Možnosti návrhu webových aplikací (Výňatek z diplomové práce)

Lukáš Gemela, A11N0101P
lukas.gemela@gmail.com

25. června 2013

Obsah

1 Úvod

Následující text je výňatek z diplomové práce *Mobilní aplikace pro rezervace objektů*. Tato diplomová práce si klade za cíl vytvořit systém pro obecnou správu výpůjček a rezervací movitým a nemovitých objektů, a to pomocí unikátních možností mobilních zařízení. Protože výsledný systém implementuje centrální server s webovým rozhraním a dalšími službami, je součástí diplomové práce je i analýza možností implementace takové serverové aplikace. Tato analýza je poté i samotným obsahem tohoto dokumentu.

Budou zde rozebrány možnosti návrhu a implementace systému pro správu rezervací a uvedeny některé využitelné technologie založené na platformě Java (cílem tohoto textu však není poskytnutí kompletní programátorské dokumentace). Čtenář se seznámí s obecnými architektonickými principy uplatnitelnými při vytváření informačních systémů a distribuovaných webových aplikací.

2 Vícevrstvá architektura

Vícevrstvá architektura je architektonický vzor, ve kterém se model aplikace skládá z více vzájemně spolupracujících vrstev. Sousedící vrstvy komunikují přes předem definovaná rozhraní a díky tomu mohou být zaměňovány, aniž by to mělo dopad na funkčnost celé aplikace nebo bylo nutné měnit ostatní vrstvy. Přenos dat a definování vzájemného rozhraní mezi vrstvami je součástí návrhu vícevrstvé architektury.

Pravděpodobně nejznámějším zástupcem uplatnění vícevrstvé architektury je referenční OSI (*Open Systems Interconnection*) model. Jeho původním účelem byla snaha o standardizaci komunikace uvnitř počítačových sítí [1], dnes se však používají jeho jednodušší odvozeniny (např. protokol TCP/IP) a původní model má spíše metodický význam. Je však příkladem, že vícevrstvá architektura se nemusí omezovat pouze na čistě softwarovou implementaci.

Model se skládá ze sedmi vrstev. Nejvyšší vrstva v modelu zajišťuje přístup aplikací k síťovému komunikačnímu systému. Pokud klientská aplikace vytvoří požadavek na přístup do sítě, je tento předáván sestupně v zásobníku vstev až do vrstvy nejnižší, která zajišťuje přímý přístup k fyzickému médiu. Každá vrstva má přesně vymezenou sadu úkolů jako je šifrování, navazování spojení, synchronizace, směrování atp. a má jednoznačně definováno rozhraní mezi sousedními vrstvami v zásobníku.

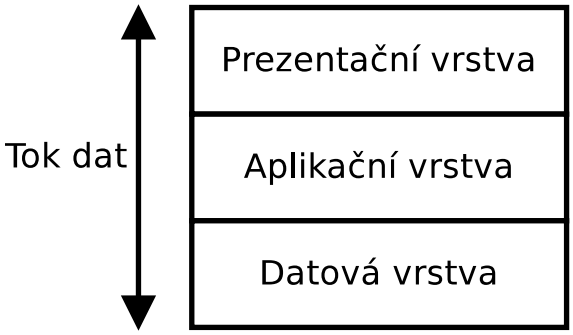
Na tom lze názorně demonstrovat výhody vícerstvých architektur. Díky tomu, že se každá vrstva omezuje na předem definovanou sadu úkolů a každá pracuje s jinou úrovní abstrakce, zbývá jen malý krok k vytvoření standardizovaných protokolů pro každou vrstvu. Jednotlivé implementace těchto protokolů bude poté možné libovolně vyměňovat bez ovlivnění funkcionality ostatních vrstev. Pokud se ukáže, že současná implementace je již příliš složitá, je možné vrstvy dále dělit. Tyto lze dokonce i slučovat, jak se také stalo ve výše zmíněné od OSI modelu odvozené architektuře TCP/IP, která využívá pouze čtyř vrstev.

2.1 Zásady pro návrh vícevrstvého modelu

Architekt aplikace by měl navrhnout novou vrstvu všude tam, kde je zapotřebí jiný stupeň abstrakce. Každá vrstva by měla zajišťovat přesně vymezené funkce zvolené tak, aby pro jejich realizaci mohly být vytvořeny standardizovaná rozhraní nebo protokoly. Počet vrstev by měl být tak velký, aby vzájemně odlišné funkce nemusely být zařazovány do stejné vrstvy, a současně s tím tak malý, aby celá architektura zůstala dostatečně přehledná. Možné budoucí komplexní přepracování vrstvy nesmí zásadně ovlivnit sousední vrstvy. Rozhraní mezi vrstvami by měla být zvolena tak, aby byl minimalizován tok dat.

2.2 Třívrstvá architektura

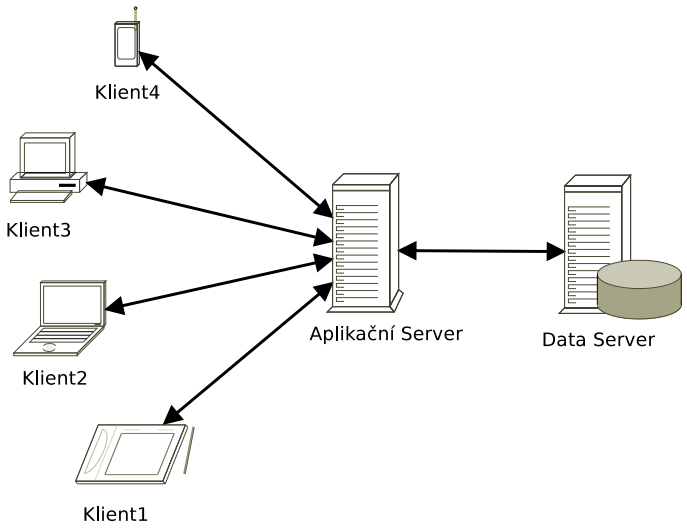
Třívrstvá architektura je obecný typ vícevrstvé architektury. Je s oblibou využívána v kombinaci s modelem distribuované aplikace klient-server. Rozděluje aplikaci do tří separátních logických vrstev, které nemusí nutně běžet na stejných zařízeních nebo operačních systémech. Rozdělení architektury je zjednodušeně znázorněno na Obr. 2.1.
Prezentační vrstva zajištuje vizualizaci dat pro uživatele a interakci s uživatelem [3]. Aplikační (doménová) vrstva obsahuje samotné jádro aplikace, funkce pro výpočty nebo zpracování dat.



Obrázek 2.1: Základní schéma třívrstvé aplikace.

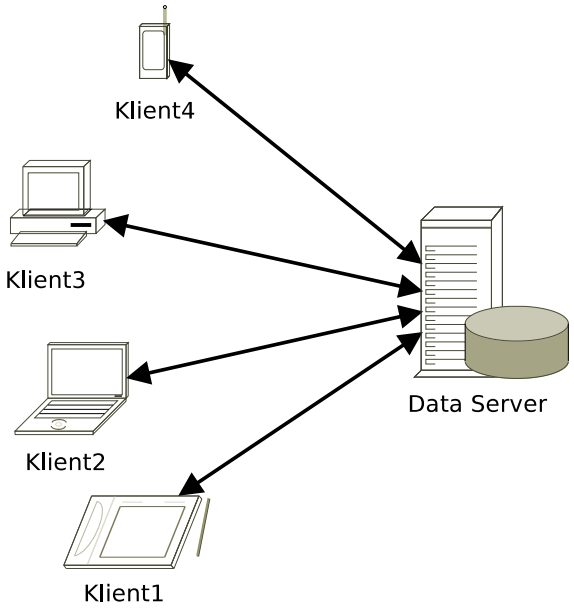
Datová vrstva se poté stará o persistenci dat. Je nutné zmínit, že v čistě třívrstvé architektuře se tok dat děje vždy jen mezi sousedícími vrstvami, tzn. při přístupu z prezentační vrstvy nemůže být prostřední vrstva překročena a opačně. Úkolem třívrstvé architektury není definování rozhraní mezi vrstvami nebo technologického základu pro třívrstvé aplikace, klade si za cíl pouze rozvržení logického modelu aplikace.

Na Obrázku 2.2 je znázorněno typické rozvržení třívrstvé aplikace. Datová vrstva je zde reprezentována relační databází na vzdáleném serveru. K této databázi se připojuje aplikační server, zajišťující výpočetní jádro systému. K aplikačnímu serveru se připojují jednotliví klienti – na těchto zařízeních jsou data vizualizována a je umožněno uživatelům těchto klientů s daty pracovat. Díky existenci aplikačního serveru tito klienti neobsahují žádnou další aplikační logiku a hrají roli tzv. *tenkých klientů*.



Obrázek 2.2: Příklad třívrstvé architektury (*tenký klient*).

