České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra počítačů

Bakalářská práce

Správa vzdáleného souborového systému přes webový prohlížeč

Marek Fišera

Vedoucí práce: Ing. Tomáš Černý

Studijní program: Softwarové technologie a management, strukturovaný bakalářský

Obor: Softwarové inženýrství
květen 2011

PODĚKOVÁNÍ

Tímto bych rád poděkoval svému vedoucímu Ing. Tomáši Černému, za ochotu, pomoc a čas který mi věnoval.

PROHLÁŠENÍ

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 20.5.2011 .....

ABSTRACT

The main goal of this thesis is development of application for filesystem management. Application will have user interface similar to user interface in graphic operating system. Beside agenda of filesystem management application will also offer simple user management. The first part of thesis is focused on comparision of existing solutions. The second part then deals with analysis, choice of suitable frameworks for each part of application and actual implementation. The implementation part is focused on development of service layer framework and communication between klient and server.

ABSTRAKT

Hlavním cílem práce je vytvoření aplikace se zaměřením na správu souborového systému. Aplikace bude mít uživatelské rozhraní podobné grafickým operačním systémům. Kromě agendy pro správu souborového systému bude nabízet jednoduchou správu uživatelů. První část se zabývá porovnáním s již existujícími řešení. Druhá část se pak věnuje počáteční analýze, výběru vhodných frameworků pro jednotlivé části aplikace a vlastní implementaci. V implementační části se zaměřuje na implementaci frameworku pro servisní vrstvu a na komunikaci mezi klientem a serverem.

OBSAH

SEZNAM OBRÁZKŮ

KAPITOLA 1
ÚVOD

Práce se zabývá analýzou a implementací aplikace pro správu souborového systému pomocí webového prohlížeče. První část, Kapitola 2, porovnává aplikace, které danou problematiku již řeší. Na internetu je dostupných několik řešení, tato řešení většinou nabízejí možnost z uživatelského pohledu prozkoumat. Druhá část se pak zaměřuje nejprve na alanýzu systému. Za tímto účelem byly vytvořeny UML diagramy modelující danou problematika. Následuje volba vhodných technoligí pro vývoj aplikace. V Kapitole 4 je shrnut vývoj aplikace, hlavní důraz je kladen na implementace frameworku pro servisni vrstvu aplikace a na komunikaci mezi klientem a serverem. Kapitola 5 pak shrnuje testování aplikace jak při vývoji, tak formou beta testování.

Práce obsahuje uživatelskou příručku, která poskytuje popis uživatelského rozhraní a popis jednotlivých aplikací a jejich funkcionality, viz Příloha C. V Příloze D je pak příručka jak aplikaci nainstalovat, včetně všech potřebných aplikací, a spustit.

1.1. MOTIVACE

Práce vnikla jako základní stavební kámen pro informační systém. Služby, které poskytuje, jako je správa souborového systému a uživatelů, se budou dát využít v dalších agendách. Z tohoto důvodu bylo extrémně důležité vybrat technologie a architekturu umožňující snadnou rozšiřitelnost aplikace podle dalších požadavků. Dalším důležitým faktorem je možnost spolupráce s externími aplikacimi, proto je důležité aby aplikace poskytovala API, které bude dále možné využít.

KAPITOLA 2
POROVNÁNÍ S JIŽ EXISTUJÍCÍMI ŘEŠENÍMI

2.1. NA CO SE ZAMĚŘIT

Při hledání již existujících řešení jsem se zaměřil dva faktory, prvním bylo uživatelské rozhraní, které by mělo připomínat plochu známou z Windows nebo grafického prostředí pro Linux, druhý pak implementace přenosu dat po síti a struktura serverové části.

Co se týká prvního faktoru, zde bylo vše jednodušší, většina aplikací má volně dostupnou demoverzi, která nabízí dostatek informací.

Naopak ve velké většině případů struktura serverové části již dostupná nebyla, snad jedinou vyjímkou byla aplikace Cloudo, která poskytuje stručný přehled o tom, jak je serverová část koncipována.

