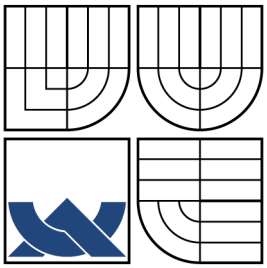
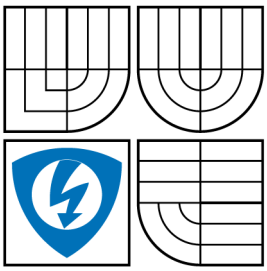


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND
COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

ZABEZPEČENÍ A ARCHITEKTURA SPRÁVCE IPTV VYSÍLÁNÍ

SECURITY AND ARCHITECTURE OF IPTV MANAGER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — BC. DANIEL ŠIMKO
AUTHOR

VEDOUCÍ PRÁCE — ING. RADIM BURGET
SUPERVISOR

BRNO 2008

ZDE VLOŽIT LIST ZADÁNÍ

Z důvodu správného číslování stránek

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami:

1. Pan/paní

Jméno a příjmení: Bc. Daniel Šimko
Bytem: Hrubínova 521, 57201, Polička - Dolní Předměstí
Narozen/a (datum a místo): 25.8.1983, Polička

(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta elektrotechniky a komunikačních technologií
se sídlem Údolní 244/53, 60200 Brno 2
jejímž jménem jedná na základě písemného pověření děkanem fakulty:
prof. Ing. Kamil Vrba, CSc.

(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - ☐ disertační práce
   - ☑ diplomová práce
   - ☐ bakalářská práce

   jiná práce, jejíž druh je specifikován jako .....

(dále jen VŠKP nebo dílo)

Název VŠKP: Zabezpečení a architektura správce IPTV vysílání
Vedoucí/školitel VŠKP: Ing. Radim Burget
Ústav: Ústav telekomunikací
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

- ☑ tištěné formě - počet exemplářů 1
- ☑ elektronické formě - počet exemplářů 1

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2

Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   - ☑ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3

Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

ABSTRAKT

Cílem práce je analyzovat, navrhnout a implementovat aplikaci pro správu IPTV vysílání. Analýza zkoumá požadavky na aplikaci a její výsledek je zpracován v podobě UML diagramu. Návrh je zaměřen na architekturu aplikace a na výběr moderních technologií, které umožní aplikaci realizovat. V návrhu i samotné implementaci je kladen velký důraz na bezpečnost aplikace.

KLÍČOVÁ SLOVA

IPTV, webová aplikace, Java, JEE, Spring, Hibernate, Stripes, Acegi, testování

ABSTRACT

The aim of my thesis is to analyze, propose and implement the application of IPTV Manager.The analysis probes the requirements for the application and its results are displayed in UML diagrams. The proposal is mainly focused on application's architecture and on the choice of modern technologies that faciliate its realisation.The security of the application is highly emphasized in the proposal and in the implementation itself.

KEYWORDS

IPTV, web application, Java, JEE, Spring, Hibernate, Stripes, Acegi, testing

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „ZABEZPEČENÍ a ARCHITEKTURA SPRÁVCE IPTV VYSÍLÁNÍ“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

ÚVOD

Počet IPTV (Internet Protocol TeleVision) uživatelů ve světě úspěšně roste, v polovině roku 2007 služby přenosu TV signálu po IP sítích využilo podle [13] přes 8 miliónů uživatelů, což je téměř trojnásobek oproti roku předchozímu. Tato práce se zabývá návrhem a vývojem systému pro správu IPTV vysílání, který bude implementován pro skupinu Multicast IPTV Research Group [14], která se na ústavu telekomunikací VUT v Brně zabývá výzkumem IPTV pro velmi velké vysílací skupiny. Tento systém se skládá z jednoho IPTV Manažera a libovolného počtu IPTV Serverů (návrh a implementace IPTV Serveru není součástí této práce). IPTV Manažer je webová aplikace, která je nasazena na samostatném serveru a jejímž úkolem je komunikace se zákazníky (umožnit ovlivňovat a sledovat IPTV vysílání) a správa jednotlivých IPTV Serverů. Manažer nedisponuje žádnými video soubory, ty jsou na jednotlivých IPTV Serverech, které mají za úkol spustit požadované vysílání v čase, který jim byl prostřednictvím Manažera naplánován. Zákazník není nikterak schopen komunikovat s IPTV Serverem přímo, ale jen prostřednictvím Manažera.

Jelikož je tento systém úplný prototyp, který ještě není vyzkoušen, bude nutné provést důkladnou analýzu a věnovat zvýšenou pozornost volbě technologií a návrhu architektury aplikace. Dobře navržená architektura v budoucnu usnadní provádění změn a především rozšiřování aplikace. Výběr technologií, které umožňují jednotlivé části architektury realizovat, má na úspěšnost projektu neméně velký vliv jako architektura samotná.

Analýzou, návrhem architektury a výběrem vhodných technologií se zabývají první dvě kapitoly. Další dvě kapitoly jsou věnovány samotnému vývoji aplikace a popisu různých bezpečnostních opatření, které jsou v aplikaci implementovány. Poslední dvě kapitoly obsahují podrobný návod, jak aplikaci nainstalovat na server a popis různých technik testování, které byly v aplikaci použity.

1 ANALÝZA

Při vývoji takovéto komplexní webové aplikace je nezbytným prvním krokem analýza, která v podstatě slouží k doruzumnění zákazníka (zadavatele) a programátora (případně analytika, který se doruzumí s programátorem). V analýze se používá standardizovaný jazyk UML [16], který je velice názorný a snadno pochopitelný i pro zadavatele, který nemá žádnou zkušenost s programováním. První část analýzy bývá zaměřená na samotnou funkcionalitu aplikace, tedy na to, co všechno má požadovaná aplikace umět. K tomu se používají takzvané UML diagramy případů užití (use-case). V tomto diagramu by měla být zakreslena veškerá požadovaná funkčnost aplikace, podporované typy uživatelů (z pohledu oprávnění) a jejich možné činnosti. Diagram by tedy měl aplikaci přesně popisovat a stanovovat její hranice, proto bývá možné na jeho základě předběžně odhadnout i cenu vývoje aplikace.

Podoba diagramu IPTV Manažera se drobně měnila a doplňovala v průběhu několika konzultací se zadavatelem (vedoucím této práce). Konečná verze, podle které byla aplikace naprogramována je na obr. 1.1.

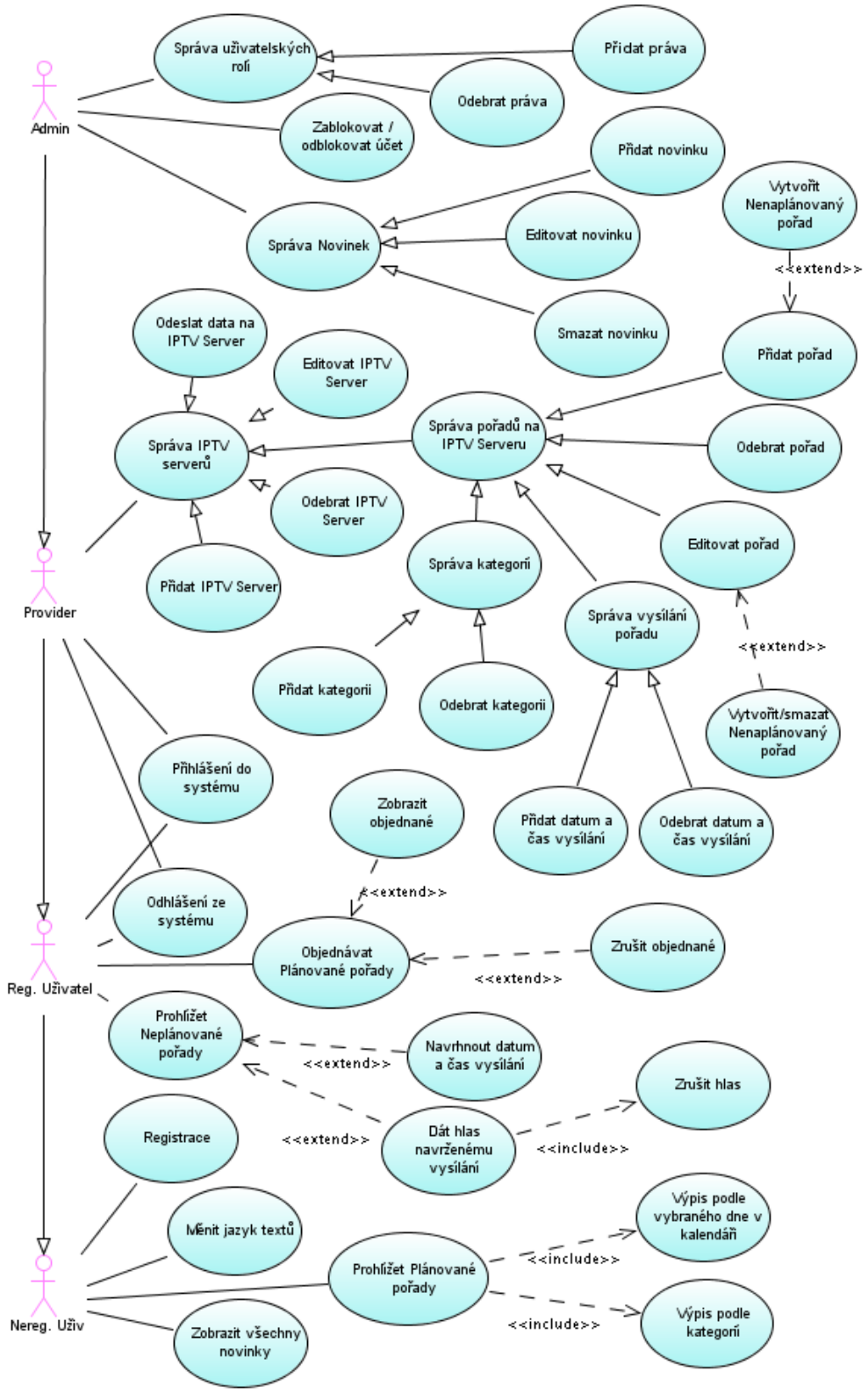
1.1 Požadavky na funkcionalitu

Z diagramu případů užití na obr. 1.1 je vidět, že aplikace podporuje čtyři skupiny uživatelů, kterými jsou veřejní internetoví uživatelé, registrovaní uživatelé (zákazníci), správci IPTV Serverů a administrátoři. Vertikální šipky mezi skupinami uživatelů znázorňují vztah dědičnosti, tedy že např. registrovaní uživatelé mohou vykonávat všechny akce jako neregistrovaní, plus mají další akce, které neregistrovaní nemají. Všichni uživatelé přistupují k aplikaci pomocí tenkého klienta, kterým je webový prohlížeč. V následujících kapitolách budou blíže popsány jednotlivé role.

1.1.1 Veřejní internetoví uživatelé

Neregistrovaní uživatelé mohou prohlížet aktuální nabídku plánovaných pořadů. Plánované pořady jsou ty, které mají určen alespoň jeden datum a čas vysílání. Tyto pořady mohou být zobrazeny podle časů vysílání v konkrétní den, který je možné vybrat v kalendáři (podobně jako TV program), nebo podle kategorií (žánru pořadů). Po kliknutí na název pořadu se zobrazí jeho detail, kde je stručný popis, cena, obrázky, atd.

Dále uživatelé mohou zobrazovat archív novinek, měnit lokalizaci celé aplikace (jazyk textů, zobrazení datumů, atd.) a zaregistrovat se.



Obr. 1.1: UML diagram případů užití IPTV Manažera.

1.1.2 Registrovaní uživatelé

Po registraci a přihlášení do systému se uživatel stává registrovaným. Což mu navíc od neregistrovaného umožňuje prohlížet katalog nenaplánovaných pořadů, u kterých má možnost navrhnout datum a čas vysílání a hlasovat pro již navržené vysílání. U každého pořadu je zobrazen potřebný počet hlasujících, který když je splněn, zkopíruje se pořad do naplánovaných.

Přihlášenému uživateli je v detailu pořadu zobrazeno tlačítko, pomocí kterého si může pořad objednat a v konkrétní čas na určité adrese sledovat. Tyto objednané pořady si může zobrazit a případně objednávku zrušit.

1.1.3 Správci IPTV serverů

Správci mohou přidat do systému nový IPTV Server nebo editovat či odstranit server již obsažený v systému. Přidání serveru obnáší zadání jeho jména a adresy na které bude Server s Manažerem komunikovat.

Pro jednotlivé IPTV Servery poté mohou přidávat, editovat nebo mazat pořady. U pořadů lze nastavit libovolný počet časů vysílání, nebo naopak konkrétní čas zrušit. Lze také určit, jestli bude pořad i v nenaplánovaných pořadech a případně nastavit potřebný počet uživatelů, kteří se musí schodnout na čase vysílání.

Kvůli jednotnému pojmenovávání kategorií a usnadnění tvorby podkategorií je využit systém správy kategorií společný pro všechny IPTV Servery.

1.1.4 Administrátoři

Kromě všeho co bylo zmíněno u předchozích typů uživatelů smějí měnit oprávnění uživatelů (např. z registrovaného uživatele udělat správce nebo naopak), zablokovat účet uživatele (kterému pak není umožněno se přihlásit do systému) a psát, editovat či mazat novinky.

1.2 Ostatní požadavky na aplikaci

Kromě samotné funkcionality se analýza musí zabývat i otázkami jako jsou např.:

- požadovaná míra bezpečnosti aplikace,
- nutnost asynchronní komunikace s jinými aplikacemi (messaging),
- poskytnutí rozhraní pro synchronní vzdálenou komunikaci s jinými aplikacemi,
- požadavky na odezvu aplikace, s čímž souvisí vyrovnávání zátěže a clustering,

- požadavek podpory více typů grafických rozhraní, například webového i grafického desktopového,
- nutnost správy transakcí nad více datovými zdroji apod.

Po konzultaci se zadavatelem vyplynulo, že asynchronní komunikace s jinými aplikacemi, více typů grafických prostředí ani clustering na více serverů nebude potřeba. IPTV Manažer poběží na jediném serveru v servletovem kontejneru Tomcat. Na stejném serveru bude i jediná databáze MySQL, takže správu transakcí nad více datovými zdroji také nebude potřeba řešit. Aplikace však bude poskytovat rozhraní pro vzdálenou komunikaci s IPTV Serverem.

1.2.1 Bezpečnost aplikace

Na bezpečnost je v aplikaci kladen velký důraz, měla by být zajištěna následovně:

- veškerá komunikace s uživateli i jednotlivými IPTV Servery zabezpečeným kanálem pomocí protokolu https,
- v databázi neukládat hesla v čitelné podobě, ale ve formě otisku - hash s přídavkem soli,
- kontrolovat všechny vstupní data od uživatelů,
- zajistit odolnost proti útoku přepsáním parametrů v URL (Uniform Resource Locator),
- zobrazení IP adresy, datumu a času posledního přihlášení do systému,
- potenciálně nebezpečné akce protokolovat do souboru

1.3 Rozsah řešení a hranice systému

Součástí projektu nebude systém řešící finanční stránku věci, tedy jakým způsobem budou zákazníci za službu platit. Každý nově zaregistrovaný zákazník obdrží při registraci určitý kredit zdarma, ze kterého bude moci čerpat.

Způsob jakým bude možnost kredit navýšit není zatím určen. Jedním z řešení jsou populární SMS. Tento způsob je ale neefektivní, podle [15], až 80% z částky připadne zprostředkovateli platby. Nejlepším způsobem by pravděpodobně byly platby přes systém PayPal, ten ale není v ČR zatím tolik rozšířený.

2 NÁVRH ARCHITEKTURY A VOLBA TECHNOLOGIÍ

Jako většina složitějších webových aplikací, bude i IPTV Manažer postaven na klasické třívrstvé architektuře, která rozlišuje tyto vrstvy:

- Prezentační vrstva – má na starosti obsluhu požadavků webových klientů a generování odpovědí ve standardním webovém formátu (X)HTML.
- Aplikační vrstva – někdy také označovaná jako servisní vrstva slouží pro implementaci logiky aplikace. Obsahuje tedy jednotlivé aplikační algoritmy.
- Datová vrstva – slouží k práci s úložištěm dat, je v ní tedy pouze naimplementovaná logika pro zacházení s daným datovým zdrojem.

Každá vrstva poskytuje navenek určité rozhraní, přes které s ní může druhá vrstva komunikovat, díky tomu není žádný problém změnit v budoucnu implementaci vrstvy (např. poskytovatele datového úložiště). Izolace vrstev přináší také výhodou v podobě snazšího testování.