Nicméně díky obecnosti třívrstvé architektury je možné si představit celou řadu jiných scénářů. Kupříkladu relační databáze může fungovat přímo na aplikačním serveru nebo může úplně chybět a roli datové vrstvy sehraje např. implementace transformující data do XML souborů. Na Obr. 2.3 je pak znázorněna situace, kde aplikační server zcela chybí. Aplikační vrstva může být poté přítomna jako součást relační databáze v podobě uložených procedur a tzv. *triggerů* (viz dále), nebo mohou jednotliví klienti s sebou nést i aplikační logiku – stanou se tzv. *tlustými klienty*. Je možné tyto postupy i kombinovat. Je možné (a dnes již celkem obvyklé), aby část aplikační logiky ležela na aplikačním serveru a část kódu se spouštěla na jednotlivých klientech.



Obrázek 2.3: Příklad třívrstvé architektury (*tlustý klient*).

Využití třívrstvé architektury je tedy velice flexibilní. Při návrhu systému si softwarový architekt musí velmi dobře rozmyslet celkové rozvržení jednotlivých logických vrstev s ohledem na výkon, bezpečnost, udržitelnost, technické vlastnosti a možnosti klientských zařízení a další rozšiřitelnost navrhovaného systému. Je také nutné vždy jednoznačně definovat komunikační rozhraní mezi vrstvami podle zásad návrhu vícevrstvých aplikací tak, aby bylo možné jednotlivé vrstvy zaměnit nebo je standardizovat.

Nyní si detailněji rozebereme jednotlivé části a další architektonické vzory třívrstvé architektury s ohledem na možné využití při implementaci distibuovaných webových aplikací.

3 Aplikační vrstva

Aplikační (nebo také doménová) vrstva je taková vrstva, do které by mělo být soustředěno maximum výkonného kódu a vlastní logiky aplikace. Existuje celá řada obecných architektonických vzorů uplatnitelných při návrhu této vrstvy, nicméně zejména u webových aplikací se vrstva obvykle rozdělí na *doménový model*, ve kterém jsou definovány základní entity aplikace a jejich vzájemné vazby, a *servisní vrstvu*, která s tímto modelem manipuluje nebo vytváří jeho rozhraní [3].

3.1 Doménový model

Doménový model vytváří jakousi síť objektů navzájem spojených vazbami, kde každý objekt obvykle reprezentuje nějakou entitu v reálném světě. Návrhem doménového modelu by měl začínat jakýkoliv návrh implementace aplikace a jeho programové realizaci obvykle předchází vytvoření analytického doménového modelu za využití jazyka UML (*Unified Modeling Language*).

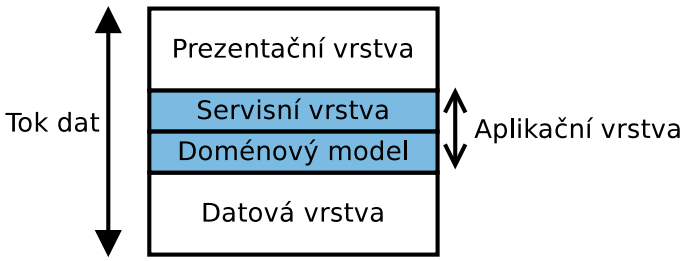
Objekty doménového modelu mají sadu atributů definovanou tak, aby se co nejvíce přiblížily svému reálnému vzoru. Každý objekt doménového modelu by však měl být jen tak velký, jak je to bezpodmínečně nutné. V případě příliš komplexního objektu je vhodné ho rozdělit na menší objekty a tyto propojit vazbami. Kvůli snadné rozšiřitelnosti a udržitelnosti je důležité mít neustále možnost během životního cyklu aplikace snadno doménový model měnit a testovat. Dále je nutné udržovat co nejméně vazeb modelu na ostatní části systému (což je mimo jiné cílem mnoha návrhových vzorů).

Ve světě Javy jsou obvykle jednotlivé doménové objekty realizovány pomocí POJO (*Plain Old Java Object*) objektů. Definice POJO vznikla jako reakce na nepříliš flexibilní EJB2 (*Enterprise Java Bean*), které se váží na použitý framework a nesplňují tak požadavek na co nejnižší počet vazeb modelu na okolní programové prostředí. Pro POJO neexistuje žádný uchopitelný standard, nicméně lze je definovat jako instance každé *Java Beany*, která se snaží minimalizovat své vazby vůči okolí ať už po stránce anotací, dědičnosti nebo implementace rozhraní. To samozřejmě není (například v případě spolupráce aplikace s jinými frameworky) téměř nikdy možné, nicméně zejména podmínka neexistující dědičnosti od nějaké komponenty užitého frameworku by měla být splněna.

Pod pojmem *Java Bean* se poté chápe taková třída, k jejímž členským proměnným se přistupuje pouze pomocí veřejných *getterů* a *setterů* a tyto dodržují jisté jmenné konvence. Pro přístup k jednotlivým členským proměnným se díky jednoduchosti POJO může využít pokročilých programovacích technik, jako je aspektové programování a reflexe. Obdobná situace je i v jiných programových prostředích (např. v *.NET Frameworku*), i když syntax kódu a užitá terminologie se samozřejmě liší.

3.2 Servisní vrstva

Jak ukazuje Obr. 3.1, servisní vrstva je v hierarchii vrstev umístěna na vyšší úrovni abstrakce než je doménový model a současně vytváří programové rozhraní (*Application Programming Interface*, zkratkou API) pro prezentační vrstvu. Jinak než skrze servisní vrstvu nelze s doménovým modelem pracovat.



Obrázek 3.1: Základní rozdělení aplikační vrstvy.

Servisní vrstva je díky tomu, že tvoří „úzké hrdlo“ v toku dat, dobrým místem nejen pro umístění aplikační logiky, ale i zabezpečení nebo kontroly dat. Podle [3] existují dva přístupy, jak vytvořit servisní vrstvu. První z nich je tzv. **Doménová Fasáda**. Ta principiálně deleguje aplikační logiku do doménového modelu a tvoří pouze zmiňované rozhraní mezi prezentační a aplikační vrstvou a celou servisní vrstvu tvoří pouze množina *fasád* nad doménovým modelem. Fasády obsahují pouze ty operace, které jsou poskytnuty klientovi nad doménovým modelem a samotná servisní vrstva je tak velice tenká.

Druhý princip, tzv. **Operation Script**, funguje přesně opačně. Doménový model neimplementuje žádnou aplikační logiku a tato je celá implementována přímo v servisní vrstvě. Rozhraní dostupné klientovi tedy nedeleguje práci do doménového modelu, ale s doménovým modelem přímo pracují. V tomto případě se dá již hovořit o robustní servisní vrstvě a třídy hrající roli servis bývají programově velmi rozsáhlé.

Oba přístupy mají své klady a zápory. V případě doménové fasády se dá jistě hovořit o lepší dekompozici aplikační logiky a tím pádem rozšiřetelnější implementaci. S tím se ovšem také pojí netriviální návrh aplikace. Navíc díky tomu, že je aplikační logika rozeseta do mnoha míst v doménovém modelu, může se s rostoucím počtem doménových objektů tvořit duplicitní kód. U druhého přístupu je aplikační kód pouze na několika málo místech a to umožňuje snažší refaktorizaci kódu, na druhou stranu to znesnadňuje budoucí rozšiřitelnost doménového modelu.

4 Vrstva persistence dat

V předchozím textu jsme zavedli základní terminologii vrstev a třívrstvého architektonického vzoru. Nyní je nutné detailněji rozebrat otázku zachování persistence dat a programového přístupu k těmto datům (tedy nejnižší vrstvy třívrstvého modelu).

4.1 Systém řízení báze dat

Řešení problému uchovávání dat spočívá obvykle v nasazení některé z implementací systému řízení báze dat (SŘBD nebo anglicky DBMS – *Database management system*). Jejich hlavním účelem je poskytnutí řízení (vkládání, editace, mazání) nad bází dat (neboli *databází*), definovat strukturu uložení těchto dat a vytvořit rozhraní mezi aplikačními programy a uloženými daty [2]. Pojmem „databázový systém“ poté obvykle označuje spojení SŘBD se samotnou bází dat.

Použití některého z databázových systémů přináší celou řadu výhod, jako je udržování strukturovanosti dat, podpora transakcí, možnost agregace dat, řízení přístupových práv atd. Databázové systémy se obvykle snaží maximálně využít operačního a souborového systému tak, aby bylo získávání a ukládání dat co nejvíce efektivní. Převážná většina dnes používaných SŘBD pak pro strukturování dat v databázi využívá tzv. *relační model*.

Relační model

Relační model sdružuje data do tabulek, které obsahují n-tice jednotlivých entit (řádků). Tabulka je struktura entit s pevně stanovenými atributy (sloupci). Každý sloupec má jasně definován jednoznačný název, typ a možný rozsah vkládaných dat. Pokud jsou v různých tabulkách sloupce stejného typu, pak tyto sloupce mohou vytvářet vazby mezi jednotlivými tabulkami. Tabulky se poté naplňují konkrétními daty. Kolekce více tabulek, jejich funkčních vztahů, indexů a dalších součástí tvoří samotnou relační databázi.

