybral záměrně a nyní se je pokusím kategorizovat s pohledu správy dat.

IFTTT služba se chová k datům jako jakýsi prostředník mezi ostatními službami. Bere data z jedné služby, v případě potřeby je transformuje a propaguje data do služby druhé.

Evernote služba naopak střádá data, která jsou uživateli vkládána ručně. Nedílnou součástí je synchronizace a podpora různých zařízení, počínaje webovou službou a konče mobilními zařízeními.

Feedreader služba data ani netransformuje, ani nepropaguje, dokonce ani nearchivuje, pouze je zobrazuje.

Představili jsme si zde tři různé přístupy k manipulaci či správě dat. Každá ze zmíněných služeb má konkrétní zaměření, které se snaží nějakým způsobem pokrýt svojí implementací. Společný

znak všech tří služeb je poskytování předem jasné sady funkcionality a nástrojů ke správě dat a tím i definovat poměrně úzké možnosti jejich využití.

Cílem tohoto průzkumu trhu bylo poskytnout přehled nad různými řešeními, která se na trhu vyskytují, popsat jejich základní zaměření a možnosti. Díky tomu je nyní možné se pokusit navrhnout jiné řešení, které bude mít odlišný přístup ke správě uživatelských dat, a poskytnout tak funkcionalitu, která je svým způsobem unikátní a řeší to, co jiné služby neřeší.

Kapitola 3

Systémové požadavky

V počátcích životního cyklu každé aplikace nesmí chybět specifikace a analýza požadavků. Podle charakteru aplikace se můžeme setkat s různými zdroji požadavku. Požadavky k vývojovému týmu mohou proudit například ze strany tzv. stakeholderů, kteří své myšlenky a plánované funkce propagují k vývojářům skrze projektového manažera. Nejprve vznikají velmi abstraktní požadavky, které říkají, jaký problém má aplikace vyřešit. V raných fázích projektu, kdy ještě ani nemusí být sestavený vývojový tým, bývají požadavky popsány na vysoké úrovni abstrakce a často ještě nezahrnují žádné implementační detaily. Mohou se zde však již vyskytovat náznaky infrastruktury či jednoduché integrační diagramy.

V závislosti na zvolené metodologii vývoje se mohou požadavky na cestě od zadavatelů k vývojářům pozměnit. Z pravidla to bývají pouze detaily, občas se ale může jednat i o zásadní rozhodnutí, jakými může být výběr technologie. Například v metodologii SCRUM [3] se provádí tzv. sprint planning schůzky, jejichž cílem je právě dospecifikování požadavků a především zadaní konkrétní práce vývojovému týmu. Vývojový tým se zaváže splněním určité množiny funkcionality, kterou po určité době předvede zadavatelům. Následuje další sprint palnning schůzka a celé se to opakuje až do doby, kdy jsou všechny podstatné požadavky implementovány.

V případě této diplomové práce se jedná o velmi podobný scénář. Požadavky jsou nejprve specifikovány velmi abstraktně v podobě zadání diplomové práce, které je v prvotní fázi schválené patřičnými orgány školy. V momentě schválení zadání přichází na řadu bližší specifikace zadání, kterou v ideálním případě schválí vedoucí diplomové práce. Zde lze vidět analogie mezi projekt manažerem a vedoucím DP.

Tato diplomová práce nemá ze zadání určenou žádnou metodologii vývoje a to ani na úrovni, zda se bude k vývoji přistupovat agilně, či jiným způsobem. Charakter této práce však vybízí řídit se principy vodopádového vývoje, z kterého může vytěžit výhody, které tento přístup k vývoji přináší. Jedná se především o jasné a v průběhu projektu neměnné požadavky, které jsou specifikované již v počátečních fázích projektu.

Protože se neočekává, že by se požadavky v průběhu implementace měnili, či přibývaly nebo ubývaly, je moudré se z počátku na analýzu požadavků pečlivě zaměřit a udělat si celkový a jasný přehled toho, co má být na konci vývoje doručeno.

Z výše řečeného se proto požadavky zabývám již v počátku, a věnuji jim celou jednu kapitolu. Pojďme se tedy na ně podívat více zblízka.

Dříve než začnu vypisovat konkrétní funkční požadavky, rekapituluju zde zadání. Z definice zadání vyplývá, že cílem této práce je vytvořit portál, který umožní jeho uživatelům organizovat si data, která jsou ve středu jejich zájmů. Dále je v zadání specifikováno, že pro docílení kýžené

aplikace mají být použity volně přístupné technologie, zejména pak programovací jazyk Java. Z pohledu funkčních požadavku vyplívá, že data můžou být do aplikace vkládána buďto ručně, či nějakým automatizovaným mechanismem. Pro vkládání dat mají být použity formuláře, které definují formu vkládaných dat. Posledním významným požadavkem je umožnit uživatelům definovat vazby mezi těmito formuláři, aby bylo možné čerpat data z jedné struktury do druhé.

Protože takto abstraktně definované zadání je nedostatečné pro konkrétní implementační, infrastrukturní či jiná rozhodnutí technického rázu, je třeba ze zadání extrahovat více detailní popis. Tomuto procesu říkáme analýza požadavků, jejž artefaktem jsou jednotlivé požadavky. Pokud z analýzy požadavků nedostaneme dostatečně detailní a konkrétní popisy dílčích částí systému a zamýšlených funkcionalit, můžeme požadavky dále dělit na menší celky, které jsou v žargonu IT pojmenovány různě, a zpravidla se toto pojmenování řídí zvolenou metodologií vývoje (např. story, task, issue atd.). Mým cílem je však požadavky identifikovat a popsat na dostatečně detailní úrovni, aby nebylo zapotřebí dále vytvářet vetší granularitu.

3.1 Analýza požadavků

Pro přehlednost rozdělím požadavky do několika ucelených částí, kterými nakonec pokryji celý zamýšlený systém. Pro usnadnění zde budu používat zkratku POUD, což je akronym pro Portal for Organizing User Data.

3.1.1 Uživatelé

[1] POUD umožní vytvářet uživatelské účty. Pro vytvoření uživatelského účtu bude potřeba, aby uživatel vyplnil unikátní uživatelské jméno alespoň délky třech znaků a heslo.

[2] POUD bude po uživatelích vyžadovat přihlášení pro zpřístupnění základních funkcí. Mezi tyto funkce patří správa formulářů, správa dat vložených pomocí formulářů, zobrazovaní dat a přístup pomocí webové služby.

3.1.2 Formuláře

[3] Formulář je základní stavební jednotkou POUD a každému uživateli bude umožněno tyto formuláře vytvářet a spravovat.

[4] Každý formulář bude mít právě jednoho vlastníka, který k němu bude mít výhradní právo. Tj. pouze vlastník může pomocí svých formulářů vkládat data do systému a pouze vlastník, muže s těmito daty manipulovat a zobrazovat je.

[5] POUD umožní uživateli definovat libovolné množství formulářů, u kterých bude moci definovat jejich strukturu a tím pádem i strukturu dat, které bude pomocí těchto formulářů do systému vkládat.

[6] Každý formulář bude moci obsahovat textová pole, která umožní vkládání obecného neformátovaného textu.

[7] Každý formulář bude moci obsahovat pole pro definování data.

[8] Každý formulář bude moci obsahovat pole pro vkládání atomických logických výrazů, tedy tzv. checkbox prvky.

[9] Uživatel bude moci pro každý prvek ve formuláři zvolit jeho pořadí a popisek. Toto se odrazí na vzhledu formuláře ať už při vkládání dat, tak při čtení dat.

[10] Formuláře budou moci také obsahovat kompozitní prvky, které se mohou skládat z libovolného počtu jednoduchých prvků (textová pole, data a checkboxy). Tímto se docílí vytváření datových struktur podobných tabulkám na úrovni formulářů.

[11] Pro každý jednoduchý prvek, ať už přímo na úrovni formuláře nebo na úrovni kompozitních prvků, bude uživateli umožněno definovat tzv. reference. Referencí se myslí odkaz na jiný prvek libovolného formuláře. Tato reference bude sloužit pro specifikování prostoru, ze kterého mohou být pro daný prvek čerpaná data.

3.1.3 Adaptéry

[12] Formulář představí funkcionalitu automatického vkládání dat pomocí tzv. adaptérů. Každý adaptér se bude zaměřovat na jeden obecný, ale unifikovaný zdroj dat. Tedy například na čtení RSS feedů z libovolného zdroje.

[13] Každý adaptér bude jasně definovat rozhraní výstupu, který bude moci být mapován na libovolný formulář.

[14] Každému adaptéru bude možno přidělit časovač, který bude v zadaném intervalu automatický vkládat do systému data ze zdrojů charakteristických daným adaptéru.

[15] Všechny adaptéry budou přístupné všem uživatelům. Každý uživatel bude mít svá specifická nastavení, tedy svá mapovaní na své formuláře a své vlastní časovače.

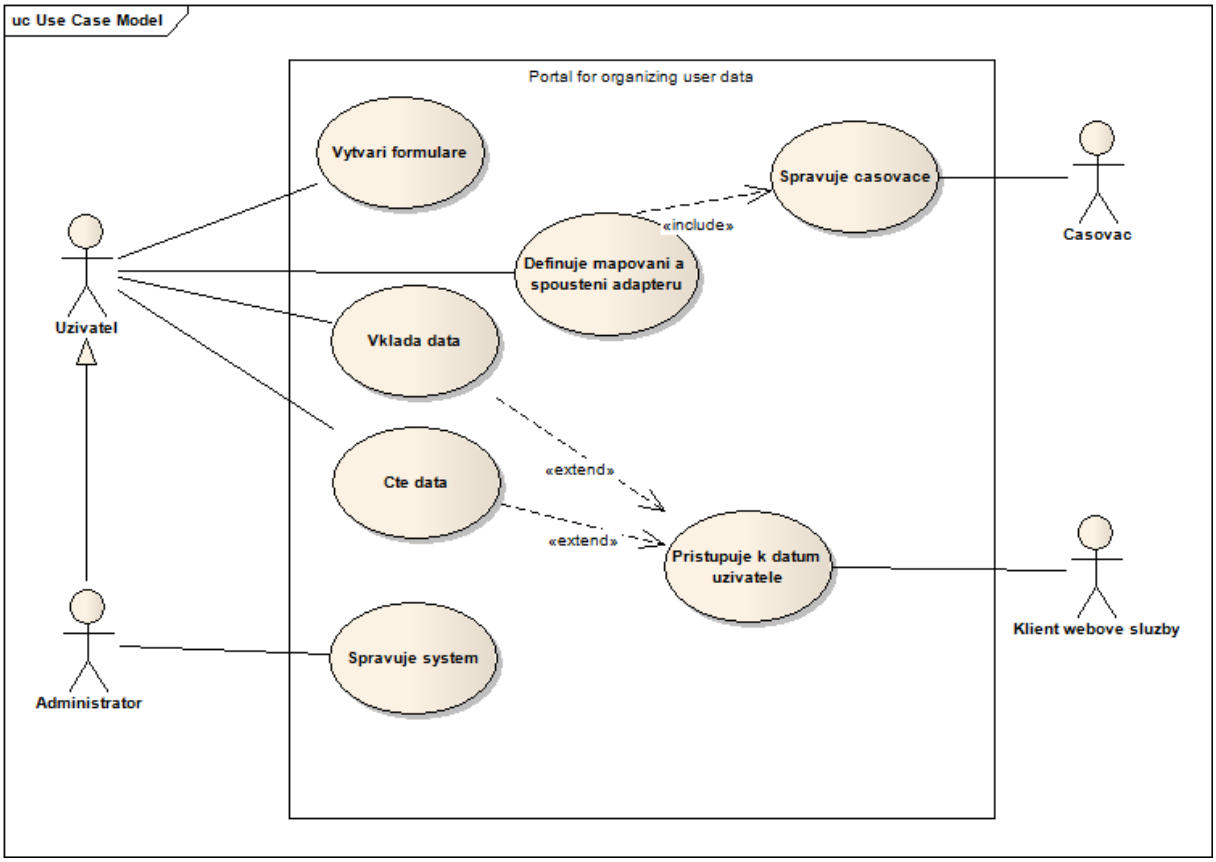
3.1.4 API

[16] Každý uživatel bude moci přistupovat ke svým datům skrze webovou službu.

[17] Webová služba poskytne rozhraní, které umožní číst libovolná uživatelská data a zároveň umožní data vkládat. Vkládaná data musí respektovat vždy jednu formu konkrétního formuláře.

Pro lepší pochopení konceptu POUD, zde přidávám jednoduchý use-case diagram, který identifikuje hlavní aktéry a hlavní případy užití.

Pro lepší pochopení konceptu POUD, zde přidávám jednoduchý use-case diagram, který identifikuje hlavní aktéry a hlavní případy užití.



Obrázek 2: Abstraktní use-case diagram aplikace POUD

3.2 Od požadavků k realizaci

Programování, neboli abstraktněji implementaci, předchází analýza a návrh. Jim Arlow ve své knize [4] popisuje metodiku vývoje Unified Process - UP, kde rozděluje životní cyklus aplikace na 5 základních celků (Požadavky, Analýza, Návrh, Implementace, Testování). Metodika UP, a její komerční verze RUP byly vyvinuty, aby usnadnily vývoj projektů prakticky jakéhokoliv rozsahu. UP a RUP jsou specifikované poměrně obecně a jsou zaměřeny na skutečnost, že se podmínky a zejména požadavky mohou v průběhu životního cyklu měnit. Dobrou vlastností těchto metodik je jejich míra flexibility a také fakt, že si je každý projekt může přizpůsobit svým potřebám.

Těmito metodikami jsem se inspiroval i u této práce a vybral jsem si některé vlastnosti a procesy, které mi pomohou a v minulosti již pomohli řídit vývoj softwaru. V předchozí kapitole jsem zmínil vodopádovou metodiku vývoje a to, jakým způsobem ji mohu využít u této práce. I přestože charakter mé diplomové práce v podstatě vybízí se řídit vodopádovým modelem, našel jsem zde prostor pro kombinaci určitých aspektů i z pohledů jiných metodik.

Z vodopádového modelu jsem využil výhod, které přináší specifikace a analýza požadavku již v prvotních fázích projektu. Nicméně v průběhu práce na POUD aplikaci bylo zapotřebí několikrát přehodnotit stávající návrh a dokonce i změnit některé již implementované části, proto jsem začal vodopádový model více „ohýbat“ směrem k principům podobným agilním metodikám.

Již v minulosti se mi osvědčilo řízení projektu ve smyslu metodiky SCRUM. Zejména pak tzv. sprint planing schůzky a sprint review schuzky, které se v pravidelných iterací opakují. Podstata těchto organizovaných schůzek, kde se setkávají vývojové týmy se zadavateli a tzv. stakeholdery, tedy osoby a subjekty, které poskytují prostředky pro realizaci projektu, je v tom, aby jednak vývojové týmy předvedli dosavadní vykonanou práci, a jednak, aby měli zadavatelé přehled o tom, v jakém stavu se projekt nachází a mohli dále a lépe projekt řídit. Podobných principů jsme společně s vedoucím této diplomové práce využili i u POUD aplikace. Snaha byla si v rozumě dlouhých intervalech předávat informace ohledně dosavadně odvedené práce a nazpět získávat pohled a názory vedoucího DP práce. Z těchto schůzek zpravidla vzešli potřeby některé požadavky pozměnit a nové přidat. Z těchto důvodu jsem přistoupil k vývoji více iterativněji a agilněji. Z prvotního vodopádového modelu jsem tak přešel na kombinaci s iterativním vývojem, kde jsem v podstatě z jednoho „vodopádu“ udělal více menších.

V této kapitole a ani v dalších nebudu chronologicky popisovat ani jinak vyjadřovat, jak se požadavky, analýza či implementace v průběhu vypracování toto práce měnili, ale budu k popisu jednotlivých fází přistupovat tak, jako by byly od počátku zpracovány kompletně a v průběhu času nedocházelo k žádným změnám.

Kapitola 4

Analýza

Analýzu požadavků rozdělím na několik podkapitol, které zastřešují logické celky aplikace. Jednotlivé celky jsou více či méně propojeny, ale i přesto je lze popsat odděleně a nezávisle na sobě.

4.1 Uživatelé

Zadání ani požadavky nijak detailně nepopisují, na jakých principech má uživatelský management fungovat. Budu se tedy řídit obecnými a osvědčenými praktikami.

Každý uživatelský učet potřebuje pro minimální funkčnost nějaký identifikátor uživatele a zabezpečovací prvek, které zabrání ostatním uživatelům se vydávat za uživatele jiné. Říkejme těmto prvkům přihlašovací jméno a heslo.

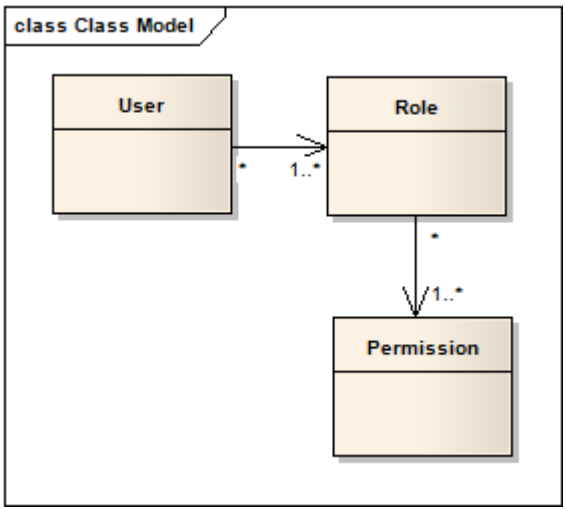
U uživatelského jména je zapotřebí zajistit jeho unikátnost, což není nijak složitá záležitost a lze ji vyřešit mnoho způsoby. U POUD aplikace jsem zvolil řešení pomocí databázového omezení, tedy tzv. UNIQUE Constraint. Výhoda tohoto přístupu je v jeho robustnosti. Podmínku nelze nijak obejít na vyšších vrstvách aplikace, které s databází pracují. Nevýhodou je pak omezení možností, které můžeme pomocí business logiky zachytit. To nám ale v tomto případě nevadí.

Správa uživatelského hesla je již složitější záležitost. Jednak chceme pomocí hesla zajistit, aby se nikdo nemohl vydávat za jiného uživatele a jednak chceme uživatelům poskytnout mechanismus, jak si chránit vlastní data.

Při správě uživatelských hesel je osvědčenou a dnes již nezbytnou praktikou chránit hesla i před samotnou aplikací a tedy i jejich správci. Aby nebylo možné heslo odcizit ani s přímým přístupem do databáze, budou hesla ukládána v podobě jejich hashů. Pro ještě větší bezpečnost bude použit princip tzv. password salt. Tento postup spočívá v tom, že před výpočtem hashe z hesla (ať už při jeho vytváření, nebo při přihlašování) je k heslu přidán text, který je znám pouze aplikaci. Hlavním cílem tohoto mechanismu je zabránit slovníkovým útokům. Dalším důvodem, proč použít password salt, je neochota uživatelů používat pro různé systémy různá hesla. Pokud by došlo k úspěšnému odcizení hesel, například pomocí SQL Injection útoku, nebude možné pro konkrétního uživatele zjistit jeho skutečné heslo a zneužít ho v jiném systému. Nemluvě o tom, že prolomení jednocestných moderních hashovacích funkcí je prakticky výpočetně nemožné.

Aplikace POUD nebude sice ve své základní implementaci nijak zásadně využívat hierarchii uživatelů, nicméně pro budoucí využití a potencionální snadné rozšíření o nové části systému zde zavedu uživatelské role a oprávnění. V kapitole zabývající se implementačními detaily

popíšu tento mechanismus více dopodrobna včetně vlastností a možností, které použité technologie nabízí, ale pro účely analýzy zde představím pouze jednoduchý diagram tříd.



Obrázek 3: Uživatele, role a oprávnění

Na obrázku 3 je znázorněna vazba mezi uživatelem, rolemi a oprávněními. Tento jednoduchý model umožní v budoucnu definovat komplexní řízení uživatelských práv. Hlavní vlastností je vazba mezi rolemi a oprávněními (Role & Permission), které je M:N. Díky tomuto bude možné vytvářet skupiny uživatelů, které mají v určitých částech systému některá oprávnění stejná, zatímco v jiných částech systému mají omezení různá. Rozhodnutí, zda má konkrétní uživatel právo pracovat s určitými zdroji či objekty, bude prováděno na úrovni servisních vrstev, popřípadě na vrstvě prezentační. Zde především pak pro rozhodování, které grafické prvky uživateli zpřístupnit a které ne. Příkladem mohou být položky v menu.

4.2 Formuláře

Jádrem a pomyslným středem POUD aplikace jsou formuláře. V podstatě všechny ostatní části systému s formuláři do určité míry interagují. Formuláře představují mechanismus, pro definování struktury dat, s kterými pak zbytek aplikace a také uživatelé pracují. Účel formulářů je zajistit následující:

- Definovat datové struktury
- Zajistit konzistenci a typovou bezpečnost dat při jejich vkládání
- Umožnit strukturované čtení dat
- Vytvářet vazby mezi formuláři

4.2.1 Formulář jako datová struktura

Pro vytváření komplexních datových struktur není potřeba vymýšlet žádné složité mechanismy. V podstatě si vystačíme s několika základními datovými typy a zavedením jednoduchých vazeb mezi jednotlivými částmi formuláře. V následujících odstavcích popíši, z čeho se formulář skládá.

Datové typy

Formulář definuje množinu elementů. Každý element představuje objekt, pomocí kterého můžeme manipulovat s daty. Základní typy manipulace jsou ukládání dat a čtení dat. Formulářový element si drží informaci, s jakým typem dat může pracovat, jak se element jmenuje či jakou má pozici v rámci formuláře (využívá se pro vkládání dat a zobrazování dat).

Předtím, než blíže formuláře popíši, je třeba si ujasnit několik základních termínů a vymezit jejich význam.

Formulářový záznam je množina dat, uložená pomocí konkrétního formuláře. Takováto množina má danou jasnou strukturu, která vychází ze struktury samotného formuláře. Jako příklad může být formulář pro vkládání jména. Takovýto formulář obsahuje dvě textová pole, jedno pro křestní jméno a druhé pro příjmení. Data vložená pomocí tohoto formuláře se pak vždy skládá z dvou textů. Formulářový záznam lze číst pouze pomocí formuláře, kterým byl záznam vytvořen.

Formulářový element je prvek formuláře, který definuje dílčí strukturu datové struktury formuláře. Pomocí formulářového elementu se provádí skutečné vkládání dat. Elementy tedy definují, o jaké datové typy se jedná, jak jsou data v rámci formuláře uspořádána či jakým způsobem se mají data při čtení prezentovat.

Formulář je množina formulářových elementů. Mimo elementy také obsahuje meta-data o formuláři jako kdo je vlastníkem formuláře či jak se formulář jmenuje.

Správa dat formulářem je mechanismus zahrnující vytváření datových záznamů, které mají strukturu definovanou daným formulářem. Dále se pod tímto pojmem schovává operace čtení (viz. formulářový záznam).

Datový typ Plain text

Jeden z nejuniverzálnějších záznamových mechanismů/formátů, které se v IT vyskytuje je text, nebo také plain text. Text se v praxi používá především pro záznam dat, která mají být snadno čitelná pouhým lidským okem. Pro záznam neformátovaného textu poslouží jednoduché textové pole.

Na první pohled se může zdát, že umožnit uživatelům vkládat textová data je jednoduchá a triviální záležitost. Byla by to pravda v případě, kdy jsme schopni s dostatečnou přesností určit charakter textu, který bude pomocí textových polí vkládaný. Jelikož se snaží POUD poskytnout maximální flexibilitu co se týče definování datových struktur, není lehké definovat pravidla omezující vkládaný text.

Zásadní parametry, které musíme u textu řešit je jeho délka a znaková sada, ve které je text reprezentován. Jako dobré řešení s širokou podporou různorodých znaků a abeced využiji znakovou sadu UTF-8 [5] . Mezi výhody UTF-8 kódování patří především jeho rozšíření a objem znaků, které podporuje. UTF-8 má také příjemnou vlastnost v oboustranné kompatibilitě vůči ASCII znakové sadě (UTF-8 využívá pro uložení znaků 1 až 6 Bytů, přičemž ASCII znaky jsou uložené pomocí 1 Bytu)

S délkou textu je to více problematické. Protože pro ukládání textu bude využita databáze, musíme definovat omezení, které bude muset vkládaný text respektovat. Nabízí se nám 2

možnosti. První možností je definovat délku textu explicitně a omezit tak uživatelům možnosti vkládání textu v podobě maximální možné délky. Výhoda tohoto řešení je zpravidla rychlost čtení a zápisu a nezávislost na použité znakové sadě (pokud definujeme maximální délku textu na X znaků, pak bez ohledu na to, kolik bytů je použito na kódování textu, nám bude na databázové úrovni zaručeno, že budeme vždy schopni vložit text délky X). Druhou možností je použít datový formát s variabilní délkou, v žargonu databázového světa jej obvykle značíme jako CLOB – Character Large Object. Tato datová struktura se obvykle ukládá na separátním místě, mimo tabulku, která se na CLOB referencuje. Výhodou tohoto řešení je velké množství dat, které může CLOB pojmout, zpravidla 2 GB až 4 GB. Další výhodou je dynamická alokace paměti. Databáze alokuje pouze tolik místa, kolik skutečně vkládaná data zabírají. Nevýhodou tohoto řešení je zpravidla pomalejší operace čtení a zápisu a to až o jeden řád. Za další podstatnou nevýhodu považuji neschopnost vykonávat určité operace nad daty typu CLOB (množina operací, které nelze provádět, nebo lze, ale v omezené míře, se liší podle použitého DBMS – Database Management System). Nejčastěji jsou omezené funkce vyhledávání (např. klausule LIKE v SQL [6] jazyku), či formátovací funkce typu SUBSTRING.

Jako řešení jsem zvolil explicitně stanovenou délku 2048 znaků pro všechny texty bez výjimky a to z následujících důvodů:

- CLOB je pomalý a omezuje některé funkce, které budou pravděpodobně v budoucnu vyžadované, především fulltextové vyhledávání.
- POUD umožní vkládat data automatizovaně, a proto je kladen důraz na výkonnost IO operací.

Nevýhodou tohoto řešení je, že nelze uložit data delší než 2048 znaků a také plýtváním místa v případě vkládání dat řádově menších. Pro základní implementaci POUD a pro prvotní vyzkoušení celého konceptu aplikace to však postačí.

Datový typ pro vyjádření času

Data jsou velmi často používaná forma informace. Oproti plain textu popsaného v předchozí části aplikace zde nemusíme řešit problémy spojené se znakovou sadou či délkou záznamu. Datum lze jednoduše vyjádřit jedním číslem. V případě programovacího jazyka je datum reprezentovaný číslem typu Long (64bitů), kde toto číslo znamená počet milisekund od 1. ledna 1970. V podstatě jediné, co u data řešíme, je časová zóna a formátování. Protože POUD není aplikací, která by se zabývala formátováním textu, editací či jinou formou úpravy, budu využívat jednotné formátování pro všechny data. Co se týče časové zóny, ta pro POUD také nepředstavuje problém, protože data zde nefigurují jako součásti jakýchkoliv výpočtů či se nepodílí na synchronizaci systémů, jejichž činnost je na správně vyjádřeném datu závislé. Data v POUD aplikaci mají informativní charakter, který dává absolutní smysl pouze subjektu, který datum do systému vložil.