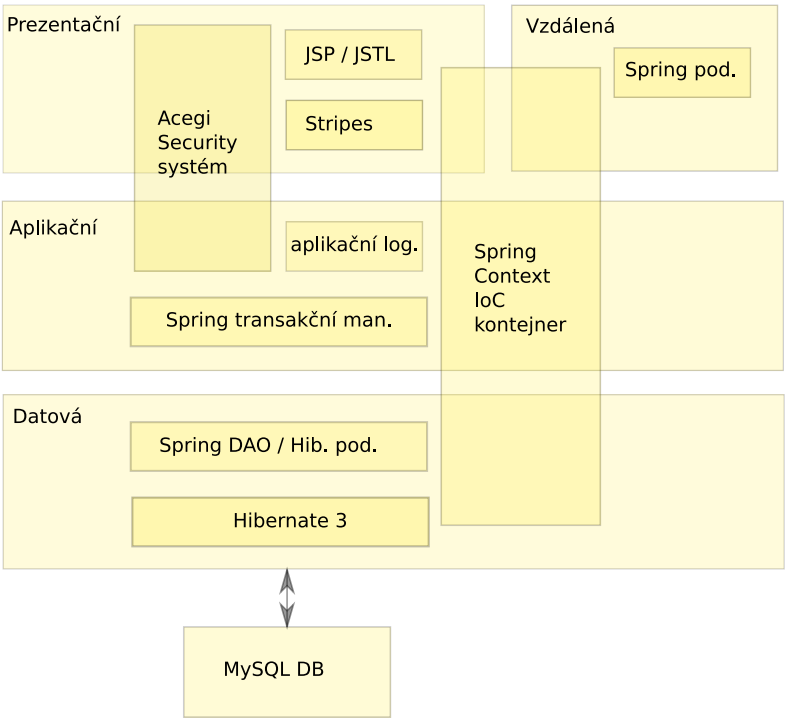
Na obrázku 2.1 je vidět výsledná architektura a hlavní technologie zvolené pro implementaci aplikace. Výběr technologií a jejich stručný popis následuje dále v této kapitole.

2.1 Aplikační rámec

Vývoj takovéto vícevrstvé Java Enterprise Edition (JEE) aplikace se prakticky neobejde bez využití aplikačního rámce, jakým je např. Enterprise JavaBeans (EJB) nebo Spring framework. EJB a Spring jsou v současné době nejpoužívanějšími technologiemi pro vývoj Java Enterpise aplikací. Tedy aplikací, v nichž se předpokládá současný přístup více uživatelů k aplikaci, která musí být zabezpečena, musí komunikovat s jinými systémy a mít požadavky na výkonnost atd. [4]

2.1.1 Enterprise JavaBeans (EJB)

EJB technologie je oficiální část API (Application Programming Interface - rozhraní pro programování aplikací) JEE vyvíjená firmou Sun Microsystems. EJB vznikla za účelem poskytnout standardní cestu k řešení typických problémů, se kterými se setká každá enterprise aplikace. Těmito problémy je myšleno např. perzistence (ukládání dat do DB), transakční zpracování, bezpečnost atd. EJB tedy specifikuje detaily, jak má aplikační server (EJB kontejner) tyto služby poskytovat. Konkrétních implementací aplikačních serverů, open-source i komerčních, je dnes celá řada.



Obr. 2.1: Architektura aplikace.

Na základě vysledků analýzy je tedy jasné, že EJB nebude možno pro tento projekt použít, protože IPTV Manažer nepoběží v aplikčním serveru, ale pouze v servletovém kontejneru Tomcat. Navíc podle řady JEE architektů [15] je vývoj s EJB výrazně nákladnější a pomalejší, než je možno dosáhnout s moderním rámcem Spring. Výsledné aplikace jsou náročnější na údržbu a rozšiřování, jejich kód je složitější a hůře testovatelný.

2.1.2 Aplikační rámec Spring

Je open-source modulární Java/JEE aplikační rámec, který byl autorem Rodem Johnsonem navržen jako reakce na EJB, které byly ve svých dřívějších verzích (EJB 1.0 až 2.1) dosti těžkopádné a vyžadovaly psaní mnoha kódu, který nesouvisel s cílem aplikace. EJB 3.0 však naopak přebraly pár myšlenek ze Springu, což vedlo k podstatnému zjednodušení programování pod touto technologií. Spring narozdíl od EJB nediktuje vývojáři architekturu aplikace ani nenutí dědit žádnou specifickou třídu. Místo toho podporuje POJO[1] programovací model, díky čemuž nevzniká žádná závislost aplikačního kódu na rámci. Spring totiž poskytuje důmyslný konfigurační

[1]Termín POJO je zkratkou Plain Old Java Objects, což volně přeloženo znamená "staré dobré Java objekty". Termín POJO formuloval Martin Fowler a označuje objekty, které nejsou svázané technologicky specifickým rozhraním.

management pro POJOs, který je založen na návrhovém vzoru IoC - Inversion of Control (Obrácení řízení). Spring jde však ještě dále, než jak je tomu u vzoru IoC, proto byl zaveden nový pojem DI – Dependency Injection (Injektáž závislostí). Zjednodušeně řečeno lze postup při použití DI popsat takto: Mějme závislost objektu a na objektu B, jinými slovy objekt a obsahuje odkaz na objekt B. Při použití DI budou při startu kontejneru, který je spravuje, oba objekty vytvořeny a v objektu a bude inicializován odkaz na objekt B. [15]

Rámec Spring podporuje všechny vrstvy třívrstvého modelu. Ke službám, které nabízí, patří např.

- centralizovaný, automatizovaný způsob konfigurace aplikace,
- jednotný procedurální i deklarativní transakční management,
- pokročilá procedurální i deklarativní správa zabezpečení,
- podpora všech nejpoužívanějších perzistentních technologií (JDBC, Hibernate, Toplink atd.),
- integrovaný AOP - Aspect Oriented Programming (programování orientované na aspekty) rámec,
- flexibilní MVC - Model View Controller (model-pohled-řadič) webový rámec,
- správa provázání a závislostí objektů, pooling objektů a další.

Stěžejními technikami, které poskytování těchto služeb umožňují, jsou již zmíněné AOP a návrhový vzor DI [5].

2.2 Zabezpečení aplikace

Součástí aplikace je komplexní řízení autentizace a autorizace uživatelů systému. Vzhledem k tomu, že jde o tzv. napříč jdoucí (crosscutting) koncept, je pro řešení této problematiky vhodné využít specializovanou technologii, jako je např. JAAS nebo Acegi Security.

2.2.1 Authentication and Authorization Service (JAAS)

JAAS byl představen jako volitelný balíček do Java 2 SDK (Software Development Kit) verze 1.3 a od verze 1.4 je přímo integrován do SDK. JAAS je oficiální standardní API, které umožňuje získání identity uživatele, kterou lze následně použít pro autorizaci jeho akcí. Většina webových a EJB kontejnerů poskytuje určitou formu podpory pro JAAS.

Naproti tomu Acegi Security je “de facto“ standart pro aplikace postavené na technologii Spring. Implementace požadované míry zabezpečení bude tedy s tímto rámcem jednodušší, než by byla s technologií JAAS.

2.2.2 Acegi Security

Nedávno se dokonce Acegi Security stal oficiálním projektem Spring portfolia a nejaktuálnější verze už nese název Spring Security. V této aplikaci však bude využita předchozí verze tohoto mocného open source bezpečnostního rámce, proto bude nazýván zažitým jménem Acegi. Acegi se skládá ze sady rozhraní a tříd konfigurovatelných pomocí DI a stejně jako rámec Spring, klade důraz na možnost vyměnitelnosti téměř všech částí systému. Díky tomu umožňuje volbu mezi různými strategiemi při implementaci bezpečnosti.

Acegi umožňuje pomocí řady webových filtrů nakonfigurovat například následující funkcionality[7]

- Vyzvat uživatele k vyplnění přihlašovacích údajů před zpřístupněním zabezpečeného zdroje (např. zobrazením stránky s formulářem pro zadání přihlaš. údajů).
- Autentizovat uživatele ověřením jeho bezpečnostního tokenu (např. hesla).
- Kontrolovat zda autentizovaný uživatel má oprávnění přístupu k zabezpečenému zdroji (autorizace).
- Přesměrovat úspěšně autentizovaného a autorizovaného uživatele na požadovaný zabezpečený zdroj.
- Zobrazení příslušné stránky „přístup odepřen“ uživateli, který nemá dostatečné oprávnění k zabezpečenému zdroji.
- Zapamatovat úspěšně autentizovaného uživatele na straně serveru a nastavení cookie na straně klienta. Příští autentizace může být vykonána použitím cookie, bez zobrazování přihlašovací stránky.
- Uložení autentizačních informací na straně serveru do objektu session pro následující požadavek zabezpečeného zdroje.
- Vybudování a udržení cache pro uchování bezpečnostních informací na straně serveru, čímž je dosaženo zlepšení výkonu.
- Zrušení bezpečnostní informace v objektu session po odhlášení uživatele.

- Přihlašovací údaje a oprávnění uživatelů mohou být čteny z relační databáze nebo z mnoha jiných podporovaných systému (XML, LDAP, CAS, ...).

Z výčtu je vidět, že Acegi Security filtry umožňují téměř cokoliv, co bychom mohli od webové aplikace požadovat. Dále rámec Acegi Security např. umožňuje:

- Přepínat mezi nezabezpečeným (http) a zabezpečeným (https) kanálem na základě URL.
- ACL (Access Control List) řízení přístupu k doménovým objektům na základě oprávnění.
- Odstranění objektů z kolekce, pokud uživatel nemá oprávnění na jejich čtení.
- Ukládat heslo v podobě jeho hashe s přídavkem soli.
- Lokalizaci zobrazovaných hlášek (např. důvod neúspěšné autentizace).
- Ochranu proti webovým robotům např. technologií JCAPTCHA.
- Použít knihovny tagů (tag libraries), díky kterým lze snadno dosáhnout toho, že autentizovanému uživateli se zobrazí jenom takový obsah webové stránky, na který má oprávnění.
- Audit a protokolování událostí.

Díky rámci Acegi nebude problém dosáhnout v IPTV Manageru požadované míry bezpečnosti.

2.3 Prezentační (Presentation) vrstva

Jak už bylo zmíněno výše, prezentační vrstva má na starosti obsluhu požadavků webových klientů a generování odpovědí v HTML formátu. Spring k tomuto účelu poskytuje svůj propracovaný MVC webový rámec. Avšak jak už vyplývá z filozofie Springu, je možné použít i kterýkoliv jiný webovy rámec. Pro tento projekt byl vybrán rámec Stripes, který díky svým vlastnostem umožňuje o trochu rychlejší vývoj než s použitím např Spring MVC nebo hodně populárních Struts.

Kromě Stripes jsou základními prvky této vrstvy ještě JSP (Java Server Pages) a JSTL (JavaServer Pages Standard Tag Library).

2.3.1 Stripes

Stripes je poměrně nový open source webový rámec založený na principech, které umožňují vyvíjet webové aplikace jednodušeji a rychleji než jak je tomu u většiny starších rámců. Stripes totiž narozdíl od nich nepotřebuje žádné konfigurační soubory, využívá anotace a generics, zavedené v Java 5.0, a konvence (princip convention-over-configuration, zavedený v Ruby-On-Rails) [8].

2.4 Aplikační (Business) vrstva

Tvoří prostředníka mezi vrstvou prezentační a vrstvou datovou. Obsahuje tzv. business logiku aplikace. Spring je této vrstvě velmi užitečný zejména správou životního cyklu a provázání aplikačních objektů, které je založeno na Dependency Injection (Injektáž závislostí). Dále mocným transakčním managementem a poskytnutím možnosti využít aspektově orientované programování AOP. Spring AOP i transakční management je podrobněji popsán dále v této kapitole.

2.5 Datová (Persistence) vrstva

Bez této vrstvy by byl kód přistupující k datům roztroušen po celé aplikaci, což by vedlo k jeho duplikaci a obtížnému nebo nemožnému udržování. Datová vrstva tedy obsahuje funkčnost pro přístup k datovému úložišti (ať už k relační databázi, souborovému systému atp.). Aplikace postavené nad aplikačním rámcem Spring obvykle využívají v této vrstvě návrhového vzoru DAO - Data Access Object (Objekt přistupující k datům). Návrhový vzor DAO patří k základním JEE vzorům a zjednodušeně lze popsat tak, že zavádí rozhraní pro práci s daty a zároveň implementuje potřebnou logiku. Díky DAO, který koncentruje všechnu logiku pro získání/ukládání dat, je aplikační vrstva naprosto odstíněna od zdroje nebo zdrojú dat. DAO implementace může používat buď klasický JDBC nebo některý z ORM nástrojů, jakým je např. Hibernate. Spring plně podporuje jak variantu s čistým JDBC, tak většinu z dnes používaných ORM nástrojů. Práce s JDBC je však poměrně komplikovaná a zdlouhavá, protože musíme řešit problém propojení relačního a objektového světa. Proto tato aplikace využívá nástroj Hibernate, který poskytuje techniku pro objektově-relační mapování doménových tříd na tabulky relační databáze.

2.5.1 Hibernate

Hibernate je open source objektově - relační mapovací (ORM) nástroj. ORM je v podstatě převod řádku tabulky na instanci odpovídající třídy a naopak, navíc

sloupce obsahující cizí klíče ve vytvořeném objektu jsou nahrazeny odkazovaným objektem, vztahy 1:n a n:n převedeny na kolekce objektů. Kromě ORM Hibernate poskytuje v současné době nejmocnější funkce pro objektově relační dotazování s třemi dotazovacími prostředky. Jsou jimi Hibernate Query Language, které umožňuje objektově orientované sestavování dotazů. Hibernate Criteria Query API, které je velmi výhodné pro funkcionalitu jako je např. vyhledávání. A v neposlední řadě široká podpora pro psaní dotazů v nativním SQL dialektu určité databáze. Hibernate dále poskytuje mechanismy pro polymorfní perzistenci, kešování, transakční zpracování a mnoho další funkcionality [3].

2.6 Vrstva vzdáleného volání (Remote service)

Jak vyplývá z analýzy v předchozí kapitole, IPTV Manažer potřebuje tuto vrstvu k zajištění komunikace s jednotlivými IPTV Servery, které poběží na jiných počítačích.

Přestože je Spring zaměřen hlavně na podporu aplikační vrstvy (správu volné vazby mezi komponenty stejného procesu), poskytuje mocnou podporu pro vrstvu vzdáleného volání. Oproti tradičnímu aplikačnímu rámci EJB umožňuje export i konzumaci vzdálených služeb poměrně snadno. Spring totiž podle filozofie, která se line celým rámcem, nabízí konzistentní POJO model pro jakýkoliv z následujících podporovaných protokolů:

- Hessian - malinký binární protokol založený na HTTP. Je jazykově nezávislý, však převážně využívaný v jazyce Java. Pochází z dílny firmy Caucho, která je mimo jiné i výrobcem známého aplikačního serveru Resin.
- Burlap - malinký XML protokol také založený na HTTP. Stejně jako jeho sourozenec Hessian, je jazykově nezávisly, avšak převážně Java orientovaný.
- HTTP invoker - vlastní řešení Springu, také založené na HTTP, ale na rozdíl od Hessianu a Burlapu využívá Java serializaci, což má jisté výhody, které budou zmíněný dále v této práci.
- RMI (Remote Method Invocation) - standardizované Java/JEE řešení využívající protokol JRMP nebo IIOP.
- WSDL/SOAP přes JAX-RPC - pro standardizované WSDL webové služby (Web Services) se SOAP přes HTTP.

Vzdálená komunikace podle scénáře klient-server bývá obvykle založená na HTTP, díky čemuž nemá problémy projít standardně nastaveným firewallem.

2.6.1 Spring http invoker

Ke komunikaci s IPTV serverem byl původně vybrán protokol Hessian, v průběhu vývoje aplikace se však ukázalo, že vhodnější řešení je http invoker. Ten narozdíl od Hessianu používá Java serializaci, která se používá i v nástroji Hibernate a tudíž to nevedlo ke konfliktům při inicializaci líně (lazy) načítaných objektů.

2.7 Společné aspekty (crosscutting concerns)

Některé aspekty se prolínají přes jednotlivé vrstvy a není proto vhodné je natvrdo "zadrátovat" do aplikace. Těmito aspekty jsou např.:

- transakce,
- bezpečnost,
- audit (zaznamenávání uživatelských akcí do databáze),
- protokolování (např. logování volání aplikačních metod a jejich návratových hodnot, logování výjimek, ...)