2.2. EXISTUJÍCÍ ŘEŠENÍ

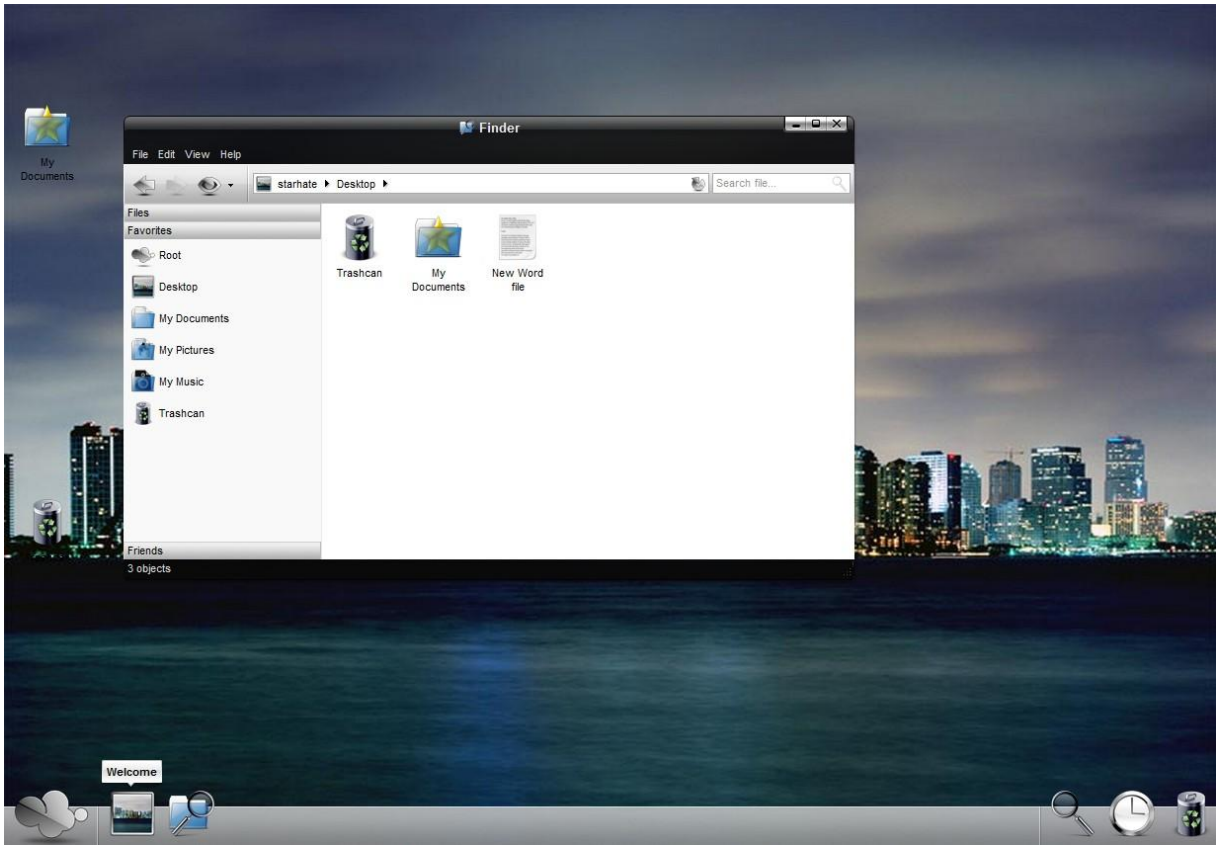
2.2.1. Web File Explorer

Tato aplikace nabízí základní rozhraní pro práci se souborovým systémem. Umožňuje vytvářet a spravovat složky a soubory. Nabízí takové možnost stáhnout obsah celé složky ve formátu zip.

Komunikace se serverem probíhá klasickým způsobem. Při požadavku na obsah složky je serveru předán parametr s cestou k požadované složce, na serveru tento požadavek zpracuje dynamická stránka (ASP skript), který vygeneruje výsledné XHTML s obsahem složky, které server odešle zpět klientovi, který ho zobrazí do jednoho z rámců framesetu.