4.2.2 Atomické binární výrazy

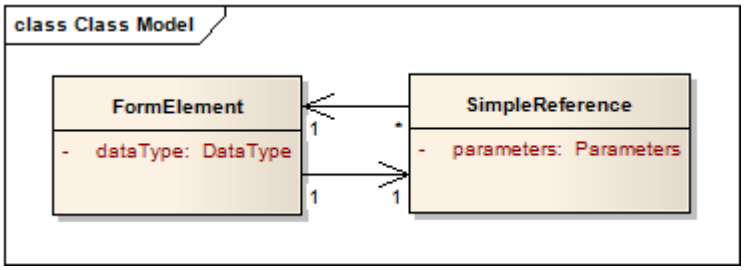
Již od počátku grafických uživatelských rozhraní bylo možné využívat tzv. checkboxy, tedy prvky, které umožnily zachytit jednoduchý logický výraz, true/false, 0/1, ano/ne. Síla tohoto datového typu spočívá v jeho jednoduchosti. Rozděluje svět na dvě poloviny, na „černou a bílou“. Bez možnosti vybírat mezi ANO a NE je praktiky nemožné vytvářet základní rozhodovací logiku a v budoucnu by to přineslo nemožnost vytvářet jednoduchá pravidla, která

se budou chovat podle toho, zda jste zadali 1 nebo 0. Z tohoto důvodu nesmí binární atomické datové typy chybět mezi standardní sadou formulářových polí.

Toto byl základní výčet datových typů, které bude prvotní verze POUD podporovat.

4.3 Reference

Vazby neboli reference je mechanismus umožňující definovat závislosti mezi jednotlivými základními formulářovými elementy.



Obrázek 4: Jednoduchý model znázorňující vztah mezi odkazujícím se objektem, referencí a odkazovaným objektem

Obrázek 4 znázorňuje model referencí. FormElement představuje jedno konkrétní pole formuláře daného datového typu (formulářový element). SimpleReference objekt pak drží informace o parametrech reference. POUD umožní vytvářet reference v následujícím smyslu:

- Každý element může definovat nejvýše jednu referenci.
- Element se může odkazovat sám na sebe.
- Odkazovaný element musí být stejného datového typu jako odkazující se element.

Z bodů 1 – 3 vyplívá, že na jeden element se může odkazovat libovolné množství elementů. Bod 1 říká že, reference je nepovinná. Odkazování elementu na sebe sama, definované v bodě 2., může na první pohled vzbuzovat obavy z potenciálních problémů, jakými by mohlo být zacyklení, a to ať už na úrovni logického návrhu, nebo na úrovni databáze. Tyto pochybnosti budou vyvráceny v kapitole zabývající se implementací. Účelem bodu 3 je zajistit typovou bezpečnost napříč všemi formuláři.

Aplikace POUD bude ve svém základu podporovat pouze jednoduché relace, které v případě definování dovolí uživateli vkládat data pouze z množiny dat definované odkazovaným elementem.

4.4 Kompozitní datové typy

V POUD aplikace je také možné definovat datové typy, které umožní vytvářet struktury podobné obecným tabulkám. Takovéto objekty definují množinu základních datových typů, s daným pořadím a názvem, které v analogii s tabulkami představují sloupce tabulky. Na úrovni formuláře se budou kompozitní datové typy chovat jako ostatní základní datové typy (plain text, datum, atomická binární hodnota) s tím rozdílem, že není možné definovat reference na úrovni kompozitních datových typů (příchozí ani odchozí). Je však možné vytvářet reference

z „vnitřku“ kompozitního elementu na úrovni jeho potomků (sloupce - analogie s tabulkami) na základní datové typy (na úrovni formulářů).

Bez kompozitních datových typů by bylo možné vytvářet záznamy pouze jako množinu dat s pevně danými hranicemi, co se týče datové struktury formulářů. Tedy například záznamy, které obsahují vždy jedno textové pole a jeden datum. Díky kompozitním typům sice nevytvoříme variabilitu, co se týče datové struktury, ale umožníme vkládat různé množství dat v rámci jednoho formulářového záznamu.

4.5 Datová reprezentace formulářů

POUD využije databázi pro ukládání jak formulářových meta-dat, tak dat spravovaných formuláři. Hlavním důvodem je možnost jednoduše a dynamicky za chodu aplikace měnit jak strukturu formulářů, tak data samotná.

4.5.1 Reprezentace formulářů

Jednou z důležitých otázek je, jakým způsobem budou reprezentována data formulářů. Rozlišujeme zde dvě základní množiny dat, které se k formulářům jako takovým vztahují. Jednak meta-data formulářů a pak data, která formuláře spravují. Meta-daty rozumíme data, která definují formuláře samotné, tedy kdo formulář vlastní, z jakých prvku se skládá atd. Daty spravovanými formulářem se rozumí data, která se do systému pomocí formuláře vkládají a která se pomocí formulářů čtou. Tyto dvě skupiny mají své vlastnosti, pro které se hodí různé zacházení.

4.5.2 Meta-data formulářů

Meta-data jsou data o datech. V asociaci s formuláři mluvíme o datech, která definují základní nastavení formuláře, tedy vlastníka formuláře a jméno formuláře a k těmto základním datům také patří data definující formulářové elementy, jejich reference, jejich datové typy, popisy, jména, datové typy či pořadí v jakém se ve formuláři vyskytují vůči ostatním elementům. Co se týče datové náročnosti v porovnání s daty, které pomocí formulářů spravujeme, pak se jedná o zanedbatelné množství. Meta-data formulářů se vyskytují v aplikaci pouze jednou pro každý formulář, zatímco data s kterými formulář pracuje (čtení, ukládání atd.) se dynamicky mění, jejich množství roste či se zmenšuje a velmi pravděpodobně přesáhne mnohonásobek velikosti meta-dat.

Protože se meta-data formuláře skládají z jasně definovaných částí, můžeme tyto meta-data reprezentovat na úrovni databáze jasně definovaným schématem a na úrovni aplikace jasně definovanou sadou DAO objektů (Data Access Object). Není třeba řešit dynamické aspekty jakéhokoliv druhu. To ovšem neplatí o datech, které formuláře spravují.

4.5.3 Data spravovaná formuláři

POUD předem neví nic o struktuře dat, které budou budoucí formuláře spravovat. Jediné s čím POUD počítá, jsou datové typy, které je možné na úrovni formuláře definovat. Uživatel tvořící formuláře může definovat libovolně komplexní strukturu formuláře. Nabízí se nám zde dvě možnosti, jakým způsobem můžeme tyto datové struktury reprezentovat, a to ať už na úrovni aplikace, tak na úrovni databáze.

Statické datové schéma

Tato technika spočívá ve vytvoření meta-modelu obecného formuláře, respektive jeho obsahu. Následně je tento meta-model vyjádřen pomocí tabulek v databázi a vazbami mezi nimi. Korespondující DAO objekty jsou samozřejmostí. Vytvoření meta-modelu může být poměrně jednoduchý úkol, avšak jeho reprezentace na úrovni databáze může být velice složitá. Obecně spočívají výhody tohoto řešení v tom, že máme robustní prostředí pro práci s daty a neměnnou logiku, která se za ním skrývá. Další velkou výhodou je dobrá testovatelnost. Pokud máme statickou strukturu (neměnnou), můžeme napsat testy šité na míru. Nevýhodou tohoto řešení může být exploze tabulek v databázi společně s množstvím kódu, který nad touto strukturou operuje. Navíc pokud máme příliš složité databázové schéma, můžeme pociťovat razantní pokles výkonu při základních CRUD operacích.

Dynamické datové schéma

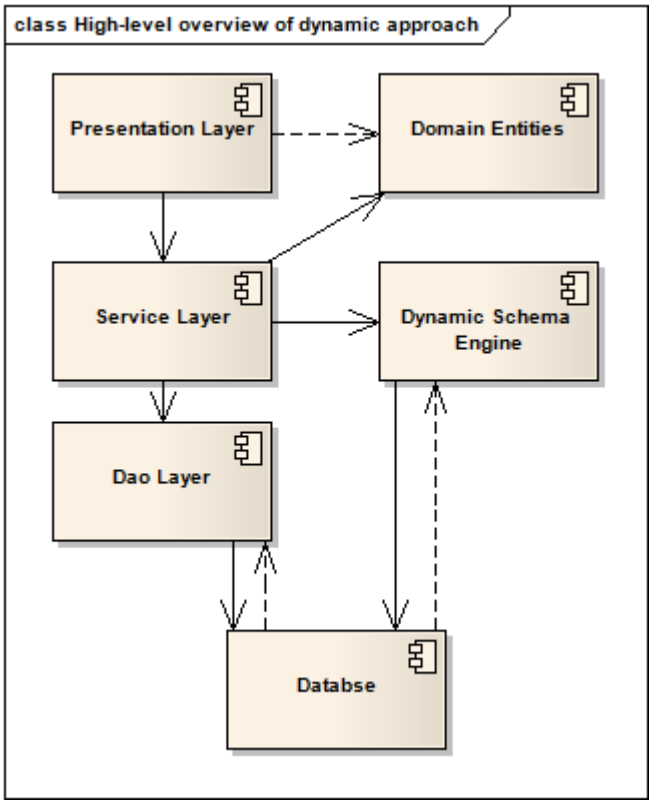
Nejedná se o příliš obvyklý přístup, nicméně některé systémy ho využívají. Například platforma ServiceNow [7] , která dovoluje definovat vlastní datové struktury a na základě nich dynamicky mění obsah databáze (schématu). Tato technika vyžaduje zavedení enginu, který se bude starat o dynamičnost tohoto řešení. Obecně tedy bude obstarávat vytváření datových struktur na úrovni databáze a jejich provázání s aplikační logikou na úrovni aplikace. Obrovskou výhodou tohoto řešení je jeho univerzálnost. Jsme vázáni pouze schopnostmi daného enginu. Nemusíme předem vědět jak objekty, které budeme do databáze ukládat, budou vypadat. Naproti tomu velkou nevýhodou je komplexnost takovéhoto řešení. Jednak musíme v první řadě vytvořit zastřešující engine a pak musíme zbytek systému implementovat natolik obecným způsobem, že bude schopen s předem nedefinovatelnými daty pracovat. Bohužel zde trpí velkou měrou i testovatelnost, protože na rozdíl od statického schématu nemůžeme testovat přímo datovou vrstvu a aplikační logiku, protože ta bude tvořena dynamicky za chodu aplikace. Můžeme tedy testovat pouze to, zda bude daná datová a aplikační vrstva tvořena správně.

V kapitole o požadavcích jsem se zmínil, že během realizace této práce došlo několikrát ke změně funkcionality a implementace. Jedna z nejzásadnějších změn proběhla právě ve způsobu reprezentace formulářů. Z počátku jsem se vydal cestou statického řešení, kde jsem ale narazil na několik problémů, které mi lidově řečeno přerostly přes hlavu. Narazil jsem na dva problémy, přičemž jsem se musel smířit vždy s jedním, nebo druhým. Prvním problémem byla komplexita databázového schématu a počet operací, které bylo potřeba pro základní CRUID operace. Druhým problémem byly datové typy na úrovni aplikace a jejich reprezentace v databázi.

První problém spočíval v tom, že jsem měl definovaný meta-model, který mimo jiné definoval formulářový element jako objekt, který si kromě dalších informací drží také informaci o datovém typu. Pro zajištění datové konzistence mezi aplikací a databází jsem vytvořil po jedné tabulce pro každý podporovaný datový typ. To vyústilo v několik tabulek jen pro ukládání samotných dat. Protože každý formulář se může skládat z libovolného počtu elementů, nebylo možné vytvořit jednu tabulku pro data. Řešením byla jedna tabulka představující samotný formulář a pak několik tabulek, jedna pro každý datový typ, které měli referenci na daný záznam v tabulce pro formuláře. Při CRUID operacích se pak provádělo několik JOIN operací najednou. Druhým problémem bylo, jak rozumně rozlišit základní datové typy od kompozitních a zároveň se snažit držet principu DRY. Na aplikační úrovni to není složitá záležitost a pomocí

dědičnosti a polymorfismu toho docílíme poměrně snadno. Nicméně to celé pak převést na databázovou úroveň se ukázalo jako velmi složitá záležitost.

Rozhodnul jsem se tedy, že se vydám jinou odlišnou cestou a to cestou dynamického schématu. Na následujícím obrázku je zobrazena jedna z možností, jakým může být aplikace postavena při použití dynamického přístupu.

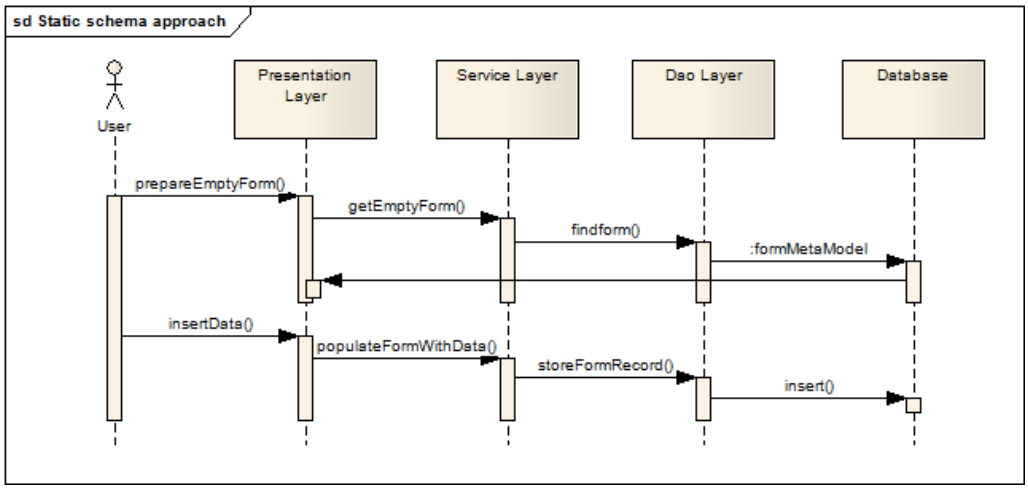


Obrázek 5: Takto může vypadat jedno z řešení přístupu dynamického schématu na úrovni aplikace

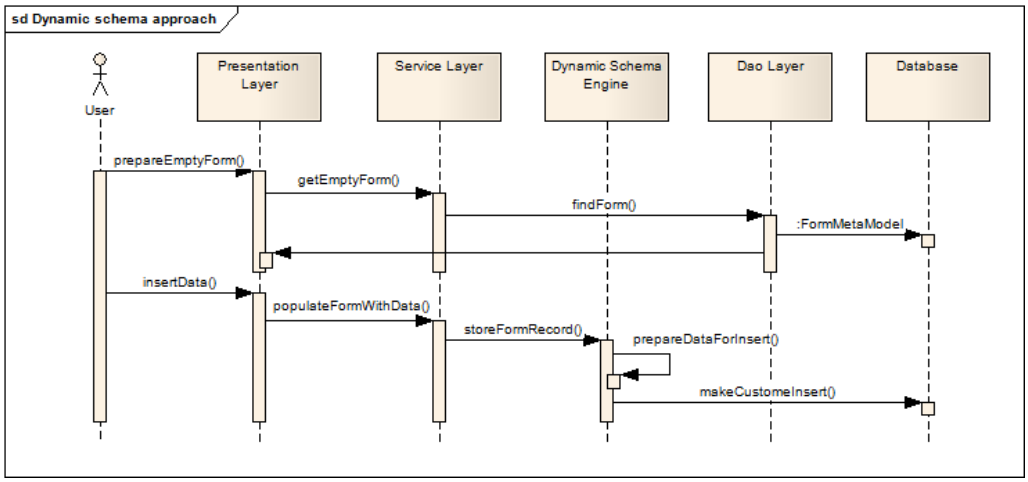
Jak lze vidět na obrázku č. 5 aplikace využívá 2 druhy přístupu k databázi. Dao Layer využívá principů ORM (Object-Relation mapping). K tomu může sloužit jeden z mnoha ORM frameworků (EclipseLink, Hibernate atd.), které za nás řeší „překlad“ dat do jednotlivých tabulek. Aby toto celé fungovalo, je třeba jasně a předem definovat mapování z doménových objektů na databázové tabulky.

DSE (Dynamic Schema Engine) přistupuje k databázi přímo. Na základě dat, která mu jsou předány, vytvoří odpovídající příkazy, které data uloží odpovídajícím způsobem do databáze. Stejně tak se chová i v případě čtení. Na základě vstupních parametrů vytvoří dotazy, kterými získá data z databáze zpět. Můžeme říct, že se jedná o alternativu k Dao vrstvě s tím rozdílem, že Dao vrstvě předáváme doménové objekty v případě, kdy chceme data ukládat a identifikátory (asociované s primárními klíči v korespondujících tabulkách) či další parametry v případě kdy chceme data číst. Naproti tomu DSE předáváme „syrová“ data, která můžeme v případě potřeby zabalit do tzv. Data Transfer object (DTO, využívá se především v situacích, kdy pracujeme s komplexními datovými strukturami, pro které nám nestačí např. pole či kolekce). DSE tyto data přetransformuje do odpovídajících příkazů, které poté nad databází vykoná. V případě čtení se DSE chová velmi podobně jako Dao vrstva.

Následující sekvenční diagramy demonstrují ukázkovým způsobem rozdíl mezi dynamickým a statickým schématem co se ukládání dat týče (čtení je v obou případech velmi podobné).



Obrázek 6: Sekvenční diagram znázorňující zápis dat pomocí statického schématu



Obrázek 7: Sekvenční diagram znázorňující zápis dat pomocí dynamického schématu

Hlavním rozdílem obou přístupů je využití rozdílných vrstev v případě zápisu dat. Statický model používá pouze Dao vrstvu. Naproti tomu dynamický model používá Dao vrstvu pro získání dat o formulářích, tedy jejich meta-modely a DSE pro ukládání formulářových záznamů.

4.6 Adaptéry

Ruční vkládání dat pomocí formulářů není zpravidla vhodné řešení pro zpracování velkého množství dat. Nemluvě o tom, že uživatelé nemůžou být stále přístupní, aby data mohli vkládat. Řešení pro tyto „neduhy“ nabízí právě tzv. adaptéry.

Adaptér je ve své podstatě bránou mezi datovými zdroji a aplikací. Základní vlastností adaptéru je schopnost z určitého datového zdroje data číst a transformovat je na existující datové struktury v aplikaci, tedy na struktury definované formuláři. Transformace ze své podstaty nemusí být bezztrátová. Jednak se může stát, že formulář, na jehož strukturu jsou data transformována, nenabízí dostatečně bohaté prostředí pro zpracování všech příchozích dat,

anebo naopak záměrně nechceme všechna data z datového zdroje zpracovávat, protože pro nás nepředstavují žádnou významnou hodnotu.

Adaptér se tedy skládá z následujících částí:

- Množina parametrů, které jsou potřeba ke správné funkčnosti adaptéru.
- Funkce, která čte data z datového zdroje.
- Množina výstupních objektů, na které jsou čtená data transofmována.

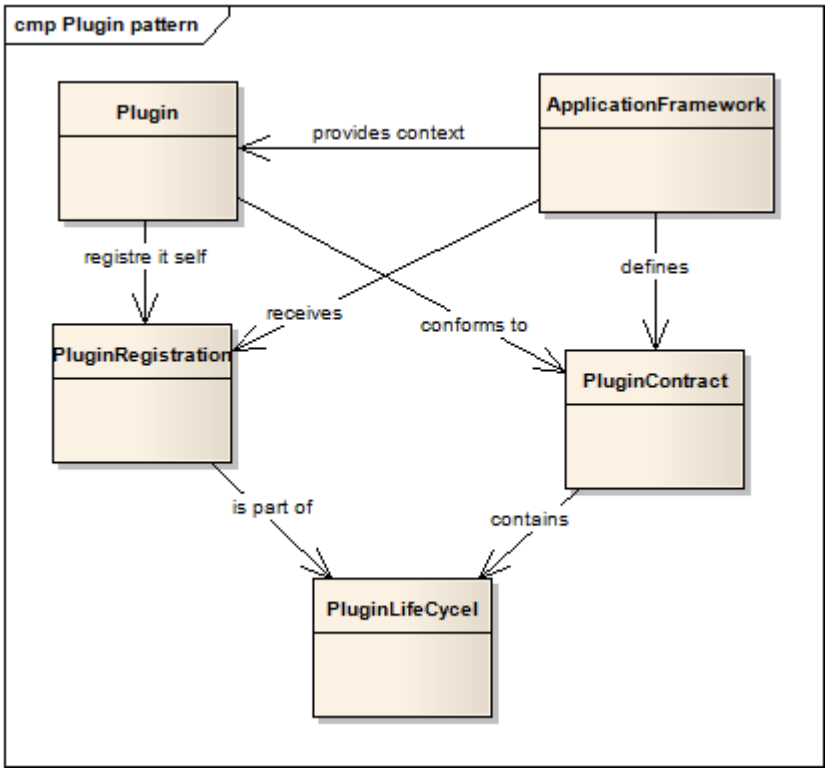
Jako příklad vstupního parametru adaptéru si můžeme představit URL datového zdroje, z kterého adapter může data číst. Některé parametry můžou být nepovinné, jiné jsou povinné.

Funkce pro čtení dat je srdcem adaptéru. Jedná se o explicitně definovaný algoritmus/kód, který dokáže na základě vstupních parametrů adaptéru, číst data z datového zdroje. Různé druhy adaptérů se liší především v použitém algoritmu. Je velmi obtížné obecně popsat algoritmus obecného adaptéru na takové úrovni, aby bylo možné tyto algoritmy definovat na vyšší úrovni abstrakce než na úrovni programovacího jazyka. Z tohoto důvodu se touto myšlenkou nebudu dále zabývat.

Je jasné, že čím více adaptérů bude aplikace podporovat, tím více způsobů pro získávání dat bude přístupno. Pokud tedy budou adaptéry tvořený psaním zdrojového kódu, pak nastává otázka, jakým způsobem se budou nové adaptéry do aplikace začleňovat? Je několik standardních postupů jak tuto nelehkou problematiku řešit. V následujících odstavcích popíši některé z nich společně s výhodami a nevýhodami.

4.6.1 Adaptér jako plugin

Filosofii skrývající se za pluginy zde nebudu do detailů rozebírat. K tomuto účelu může posloužit článek [8] . Následující obrázek ukazuje jednoduchý modelový příklad návrhu aplikace podporující paginy.



Obrázek 8: Jednoduchý návrhový vzor pluginů

Základní myšlenkou je umožnit rozšíření stávajících funkcí aplikace za běhu (tzv. run time). Celé to spočívá ve dvou základních komponentách a to Plugin (viz. Obrázek 8) a APllicationFramewrok. Plugin obsahuje novou přidanou funkcionalitu zatímco ApplicationFramework poskytuje prostředí pro vykonávání této funkcionality.

Výhodou tohoto řešení je již zmíněná rozšiřitelnost. Nevýhodou je pak nutnost řešit věci ohledně životního cyklu pluginu, jeho dynamické odebírání a přidávání do aplikačního kontextu a v neposlední řadě také nutnost dodržet jasně definované rozhraní poskytnuté Application Frameworkem.

4.6.2 Adaptér jako zdrojový kód

Jedná se v podstatě o součást zdrojového kódu aplikace. Aplikace již ve svém základu definuje sadu takových to adaptéru a zpřístupňuje zpravidla neprogramové rozhraní pro interakci. Často jsou takovéto adaptéry psané přímo na míru aplikace a jsou vyvíjené samotnými autory aplikace.

Výhodou je bezesporu prakticky neomezený přístup k jednotlivým částím aplikace (nemusí zde existovat ohraničující API). Mezi další přednosti tohoto přístupu patří jejich distribuce. Adaptéry jsou již součástí systému, a proto není třeba, aby se uživatel staral o jejich zprovoznění či konfiguraci. Z toho vyplívá i větší míra kontroly nad funkcionalitou, kterou do systému adaptéry přináší. Stinnou stránkou je nemožnost měnit sadu adaptérů. Pro přidání či ubrání adaptéru je potřeba zasáhnout do zdrojového kódu aplikace, vydat novou verzi a distribuovat je mezi uživatele (což nemusí být vždy problém, například v případě webové aplikace stačí nasadit novou verzi aplikace a restartovat server, uživatelé si přitom nemusí ničeho ani všimnout).

V aplikaci POUD jsem si zvolil druhou zmíněnou možnost a to z následujících důvodů:

- Chci se vyhnout složitému managementu pluginů, jejich životního cyklu či registračního managementu.
- Adaptéry jsou přístupné pro všechny uživatele, nejsou na ně nijak asociovány, a proto není potřeba, aby byly adaptéry přístupné jako samostatné komponenty, které by si uživatelé mohli sami do systému přidávat.
- Chci mít kontrolu nad funkcí, kterou adaptéry poskytují a díky začlenění adaptérů do zdrojového kódu, mají přístup ke zbytku aplikace.
- Protože je POUD webová aplikace, je snadné přidat nový adaptér a odstínit uživatele o jejich aktivaci/nastavení, což by museli řešit v případě pluginů.

Samozřejmě je možné tento přístup více sofistikovat a vytvořit tak více přívětivé prostředí pro vývojáře. Toto je i cílem návrhu, který bude do větších detailů popsán v kapitole věnující se návrhu a implementaci.

4.7 Časovače adaptérů

Adaptéry musí někdo řídit. Protože účel adaptérů je nahradit potřebu lidské činnosti při vkládání dat do aplikace, je třeba zajistit mechanismus, který bude adaptéry za samotné uživatele spouštět. K tomu slouží časovače. Každý uživatel si bude moci vytvořit své vlastní instance daných adaptérů a ke každé instanci si definuje i časovač. Instance adaptérů jsou přístupné pouze jeho vlastníkovi, stejně tak i časovače.

Existuje plno přístupů, jak plánovat spouštění adaptérů. Můžeme například chtít, aby se adaptér spustil pouze v určitou hodinu, nebo naopak chceme aby, se periodicky spouštěl každých 10 minut.

Existuje standard, který umožňuje definovat obě tyto varianty a dokonce ještě více. Mluvím zde o Cronu [9] . Unixové systémy tento mechanismus využívají velmi často. Poud Cronu využije, i když nenabídne všechny možné variace časování, které Cron umožňuje. Jednak se neočekává, že by průměrný uživatel potřeboval vytvářet komplexní časování a jednak by se nastavení časovače mohlo stát velmi zmatečné až nezvládnutelné pro průměrně zdatného uživatele.

Jak tedy Cron funguje a které jeho možnosti POUD využije? Definice časování pomocí Cronu se skládá z 6 částí:

- Minuta (0-59)
- Hodina (0-23)
- Den v měsíci (1-31)
- Měsíc (1-12)
- Den v týdnu (1-7, 1 = pondělí)
- Rok (1900 - 3000)

Každá z těchto hodnot je zadaná bud celým číslem, znakem *, výčtem hodnot, rozsahem hodnot nebo inkrementačním přírůstkem (Cron ve skutečnosti podporuje ještě další komplexnější mechanismy, ty ale nejsou pro nás podstatné). Pro účely adaptérů budou využity následující možnosti Cronu:

- Definování konkrétního času spuštění, tedy například, v 8:00, bude se periodicky opakovat každý den.
- Interval v jakém se má spouštění adaptérů uskutečnit, tedy například každých 10 minut.

Více z možností, které Cron nabízí v POUD prozatím nevyužiji. Jednak cílem není poskytnout bohaté rozhraní pro časování, a pak v momentě, kdy bude aplikace umět s Cronem pracovat a bude architektonicky připravena na konfigurování Cronu z uživatelského rozhraní, pak už bude podstatně jednodušší přidávat další a komplexnější možnosti časování.

4.8 POUD API

V této kapitole se zabývám API (Application program interface) POUD aplikace. API je sada rutin a nástrojů pro specifikování způsobu, jakým spolu softwarové komponenty můžou interagovat. Existuje mnoho typů API, pro operační systémy, aplikace či webové služby. Pro potřeby POPD aplikace nás zajímá především API v podobě webové služby, či v podobě REST rozhraní. Rozhodnutí, který typ má aplikace implementovat, by mělo být založené na racionelním očekávání a požadavcích. Zkusím je zde rozebrat spolu s popisem obou typů přípustných rozhraní.