Relační model umožňuje přirozenou reprezentaci zpracovávaných dat a je v něm možné při návrhu databáze snadno a poměrně intuitivně definovat jednotlivé vazby mezi tabulkami. Model také klade velký důraz na zachování integrity dat. Model dále pracuje s termíny jako je *referenční integrita*, *cizí klíč*, *primární klíč*, *normální forma* apod. Relační databázový systémy obvykle pro kontrolu databáze a nastavování dalších direktiv využívají strukturovaný dotazovací jazyk SQL (*Structured Query Language*).

Problematika databází je velmi obsáhlá a pro účely této práce postačí, pokud se nadále budeme zabývat jen některými vlastnostmi relačních databázových systémů uplatnitelných v prostředí webových distribuovaných aplikací.

4.1.1 Uložené procedury a triggery

Jak již bylo zmíněno v předchozím textu, databázové systémy umožňují částečně převzít roli aplikační vrstvy díky uloženým procedurám a tzv. *triggerům*. To jsou v podstatě bloky kódu napsaného v některém programovacím jazyce (např. PL/SQL) vytvořeného speciálně pro účel běhu v prostředí relační databáze. Lze je kupříkladu využít pro zachování referenční integrity dat. Výhodou je zpravidla mnohem větší rychlost zpracování než pokud by tyto bloky kódu měly být řízeny vzdáleně klientskou aplikací.

Kromě velmi specifických případů však nelze použití procedur a *triggerů* příliš doporučit. Jsou jen velmi obtížně udržovatelné, na velmi nízké úrovni abstrakce, není možné je rozumně debuggovat a jejich podpora se liší v závislosti na použité implementaci databázového systému.

4.1.2 Nevýhody relačních databází

Ačkoliv existují standardy pro jazyk SQL, mezi jednotlivými implementacemi relačních databází jsou značné rozdíly v jejich podpoře. Obvykle nejsou implementovány všechny požadavky standardu a naopak, každá implementace obsahuje nejrůznější prvky, které nejsou ve standardech obsaženy. Přenositelnost SQL sekvencí mezi jednotlivými databázemi je proto omezená. Současně není možné jednoduše přenášet data mezi různými databázemi.

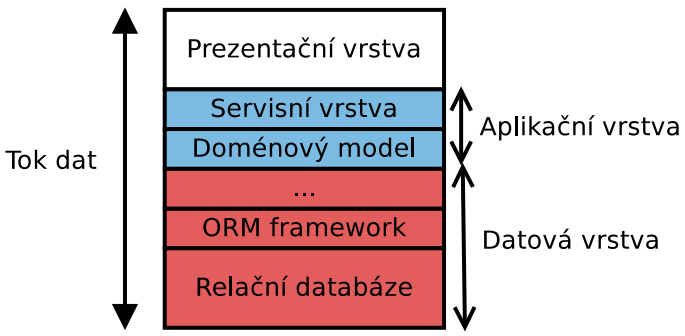
Relační databázové systémy jsou dobré pro řízení velkého množství dat, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. SQL sice umožňuje tabulky za běhu propojit, nicméně tento přístup je velmi neintuitivní a nepřehledný. Relační model databází je sice jednoduchý a přitom velmi robustní a flexibilní, je ale také naprosto rozdílný od objektového modelu. Relační databáze nejsou navrhovány pro ukládání objektů a vytvoření rozhraní pro ukládání těchto objektů v relační databázi je netriviální úkol.

Alespoň částečné řešení obou výše nastíněných problémů tkví v použití buď objektově orientovaných databází (které ovšem nejsou příliš využívané), nebo v nasazení tzv. *objektově-relačního mapování*.

4.2 Objektově-relační mapování

Jako objektově-relační mapování (*Object-Relational Mapping*, zkratkou ORM) se označuje sada programovacích technik, které se snaží zajistit automatickou konverzi dat mezi relační databází a doménovým modelem a následnou synchronizaci doménových objektů v aplikaci a jejich reprezentaci v databázovém systému tak, aby byla zajištěna persistence dat (Obr. 4.1). Díky ORM je možné přirozeně pracovat s doménovými objekty, aniž by se programátor staral jak zajistit jejich perzistenci.

Ve [3] jsou rozebrány některé základní návrhové vzory, které umožní svépomocí implementovat ORM. Nicméně protože zajištění synchronizace mezi databází a doménovým modelem je netriviální problém zejména v kontextu paralelního přístupu a výkonnosti celé aplikace, využívá se dnes obvykle již ustálených a standardizovaných frameworků, které ORM mapování zajistí.



Obrázek 4.1: Využití objektově-relačního mapování.

Některé implementace ORM frameworků navíc programátora odstíní od poněkud nepohodlného SQL jazyka a poskytnou mu jiné možnosti přístupu do databáze, které obvyklé implementace databázových systémů nenabízí. Dále také umožňují částečně smazat rozdíly mezi jednotlivými implementacemi databázových systémů – je možné konfigurovat SQL dialekt, se kterým framework

komunikuje s databází a tím je lze při změně databázového systému místo přepisování celé datové vrstvy pouze překonfigurovat ORM framework.
I přes nesporné výhody je však nasazení těchto frameworků spíše počátek cesty za bezproblémovým mapováním mezi doménovým modelem a relační databází a v některých případech se dokonce vyplatí jejich nasazení úplně vyhnout.

4.2.1 Java Persistence API

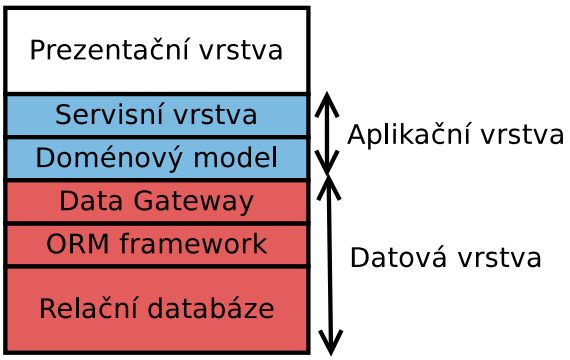
Specifikací pro ORM frameworky na platformě Java je tzv. *Java Persistence API* (zkratkou JPA). Původně existovalo několik různých ORM frameworků, které vznikly jako reakce na příliš komplikovaný tehdejší model EJB, který se navíc dal použít pouze ve světě rozsáhlých Java EE aplikací [4]. V rámci standardizace těchto ORM frameworků poté dodatečné vznikla specifikace JPA, jejíž podporu následně stávající frameworky převzaly. Nejznámějšími implementacemi JPA je Hibernate a EclipseLink. Při dodržení konvencí předepsaných JPA je možné tyto dva frameworky na ORM vrstvě navzájem zaměnit bez změny implementace.

JPA mimo jiné definuje i dotazovací jazyk JPQL (*Java Persistence Query Language*). Ten je syntaxí podobný SQL, nicméně namísto tabulek se pohybuje na úrovni doménových objektů. Některé frameworky jazyk dále rozšiřují, například Hibernate má vlastní jazyk HQL (*Hibernate Query Language*), jemuž je JPQL podmnožinou.

4.3 Data Gateway

I v okamžiku využití ORM frameworků se nelze vyhnout kódu, který přímo závisí na logice relačních databází. Zejména pro CRUD (*Create-Read-Update-Delete*) operace je stále nutné využít at' už generické SQL nebo například výše zmíněný jazyk JPQL. Je naprosto nežádoucí „míchat“ tento specifický kód do aplikační vrstvy už jen s ohledem na to, že některá z budoucích odvozenin aplikace může využívat místo relačních databází například XML soubory, s nimiž se ovšem pracuje diametrálně odlišně.

Obvyklým postupem je tedy vytvoření programového rozhraní pro přístup k persistentní vrstvě. Implementace tohoto rozhraní bude poté obsahovat všechen kód specifický pro technologii použitou pro persistenci dat. Takové implementace poté tvoří jakousi bránu (*gateway*) mezi architektonickými věty aplikační a persistentní vrstvy (Obr. 4.2).



Obrázek 4.2: Rozhraní mezi aplikační a persistentní vrstvou.

Není žádoucí do těchto objektů vkládat aplikační logiku – rozhraní by mělo být co nejjednodušší a zajišťovat pouze základní CRUD operace. Jakákoliv komplexní logika by měla být součástí klienta tohoho rozhraní. Implementace se poté často neprogramují ručně, ale využívá se spíše různých možností dědičnosti nebo generátorů kódu. Je dokonce možné vytvářet implementace rozhraní přímo za běhu aplikace pomocí *aspektově orientovaného programování* (viz dále).