Tento přístup vyžaduje přenesení zbytečného množství dat mezi klientem a serverem, požadavek na obsah složky s 10 položkami vyžaduje přenesení okolo 65KB dat. Navíc další požadavky na server jsou generovány na další zdroje, které vygenerované XHTML obsahuje.

2.2.2. Cloudo



Obrázek 2.1 Klientské rozhraní aplikce Cloudo

Cloudo komplexní rozhraní operačního systému zabudovaného do webového prohlížeče, viz Obrázek 2.1. Uživatelské rozhraní je velmi podobné známým grafickým operačním systémům. Toto rozhraní nabízí uživateli kompletní možnost práce se souborovým systémem, krom toho také nabízí několik "programů", ve kterých je možné s určitými druhy souborů pracovat přímo v okně prohlížeče.

Komunikace se serverem pak neprobíhá klasickým způsobem. Ke kominukaci je využívána výhradně metoda POST, jako datový formát je používáno XML. Komunikace pak probíhá tak, že je vždy server odeslán XML dokument, nikoli je hodnoty jednotlivých formulářových polí. Ten se na serveru zpracuje a odpověď je opět odeslána ve formátu XML. Tato komunikace sice přináší mírně větší množství odeslaných dat na server, ale výrazně redukuje data odesílaná serverem zpět ke klientovi. Jedna odpověď od serveru pak obsahuje řádově jednotky KB, nečastěji okolo 2KB dat.

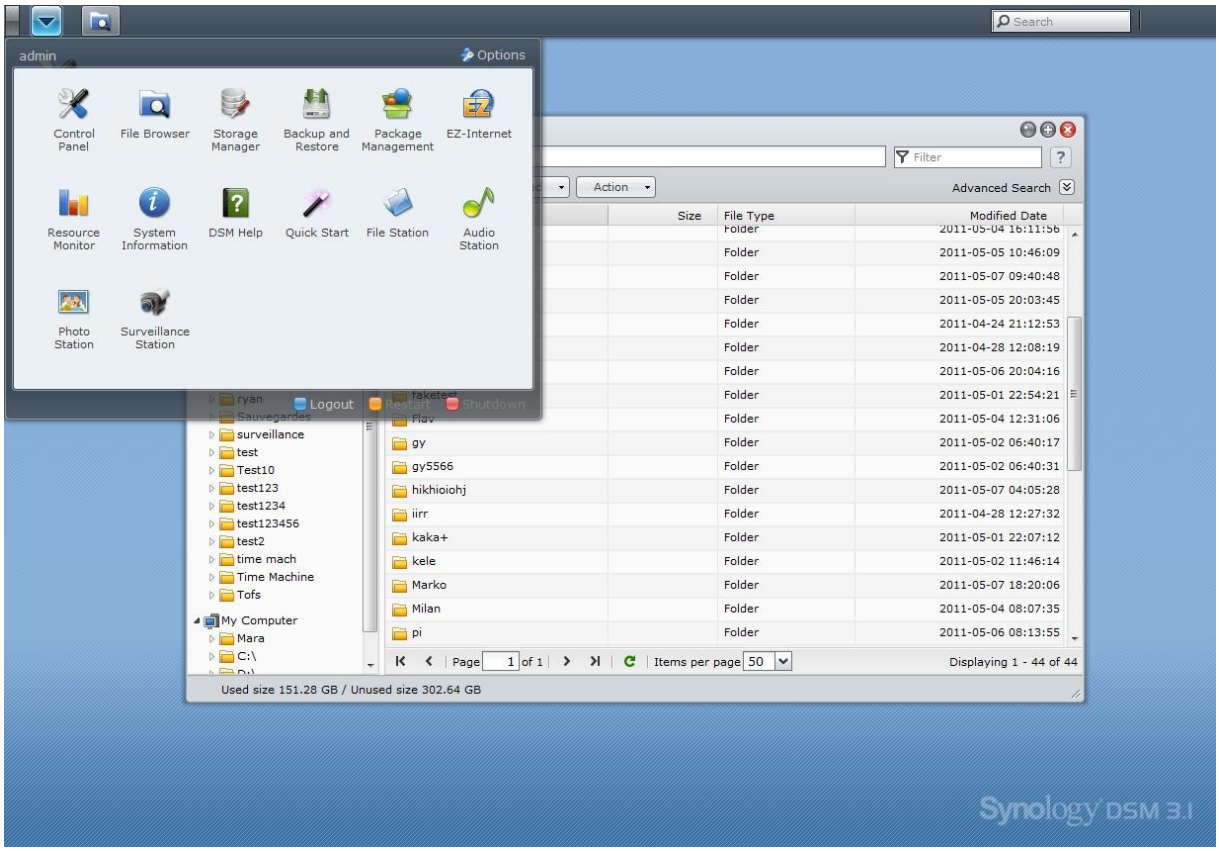
Celá tato aplikace je postavena na XML a XSLT transformaci. Tedy i uživatelské aplikace jsou tvořeny směsicí XML, XHTML a Javascriptu, obvykle namíchaného do jednoho souboru dané aplikace. Tento přístup znepřehledňuje vývoj větších aplikací.