4.8.1 Požadavky na API

Na vysoké úrovni abstrakce poskytuje API vstupní bod do aplikace a dovoluje provádět různé operace nad daty aplikace. Kromě definice sady podporovaných operací, je potřeba se také zabývat otázkou bezpečnosti a schopnosti konzumentů implementovat klienty, které budou schopné poskytnuté operace volat. V následujících odstavcích popíši klíčové aspekty, které by každé API mělo do určité míry řešit.

Podpora standardů podle RFC, ISO, ANSI a dalších

Standardy slouží k jednotnému přístupu při řešení nějakého problému. Díky standardům si jednotlivé organizace nemusí předávat detaily ohledně implementace jejich API, postačí pouze určitá forma dokumentace jejich rozhraní, která popíše pouze veřejnou část a to pomocí žargonu daných standardů. Využitím standardů přináší i další výhody. Mezi ně patří i to, že jsou standardy volně přístupné a bezplatně využitelné. Navíc pro standardy běžně používané na webu (HTML, http, JSON atd.) existuje velmi široká komunita a podpora.

Principy ochrany dat

Pokud pracujeme s daty, které jsou z nějakého důvodu citlivá, musíme je chránit před zneužitím. K tomu slouží celá řada technik a technologií. Je velmi důležité zvolit vhodný způsob ochrany dat, který bude poskytovat dostatečně robustní ochranu a zároveň přijatelnou výpočetní zátěž.

Ochrana dat je široký pojem, a proto se zde vymezím pouze na šifrovaný přenos dat. Rozlišujeme různé úrovně zabezpečení komunikace. Data lze šifrovat již na úrovni databáze, kde máme možnosti šifrovat data i na úrovni jednotlivých sloupců tabulek. Dnešní moderní databáze (např. Oracle) podporují tento druh šifrování. Navíc poskytují ovladače připojení (connection driver), které za pomocí zpravidla symetrických šifer přenáší data mezi aplikací a databází v šifrované podobě. Veškeré výpočty spojené s šifrováním se provádí na pozadí a

aplikace ani uživatelé se nemusejí explicitně o nic starat (kromě počáteční konfigurace). Data v rámci aplikace zpravidla nešifrujeme, vše je uložené v operační paměti hostujícího počítače a dnešní operační systémy neumožňují jiným aplikacím a procesům číst z paměťového prostoru, kterým jim nebyl přidělen. Existují však výjimky, kdy lze přečíst z paměti data, které danému procesu nepatří. Zpravidla se jedná o chyby v systémech, které se dříve či později odstraní a tak se i v budoucnu zamezí neoprávněnému zcizení dat. Při psaní této práce se objevila bezpečnostní chyba v protokolu OpenSSL, kdy za určitých okolností, bylo možné číst mimo paměťový prostor procesu, který dělal kontrolní součty paketů, které OpenSSL protokol používá. Údajně tak šlo získat privátní klíče k certifikátům, kterých daný protokol využíval.

Jinak je tomu u interakce mezi jednotlivými systémy a aplikacemi. Obzvlášť u webové služby či aplikace, které nějakým způsobem vyžadují autentizaci uživatele. Data, která uživatelé posílají, nejčastěji uživatelské heslo a jméno, je potřeba před odesláním z klientského počítače zašifrovat, aby je po cestě nebylo možné odposlechnout.

Webové aplikace pro tento účel používají HTTPS protokol. Technicky to ale není přesně řečeno, protože šifrování se provádí nikoliv na úrovni aplikace ale na úrovni kontejneru (serveru), který aplikaci hostuje. Aplikace tak ani nemusí vědět, že k nějakému šifrování dochází.

Https protokol využívá asymetrického šifrování, nejčastěji pomocí protokolu SSL. SSL operuje v modelu ISO/OSI mezi vrstvou transportní (např. TCP/IP) a vrstvou aplikační (např. http protokol) a poskytuje jak šifrování komunikace, tak podporu autentizace.

Nezávislost na implementaci

Pod pojmem platform independent si představíme koncept, který řeší nějaký problém bez ohledu na implementaci v konkrétní technologii. Vždy se zde operuje s určitou mírou abstrakce, díky níž se dokážeme zprostit závislostí na jiných technologiích. API, které využívá standardů platformě nezávislých, tak otevře cestu pro komunikaci všem, kdo implementují daný standard. Mezi široce známé, platformě nezávislé a velmi hojně využívané standardy pro práci s daty bezesporu patří například, XML formát pro strukturovaná, člověkem i strojem čitelná data, http protokol pro přeno hypertextového obsahu napříč internetem, BMP obrázkový formát pro bezeztrátovou reprezentaci obrázků, JSON formát pro reprezentování objektově strukturovaných dat, WSDL jazyk pro definování webových služeb a mnoho dalších.

Nezávislost na přenášených datech

Do určité míry jsme vždy závislí na přenášených datech. Omezujícími faktory může být operační paměť, šíře komunikačního pásma či výpočetní výkon. Přenášená data by měla být reprezentovaná v takové podobě, abychom je byli schopni bez problému a zbytečný komplikací přenést. V podstatě rozlišujeme dva typy dat. A to data textová a binární. V případě textových dat musíme definovat správné kódování, abychom byli schopni s daty pracovat. Binární data jsou z tohoto pohledu více problematická, protože pro jejich reprezentování nestačí vždy jen použít správné dekódování. Mnoho formátů obsahuje navíc k samotným datům i meta-data, která slouží pro identifikaci obsahu nebo i pro zvolení správného nástroje pro zpracování těchto dat. API by se těmito problémy nemělo zabývat a tato zodpovědnost (tedy jak data reprezentovat, jak je číst atd.) by měla být přenesená na příjemce a odesílatele.

Kapitola 5

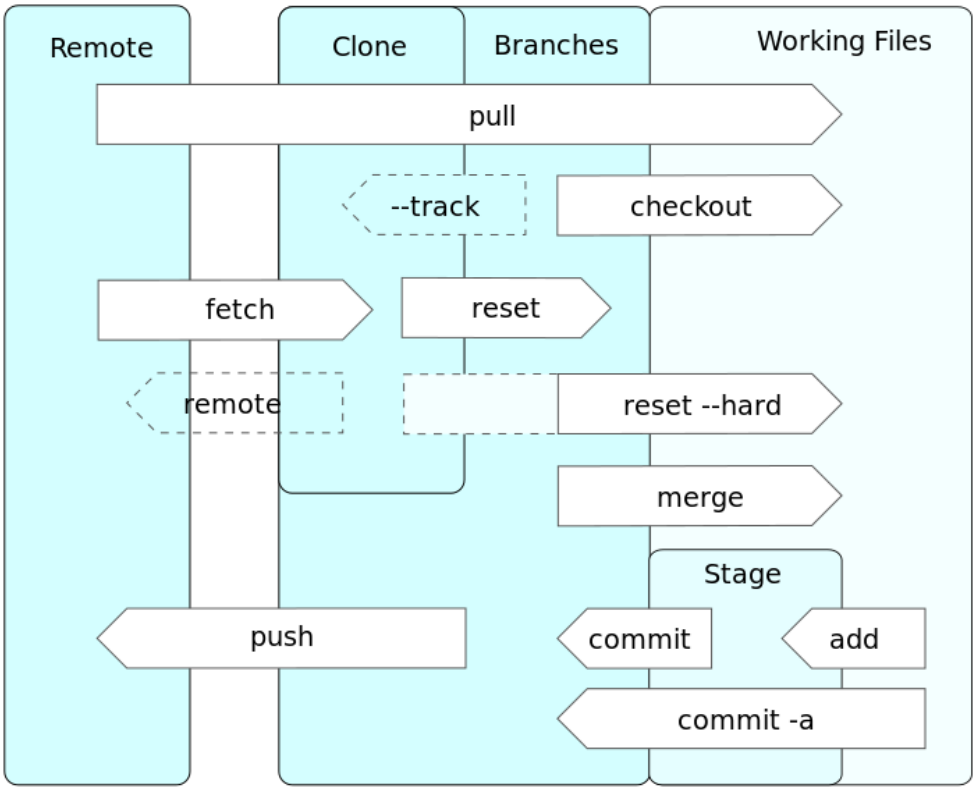
Infrastruktura projektu

Vývoj projektu je komplexní souhrn aktivit, které vyžadují řádnou organizaci. Je nanejvýše nutné zajistit správnou posloupnost úkonů a předávání informací mezi jednotlivými subjekty, které se na vývoji podílí. V následujících odstavcích popíši, jak vypadá infrastruktura projektu POUD a jakým způsobem ovlivňovala či určovala vývoj POUD.

5.1 Správa zdrojových kódů – SCM

V minulosti jsem se setkal s projekty, které sice spravovaly kód v určitém smyslu SCM, ale ve velmi omezené míře. Například platforma WebMethods [11] , jenž poskytuje prostředí typů SAAS (Software as a service) pro integraci nejrůznějších systémů za použití široké škály technik a protokolů. SCM v podání WebMethods existuje pouze na úrovni zamykání jednotlivých balíčků, které obsahují zdrojové kódy. Vývojář si tedy může pro své účely zamknout daný balíček a zamezit ostatním vývojářům zdrojové kódy v tomto balíčku měnit. WebMethods mají svojí vlastní strukturu projektů a je nanejvýše nevhodné na ně využít některý ze standardních SCM nástrojů (SVN, GIT). Z absence řízení revizí kódu bylo prakticky nemožné vyvíjet ve více paralelních větví najednou, či dělat MERGE operace dvou verzí zdrojového kódu. Zde si člověk uvědomí opravdovou přidanou hodnotu SCM nástrojů.

POUD aplikace využívá nástroje GIT [11] . GIT je open-source nástroj, původně zamýšlený pro správu zdrojového kódu unixových systémů. Jedná se o tzv. decentralizovaný SCM nástroj. To znamená, že každý vývojář má svůj lokální repozitář, kde si řídí vlastní verzi zdrojového kódu. V případě, kdy si je vývojář jist, že dokončil práci na jedné části aplikace, může kód nahrát do centrálního repozitáře. Tím se stane kód přístupný všem ostatním vývojářům. Samozřejmostí GIT nástroje je kromě jiného také vytváření větví pro separátní vývoj funkcí aplikace, tagování, které slouží k označení určité fáze vývoje štítkem či MERG funkce, kdy se změny v jedné větvi spojí s verzí ve větvi jiné.



Obrázek 9: Přehled základních operací SCM nástroje GIT

Protože jsem byl jediný vývojář, který měl práva na zápis do repositáře, nebylo zapotřebí využívat operací PULL, CHECKOUT, FETCH či MERGE. Naproti tomu operace ADD, COMMIT a PUSH byly využívané frekventovaně. I přesto že se na vývoji podílel jediný vývojář a i přesto, že jsem se nedostal do situací, kdy bych využil velmi užitečnou funkci MERGE, přináší GIT a obecně SCM nástroje výhody spojené s revizemi a možností se kdykoliv vrátit k předchozímu stavu zdrojového kódu.

5.2 Struktura projektu

Podle charakteru vyvíjeného projektu se odvíjí i organizace jeho částí. POUD projekt má charakteristiku webové aplikace, která je vyvíjená na jediné platformě a má závislosti na různorodé frameworky, knihovny a technologie. Z podstaty věci vyplívá, že při vývoji se musí řešit následující náležitosti:

1. SCM – řízení zdrojových kódu, viz předchozí kapitola
2. Zpřístupnění závislostí v podobě knihoven, frameworků a technologií, a to jak ve vývojovém prostředí, tak v produkčním prostředí
3. Testování
4. Kompletace spustitelné aplikace
5. Tzv. visibility, tedy dostupnost informací potřebných a dostačujících pro řízení, správu a buildování aplikace a pro případné řešení problémů s tímto spojených

Všechny tyto vlastnosti pokrývá kombinace vývojového prostředí Eclipse IDE s verzovacím nástrojem GIT společně s buildovacím frameworkem Maven. To co určilo v případě POUD strukturu projektu, byly tyto faktory:

1. Vývojová platforma JAVA
2. Buildovací framework Maven

Dříve než popíši strukturu projektu POUD, vysvětlím, jak funguje buildovací framework Maven a jakým způsobem ovlivnil strukturu projektu POUD.

5.2.1 Maven

Maven [12] je nástroj pro správu a automatizaci buildů. Cílová platforma je Java, Maven však podporuje i ostatní jazyky. Základním principem Mavenu je popsání projektu pomocí tzv. Project Object Model (POM). POM slouží k popisu projektu z několika pohledů. Jednak z pohledu zdrojových kódu a jeho závislostí na externích knihovnách, frameworcích a jiných subjektech, a jednak z pohledu buildovacího procesu a funkcí s tím spojených (jako je spouštění testů, generování různých reportů, mezi kterými můžou být reporty zahrnující míru pokrytí zdrojového kódu testy).

Maven sám o sobě je navržen jako modulární platforma a funguje na principu volání jednotlivých pluginů. V podstatě základní funkce Mavenu je řízení životního cyklu buildování projektů a spouštění pluginů, které pak obstarávají samotné sestavení spustitelné aplikace.

Základní struktura Maven projektu je znázorněn v následujícím bloku:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

Pom.xml je množina konfigurací, definic závislostí a nastavení životního cyklu buildění aplikace.

Složka *src/main/java* obsahuje veškeré zdrojové kódy aplikace. Zdrojové kódy nejsou přímo v této složce, ale jsou umístěné tzv. balíčcích (package). Jmenná konvence balíčků má své opodstatnění a je důležité ji věnovat patřičnou pozornost. Jméno balíčku obsahuje zpravidla

čtyři hlavní části: typ organizace, jméno organizace, jméno aplikace a jména konkrétních balíčků aplikace. Typ organizace se dělí zpravidla na typy com (commerce – zaměření pro komerční účely) a org (organization, přesněji non-profit organization, tedy neziskové organizace). Jméno organizace pak definuje jméno, pod kterým je daný subjekt (vlastník aplikace) znám na trhu. Jméno aplikace definuje jméno v rámci organizace. Pojmenování konkrétních balíčků aplikace je silně závislé na vnitro-organizačních jmenných konvencích.

Složka *src/main/resources* obsahuje konfigurační soubory aplikace. Do této složky se dávají konfigurace především pro nastavení frameworků a knihoven, které aplikace využívá. Lze zde umístit také konfigurační soubory samotné aplikace. Toto se však v praxi příliš neděla. Místo toho se konfigurační soubory specifické pro aplikaci (např. konfigurace spojení s databází či URI systémů, s kterými aplikace nějakým způsobem interaguje) umisťují do složky, která se nevyskytuje na classpath [25] aplikace. Složky src/main/resources a src/main/java se po sestavení aplikace zabalí do některého standardního formátu (jar, war, ear atd.) a nejsou přístupné vně aplikace. Důvod proč není dobré konfigurace specifické pro aplikaci uložit na classpath je ten, že chceme mít možnost tyto konfigurace měnit bez nutnosti znovu sestavení aplikace a v ideálním případě bez nutnosti restartování aplikace či serveru, na kterém je aplikace nasazena. Je jednoduché číst externí konfigurační soubory, ale dle specifikace není přípustné měnit obsah zapouzdření aplikace (jar, war, ear atd.) za běhu aplikace, či bez jeho znovu sestavení.

Složka src/test/java analogicky ke složce src/main/java obsahuje zdrojové kódy testů. Patří sem automatické testy, tedy v případě javy JUnit testy. Dále se zde vyskytují testy pro testování uživatelských rozhraní (například Selenium testy).

Podobná analogie platí i v případě src/test/resources. Platí zde však jistá rozdílnost oproti složce src/main/resources a to ta, že jsou zde uložené veškeré konfigurační soubory, tedy konfigurace specifické pro aplikaci. Pro testování aplikace totiž nepotřebujeme mít tyto konfigurace přístupné z vnějšku. Stačí nám, když je máme přístupné v momentě spuštění testů. To zpravidla máme skrze vývojové prostředí, ve kterém testy spouštíme. Pokud využíváme metodik continuous integration, kdy za nás testy spouští nástroj, který se stará i o sestavování aplikace, pak jsou tyto konfigurace přístupné na classpath.

Projekt může obsahovat i další složky. Například složky obsahující certifikáty, či šablony e-mailů. Tyto složky nemají definovanou strukturu a je čistě na organizaci, jakým způsobem si je bude spravovat. Místo pro tyto složky je v kořenovém adresáři projektu, tety pod složkou my-apps.

Životní cyklus Maven frameworku

Životní cyklus Maven frameworku se skládá z tzv. fází. Každá fáze se vykonává ve specifickém pořadí. Následující výčet popisuje všechny fáze Maven frameworku v pořadí, v jakém se vykonávají.

1. Validation – kontrola zda je projekt validní a obsahuje všechny nezbytné informace
2. Compile – kompilace zdrojového kódu aplikace
3. Test – spuštění testů obsažených ve složce src/test/resources. Testy se vykonají nad zkompilovaným kódem
4. Package – „zabalení“ zkompilovaného kódu aplikace do jeho distribučního formátu (jar, war, ear atd.)

5. Integration-test – V případě potřeby, nasazení distribučního balíčku z předchozího kroku do prostředí, kde se následně vykonají integrační testy
6. Verify – Pokud jsou definovaná pravidla pro zkontrolování kvality distribučního balíčku, pak jsou v tomto kroku vykonána
7. Install – nahrání distribučního balíčku do lokálního repozitáře Mavenu (neplést si s repositáři ve smyslu SCM), pro účely použití jako závislost pro jiné projekty
8. Deploy – poslední fáze, nahrání distribučního balíčku do centrálního Maven repozitáře. Tímto se balíček zpřístupní ostatním vývojářům a projektům

Jednotlivé funkce a paginy definované v pom.xml se váží vždy na jednu z konkrétních fází. Fáze se vykonávají tak, že se zvolí konečná fáze, která se má vykonat, tedy například Install. Životní cyklus pak začíná ve fázi Validation a postupně prochází dalšími fázemi, až dokud nevykoná zvolenou fázi. Životní cyklus poté končí.

Příklad:

V příkazové řádce se přepneme do složky projektu, do kořenového adresáře, kde se nachází pom.xml. Vykonáme příkaz mvn install. V tomto pořadí se vykonají následující fáze: Validation, Compile, Test, Package, Integration-test, Vefify a Install. Všechny funkce a pluginy navázané na tyto fáze se vykonají v pořadí, v jakém se vykonali fáze, na které jsou navázané.

5.3 Struktura POUD projektu

Struktura projektu POUD má následující podobu:

```
my-app
|-- pom.xml
|-- src
|   |-- assembly
|   |-- main
|   |   |-- java
|   |   |   `-- org
|   |   |       `-- cvut.cz
|   |   |           `-- poud
|   |   |               `-- `balíčky aplikace`
|   |   |-- resources
|   |   |   `-- application.properties
|   |   `-- webapp
|   |       |-- pages
|   |       |   |-- protected
|   |       |   `-- public
|   |       |       `-- error
|   |       |-- resources
|   |       |   |-- css
|   |       |   |-- images
|   |       |   `-- js
|   |       `-- WEB-INF
|   |           `-- templates
|   `-- test
|       |-- java
|       |   `-- org
|       |       `-- cvut.cz
|       |           `-- poud
|       |               `-- `balíčky testů`
|       `-- resources
|           `-- test.properties
`-- etc
    |-- conf
    |-- sql
    |   `-- init
    `-- doc
```

Navíc oproti standardu, se v POUD aplikaci nachází tyto složky:

Src/assembly – obsahuje konfigurační soubor, který definuje strukturu a obsah distribučního balíčku, který se vytvoří ve fázi Install (viz. Maven lifecycle).

Src/main/webapp – obsahuje data spojená s prezentační vrstvou aplikace.

Src/main/webapp/pages – obsahuje zdrojové kódy jednotlivých webových stránek, ze kterých se POUD aplikace skládá

Src/main/webapp/pages/protected – webové stránky, ke kterým je omezený přístup. Uživatel musí být přihlášený a mít patřičná oprávnění pro zobrazení obsahu.

Src/main/webapp/pages/public – veřejně přístupný webový obsah aplikace POUD. Zejména uvítací stránka a Login stránka.

Src/main/webapp/pages/public/error – obsahuje chybové stránky, které se uživateli zobrazí v případě, kdy v systému nastane určitá chyba. Lze definovat, jaká stránka se pro konkrétní typ chyby ukáže.

Src/main/webapp/resources – konfigurace, skripty a další podpůrné data potřebná pro správné fungování webových stránek

Src/main/webapp/resources/css – obsahuje definice stylu (CSS)

Src/main/webapp/resources/images – obsahuje grafické prvky použité na webových stránkách POUD

Src/main/webapp/resources/js – obsahuje java skripty

Src/main/webapp/WEB-INF – tato složka je velmi důležitá. Její absence by znemožnila spuštění aplikace, protože obsahuje soubor web.xml [23] , který je ze specifikace WAR distribučního balíčku (Web application Archive) povinný. Tento konfigurační soubor definuje strukturu webové aplikace[1]. Nachází se zde taky konfigurační soubor faces-config.xml. O tom si povíme více v kapitole o implementaci.

Src/main/webapp/WEB-INF/templates – zde jsou uložené šablony, pomocí kterých se pak generují jednotlivé webové stránky. Důvod, proč jsou šablony uloženy zde a né ve složce src/main/webapp/pages je prostý. Vše co je uložené ve složce Src/main/webapp/WEB-INF, k tomu nelze přistoupit zvenčí. Kontejner, který implementuje specifikace WAR archivačního souboru znemožní jakoukoliv interakci se složkami a soubory v této složce. Zatímco adresář src/main/webapp/pages a vše co se vněm nachází je odkazovatelný z vnějšku. Šablony obsahuji informace o struktuře webových stránek, což se dá analogicky přirovnat k zdrojovým kódům aplikace, a ty chceme přirozeně chránit. Jednotlivé stránky ve složce src/main/webapp/pages pak definují pouze obsah, nikoliv strukturu.

Etc/conf – obsahuje konfigurační soubory specifické pro POUD aplikaci. Zejména pak konfigurační soubory pro spojení s databází, pro logování a pro konfigurace dílčích částí aplikace. Příkladem může být konfigurace úrovně oprávnění pro webové stránky, pro které není explicitně nastavené žádné pravidlo.

Etc/conf/sql/init – zde jsou skripty obsahující DDL skripty pro vytvoření databázového schématu pro POUD aplikaci a DML skripty pro vytvoření základní sady dat a pro vytvoření tzv. master dat. Mezi základní sadu dat patří například data definující role, oprávnění a mapování mezi nimi.

Etc/doc – zde se nachází různé informace o aplikaci ve formě jednoduchých textových souborů. Tyto dokumenty slouží vývojářům a dalším členům týmů pro předávání důležitých informací ohledně vnitřku aplikace. Například jsou zde informace popisující rozchození aplikace ve vývojářském prostředí. Pokud si projekt udržuje i changelogy, ty se pak udržují zde.

Výše je popsaný kompletní výčet adresářů aplikace POUD. Není zde však zachyceno, co vše je součástí výsledného distribučního balíčku a co vše je dostupné na classpath aplikace. POUD aplikace rozlišuje dva druhy classpath, a to test classpath určené pro běh testů a runtime classpath určené pro běh samotné aplikace.

[1] Ne aplikace samotné z pohledu zdrojových či konfiguračních souboru, ale z pohledu jeho webových komponent, zejména pak Servletů, mapování URL aplikace na kontrolery atd. Contejnér, tedy server, který pak takovýto soubor načte, ví, kde hledat základní části aplikace a jak je inicializovat a zpřístupnit okolnímu světu.

Classpath je pojem ze světa Javy a definuje místo, ze kterého Java Virtual Machine (JVM) nebo Java Compiler načítají javovské třídy, balíčky, konfigurační soubory a další důležitá data potřebná pro chod aplikace. Co se nenachází na classpath, není pro JVM viditelné a musí se k tomu explicitně přistupovat, například jako k binárním datům uložených na souborovém systému zastřešujícího operačního systému. Takováto data jsou pak načítána knihovnami (Java má širokou sadu takovýchto knihoven ve svém základním balíčku JRE – Java Runtime Environment – součást JVM), které jsou schopny skrze JVM a následně skrze API operačního systému tyto data číst.

Seznam složek, jejichž obsah se nachází na runtime classpath:

- src/main/java
- src/main/resoucres
- etc/conf

Složky obsahující data ohledně presentační vrstvy (webové stránky, obrázky, java skripty adt.) nejsou na classpath, protože nejsou vykovávány aplikací samotnou. Tyto soubory jsou zpracovávány zastřešujícím kontejnerem (serverem), který podporuje danou technologii, ve které jsou webové stránky vytvořeny.

POUD využívá pro prezentační vrstvu frameworku Java server faces (JSF). Může se zdát matoucí, že ač je framework JSF v podstatě sada Javovských tříd, tak přesto webové stránky které jsou pomocí JSF napsané, nejsou na classpath projektu. Vysvětlení je následující. JSF třídy jsou na classpath kontejneru. Web.xml soubor definuje mimo jiné, že příchozí requesty na server mají být zpracované servlety JSF frameworku. Dále v konfiguračním souboru faces-config.xml je definováno mapování konkrétních URL na konkrétní kontrolery, které jsou už součástí classpath aplikace. Jedná se tedy a pomyslný můstek mezi classpath kontejneru a aplikací. Příchozí http request je kontejnerem nasměrován na patřičný servlet JSF frameworku (definovane ve web.xml souboru) a tento Servlet na základě konfigurace v souboru faces-config.xml pak deleguje tento request na controller v aplikaci. Kontroler request obslouží, vygeneruje webovou stránku a naplní ji daty. Webová stránka je pak následně stejnou cestou vrácena webovému kontejneru, který jí opět pomocí http protokolu vrátí volajícímu klientovi.