4.3.1 Data Access Object

Jako *Data Access Object* (*objekt zpřístupňující data*, zkratkou DAO) se tradičně označuje uplatnění návrhového vzoru *Data Gateway* v prostředí Java aplikací. Rozhraní DAO navenek poskytuje metody pro práci s doménovými objekty – instancemi POJO nebo EJB, a umožňuje jejich persistenci. Vnitřně poté zajišťují „překlad“ těchto objektů do tvaru, kterému rozumí relační databáze. To může být zajištěno mnoha cestami – parametrizací generických SQL dotazů nebo například využitím již zmiňovaných ORM frameworků.

Seznámili jsme se s obvyklými architektonickými návrhovými vzory pro spojení datové a aplikační vrstvy. Nyní je nutné vyřešit poslední prázdné místo v rozboru třívrstvé architektury – prezentační vrstvu.

5 Prezentační vrstva

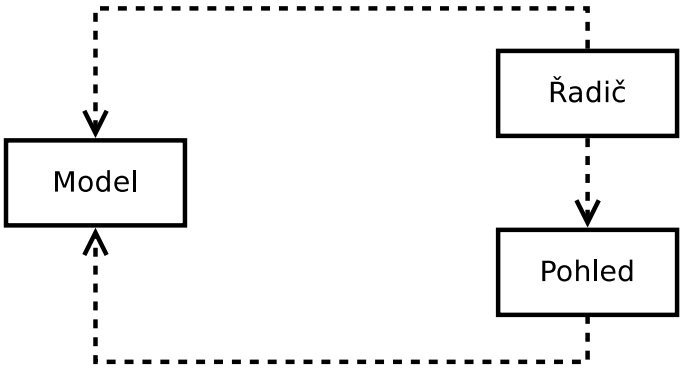
Úkolem této kapitoly je ukázat obvyklá architektonická řešení rozhraní mezi aplikační a prezentační vrstvou specifická pro webové aplikace a představit některé technologie, které tyto obecné vzory implementují.

5.1 Architektura MVC *(Model-View-Controller)*

MVC je architektonický vzor, který odděluje doménový model aplikace od uživatelského rozhraní. Vznikl spolu s prvními návrhy grafického uživatelského rozhraní a dnes se uplatuje zejména v rámci distribuovaných webových aplikací. Popis jednotlivých komponent je následující:

- *Model* – v klasickém třívrstvém modelu obvykle odpovídá doménovému modelu v aplikační vrstvě. Nicméně frameworky implementující MVC obvykle tyto komponenty neumí navzájem synchronizovat a je tedy na programátorovi, aby synchronizaci zajistil (například přes servisní vrstvu).
- *Pohled (View)* –převádí data reprezentovaná modelem do podoby vhodné k zobrazení uživalům.
- *Řadič (Controler)* – reaguje na události od uživatele a zajišťuje změny v modelu nebo v pohledu.

Ačkoliv by se mohlo zdát, že MVC je vlastně třívrstvá architektura (a oba architektonické vzory opravdu bývají často zaměňovány a jsou si na první pohled podobné), není to vůbec pravda. Topologie MVC totiž není lineární, ale tvoří jakýsi trojúhelník. Na rozdíl od obecné třívrstvé architektury také definuje základní tok dat a pohybuje se na rozhraní aplikační a prezentační vrstvy, způsoby zajištění persistence doménového modelu nejsou architekturou nijak pokryty a obecně nelze zaměňovat datovou vrstvu s MVC *Modelem*. Na Obrázku 5.1 je zobrazeno základní schéma MVC architektury.



Obrázek 5.1: Základní schéma MVC architektury.

Důležitý je zejména fakt, že *Řadič* a *Pohled* závisejí na *Modelu*, ale *Model* nezávisí ani na jedné z těchto dvou komponent. To (mimo jiné) umožňuje testovat doménový model aplikace nezávisle

na vizuální prezentaci dat. Dále můžeme podle [3] rozdělit MVC architekturu na dva základní typy.

5.1.1 Pasivní MVC

V pasivním modelu je to pouze *Řadič*, který je oprávněn manipulovat s *Modelem*. *Řadič* modifikuje *Model* a poté informuje *Pohled*, že se *Model* změnil. Tento se poté překreslí, aby zobrazoval aktuální data. *Model* je v pasivním MVC naprosto nezávislý a proto nemusí nijak notifikovat změnu sám sebe – tuto činnost obstará *Řadič*.

Příkladem pasivního MVC může být klasická webová prezentace přes HTTP protokol. Bez využití dalších technologií neexistuje cesta, jak jednoduše přijímat asynchronní aktualizace dat ze serveru. Webový prohlížeč zobrazí *Pohled* a umí zachytávat vstupy od uživatele, neumí však detekovat žádné změny modelu na serveru.

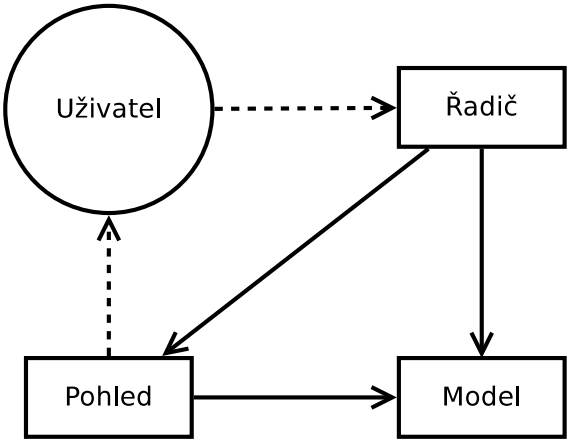
5.1.2 Aktivní MVC

V aktivním MVC může *Model* měnit svůj stav bez jakéhokoliv zapojení *Řadiče*. To se může stát v případě, kdy jsou data měněna z nějakých dalších zdrojů (například jiným uživatelem, přímo v databázi, opakující se paralelní úlohou atp.) a změny se musí okamžitě projevit do *Pohledu*. *Model* tedy notifikuje změnu sama sebe, *Pohled* tuto změnu zachytí a dojde k překreslení okna grafického uživatelského rozhraní. Zapojení *Řadiče* se nicméně nijak nevylučuje. Může to být právě *Řadič*, který *Model* změní, nicméně aktualizaci *Pohledu* nebude sám zajišťovat a nechá ji plně v kompetenci *Modelu*.

5.1.3 Tok událostí v MVC

Pro úplné pochopení MVC architektury jsou na Obrázku 5.2 znázorněny typické interakce mezi uživatelem a aplikací využívající MVC. Tok událostí v aplikaci vypadá následovně:

- uživatel vykoná nějakou akci v uživatelském rozhraní.
- událost je zachycena *Řadičem*.
- *Řadič* provede rozhodnutí o tom, jak na událost zareagovat a změní hodnoty v modelu nebo přímo ovlivní *Pohled*.
- *Pohled* se aktualizuje a uživateli zobrazí změny v *Modelu* (nebo se zobrazí zcela jiný *Pohled*).

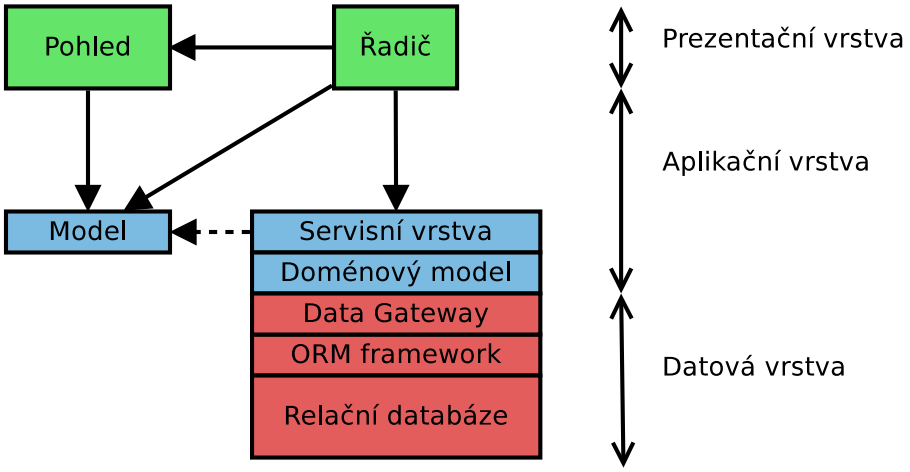


Obrázek 5.2: Scénář interakce v MVC architektuře.

V celém modelu je to tedy *Řadič*, který hraje dominantní úlohu a určuje, jak se bude reagovat na události vzniklé na straně uživatele – odtud anglický název *Controller*.

5.1.4 Spojení MVC a servisní vrstvy

Nabízí se otázka, jak vyřešit již dříve zmíněný problém synchronizace *Modelu* s doménovým modelem aplikace. V Kap. 3 jsme definovali servisní vrstvu, která bude vytvářet programové rozhraní mezi prezentační vrstvou a doménovým modelem aplikace. Na obrázku 5.3 je znázorněno jedno z možných spojení MVC se servisní vrstvou (někdy nazývané jako MVCS - *Model View Controller Service*). Jak již bylo dříve zmíněno, servisní vrstva může s doménovým modelem manipulovat přímo podle vzoru *Operation Script*, nebo může tvořit pouze doménovou fasádu a delegovat logiku do doménového modelu.