Pokud jsou tyto aspekty tradičně "zadrátované" do aplikace, vede to k tomu, že jejich kód je roztroušen po celé aplikaci. Např. protokolování musí být zaneseno do každé aplikační metody. To samozřejmě přináší spoustu nevýhod v podobě:

- duplicitě kódu, která přináší významné problémy s udržováním kódu,
- aplikační (bussines) logika dělá několik věcí (porušení zásad OOP - objektově orientovaného programování),
- snadno se na takový kód zapomene, vzniká tedy nespolehlivá a křehká aplikace

Daleko výhodnější je proto neinvazivních metod jako jsou metadata (anotace, popisné XML) a Aspektově Orientované Programování.

2.7.1 Aspektově orientované programování (AOP)

AOP v podstatě řeší to, jak elegantně dosáhnout vyčlenění opakovaných, napříč jdoucích částí kódu do tzv. pokynů (advices). Ty jsou vyčleněny do samostatných tříd a deklarativním způsobem je určeno, kam všude se příslušný pokyn má aplikovat (na místa zvaná point-cuts). Použitý nástroj (např. AspectJ) pak zajistí aplikování příslušného pokynu čímž vzniká tzv. aspekt. [1]

V aplikacích využívajících Spring lze použít jednak vlastního AOP řešení rámce Spring, které je postaveno na standardních dynamických proxy jazyka Java, nebo plnohodnotný AOP open-source nástroj AspectJ. V této aplikaci bohatě postačí AOP řešení Springu.

2.7.2 Transakce

Transakce je pracovní jednotka skládající se ze sady operací, proti jednomu nebo několika zdroji (např. databázi), které musejí být dokončeny jako celek. Pokud jedna z operací není v pořádku vykonána, celá transakce se "rollbackne", což má za následek, že všechny operace v transakci jsou stornovány a vráceny do původního stavu.

Jednou z obrovských zbraní Springu je elegantní rámec pro deklarativní správu transakcí, který je postaven na možnostech AOP. Díky tomuto rámci je jakémukoliv POJO (např. aplikačnímu objektu) umožněno využít výhod transakčního chování. Stačí pouze nadefinovat toto chování a přidat transakční proxy do aplikačního kontextu aplikace (konfiguračního souboru). Není potřeba modifikovat nebo dopisovat žádný kód do aplikačního objektu.

Transakce budou v aplikaci zadefinovány (deklarativním způsobem) na úrovní aplikační vrstvy a propagovány směrem dolu (do datové vrstvy).

2.7.3 Audit a protokolování

K protokolování chování aplikace a auditu chování uživatelů bude využit open source logovací nástroj log4j, který poskytuje velmi rychlé a snadno konfigurovatelné logovací API. Zanesení tohoto API přímo do všech metod, které budou protokolovány, je velmi nevýhodné z důvodů uvedených výše. Proto zde bude také použito AOP.

3 VÝVOJ APLIKACE

Jeden z dalších hlavních důvodů proč byl IPTV Manažer postaven nad aplikačním rámcem Spring je, že umožňuje návrh a vývoj aplikace užitím tradičních OOP zásad mnohem snadněji, než při použití standardních JEE technologií. Jak již bylo zmíněno, Spring je neinvazivní, nenutí dědit naše objekty žádné svoje třídy ani implementovat svá rozhraní. Navíc díky DI, který za nás řeší vazby mezi komponentami, máme poměrně velkou volnost návrhu aplikace.

3.1 Budování datové vrstvy

Vývoj aplikace začíná tím, že se vytvoří takzvaný Doménový objektový model (DOM - Domain Object Model), který zapouzdřuje data a chování aplikace. S doménovým modelem pak pracují všechy vrstvy aplikace. Datovou vrstvu zajímají ty objekty z doménového modelu, které slouží k uchování dat. Tyto doménové třídy se pomocí ORM nástroje Hibernate namapují na tabulky relační databáze. Mapovaným třídám se pak říká perzistentní třídy (nebo entity).

3.1.1 Doménový model

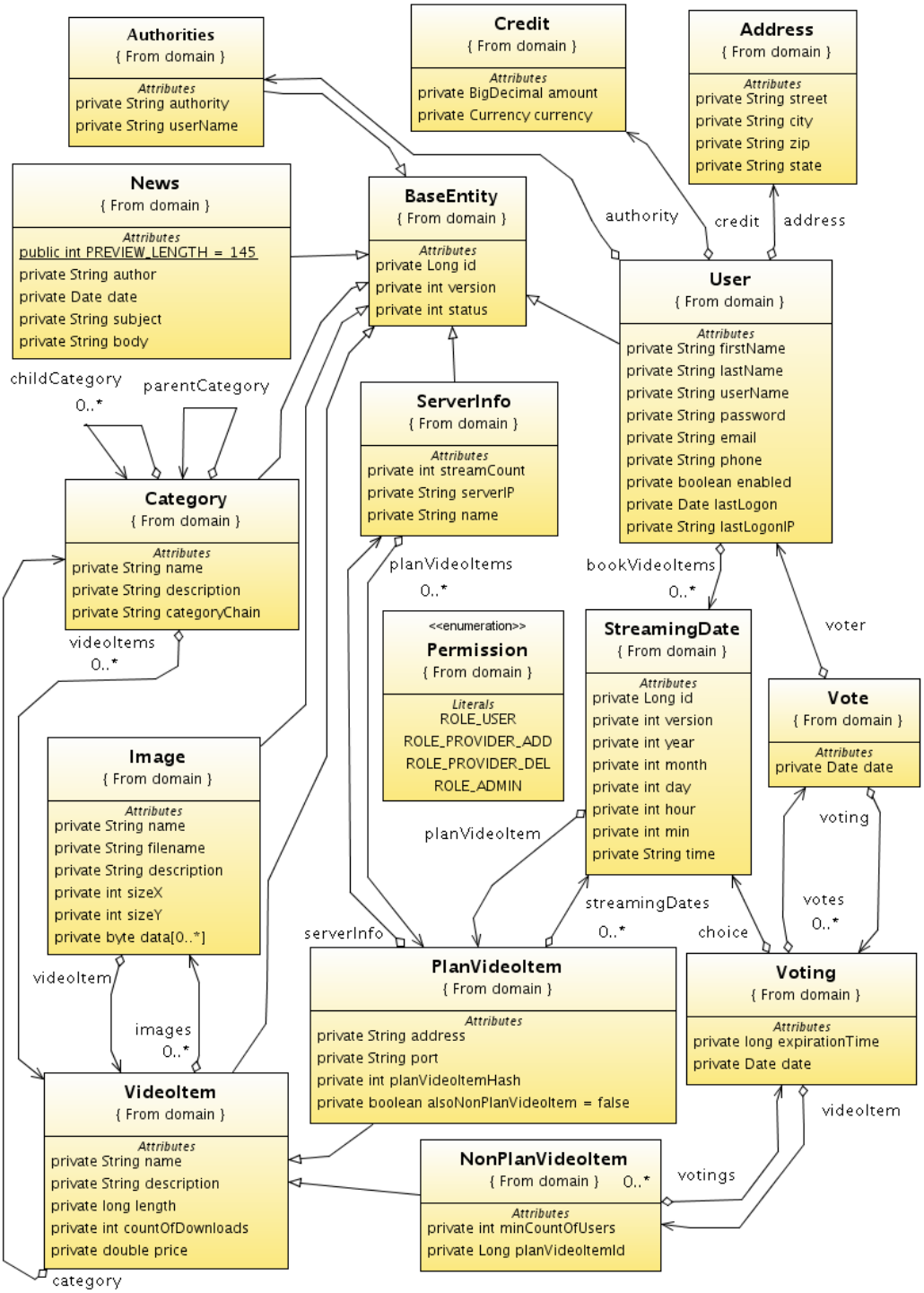
Vytváření DOM je velice podobné databázovému modelování, výsledek je ovšem zřídka kdy stejný. Tvorba DOM je zaměřená na vyjádření systémových entit a vztahů mezi nimi, zatímco databázové modelování se provádí za účelem dosažení co nejvyšší výkonnosti (rychlosti) práce s daty. Upravování DOM za účelem větší výkonnosti se provádí většinou až v okamžiku, kdy se objeví výkonostní problémy. Výsledný doménový model IPTV Manažera je na obr. 3.1.

Objekty doménového modelu jsou jedinými odjekty v aplikaci, jejichž životní cyklus není důsledně řízen Springem. DOM objektů se vytváří mnoho instancí a používat na vytváření a získávání těchto objektů Spring, by byl zbytečný "kanón na vrabce". Navíc DOM objekty nepotřebují využívat výhod DI, protože nemají žádné závislosti mimo doménový model a také nepotřebují žádnou konfiguraci.

Na obr. 3.1 doménového modelu IPTV Manažera je vidět, že všechny perzistentní třídy (třídy jejichž instance jsou pomocí ORM ukládány do databáze) dědí od třídy `BaseEntity`, která má následující tři atributy:

- id - ve všech dceřiných[1] třídách slouží jako Hibernate id (primární klíč v databázi),

[1]Třída která rozšiřuje svoji rodičovskou třídu (dědí od ní) je označována jako dceřinná.



Obr. 3.1: Doménový model IPTV Manažera v jazyce UML digramů tříd [16].

- version - využívá nástroj Hibernate k umožnění optimistického zamykání záznamů v databázi (viz. dále),
- status - obecný status, využíván v aplikační vrstvě každou třídou jinak, některé ho nepoužívají vůbec.

Navíc třída `BaseEntity` implementuje rozhraní `java.io.Serializable`, které je nezbytné k tomu, aby mohly být objekty posílány po síti. Definování takového společného předka má nejen tu výhodu, že není potřeba kód opakovat ve všech dceřinných třídách, ale také např to, že je možné využít obecnou metodu pro vyhledání konkrétního objektu v kolekci[2]. Kód takové motody je vidět v následujícím výpisu:

```
public static BaseEntity getById(Collection entities,Class entityClass,Long entityId)
        throws ObjectRetrievalFailureException {

   for (Iterator it = entities.iterator(); it.hasNext();) {
      BaseEntity entity = (BaseEntity) it.next();
      if (entity.getId().intValue() == entityId && entityClass.isInstance(entity)){
         return entity;
      }
   }
   throw new ObjectRetrievalFailureException(entityClass, new Long(entityId));
}
```

Další zajímavou třídou doménového medelu je `VideoItem`, instance této třídy uchovává základní informace o video pořadu (např. o nějakém filmu). Tyto základní informace jsou - jméno, popis, délka, cena a počet schlédnutí. Dále obsahuje kolekci ve které jsou odkazy na instance třídy `Image`. Každá kolekce v perzistentní třídě vyjadřuje vazbu *one-to-many* nebo *many-to-many*. Pokud instance jejichž odkazy jsou uloženy v kolekci obsahují odkaz na objekt vlastnící kolekci, znamená to, že vazba je obousměrná. Obousměrná vazba je při používání ORM nástroje velmi výhodná a jak bude ukázáno dále v této práci, dokáže ušetřit spoustu práce. Kromě kolekce obrázků obsahují instance této třídy ještě odkaz na instanci třídy `Category`. Tato vazba je obousměrná *many-to-one*, protože jedna kategorie smí obsahovat více videí. Požadavek vytváření jednotlivých podkategorií je splněn tím, že instance třídy `Category` obsahuje jednak kolekci instancí této třídy nazvanou childCategory, obsahující jednotlivé podkategorie a také odkaz na nadřazenou kategorii (parentCategory), který zajišťuje obousměrnost. Třídu `VideoItem` rozšiřují dvě specializované třídy `PlanVideoItem` a `NonPlanVideoItem`.

První z nich přidává informace o IPTV Serveru, který bude video streamovat. A také kolekci nazvanou streamingDates obsahující instance tříd `StreamingDate`, které uchovávají informaci o tom, kdy se bude dané video vysílat. Aby byla tato *one-to-many* vazba obousměrná, tedy aby i instance třídy `StreamingDate` věděla která

[2]kolekcí je myšlena jakákoliv implementace rozhraní `java.util.Collection` tedy např. `java.util.HashSet`, bude-li se dále v kapitole hovořit o kolekci, bude to vždy v tomto významu.

instance `PlanVideoItem` na ní má odkaz v kolekci, obsahuje třída `StreamingDate` mimo datumu a času vysílání i odkaz na `PlanVideoItem`.

Druhou třídou rozšiřující `VideoItem` je `NonPlanVideoItem`, která má obousměrnou *one-to-many* vazbu se třídou `Voting`. Každá instance třídy `Voting` slouží k uchování informací o jednom hlasování. Tyto informace jsou - datum a čas začátku hlasování, doba po které se má hlasování ukončit a kolekce instancí tříd `Vote`. Každá instance třídy `Vote` znamená jeden hlas v hlasování. `Vote` nese informaci o hlasující osobě - odkaz na instanci třídy `User` a datum kdy byl hlas udělen. Vazba mezi `Voting` a `Vote` je také obousměrná.

Třída `User` kromě jména, emailu, hesla a dalších informací o uživateli obsahuje odkaz na instance tříd `Address`, `Credit` a `Authorities`. `Credit` nese informaci o výši a měně uživatelova kreditu. Význam třídy `Authorities` a `Permission` bude popsán v části zabývající se bezpečností.

3.1.2 Objektově relační mapování

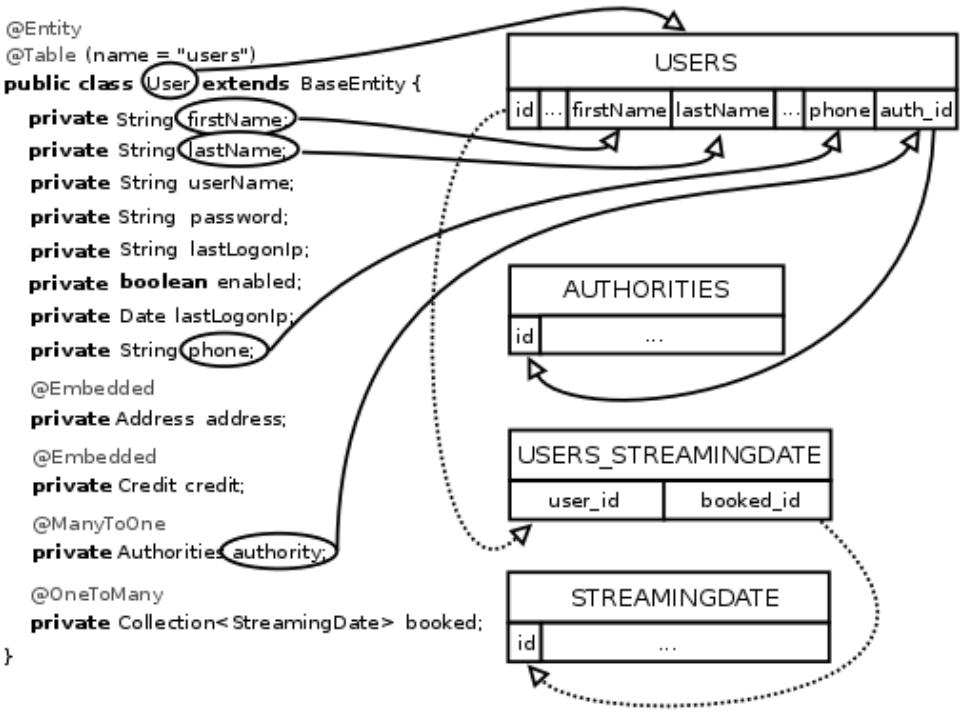
Hibernate zajišťuje automatickou konverzi dat mezi Java objekty a relačními databázovými tabulkami. Aby však mohl tento převod provádět, musí vědět, která tabulka odpovídá dané třídě a jak atributy této třídy správně převést na datové typy databáze. Informace, potřebné k tomuto převodu, lze předávat v podobě xml souborů nebo zapisovat přímo do kódu doménových tříd v podobě anotací. Druhý případ je sice kritizován tím, že náš kód tak získává závislost na rámci Hibernate, v případě IPTV Manažera ale tato závislost není nijak na závadu, proto jsou k mapování použity anotace. Příklad oanotované třídy `User` a její převod na databázovou tabulku je naznačen na obr. 3.2.

Podle anotace `@Entity` Hibernate pozná, že se jedná o perzistentní třídu, která je určena k procesu automatické konverze na databázové tabulky. Aby mohla být konverze úspěšná, musí třída minimálně ještě obsahavat oanotovaný atribut id, podle kterého jsou jednotlivé instance rozlišeny. Třída `User`, stejně jako všechny ostatní perzistentní třídy v ITPV Manažeru dědí atribut id spolu s dalšími od svého předka `BaseEntity`, jehož kód je vidět v následujícím výpisu:

```
@MappedSuperclass
public class BaseEntity implements Serializable {
   private static final long serialVersionUID = -2141785964197263472L;

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;

   @Version
   private int version;
   private int status;
   ...
}
```



Obr. 3.2: Mapování třídy User na tabulku USERS.