Pro úplnost, zde je seznam složek, které jsou na test classpath:

- src/test/java
- src/test/resources

5.4 POUD a Maven

Vazba mezi Aplikací a Maven se provádí skrze soubor pom.xml. POM musí definovat základní minimum ve formě identifikačních údajů aplikace.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

      <groupId>org.cvut.cz</groupId>
      <artifactId>poud</artifactId>
      <version>0.0.1-SNAPSHOT</version>

</project>
```

Výše je ilustrační ukázka pom.xml souboru. Tag <project> je kořenovým elementem, který definuje verzi pom.xml souboru, dále definuje XSD soubor pro validační potřeby a definice jmenných prostorů. Nezbytné minimum, které musí každý projekt založený na Maven je následující [13] :

- Group ID – definice organizace, či obecněji subjektu, který zastřešuje konkrétní projekt. Filosofie je taková, že každá organizace by měla mít svůj vlastní a především globálně (rozuměj, celosvětově unikátní) unikátní identifikátor.
- Artifact ID – identifikátor softwaru v rámci organizace.
- Version – jak již název napovídá, jedná se o verzi softwaru. V ilustrační ukázce výše se ve verzi vyskytuje slovo 'SNAPSHOT'. Toto slovo indikuje, že se jedná o verzi v určité fázi vývoje, ne o finální tzv. major verzi. Volba techniky číslování verzí je ponechána jednotlivým zastřešujícím organizacím. Je ale dobrou praxí rozlišovat alespoň tři úrovně. Majoritní, minoritní a tzv. hotfix verze.

Aby tyto tři identifikátory jednoznačně identifikovali software a to jak v rámci trhu, tak v rámci organizace, je zapotřebí, aby kombinace Group ID, Artifact ID a Version byly globálně unikátní. Dvojice Group ID, Artifact ID dokáže jednoznačně určit software jako takový, nicméně až se spojením s Version definujeme konkrétní produkt s konkrétními vlastnostmi a funkcemi.

Maven poskytuje velikou flexibilitu a možnosti co se týče konfigurace buildovacího procesu a s ním spojené další funkce (generování reportů, předzpracování dílčích částí projektu atd). Popíši zde pouze důležité a zajímavé prvky, které projekt POUD využívá.

5.4.1 Maven pluginy

Jak už bylo zmíněno v kapitole věnující se základnímu popisu frameworku Maven, pluginy jsou funkce, které se váží na konkrétní fáze životního cyklu. Následuje výčet a popis pluginů, které POUD aplikace využívá.

Compiler plugin

```
<plugin>
	<artifactId>maven-compiler-plugin</artifactId>
	<version>3.1</version>
	<configuration>
		<source>${java.version}</source>
		<target>${java.version}</target>
	</configuration>
</plugin>
```

Plugin slouží k základnímu nastavení kompilace. Podstatná část nastavení je definovaná tagem <target>, která definuje, pro jakou verzi Javy bude aplikace skompilována. Zde se na chvíli zastavím.

Kromě verze Javy, ještě rozlišujeme platformu, pro kterou je zdrojový kód zkompilovaný. V dnešní době máme v podstatě tři možnosti. A to platformu x86/x64 a ARM platformu používanou v mobilních zařízeních. Podle specifikace, Java je tzv. cross-platform solution. To v jednoduchosti znamená, že zkompilovaná aplikace lze nasadit na různých prostředích bez ohledu na implementaci aplikace. Java se takto zakořenila v povědomí mnoha vývojářů, nicméně vlastnost cross-platform má své limity. V podstatě se nelze spolehnout na 100% kompatibilitu vaší aplikace v žádném jiném prostředí, než ve kterém jste aplikaci zkompilovali.

V praxi se téměř vždy setkáváme s několika druhy prostředí. Vývojáři mají svá vlastní prostředí nakonfigurované na svých vývojových stanicích, další prostředí se udržuje na strojích zajišťující tzv. continuous integration, které se stará mimo jiné o kompilaci aplikace, kterou v případě potřeby může automaticky nasadit na prostředí testovací. Dalším prostředím je prostředí produkční, tedy prostředí určené pro hostování aplikace pro účel ostrého provozu. Počet prostředí, který může rychle růst, nám přináší nutnost vkládat úsilí do kontroly, zda je aplikace správně zkompilovaná a zda je kompatibilní s cílovým prostředím. K tomu nám slouží právě výše zmíněný plugin. Je jednoduší přizpůsobit nastavení kompilace cílovému prostředí, než přizpůsobit prostředí zkompilované aplikaci.

Assembly plugin

```
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-assembly-plugin</artifactId>
	<version>2.2-beta-3</version>
	<configuration>
		<descriptor>src/assembly/poud.xml</descriptor>
		<finalName>poud-${pom.version}</finalName>
	</configuration>
</plugin>
```

Účel tohoto pluginu je definovat a řídit proces kompletace aplikace. Pomocí tohoto pluginu definujeme, jakou podobu bude mít výstupní artefakt z procesu sestavování výsledného softwaru. Základem konfigurace pluginu je deskriptor, což je v podstatě XML konfigurační soubor.

```
<assembly>
    <id>poud</id>
      <formats>
        <format>zip</format>
    </formats>
      <fileSets>
            <fileSet>
                  <directory>target</directory>
                  <outputDirectory>binaries</outputDirectory>
                  <includes>
                        <include>*.war</include>
                  </includes>
            </fileSet>
            <fileSet>
                  <directory>etc/doc</directory>
                  <outputDirectory>documentation</outputDirectory>
                  <includes>
                        <include>readme.txt</include>
                  </includes>
            </fileSet>
            <fileSet>
                  <directory>etc/conf</directory>
                  <outputDirectory>configuration/poud</outputDirectory>
                  <includes>
                        <include>poud.properties</include>
                        <include>poud_test.properties</include>
                        <include>poud_live.properties</include>
                        <include>**/*.sql</include>
                        <include>log4j-poud-live.properties</include>
                        <include>log4j-poud-test.properties</include>
                  </includes>
            </fileSet>
      </fileSets>
</assembly>
```

Ukázka deskriptoru výše definuje jako výstupní artefakt zip soubor, který obsahuje kromě samotné spustitelné aplikace (war soubor) také sadu konfigurační souborů, databázových inicializačních skriptů a dokumentaci. Díky tomu fáze sestavení aplikace vyprodukuje jediný soubor, který obsahuje veškerá nutná data potřebná k nasazení aplikace, její konfiguraci a pokud aplikaci nasazujeme poprvé, tak jsou zde přítomné i SQL skripty, které inicializují databázové schéma a vloží nezbytně nutná data pro základní běh aplikace. Rozdíl mezi kompilačním pluginema a assembly pluginem je tedy v tom, že kompilační plugin se stará o kompilaci, tedy nějakým způsobem přetváří zdrojové kódy do spustitelné aplikace, zatímco assembly plugin pouze manipuluje s již hotovými daty a kompletuje z nich logické celky (artefakty). Assembly plugin nijak nezasahuje do obsahu či struktury stávajících dat.

Sysdeo Tomcat plugin

Tento plugin slouží jako podpora vývoje. Základní funkcí pluginu je poskytnout integraci mezi aplikací vyvíjenou v Eclipse IDE a běhovým prostředí. Podrobnými informacemi o běhovém prostředí se zabývá kapitola Vývojové a běhové prostředí.

Ukázka konfigurace Szsdeo Tomcat pluginu

```
<plugin>
	<groupId>org.codehaus.mojo</groupId>
	<artifactId>sysdeo-tomcat-maven-plugin</artifactId>
	<version>1.1</version>
	<configuration>
		<useClasspathVariable>false</useClasspathVariable>
		<webPath>poud</webPath>
	</configuration>
</plugin>
```

Výhodou pluginu a také důvod proč jej využívám, je možnost rychlého přechodu mezi vyvíjenou aplikací a spustitelnou aplikací. Pro spuštění aplikace není potřeba ručně nasazovat zkompilovaný kód na server. Vše se děje podle konfigurace pluginu a konfigurace serveru. Přednosti pluginu spočívají především v tomto:

- Plugin řeší interakci mezi vyvíjenou aplikací a běhovým prostředí Tomcat.
- Umožnuje spuštění a zastavení Tomcatu.
- Řeší závislosti projektu na externích knihovnách/frameworcích.
- Podporuje tzv. hot-swap funkci, díky které je možné bez nutnosti restartu serveru a ztráty informací ohledně právě probíhající session měnit implementace a kompilace java tříd, a to za běhu serveru. Je tedy možné rychle reagovat na aktuální stav aplikace, měnit kód a ihned vidět změny.
- Mapuje Tomcat procesy na Eclipse Java debugger a podporuje debugování v reálném čase.

5.4.2 Maven závislosti

Projekt POUD má mnoho závislostí na externích projektech. Díky schopnostem Mavenu je vývoj značně usnadněn tím, že se vývojář stará pouze o definici závislosti a zdroj, kde je závislost k mání a zbytek je obstarán automaticky. Především se jedná o zpřístupnění závislostí v čase kompilace a kompletace, testování či vývoje. Níže je ukázka definice závislosti na ovladači spojení s databází MySql.

```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<version>${mysql.connector.version}</version>
	<scope>compile</scope>
</dependency>
```

Z ukázky je patrné, že se zde vyskytuje jednoznačný identifikátor softwaru, který chceme do aplikace zavést jako závislost. Tím je kombinace trojice Gourp ID, Artifact ID a Verion. Scope tag definuje, kdy je závislost přístupná. Compile hodnota říká, že závislost je přístupná v čase kompilace. Maven umožnuje definovat různé scope nastavení, z nich dalším zajímavým je test scope. Ten zaručí, že daná závislost bude přístupná pouze ve fázi testování (viz. kapitola o životním cyklu Mavenu) a nebude součástí zkompilovaného kódu, připraveném k nasazení do produkce.

Zdroje, ve kterých Maven hledá dílčí závislosti, se definují buďto přímo v pom.xml souboru, nebo v globální konfiguraci Maven. Konfigurace pomocí pom.xml souboru může vypadat následovně.

```
<repositories>
        <repository>
            <id>central</id>
            <url>${url maven repositáře}</url>
        </repository>
</repositories>
```

V případě že daná závislost není nalezena, pak fáze, ve které je daná závislost vyžadována (compile, test, …), skončí s chybou a aplikace nebude zkompilována.

Možností, které Maven poskytuje je opravdu mnoho a jejich prostudování nechám na čtenáři samotném. Dokumentace k Maven frameworku je dostupná zde [doplnit url…].

V ukázkách konfigurace v pom.xml souboru se často vyskytují tzv. config place holdem, tedy místa, které se odkazují na konkrétní hodnoty definované stranou od konkrétní konfigurace. Následuje krátká ukázka definice hodnoty a odkazování se na ni pomocí place holderu.

```
<project>
        <properties>
                <hibernate.version>3.6.10.Final</hibernate.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.hibernate</groupId>
                        <artifactId>hibernate-core</artifactId>
                        <version>${hibernate.version}</version>
                </dependency>
        </dependencies>
</project>
```

Výhody jsou zjevné. Oddělení definice od konfigurace konkrétních hodnot. Jeden place-holder může být použit na více místech najednou. Tím se umocňuje výhoda tohoto přístupu.

Kapitola 6

Vývojové nástroje a běhové prostředí

V této kapitole se zaměřím na nástroje, které jsem při vývoji aplikace POUD využil. Vysvětlím jejich účel a popíši jejich vlastnosti, kvůli kterým jsem zvolil právě je. Dále v této kapitole popíši běhové prostředí, jeho nastavení a využití jak pro vývoj, tak pro testování.

6.1 Eclipse IDE

Eclipse IDE [14] je vývojové prostředí s podporou pluginovatelného systému. Díky rozšířením pomocí pluginů je možné do Eclipse přidávat nové funkce. Toto vývojové prostředí je primárně určené pro jazyk Java, avšak podporuje, díky pluginům, i další jazyky a technologie. V základu je Eclipse prostředí, které toho samo o sobě mnoho nenabízí. Jeho síla spočívá v rozšiřitelnosti. Existuje mnoho variací, které se liší sadou poskytnutých rozšíření. Vývojář si tak může vybrat některou z nabízených distribucí a dodatečně si je podle potřeby může obohatit o další funkce.

Hlavními důvody, proč jsem si vybral Eclipse IDE jsou tyto:

- Eclipse IDE je open-source projekt s licencí Eclipse Public Licence. Tato licence dovoluje využití Eclipse IDE jak pro nekomerční účely, tak i pro komerční a to bez jakýchkoliv licenčních poplatků
- Komunita stojící za Eclipse projektem je stabilní a projekt Eclipse je stále „živý“ (vyvíjí se, vývojáři reagují na podměty k zlepšení, či na nahlášené chyby)
- Pokud se vývojář setká s problémy spojenými s danou distribucí, či s platformou Elipse samotnou, je vysoká šance nalézt řešení buď na oficiálním fóru dané distribuce, nebo jinde na internetu
- Rozšiřitelnost přináší vývojáři široké spektrum využití, lze si do jednoho prostředí integrovat běhové prostředí, editory různých datových formátů, utility pro testování atd.

6.2 Spring Tool Suite

Pro vývoj aplikace POUD jsem si vybral Eclipse distribuci jménem Spring Tool Suite (STS). Důvodem pro mou volbu byla především sada pluginů a funkcí, které ve svém základě STS má již předinstalované. Zejména se jedná o tyto funkce a pluginy:

- Maven 2
- Integrované nástroje pro podporu vývoje se Spring frameworkem
- Podpora pro Tomcat server
- Podpora pro GIT verzovací nástroj

Především podpora pro vývoj se Spring frameworkem (bližší informace nalezne čtenář v kapitole o použitých technologiích) přináší opravdové přednosti této distribuce. Díky vestavěným editorům Springovských konfiguračních souborů, které podporují funkce jako našeptávání hodnot, které lze dosadit do konkrétních polí, „proklikávání“ se z konfiguračního souboru do zdrojového kódu, validace nejen na základě XSD schématu, ale také na základě vestavěné logik, kdy jsou vám poskytnuty i bližší informace, proč je daná konfigurace špatně definovaná, či jak by měla správná konfigurace vypadat s ohledem na aktuální verze a praktik Spring frameworku.

6.3 MySQL

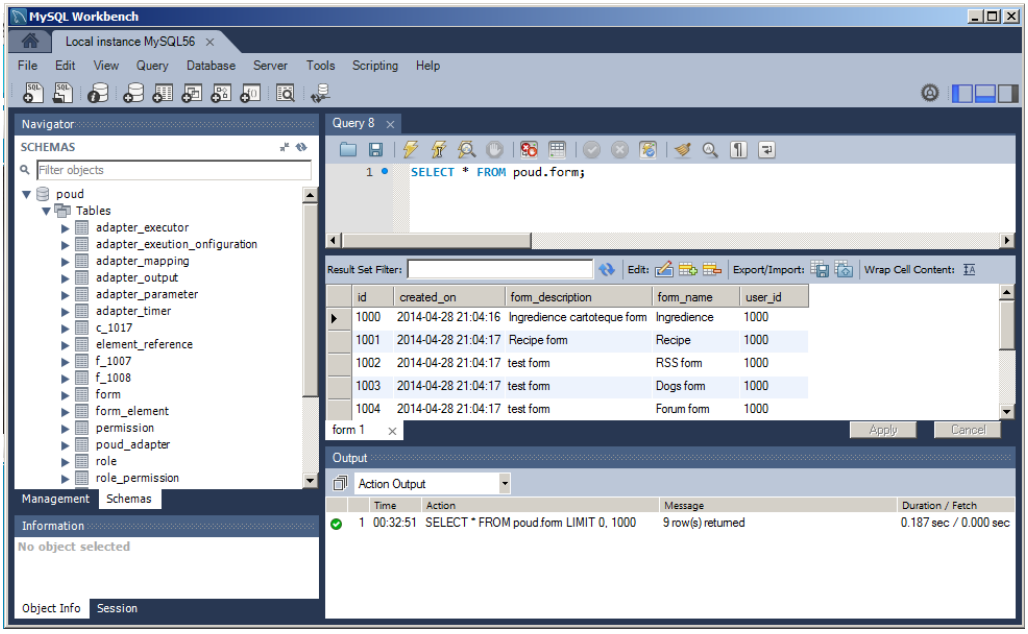
Pro práci s databází jsem svolil databázi MySQL [15] Jedná se o open-source projekt, který je k březnu 2014 druhým nejvyužívanějším databázovým systémem na světě. K mání jsou různé licence od GPL po specifické komerční licence vázané ke konkrétním edicím MySQL.

V základu je databáze MySQL distribuována bez GUI rozhraní. Pro základní konfiguraci a práci s databází je tak potřeba využít příkazového řádku. Nicméně existuje několik dalších variant, které si lze bezplatně obstarat. Zejména administrační nástroje ve formě webové aplikace, či desktopové aplikace.

Z následujících důvodů jsem si vybral MySQL databázi:

- Jedná se o multi-platformní řešení
- GPL licence
- Podpora SQL specifikace ANSI SQL 99
- Podpora UTF-8
- Podpora kešování SQL dotazů
- Podle zvoleného enginu jsou přístupné funkce jako transakce či referenční integrita

Specifikum MySQL databáze je možnost volby enginu, který se stará o práci s daty. Pro aplikaci POUD jsem zvolil InnoDB engine. Jednak podporuje transakce a také referenční integrity, základní CRUD operace či různé úrovně izolovanosti transakce. Pro administraci MySQL databáze jsem zvolil desktopové řešení v podobě aplikace MySQL Workbanch.



Obrázek 10: MySQL Workbanch s otevřeným schématem aplikace POUD

6.4 Google Chrome

Nedílnou součástí vývoje webové aplikace je webový prohlížeč. Je to základní pomůcka při ověřování správnosti implementace. Google Chrome jsem zvolil především na základě osobních zkušeností. Velmi zřídka jsem se setkal se situací, kdy by nějaké části tzv. front-endu aplikace nefungovali v Chromu, ale v konkurenčních prohlížečích ano. Nicméně toto má i svojí stinnou stránku. Pokud se zaměříme při vývoji jenom na jeden prohlížeč, lehko se můžeme dostat do situace, kdy naše aplikace nebude řádně fungovat v prohlížečích jiných. Zejména se jedná o problémy se stylizací a asynchronní komunikací se serverem.

Dalším důvodem, i když ne absolutní předností, je vestavěná sada nástrojů, která umožní v reálném čase kontrolovat data, která si prohlížeč se serverem vyměňují, WYSIWYG (what you see is what you get) editor stylů, debuger java skriptů či prohlížeč Cookies.

6.5 Enterprise Architect

Enterprise Architekt (EA) je vizualizační modelovací nástroj držící se standardu UML 2.0 spravovaný organizací OMG. EA přináší nástroje pro podporu jednotlivých fází životního cyklu vývoje aplikace, počínaje požadavky, návrhem, implementací a konče testováním a údržbou. Nedílnou součástí je také implementace Model-Driven Architecture (MDA) vývojového přístupu.

I přes velké možnosti a podporované funkce jsem EA využil jen na malou podmnožinu úkonů. Na projektu POUD jsem EA využil především k analýze požadavků a k tvorbě diagramů komponent, sekvenčním diagramům a k diagramům tříd.

6.6 Apache Tomcat

Tomcat [17] nelze obecně za vývojové prostředí považovat, nicméně jej zde uvádím, protože nedílnou součástí vývoje POUD aplikace.

Tomcat je v mnoha článcích a literatuře chybně označován jako Aplikační server. Tomcat je Java Servlet a Java Server Pages kontejnerem podle specifikace Java Community Process. Rozdíl mezi obecným aplikačním serverem a Tomcatem je především v dostupnosti funkcí, které jednotlivá řešení poskytují a v pokrytí potřeb běžné aplikace, které lze těmito technologiemi zastřešit. Tomcat například nemá možnost definovat databázová spojení, či robustní podporu pro škálování aplikace.

Tomcat je jednoduchý, leč dostatečně robustní a flexibilní řešení http webového serveru, který poskytuje základní dovednosti. Jmenovitě se jedná zejména o následující funkce:

- User a deployment management
- Podpora webových kontextů (aplikace běží pod svým kontextem, který má separátní konfiguraci)
- Logování (možnost odklonit standardní výstup aplikace do logu Tomcatu, logování přístupu k aplikaci pomocí http protokolu a další)
- Podpora HTTPS protokolu
- Multi-platformní řešení
- Možnost řešit závislosti projektu tak, že Tomcatu definujeme složku, ve které má tyto závislosti hledat. Tím je pak možné dodávat distribuční balíček softwaru bez těchto závislostí. Tím se v praxi můžeme dostat z několik desitek MB velkého souboru až na jednotky KB.

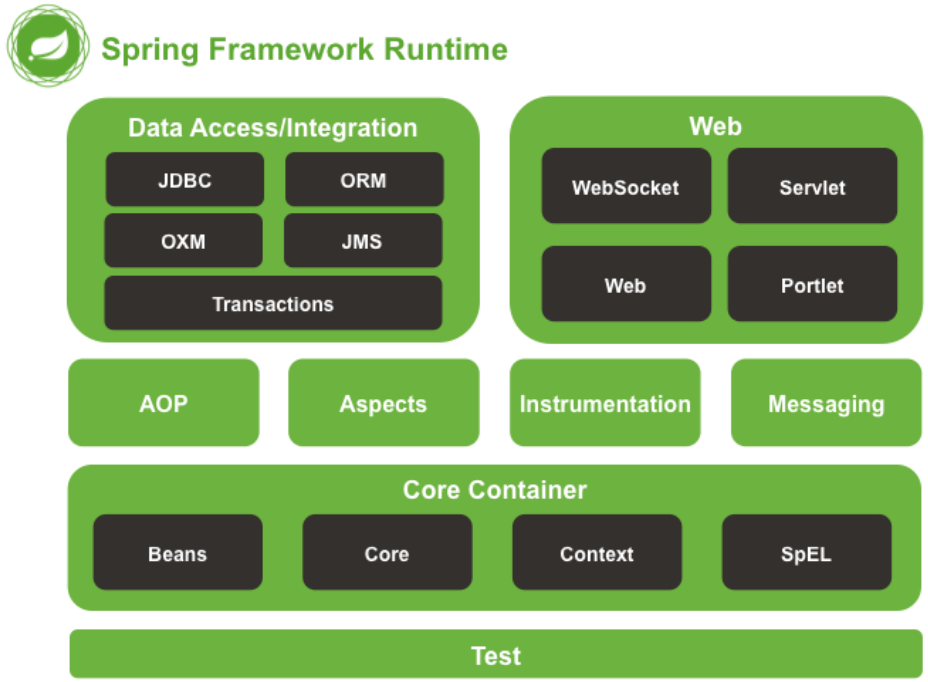
Kapitola 7

Použité technologie

V následujících odstavcích se budu věnovat použitým technologiím. Zaměřím se na ty nejvýznamnější technologie, co se týče dopadu na styl vývoje aplikace a jejich invazivnost, tedy míry pokrytí jednotlivých částí systému POUD.

7.1 Spring framework

Spring [18] je open-source aplikační framework podporující inversion of control kontejnerem pro platformu Java. Jeho velmi pěknou vlastností je neinvazivnost. Framework ve velmi omezené míře diktuje architekturu vašeho kódu a tím ponechává vývojáři volnou ruku. Další podstatnou výhodou je, že Spring ke své činnosti nepotřebuje žádný zastřešující kontejner. Spring je rozdělen do několika základních modulů, které lze v aplikaci použít samostatně. Není tedy třeba zanášet zbytečné závislosti. Příklady modulů mohou být Data access modul, Model-view-controller modul či testing modul.



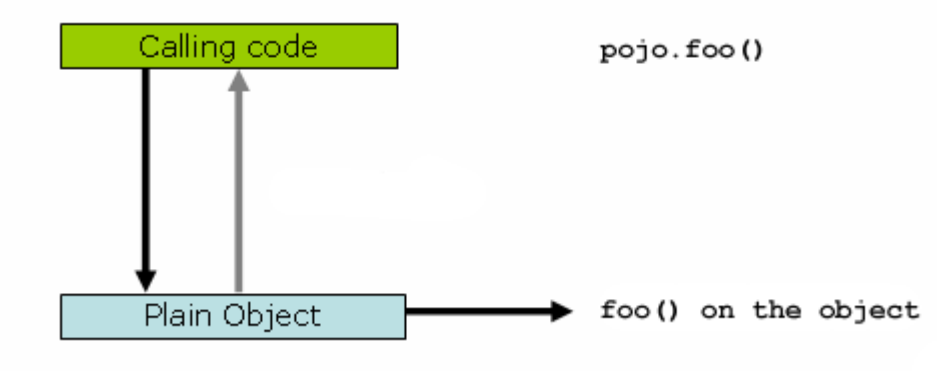
Obrázek 11: Ilustrační pohled na jednotlivé moduly Spring frameworku

Spring framework nabízí dva základní způsoby konfigurace. Jsou jimi konfigurační soubory anebo anotace. V současné době, kdy je aktuální veze springu 4.0, jsou anotace na dostatečně

vyspělé úrovni, aby umožnili pokrýt většinu možností, které poskytují konfigurační soubory. Výhody a nevýhody jednotlivého přístupu jsou na zvážení každého vývojáře. Mělo by však platit, že by se měli vývojáři dohodnout na jednom přístupu v rámci jednoho projektu. Praxe však ukazuje, že do určité míry je výhodné tyto dva způsoby kombinovat.

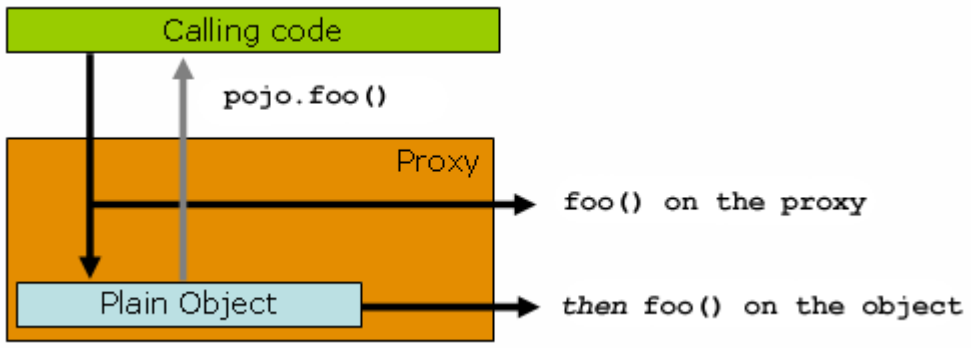
Jádrem spring frameworku je tzv. Inversion of Control kontejner. Spring zpravuje Aplikační kontext, ve kterém jsou registrované tvz. bean objekty. Jedná se o Java objekty. Pomocí aplikačního kontextu je pak možné provádět dependency injection nebo Dependency lookup. Dependency injection je vzor, který dodává požadované závislosti ostatním bean objektům. Existuje několik způsobů jak tyto závislosti předat, například Constructor injection, Setter injection atd. Naopak Dependency lookup umožňuje konkrétnímu bean objektu si vyžádat závislost od aplikačního kontextu.

Aplikační kontext Springu pracuje ne přímo s objekty tříd, ale namísto toho je „obalí“ proxy objekty, které zachycují volání vnějších objektů. Tato volání jsou pak před předáním cílovému objektu zpracována podle toho, jaký typ proxy objektu byl použit.



Obrázek 12: Na obrázku je ilustrace provolávání objektů bez proxy

Bez proxy objektů probíhá komunikace přímo. Na obrázku 11 je tato skutečnost znázorněna jako jednoduché volání metody foo(). Není zde žádný prostředník.



Obrázek 13: Volání referencovaného objektu pomocí proxy

Obrázek 11 ilustruje proxy návrhového vzoru. Volající objekt neví, že volá proxy objekt. Ten implementuje stejné rozhraní jako Plain Object, a tudíž poskytuje stojné veřejné rozhraní. Proxy zachycuje veškeré volání a před předáním tohoto volání cílovému objektu jej může zpracovat, popřípadě vykonat další funkce.

Poud aplikace využívá následující moduly Spring frameworku:

- Spring Core – poskytuje Inversion of Control kontejner spolu s Dependency injection funkcionalitou. Centrální správou bean objektů je pak tzv. Aplikační kontext, který intuitivně zajišťuje správnou funkcí Dependecy a Lookup injection.
- Spring Web & Spring MVC – podpora pro architektonický návrhový vzor MVC pro webové aplikace. Tento modul obsahuje Servlet Dispetcher, který se stará o zpracování příchozích regestu, které mapuje na handlery a pomocí nich generuje výstup, který může nabýt různých podob (HTML stránka, JSON, XML atd.)
- Spring Security – tento modul nabízí komplexní řešení otázky autorizace a autentizace a to jak na úrovni servisní vrstvy, tak i na úrovni prezentační vrstvy. Je možné definovat více zdrojů autorit, které pomáhají vyhodnocovat přístupová práva uživatelů.
- Spring ORM – modul operující na DAO vrstvě aplikace. Nabízí řešení problematik, jakými integrace ORM frameworků třetích stran, transakční managery či konfigurace datových zdrojů.

7.2 Hibernate

Hibernate [21] je cross-platform ORM framework. Aktuální verze je 4.3.5., která plně implementuje standard JPA 2.1. Primární funkcí Hibernate je mapování doménových objektů na relační databázové schéma. Konfigurace Hibernatu se provádí budto konfiguračními soubory, nebo anotacemi.

Nedílnou součástí Hibernate frameworku je tzv. HQL – Hibernate Query Language. Jedná se o nadstavbu nad jazykem SQL, který usnadňuje práci s mapovanými objekty (Entitami). HQL umí ve své syntaxi pracovat s dědičností a polymorfismem. Zápis příkazů se provádí pomocí tečkové notace, která je následně převedena na pozadí na nativní SQL dotaz. HQL tímto odstiňuje uživatele od různých databázových strojů, kdy rozdílné prvky databází řeší automaticky na pozadí. V základu jsou podporovány všechny standardní databáze včetně Oracle, MySQL či PostgreSQL.

Důvody proč zvolit právě Hibernate jako ORM framework jsou především:

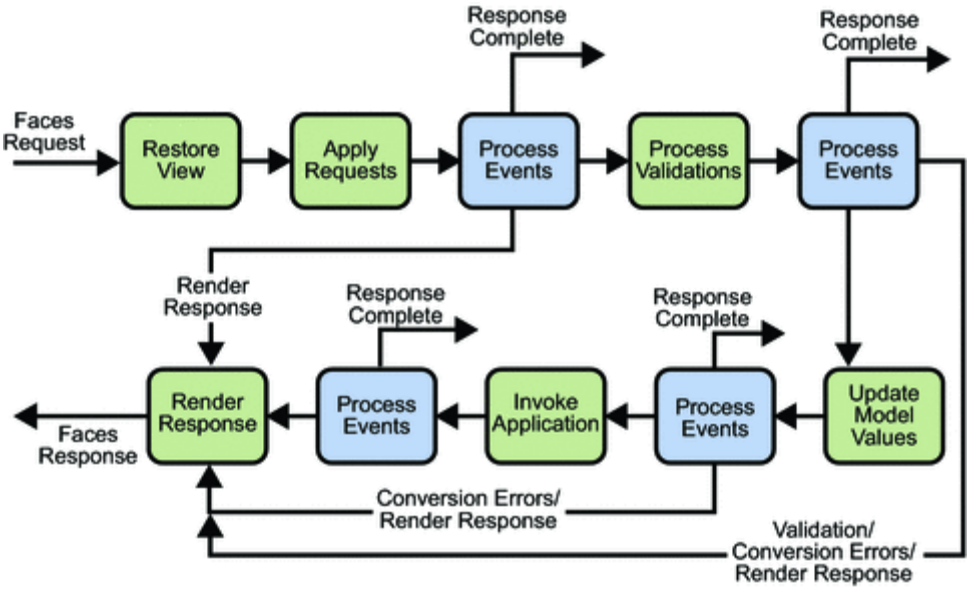
- Implementace standardu JPA 2.1., díky tomu je možné DAO vrstvu aplikace znovu použít i v jiném projektu s jiným ORM frameworkem implementující JPA standard.
- HQL jazyk, který velmi zjednodušuje vytváření komplexních a aplikačně specifických dotazů do databáze s podporou dědičnosti a polymorfismu.
- Podpora standardních funkcí, mezi které patří transakce a řízení úrovně izolovanosti transakcí, kešování na různých úrovních a logování
- Velmi dobrá podpora v podobě komunity stojící za Hibernatem včetně podrobné dokumentace
- Hibernate je open-source projekt s volnou komerční licencí

7.3 Java Server Faces - JSF

Java Server Faces je technologie vivinutá společností Sun Microsystems (dnes Oracle) a je součástí Java 5 Enterprise edition. Framework se řadí mezi tzv. front-end frameworky, tedy frameworky, které se starají o prezentační vrstvu aplikace a zajišťují komunikaci se servisní

vrstvou aplikace. JSF rozděluje aplikaci na uživatelské rozhraní (UI) a na aplikační logiku. UI definuje pomocí tzv. Facetů, což je ve své podstatě sada značek, kterými programátor definuje strukturu a chování webové stránky. Obsah se pak plní pomocí tzv. Managed bean objektů, které v analogii s MVC architektonickým návrhovým vzorem korespondují s komponentou Controller.

Při každém příchozím requestu se vytvoří JSF kontext, který se řídí podle jasně definovaného životního cyklu.



Obrázek 14: Životní cyklus počínaje přijetím requestu až po renderování výsledné webové stránky.

Z obrázku je patrné, že životní cyklus může skončit v několika fázích. Jedná se o případy, kdy dojde například k validační chybě. JSF framework ve velké míře využívá pro své UI komponenty java skriptu, který synchronně či asynchronně komunikuje se serverovou částí.

Jak již bylo řečeno, JSF framework je z vývojářského hlediska sada značek, které jsou definované v tzv. tag library definition (TLD), na které se jednotlivé webové stránky odkazují pomocí meta-dat definovaných v hlavičkách stránek. Díky tomuto konceptu je možné definovat své vlastní TLD soubory. V praxi se tak příliš nesetkáme se standardní TLD JSF frameworku, ale spíše některou s knihoven třetích stran.

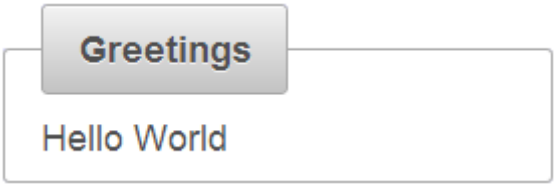
JSF poskytuje moderní prvky při vývoji front-end části aplikace. Patří mezi ně templatovací techniky, komplexní UI prvky jako tabulky se stránkování, asynchronní Ajaxové volání, modální dialogová okna, integraci s dalšími frameworky (např. Spring security pro komplexní rozhodovaní co, kdy a kde se má uživateli zobrazit na základě oprávnění přihlášeného uživatele) a mnoho dalších funkcí.

7.4 PrimeFaces

PrimeFaces [22] je TLD knihovna, nadstavba nad JSF frameworkem. Přináší sadu bohatých UI komponent. Výhodou PrimeFaces je v jednoduchosti s jakou ho lze začlenit do projektu. Není třeba žádné explicitní konfigurace nebo poskytnout další závislosti na externích projektech. Stačí stáhnout jeden jar soubor, importovat namespace do webové stránky a PrimeFaces framework je připraven k použití.