2.2.3. CloudMe

Tato aplikace funguje obdobně jako Cloudo. Navíc nabízí více uživatelských rozhraní, krom desktopového například také konzoloné nebo například rozhraní pro mobilní telefony. Také paleta online aplikací je širší než v případě aplikace Cloudo. Krom kominukace po HTTP protokolu, také nabízí možnost interakce se systémem po protokolu WebDAV.

Komunikace se serverem opět probíhá výhradně pomocí metody POST, je využíván protokol SOAP. Opět tedy server vrací čistá data a umožňuje klientovi je zpracovat dle potřeby.

2.2.4. Synology



Obrázek 2.2 Klientské rozhraní aplikace Synology DSM

Synology DSM, viz Obrázek 2.2, slouží jako administrační panel pro správu serverů. Grafické prostředí využívá klihovnu ExtJS, která nabízí množství uživatelských komponent. Při komunikaci se serverem odesílá data opět POST metodou, nicméně využívá klasických proměnných z předáváných z odesílaného formuláře, odpověď od serveru je pak ve formátu JSON, který oproti formátu XML je na přenášená data ještě úspornější. V prohlížeči souborového systému stojí za povšimnutí použití Java Appletu, který umožňuje přimo v prohlížeči procházet lokální souborový systém a soubory nahrát na server pomocí “drag-and-drop” funkcionality.

2.3. SHRNUTÍ

Porovnávané aplikace mají velmi odlišný přístup k řešení dané problematiky.

2.3.1. Komunikace se serverem

Aplikace Web File Explorer generuje celé XHTML na serverové straně. Klient tedy při zpracování odpovědi od serveru musí pouze zobrazit dané XHTML. Tímto jsou zaručeny menší nároky na výkon klienského počítače.

Na druhé straně aplikace jako Cloudo nebo Synology nabízí uživateli čistě datově orientované rozhraní, jak prijatá data klient zpracuje již nechávají na něm samotném. Tento přístup šetří serverové zdroje, protože server není nucen vygenerovat výsledné XHTML, a přenáší část výpočetního zatížení na klienta. Toto již v dnešní době nebývá problém, vývoj Javascriptu a hlavně rychlosti jeho zpracování, je jedním z klíčových faktorů vývoje internetových prohlížečů.

2.3.2. Uživatelské rozhraní

I zde je jasná výhoda uživatelského rozhraní sestaveného až na klientovi pomocí Javascriptu. Toto zlepšuje interakci z uživatelem a zjednodušuje vývoj. Vhodná implementace AJAX technologie a vlastnosti webového rozhraní, které se více podobá tomu desktopovému, jsou jednodušeji zanesitelné do rozhraní, které je generováno na klientovi. V takovém případě máme kromě dalšího rovnou k dispozici reference na všechny vytvořené prvky, nemusíme je tedy teprve získávat pomocí indetifikátorů. Další výhodou je možnost prizpůsobit aplikaci klientově prohlížeči, tak, aby zobrazila optimálně. Při serverovém generování rozhraní sice máme také k dispozici informace o uživatelově prohlížeči, ale není jich z daleka tolik.