Obrázek 5.3: MVC architektura ve spojení se servisní vrstvou.

Řadič tedy může přímo měnit *Model* a na žádost uživatele může provést například uložení dat z *Modelu* do databáze prostřednictvím servisní vrstvy. Servisní vrstva může mít přímou referenci na *Model* nebo může být aktualizace provedena přes *Řadič*.

5.2 MVC v prostředí webu

Rozlišovat rozdíl mezi *Pohledem* a *Řadičem* není u desktopových aplikací až tak důležité a některé frameworky explicitně logické rozdělení příslušné části prezentační vrstvy do těchto MVC komponent nevyžadují (například stále široce používaný Swing toolkit).
Nicméně zejména u webových aplikací je situace zcela jiná a MVC architektura je zde naprosto přirozeně využita pro dekompozici komponent starajících se o generování HTML kódu od částí systému obsluhujících HTTP požadavky:

- *Model* je identický s *Modelem* v desktopových technologiích (obsahuje data a doménovou logiku).
- *Pohled* je ta část serverového kódu, která se stará o generování HTML kódu. Může však prezentovat data i jinak, například ve formátu XML, JSON (*JavaScript Object Notation*), PDF (*Portable Document Format*) atp.
- *Řadič* se v prostředí webu nejčastěji skládá ze dvou hlavních částí. První je tzv. *Front Controller*, který zachytává všechny HTTP požadavky. Ty následně zpracuje a přepošle dalším *Řadičům*, které jsou obvykle identifikovány podle URI. Konkrétní *Řadič* potom typicky přijme data původně pocházející z HTTP požadavku, uloží je do *Modelu* a ten prováže s konkrétním *Pohledem*.

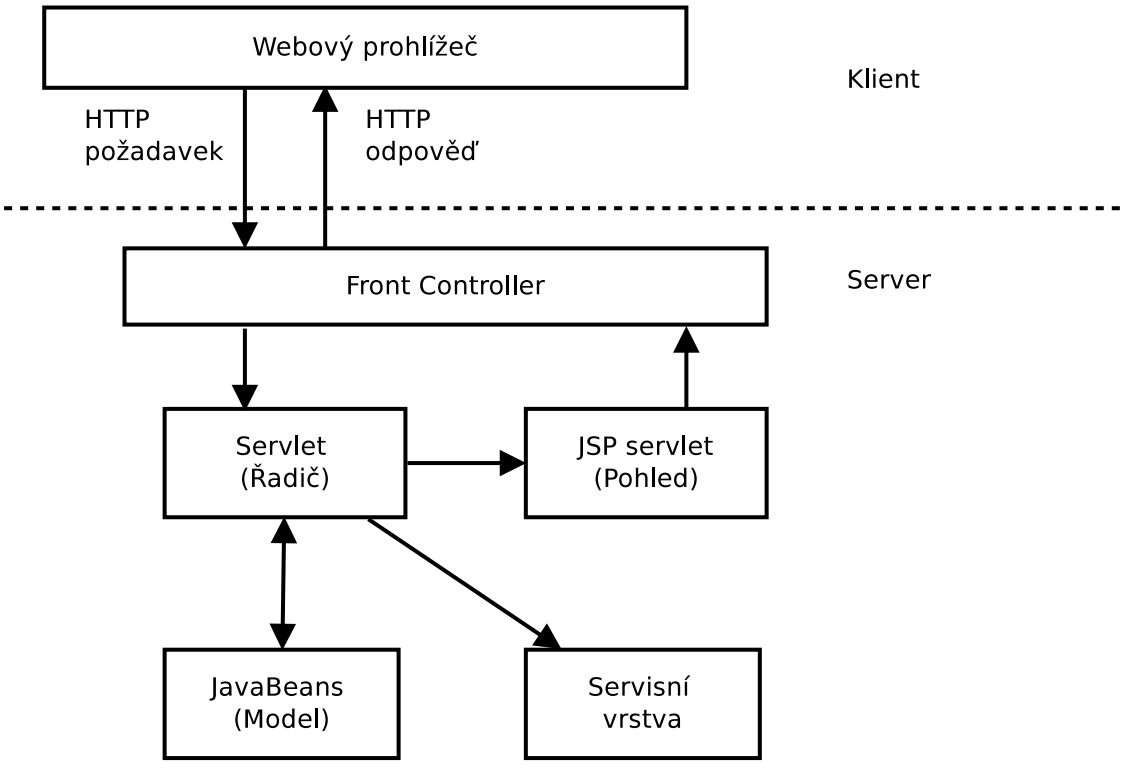
Situace se komplikuje v případě, kdy je využito technologií jako je AJAX (*Asynchronous JavaScript and XML*). V těchto situacích je značná část prezentační logiky (tedy *Pohledu*) přesunuta na

klienta (jak velká část záleží na konkrétní implementaci). Nicméně lze říci, že i v těchto případech je architektura MVC využitelná.

JavaServer Pages

Jako nejtypičtějšího představitele uplatnění MVC ve světě webových Java aplikací můžeme jmenovat technologii tzv. *servletů* a s tím související *JavaServer Pages* (JSP). Servlet je velmi obecně řečeno třída odpovědná za zpracování HTTP požadavku a vytvoření odpovídající odpovědi [5]. Servletu bývá v průměrné webové aplikaci celá řada a jejich organizaci a mapování na jednotlivá URL má na starosti tzv. *servletové* nebo také *webové kontejnery*, což jsou vlastně obecně známé „Java servery“ jako je například Apache Tomcat nebo GlassFish. Tyto kontejnery obvykle vytvoří jednu nebo více instancí od každého servletu a pokud na server přijde HTTP požadavek, je vyvolána odpovídající metoda instance servletu namapovaného na URL tohoto požadavku.

JSP servlet je poté velmi specifický typ servletu. Jeho programový zápis totiž není realizován v Javě, ale ve struktuře velmi blízké HTML. Je možné do značek HTML „míchat“ další tagy ze standardních JSP knihoven, vytvořit si knihovnu vlastní nebo dále používat dalších šablonovacích nástrojů. Lze také omezeně využít Java kód. Takto zapsaný servlet je poté interně v servletovém kontejneru automaticky konvertován na Java zdrojový kód a následně se chová jako každá jiná třída. Nicméně jak ukazuje Obr. 5.4, cílem JSP servletů je jediná věc – formátovat data obdržená ve od *Řadiče* do podoby HTML kódu (schéma není zcela přesné, obvykle nejsou data do JSP servletu předána přímo přes *Řadič*, ale přes nadřazený *Front Controller*).



Obrázek 5.4: Základní architektura JSP.

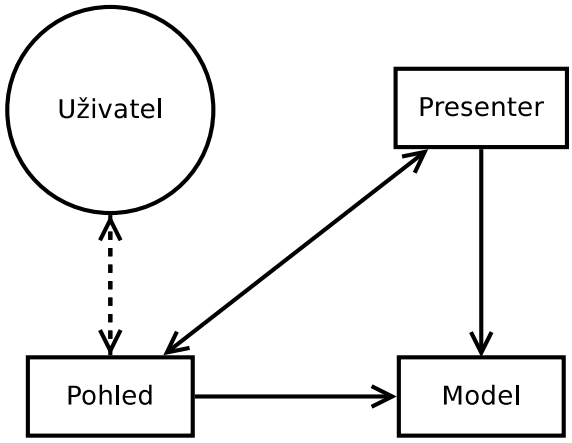
Řadič poté obvykle bývá realizovaný jako klasický programově implementovaný servlet. V servletu *Řadiče* může být soustředěna veškerá logika související s přípravou dat k zobrazení nebo volání nižších vrstev pro načtení dat do *Modelu*. *Řadič* také může provádět přesměrování a další věci, které nutně nesouvisí s aplikační logikou – ta by měla ležet vždy v aplikační vrstvě.

Původní čisté JSP se dnes již u důvodu velké pracnosti vývoje moc nepoužívá a v praxi tuto technologii nahradily frameworky, které možnosti JSP dále rozšiřují. Je však nutné pochopit

principy fungování JSP a MVC architektury obecně, protože z nich většina moderních frameworků přímo vychází.

5.3 Architektura MVP (*Model-View-Presenter*)

Architektonický vzor MVP je principiálně podobný MVC a oba vzory bývají dost často navzájem zaměňovány. Nicméně jednotlivé komponenty hrají v MVP trochu jinou roli (Obr. 5.5).



Obrázek 5.5: Scénář interakce v MVP achitektuře.