```
<html                              xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">

      <h:head></h:head>

      <h:body>

            <p:fieldSet legent=”Greetings”>

                  <h:outputText value=”Hello World” />

            </p:fieldSet>

      </h:body>

</html>
```

V bloku kódu výše je zobrazena jednoduchá webová stránka, která zobrazí zprávu Hello World. Z kódu je patrné, že jsme pro vyrenderování obsahu stránky použili dvě TLD knihovny, a to JSF nativní knihovnu s namespace „h“ a PrimeFaces knihovnu s namespace „p“. Je tedy možné použít více knihoven na jedné webové stránce, což ponáší další výhody spojené množstvím dostupných UI prvků a funkcionalit.



Obrázek 15: Ukázka jednoduchého UI prvku renderovaného pomocí PrimeFaces frameworku

Obrázek 15 ilustruje výstup ze zdrojového kódu, který je demonstrován výše. Lze vypozorovat, že výstup už má základní styly, které přichází společně s frameworkem. Samozřejmostí je možnost tyto styly pomocí CSS technologie měnit dle libosti.

7.5 JUnit

JUnit framework je dnes považován za standardní nástroj pro testování aplikací založených na platformě Java. Tento framework je vhodný jak pro Test Driven Development (TDD) styl vývoje tak i pro různé metodiky testování, zahrnujíc black & white box testování či integrační i regresní testování. Široká škála frameworků třetích stran (např. Spring framework) přímo integruje podporu pro JUnit framework a nabízí flexibilní nástroje pro testování aplikace.

Pomocí anotací definujeme chování testů. Mezi důležité anotace patří @BeforeClass, @Before, @AfterClass a @Test.

@BeforeClass – tato anotace se může použít pouze na statických metodách. V praxi se používá pro inicializace sdílených nastavení, které jsou nějakým způsobem náročné na zpracování. Např. připojení do databáze.

@Before – metoda označená touto anotací se vykoná před každým jednotlivým testem. Využití je především v případě, kdy chceme připravit data pro jednotlivé testy a ujistit se že všechny potřebné objekty a reference jsou přístupné v době provádění testů.

@AfterClass – opět lze použít pouze ve spojení se statickou metodou a slouží k „uklizení“ nepotřebných dat a instancí po skončení posledního testu.

@Test – metoda označená touto anotací se stává testovací metodou. Při spuštění testů se jednotlivé metody takto označené postupně vykonají, framework JUnit vyhodnotí jejich běh a na standardní výstup vypíše výsledek. U této anotace je nutné si uvědomit, že pořadí vykonávaných metod nezávisí na pořadí, v jakém jsou metody touto anotací označené.

Základními principy testování pomocí JUnit frameworku, a můžeme říci i testování obecně, jsou zejména tyto:

- Testy musejí být na sobě nezávislé a neměli by brát v úvahu pořadí, v jakém budou vykonány
- Pokud testy sdílejí nějaká data či zdroje, měli by po svém vykonání uvést tyto subjekty do počátečního stavu (pokud jeden test selže, může ovlivnit výsledek ostatních testů)
- Testovat by se měli i chybové případy, tedy stavy, za kterých aplikace selže a zda je na tyto stavy adekvátně reagováno (například vyhozením výjimky)
- Testy musejí být za všech okolností deterministické a reprocesovatelné
- Testovat by se měla business logika, ne implementace (samozřejmě závisí na charakteru aplikace). Dobře napsaný test selže v případě, kdy nastane skutečná chyba nebo se změní požadavky na systém. Refactoring (změna implementace bez změny logiky) by neměl zapříčinit změnu logiky testů.
- Dobrý test je takový test, který nalezne chybu

7.6 EasyMock

EasyMock framework přináší podporu testování v podobě mechanismu mockování. Mock je v terminologii EasyMock frameworku proxy objekt, který simuluje činnost mockovaného objektu. Proxy má stejné rozhraní, ale na rozdíl od mockovaného objektu má explicitně definované chování. Mockování se používá tam, kde potřebujeme otestovat konkrétní funkci daného objektu, k jehož provozu potřebuje určité závislosti, které z nějakého důvodu nemůžeme, nebo nechceme zpřístupnit. Jedná se například o případy DAO vrstev, či externích systémů, kde potřebujeme pouze reagovat na provolání určité vrstvy či funkce.

EasyMock společně s JUnit frameworkem přináší řešení, díky kterému se můžeme při testování částečně nebo úplně zprostit závislostí na konkrétních částech interních či externích systémů. Společně tyto dva frameworky poskytují velmi dobrý základ pro Test Driven Development styl vývoje aplikace.

```
@Test
public void test() {
	List mock = EasyMock.createNiceMock(List.class);
	EasyMock.expect(mock.get(0)).andReturn("one").times(1);
	EasyMock.replay(mock);
	Assert.assertEquals("one", mock.get(0));
	EasyMock.verify(mock);
}
```

Blok kódu výše ilustruje příklad použití EasyMock frameworku společně s JUnit frameworkem. Object mock je proxy object třídy List, která definuje reakci na volání metody get(0) a zároveň vymezuje hranice v podobě počtu povolených volání. V tomto příkladě je povoleno zavolat tuto metodu pouze jedenkrát. Příkaz `EasyMock.replay(mock);` započne monitorování definovaného mock objektu, zatímco příkaz `EasyMock.verify(mock);` vyhodnotí dosavadní chování mockovaného objektu. Tento test proběhne bez chyby a testuje následující:

- Kolekce mock obsahuje hodnotu „one“ na prvním místě (nultý index)
- Metoda get(0) je volána právě jednou

Z příkladu lze vidět, že jsme otestovali funkci standardní kolekce typu List. Ve skutečnosti je List interface, a tudíž je příkladem demonstrováno, že není třeba k provedení testu jeho konkrétní implementace. Díky tomuto můžeme například simulovat databázovou vrstvu, či volání externích webových služeb ve složitých testovacích případech aniž bychom měli v době testování tyto subjekty k dispozici.

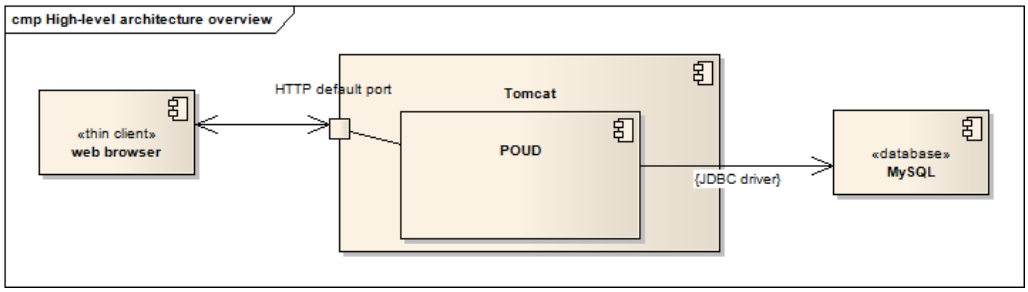
Kapitola 8

Architektura projektu

Tato kapitola pojednává o jednotlivých komponentách projektu a to jak z fyzického hlediska, tak i z hlediska logického. První část se zabývá fyzickými částmi projektu, konkrétně rozdělení na klientskou část, serverovou část a databázi. Následující část popisuje logické rozdělení části serverové do několika ucelených logických vrstev.

POUD je webová aplikace s dynamickým obsahem dat. Touto specifikací je i do značné míry definovaná architektura projektu. Je nutné řešit uložiště dat, aplikační a business logiku a to vše zastřešit uživatelským rozhraním, které data zobrazuje a naopak od uživatelů přijímá jejich akce.

Na následujícím diagramu komponent je znázorněn vysoko úrovňový pohled na architekturu aplikace POUD.



Obrázek 16: Vysoko úrovňový pohled na architekturu aplikace POUD

Obrázek 16 znázorňuje tři hlavní komponenty aplikace, web broker, POUD a MySQL a komponentu Tomcat, která poskytuje běhové prostředí aplikaci POUD.

8.1 Komponenta MySql

Komponenta představuje databázový stroj, konkrétněji MySQL databázi s datovým engine InnoDB. Spojení mezi aplikací a databází je řízeno ze strany aplikace, která k tomuto účelu využívá ovladač com.mysql.jdbc.Driver. Aplikace se stará o kompletní management připojení do databáze včetně přihlášení, spravování poolu konektivit, řízení životního cyklu jednotlivých session a další. Spojení mezi POUD aplikací a databází je provedeno napřímo, je tedy snadné se v případě potřeby přesměrovat na jinou databázi. Vše podstatné (nastavení spojení s databazí…) je konfigurovatelné na úrovni aplikace. Komunikace probíhá jednocestně a to směrem od aplikace k databázi. Aplikace posílá databázi asynchronní příkazy, které po zpracování databází vrátí výsledek, na který aplikace dále může nebo nemusí reagovat.

8.2 Komponenta Tomcat a Web browser

Tomcat [17] je Java Servlet a Java Pages kontejner od společnosti Apache. Slouží jako běhové prostředí pro aplikaci POUD. Tomcat odstinuje celou aplikaci od okolního světa a každý příchozí požadavek (request) je nejdříve zachycen Tomcatem a poté delegován aplikaci.

Web browser je pak webový prohlížeč nainstalovaný na uživatelském počítači. V podání POUD aplikace se jedná o tzv. tenkého klienta (thin client). Klient tedy neprovádí žádné výpočty či business logiku. Pouze data zobrazuje, validuje uživatelské vstupy a zprostředkovává komunikaci se serverem. Komunikace mezi klientem a aplikací probíhá dvěma způsoby. Synchronně a asynchronně. Asynchronní volání je prováděno výhradně JSF UI komponentami.

Před tím, než vysvětlím, jak funguje delegování a zpracování HTTP požadavků, popíši co je to kontext aplikace a jak vypadá. URL aplikace se skládá z následujících částí:

```
{transport_protocol}://{server_url}:{port}/{application_context}/{application_specific_url}
```

Transport_protocol je aplikační protokol operující na aplikační vrstvě ve smyslu ISO/OSI modelu. Možnosti jsou zde HTTP nebo HTTPS.

Server_url je adresa serveru hostujícího aplikaci.

Port definuje fyzický port počítače, na kterém naslouchá. V případě že server (Tomcat, nezaměňovat s fyzickým strojem, hostujícím počítačem) využívá výchozí port pro protokoly HTTP (80) nebo HTTPS (443), pak se port neuvádí.

Application_context je unikátní identifikátor aplikace v rámci serveru. Existují dvě možnosti jak aplication_context vyjádřit. Aplikace totiž může být nasazená v kořenovém adresáři serveru, v tomto případě se pak application_context část vynechává. V druhém případě běží na serveru více aplikací, které se odlišují kontextem.

Application_specific_url definuje konkrétní zdroj (resource) aplikace na který se lze odkazovat. Zpravidla je application_specific_url mapována na controller aplikace, který požadavek zpracuje, nebo deleguje jiné části aplikace.

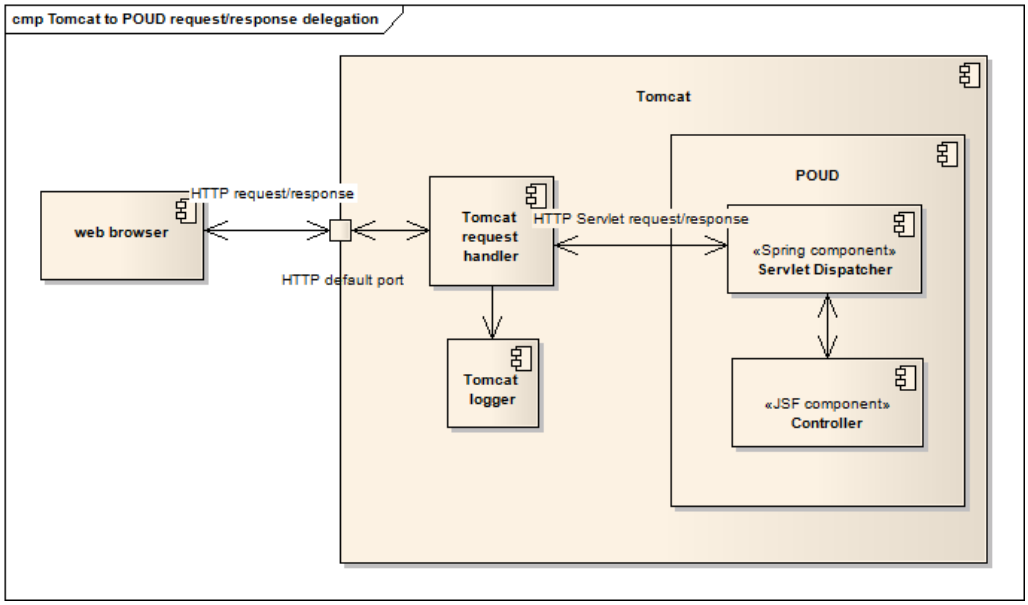
```
https://www.poud.cz/pages/public/login.xhtml
http://localhost:8080/poud/pages/public/login.xhtml
http://www.freehosting.cz/poud/pages/public/login.xhtml
```

V bloku kódu výše je několik ukázek URL aplikace POUD, která je nasazena na Tomcat serveru, který z výchozí konfigurace běží na portu 8080. Tyto URL se odkazují na stejnou stránku a to stránku s přihlašovacím formulářem. URL jsou si ekvivalentní a jejich struktura je dána prostředím, ve kterém je aplikace a hostující server nasazen.

Z příkladu lze vypozorovat, že jediá neměnná část je application_specific_url. Je to dáno tím, že mapování se prování v rámci aplikace a to nezávisle na běhovém prostředí.

8.2.1 Delegace a zpracování http požadavků

V momentě odeslání požadavku (viz. obrázek 17) z uživatelského webového prohlížeče dojde k zachycení na straně Tomcatu, který vytvoří záznam o této skutečnosti do logovacího souboru (komponenty Tomcat request handler a Tomcat logger). Tomcat poté podle URL zjistí kontext, na který je požadavek zaslán a zjistí, zda hostuje aplikaci s tímto kontextem. Pokud ano, přečte si web.xml soubor dané aplikace a podle konfigurace předá požadavek komponentě, která se v aplikaci stará o zpracování příchozích požadavků (komponenta Servlet Dispatcher). Od této chvíle je požadavek pod plnou kontrolou aplikace.



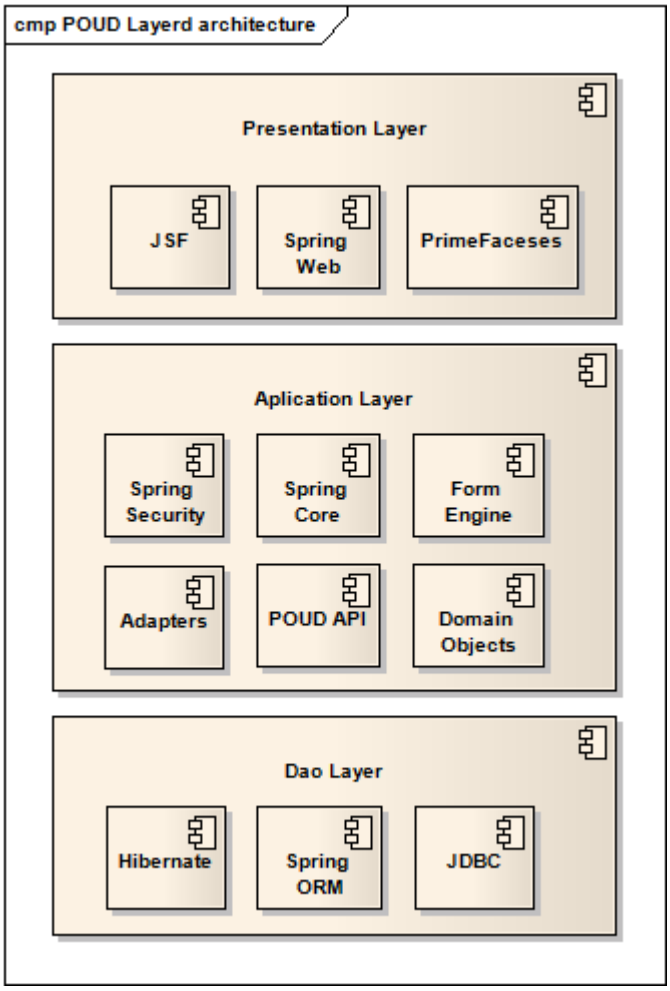
Obrázek 17: Ilustrace propojení jednotlivých komponent starajících se o delegování a zpracování HTTP regestů

Stejnou cestou se pošle odpověď, která se v podobě webové stránky či v jiné formě dostane až k uživateli, respektive do jeho webového prohlížeče.

Jak již bylo zmíněno, web browser komponenta představuje tenkého klienta uživatele v podobě webového přohlížeče. POUD sice velmi často využívá Java skriptu a asynchronních volání pomocí Ajax technologie, ale v drtivé většině se jedná pouze o funkce UI prvků poskytovaných frameworkem JSF. Klient kromě komunikace se serverem vykonává i validace uživatelských vstupů.

8.3 Vrstvená architektura

V příštích několika odstavcích popíši logické celky aplikace POUD, do kterých je rozdělena. Popíši jejich účel, funkce a hranice odpovědnosti. Architektura aplikace POUD je navržena do 3 základních vrstev, které zastřešují aplikační a business logiku, práci s databází a komunikaci s klientem. Teorie točící se kolem vrstvené architektury si může čtenář prostudovat v publikaci [24] .



Obrázek 18: Vrstvená architektura aplikace POUD

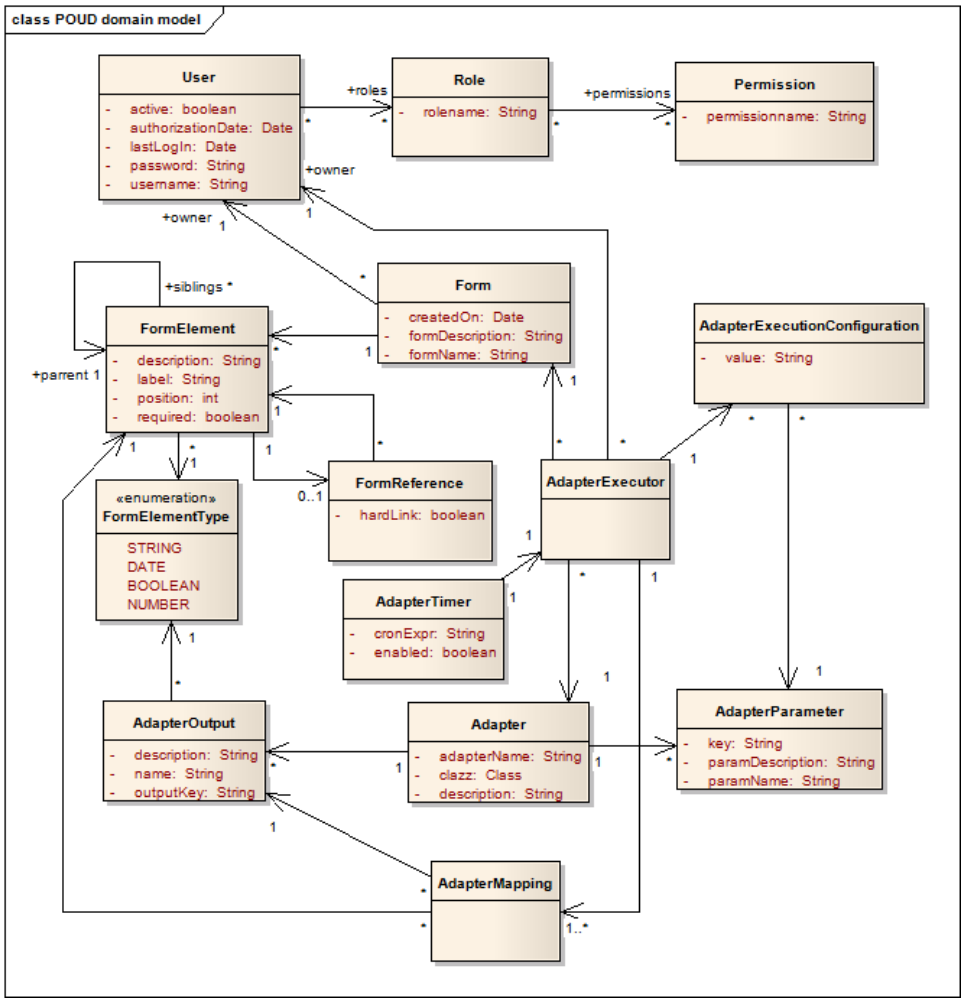
Obrázek 18 ilustruje jednotlivé vrstvy POUD aplikace společně s důležitými komponentami. Komunikace probíhá vždy mezi sousedními vrstvami. Za povšimnutí stojí Dao vrstva, která obsahuje dvě důležité komponenty. Hibernate komponenta přdstavuje ORM řešení pro práci se staticky definovanými doménovými objekty, zatímco JDBC komponenta poskytuje most mezi databází a dynamicky tvořenými strukturami, které jsou spravované Form Engine komponentou.

Kapitola 9

Implementace

Až doposud, jsme mluvili o aplikaci z hlediska požadavků, návrhu a architektury. Všechny tyto části jsou při vývoji aplikace velmi důležité a mají své pevné místo v životním cyklu aplikace. Nicméně pro hmatatelný výsledek, pro použitelný produkt, je potřeba se z těchto fází přenést do fáze implementace. Tomuto tématu věnuji jednu celou kapitolu, kterou rozdělím do několika podkapitol zabývajícími se podstatnými a zajímavými implementačními detaily. Budu se snažit jednotlivé kapitoly strukturovat tak, abych poznatky z kapitol dřívějších mohl použít pro bližší vysvětlení mechanismů použitých v kapitolách pozdějších.

9.1 Doménové objekty



Obrázek 19: Doménový model aplikace POUD

Na obrázku 19 jsou zobrazené základní doménové objekty aplikace POUD. V následujících odstavcích nalezne čtenář bližší popis nejdůležitějších doménových entit.

Form – Entita představuje formulář, který si drží základní meta-data v podobě jména a popisu. Formulář je kontejnerem pro FormElement entity, čímž definuje svojí strukturu (v aplikaci využité především pro IO operace).

FormElement – element držící si základní meta-data ohledně popisku, popisu a typu. Kde typ definuje, s jakým datovým typem element pracuje. Element může být kontejnerem pro ostatní elementy, což ho učiní kompozitním elementem. Tento mechanismus dovoluje vytvářet datové struktury podobné obecným tabulkám.

FormReference – definuje závislosti mezi elementy. Pokud je element typu kompozitního, nelze referenci vytvořit přímo, lze však reference vytvořit na úrovni jeho potomku. Toto není na doménové úrovni ošetřené, a proto se o to stará aplikační vrstva.

Adapter – Hlavní vlastností adapteru je informace o třídě (její fyzické umístění v aplikaci POUD), která implementuje konkrétní funkcionalitu adapteru. Důvodem je možnost persistovat jednotlivé adaptery v databázi společně s informacemi, jako jsou parametry adapterů, výstupy adapterů a mapování adapterů. Díky tomu je možné v případě potřeby restartování aplikace (například z důvodů nasazení nové verze aplikace) jednoduše načíst všechny adaptery do paměti a dále s nimi programově pracovat (spustit časovače atd.). Pokud se objeví požadavek na vytvoření nového adapteru, vývojář musí udělat následující minimum:

1. Implementovat rozhraní PoudAdapter (není znárodněno na obrázku 19)
2. V databázi vytvořit nová aplikační data v podobě záznamu v tabulce Adapter, který namapuje parametrem clazz na implementaci adapteru.
3. V databázi vytvořit záznamy v tabulkách AdapterOutput a AdapterParameter. Tyto záznamy specifikují, jaká nastavení adapter ke své činnosti potřebuje, a jaké má výstupy

Pomocí výše popsaného minima je aplikace POUD schopna automaticky uživateli umožnit adaptéry používat, vytvářet svá vlastní mapování a časovače.

AdapterExecutor – tato entita si drží konkrétní nastavení pro jeden adaptér a konkrétní mapovaní mezi adaptérem a zvoleným formulářem. AdapterExecutor vždy patří právě jednomu uživateli.

AdapterParameter – představuje jednu konkrétní konfigurační hodnotu adaptéru. Definuje jméno parametru, jeho klíč (krátký text, identifikátor) a popis. Každý Adapter si drží sadu těchto parametru, kterými popisuje, jaké hodnoty potřebuje k svému běhu.

AdapterOutput – entita definující výstup adapteru, včetně typu FormElementu na který lze namapovat.

9.2 Autorizace a autentizace v podání Spring frameworku

Spring Security [19] poskytuje zabezpečení na různých úrovních aplikace. Mezi standardně dostupné metody pro webové aplikace patří autorizace na úrovni URL a autorizace na úrovni volání metod. POUD využívá autorizaci a autentizaci na úrovni URL.

Autorizace na úrovni URL se provádí tak, že se v konfiguračním souboru web.xml definuje Servlet filter, jehož konfigurace se provede pomocí springovského konfiguračního souboru. Servlet se poté namapuje na požadované URL (relativně vůči kontextu aplikace). Protože se jedná o standardní Servlet filter, není zde žádná závislost na Spring Mvc modulu. Je tedy možné využít jiný framework pro presentační vrstvu a Spring Security pro autentizaci a autorizaci. Každý příchozí požadavek na namapovanou URL je zpracován tímto servlete, který rozhodne, zda je uživatel autentizován či autorizován požadovat zdroje odkazující se danou URL. Pokud proces selže, Spring vyhodí vyjímku.

Zdroje, ze kterých může Spring čerpat autentizační data jsou:

- HTML formulář (POUD využívá tento typ zdroje)
- Single sign-on
- OpenID
- X.506
- Další…

Spring může evaulovat autentizaci vůči jednomu či více repositáři spravující autentizační data uživatelů:

- JDBC, nebo jiný zdroj na JDBC založený (Hibernate např., využívaný POUD aplikací)
- LDAP
- JEE Containters (JBoss, GlassFish, atd.)