KAPITOLA 3
ANALÝZA

V této kapitole definujeme požadavky na systém, odvodíme případy užití. Na základě těchto požadavků a porovnání s již existujícími řešeními z minulé kapitoly vybereme technologie pro serverovou a klientskou část.

3.1. KATALOG POŽADAVKŮ

3.1.1. Funkční požadavky

- Aplikace bude přístupná přes webové rozhraní
- Správa souborového systému
- Správa uživatelských účtů
- API serverové části pro přístupnost dat
- Jednoduchá rozšiřitelnost
- Flexibilita pro datové formáty

3.1.2. Nefunkční požadavky

- Webový prohlížeč

3.2. AKTÉŘI SYSTÉMU A PŘÍPADY UŽITÍ

3.2.1. Nepřihlášený uživatel (viz obrázek A.1)

Jedinou funkcionalitou pro nepřihlášeného uživatele bude možnost se přihlásit.

3.2.2. Přihlášený uživatel (viz obrázek A.2 a A.3)

Přihlášený uživatel má k dispozici celé rozhraní aplikace obsahující množinu funkcí pro správu souborového systému, tedy možnost prohlížet, editovat, mazat, přesouvat a kopírovat složky a soubory, spravovat si svoje jednotky a textové a HTML soubory editovat. Dále pak fukce pro vytváření, editaci a mazání uživatelských účtů. Všechny tyto funkce se vztahují pouze na objekty, tedy uživatelské účty, soubory a složky, ke kterým má uživatel příslušná oprávnění.

3.3. DOMÉNOVÝ MODEL

Doménový model je konceptuální datový model (obrázek A.4). Tento model nám ještě neříká nic o tom, jak bude model konkrétně implementovám v relační databázi, ale spíše zobrazuje model z pohledu reálného světa.

3.4. RELAČNÍ DATABÁZOVÝ MODEL

Relační databázový model (obrázek A.6) již přesně popisuje strukturu a vztahy mezi prvky relační databáze. Tento model je již oproti doménovému modelu konkrétní implementací řešeného problému.

3.5. Architektura aplikace

Architektura aplikace je zobrazena na obrázku A.5. Celá aplikace je rozdělena do dvou částí. Obě části budou vyvíjeny separovaně, tímto přístupem zajistíme snazší splnění požadavků na vytvoření serverového API, protože vytvořené API rovnou otestuje zda poskytuje dostatečnou flexibilitu a použitelnost. Pokud by jsme API vyvíjeli bez vlastního velkého použití, mohlo by se snadno stát, že API nebude dostatečné flexibilní.

3.5.1. Serverová část

Serverová část aplikace bude běžet na vzdáleném serveru, ke kterému se budou uživatelé připojovat. S klientskou aplikací bude komunikovat po HTTP protokolu. Bude se skládat z několika vrstev:

ORM vrstva - tato vrstva se stará o komunikaci s uložištěm dat. Přináší nám vrstvu abstrakce mezi doménovým modelem a uložištem dat. Poskytuje mapování mezi datovým uložištěm a doménovým modelem.

DAO vrstva – tato vrstva slouží v aplikaci jako vrstva aplikační logiky. Komunikuje s ORM vrstvou, avšak není nijak vázána na webovou vrstvu.

Servisní vrstva – vrstva služeb na webové vrstvě, zpostředkovává komunikaci mezi http požadavky od klienta a DAO vrstvou.

3.5.2. Klientská část

Klientská část poběží v uživatelově internetovém prohlížeči. Bude zprostředkovávat uřivatelské rozhraní pro práci s daty získanými ze serverové části aplikace. Bude rozdělna do dvou základních vrstev

Servisní vrstva – ta bude aplikaci zpostředkovávat veškerou komunikaci se serverovou částí po HTTP protokolu.

UI vrstva – vrstva uživatelského rozhraní, která bude nabíze interakci s uživatelem, zpracovávat jeho požadavky a předávat je servisní vrstvě k odeslání na server.

3.6. Výběr technologie – Serverová část

Na straně sereru se nabízí několik svojí strukturou velmi odlišných technologií. Liší se od struktury jazyka, přes poskytované knihovny a knihovny třetích stran, po dostupnost na různých web-hostingových programech.