Anotace `@GeneratedValue(strategy = GenerationType.AUTO)` říká rámci Hibernate, že ke generování hodnot pro atribut id se použije buď strategie dané databáze, nebo pokud databáze tuto funkci nepodporuje, Hibernate sám zjistí hodnotu id posledního záznamu v databázi a pro nový záznam ji zvětší o jedna. V případě databáze MySQL se o generování stará datábáze, Hibernate pouze nastaví u sloupce id příznak „auto_increment“.

Atribut version s anotací `@Version` používá Hibernate k takzvanému optimistickému zamykání záznamů, které řeší konkurenční přístup k datům. Optimistické zamykání slouží v podstatě k tomu, aby uživatel nezměnil data v databázi, která mezitím co si je prohlížel, změnil někdo jiný. Funguje to tak, že Hibernate inkrementuje (zvětší o jedna) atribut version u objektu pokaždé, když jej modifikuje. A před tím než objekt uloží do databáze kontroluje, jestli atribut version nebyl změněn v jiné transakci a v případě že byl, vyhodí příslušnou výjimku.

Na obrázku 3.2 je dále znázorněno, že atributy základních java typů, jako je String, int, boolean, Date atd. nemusejí být oanotovány. Hibernate podle použité databáze sám zvolí vhodný datový typ. U databáze MySQL se např. String konvertuje na varchar. Anotace `@Embedded` u atributů Address a Credit říká, že data těchto tříd se mají ukládat do tabulky USERS. Anotace `ManyToOne` u atributu authority typu Authorities přidá do tabulky sloupec s názvem authority_id, což je vlastně cízí

klíč, vztažený k sloupci id v tabulce AUTHORITIES. `@OneToMany` u kolekce typů StreamingDate nazvané booked způsobí, že Hibernate v databázi vytvoří novou tabulku nazvanou USERS_STREAMINGDATE. V této tabulce jsou pouze dva cicí klíče, které vytváří spojení mezi záznamy v tabulkách USER a STREAMINGDATE.

3.1.3 DAO - objekt přistupující k datům

Jak už bylo zmíněno v předchozí kapitole, základ datové vrstvy tvoří návrhový vzor DAO (Data Access Object - objekt přistupující k datům), který díky abstraknímu rozhraní umožní dokonalé oddělení implementace datové vrstvy od zbytku aplikace. Nevýhodou tohoto vzoru ovšem je, že vzniká potřeba vytvořit velké množství rozhraní a tříd. Zpravidla pro každou perzistentní třídu z DOM jedno DAO rozhraní a jednu DAO třídu. Navíc základní kód takzvaných CRUD (Create, Read, Update, Delete) operací je ve všech DAO třídách téměř stejný. Situace se však zlepšila s příchodem Javy verze 1.5, která zavádí generické typy, pomocí kterých lze vytvořit generického (obecného) předka. Tento obecný předek implementuje DAO rozhraní v generické podobě:

```
public interface GenericDao<T, U extends Serializable> {

   public void saveOrUpdate(T object);

   public void delete(T object);

   public T findById(U pk);

   public Collection<T> findAll();

   void query(PaginatedListImpl page, String q, Object... params);
}
```

Obecný DAO předek implementující `GenericDao` používá ORM nástroj Hibernate. Aby mohl také snadno využívat výhod, které přináší podpora Springu pro tento nástroj, rozšiřuje třídu `HibernateDaoSupport`. Potomkům této třídy je pouze potřeba pomocí DI předat odkaz na objekt `SessionFactory`. Tento objekt je konfigurovatelný přes aplikační kontext Springu. Jinými slovy v konfiguračním souboru (v této aplikaci nese název applicationContext-hibernate.xml) mu lze předat potřebné informace o databázi (název, jméno, heslo atd.) a názvy oanotovaných perzistentních tříd z DOM. Mezi výhody, které `HibernateDaoSupport` přináší patří např. řízení transakcí, které jsou propagovány z aplikační vrstvy (viz. dále). Práce s Hibernate je poté daleko jednodušší, o čemž se lze přesvědčit v následujícím výpisu zdrojového kódu třídy `GenericDAOHibernateImpl`:

```
public abstract class GenericDAOHibernateImpl<T, U extends Serializable>
      extends HibernateDaoSupport implements GenericDao<T, U> {

   private Class<T> type;

   public GenericDAOHibernateImpl(Class<T> type) {
```

```
        this.type = type;
    }

    public T findById(U pk) {
        return (T) getHibernateTemplate().get(type, pk);
    }

    public void saveOrUpdate(T object) {
        getHibernateTemplate().saveOrUpdate(object);
    }

    public void delete(T object) {
        getHibernateTemplate().delete(object);
    }

    public Collection<T> findAll() {
        return getHibernateTemplate().find("from " + type.getName());
    }

    public void query(final PaginatedListImpl page, final String q,
            final Object... params) {
        getHibernateTemplate().execute(new HibernateCallback() {
            public Object doInHibernate(Session session)
                    throws HibernateException, SQLException {

                String queryString = page.addOrderBy(q);
                Query query = session.createQuery(queryString);
                query.setFirstResult(page.getFirstRecordIndex());
                query.setMaxResults(page.getPageSize());
                setParameters(query, params);
                page.setList(query.list());
                Query count = session.createQuery("select count(*) " + q);
                setParameters(count, params);
                page.setTotal(((Number) count.uniqueResult()).intValue());
                return null;
            };
        });
    }

    private void setParameters(Query query, Object... params) {
        for (int i = 0; i < params.length; i++)
            query.setParameter(i, params[i]);
    }

}
```

Z výpisu je vidět, že základní CRUD metody jsou s použitím `HibernateDaoSupport` velmi jednoduché. Jediná složitější a poměrně nepřehledná obecná metoda query, slouží k vykonání složitějších SQL dotazů. Chce-li uživatel např. zobrazit seznam plánovaných pořadů, kterých může být velké množství, není určitě vhodné použít metodu findAll(), která vrací všechny záznamy najednou a jejíž vykonání tudíž může trvat poměrně dlouhou dobu. Daleko výhodnější je použít právě metodu query, umožňující záznamy stránkovat a třídit podle zvoleného kritéria.

Zavedení obecného DAO předka nejen výrazně sníží množství kódu, ale také přispívá k větší celkové přehlednosti datové vrstvy. Navíc při změně DAO rozhraní již není třeba editovat všechny DAO třídy, ale jen obecného předka. Následující dva výpisy ukazují účinnost řešení na rozhraní `UserDao` a jeho implemntaci `UserDaoHibernateImpl`.

```
public interface UserDAO extends GenericDao<User, Long> {
    User findByUserName(String userName);
```

```
}
```

`UserDao` už definuje pouze jednu metodu, všchny ostatní metody dědí od `GenericDao`, které předává typ perzistentní třídy (`User`) a typ identifikátoru (`Long`).

```
public class UserDAOHibernateImpl extends GenericDAOHibernateImpl<User, Long>
      implements UserDAO {

   public UserDAOHibernateImpl(Class<User> type) {
      super(type);
   }

   public User findByUserName(final String userName) {
      return (User) this.getHibernateTemplate().execute(
            new HibernateCallback() {
               public Object doInHibernate(Session session) {
                  Criteria criteria = session.createCriteria(User.class);
                  criteria.add(Restrictions.like("userName", userName));
                  return criteria.uniqueResult();
               }
            });
   }

}
```

`UserDAOHibernateImpl` implementuje pouze jednu specifickou metodu (findByUserName), implementace ostatních obecných metod dědí od `GenericDAOHibernateImpl`.

Pro úplnost následující výpis ukazuje část obsahu konfiguračního souboru applicationContext-hibernate.xml, která definuje userDao beanu:

```
<bean id="userDao" class="cz.vutbr.xsimko02.dao.hibernate.UserDAOHibernateImpl">
   <constructor-arg>
      <value>cz.vutbr.xsimko02.domain.User</value>
   </constructor-arg>
   <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

3.2 Budování aplikační vrstvy

Aplikační vrstva obsahuje všechnu business logiku aplikace. Kdyby nebyla veškerá business logika soustředěna na jednom místě, ale byla by typicky roztroušena v kódu prezentační vrstvy, vedlo by to k velké duplikaci kódu, nesnadnému udržování, nemožnosti definovat transakce a ke spoustě dalším problémům. Jasně definovaná aplikační vrstva je jakousi bránou do aplikace, poskytující prezentační vrstvě jednoduchou a sjednocenou cestu k business logice.

Všechna business logika IPTV Manažera je obsažena v jedné servisní třídě nazvané `IPTVManagerImpl` a prezentační vrstva přistupuje k této logice přes rozhraní nazvané `IPTVManager`. Kód tohoto rozhraní je vidět v následujícím výpisu:

```
public interface IPTVManager {

   void createNewUserAccount(User user);
   void createNewProviderAccount(User user);
   void createNewAdminAccount(User user);
   void seveOrUpdateUser(User user);
```

```
    List<User> getAllUsers();
    User getUserById(Long id);
    User findUserByUserMame(String userName);
    void deleteUsersAccounts(Long[] ids);

    VideoItem getVideoItemById(Long id);
    void saveOrUpdateVideoItem(VideoItem videoItem);
    void deleteVideoItem(VideoItem videoItem);
    void saveOrUpdateCategory(Category category);
    public void deleteCategory(Category category);
    Category getCategoryById(Long id);
    Collection<Category> getAllCategories();
    Collection<StreamingDate> findPlanVideosByStreamingDay(int y, int m, int d);
    Collection<StreamingDate> findAvailableStreamingDays(VideoItem video);
    List<StreamingDate> findAvailableStreamingHours(VideoItem v,StreamingDate d);
    List<StreamingDate> findAvailableStreamingMins(VideoItem v,StreamingDate d);
    Collection<VideoItem> findTopVideos(int firstResult, int maxResult);
    void createVotingForNonPlanVideo(User u, NonPlanVideoItem v, StreamingDate d);
    void addVoteForNonPlanVideoItem(User user, Long votingId);
    void removeVoteForNonPlanVideoItem(Long voteId);
    NonPlanVideoItem findNonPlanVideoByPlanVideoId(final Long planVideoItemId);
    void bookVideoItem(String username,Long streamingDateId);
    void stornoBookedVideoItem(String userName, Long streamingDateId);
    StreamingDate getStreamingDateById(Long id);
    void deleteStreamingDate(StreamingDate streamingDate);

    void createNewServerInfo(ServerInfo serverInfo);
    public void deleteServerInfo(ServerInfo serverInfo);
    public List<ServerInfo> getAllServerInfo();
    public ServerInfo getServerInfoById(Long id);

    void saveOrUpdateNews(News news);
    News findNewsById(long id);
    public void deleteNews(News news);
}
```

Jakýkoliv kód, který bude interagovat s aplikační vrstvou, to bude dělat skrz toto rozhraní. Nahrazení implementace za rozhraní zařídí rámec Spring využitím DI. Implementace těchto metod, která se nachází ve třídě `IPTVManagerImpl`, je poměrně jednoduchá a ve většině případů se jedná pouze o volání metod DAO tříd. Jak už ale bylo zmíněno v předchozí kapitole, Spring přináší aplikační vrstvě obrovskou podporu v podobě transakcí. Transakce je možné zadefinovat na úrovni aplikační vrstvy a propagovat je do datové vrstvy. Transakce může být zadefinována programově voláním transakčního API přímo z vlastního kódu a nebo pomocí AOP, tomuto způsobu se říká deklarativní.

3.2.1 Deklarativní transakční management

Deklarativní transakční management je založen na AOP, díky čemuž je úplně oddělen od kódu aplikační vrstvy. Navíc konkrétní transakční manažer je definován pouze v konfiguračním souboru, takže nevzniká ani žádná závislost[3] mezi transakčním managementem a datovou vrstvou. Konfigurace servisní třídy IPTV Manažera, která se nachází v souboru applicationContext.xml je vidět v následujícím výpise:

```
<bean id="iptvManager"
```

[3] závislost je zde myšlena na úrovni námi psaného kódu

```
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="postInterceptors">
        <list>
            <ref bean="securityInterceptor" />
            <ref local="auditAdvisor"/>
        </list>
    </property>
    <property name="transactionManager" ref="transactionManager"/>
    <property name="target" ref="iptvManagerTarget"/>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>

<bean id="iptvManagerTarget" class="cz.vutbr.xsimko02.service.IPTVManagerImpl">
    <property name="userDao" ref="userDao" />
    <property name="videoItemDao" ref="videoItemDao" />
    <property name="categoryDao" ref="categoryDao" />
    <property name="votingDao" ref="votingDao" />
    <property name="serverInfoDao" ref="serverInfoDao" />
    <property name="newsDao" ref="newsDao" />
    <property name="imageDao" ref="imageDao" />
    <property name="streamingDateDao" ref="streamingDateDao" />
    <property name="iptvServerService" ref="iptvServerServiceHttpInvokerProxy"/>
    <property name="passwordEncoder" ref="passwordEncoder" />
    <property name="trustStore"><value>${security.trustStore}</value></property>
    <property name="trustStorePassword">
        <value>${security.trustStorePassword}</value>
    </property>
</bean>

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

<bean id="auditAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="auditInterceptor" />
    </property>
    <property name="patterns">
        <list>
            <value>.*delete.*</value>
            <value>.*book.*</value>
        </list>
    </property>
</bean>

<bean id="auditInterceptor"  class="cz.vutbr.xsimko02.aop.AuditInterceptor"/>
```

Od verze 2.0 nabízí Spring specializovanou třídu `TransactionProxyFactoryBean`, pomocí které je konfigurace deklarativního transakčního ohraničení poměrně snadná. V podstatě obnáší specifikovat pouze následující tři vlastnosti:

1. **transactionManager** – definuje konkrétní transakční manažer, který provádí vlastní management transakcí. Tato aplikace využívá na datové vrstvě Hibernate, proto je použit `HibernateTransactionManager`.

2. **target** – cílový objekt, pro který je transakční proxy vytvořena.

`TransactionProxyFactoryBean` obalí tento cílový objekt transakční náhradou (proxy), nahrazením všech rozhraní, které cílová třída implementuje. v této aplikaci je tímto objektem instance servisní třídy `IPTVManagerImpl`, která obsahuje veškerou business logiku aplikace. Ve výpisu výše je také vidět, že objektu `IPTVManagerImpl` jsou pomocí DI nastaveny všechny DAO objekty, pomocí kterých získává/ukládá data z/do databáze.

3. **transactionAttributes** – transakční atributy, které umožňují nastavení transakcí dle vlastní volby. PROPAGATION_REQUIRED je asi nejvíce používaný typ transakce, který funguje tak, že se snaží využít existující transakci a pokud žádná neexistuje, vytvoří novou. Pro metody začínající na get, load a find je navíc nastaveno, že v transakci není umožněno měnit nebo ukládat data.

Nepovinná vlastnost **postInterceptors** umožňuje přidat takzvané interceptory (sledovače), které jsou namapovány na volání metod. Z výpisu je vidět, že aplikace používá dva interceptory, securityInterceptor a auditAdvisor. SecurityIntercepor je namapován na volání všech metod a slouží k zamezení volání metod uživatelům, kteří nemají patřičné oprávnění. Podrobněji bude vysvětlen v kapitole zabývající se bezpečností. AuditAdvisor slouží k namapování AuditInterceptoru. Tímto způsobem lze snadno a bez zásahu do kódu metod určovat, které metody budou svoje volání zaznamenávat do log souboru. V konfiguraci zobrazené ve výpisu výše se budou zaznamenávat všechny metody začínající na delete a book. Kód samotného AuditInterceptoru je vidět v následujícím výpise:

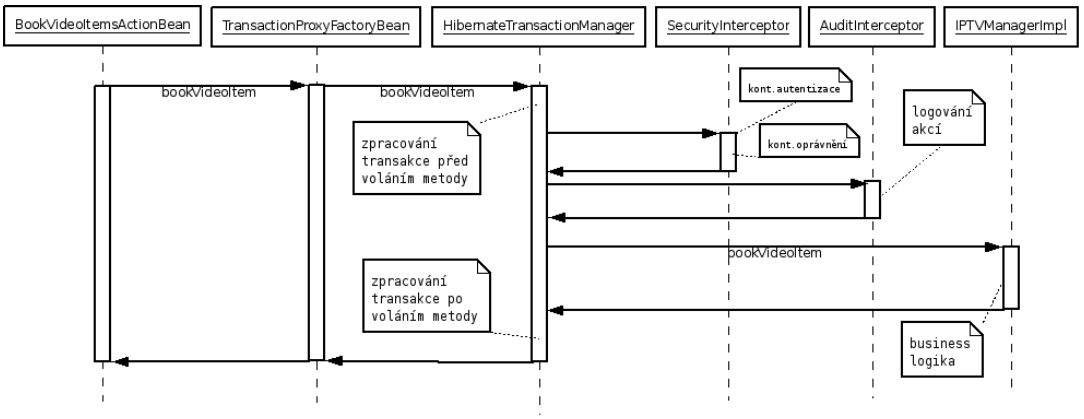
```
public class AuditInterceptor implements AfterReturningAdvice {