```
<filter>
	<filter-name>springSecurityFilter</filter-name>
	<filter-class>
org.springframework.web.filter.DelegatingFilterProxy
	</filter-class>
</filter>

<filter-mapping>
	<filter-name>springSecurityFilter</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

Ukázka výše je vyňata přímo z konfiguračního souboru web.xml aplikace POUD. První část definuje springovský filter, zatímco část druhá jej mapuje na všechny aplikačně specifické URL. Rozhodnutí, zda je uživatel autorizován či autentizován se provádí na základě springovského konfiguračního souboru:

```
<sec:http pattern="/pages/public/**" security="none" />                  1)
<sec:http pattern="/javax.faces.resource/**" security="none" />
<sec:http use-expressions="true">
       <sec:intercept-url pattern="/pages/protected/form/**"            2)
              access="hasRole('PERMISSION_MANAGE_ FORMS')" />
       <sec:intercept-url pattern="/pages/protected/adapter/**"         3)
              access="hasRole('PERMISSION_MANAGE_ADAPTERS')" />
       <sec:intercept-url pattern="/??/**"                              4)
              access="${environment.access.configuration}" />
       <sec:form-login login-page='/pages/public/login.xhtml'           5)
                     default-target-url="/pages/protected/index.xhtml"
                     authentication-failure-url="/pages/public/login.xhtml" />
       <sec:logout logout-success-url="/pages/public/login.xhtml" />
</sec:http>
```

Řádek 1) definuje, že u stránek na url {base}/poud/public/* se neprovádí žádná autentizace ani autorizace.

Řádek 2) říká, že pokud chce uživatel přistoupit k jakékoliv stránce s formuláři, musí mít oprávnění „PERMISSION_MANAGE_FORMS“.

Rádek 3) je analogický k řádku 2) s tím rozdílem že se týká adaptérů.

Řádek 4) definuje, že přístup na všechny ostatní URL nespecifikované v tomto konfiguračním souboru se řídí podle proměnné „environment.access.configuration“, která je nastavená v konfiguračním souboru aplikace na hodnotu „permit all“

Řádek 5) definuje, na jaké URL se nachází přihlašovací formulář a na jakou URL se má přejít v případě úspěšného nebo neúspěšného přihlášení.

Přihlašovací formulář pak přijímá přihlašovací heslo a jméno, které pošle http požadavkem (zakodované algoritme Base64, je tedy nutné použít HTTPS protokol). Tento požadavek je explicitně zachycen Springovským filtrem, který pak rozhodne, zda je uživatel autentizován.

Evaulace autentizace pak provádí tzv. authentication provider (Jedná se o interface Springu. Lze si vytvořit vlastní provider, nebo použít některý ze základních možností poskytované Springem)

```
<sec:authentication-manager>
       <sec:authentication-provider ref="authenticationProvider" />
</sec:authentication-manager>

<bean id="authenticationProvider"
       class="org.springframework.security.authentication.dao.DaoAuthen
ticationProvider">
       <property name="userDetailsService" ref="userDetailsService" />
       <property name="saltSource" ref="saltSource" />
       <property name="passwordEncoder" ref="passwordEncoder" />
</bean>
```

Výše je ukázka z nastavení autentization provideru. Provider je typu DaoAuthenticationProvider, což znamená, že je využita databáze jako repositář s autentizačními daty uživatelů. Parametr userDetailsService je pak konkrétní implementace aplikace POUD, která se stará o načtení autentizacích dat z databáze a předání je Springu. Parametr saltSource je

přídavný zabezpečovací mechanismus popsaný v kapitole věnující se analýze uživatelů. A nakonec parametr passwordEncoder se stara o hashovaní uživatelského hesla.

9.3 Databázová vrstva

POUD využívá pro persistenci dat dva přístupy. Jedním je ORM framework Hibernate, který pomocí oanotovaných entit provádí veškerou komunikace s databází za nás a druhým přístupem je plain SQL v kombinaci s JDBC.

Hibernate se využívá v případech, kdy můžeme v čase spuštění definovat konečnou a neměnnou množinu mapování entit na databázové schéma. Jedná se zejména o tyto případy:

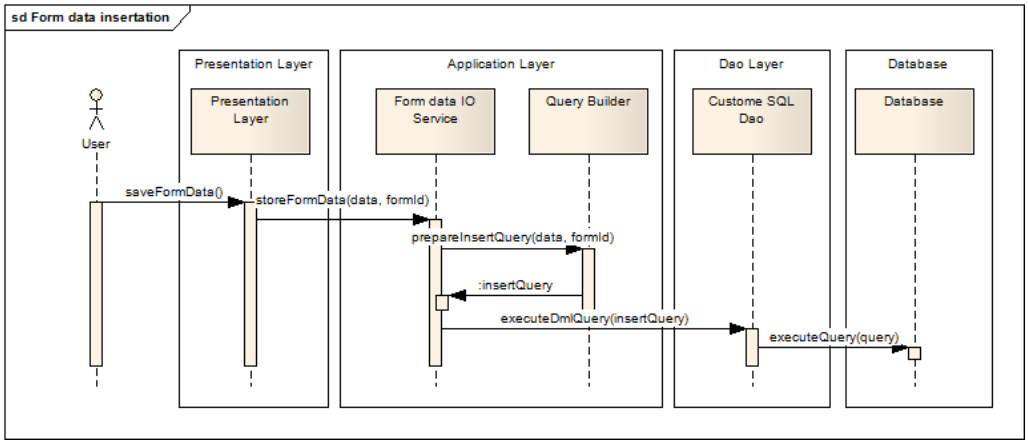
- Informace o uživatelích, jejich rolích a oprávnění
- Meta-data formulářů (entita Form)
- Formulářové elementy a vazby a reference mezi nimi
- Adaptéry včetně jejich konfigurace a výstupních parametrů
- Instance adaptérů (entita AdapterExecutor), včetně mapování výstupů na formuláře, konfigurace a časování

SQL společně s JDBC se používá ve spojitosti s formuláři CRUD operacemi nad jejich daty. Důvod byl již popsán v kapitole věnující se analýze formulářů. Pro tento účel byl vytvořen jednoduchý formulářový engine.

9.3.1 Formulářový engine

Účelem formulářového enginu je nahradit ORM framework pro formulářová data. Protože každý formulář má svojí specifickou strukturu, není možné vytvořit obecnou tabulku pro data všech formulářů. Data se proto ukládají do tabulek, které jsou za chodu aplikace dynamicky vytvářené a spravovávané pro potřeby konkrétních formulářů. Každý formulář je asociován s alespoň jednou tabulkou, do které může ukládat data. Na základě meta-dat je engine schopen data zapisovat a číst ze správných tabulek.

Engine ve své podstatě nemá podobu knihovny či frameworku v distribuovatelné formě. V podání POUD se jedná o souhrn tříd, které zasahují do více vrstev aplikace. Malou, ale důležitou část hraje engine na DAO vrstvě, kde je jednoduchá služba pro vykonávání SQL příkazů (využívá se zde kombinace s Hibernate a plain JDBC), které engine generuje. Na servisní, neboli aplikační vrstvě, je pak logika, která z dat zadaných uživatelem vytvoří zmíněné SQL příkazy, které následně nechá DAO vrstvu vykonat. Obrázek 20 sekvenční diagram ilustruje postup při vytváření formulářového záznamu uživatelem.



Obrázek 20: Sekvenční diagram procesu ukládání formulářových dat

Engine se drží několika jednoduchých pravidel, která lze shrnout do několika bodů:

- Pro každý nově vytvořený formulář vytvoř jednu tabulku, do které se budou ukládat data nekompozitních elementů formuláře. Tabulka se bude jmenovat podle vzoru „F_{id_formuláře}“. Tabulka bude ve svém základu obsahovat sloupec „id“, který bude nastaven na primární klíč tabulky s automatickým inkrementováním.
- Pro každý nově přidaný nekompozitní element vytvoř sloupec v tabulce pro daný formulář, který ponese název „E_{id_elementu}“. Datový typ sloupce se bude odvíjet od typu nekompozitního elementu.
- Pro každý kompozitní element formuláře vytvoř novou tabulku s názvem „C_{id_elementu}“. V základu bude tato tabulka obsahovat sloupce „id“ jako primární klíč a sloupec „parent_id“, který bude jednotlivé záznamy vázat ke konkrétním záznamům v tabulce pro formulář, který obsahuje daný kompozitní element.

  Pro bližší vysvětlení předchozího odstavce, je zapotřebí pochopit následující. Data, která se pomocí formuláře uloží do systému se v terminologii POUD označují jako formulářový záznam. V případě, kdy formulář obsahuje kompozitní element, nastává situace, kdy se k jednomu formulářovému záznamu může vztahovat více záznamů v rámci kompozitního elementu (analogie s tabulkou). Protože tabulka pro kompozitní element obsahuje data pro všechny formulářové záznamy, je zapotřebí nějakým způsobem rozlišit, které záznamy v této tabulce patří ke kterému formulářovému záznamu. A právě k tomuto účelu slouží sloupec „parent_id“
- Pro vytvoření formulářového záznamu urči jméno tabulky/tabulek podle meta-dat formuláře (id formuláře, id elementů atd). Stejně tak urči množinu a jména sloupců, do kterých se data zapíší. Následně vytvoř SQL příkaz. To samé platí analogicky pro čtení dat.
- V případě smazání formuláře, smaž i tabulku se jménem „F_{id_formuláře}“ a všechny korespondující tabulky pro kompozitní elementy, pokud nějaké existují.
- V případě smazání formulářového elementu, odeber bud korespondující sloupec (v případě nekompozitního elementu) pojmenovaném „E_{id_elementu}“ v tabulce se jménem „F_{id_formuláře}“, nebo korespondující tabulku (v případě že se jedná o kompozitní element) se jménem „C_{id_elementu}“.
- Pro vkládání dat do tabulek kompozitních elementu, odebírání nebo přidávání elementu kompozitním elementům platí stejná pravidla jako v případě formuláře.

Díky těmto jednoduchým pravidlům je formulářový engine schopen zprostit vývojáře od přímé interakce s databází (stejně jako ORM frameworky) díky automatizovanému mapování formulářů libovolné struktury na schéma databáze.

Formulářový engine a reference

Předchozí odstavce popisovaly formulářový engine a jeho práci s formulářovými daty. Reference mezi elementy však nespadají do skupiny dat jako takových, ale spadají mezi meta-data formulářů. Reference nejsou tedy nijak formulářovým enginem zohledněny a je na servisní vrstvě, aby se o zastřešení této funkcionality postarala.

9.3.2 Omezení formulářového enginu

Formulářový engine má některá omezení, která zde v bodech popíši. Z pohledu uživatele se tyto nedostatky však nijak neprojeví:

- Pro každý formulář se vytvoří minimálně jedna tabulka. To může vést k velkému množství tabulek. MySQL s InnoDB enginem nemá omezení, co se počtu tabulek v jednom schématu týká, nicméně zde mohou být omezení na straně souborového systému. Podle zvolené databáze se fyzické uložení tabulek provádí zpravidla buďto do jednoho souboru, který představuje celou databázi či schéma nebo je pro každou tabulku vytvořený jeden soubor. Zde můžou nastat komplikace spojené s počtem souborů v jedné složce. Zejména z pohledu omezení počtu souborů, které může jedna složka obsahovat. Moderní souborové systémy s tímto zpravidla nemívají problém (NTFS může adresovat až 4 294 967 295 souborů, unixový Ext2 může adresovat 2^{18} souborů).
- Aktuální implementace POUD spravuje pouze jedno schéma jak pro doménové tabulky, tak i tabulky pro formulářová data. Administrace databáze, zejména pak doménové části se tak může stát značně nepřehledá a složitá. Řešením může být vytvořením druhého schématu, které bude sloužit pouze pro datové tabulky formulářů.
- Jména dynamicky vytvořených tabulek a sloupců (formuláře) jsou automaticky generovaná a vychází z primárních klíčů formulářů (pro zajištění jmenné unikátnosti). V případě, že bude třeba ručně manipulovat s dat formulářů, bude muset patřičná osoba nejdříve odvodit jméno tabulky a popřípadě i jméno sloupce z meta-dat formulářů.

9.4 Aplikační vrstva – Spring dependency injection

Aplikační vrstva, stejně tak i Dao vrstva a prezentační vrstva aplikace POUD hojně využívá dependency injection v podání frameworku Spring [18] . Při implementaci jsem se držel jednoduchého pravidla, a to pravidlo zní „programuj vůči rozhraní“. Myšlenka je jednoduchá, všechny třídy, které se injektují v podobě závislostí, jsou implementací nějakého rozhraní. Datový typ injektovaného objektu (bean v žargonu Springu) je pak typu daného rozhraní. Psaní zdrojového kódu vůči těmto beanám je pak nezávislý na implementaci těchto rozhraní. Tímto je výrazně zvýšená testovatelnost aplikace (snadné mockování, při změně implementace rozhraní není třeba měnit testy, ty budou stále kompilovatelné) a také v případě nutnosti je změna implementace otázkou oanotování nové implementující třídy, či změny konfiguračního souboru springu. Správné dosazení závislosti je pak provedeno automaticky na pozadí Springem.

```
@Service
public class FormMetaModelServiceImpl implements FormMetaModelService {

	private Logger logger = LoggerFactory.getLogger(getClass());

	@Autowired
	private FormMetaModelDao formMetaModelDao;

	@Autowired
	private SqlDao sqlDao;
```

V textovém bloku výše je ukázka definice Springovské beany a definice závislosti na Dao objektu. Anotace Service říká, že tato třída je bean objektem, který se při startu aplikačního kontextu Springu (děje se při startu aplikace, definováno ve web.xml konfiguračním souboru) zaregistruje jako singleton bean (Singleton bean znamená, že kdykoliv je požadovaná závislost typu FormMetaModelService, bude vrácen pokaždé stejný objekt. Spring nabízí více možností, například beana typu prototype vytvoří pro každou odkazující se beanu novou instanci). Anotace Autowired pak Springu říká, že tuto lokální proměnnou má inicializovat beanou typu FormMetaModelDao. Spring se při vytváření aplikačního kontextu podívá, zda má pro tento datový typ zaregistrovanou beanu a pokud ano, pak ji dosadí. Spring využívá pro dependency injection reflexi Javy a proto není třeba ani pro privátní lokální proměnné vytvářet konstruktor s týmže parametrem, či seeter metodu. Zadně inicializace ze strany programátora není potřeba. V kódu s proměnou formMetaModelDao pracuje programátor tak, jako by byla inicializovaná.

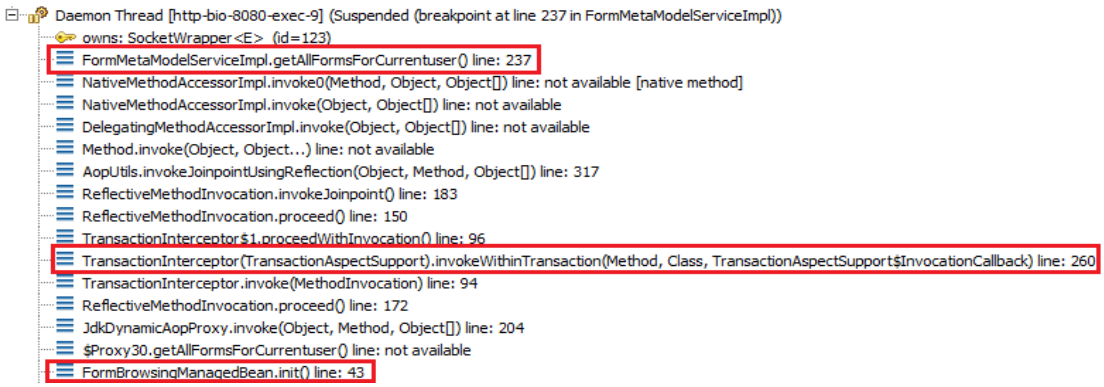
```
@Override
@Transactional
public void deleteForm(Long formId) {
	Form form = formMetaModelDao.findById(formId);
	for (FormElement element : form.getElements()) {
		if (FormElementType.COMPOSED.equals(element.getElementType())) {
			DropTableBuilder dtb =
QueryBuilderUtils.prepareDefaultDropTableBuilder(getFormComposedElementTableNa
me(element));
			String query = dtb.buildQuery();
			sqlDao.executeDdlQuery(query);
		}
	}
	DropTableBuilder dtb =
QueryBuilderUtils.prepareDefaultDropTableBuilder(getFormTableName(form));
	String query = dtb.buildQuery();
	sqlDao.executeDdlQuery(query);
	formMetaModelDao.delete(formId);
	logger.info("Form with id {} has been deleted.", formId);
}
```

Jak lze vidět v ukázce výše (metoda pro smazání formuláře), proměnná formMetaModelDao je využita bez explicitní inicializace. Spring framework poskytuje další velmi užitečnou funkcionalitu, kterou v POUD aplikaci hojně využívám. Je jí anotace Transactional (viz. blok kódu výše). Tato transakce říká, že před vstupem do této metody vytvoří transakční manager novou transakci, která bude aktivní po celou dobu vykonávání metody deleteForm. V momentě opuštění této metody se provedou patřičné operace (závislé na konfiguraci transakčního manageru v konfiguraci Springu). V případě POUD je transakční manager nastaven tak, aby po opuštění metody označené anotací Transactional byl vykonán commit.

```
<bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

POUD využívá transakčná manager Hibernatu, jehož implementace je poskytnuta Spring frameworkem [20] . Transakční manager ke svému chodu potřebuje tzv session faktory objekt, který se stará o komunikaci s databází. Tedy vytváří pro nás session objekty (analogie s JPA EntityManagerFactory a EntityManager), pomocí kterých se vykonávají jednotlivé SQL příkazy. Session Factory pak využívá tzv. data source objekt (taktéž nakonfigurované pomocí Spring bean objektu), který definuje spojení s databází.

V případě že by nastala chyba (vyhodila by se výjimka) před tím, než se vykoná commit příkaz, transakční manager automaticky provede roll back operaci. Z ukázky je patrné, že se vývojář explicitně nestará o správu transakcí. V kapitole o použitých technologií v podkapitole věnující se Springu je zmíněn proxy model využívaný Springem. Veškeré závislosti (proměnné označené anotací Autowired) jsou ve skutečnosti proxy objekty a aby Springovské anotace typu Autowired fungovaly, je potřeba používat tyto proxy objekty. Pokud by programátor zavolal v rámci jedné třídy lokální metody označenou anotací Transactional, pak se tato anotace nebude brát v potaz, protože volání metody neproběhl skrze proxy objekt, ale napřímo.



Obrázek 21: Stack trace ukazující zaobalení servisní třídy proxy objekty, mezi nimiž je i transakční manager.

```
public class FormBrowsingManagedBean implements Serializable {

	@Autowired
	private FormMetaModelService formMetaModelService;

	@PostConstruct
	private void init() {
		formMetaModels                                                    =
		formMetaModelService.getAllFormsForCurrentuser();
	}
```

Na obrázku 21 je výpis ze stack trace aplikace POUD, kde lze vidět volání Managed bean třídy FormBrowsignManagedBean (controller v podání JSF), který se snaží inicializovat tím, že načte všechny dostupné formuláře aktuálně přihlášeného uživatele, které pak následně zobrazí. Volání je znázorněné v bloku kódu výše a z pohledu vývojáře se zde pracuje s rozhraním namísto proxy objektu. Ze stack trace výpisu je však patrné, že se provolává metoda proxy objektu vytvořeného Springem (2. řádka ze spodu výpisu), který před tím než zavolá skutečnou

implementaci rozhraní FormMetaModelService::getAllFormsForCurrentuser() (1. řádek výpisu) provede několik operací. To co se mezi zavoláním proxy objektu a skutečném provolání naší implementace stane je závislé na konfiguraci Springu. Protože naše metoda FormMetaModelService::getAllFormsForCurrentuser() je označená anotací Transactional, je před zavoláním této metody zavolaný transakční manager, který pro nás transakce spravuje automaticky (je zde i mnoho jiných volání, ty jsou ale spojené s reflexí Javy, kterou spring využívá). V momentě, kdy se ukončí vykonávání metody FormMetaModelService::getAllFormsForCurrentuser() se exekuce postupně vrací stejnou cestou, jakou byla vykonána. Tedy vrátí se i skrze transakční manager, který vyhodnotí, zda došlo k chybě (vykoná roll back), nebo zda udělá commit. Nakonec se kontrola řízení vrátí controlleru, který si původně řekl o kýžené formuláře.

Předchozí příklad demonstroval na reálném příkladu, jak funguje dependency injection spolu s proxy modelem v podání Spring frameworku.

9.5 Prezentační vrstva – JSF šablony, PrimeFaces UI komponenty

Pro prezentační vrstvu je použita kombinace frameworku JSF a jeho nadstavba v podobě frameworku Primefaces, který přidává navíc k JSF UI prvky. Základními prvky JSF frameworku je FacesContext (Popsaný v kapitole o použitých technologiích v sekci o JSF frameworku) a tzv. managed bean objektech. FacesContext je objekt, který představuje jeden životní cyklus http požadavku. FacesContext je spjat s jedním či více managed bean objekty, které představují v analogii s MVC architektonickým vzorem controllery. Tyto bean objekty spravují data, s kterými uživatel pomocí GUI manipuluje, a v případě potřeby provolávají nižší vrstvy aplikace. Managed bean objekty mají svůj vlastní životní cyklus nezávislý na FacesContext objektu.

Aplikace POUD využívá tyto scopy pro managed bean objekty:

- Request scope: použito pro přihlašovací stránku. Data, která jsou držena v paměti, jsou platná pouze po dobu jednoho requestu. Pokud přihlášení selže a uživatel se vrátí opět na přihlašovací stránku, je to z pohledu scope nový regest a tudíž se vytvoří i nový managed bean objekt.
- Session scope: Životní cyklus začíná v momentě prvního http requestu a končí v momentě invalidování session na straně serveru. Tento scope využívá managed bean objekt, který drží informace o přihlášeném uživateli. Je to jediný výskyt tohoto scope v celé aplikaci. Obecně platí, že by si server měl držet co nejméně dat v session objektech a právě přihlašovací údaje uživatele (jméno, role, oprávnění atd.) jsou minimum, pro bezproblémový chod aplikace.
- View scope: První http request, pro který se vytvoří FacesContext objekt, který se odkazuje na managed bean objekt s view scope: vytvoří instanci této managed beany a vyrenderuje patřičnou webovou stránku. Životní cyklus scope je platný do doby, než uživatel opustí danou webovou stránku. Je tedy možné v rámci jedné webové stránky provádět více http požadavků bez ztráty stavu controlleru. Tento scope je v POUD aplikaci využíván nejvíce. Především pro webové stránky pro práci s formuláři a adaptéry.

Díky Spring frameworku je možné z JSF managed bean vytvořit Springovský managed bean objekt, a proto je i možné využít dependency injection funkcionality.