3.6.1. PHP

PHP je dynamický jazyk používaný především pro vývoj webových aplikací, ale je možné ho použít pro desktopové aplikace. Je z kategorie skriptovacích jazyků. Má dynamickou typovou kontrolu. Je to jazyk, který “se stal“ objektovým, chybí v něm tedy základní objektová struktura, nalezitelná v jazycích, které od svého počátku již byly objektovými.

3.6.2. JAVA

Java je multiplatformní programovací jazyk s dobrou objektovou strukturou. Používá statickou typovou kontrolu. Java EE (1) je používána pro vývoj webových aplikací. Poskytuje dobrou strukturu aplikace a je velmi rozšířen, má dobrou podporu ve vývojových prostředích.

3.6.3. .NET (ASP.NET)

Prostředí .NET podporuje více programovacích jazyků. Jeho v současné době nejvíce podporovaný jazyk je C#, který vychází z jazyku Java. Je velmi intenzivně vyvíjen společností Microsoft. ASP.NET (2) je pak framework postavený na technologii .NET určený pro vývoj webových aplikací.

3.6.4. Shrnutí

Ačkoli technologie .NET je v dnešní době již před technologií Java, pro serverovou část jsem zvolil právě Javu hned z několika důvodů. Vše, od samotné technologie, přes aplikační server a k množství frameworků třetích stran, je ve velké většině volně šiřitelné, je také multiplatformní, bude tedy možné výslednou aplikace používat jak na Windows Serverech, tak na Unixových serverech.

3.7. VÝBĚR TECHNOLOGIE – FRAMEWORKY PRO SERVEROVOU ČÁST

3.7.1. ORM

Pro tuto vrstvu jsem zvolil dnes již standartní vrstvu používanou v Java aplikacích – JPA (1). Tato vrstva navíc nabízí abstrakci nad používaným databázovým serverem, díky které je možné databázový server vyměnit za jiný pouhou změnou konfigurace.

3.7.2. DAO

Pro tuto vrstvu jsem navrhl vlastní rozhraní, které vyhovuje návrhu a základním vlastnostem doménového modelu. Rozhraní pro tuto vrstvu je vyčleněno do vlastní knihovny a je tedy možné ho použít v dalších projektech.

3.7.3. Servisní vrstva

Pro tuto vrstvu jsem navrhl vlastní rozhraní, které poskytuje jednotlivým službám potřebné parametry, transakční zpracování a zpracovává jejich výstup.

3.8. VÝBĚR TECHNOLOGIE – KLIENTSKÁ ČÁST

V klientské části již tak široký výběr technologií není, navíc je zde velký rozdíl mezi jednotlivými operačními systémi a prohlížeči, dokonce i mezi jednotlivými verzemi prohlížečů:

3.8.1. XHTML + Javascript

Tato varianta je v dnešní době nejrozšířenější. Většinu rozhraní tvoří XHTML stránka stažená z web serveru doplněná o Javascript skripty, poskytující uživateli interakci s daty a serverem. Pro projekty s větším množstvím Javascriptových funkcí je však tato varianta hůře spravovatelná.

3.8.2. Silverlight

Nástroj, postavený na technologii .NET, umožňující vytvořit uživatelské prostředí blížící se desktopovému. Vyžaduje však instalovaný plugin v prohlížečích. Má horší podporu na Unixových operačních systémů a v méně používaných prohlížečích. (3)

3.8.3. Flash

Technologie, která je dnes již na ústupu. Obdobně jako Silverlight vyžaduje plugin v uživatelově prohlížeči a i zde je horší podpora na Unixových operačních systémech. Flash je spíše využíván pro menší doplňky různých webových aplikací, případně pro webové stránky, které využívají grafické efekty, kterých není možné v klasickém XHTML a Javascriptu dosáhnout. (4)

3.8.4. GWT

GWT (5) je nástroj umožňující celý vývoj klientského rozhraní provádět v jayzku Java, který je následně překompilován do XHTML a Javascriptu. Tento postup přináší velké množství výhod jak při vývoji samotném, tak při zmiňované kompilaci.