   private static Log log = null;

   public void afterReturning(Object arg0, Method arg1, Object[] arg2,
         Object arg3) throws Throwable {
      log = LogFactory.getLog(arg3.getClass());
      log.warn("Uživatel: "
            + SecurityContextHolder.getContext().getAuthentication()
                  .getName() + " zavolal metodu: " + arg1.getName());
   }
}
```

AuditInterceptor tedy zaloguje čas, jméno uživatele a jméno metody, kterou uživatel zavolal. Přičemž se využívá velmi užitečná schopnost rámce Acegi Security kdekoliv v aplikaci, pomocí třídy `SecurityContextHolder`, snadno zjistit přihlášeného uživatele.

Jak toto všechno dohromady funguje je znázorněno UML sekvečním diagramem na obr. 3.3. Diagram zachycuje chování aplikace na příkladu objednání video pořadu. Uživatel v tomto případě klikne v detailu videa na tlačítko objednat. To způsobí, že webový rámec Stripes vytvoří instanci třídy `BookVideoItemsActionBean`, která je na tuto akci namapována. V této třídě je zavolána metoda bookVideoItem,



Obr. 3.3: UML diagram sekvecí pro objednání video pořadu.

která ve svém těle volá metodu bookVideoItem na objektu implementujícím rozhraní `IPTVManager`, který jí předá Spring pomocí DI. Tímto objektem je proxy objekt vytvořený `TransactionProxyFactoryBean`, který obaluje `IPTVManagerImpl`. Tento proxy objekt deleguje vykonání metody na `HibernateTransactionManager`, který v připravené transakci volá metodu bookVideoItem na objektu `IPTVManagerImpl`. Protože jsou ale na tuto metodu navěšeny dva interceptory, volají se nejdříve jejich příslušné metody. Pokud ani jeden z interceptorů nevyhodí runtime výjimku, volá se metoda bookVideoItem z `IPTVManagerImpl`. Pokud toto volání proběhne bez problémů, `HibernateTransactionManager` transakci dokončí a všechny změny se promítnou do databáze. Pokud Jeden z interceptorů, nebo `IPTVManagerImpl` vyhodí runtime výjimku, celá transakce se „rollbackne“ a data v databázi zůstanou beze změny.

Například operace objednání videa sestává ze dvou úkonů, první je odečtení příslušné částky z kreditu za objednané video, druhá je samotné zpřístupnění videa uživateli. Tím, že se oba úkony vykonávají v transakci je zajištěno, že nenastane situace, aby jeden z úkonů byl proveden a druhý kvůli chybě zůstal nevykonán.

3.3 Budování prezentační vrstvy

Hlavním úkolem prezentační vrstvy je poskytnout UI (User interface - uživatelské rozhraní). Některé aplikace nabízejí více typů UI, jak ale vyplývá z analýzy provedené v kapitole 1, IPTV Manažer nabízí pouze webové UI. Díky tomu, že má aplikace pouze jedno UI, bude prezentační vrstva vykonávat i validaci (ověření správnosti) vstupních dat, zadaných uživateli. Kdyby aplikace nabízela více typů UI, bylo by výhodnější provádět validaci až v aplikační vrstvě. Kromě validace má tato vrstva na starosti ještě poměrně dost věcí a proto by byla velká chyba nepoužít některý

z řady webových rámců, které dokáží ušetřit spoustu práce a výrazně tak zkrátit dobu vývoje aplikace. Webové rámce ve světě JEE v současné době existují dvojího druhu. Jsou to rámce orientované na zpracování HTTP požadavku (např. Spring MVC, Apache Struts, Stripes) a rámce orientované na vizuální komponenty (např. JSF, Tapestry, Cocoon). Jak již bylo řečeno v kapitole 2, pro tuto aplikaci byl zvolen rámec Stripes. Stripes umožňuje využití velice výhodného architektonického vzoru MVC (Model View Controller - Model Pohled Řadič), díky kterému je prezentační vrstva přehledná a snadno rozšiřitelná. Vzor MVC předepisuje rozdělení prezentační vrstvy do 3 základních částí:

1. **model** – je tvořen objekty nesoucími data, která budou zobrazena prostřednicvím pohledu. V této aplikaci jsou těmito objekty instance tříd z DOM.

2. **pohled** – zobrazuje data (reprezentovaná modelem), která obdrží od řadiče. Pohled je v této aplikaci tovřen stránkami JSP, kde se navíc využívají technologie JSTL, knihovna značek Stripes, HTML, CSS, JavaScript, atd.

3. **řadič** – reaguje na události uživatele, podle kterých vytváří model a posílá ho příslušnému pohledu. Tvoří jej třídy, takzvané ActionBeany, které jsou řízeny rámcem Stripes.

Hlavní výhodou Stripes oproti ostatním MVC rámcům je, že nepotřebuje žádné konfigurační soubory. Při dodržování určitých konvencí pro pojmenovávání tříd a jsp stránek se o veškeré mapování postará Stripes automaticky. Jediná nutná konfigurace tohoto rámce se nachází v inicializačním souboru celé aplikace, který má standardizované umístění (v adresáři WEB-INF) a název web.xml. Nejdůležitější části tohoto souboru jsou vidět v následujícím výpise:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <!--umístění jednotlivých konfiguračních souborů pro aplikační kontexty
    použité v aplikaci-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/applicationContext.xml
            /WEB-INF/applicationContext-hiberanate.xml
            /WEB-INF/applicationContext-acegi-security.xml
            /WEB-INF/applicationContext-authorization.xml
            /WEB-INF/applicationContext-remote.xml
        </param-value>
    </context-param>

    <!--načtení aplikačních kontextů.-->
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
```

```
</listener>

<!--zavedení sady bezpečnostních filtrů, které jsou konfigurovány
v applicationContext-acegi-security.xml-->
<filter>
    <filter-name>Acegi Filter Chain Proxy</filter-name>
    <filter-class>
        org.acegisecurity.util.FilterToBeanProxy
    </filter-class>
    <init-param>
        <param-name>targetClass</param-name>
        <param-value>
            org.acegisecurity.util.FilterChainProxy
        </param-value>
    </init-param>
</filter>

<!--konfigurace rámce Stripes-->
<filter>
    <display-name>Stripes Filter</display-name>
    <filter-name>StripesFilter</filter-name>
    <filter-class>
        net.sourceforge.stripes.controller.StripesFilter
    </filter-class>
    <init-param>
        <param-name>Configuration.Class</param-name>
        <param-value>
            cz.vutbr.xsimko02.web.stripes.RuntimeConfiguration
        </param-value>
    </init-param>
    <init-param>
        <param-name>ActionBeanContext.Class</param-name>
        <param-value>
            cz.vutbr.xsimko02.web.stripes.CustomActionBeanContext
        </param-value>
    </init-param>
    <init-param>
        <!-- umístění action beanů -->
        <param-name>ActionResolver.UrlFilters</param-name>
        <param-value>WEB-INF/classes</param-value>
    </init-param>
</filter>

<!--zařídí otevření Hibernetí Session v prezentační vrstvě-->
<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>
        org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
    <param-name>sessionFactoryBeanName</param-name>
    <param-value>sessionFactory</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>Acegi Filter Chain Proxy</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>StripesFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
    <filter-name>StripesFilter</filter-name>
    <servlet-name>StripesDispatcher</servlet-name>
```

```
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
    <servlet>
        <servlet-name>StripesDispatcher</servlet-name>
        <servlet-class>
            net.sourceforge.stripes.controller.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>StripesDispatcher</servlet-name>
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>

    <!-- definice vlastní knihovny tagů - kalendář -->
    <jsp-config>
        <taglib>
              <taglib-uri>http://xsimko02.vutbr.cz/taglibs</taglib-uri>
              <taglib-location>/WEB-INF/myTagLibs.tld</taglib-location>
        </taglib>
    </jsp-config>
</web-app>
```

Pokud není potřeba žádné speciální nastavení, tak vše co Stripes potřebuje aby se probudil k životu, je konfigurace dispatcher servletu a Stripes filtru. Speciálním nastavením je zde myšlena úprava jeho výchozí konfigurace, která je možná přidáním inicializačních parametrů. V této aplikaci byly takové parametry přidány dva. Parametru Configuration.Class byla předána třída `RuntimeConfiguration`, která rozšiřuje třídu `DefaultConfiguration` a umožňuje přepínání lokalizace mezi angličtinou a češtinou na základě dat v requestu. Dalšímu parametru ActionBeanContext.Class byla předána třída `CustomActionBeanContext`, ze které budou dědit všechy ActionBeany. Tento společný předek byl vytvořen kvůli těmto třem vlastnostem:

```
    @SpringBean
    protected IPTVManager iptvManager;
    protected String basePath = "WEB-INF/jsp/stripes/";
    protected String centralMenu;
```

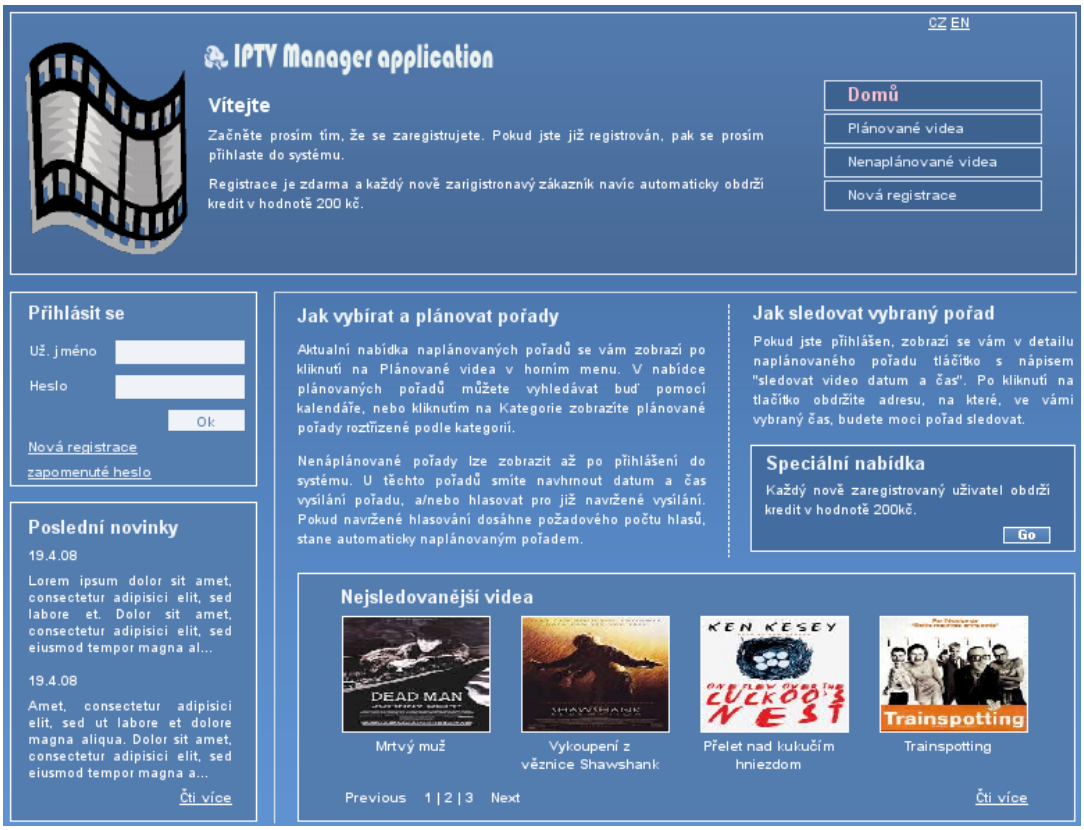
díky kterým budou mít všichni potomci přístup k objektu iptvManager (obsahuje business logiku), budou znát cestu k jsp stránkám a poslední centrálMenu slouží ke zvýraznění aktuálně vybrané položky v menu.

V souboru web.xml se kromě konfigurace rámce Stripes nachází spousta další důležité konfigurace, jako například zavedení `ContextLoaderListenera`, který spustí nastartování samotného rámce Spring. Filtr `FilterToBeanProxy` slouží k zavedení sady bezpečnostních Acegi filtrů, které jsou konfigurovány pomocí DI v aplikačním kontextu aplikace. Dalším zajímavým filtrem je `OpenSessionInViewFilter`, který zařídí, že transakční manažer ponechá Hibernetí Session otevřenou až do doby, než je dokončeno renderování pohledu. Hibernate totiž v této aplikaci pracuje v takzvaném „lazy init“ režimu (lína inicializace), který dokáže ušetřit stovky zbytečných dotazů do databáze a stovky vytváření nepotřebných objektů. Hibernate v tomto režimu dotahuje data a vytváří objekty, až když jsou opravdu potřeba. Díky tomuto

filtru je umožněno dotahovat data[4] i v prezentační vrstvě. Bez něho by byla potřeba všechny objekty inicializovat už v aplikační vrstvě.

3.3.1 Tvorba webového designu

Design je velmi důležitou součástí aplikace a často může být dokonce tím, co rozhodne o dalším setrvání, nebo odchodu uživatele z našeho webu. Při tvorvě webového designu IPTV Manažera byla použita jedna ze šablon, které jsou dostupné na http://www.freewebsitetemplates.com. Tyto šablony lze upravit a použít prakticky bez omezení, dokonce i odkaz na jejich internetové stránky je povolené smazat. Na obrázku 3.4 je vidět webový design hlavní stránky.



Obr. 3.4: Uživatelské rozhraní hlavní stránky.

Layout (uspořádání) všech stránek aplikace je stejný a skládá se ze čtýř oddílů. Na základě uživatelských akcí se pouze v jednotlivých oddílech mění zobrazovaný obsah.

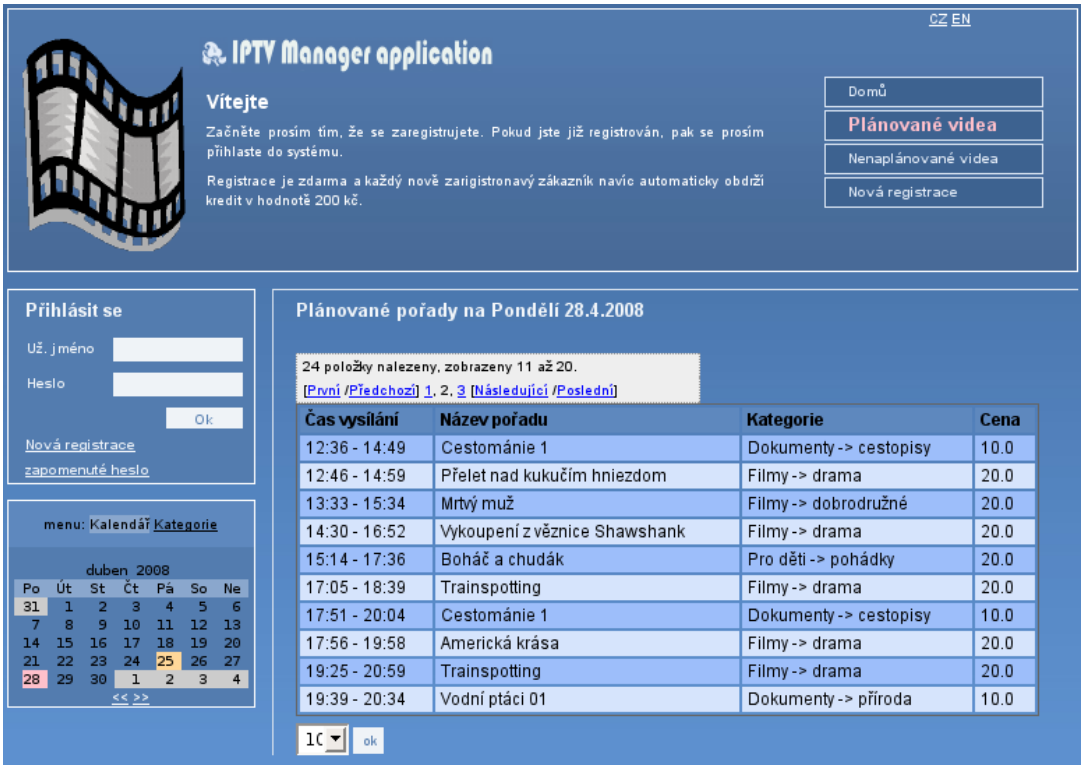
V horním oddílu se nachází hlavní navigační menu, které zvýrazňuje aktuálně vybranou položku. V této části se také nachází odkazy, pomocí kterých lze měnit

[4]dotahovámím dat je myšleno např. automatické vytváření objektů, na které odkazují objekty předané v modelu.

jazyk všech textů mezi češtinou a angličtinou. Přihlášenému uživateli se zde také zobrazuje datum a čas posledního přihlášení a adresa, ze které to bylo.