```
@Component("formBrowsingBean")
@Scope("view")
@SuppressWarnings("serial")
public class FormBrowsingManagedBean implements Serializable {

…
```

Ukázka kódu výše demonstruje způsob, jakým se JSF Managed Bean objekt zaregistruje v aplikačním kontextu Sprigu. Slouží k tomu anotace Component. Anotace Scope patří také Springu a jejím účelem je vytvořit ze spring bean objektu managed bean objekt frameworku JSF.

9.6 Rozvržení webových stránek

Poud aplikace využívá JSF tagy, především pak include, insert, define a composition z namespace http://java.sun.com/jsf/facelets pro definování šablon webových stránek a jejich obsahu. Následující odstavce popisují tyto tagy více z blízka.

Include

```
…
<h:head>
	<ui:include src="header.xhtml" />
</h:head>
…
```

Tento tag slouží k in-line vložení obsahu jedné webové stránky do stránky druhé. Díky tomu je možné vytvořit větší granularitu webových stránek a vytvořit tak znovupoužitelné části. Cesta zadaná v parametru src je relativní vůči stránce, která tag include používá. Kořen cesty je však ohraničen složkou WEB-INF, mimo kterou se nelze odkazovat.

Insert

```
<div id="contentFrame">
	<p:messages id="messages" autoUpdate="true" closable="true"
		escape="false" />

	<ui:insert                                          name="content">
Put default content here, if any.
	</ui:insert>

	<ui:include src="/WEB-INF/templates/footer.xhtml" />
</div>
```

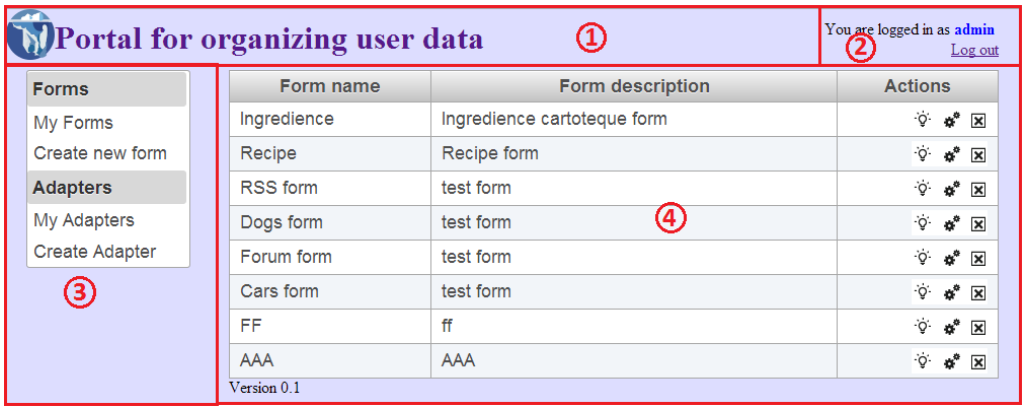
Tag insert slouží k definování části šablony, která je rezervovaná pro vložení obsahu jinou stránkou, která tuto šablonu využívá. Využití je tedy především k definování znovupoužitelné kostry webových stránek.

Define & composition

```
<ui:composition
        template="/WEB-INF/templates/layout.xhtml"
        xmlns="http://www.w3.org/1999/xhtml"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:h="http://java.sun.com/jsf/html"
         xmlns:p="http://primefaces.org/ui">
    <ui:define name="content">
            <p:fieldset legend="Greetings">
                  <h:outputText value="Hello World" />
            </p:fieldset>
     </ui:define>
</ui:composition>
```

Tag define spolu s tagem composition vytváří konkrétní webovou stránku na základě definované šablony a plní ji obsahem. Tag composition definuje šablonu společně se jmennými prostory dalších tagů a define tag definuje obsah jednotlivých částí šablony. Tag composition je kořenový element a tvoří obsah jednoho fyzického souboru (nejčastěji xhtml).

Aplikace POUD je pomocí těchto značek rozvržena do 4 základních sekcí, které jsou zobrazeny na obrázku22.



Obrázek 22: Základní rozvržení webových stránek aplikace POUD

Oblast 1 slouží jako banner aplikace a po kliknutí je uživatel přesměrován na index.xhtml stránku, která je nastavena i jako tzv. welcome-file, tedy stránka, která se zobrazí, pokud uživatel zadá URL aplikace bez části definující aplikačně specifickou URL.

Oblast 2 slouží jednak k přihlášení, pokud není uživatel přihlášen, a jednak k zobrazení jména přihlášeného uživatele s odkazem na odhlášení.

Oblast 3 vykresluje menu. Obsah menu je závislý na rolích přihlášeného uživatele. Blok kódu níže je výňatkem kódu referující menu z obrázku 22. Všimněte si parametru rendered u jednotlivých submenu částech. Tento parametr přijímá hodnoty *true* nebo *false* a na základě nich danou UI komponentu zobrazí nebo ne. Toto je pouze jedna část, která rozhoduje o autorizaci uživatele a která pouze rozhoduje, zda se zobrazí odkaz v podobě menu. Pokud by však uživatel znal URL, na kterou se menu odkazuje, mohl by se na danou webovou stránku dostat pomocí ručně psané URL. Pro zamezení tohoto efektu slouží již popsaný mechanismus v podobě autentizace a autorizace na úrovni URL, který byl již popsán v kapitole věnující se popisu Spring Security modulu.

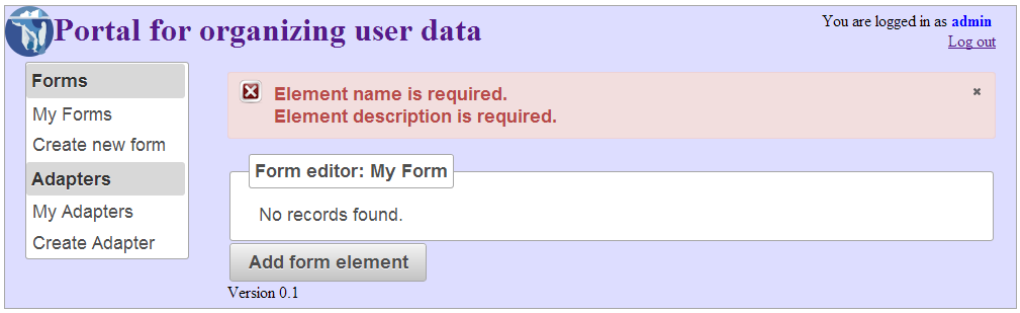
```
<ui:composition>
	<p:menu id="menu">
		<!--Forms-->
		<p:submenu label="#{msg['menu.form']}" styleClass="menuTitle"
	rendered="#{userMB.hasPermission('PERMISSION_MANAGE_OWN_FORMS')}">
			<p:menuitem value="#{msg['menu.form.myForms']}"
				url="/pages/protected/form/browsForms.xhtml" />
			<p:menuitem value="#{msg['menu.form.create']}"
				url="/pages/protected/form/createForm.xhtml" />
		</p:submenu>
		<p:submenu label="#{msg['menu.adapters']}"
styleClass="menuTitle"
	rendered="#{userMB.hasPermission('PERMISSION_MANAGE_ADAPTERS')}">
			<p:menuitem value="#{msg['menu.adapters.brows']}"
				url="/pages/protected/adapter/myAdapters.xhtml" />
			<p:menuitem value="#{msg['menu.adapters.create']}"
			url="/pages/protected/adapter/availableAdapters.xhtml" />
		</p:submenu>
	</p:menu>
</ui:composition>
```

9.6.1 Použité UI komponenty frameworků JSF a PrimeFaces

PrimeFaces framework je bohatá knihovna UI prvku, aplikace POUD však využívá pouze menší podmnožinu. V této podkapitole jmenovitě vyjmenuju a krátce popíši nejfrekventovaněji využívané UI prvky PrimeFaces.

DataTable: Slouží pro vykreslení tabulky. Lze definovat jednotlivé sloupce, jejich obsah či nadpis. Samozřejmostí je podpora řazení, filtrování či stránkování. Jedná se o velmi jednoduchý způsob, jak zobrazit uživateli data ve strukturované formě. PrimeFaces framework také dovoluje definovat tabulku, která má variabilní počet a typ sloupců. Díky tomu může POUD vykreslovat tabulku s obsahem formulářových dat bez ohledu na jejich strukturu. Finální podoba tabulky je generovaná s meta-dat formulářů.

CommandLink/CommandButton: odkaz či tlačítko podporující asynchronní Ajaxové volání. Zpravidla se využívá tam, kde chceme změnit určitou část aplikačního modelu a aktualizovat obsah webové stránky bez nutnosti načíst ji celou. Je to například vhodné pro vytváření struktury formulářů, kdy uživatel definuje, jaké elementy obsahuje a po přidání elementu ihned vidí výsledek, aniž by zaznamenal jakékoliv načtení stránky. Tento přístup má ovšem i své nevýhody. Jedná se především o nevýhody spojené s asynchronní podstatou řešení. V případě, že akce uživatele skončí chybou, výjimka na straně serveru není propagována klientovi (webovému prohlížeči) v podobě chybového http status kódu (respektive je ale pouze pro asynchronní dotaz, stránka se tedy může stále jevit, že je vše v pořádku). Řešením v podání JSF je UI prvek jménem messages, který umí z FacesContext objektu zjistit, zda došlo na straně serveru k chybě a pokud ano, pak umí zobrazit chybovou hlášku uživateli.



Obrázek 23: Ilustrace funkce UI prvku message (červená oblast v horní části)

Obrázek 23 demonstruje zachycení výjimky, která nastala při validování vstupu od uživatele, který se snažil přidat formulářový element (asynchronní volání) bez vyplnění jména elementu a jeho popisu.

Dialog: PrimeFaces podporuje dialogy, jejichž obsah je psaný v HTML kódu, respektive pomoci tagu JSF a PrimeFaces. Díky Javascriptu, kterého je v JSF a PrimeFaces hojně využíváno, je možné definovat pokročilé funkce, které známe z desktopových aplikací. Mluvím zde především o modálních dialozích, o funkcích umožňujících přesunování či změny velikosti dialogu. Podpora tlačítka X pro uzavření dialogu, či definování klávesové zkratky pro tento účel. Dialog je dobrá volba pro nabídnutí uživateli UI prostoru, ve kterém může blíže specifikovat určité akce či vstupy, kterými nechceme zaplnit místo na hlavní stránce.

OutputText: Užitečný tag, který se doporučuje používat pro všechny textové výstupy. Jeho největší předností je ochrana proti tzv. Cross-sitescripting útoku.

9.7 Lokalizace

Lokalizace je v POUD aplikaci řešena také pomocí JSF frameworku. Konfigurační soubor WEB-INF/faces-config.xml definuje soubor s lokalizačními soubory. Blok kódu níže demonstruje ukázku lokalizačního souboru pro anglický jazyk.

```
application.name = Portal for organizing user data

index.title = Hello World

loginForm.title = Log in
loginForm.username = User name
loginForm.password = Password
error.login.badCredentials = Bad credentials!
button.login  = Log in
```

Struktura lokalizačních souborů je v podobě páru klíč = hodnota. V kódu stránky se pak na tyto hodnoty odkazujeme jednoduše přes výraz *#{msg['klíč']}*.

```
<p:column
headerText="#{msg['form.metamodel.overview.header.elementLabel']}">
      <h:outputText value="#{formElement.label}" />
</p:column>
```

Ukázka kódu výše demonstruje lokalizaci záhlaví jednoho sloupce tabulky. Aplikace POUD v aktuální implementaci nepodporuje více lokalizací. Nicméně mechanismus je připraven pro snadné přidání dalších lokalizačních souborů.

Rozhodnutí, která lokalizace se má použít je v POUD definována pomocí defaultní hodnoty. Tedy nastavené na „en“. Tato hodnota je držena ve FacesContext objektu a její změnou přinutíme JSF aby byl načten jiný patřičný lokalizační soubor. Lze to učinit například pomocí managed bean objektu, který zavolá kód podobný tomuto:

```
FacesContext.getCurrentInstance().getViewRoot()

.setLocale((Locale)entry.getValue());
```

kde entry je typu java.util.Locale.

9.8 Konfigurační soubory a Spring profiles

POUD projekt je nastaven tak, aby bylo možné snadno měnit sady konfiguračních souborů na základě prostředí, ve kterém je aplikace nasazena. Vývojáři mají tak svou sadu konfiguračních souborů, které používají pro nastavení aplikace na svých lokálních vývojových stanicích a stejně tak testovací či produkční prostředí mají také své sady konfiguračních souborů. Systém je nastaven tak, aby při přechodu mezi prostředími nebylo nutné nastavit žádný přepínač, který by určil, jaká sada konfiguračních souborů se má použít. Vše se provede automaticky a správná sada konfiguračních souborů je odvozena od prostředí, ve kterém aplikace běží.

Konfigurační soubory aplikace POUD lze rozdělit na dvě části:

- Konfigurační soubory interní, které slouží k nastavení frameworků, použitých v aplikaci (Spring security, transakční manager, JSF, PrimeFaces atd)
- Konfigurační soubory externí, které udržují konfigurace jako je nastavení spojení s databází, či dodatečná nastavení zabezpečení

Konfigurační soubory interní se nacházejí vždy na classpath aplikace a nelze k nim přistupovat z vnějšku distribučního balíčku (war, jar, ear, atd.). Zatímco konfigurační soubory externí mohou být přístupné buďto na classpath, v tom případě pro ně platí stejné omezení z pohledu přístupu jako pro interní konfigurační soubory, anebo mohou být přístupné externě, tedy mimo distribučního balíčku. Tento přístup je preferovaný zejména z následujících důvodů:

- Konfiguraci lze měnit bez nutnosti re-kompilace aplikace.
- Za běhu aplikace lze zjistit, jak je aplikace nakonfigurována. Je tedy například možné se podívat, k jaké databázi je aplikace připojená.
- Po změně konfigurace stačí aplikaci restartovat.
- Lze jednoduše vyměnit starou konfiguraci za novou

Aplikace POUD spravuje tyto konfigurační soubory:

Interní konfigurační soubory:

- Web.xml – konfigurační soubor popisující webovou Java aplikaci. Instruuje webový kontejner, jaké třídy se mají načíst či jaké parametry se mají nastavit.
- Pom.xml – spravuje informace a nastavení projektu, na základě nichž je Maven schopen projekt zkompilovat a sestavit.
- Poud-core.xml – základní nastavení Spring frameworku. Obsahuje informace o balíčcích, ve kterých se nachází třídy, které mají být registrovány jako komponenty, či servisní bean objekty.
- Pond-data.xml – konfigurace vztažené k databázi (neobsahuje konkrétní data ohledně spojení s databází, ty jsou poskytnuty skrze place-holdery které čtou konfigurace z externích konfiguračních souborů). Je zde nastavení transakčního manageru, data source objektu a session objektu.
- Poud-security.xml – nastavení autorizace a autentizace aplikace POUD
- Faces-config.xml – nastavení frameworku JSF a PrimeFaces
- Poud.xml – konfigurace Assembly pluginu (pro framework Maven), který generuje distribuční balíčky aplikace POUD
- Poud.properties – data-holder konfigurační soubor, který poskytuje konkrétní konfigurační hodnot interním konfiguračním souborům, které jsou stejné pro všechny běhové prostředí. Příkladem může být textový řetězec password-salt, který se používá pro zvýšení bezpečností hashovacích funkcí pro práci s hesly.

Externí konfigurační soubory jsou rozdělené na konfigurace logování a na data-holdery, které poskytují konkrétní konfigurační hodnoty souborům interním. Tyto soubory se dále dělí podle běhového prostředí, pro které jsou určeny. Zde je zavedená jmenná konvence, která musí být dodržena, aby bylo možné automatické rozpoznávání konfiguračních souborů na základě běhového prostředí.

Externí konfigurační soubory logování mají následující jmennou konvenci:

```
log4j-poud-{env}.properties
```

Kde {env} je název běhového prostředí. POUD rozlišuje následující běhová prostředí:

- live: produkční běhové prostředí
- test: testovací prostředí
- develop: vývojové prostředí

Na konfigurace logování se odkazuje konfigurační soubor web.xml:

```
<context-param>
	<param-name>log4jConfigLocation</param-name>
	<param-value>
		${poud.config.location}log4j-poud-
		${spring.profiles.active}.properties
	</param-value>
</context-param>
```

Kde {poud.config.location} je hodnota, která určí, zda se konfigurační soubor má načíst s classpath, nebo z jiného zdroje (například ze souborového systému). {spring.profiles.active} pak určí, o jaké běhové prostředí se jedná a podle toho načte patřičný soubor.

Externí konfigurační prostředí pak mají podobnou jmennou konvenci:

```
poud_{env}.properties
```

Kde {env} je název běhového prostředí. Konfigurační soubory jsou odkazované z interního konfiguračního souboru poud-core.xml:

```
<beans profile="develop">
  <bean
       class="org.springframework.beans.factory.config.PropertyPlaceh
       olderConfigurer">
    <property name="locations">
      <list>
        <value>${poud.config.location}poud.properties</value>
        <value>${poud.config.location}poud_develop.properties</value>
      </list>
    </property>
    <property                          name="systemPropertiesModeName"
value="SYSTEM_PROPERTIES_MODE_OVERRIDE" />
  </bean>
</beans>
```

Parametr profile definuje sadu Springovských bean objektů, které se mají načíst na základě běhového prostředí. Ukázka kódu výše demonstruje načtení externích konfiguračních souborů pro prostředí develop, čili vývojové prostředí. {poud.config.location} má stejný význam, jako u konfiguračních soborů logování, tedy určit místo, ze kterého se konfigurační soubory načtou.

Výše jsem popsal, jakým způsobem se provádí konfigurace aplikace POUD a také jsem vysvětlil, že se konfigurační soubory rozlišují na externí a interní, kde externí se dále dělí na konfigurační soubory logování a tzv. data-holdery. Externí soubory přicházejí v několika sadách, každá pro jedno z podporovaných vývojových prostředí. Co jsem však nepopsal je, kde se vezmou hodnoty {poud.config.location} a {spring.profiles.active}. Aplikace POUD tyto hodnoty načítá z JVM (Java Virtual Machine) parametrů. Tyto parametry musí být nastaveny při spuštění běhového prostředí. V případě Tomcat serveru, který byl využit pro vývoj aplikace POUD má upravený spouštěcí skript tak, aby při startu tyto parametry nastavil:

```
set JAVA_OPTS=-Dspring.profiles.active=develop -Dpoud.config.location=file:C:\poud\conf/
```

Z ukázky kódu výše je patrné, že běhové prostředí je nastavené jako vývojové a konfigurační soubory se nacházejí ve složce C:\poud\conf. Toto jsou kompletní informace, které je nutné nastavit, aby aplikace POUD byla schopna určit sadu konfiguračních souborů.

Kapitola 10

Testování

Testování je nedílnou součástí vývoje každého softwaru. Historie nám už mnohokrát ukázala, že podceněním testování si z dlouhodobého hlediska znesnadníme budoucí vývoj a údržbu aplikace. Bez řádného testování se zpravidla zvýší cena budoucího vývoje a údržby. Proto je důležité se testování věnovat na dostatečné úrovni. Existuje mnoho druhů testování, které se provádí v různých fázích vývoje a s různou periodou opakování. Aplikace POUD je však podrobena pouze testování na úrovni unit testů, které zastupují testování typu black box/white box a regresní testování.

Testy lze spustit kdykoliv pomocí vývojového prostředí, nebo pomocí Maven. Unit testy se také automaticky spouštějí při kompilaci a při sestavování distribučního balíčku.

Aplikace POUD a její testovací část/konfigurace podporují následující způsoby testování:

- Mockování: objekt, který je podroben testování může některé nebo všechny svoje části nahradit mock objekty, které pak simulují reálný běh aplikace.
- Simulace reálného běhu: díky podpory ze strany Spring frameworku, je možné spustit a inicializovat aplikační kontext i na úrovni junit testů. Testovací třídy můžou pak využít dependency injection funkcionality frameworku Spring, a tím si opatřit jednotlivé části/instance tříd aplikace, aniž by je jednotlivé testy museli samy inicializovat či mockovat.
- In-memory databáze: POUD je nakonfigurován tak, aby bylo možné pro účely testování kompletně nasimulovat reálné běhové prostředí, včetně databáze. Databáze je v tomto případě kompletně udržovaná v paměti, což přináší výhodu zejména v tom, že tester nemusí poskytnout skutečné připojení do databáze. Vše je vyřešeno virtuálně v operační paměti počítače. Takováto databáze se však chová jako skutečná. Díky tomuto řešení není dále třeba vytvářet mock objekty Dao vrstvy či aplikační vrstvy. Konfigurace je však v tomto případě značně složitější, než v případě testování pomocí mock objektů. Na druhé straně však získáme testovací prostředí velmi blízkému produkčnímu prostředí. Psaní testů se tím také do značné míry usnadní, protože se nemusíme zabývat ruční inicializací jednotlivých testovaných částí aplikace.

V aktuální implementaci aplikace POUD nekomunikuje s žádnými externími službami, a proto nejsou součástí vývoje žádné integrační testy. Naproti tomu je aplikace v průběhu vývoje testovaná manuálně a to vždy, když se vyvine nebo redaktoruje stávající část kódu. Díky pluginu Sysdeo-tomcat (popsaný v kapitole věnující se použitým vývojovým nástrojům), je velmi snadné provést změnu v kódu aplikace a ihned vidět její dopad. Vývoj tak není brzděn opakující se kompilací, sestavováním distribučního balíčku či nasazování na server. Vše se děje automaticky v řádu sekund.

Kapitola 11

Budoucí práce

V této kapitole se budu věnovat dílčím částím aplikace POUD, které jsou do určité míry rozpracované, a popíši, jakým způsobem lze tyto části dokončit. Budu se zde také zabývat funkcemi, které nejsou rozdělané a to ani ve stádiu analýzy, či návrhu, ale které svojí podstatou spadají do konceptu aplikace POUD.

11.1 Reference

Reference mezi elementy představují základní mechanismus, pro vytváření komplexních struktur, které se nedefinují pouze na úrovni elementů v rámci jednoho formuláře, ale definují na úrovni formulářů samotných. Databázové schéma společně s korespondujícími doménovými entitami jsou na koncept referencí připravené, a však aplikační vrstva s referencemi nepracuje. Prezentační vrstva je jen z části připravená, ale z důvodů absence podpory ze strany aplikační vrstvy jsou UI prvky spjaté s konceptem referencí „zakomentované“.

11.2 Adaptéry

Koncept adaptérů je analyzován a na úrovni databáze a doménových objektů připraven. Presentační vrstva dovoluje vytvářet mapování mezi formuláři a adaptéry, nicméně aplikační vrstva má velmi malou část implementace připravenou k použití.

Druhou částí konceptu adaptérů, která není zcela implementována, jsou konkrétní implementace adaptérů. Aplikace POUD má připravený návrh adaptéru pro čtení RSS feedů a architektonicky je POUD připraven, pro operování s adaptéry. Nicméně kód, který by vykonal čtení RSS zdrojů, není momentálně dostupný.

Třetí částí konceptu adaptérů jsou časovače. I zde je připravena architektonická stránka věci společně s databází a doménovými objekty, prezentační vrstva má také pro časovače podporu, nicméně aplikační vrstva není plně funkční. Chybí zde především ukládání a načítání časovačů do databáze a také zde chybí mechanismus, který by při startu aplikace načetl všechny aktivované časovače a automaticky je spustil.

11.3 Správa uživatelských účtů

Aplikace POUD nepodporuje zakládání uživatelských účtů. Aktuální implementace neobsahuje formulář pro vytvoření uživatele, a proto jedinou možnou cestou, jak uživatele do aplikace přidat je ruční zásah do databáze. Tento nedostatek nevyžaduje příliš úsilí pro jeho nápravu, nicméně jedná se o zásadní překážku k reálnému využití aplikace POUD.

11.4 API

Aplikace POUD je dobrým kandidátem pro zavedení programového rozhraní, pomocí kterého by mohli ostatní systémy využívat funkce a data aplikace POUD. API je pouze ve fázi analýzy a ze strany návrhu a ani implementace nejsou doposud učiněny žádné kroky.

11.5 Optimalizace databáze

POUD ukládá veškerá data do jednoho schéma v databázi. Minimum, které je nutné provést je rozdělit schéma na aplikační data a na uživatelská data, respektive na data formulářů. Protože formulářový engine vytváří mnoho tabulek na základě nově vytvořených formulářů, stává se aktuální schéma databáze špatně udržovatelné.

11.6 Fultextové vyhledávání, filtrování, stránkování

S automatickým plněním dat pomocí adaptérů bude objem dat velmi rychle stoupat. Bude proto nutné představit mechanismy, které usnadní orientaci v uživatelských datech. Fultextové vyhledávání je jednou z metodik, které usnadní uživatelům přistupovat k datům, které potřebují. Orientace v datech může být dále usnadněna pomocí filtrování dat, tedy mechanismu, který specifikuje, jaké typy dat nebo jaké hodnoty se mají ve vyhledávání zobrazit. Protože objemy dat budou velmi rychle růst, je zapotřebí také optimalizovat načítání dat z databáze. Poměrně jednoduchým řešením s dobrým poměrem cena/přidaná hodnota je technika zvaná stránkování. Tedy načítání dat po stránkách, které obsahují počet záznamů, který je omezen pevně danou horní hranicí.

11.7 Uživatelsky definovatelné UI rozhraní pro formuláře

Aktuální implementace POUD aplikace poskytuje formulářové záznamy v podobě tabulky, která zobrazuje pro každý formulářový element jeden sloupec s patřičnými daty. V případě strukturálně objemných formulářů není schopno UI správně, či přehledně zobrazit formulářové záznamy. Z tohoto důvodu je zapotřebí umožnit uživateli, aby si nadefinoval, jaká data chce v přehledu formulářových záznamů zobrazit. V praxi by to mohlo vypadat tak, že si uživatel zvolí množinu elementů, které mají být v náhledu viditelné a zbylé elementy by byli přístupné pouze na webové stránce s detailem formulářového záznamu.

11.8 Import/export formulářů

Funkce migrace meta-dat formulářů by umožnila uživatelům předávat si jednotlivá schémata formulářů mezi sebou. V ideálním případě by funkce nahrála do nějakého veřejně přístupného repositáře formulářů daný formulář, respektive jeho strukturu, kterou by si uživatelé mohli importovat do svého profilu. Nový uživatel by tak nemusel vytvářet formuláře manuálně, ale mohl by si vybrat formuláře, které jsou již předpřipravené k použití.

11.9 Datové konvertory

Data ukládaná uživateli, budou mít různorodý charakter. Každý uživatel bude mít své vlastní nároky na zpracování dat (formátování data či čísel). Formuláře spolu s adaptéry umožní pouze data načíst z určeného zdroje a uložit je do databáze. Aktuální implementace aplikace POUD neposkytuje žádnou funkcionalitu, která by umožnila data před uložením do databáze upravit. Zavedením mechanismu, který by u definovaných částí formulářů prováděl různě operace datových konverzí, by dovolil uživateli ukládat data v požadovaném formátu, anebo pouze části dat, o která má uživatel skutečně zájem.

11.10 Business rules

Koncept business rules by spočíval v představení skriptovacího jazyka, který by dovolil uživateli vytvářet skripty či makra, která by reagovala na určité akce spojené s formuláři. Například by bylo možné vytvořit skript, který by v případě vytvoření formulářového záznamu zaslal uživateli email. Business rules nemusí mít nutně formu skriptovacího jazyka v pravém slova smyslu, může se jednat o sadu předem definovaných akcí, ze kterých si uživatel může vybrat a která by byla jednoduše nastavitelná skrze GUI.

Kapitola 12

Závěr

Tato diplomová práce mě i přes mé přesvědčení, že téma není zcela neznámé a mé zkušenosti s vývojem webových aplikací neřadím mezi začátečnické, přesvědčila o tom, že i zdánlivě jednoduchý projekt může mít své zapeklitosti a pracnost při řešení dílčích problémů se může velmi snadno minout se svým odhadem. Z počátku jsem se snažil poměrně velkou mírou věnovat analýze a návrhu aplikace POUD, což na jednu stranu přineslo své ovoce v podobě pochopení rozsahu projektu, ale na druhou stranu jsem udělal některá rozhodnutí, které jsem v pozdějších fázích vývoje musel přehodnotit.

Na většině projektů jsem se setkal s vlivy, které nepřímo, nebo vůbec nesouvisí s vývojem dané aplikace, ale jejich dopad na výslednou podobu projektu byl zásadní. Bohužel ani projekt POUD nebyl před účinky těchto vlivů uchráněn. Před započetím analýzy jsem špatně odhadl pracnost projektu a také mojí časovou dostupnost. Díky tomu jsem pod časovým tlakem dělal chyby, které vedly k nesplnění zadání této diplomové práce. Aplikace POUD je sice ve spustitelné formě a poskytuje některé základní funkce, ale její zamýšlený přínos nemůže být využit, protože pro bezproblémový chod je zapotřebí, aby byly implementovány všechny části aplikace dle zadání. Aplikace POUD tak není vhodná pro produkční provoz a je zapotřebí dodělat definované minimum dílčích části, zejména pak správu uživatelských účtů, adaptéry a API rozhraní.

I přes neúspěch mi tento projekt přinesl zkušenosti, které mohu v budoucnu využít. Projekt POUD mi ukázal, že vyvinout projekt od „ničeho“ do zdárného konce, znamená pro jednoho vývojáře velkou odpovědnost. Projekt musí být v první řadě funkční a kvalitní. Pro ověření funkčnosti dobře poslouží automatizované a manuální testy, nicméně pojem kvalitní software nelze moc dobře uchopit. Při vývoji aplikace POUD jsem se snažil dbát na logické uspořádání jednotlivých částí projektu a jeho konfigurací. Brzo jsem zjistil, že navrhnout architekturu uspořádání a infrastrukturu i poměrně jednoduchého projektu se vším, co k tomu patří, je samo o sobě velmi komplexní problematika.

Neméně důležitý poznatek, který jsem si při práci na projektu POUD odnesl je, že bez řádné podpory pro jednotlivé technologie, hrozí riziko, že se vývoj dostane do slepé uličky nebo se může zastavit na technickém problému, který lze sice vyřešit například přidáním jednoho řádku do konfiguračního souboru, ale nalezení řešení může zabrat i několik dnů pátrání v dokumentaci, hledání na internetu či testování prototypů.

O výsledku práce nemohu říct, že by byl pro mne absolutním zklamáním. Získal jsem nové zkušenosti jak v oblasti technologické, tak i v oblasti řízení vývoje projektu. Díky problémům, kterým jsem musel čelit, jsem dostal větší nadhled nad řízením vývoje a vývojem softwaru obecně a do budoucna se můžu na problematické části více zaměřit.

Za hlavní přednosti výstupu této práce považuji návrh struktury projektu, organizaci konfiguračních souborů, flexibilitu dovolující snadné rozšíření o nové funkce a to na všech vrstvách aplikace a v neposlední řadě zvolené technologie, které se řídí obecně uznávanými standardy a malou měrou diktují architekturu aplikace. Aplikace tak má robustní základ pro další vývoj.

Literatura

[1] IFTTT: About IFTTT. [online]. [cit. 2014-04-11]. Dostupné z: https://ifttt.com/wtf

[2] Evernote. [online]. [cit. 2014-04-11]. Dostupné z: https://evernote.com/

[3] SCHWABER, Ken a Jeff SUTHERLAND. The Scrum Guide: The Definitive Guide to Scrum. [online]. s. 16 [cit. 2014-05-12]. Dostupné z: https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf#zoom=100

[4] ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2007, 567 s. Knihovna programátora (Grada). ISBN 978-80-251-1503-9.

[5] UTF-8: RFC. [online]. [cit. 2014-04-11]. Dostupné z: http://tools.ietf.org/html/rfc3629

[6] Database Language SQL: Specification. [online]. [cit. 2014-04-11]. Dostupné z:http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[7] ServiceNow: The Enterprise IT Cloud company. [online]. [cit. 2014-04-11]. Dostupné z: https://www.servicenow.com/

[8] MARQUARDT, Klaus. Patterns for Plug-Ins. 1999, s. 37. Dostupné z:http://www.kmarquardt.de/papers/EuroPLoP1999.PlugInPatterns.pdf

[9] Cron: How to. [online]. [cit. 2014-04-11]. Dostupné z: https://help.ubuntu.com/community/CronHowto

[10] Agility - webMethods. [online]. [cit. 2014-05-12]. Dostupné z:http://www.softwareag.com/Corporate/products/wm/default.asp

[11] GIT Documentation: Reference manual. [online]. [cit. 2014-05-12]. Dostupné z: http://git-scm.com/docs

[12] Maven Documentation: Documentation. [online]. [cit. 2014-05-12]. Dostupné z: http://maven.apache.org/guides/

[13] Maven Naming Conventions: Guide to naming conventions on groupId, artifactId and version. [online]. [cit. 2014-05-12]. Dostupné z: http://maven.apache.org/guides/mini/guide-naming-conventions.html

[14] Eclipse Project. [online]. [cit. 2014-05-12]. Dostupné z: http://www.eclipse.org/eclipse/

[15] WILLIAMS, Hugh E a Ila NEUSTADT. *PHP a MySQL: vytváříme webové databázové aplikace : podrobný průvodce tvůrce WWW stránek*. Vyd. 1. Překlad Bogdan Kiszka. Praha: Computer Press, 2002, xx, 530 s. Knihovna programátora (Grada). ISBN 80-722-6760-4.

[16] MySQL Documentation: Reference manual. [online]. [cit. 2014-04-11]. Dostupné z:http://dev.mysql.com/doc/refman/5.6/en/index.html

[17] Apache Tomcat 7: Documentation Index. [online]. [cit. 2014-05-12]. Dostupné z: http://tomcat.apache.org/tomcat-7.0-doc/

[18] Spring framework 3: Reference Documentation. [online]. [cit. 2014-05-12]. Dostupné z:http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/

[19] Spring Security: Reference Documenatatio. [online]. [cit. 2014-05-12]. Dostupné z: http://docs.spring.io/spring-security/site/docs/3.0.x/reference/springsecurity.html

[20] Spring ORM: Reference Documenatatio. [online]. [cit. 2014-05-12]. Dostupné z: http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/orm.html

[21] Hibernate Documentation: Reference Documentation. [online]. [cit. 2014-04-11]. Dostupné z:http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/

[22] PrimeFaces: Documentation. [online]. [cit. 2014-05-12]. Dostupné z:http://primefaces.googlecode.com/files/primefaces_users_guide_4_0_edtn2.pdf

[23] Web Applications Basics. [online]. [cit. 2014-05-12]. Dostupné z:http://docs.oracle.com/cd/E13222_01/wls/docs81/webapp/basics.html

[24] BALABAN, Mira. Layerd Architecture: Documentation. [online]. 2004 [cit. 2014-05-12]. Dostupné z:http://www.cs.bgu.ac.il/~oosd051/uploads/classes/4-Layered-architectures.pdf

[25] Java SE Documentation: Setting the Class Path. [online]. [cit. 2014-05-12]. Dostupné z:http://docs.oracle.com/javase/8/docs/technotes/tools/unix/classpath.html

Příloha A

A. Příručka nasazení aplikace POUD

Tato příručka popisuje krok po kroku, jak lze aplikaci zkompilovat, nastavit a nasadit na běhové prostředí Tomcat. Na přiloženém CD jsou veškeré potřebné nástroje, pomocí kterých lze celý proces uskutečnit. Taktéž se na přiloženém CD nachází kompletně nastavené běhové prostředí s nasazenou aplikací. Správnost nastavení běhového prostředí je testováno v prostředí Windows 7 x64. Pokud se nechcete zabývat kompilací, konfigurací, či nasazením aplikace, přejděte rovnou k části A.2 konkrétně k bodům 3 až 5.3 (instalace MySql databáze a konfigurace spojení s databází ze strany POUD aplikace) a dále k části A.3.

A. 1 Kompilace

Pro správný běh kompilace je zapotřebí mít nainstalovaný a nakonfigurovaný Maven alespoň verze 2. Na přiloženém CD ve složce /Test-runtime-environment/Maven 3 je připravený Maven, který je třeba nakonfigurovat dle prostředí, ve kterém se kompilace provádí. Nutné kroky pro nastavení Maven jsou následující:

1. Zkopírujte soubor CD/Test-runtime-environment/Maven 3/settings.xml do adresáře ${user.home}/.m2/.
2. Vytvořte složku pro Maven repozitář ${user.home}/.m2/repository
3. Změňte konfiguraci zkopírovaného souboru settings.xml tak, aby ukazoval na právě vytvořený maven repositář.
4. Maven je nyní připraven k použití. Příkaz mvn je ve složce CD/Test-runtime-environment/Maven 3/apache-maven-3.2.1/bin

Kompilace

1. Zkopírujte složku CD/Poud na libovolné místo na souborovém systému vašeho počítače, kde máte právo zápisu
2. V adresáři {base}/Poud/ vykonejte příkaz ’mvn assembly:assembly’. Alternativa k tomuto bodu je po provedení konfigurace Maven spustit dávkový soubor {base}/Poud/compile.bat.
3. Vytvoří se adresář *target*, který obsahuje distribuční balíček *poud-0.1.zip*.

A. 2 Deployment

Složka CD/Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53 obsahuje připravený webový kontejner Apache Tomcat ve verzi 7. Nicméně před tím, než je možné aplikaci nasadit, je třeba připravit databázi, nahrát konfigurační soubory POUD aplikace a nastavit Tomcat tak,

aby tyto konfigurační soubory při startu načetl. Složka CD/Test-runtime-environment/MySql obsahuje instalační balíček databáze MySql.

1. Rozbalte distribuční balíček poud-0.1.zip. Obsahuje binární soubor aplikace (binaries/poud.war) a konfigurační soubory (configuration/poud).
2. Zkopírujte konfigurační soubory do libovolné složky. Pro účely tohoto návodu nazvu vámi vytvořenou složku pro konfigurační soubory {base}/poud/conf.
3. Nainstalujte MySql databázi
4. Vytvořte schéma *poud*
5. Upravte konfiguraci POUD {base}/poud/conf/poud_develop.properties

5.2 Upravte řádek *poud.jdbc.url = jdbc:mysql://localhost:3306/poud*, kde změňte *localhost:3306/poud* tak, aby se konfigurace odkazovala na vámi vytvořenou databázi a schéma

5.3 Upravte řádky *poud.jdbc.username = root* a *poud.jdbc.password = password* tak, aby odpovídaly přihlašovacím údajům k databázi

6. Nyní je POUD aplikace nakonfigurován

Nyní připravíme server Tomcat

1. Pokud jste to již neudělali, zkopírujte si složku CD/ Test-runtime-environment na váš souborový systém do vámi zvolené složky {base}/ Test-runtime-environment
2. Ujistěte se, že jsou složky {base}/ Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53/webapps a {base}/ Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53/works prázdné. Pokud nejsou, vymažte veškerý obsah.
3. Z rozbaleného distribučního balíčku poud-0.1.zip zkopírujte soubor binaries/poud.war do složky {base}/ Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53/webapps
4. Editujte soubor {base}/ Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53/bin/POUD – startup.bat

4.2 Změňte řádek *set JAVA_OPTS=-Dspring.profiles.active=develop -Dpoud.config.location=file:..\..\poud-conf/* tak, aby část *poud.config.location=file:{path to poud config files}/* konkrétně část *{path to poud config files}* odkazovala na složku {base}/poud/conf. Konfigurace bude tedy vypadat následovně: *set JAVA_OPTS=-Dspring.profiles.active=develop -Dpoud.config.location=file:*{base}/poud/conf /

Nyní máme zkompilovanou aplikaci POUD, nainstalovanou databázi, nakonfigurovaný server Tomcat a nasazenou aplikaci. Pokud vše proběhlo bez problémů, je možné přistoupit ke spuštění aplikace.

A. 3 Spuštění aplikace POUD

Aplikace je přístupná na URL {localhost|[hosting mashine]}:8080/poud

Aplikace POUD se spustí dávkovým souborem {base}/ Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53/bin/POUD – startup.bat. Konfigurační soubory POUD jsou nastavené tak, aby při spuštění vytvořili základní tabulky v databázi se základními daty. Pro druhé a další

spuštění (restartování) aplikace POUD je doporučeno změnit konfiguraci souboru {base}/poud/conf/poud_develop.properties tak, že zakomentujete řádky:

hibernate.hbm2ddl.auto = create-drop

hibernate.populate = true

na

#hibernate.hbm2ddl.auto = create-drop

#hibernate.populate = true

Aplikace POUD se stopne společně se serverem Tomcat a to dávkovým souborem {base}/ Test-runtime-environment/Tomcat 7/apache-tomcat-7.0.53/bin/shutdown.bat.

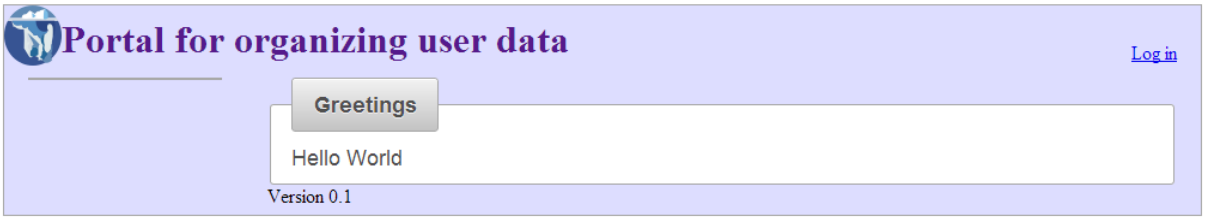
Příloha B

B. Uživatelská příručka

V příloze A je podrobný návod na zprovoznění aplikace POUD. V této příloze popíši základy práce s aplikací POUD. Doporučený webový prohlížeč je Google Chrome. Mozilla Firefox by měl také fungovat bez problému, nicméně Internet Explorer nemusí být plně kompatibilní, kvůli jeho tzv. compatibility mode, který může používat pro renderování webových stránek starší engine používaný za dob Internet Exploreru 6.

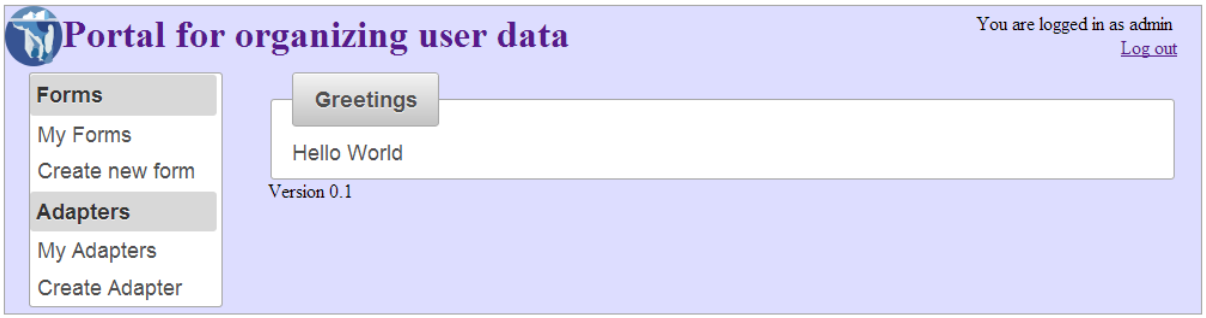
B.1 Přihlášení

Uvítací stránku aplikace naleznete na URL {localhost|[hosting mashine]}:8080/poud:



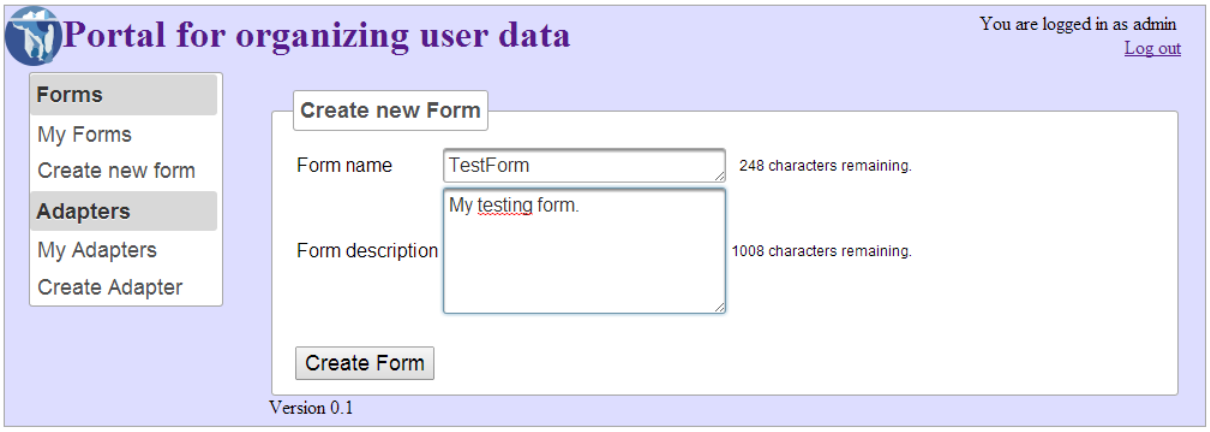
Přihlášení probíhá pomocí odkazu v pravém horním rohu *Log in*. Pro přihlášení jako administrátor, vyplntě jméno „admin“ a heslo „password“.

Zobrazí se stránka s dostupnými položkami v menu po levé straně.

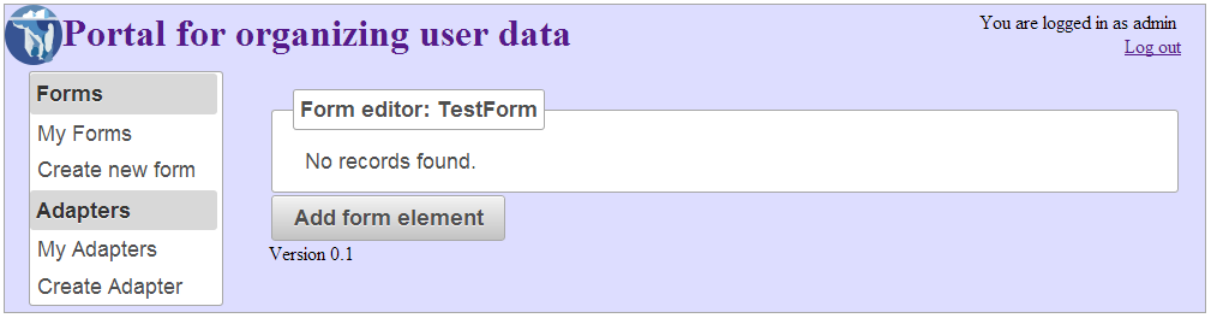


B.2 Vytvoření formuláře

Z menu po levé straně zvolte „Create new form“. Zobrazí se stránka pro vytvoření formuláře, ve které vyplníte jméno formuláře a jeho popis.



Po Vytvoření formuláře vás aplikace odkáže na stránku, ve které můžete definovat strukturu formuláře.



Přidejte následující elementy:

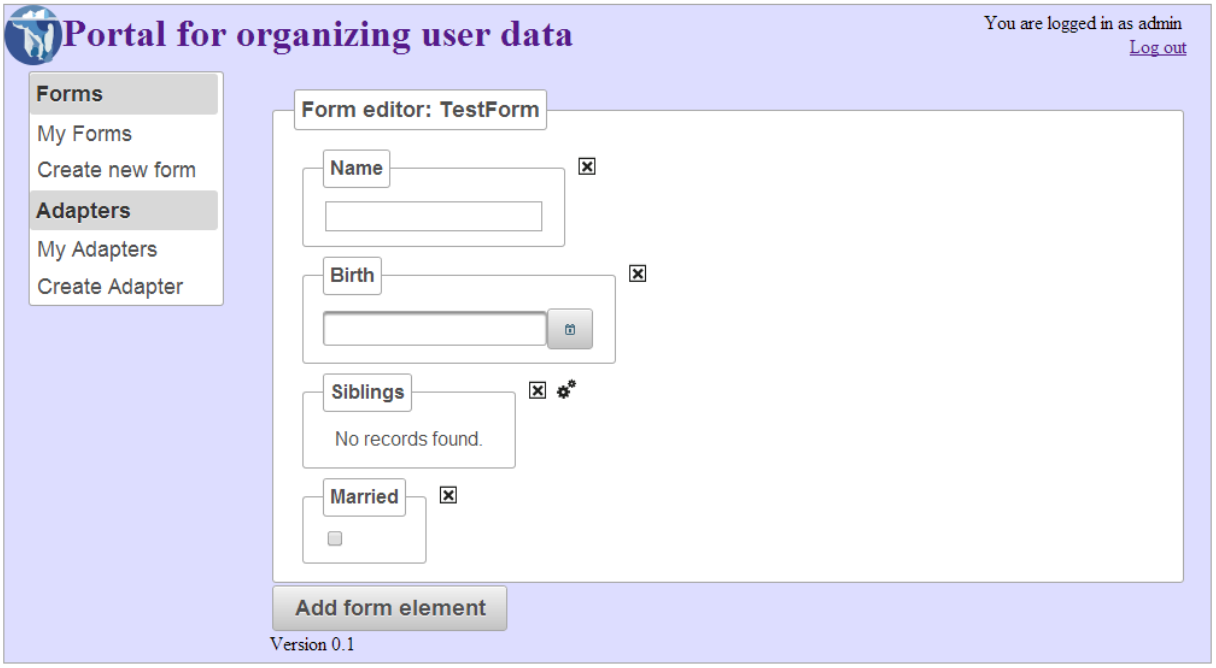
1. Typ: String
   Label: Name
   Description: name
   Position: 1

2. Typ: Date
   Label: Birth
   Description: Date of birth
   Position: 2

3. Typ: Composite
   Label: Siblings
   Description: List of Siblings
   Position: 3

4. Typ: Boolean
   Label: Married
   Description: Check it, if married
   Position: 4

Formulář by měl vypadat následovně:

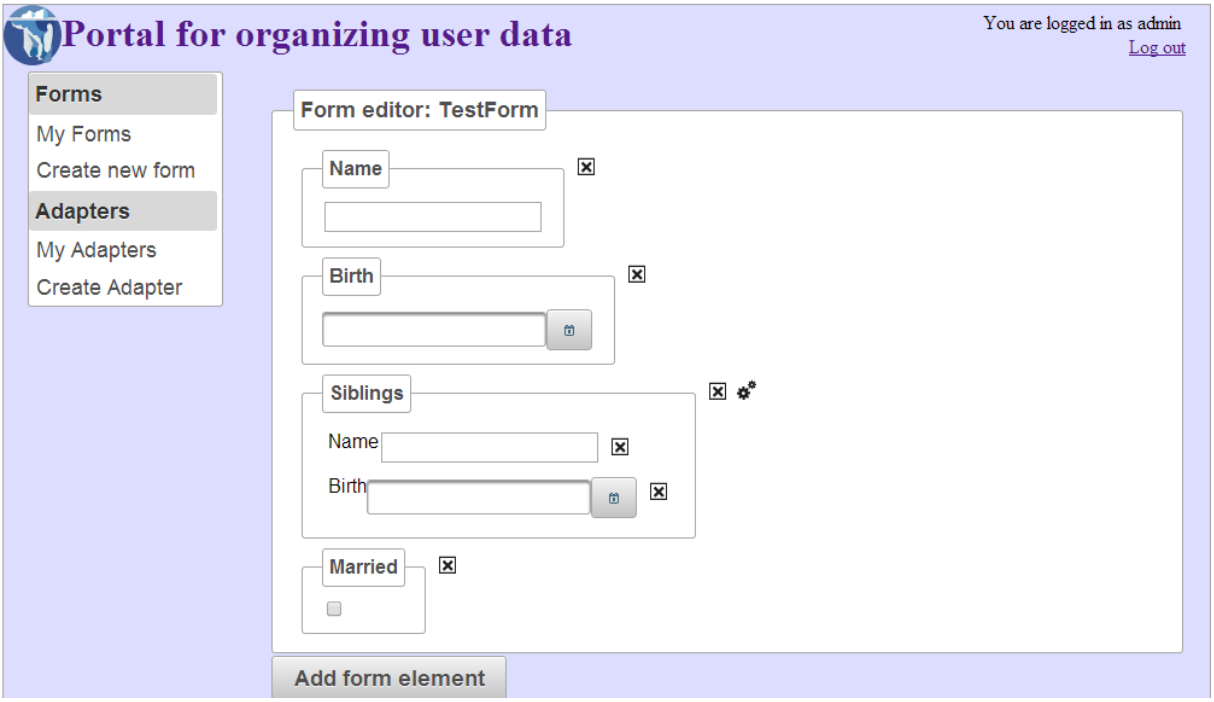


U elementu Siblings definujte pomocí ikony ozubených koleček následující elementy:

1. Typ: String
   Label: Name
   Description: name
   Position: 1

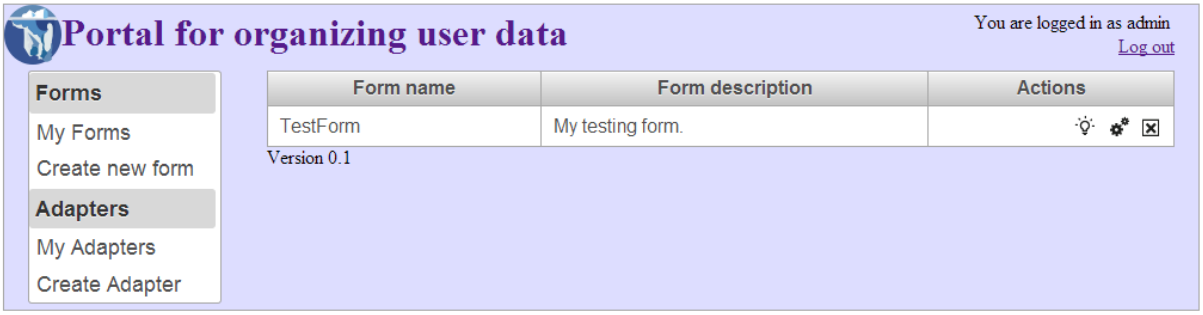
2. Typ: Date
   Label: Birth
   Description: Date of sibling birth
   Position: 2

Nyní máme vytvořenou strukturu formuláře pro vkládání dat o lidech a jejich potomcích.



Přejděte na přehled vašich formulářů přes odkaz v menu po levé straně „My Forms“. Zde můžete formuláře dále editovat, mazat, či vytvářet formulářové záznamy.

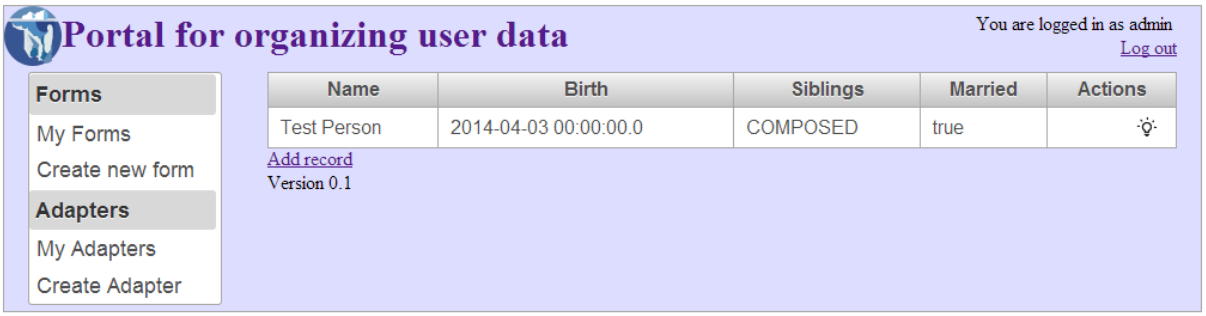
B.3 Vytvoření formulářového záznamu



Na přehledu formulářů klikněte na ikonku žárovky. Budete odkázáni na stránku pro vkládání formulářových záznamů.

Klikněte na odkaz Add rekord a vyplňte formulářový záznam podle následujícího obrázku:

Po kliknutí na tlačítko save vytvoříme formulářový záznam.



Příloha C

C. Obsah CD

Přiložené CD obsahuje zdrojové kódy Java, text diplomové práce ve formátu PDF a zdrojový soubor ve formátu docx. CD dále obsahuje běhové prostředí s nasazenou aplikací POUD a s instalačními soubory pro databázi MySQL, která je nezbytně nutná k běhu aplikace.

CD obsahuje následující složky a soubory:

- **Distribution** – obsahuje zkompilovanou aplikaci POUD
- **Poud** – zdrojové kódy aplikace POUD
- **Test-runtime-environmen** – předpřipravené běhové prostředí spolu s instalačními soubory programů, které jsou nutné pro běh aplikace POUD
- **Text** – obsahuje zdrojový text diplomové práce
- **vrchlpet_2014Dipl.pdf** – diplomová práce v PDF formátu