Uživatelský vstup a výstup tentokrát plně kontroluje *Pohled* skrze ovládací prvky uživatelského rozhraní. Primární motivací pro oddělení *Pohledu* a *Presenteru* (budeme nadále používat anglické slovo „presenter“, protože pro ně není žádný vhodný český ekvivalent) už není nutnost ošetření událostí od uživatele a kontroly vstupu, důvody jsou čistě architektonické (lepší udržovatelnost kódu). *Pohled* má typicky přímou vazbu na *Presenter* a ten obvykle přímo pracuje s *Pohledem*, takže i tato vazba je silnější než v případě MVC. *Pohled* oproti MVC navíc zpracovává uživatelský vstup a je zde tedy dominantní komponentou (typicky v reakci na událost od uživatele zavolá nějakou metodu v *Presenteru*).

Presenter může obsahovat prezentační a aplikační logiku (nebo deleguje aplikační logiku do dříve popsané servisní vrstvy). Manipuluje s *Modelem*, což (například pomocí systému posluchačů a notifikací) zajistí aktualizaci *Pohledu*, nebo ovlivňuje *Pohled* přímo (záleží na implementaci a možnostech jazyka).

Popsaný architektonický vzor se také označuje jako *Supervising Presenter*. Existují další modifikace MVP, například *Passive View*, kdy je naopak zvýšena odpovědnost *Presenteru* a *Pohled* úplně ztrácí vazbu na *Model*. Jejich popis však přesahuje účel tohoto textu. Důvod, proč zde uvádíme tento architektonický vzor je zejména fakt, že se MVP velice často a zcela nesprávně zaměňuje s MVC. Příkladem využití MVP může být toolkit WinForms, který je součástí *.NET Frameworku* od společnosti Microsoft.

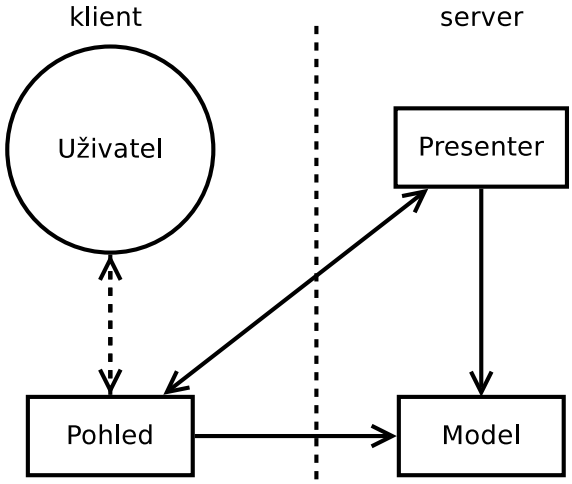
5.3.1 MVP v prostředí webu

MVC se dobře hodí pro klasické webové aplikace, kde jsou statické HTML stránky generovány přímo na serveru. Nicméně v případě moderních AJAX aplikací, kdy se část prezentační vrstvy přesouvá do klientského prohlížeče, začíná být MVC nevyhovující. Oproti tomu MVP architektura je pro tento účel jako stvořená.

Jak je vidět na Obrázku 5.6, komponenta *Pohledu* (obvykle implementovaná pomocí JavaScriptu) se přesouvá do klientského prohlížeče a místo programového rozhraní a volání metod budou tyto komunikovat například pomocí HTTP protokolu. Při vyplňování formulářů je možné validovat data přímo na straně klienta. Pokud tedy uživatel klikne na tlačítko, které má zobrazit nějaké

položky z databáze, je tento požadavek zpracován na straně webového prohlížeče a na pozadí se odešle žádost do *Presenteru*. Ten může ze servisní vrstvy načíst patřičná data za databáze a uložit do *Modelu*. Poté dojde opět pomocí HTTP protokolu k aktualizování *Pohledu* tak, aby zobrazil aktuální data.

Z toho vyplývá také potenciál snížit zátěž na servery a síť obecně. Jelikož není potřeba při každém požadavku sestavit celý HTML dokument, ale pouze zobrazit aktualizovaný *Model*, je množství vyměňovaných dat výrazně nižší. Nicméně tento přístup naopak může zvýšit počet vyměňovaných HTTP požadavků, a třebaže přenášejí nižší množství dat, při nevhodné implementaci zátěž neklesne.



Obrázek 5.6: Scénář interakce ve webové MVP aplikaci.

5.4 JavaServer Faces

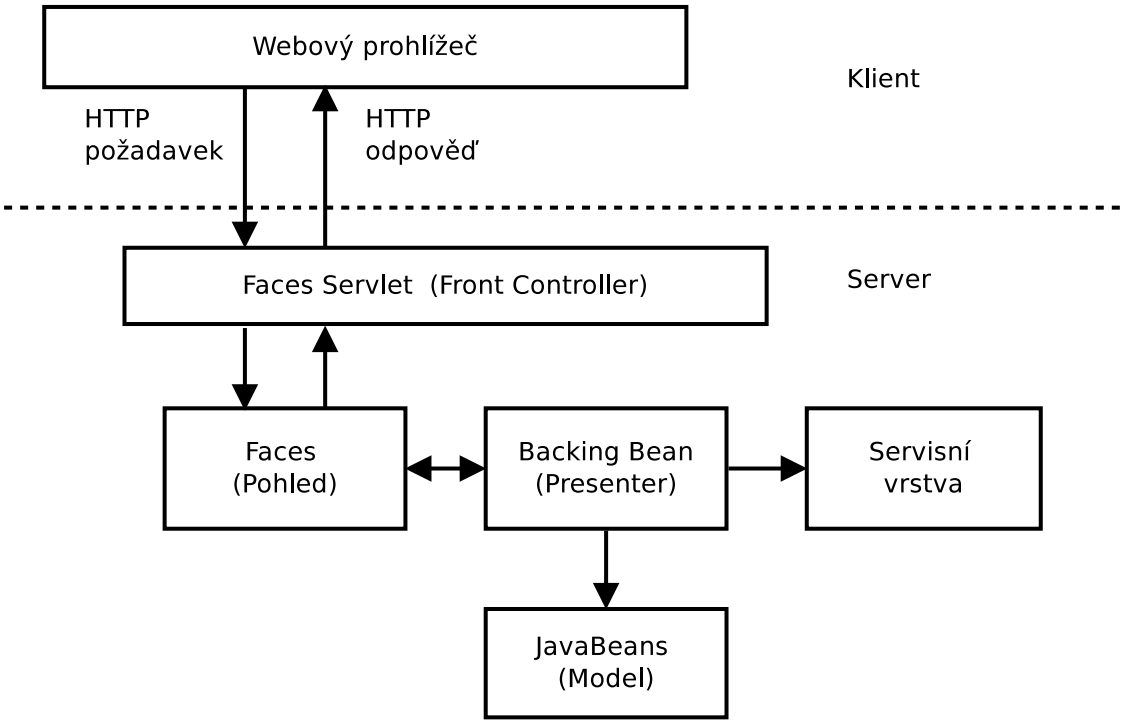
Již dříve jsme si ukázali JSP jako jedno z ukázkových nasazení architektury MVC ve webové aplikaci. Nyní si detailněji rozebereme technologii *JavaServer Faces* (zkratkou JSF), která i přes to, že z JSP přímo vychází, se naopak blíží spíše MVP architektuře.

Myšlenkou je opět oddělení definice uživatelského rozhraní (tzv. *Faces*) od vlastní aplikační a prezentační logiky. I zde se tak děje pomocí souborů podobných HTML, ve kterých je definováno vlastní uživatelské rozhraní. Při psaní těchto XML souborů se používají speciální XML tagy, které se importují z tzv. *Tag Library Description* (TLD) knihoven. I zde je využito *Front Controlleru* a vykreslování HTML stránky se odehrává na serveru [6].

První rozdíl přichází v okamžiku, kdy si uvědomíme, že konfigurace grafického rozhraní je komponentově orientovaná. Každý tag má interně v rámci své knihovny přiřazeno několik Java tříd. Soubor těchto tříd se nazývá *Komponenta*. Tato programová reprezentace vytváří aplikační logiku tagu (jako konverzi hodnot na text, validaci a přípravu pro „vykreslení“ informace). Ke každé komponentě je vytvořen její programový *Renderer*, který zapíše do výstupu vlastní HTML kód a data z reprezentace tagu. Je tedy možné vytvořit celé uživatelské rozhraní aniž bychom použili jediný HMTL tag. V JSF také odpadá nutnost využití šablonovacích frameworků (součástí je framework *Facelets*, který umožňuje šablonování).

Jak je vidět na Obrázku 5.7, JSF zavádí také pojem *backing* nebo také *managed beans*. Přestože jsou tyto „beany“ často realizovány jako POJO objekty, nemají vůbec nic společného s doménovým modelem aplikace. Reprezentují ty třídy, jejichž instance by měly být dynamicky vytvářeny za běhu aplikace spolu s generováním HTML stránek, přičemž je možné určit jejich „rozsah“ (*scope*) neboli kontext ve kterém budou konkrétní instance referencovatelné (je možné určit rozsah v rámci jednoho HTTP požadavku, uživatelské relace, v rámci celého běhu aplikace atp.). Jednotlivé člen-

ské proměnné a metody těchto objektů jsou poté za pomoci tzv. *bindování* možné referencovat přímo z XML souborů jednotlivých *Pohledů* a na události vyvolané uživatelem (například odeslání vyplněného formuláře) volat přímo metody *backing beany*.