V levém horním oddílu se nachazí vstupní pole, kam uživatel píše přihlašovací informace. Poté co se přihlásí, zobrazí se mu v této části informace a možnosti, které závisí na jeho oprávnění. Uživateli s oprávněním USER se zobrazí jeho aktuální stav kreditu a jedna položka v menu, přes kterou může spravovat svoje objednané pořady. Uživatel s rolí PROVIDER má v menu navíc položku omožňující správu IPTV Serverů. Adnim má oproti Provideru navíc dvě položky, správu novinek a správu uživatelských účtů.

Obsah levého spodního oddílu se mění na základě vybrané možnosti v hlavním menu. Například na obrázku 3.5 je zde zobrazen kalendář, ve kterém lze vybrat libovlný den, pro který si přejeme zobrazit program. Námi vybraný den je v kalendáři zvýrazněn růžovou, aktuální den oranžovou barvou. Pro tento kalendář bylo nutné vytvořit vlastní tag, jehož kód se nachází ve třídě `CalendarTag`. Díky tomuto tagu bylo také nutné přidat do web.xml direktivu jsp-config. Nad kalendářen se nachází odkaz Kategorie, který místo kalendáře zobazí menu složené z kategorií, ve kterých se nacházejí video programy.



Obr. 3.5: Uživatelské rozhraní stránky plánovaných pořadů.

V největším oddílu se podle možností vybraných v ostatních oddílech zobrazuje příslušný obsah. Na obrázku 3.5 je to program pořadů, pro zvolený den v kalendáři.

V programu jsou položky zobrazeny seřazené podle času vysílání, uživateli je však umožněno řadit záznamy i podle názvu, kategorie nebo ceny. Dále je možné měnit počet zobrazených záznamů na stránce. V případě, že je záznamů více než nastavený počet, zobrazí se banner pomocí kterého je možné přecházet mezi stránkami. Tuto funkčnost velice usnadnila open source knihovna tagů displaytag.

3.4 Budování vrstvy vzdáleného volání

Kromě toho, že IPTV Manažer slouží ke komunikaci s uživateli, kteří si přes něj vybírají a ovlivňují vysílání, musí také umět komunikovat s IPTV Serverama, které pak všechny tyto vysílání vysílají. IPTV Servery neposkytují žádné uživatelské rozhraní, veškerá správa video pořadů a časů jejich vysílání probíhá prostřednicvím Manažera. Protože IPTV Manažer i jednotlivé IPTV Servery běží na jiných počítačích, musí být schopni komunikovat prostřednicvím sítě. Jak už bylo popsáno v kapitole 2, komunikaci zajišťuje Spring http invoker, který využívá protokol http a nemá proto problémy s firewally. Aby se dala komunikace vyzkoušet, byl navržen a implementován jednoduchý IPTV Server, který podle zadání neměl být součástí této práce.

3.4.1 IPTV Server

IPTV Server má vlastní databázi, ve které uchovává informace o video pořadech a časech jejich vysílání. Zatímco Manažer uchovává informace o video pořadech pro všechny IPTV Servery, Server má informace jen o těch pořadech, které bude sám streamovat. Když je potřeba provést nějakou změnu v těchto záznamech (např. přidat čas vysílání), provede se to přes webové rozhrání Manažera, který následně o změně informuje příslušný Server, který provede změny ve své databázi. Komunikace tedy probíhá vždy z Manažera na Server. Jinými slovy, Manažer vzdáleně volá metodu na Serveru a v parametru této metody mu předává data. Tímto způsobem Manažer zajišťuje, že Server má stejné (aktuální) data jako Manažer. Zpětné potvrzování bezchybného přenosu a správného uložení dat na straně Serveru není potřeba, neboť celá konukace je obalena transakcí a Manažer si tedy může být jistý, že vše proběhlo v pořádku. Pokud by se totiž objevil problém s přenosem dat, nebo se zpracováním jak na straně Manažera, tak na straně Serveru, transakce by se rollbackla a v obou databázích by stále zůstaly pravdivý data.

Díky tomu, že i IPTV Server je postaven nad aplikačním rámcem Spring, je vystavení vzdáleně volatelné motody poměrně snadné. V souboru web.xml se nadefinuje příslušný Exporter, kterému parametrem url-pattern nastavíme adresu, na které bude možné metodu zavolat. Aby nemohl metodu volat kdokoliv, kdo zná tuto

adresu, používá IPTV Server zabezpečení v podobě standardní http Basic autentizace, viz. následující výpis, který ukazuje část web.xml souboru:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>IPTVServer Application</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>iptvManager</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>IPTVServer Application</realm-name>
</login-config>

<servlet>
    <servlet-name>iptvServerServiceExporter</servlet-name>
    <servlet-class>
        org.springframework.web.context.support.HttpRequestHandlerServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>iptvServerServiceExporter</servlet-name>
    <url-pattern>/remoting/server1</url-pattern>
</servlet-mapping>
```

V aplikačním kontextu se určí rozhraní, které obsahuje vzdáleně volatelné metody a příslušná implementace tohoto rozhraní, na které se metody budou volat. Viz. následující výpis části souboru applicationContext.xml:

```
<bean name="iptvServerServiceExporter"
    class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="iptvServerService"/>
    <property name="serviceInterface"
        value="cz.vutbr.xsimko02.service.IptvServerService"/>
</bean>

<bean name="iptvServerService"
    class="cz.vutbr.xsimko02.service.IptvServerServiceImpl">
    <property name="iptvServerDao" ref="iptvServerDao"/>
</bean>
```

Rozhraní `IptvServerService` obsahuje pouze jednu metodu nazvanou saveServerInfoOnServer:

```
public interface IptvServerService {
    void saveServerInfoOnServer(ServerInfo serverInfo);
}
```

Metoda přebírá jako parametr doménovou třídu `ServerInfo`, která obsahuje kolekci tříd `PlanVideoItem` (video pořad), které obsahují kolekce tříd `StreamingDate` (datum a čas vysílání).

Tato implemetace IPTV Serveru neumí streamovat pořady, pouze všechny pořady zobrazí v tabulce a u každého času vysílání vypíše stav (obr. 3.6). Stav všech vysílání je automaticky kontrolován každou minutu.

Všechna videa na serveru

ID	Jméno	Délka	Vysílání
53	Americká krása	122	ID / Datum / Čas / Stav 276 / 27.4.2008 / 18:11 / čeká na vysílání 277 / 28.4.2008 / 17:56 / čeká na vysílání 278 / 26.4.2008 / 19:58 / už bylo vysíláno 279 / 25.4.2008 / 13:36 / už bylo vysíláno 280 / 28.4.2008 / 22:04 / čeká na vysílání 281 / 27.4.2008 / 00:21 / čeká na vysílání 319 / 26.4.2008 / 21:00 / právě se vysílá
51	Přelet nad kukučím hniezdom	133	ID / Datum / Čas / Stav 262 / 26.4.2008 / 20:48 / právě se vysílá 263 / 28.4.2008 / 04:13 / čeká na vysílání 264 / 25.4.2008 / 08:46 / už bylo vysíláno 265 / 26.4.2008 / 00:18 / už bylo vysíláno
49	Mrtvý muž	121	ID / Datum / Čas / Stav 244 / 27.4.2008 / 21:09 / čeká na vysílání 245 / 26.4.2008 / 12:16 / už bylo vysíláno 246 / 27.4.2008 / 22:40 / čeká na vysílání 318 / 26.4.2008 / 22:00 / právě se vysílá
50	Vykoupení z věznice Shawshank	142	ID / Datum / Čas / Stav 254 / 27.4.2008 / 15:06 / čeká na vysílání 255 / 28.4.2008 / 14:30 / čeká na vysílání 256 / 28.4.2008 / 09:17 / čeká na vysílání 257 / 28.4.2008 / 08:26 / čeká na vysílání 320 / 28.4.2008 / 20:00 / čeká na vysílání

Obr. 3.6: Výpis pořadů na IPTV serveru.

3.4.2 IPTV Manažer

IPTV Server vystavil vzdáleně volatelnou metodu a úkolem této vrstvy v IPTV Manažeru je tuto metodu zpřístupnit. Spring pro tento účel nabízí továrnu[5] `HttpInvokerProxyFactoryBean`, která vytvoří proxy objekt obalující vzdálený zdroj. Tímto nás odstíní od všech implementačních detailů a jednoduše umožní přístup ke vzdáleným službám přes Java rozhraní, viz následující výpis části souboru applicationContext-remote.xml:

```
<bean id="iptvServerServiceHttpInvokerProxy"
    class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceUrl"
        value="https://adela.utko.feec.vutbr.cz:8443/iptvServer/remoting/server1" />
    <property name="serviceInterface"
        value="cz.vutbr.xsimko02.remote.IPTVServerService" />
    <property name="httpInvokerRequestExecutor">
        <bean
            class="cz.vutbr.xsimko02.remote.AuthCommonsHttpInvokerRequestExecutor">
        </bean>
    </property>
</bean>
```

Rohraní `IPTVServerService` je stejné jako bylo na IPTV Serveru (obsahuje pouze jednu metodu - saveServerInfoOnServer). Servisní třída `IPTVManagerImpl` z aplikační vrstvy toto rozhraní implemetuje, díky čemuž je jí umožněno vzdáleně volat metodu na IPTV Serveru.

Jediné komplikace, které se při vytváření této vrstvy vyskytly, byly s dynamickým zadáváním adresy IPTV Serveru a s autentizací IPTV Manažera.

Jak je vidět ve výpisu výše, adresa, na které proxy objekt vytvořený továrnou `HttpInvokerProxyFactoryBean` hledá vzdálený zdroj, se standardně zadává v konfiguračním souboru v parametru serviceUrl. IPTV Manažer ale musí být schopen komunikovat s různými IPTV Servery na různých adresách, které navíc nejsou v době vytváření proxy objektu známé. Proto musela být nalezena následující cesta, která umožnila změnu této adresy za běhu aplikace. Kdykoliv uživatel (s oprávněním PROVIDER nebo ADMIN) odešle data na IPTV Server, zavolá se v prezentační vrstvě, ve třídě `ServerInfoServicesActionBean` metoda sendToServer, ve které je na základě objektu `HttpRequest` získán aplikační kontext. Z aplikačního kontextu je získán proxy objekt vytvořený továrnou `HttpInvokerProxyFactoryBean` a tomuto objektu je nastavena příslušná adresa. Celý tento postup je nejlépe vidět v následujícím výpisu:

```
    @DontValidate
    public Resolution sendToServer() {
        ServletContext servletContext = getContext().getRequest().getSession()
                .getServletContext();
        WebApplicationContext wac = WebApplicationContextUtils
                .getRequiredWebApplicationContext(servletContext);
        HttpInvokerProxyFactoryBean factory = (HttpInvokerProxyFactoryBean) wac
```

[5] Továrna je návrhový vzor, který umožňuje rozhodnout o typu vytvářené instance až v průběhu programu.

```
            .getBean("&iptvServerServiceHttpInvokerProxy");
        serverInfo = iptvManager.getServerInfoById(serverInfoId);
        factory.setServiceUrl(serverInfo.getServerIP());
        iptvManager.saveServerInfoOnServer(serverInfo);
        return new RedirectResolution("/ServerInfoServices.action");
    }
```

Druhá komplikace byla způsobená tím, že celý IPTV Server je zabezpečen standardní http Basic autentizací. Navíc není ke komunikaci použit protokol http, ale jeho zabezpečená nadstavba https, což si také vyžádalo drobné zkomplikování.

3.4.3 Bezpečnost komunikace

Bezpečnost komunikace mezi IPTV Manažerem a IPTV Serverem je zajištěna dvěma faktory:

1. IPTV Manažer se musí IPTV Serveru úspěšně autentizovat, jinak mu není umožněno volat vzdálenou metodu.

2. Komunikace probíhá přes šifrovaný protokol https.

Jak už bylo popsáno výše, IPTV Server je zabezpečen http Basic autentizací, nastavení této autentizace souvisí spíše se servletovým kontejnérem a proto bude popsáno v kapitole 5. V IPTV Manažeru se o získání a předložení autentizačních údajů stará třída `AuthCommonsHttpInvokerRequestExecutor`, která rozšiřuje třídu `CommonsHttpInvokerRequestExecutor` z knihovny Apache commons - httpclient. `AuthCommonsHttpInvokerRequestExecutor` stejně jako logovací interceptor využívá schopnosti rámce Acegi Security kdekoliv v aplikaci, pomocí třídy `SecurityContextHolder`, zjistit uživatelské jméno a heslo přihlášeného uživatele.

Pro komunikaci pomocí https musí IPTV Server vlastnit certifikát, jehož vytvoření a přidání do aplikace bude popsáno v kapitole 5. Certifikát bohužel není podepsán Certifikační autoritou, která zaručuje, že vlastník certifikátu se nevydává za nikoho jiného. Podepsaný certifikát od některé CA se standardně považuje za důvěryhodný a není proto nutná další konfigurace. Podepsání certifikátu však není zadarmo a navíc trvá nějakou dobu. Proto aby IPTV Manažer akceptoval Serverův nepodepsaný certifikát, je potřeba tento certifikát přidat do úložiště důvěryhodných certifikátů (viz kap. 5). V metodě saveServerInfoOnServer servisní třídy `IPTVManagerImpl` se pak nastaví příslušným systémovým vlastnostem JVM (Java Virtual Machine) cesta k úložišti a heslo, kterým je úložiště zabezpečeno, viz. následující výpis:

```
public void saveServerInfoOnServer(ServerInfo serverInfo) {
    System.setProperty("javax.net.ssl.trustStore", trustStore);
    System.setProperty("javax.net.ssl.trustStorePassword", trustStorePassword);
    Collection<PlanVideoItem> videos = serverInfo.getPlanVideoItems();
    for (PlanVideoItem videoItem : videos) {
```

```
            Collection<StreamingDate> sDates = videoItem.getStreamingDates();
            if(sDates != null && sDates.size() != 0)
               videoItem.setStatus(1);
            videoItem.setAddress(serverInfo.getAddress()+"/"+serverInfo.getName()
                     + "?videoItemId="+videoItem.getId());
            videoItem.setPlanVideoItemHash(videoItem.getHashCode());
            videoItemDao.saveOrUpdate(videoItem);
        }
        iptvServerService.saveServerInfoOnServer(serverInfo);
    }
```

Ve výpisu je také vidět, že se při odesílání dat spočítá hash (otisk) každého video pořadu a ten se uloží do databáze. Díky tomu IPTV Manažer rozpozná, zda byly od posledního úspěšného odeslání provedeny nějaké změny v datech.

4 IMPLEMENTACE BEZPEČNOSTI

Na dosažení požadované míry zabezpečení se podílejí všechny vrsvy aplikace. Jelikož je ale IPTV Manažer čistě webová aplikace, připadá nejvíce práce na vrstvu prezentační.

4.1 Bezpečnost v prezentační vrstvě

Již zmiňovaný mocný bezpečnostní rámec Acegi je do aplikace integrován právě v této vrstvě. Acegi používá řady filtrů[1] konfigurovatelných pomocí DI, které zajišťují bezpečnostní služby aplikaci. Následující výpis ukazuje filtry použité v IPTV Manažeru:

```
<bean id="filterChainProxy"
    class="org.acegisecurity.util.FilterChainProxy">
    <property name="filterInvocationDefinitionSource">
        <value><![CDATA[
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /**=channelProcessingFilter,httpSessionContextIntegrationFilter,
                authenticationProcessingFilter,exceptionTranslationFilter,
                filterInvocationInterceptor
            ]]></value>
    </property>
</bean>
```

Na různá URL lze aplikovat různé řetězce filtrů, v tomto případě jsou však všechny filtry aplikovány na všechna požadovaná URL. Jednotlivé filtry jsou konfigurovány v souboru applicationContext-acegi-security.xml, tento soubor má přes dvěstě řádků, proto zde nebudou konfigurace vypsány, pouze bude řečeno, co který filtr v aplikaci vykonává.

První filtr `ChannelProcessingFilter` zajistí, že je aplikace dostupná pouze přes zabezpečený protokol. V případě, že se někdo pokusí přistoupit pomocí protokolu http, filtr ho přesměruje na https.

`HttpSessionContextIntegrationFilter` se stará o předání autentizačních a dalších informací z objektu `HttpSession` objektu `SecurityContextHolder`, o kterém už byla řeč v kapitole 3.

Filtr `AuthenticationProcessingFilter` v případě požadavku na chráněnou URL a současné absence autentizačních údajů v objektu `SecurityContextHolder` přesměruje uživatele na přihlašovací formulář. Po úspěšné autentizaci provede automatické přesměrování na původní chráněnou URL. Aby při každém požadavku autentizovaného uživatele nemusel být vykonán dotaz do databáze, využívá se kešování autentizačních údajů pomocí technologie EHCache.

[1] Filtry ze servlet specifikace, tedy třídy, které implementují rozhraní `javax.servlet.Filter`

`ExceptionTranslationFilter` je v podstatě most mezi Java výjimkami vyhozenými ostatními filtry a http odpovědmi. Zpracovává například výjimky způsobené neúspěšnou autentizací nebo autorizací a přesměrovává na příslušné stránky oznamující tuto událost.

Konfigurace filterInvocationInterceptoru je poměrně zajímavá, proto je ukázána v následujícím výpise:

```
<bean id="filterInvocationInterceptor"
    class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
    <property name="authenticationManager"
        ref="authenticationManager" />
    <property name="accessDecisionManager"
        ref="accessDecisionManager" />
    <property name="objectDefinitionSource">
        <value><![CDATA[
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /userlist.*=ADMIN
            /serverinfoservices.*=ADMIN,PROVIDER_ADD,PROVIDER_DEL
            /nonplanvideos.*=IS_AUTHENTICATED_REMEMBERED
            /bookvideoitems.*=IS_AUTHENTICATED_REMEMBERED
            /**=IS_AUTHENTICATED_ANONYMOUSLY
        ]]></value>
    </property>
</bean>
```

Tento filtr definuje přístupové atributy pro jednotlivá URL. V tomto případě filterInvocationInterceptor zabezpečí, že zobrazení seznamu uživatelů bude umožněno jenom uživateli s oprávněním admin. Zobrazit informace o IPTV Serverech smí jen uživatelé s oprávněním admin nebo provider. A zobrazování nenaplánovaných a objednaných pořadů smějí všichni úspěšně autentizovaní uživatelé.

4.1.1 Acegi knihovna tagů

Další velmi užitečnou věcičkou poskytnutou rámcem Acegi je knihovna tagů, pomocí které lze jednoduše měnit zobrazovaný obsah stránky, na základě oprávnění přihlášeného uživatele. Například následující tři odkazy se zobrazí pouze uživateli s oprávněním admin:

```
<authz:authorize ifAllGranted="ADMIN">
    <s:link href="/ServerInfoServices.action"><span>IPTV Servery</span></s:link>
    <s:link href="/UserList.action"><span>Uživatelé</span></s:link>
    <s:link href="/News.action?show=formNews"><span>Novinky</span></s:link>
</authz:authorize>
```

4.1.2 Validace vstupů a výstupů

Prezentační vrstva přijímá data, která jsou zpracována aplikační vrstvou a následně vrácena zpět k zobrazení prezentační vrstvě. Kontrola, zda uživatel zadal data ve správném formátu, zajišťuje právě tato vrstva. Validování (ověřování) vstupních dat, je s využitím rámce Stripes velice snadné a přehledné:

```
@ValidateNestedProperties( {
   @Validate(field = "userName", required = true, minlength = 5, maxlength = 10),
   @Validate(field = "password", required = true, minlength = 5, maxlength = 10),
})
public void setUser(User user) {
   this.user = user;
}
```

Díky této anotaci u metody setUser ve třídě `UserRegistrationActionBean` je zajištěno, že dokud uživatel nezadá správně (delší než 5 a kratší než 10 znaků) uživatelské jméno a heslo, bude se mu registrační formulář stále vracet s příslušnou upozorňovací hláškou.

Všechny výstupy, tedy místa, kde se zobrazují data zadané uživateli, jsou navíc vypisovány pomocí tagu z knihovny JSTL, který je ošetří před speciálními znaky:

```
<c:out value="${data}"/>
```

Pukud by se vše vypisovalo přímo, mohl by záškodník např. přidat do stránky kus JavaScriptového kódu a provést tak Cross-Site Scripting.

4.2 Bezpečnost v aplikační vrstvě

Aplikace nespoléhá pouze na ochranu založenou na chráněných URL, ale kombinuje ji se zabezpečením volání metod z aplikační vrstvy. Jak bylo ukázáno na obr. 3.3, na všechny metody ze servisní třídy `IPTVManagerImpl` je navěšený securityInterceptor, který má následující konfiguraci:

```
<bean id="securityInterceptor"
   class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
   <property name="authenticationManager">
      <ref bean="authenticationManager"/>
   </property>
   <property name="accessDecisionManager">
      <ref bean="accessDecisionManager"/>
   </property>
   <property name="objectDefinitionSource">
      <value>
         <!--cz.vutbr.xsimko02.service bylo zkráceno z důvodu úspory místa-->
         cz..IPTVManager.createNewServerInfo=PROVIDER_ADD,PROVIDER_DEL,ADMIN
         cz..IPTVManager.getAllServerInfo=PROVIDER_ADD,PROVIDER_DEL,ADMIN
         cz..IPTVManager.deleteServerInfo=PROVIDER_DEL,ADMIN
         cz..IPTVManager.getAllUsers=ADMIN
         cz..IPTVManager.updateUserAccount=ADMIN
         cz..IPTVManager.saveOrUpdateNews=ADMIN
         cz..IPTVManager.deleteNews=ADMIN
         cz..IPTVManager.saveOrUpdateCategory=PROVIDER_ADD,PROVIDER_DEL,ADMIN
         cz..IPTVManager.deleteCategory=PROVIDER_ADD,PROVIDER_DEL,ADMIN
      </value>
   </property>
</bean>
```

Metody uvedené v objectDefinitionSource mohou volat jen autentizovaní uživatelé s oprávněním uvedeným za názvem metody.

4.3 Bezpečnost v datové vrstvě

Aby byla hesla uživatelů chráněná i před těmi, kteří mají přístup k databázi, ve které jsou uchována, ukládají se v podobě otisku (hash) s přídavkem soli. Jako hashovací algoritmus je použit MD5 (Message-Digest algorithm 5), jehož výstup (proud bajtů) je ještě upraven kódováním base64 na ASCII znaky. Aby dvě stejná hesla neměla stejný otisk, přidává se hashovací funkci takzvaná sůl, kterou v tomto případě tvoří uživatelské jméno.

Jelikož aplikace používá rámec Hibernate, je automaticky odolná také proti útokům zvaným SQL injection. Jak bylo vidět v kapitole 3 v části zabývající se datavou vrstvou, Hibernate používá vlastní techniku práce s daty, která z principu neumožňuje měnit záškodnicky SQL dorazy.

5 NASAZENÍ APLIKACÍ NA SERVER

Tato kapitola se zabývá deployem (nasazením) IPTV Serveru a IPTV Manažera na servletový kontejner Tomcat. Každá webová aplikace musí dodržet určitou adresářovou strukturu, aby ji server byl schopen obsloužit. Této struktuře se říká WAR (Web Application Archive) a jejími nezbytnými částmi jsou adresář WEB-INF a v něm soubor web.xml, ve kterém se nachází základní konfigurace webové aplikace.

5.1 Nasazení IPTV Serveru

WAR IPTV Serveru má následující strukturu:

```
iptvServer
   |
   | --- WEB-INF
   |   | --- classes
   |   | --- lib
   |   | --- jsp
   |   web.xml
   |   applicationContext.xml
   |   iptvServer.log
   |   jdbc.properties
   |   log4j.properties
   | --- index.jsp
```

V adresáři classes a lib leží všechny třídy a knihovny, které IPTV Server používá, v jsp jsou jsp stránky. Soubor index.jsp, který leží v kořenu adresáře aplikace, je veřejně přístupný přes URL aplikace a slouží pouze k přesměrování na \listVideoItems.do, kde už se o zpracování postará Spring MVC.

Nasazení IPTV Serveru se skládá z následujících kroků:

1. Do adresáře TOMCAT_HOME\webapps zkopírovat celou adresářovou strukturu WAR.

2. Vytvořit v MySQL novou databázi a nastavit jí kódování UTF-8.

   ```
   CREATE DATABASE iptvserver CHARACTER SET utf8 COLLATE 'utf8_czech_ci';
   ```

3. v souboru jdbc.properties vyplnit údaje o databázi:

   ```
   jdbc.driverClassName=com.mysql.jdbc.Driver
   jdbc.url=jdbc:mysql://localhost:3306/iptvserver?characterEncoding=UTF-8
   jdbc.username=iptvserver
   jdbc.password=iptvserver
   ```

4. v souboru applicationContext.xml odkomentovat řádek:

   ```
   <prop key="hibernate.hbm2ddl.auto">create</prop>
   ```

   který po restartu Tomcatu způsobí vytvoření všech tabulek v databázi.

5. Pomocí programu keytool, který je součástí JDK (Java Development Kit) vytvořit nové úložiště certifikátů (keystore) a certifikát:

```
keytool -genkey -v -keyalg RSA -keystore keystore -dname "CN=IPTVServer,
      OU=FEKT, O=VUT, L=Brno, ST=Czech, C=CZ"
```

6. Vytvořený keystore umístit někam na Tomcat (např. TOMCAT_HOME\ shared\ iptvCertificates\ keystore) a do konfiguračního souboru Tomcatu (TOMCAT_HOME\conf\server.xml) přidat konektor pro SSL/TLS:

```
<Connector SSLEnabled="true" clientAuth="false"
   keystoreFile="TOMCAT_HOME/shared/iptvCertificates/keystore"
   keystorePass="password" maxThreads="150" port="8443" protocol="HTTP/1.1"
   scheme="https" secure="true" sslProtocol="TLS" />
```

7. Kvůli http Basic autentizaci, kterou je IPTV Server zabezpečen, se musí do souboru TOMCAT_HOME\conf\tomcat-users.xml přidat role iptvManager a všechny uživatelské jména a hesla, které budou moci komunikovat s IPTV Serverem, např.:

```
   <role rolename="iptvManager" />
   <user username="admin" password="heslo" roles="iptvManager" />
```

8. Restartovat Tomcat a v souboru applicationContext.xml zakomentovat řádek:

```
<!-- <prop key="hibernate.hbm2ddl.auto">create</prop> -->
```

5.2 Nasazení IPTV Manažera

WAR IPTV Manažera má následující strukturu:

```
iptvManager
   | --- images
   | --- javascripts
   | --- styles
   | --- WEB-INF
   |   | --- classes
   |   | --- lib
   |   | --- jsp
   |   web.xml
   |   applicationContext-acegi-security.xml
   |   applicationContext-authorization.xml
   |   applicationContext-hiberanate.xml
   |   applicationContext-remote.xml
   |   applicationContext.xml
   |   iptvManager.log
   |   jdbc.properties
   |   security.properties
   |   log4j.properties
   | --- index.jsp
```

V adresáři images jsou obrázky, které jsou součástí web designu (obrázky video pořadů jsou uloženy v databázi). Do souboru iptvManager.log se logují informace, jejichž úroveň a další nastavení se nachází v souboru log4j.properties.

Nasazení IPTV Manažera se skládá z těchto kroků:

1. Do adresáře TOMCAT_HOME\webapps zkopírovat celou adresářovou strukturu WAR.

2. Vytvořit v MySQL novou databázi a nastavit jí kódování UTF-8.

```
CREATE DATABASE iptvmanager CHARACTER SET utf8 COLLATE 'utf8_czech_ci';
```

3. v souboru jdbc.properties vyplnit údaje o databázi:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/iptvmanager?characterEncoding=UTF-8
jdbc.username=iptvmanager
jdbc.password=iptvmanager
```

4. v souboru applicationContext-hiberanate.xml odkomentovat řádek:

```
<prop key="hibernate.hbm2ddl.auto">create</prop>
```

5. Použít buď již vytvořený keystore při deploymentu IPTV Serveru nebo vytvořit nový a umístit ho někam na Tomcat (např. TOMCAT_HOME\ shared\ iptvCertificates\ keystore).

6. Jelikož IPTV Server nemá certifikát podepsaný žádnou CA, je potřeba tento certifikát z keystore exportovat a následně importovat do úložiště důvěryhodných certifikátů (truststore).

```
keytool -export -rfc -keystore keystore -alias mykey -file iptvServer.cer
keytool -import -file iptvServer.cer -keystore truststore
```

7. Do konfiguračního souboru Tomcatu (TOMCAT_HOME\conf\server.xml) přidat konektor pro SSL/TLS:

```
<Connector SSLEnabled="true" clientAuth="false"
    keystoreFile="TOMCAT_HOME/shared/iptvCertificates/keystore"
    keystorePass="password" maxThreads="150" port="8443" protocol="HTTP/1.1"
    scheme="https" secure="true" sslProtocol="TLS" />
    truststoreFile="TOMCAT_HOME/shared/iptvCertificates/truststore"
    truststorePass="changeit" />
```

8. v souboru security.properties nastavit cestu a heslo k truststore:

```
security.trustStore=TOMCAT_HOME/shared/iptvCertificates/truststore
security.trustStorePassword=changeit
```

9. Restartovat Tomcat a v souboru aapplicationContext-hiberanate.xml zakomentovat řádek:

```
<!-- <prop key="hibernate.hbm2ddl.auto">create</prop> -->
```

5.3 Možné komplikace

Po deploymentu aplikace se objevily následující dvě komplikace:

1. Aplikci se vůbec nepodařilo nastartovat, stacktrace vypsal následující informace:

```
ERROR [Thread-1] (ContextLoader.java:209) - Context initialization failed
java.lang.UnsupportedClassVersionError: Bad version number in .class file
```

   Příčina - Aplikace byla vyvíjena v IDE (Integrated development environment) Eclipse, který používal ke kompilování tříd Javu verze 6.0. Zkompilované třídy pak byly zkopírovány na server, který používá Javu verze 5.0. Tyto dvě verze nejsou na úrovni byte kódu kompatibilní a proto se nepodařilo kontext vůbec inicializovat.
   Řešení - v Eclipse se nastavilo, aby používal ke kompilování javu verze 5.0 a celý projekt se překompiloval.

2. Aplikace nastartovala, ale skončila výjimkami:

```
java.io.FileNotFoundException: TOMCAT_HOME/work/Catalina/localhost/iptvManager
        /tldCache.ser (Permission denied)
java.io.FileNotFoundException: TOMCAT_HOME/logs/catalina.out (Permission
        denied)
```

   Příčina - Přístupová práva v Linuxu.
   Řešení - Změnit přístupová práva souborů tldCache.ser a catalina.out tak, aby do nich mohla aplikace zapisovat.

Vyzkoušeno s Apache Tomcat 6.0.14 a MySQL 5.0.45.

6 TESTOVÁNÍ

Velmi důležitou etapou při vývoji software je testování. Obecně platí, že aplikace může mít skvělou architekturu, design i implementaci, ale pokud není správně testována, nemůže být považována za úspěšný produkt.

Podle rozsahu testu a způsobu testování rozlišujeme tři základní druhy testů.

6.1 Jednotkové testování (Unit Testing)

Jednotkové testy se používají pro testování určitých jednotek (tříd) v aplikaci a umožňují rychle otestovat jakýkoliv nový kód nebo změny v existujícím kódu, bez nutnosti konfigurovat server nebo deployování aplikace. Pokud je aplikace dostatečně pokryta testy (čím více, tím lépe), je také daleko jednodušší a příjemnější její další rozšiřování a pozměňování, neboť testy ohlídají, aby nebyla narušena stávající funkcionalita. Testy jsou také velmi užitečné při hledání a opravování bugů (chyb) v aplikaci a slouží také jako výborná dokumentace pro vývojáře, který by chtěl znovupoužít nějakou naší komponentu. V agilních metodikách vývoje, jako je např. Extrémní programovaní (XP) nebo Programovaní řízené testy (TDD), se testy píší dokonce ještě dříve než samotný testovaný kód.

Pro provádění jednotkových testů se na platformě Java nejvíce používá rámec JUnit, který je ve verzi 3 využit i v této aplikaci.

6.1.1 JUnit

JUnit je open source testovací rámec sloužící pro psaní opakovatelných testů. JUnit mimo jiné obsahuje funkcionalitu pro porovnávání očekávaných hodnot se skutečnými, příslušenství pro sdílení společných testovacích dat, umožňuje tvořit sady testů pro snadné spouštění testů a spoustu další funkcionality. Práci s tímto rámcem navíc ulehčují IDE nástroje (Eclipse, NetBeans, atd), které mají integrovánu jeho podporu. [10]

Podstata jednotkových testů je otestovat jednotlivé objekty samostatně a izolovaně od ostatních objektů, které s nimi v aplikaci normálně spolupracují. Díky tomu je jednotkové testování rychlé a poměrně jednoduché. K odizolování testovaných objektů se používá technika tzv. mock objektů.