Obrázek 5.7: Základní architektura JSF.

Jak je tedy patrné, architektura JSF je v přímé shodě s architektonickým vzorem MVP – dominantní komponentou zde není *Řadič* ani *Presenter*, ale je to právě *Pohled*, který je při konstrukci HTML stránky vytvořen jako první a jednotlivé *backing beany* hrají roli zpřístupnění *Modelu* a dodatečné prezentační logiky. *Backing beany* také mohou přímo ovlivňovat *Pohled*, protože jednotlivé komponenty jsou referencovatelné jako klasické Java objekty přímo z aplikačního kódu, což je další obrovská výhoda oproti JSP (i když trochu navádí k vytváření různých formátovacích „zkratek“ v *Presenteru*, což je principiálně špatně).

JSF nicméně zachází ještě dál. Některé TLD knihovny (např. MyFaces nebo Primefaces) obsahují komponenty, které mohou s *backing beanou* přímo komunikovat pomocí AJAXu. Renderery těchto komponent při vytváření (z principu statického) HTML kódu jednoduše přidají do stránky JavaScript, který umožní obousměrnou komunikaci mezi serverem a webových prohlížečem klienta. *Pohled* se pak částečně přesouvá do webového prohlížeče a programátorům je umožněno vytvářet velice rychle robustní interaktivní webové grafické rozhraní, aniž by napsali jedinou řádku HTML nebo JavaScriptového kódu.

5.5 Shrnutí

V předchozích kapitolách jsme se seznámili s architektonickými postupy uplatnitelnými při návrhu rozsáhlých webových aplikací a distribuovaných informačních systémů a nejpoužívanějšími frameworky, které tyto postupy implementují pro platformu postavenou na programovacím jazyce Java. Definovali jsme základní architektonické pojmy jako je *třívrstvá architektura*, *doménový model* a bude žádoucí se těchto termínů a postupů držet i nadále.

Nicméně dosud jsme se zabývali architekturou, která sledovala jediný cíl – poskytnutí webového rozhraní čitelného a použitelného pro lidského uživatele. Nyní je třeba si položit otázku, jak mohou

s webovými aplikacemi komunikovat jiné systémy nebo klienti, kteří neumožňují zobrazovat webové stránky nebo je z nějakého důvodu nežádoucí či nepraktické těmto klientům webové rozhraní poskytovat.

6 Architektura orientovaná na služby

Architektura orientovaná na služby (*Service-oriented architecture*, zkratkou SOA) je obecný architektonický vzor založený na spolupráci navzájem nezávislých služeb [7]. Motivací pro vytvoření SOA byla rostoucí náročnost na udržování spolupráce různých vnitropodnikových systémů. Díky jejich platformní závislosti a navzájem nekompatibilních programových rozhraní bylo jen velmi obtížně zajistit vzájemnou komunikaci těchto systémů.

6.1 Služba v SOA

Služba je určitá část funkčnosti aplikace, která je zpřístupněná pomocí definovaného rozhraní. Každý jednotlivý informační systém může poskytovat sadu služeb svému okolí. Za službou v SOA stojí většinou poměrně velké množství programového kódu a jakkoliv je princip volání služeb podobný volání metod, probíhá zde na vyšší úrovni granularity.

Rozhraní služeb také není závislé na implementaci hostitelského systému. Hranice systému se od programových rozhraní posunují dále od klasického programového API k jasně definovanému rozhraní, ve kterém daná služba data produkuje či přijímá bez ohledu na to, v jaké technologii je tato služba implementována.

Ve většině případů se pak pro komunikaci mezi službami díky vysoké prostupnosti sítí, platformní nezávislosti a přenositelnosti využívá HTTP protokol. Přenášený formát dat může být v podstatě libovolný, pokud všichni aktéři komunikace tento formát podporují. Nicméně zdaleka nejvyužívanější jsou formáty postavené (zejména z důvodu transparentnosti a multiplatformnosti) na bázi XML. Takové služby se poté zpravidla označují jako služby webové (*Web Services*). Takové služby se pak obvykle jednoznačně identifikují na základě URL adresy.

6.2 SOA a třívrstvá architektura

Vícevrstvá architektura se SOA spolu navzájem velice úzce souvisí. Jednotlivé systémy jsou obvykle distribuované, vícevrstvé aplikace s prezentační, aplikační a perzistentní vrstvou. Nicméně veškerá funkcionalita takové aplikace je vystavena prostřednictvím služeb, nebo je vytvořeno rozhraní, které tyto služby navenek poskytuje.

V SOA aplikacích se typicky neudržuje stav konverzace s klientem, komunikace obvykle probíhá způsobem dotaz-odpověď. Tím se nápadně blíží jednoduchému HTTP požadavku s tím rozdílem, že URL požadavku tentokrát neidentifikuje HTML stránku, ale přímo konkrétní webovou službu.

Je tedy zcela jistě možné (v případě předchozí dobré dekompozice problému) rozhraní webových služeb implementovat za využití architektonického vzoru MVC na prezentační vrstvě, kdy jediný rozdíl nastane v *Pohledu*, který místo zobrazitelného HTML bude generovat XML dokument o předem daném schématu.

6.3 Výhody SOA

Díky bezstavosti a slabému provázání komponent je možné služby skládat do větších celků, což ideálně splňuje potřebu podniků pokrýt a integrovat své vnitropodnikové procesy pomocí informačního systému. Jednotlivé komponenty je možné při výpadku velice rychle nahradit nebo zavést

centrální registr služeb, který umožní vyhledávání aktivních služeb a získání jejich popisu (mechanismus *Universal Description, Discovery and Integration*, zkratkou UDDI).

6.4 Protokoly komunikace v SOA

Již dříve jsme si řekli, že data jsou v SOA architektuře obvykle přenášeny pomocí XML dokumentů. Tyto lze ale dále strukturovat podle předem definovaných schémat a pravidel a vytvářet komunikační protokoly, který jsou na XML formátu založené.

6.4.1 Simple Object Access Protocol (SOAP)

SOAP je standardizovaný protokol pro výměnu dat mezi webovými službami [8], který je obvykle použit spolu s HTTP protokolem. Zprávu zde reprezentuje jednoduchý XML dokument, který má kořenový element *Envelope*. V této obálce jsou pak uzavřeny dva elementy *Header* (hlavička) a *Body* (tělo).

Hlavička se zpravidla používá pro přenos pomocných informací pro zpracování zprávy – například autorizaci klienta. V těle zprávy se přenáší identifikátor volané služby a parametry zprávy, resp. návratové hodnoty služby. SOAP používá jmenné prostory pro identifikování jednotlivých částí XML zprávy. Je třeba zdůraznit, že SOAP protokol je orientován procedurálně a funguje na principu RPC (*Remote Procedure Call*). Komunikace může být v SOAP stavová.
Typický SOAP požadavek se zasílá v těle HTTP požadavku. Používá se přitom metoda POST, která podle [9] dovoluje posílat data v těle HTTP požadavku. Nicméně ten může mít i další metody, které je možné využít. Z toho by mělo být patrné, že klasická SOAP komunikace vytváří vlastní protokol nad klasickým HTTP a nesnaží se jej nijak dále využít. HTTP protokol je přitom dostatečně robustní na to, aby bylo možné jej rozšířit až do úrovně protokolu určeného pro webové služby.

6.4.2 Representational State Transfer (REST)

REST je architektura rozhraní navržená pro distribuované prostředí. REST je narozdíl od SOAP orientován datově, nikoli procedurálně: rozhraní webových REST služeb je použitelné pro jednotný a snadný přístup k tzv. *zdrojům* (datům nebo stavům aplikace, pokud je lze popsat konkrétními daty). Všechny zdroje jsou jednoznačně identifikovatelné pomocí URI a jsou reprezentovány (a přenášeny) pomocí různých datových formátů (XML, JSON nebo jiného formátu, který je možné odeslat pomocí HTTP).

REST definuje základní CRUD(*Create-Read-Update-Delete*) metody pro manipulaci s těmito zdroji (metody *mění stav* zdroje). Tyto metody jsou poté realizovány pomocí odpovídajících HTTP metod (*GET*, *POST*, *PUT*, *DELETE*). Je tedy patrné, že zatímco SOAP používal HTTP pouze jako jeden z možných komunikačních kanálů, REST na architektuře HTTP přímo staví a dále ji rozšiřuje.

Narozdíl od SOAP neexistují žádné specifikace kladené na strukturu dat. REST je také koncipován jako bezestavový. Jednotlivé implementace REST rozhraní (tedy webové služby) se poté při splnění těchto podmínek obvykle označují jako *RESTful*.

SOAP definuje vlastní způsoby autorizace, REST využívá HTTP autorizace. Autorizační tokeny ale musí být odeslány s každým požadavkem, jinak není splněna podmínka bezestavosti (webová služba si o klientovi nesmí nic „pamatovat“).

6.4.3 Srovnání REST a SOAP

Učinit objektivní srovnání těchto dvou protokolů není zcela možné už jen kvůli tomu, že se oba od sebe odlišují samotnou svou koncepcí. Ač je to dnes populární názor, REST není vždy tou správnou volbou. Sice je implementačně mnohem méně náročný a pouze rozšiřuje standardní HTTP protokol, díky jeho „datacentrismu“ a omezené množině CRUD operací může být implementačně

příliš svazující. Také bezestavost komunikace se může ukázat jako velký problém. Jednoduchost REST rozhraní může být výhoda a slabina zároveň, vždy záleží na případu implementace. Obecně lze říci že REST je velmi silný ve spojení s technologiemi jako je AJAX nebo ke vzdálené správě databáze (k čemuž vybízí už jen základní CRUD matice operací).

SOAP je robustnější a díky stavové komunikaci a procedurální orientaci je mnohem vhodnější k realizaci komunikace mezi dvěma netriviálními informačními systémy, což je ovšem vykoupeno potřebou mnohem rozsáhlejší implementace. Navíc funguje stejně dobře i na jiných síťových vrstvách a není třeba se omezovat na pouhý HTTP protokol.

7 Technologické nástroje pro vývoj

Všechny dříve zmíněné architektonické vzory je možné implementovat „manuálně“ a svépomocí spravovat a udržovat jednotlivé vazby mezi doménovými objekty, mezi vrstvami, komponentami či užitými frameworky. Nicméně takto postavená aplikace je jen obtížně znovupoužitelná a není nutné vlastním kódem řešit problémy, které již někdo vyřešil před námi. V této kapitole si ukážeme postupy, které vedou ke zjednodušení vlastního aplikačního kódu a představíme si framework, který tyto postupy implementuje.

7.1 Aspektově orientované programování

Principem *aspektově orientovaného programování* (zkratkou AOP) je oddělení některých částí kódu, tzv. *průřezových koncernů* (*cross-cutting concerns*), od vlastní aplikační logiky. Jako průřezový koncern můžeme definovat takový kód, který se dotýká mnoha různých částí aplikace (příkladem mohou být například transakční a bezpečnostní algoritmy nebo logování) a není jednoduché ho dekomponovat. AOP se snaží tyto koncerny organizovat do tzv. *aspektů*.

AOP není náhrada jiných programovacích technik jako je např. *objektově orientované programování*, slouží spíše jako architektonický doplněk k již zavedeným technikám a uplatní se zejména tam, kde tyto techniky ve snaze dobře dekomponovat kód již nestačí nebo je jejich uplatnění nešikovné.

Aspekty upravují chování programu při spouštění metod, přístupu k atributům třídy nebo při vyhození výjimky. Tato místa se označují jako přípojné body (*joinpoints*). Samotná úprava chování aplikačního kódu (realizovaná opět jako prostý blok kódu) se poté označuje jako *advice*. V Javě se pak označení přípojných bodů v aplikačním kódu obvykle řeší pomocí *anotací*.

7.1.1 Java Anotace

Anotace jsou vlastně jenom textové značky v kódu o určitém předepsaném formátu (anotací je i notoricky známé `@Override`), jejichž cílem je nějakému programovému primitivu přiřadit příznak nebo metadata. Anotace je možné vyhodnocovat při překladu nebo při samotném běhu programu (to ovšem vyžaduje speciální *classloader*). Ve spojení s AOP mohou identifikovat ta programová primitiva, kterým se má přiřadit aspekt (ten je identifikovaný samotnou anotací). I bez využití AOP je možné vytvářet anotace vlastní pro účely např. dokumentace, statického generování kódu atp.

Anotacím je možné vytvářet parametry a těm při použití anotace předávat hodnoty, což jejich možnosti dále rozšiřuje (je možné tyto hodnoty předávat aspektům, které jsou anotacemi identifikovány).

7.2 Spring Framework

Cílem této kapitoly je představit framework, který umožňuje spojit většinu uvedených technologií a postupů do jednoho kompaktního celku – Spring Framework. Tento velice populární modulární aplikační framework umožňuje velice rychle vytvářet rozsáhlé aplikace za využití principu *Inversion of Control*, kdy je framework za běhu aplikace zodpovědný za vytvoření a provázání objektů

implementované aplikace a na programátorovi závisí pouze samotná implementace aplikační logiky [10]. To je možné díky pokročilým programovacím technikám, jako je například *reflexe*. Nicméně hlavní část Spring Frameworku stojí právě na anotacích a AOP.

Jednotlivé komponenty aplikační logiky, jako jsou např. POJO objekty, se doslova „zaregistrují“ do aplikačního frameworku a tento s těmito komponentami dále manipuluje. Je možné si představit například implementaci návrhového vzoru MVC, kdy programátor implementuje všechny tři komponenty *Pohledu*, *Modelu* a *Řadiče*, aniž by tyto na sebe měly navzájem jakoukoliv programovou vazbu nebo dědily nějakou generickou implementaci. Aplikační framework se pak za běhu aplikace postará o jejich vzájemné provázání. Za využití reflexe a AOP Spring Framework dále *injektuje* vlastní aplikační kód (neboli *aspekty*) do zdrojového kódu vytvářené aplikace, a to buď na základě XML konfigurace, nebo pomocí již zmíněných anotací.

Mezi části Spring Frameworku patří komponenty pro ORM mapování, testovací nástroje, integraci uživatelských komponent, modulárně orientovaný vývoj, webové aplikace a webové služby, bezpečnost na základě rolí a řada dalších. Podrobný popis těchto komponent jde dalece nad rámec této práce a v případě nutnosti budou jednotlivé komponenty v dalším textu stručně popsány.

8 Závěr

Nyní máme obecný přehled o tom, jaké architektonické vzory můžeme využít pro programování rozsáhlých distribuovaných systémů a známe široce využívané technologie stavějící na Java platformě, které tyto postupy implementují. Využití zde uvedených technologií však záleží čistě na programátorovi systému.

Pro další studium architektonických vzorů uplatnitelných v rozsáhlých systémech lze zejména doporučit [3], kde jsou uvedeny i případy tzv. návrhových antivzorů. Pro lehké seznámení s vývojem moderních Java EE aplikací založených na frameworku Spring pak postačí [10]. Obsahem této knihy je pak i lehké seznámení a příklady využití technologií jako je REST, JSP atp. Spring Framework sám nutí programátora do používání „správných“ návrhových vzorů, a proto lze tuto knihu doporučit zejména programátorům s žádnými předchozími zkušenostmi se Spring Frameworkem a zde uvedenými návrhovými vzory obecně.

Literatura

[1] DAY, J.D. a H. ZIMMERMANN. The OSI reference model. *Proceedings of the IEEE* [online]. roč. 71, č. 12, s. 1334-1340 [cit. 2013-04-30]. ISSN 0018-9219. DOI: 10.1109/PROC.1983.12775. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1457043

[2] LIU, Ling a M ÖZSU. *Encyclopedia of database systems.* New York: Springer, c2009, 5 v. (lxix, 3749 p.). ISBN 978-038-7496-160.

[3] FOWLER, Martin. *Patterns of enterprise application architecture.* Boston: Addison-Wesley, c2003, xxiv, 533 p. ISBN 03-211-2742-0.

[4] ELLIOTT, James, Ryan FOWLER a Tim O'BRIEN. *Harnessing Hibernate.* Sebastopol: O'Reilly, 2008, xiv, 363 s. ISBN 978-0-596-51772-4.

[5] JOEL MURACH, Andrea Steelman, Ryan FOWLER a Tim O'BRIEN. *Murach's Java servlets and JSP: training.* 2nd ed. Fresno, CA: Mike Murach, 2008, xiv, 363 s. ISBN 18-907-7444-8.

[6] BERGSTEN, Hans, Ryan FOWLER a Tim O'BRIEN. *JavaServer faces: training.* 2nd ed. Sebastopol, CA: O'Reilly, c2004, xiv, 589 p. ISBN 05-960-0539-3.

[7] HEWITT, Eben, Ryan FOWLER a Tim O'BRIEN. *Java SOA cookbook: training.* 1st ed. Sebastopol, Calif.: O'Reilly, c2009, xxiv, 714 p. ISBN 05-965-2072-7.

[8] SOAP Specifications. *World Wide Web Consortium (W3C)* [online]. 2013 [cit. 2013-05-07]. Dostupné z: http://www.w3.org/TR/soap

[9] HTTP/1.1: Method Definitions. *World Wide Web Consortium (W3C)* [online]. 2013 [cit. 2013-05-05]. Dostupné z: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[10] WALLS, Craig. *Spring in action.* 3rd ed. Shelter Island: Manning, c2011, xxiii, 400 p. ISBN 19-351-8235-8.