6.1.2 Mock objekty

Když je potřeba testovat např. metodu třídy v aplikační vrstvě, která se dotazuje DAO objektů z datové vrstvy, musí se nejprve složitě připravovat okolní podmínky

(práce s DB, atd.). Ošetření těchto okolních podmínek může nakonec zabrat několikrát více času, než je potřeba k napsaní vlastní testovací logiky. Při použití mock techniky se v tomto případě objekty z DAO vrstvy nahradí mock (falešnými) objekty, které neprovádí žádnou funkcionalitu nahrazovaného objektu, jen se jako tento objekt tváří. Místo původní logiky objektu je vloženo chování, které je v testu potřeba. Při použití této techniky se tedy soustředíme pouze na testování logiky objektu z aplikační vrstvy a předpokládáme, že okolní objekty fungují správně tak jak mají.

6.2 Integrační testování (Integration Testing)

Po úspěšném provedení všech jednotkových testů následuje integrační testování, které má za úkol potvrdit, že komponenty (objekty) systému správně spolupracují. Integrační testy jsou také nezbytné v datové vrstvě, kde implementace DAO rozhraní nemohou být jednotkovými testy účinně otestovány. Další cíle integračního testování jsou aspekty jako vzdálené volání (remote service), O/R mapování nebo transakce. Integrační testy jsou komplexní a velmi užitečné, mají však nevýhodu, že je poměrně obtížné je napsat a že jejich vykonávání trvá dlouhou dobu (práce s DB, atd.). Oba tyto problémy nám pomáhá řešit Spring.

6.2.1 Podpora Springu pro testování

Rámec Spring byl přímo navržen tak, aby umožňoval efektivní testování mimo kontejner. Jak už bylo napsáno výše, Spring je neinvazivní, takže nevzniká žádná závislost aplikačního kódu na rámci. Což umožňuje pohodlně testovat aplikační objekty jako obyčejné Java třídy (POJOs) bez nutnosti konfigurace kontejneru. Díky IoC je navíc velmi ulehčeno použití mock objektů.

Spring také poskytuje řadu tříd, které rozšiřují JUnit API a které testování značně ulehčují a urychlují. Jednou z nich je např. `AbstractDependencyInjectionSpringContextTests`, použitá ve třídě `IPTVManagerImplTest`:

```
public class IPTVManagerImplTest extends
      AbstractDependencyInjectionSpringContextTests {

   /* iptvManager nastaven pomocí DI díky
      AbstractDependencyInjectionSpringContextTests */
   public void setIptvManager(IPTVManagerImpl iptvManager) {
      this.iptvManager = iptvManager;
   }

   public void testCreateNewUserAccount() throws Exception {
      User user = new User();
      user.setUserName("dan");
      user.setPassword("heslo");
      iptvManager.createNewUserAccount(user);
      // nenulovost id zaručí správné uložení objektu do DB
      assertNotNull(user.getId());
```

```
        // kontrola vytvoření správné výše kreditu
        assertEquals(200, user.getCredit().getAmount().intValue());
        // kontrola uložení hesla v zašifrované podobě
        assertNotSame("heslo",user.getPassword());
    }
    ...
}
```

Metoda testCreateNewUserAccount má za úkol otestovat správné vytvoření a uložení nového uživatelského účtu. Urychlení integračního testování je založeno na tom, ze testovací metoda běží v transakci, která se ve výsledku rollbackne. Tím pádem nedojde k modifikaci stavu databáze a z toho vyplývá, že se nemusí provádět příprava dat (setup/cleanup) před a po každém testu.

6.3 Akceptační testování (Acceptance tests)

Jedná se vlastně o sadu testů, které mají potvrdit celkovou funkcionalitu aplikace. Tyto testy by měl na základě use cases diagramů připravovat analytik nebo nejlépe přímo zadavatel projektu. Testuje se společně chování uživatelského rozhraní a aplikační logiky. Existuje spousta open source nástrojů, které toto dokáží testovat automaticky. Jedním z nich je např. Selenium.

6.3.1 Selenium

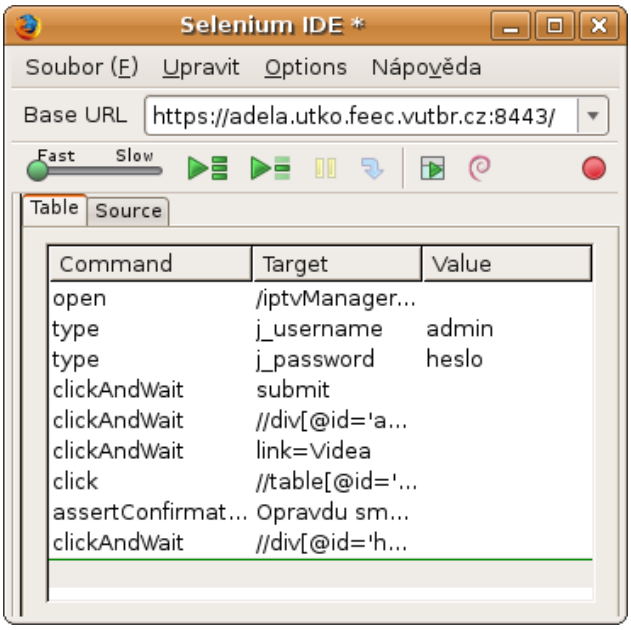
Selenium je open source nástroj pro testování webových aplikací. K vytváření testů není potřeba znalost Javy ani programování. Testy lze jednoduše tvořit tak, že přímo ve webovém prohlížeči provádíme různé akce, které Selenium IDE plug-in zaznamenává. Takto zaznamenaný scénář lze později automaticky spouštět (obr. 6.1) a testovat tak, zda se funkcionalita nezměnila.

Při vývoji aplikace bylo tímto způsobem testováno, zda nedošlo k narušení funkčnosti, při různých úpravách a refaktoringu[1] kódu.

6.3.2 JMeter

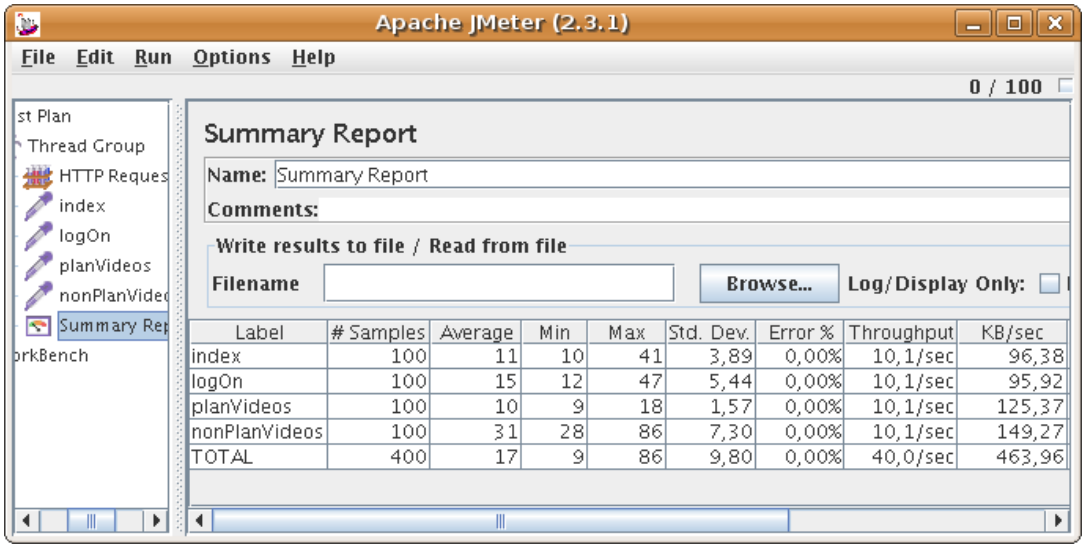
Open source Apache JMeter je nástroj pro testování chování aplikace pod zátěží a pro měření výkonnosti aplikace. JMeter umí nasimulovat reálnou zátěž webové aplikace a vyhodnotit, jak rychle dokázala aplikace požadavky obsloužit. JMeter by při testování měl být spuštěn na jiném stroji než testovaná aplikace. Vytváří totiž velké množství vláken (jedno vlákno = jeden simulovaný uživatel), které mohou spotřebovat značné množství systémových prostředků a zhoršit tak odezvu aplikace.

[1] Refaktoring je změna stávajícího kódu za účelem zlepšení jeho čitelnosti nebo strukturovanosti, přičemž funkčnost zůstává beze změny.



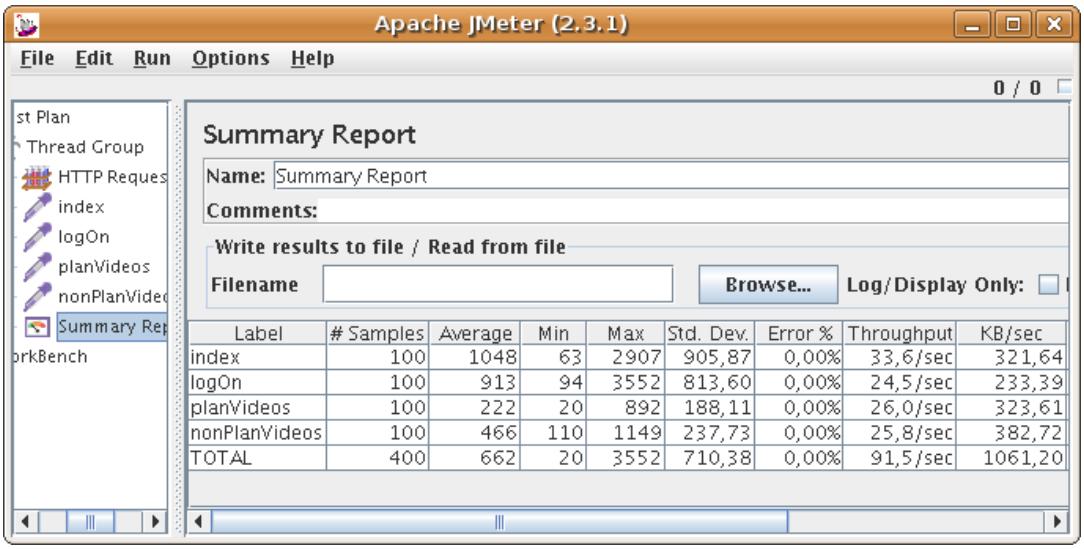
Obr. 6.1: Selenium IDE plug-in pro prohlížeč Mozilla Firefox.

Přesto bylo testování IPTV Manažera provedeno na localhostu, protože testování na vzdáleném serveru přes síť internet by výsledky zkreslilo pravděpodobně ještě více. Výsledky testů jsou na obrázcích 6.2 a 6.3.



Obr. 6.2: JMeter test se 100 uživateli během 10s.

Obrázek 6.2 zachycuje chování IPTV Manažera, kdy k němu během 10 vteřin přistoupilo 100 uživatelů a každý si nejříve zobrazil hlavní stránku (index), poté se zalogoval a zobrazil si nabídku naplánovaných a nenaplánovaných pořadů. Obrázek 6.3 ukazuje výsledky stejného scénáře, s tím rozdílem, že všech 100 uživatelů



Obr. 6.3: JMeter test se 100 uživateli v jeden okamžik.

přistoupilo k aplikaci v jeden okamžik. Tabulky na obrázcích zobrazují souhrné informace o prevedených testech. Sloupečky Average (střední doba přístupu na server), Min, Max i Std. Dev. (směrodatná odchylka) jsou vyjádřeny v milisekundách. Error vyjadřuje procentuální počet nevyřízených požadvků a Throughput (propustnost) udává, kolik požadavků za vteřinu byl server schopen obsloužit. Kb/sec je propustnost vyjádřená v kilobajtech za vteřinu.

Výkon aplikace by šlo ještě výrazně zvýšit, pokud by nebyla nasazena na Tomcat, ale na některý výkonnější server (např. Sun Web Server).

7 ZÁVĚR

Cílem této práce bylo navrhnout a implementovat pro skupinu Multicast IPTV Research Group [14] aplikaci IPTV Manažera.

Před samotným návrhem aplikace bylo nutné provést důkladnou analýzu, jejímž hlavním úkolem bylo identifikovat podstatné a nutné vlastnosti aplikace, poznat veškerou její funkcionalitu a definovat všechny typy uživatelů, kteří budou s aplikací pracovat. Na základě této analýzy (popsané v kapitole 1) byla zvolena klasická třívrstvá architektura aplikace, která definuje prezentační, aplikační a datovou vrstvu. Oddělení jednotlivých vrstev, mezi kterými je minimální vazba, má velkou výhodu při pozdějších úpravách a rozšiřování aplikace. Vše je totiž jasně definované (na základě Java rozhraní), přehledné a soustředěné na jednom místě. Implementace takovéto vícevrstvé aplikace se ovšem neobejde bez použití některého aplikačního rámce, který se prakticky postará o „poslepování“ jednotlivých vrstev v celou aplikaci. V současné době je nejvíce oblíben open-source rámec Spring, který byl také použit v této práci. Spring mimo jiné umožňuje použít k implementaci jednotlivých vrstev specializované rámce, zaměřené právě na jednu konkrétní část třívrstvého modelu. Proto byl v prezentační vrstvě použit rámec Stripes, který při dodržení určitých konvencí, při pojmenovávání tříd a jsp souborů, nepotřebuje žádnou další konfiguraci, což poměrně zrychlí vývoj. Datová vrstva využívá mocný objektově relační mapovací nástroj Hibernate, který výrazně pomáhá řešit problém propojení relačního a objektového světa. Bezpečnost, na kterou byl v analýze kladen velký důraz, je potřeba řešit v každé vrstvě aplikace. Což by bez specializovaného rámce bylo velmi náročné nebo nepřehledné, proto byl použit rámec Acegi Security, který je v současné době v aplikacích postavených nad technologií Spring standardem. Za zmínku jistě stojí i fakt, že požité technologie Spring, Acegi a Hibernate se používají i v takových aplikacích, jako jsou např. bankovní systémy.

Aby mohla být funkčnost IPTV Manažera vyzkoušena a předvedena, byl implementován i jednoduchý IPTV Server, který původně neměl být součástí této práce. Podrobný návod, jak obě aplikace nainstalovat na jakýkoliv servletový kontejnér, který splňuje Java Servlet specifikaci verze 2.4 (např. Tomcat) je obsahem 5. kapitoly.

V kapitole 6 jsou rozebrány jednotlivé techniky testování, které byly při vývoji aplikace použity.

LITERATURA

[1] Johnson, R. A Hoeller, J. A Arendsen, A. A Risberg, T. A Sampaleanu, C. *Professional Java Development with the Spring Framework*. Willey Publishing, Inc., 2005, ISBN: 978-0-7645-7483-2

[2] Harrop R. A Machacek J. *Pro Spring*. Apress Publishing, 2005, ISBN: 1-59059-461-4

[3] Bauer, Ch. A King, G. *Java Persistence with Hibernate*. Manning Publications, 2007, ISBN: 1-932394-88-5

[4] *The Source for Java Developers* [online]. Dostupné na URL: <http://java.sun.com>.

[5] *Spring Application Framework* [online]. Dostupné na URL: <http://www.springframework.org>

[6] *Relational Persistence for Java and .NET* [online]. Dostupné na URL: <http://www.hibernate.org/>

[7] *Acegi Security System for Spring* [online]. Dostupné na URL: <http://www.acegisecurity.org>

[8] *Stripes presentation framework* [online]. Dostupné na URL: <http://mc4j.org/confluence/display/stripes/Home>

[9] *Apache Tomcat* [online]. Dostupné na URL: <http://tomcat.apache.org/>

[10] *JUnit testing framework* [online]. Dostupné na URL: <http://www.junit.org/>

[11] *Selenium test tool for web applications* [online]. Dostupné na URL: <http://www.openqa.org/selenium/>

[12] *Apache JMeter* [online]. Dostupné na URL: <http://jakarta.apache.org/jmeter/>

[13] *DSL Forum* [online]. Dostupné na URL: <http://www.dslforum.org/>

[14] *Multicast IPTV Research Group* [online]. Dostupné na URL: <http://adela.utko.feec.vutbr.cz/projects/>

[15] *Dagblog – blog nejen pro kodery* [online]. Dostupné na URL: <http://www.sweb.cz/pichlik>

[16] *Unified Modeling Language* [online]. Dostupné na URL: <http://www.uml.org/>

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ACL Access Control List

AOP Aspect Oriented Programming

API Application Programming Interface

DAO Data Access Object

DI Dependency Injection

EJB Enterprise JavaBeans

IDE Integrated development environment

IoC Inversion of Control

IPTV Internet Protocol TeleVision

JAAS Authentication and Authorization Service

JDK Java Development Kit

JEE Jave Enterprise Edition

JVM Java Virtual Machine

OOP Objektově orientované programování

POJO Plain Old Java Object

SDK Software Development Kit

UML Unified Modeling Language

URL Uniform Resource Locator

WAR Web Application Archive