Při vývoji využíváme všech výhod, které jazyk Java přináší. Objektový návrh aplikace, struktura ve formě balíků oproti přímému vývoji v Javascriptu je zde na neporovnatelně vyšší úrovni. Oproti Javascriptu má také Java výrazně lepší podporu ve vývojových prostředích. Můžeme tak při vývoji využívat kontextovou nápovědu, refaktoring a další nástroje.

GWT navíc odstraňuje většinu rozdílů mezi jednolivými prohlížeči. V problémových částech podpory jednotlivých prohlížečů, kde některé prohlížeče nedodržují standarty, nabízí jednotné rozhraní a interně používá implementaci pro daný prohlížeč, případně i jeho verzi.

Při kompilaci pak provádí několik optimalizací, které zvyšují výkon výsledné aplikace. Výsledkem kompilace není jedna výsledná aplikace, ale jsou vytvořeny verze pro jednotlivé prohlížeče. Při načítání aplikace je pak podle uživatelova prohlížeče a jeho verze vybrána nejvhodnější verze aplikace, která se načte. Navíc výsledná verze neobsahuje žádné zbytečné znaky, jako je odřádkování, odsazení textu a další. Tyto dvě vlastnosti snižují množství dat, které musí uživatel ze serveru stáhnout. Další optimalizace se týká obrázků a dalších zdrojů, které aplikace využívá. V případě obrázků, kterých dnešní webové aplikace obsahují více a více, GWT umožňuje vytvořit rozhraní, do kterého definujeme hlavičky metod uvozené anotací @Resource, které předáme název daného zdroje. Při kompilaci pak k tomuto rozhraní vytvoří implementaci. Ze všech odkazovaných obrázků vytvoří jeden, ke kterému pak aplikace přistupuje. Toto znamená, že uživatel odešle jen jeden požadavek na server a získá tak všechny obrázky, které aplikace využívá, sníží se tak počet volání serveru a tím i jeho zátež.

GWT navíc přináší sadu základních komponent uživatelskéh rozhraní.

3.8.5. ExtGWT

ExtGWT (6) je rozšíření GWT, které přináší velké množství nejrůznějších komponent. Většinou se jedná o komponenty pro tvorby uživatelského rozhraní. Výsledná aplikace svým vzhledem a funkcionalitou velmi připomíná komfort desktopových aplikací. Tyto komponenty jsou také vyvýjené a testované pro běžně používané webové prohlížeče.

Mimo tyto komponenty, přináší tento framework také řadu komponent pro zpracování dat získaných ze serveru.

Tato knihovna je publikována pod licencí, umožňující ji pro Open source projekty použít zdarma, pro komerční projekty je již nutné si zakoupit licenci.

3.8.6. SmartClient (SmartGWT)

SmartClient poskytuje obdobné prostředky jako Ext GWT, oproti Ext GWT je možné ji i pro komerční použití využít zdarma. (7)

3.8.7. Shrnutí

Použití XHTML a Javascriptu by i s ohledem na množství skriptů, které bude aplikace potřebovat, nebylo vhodné. Další dvě technologie, tedy Silverlight a Flash, mají velkou nevýhodu v nutnosti instalovat plugin do uživatelova prohlížeče, navíc u obou technoligií je opět horší podpora na Unixových systémech.

I s ohledem na jeden z funkčních požadavků, a to "API serverové části pro přístupnost dat", se jeví jako ideální použít GWT, které bude přímo použit serverové API pro přístup k datům a bude tato data transformovat do XHTML a zobrazovat uživateli. Použitím GWT snížíme počet nedůležitých požadavků na server (tedy požadavků na obrázky a pod.). Díky přenosu pouhých data ze serveru a transformaci na XHTML až v uživatelově prohlížeči, snížíme velikost přenášených dat.

Mezi dvěma poskytovateli uživatelských komponent jsem zvolil Ext GWT, který poskytuje větší škálu, lépe konfigurovatelných, komponent, na vyšší úrovni je pak také podpora od samotných vývojářů. Navíc existuje velká komunita uživatelů, které tento framework používají.

3.9. DALŠÍ POUŽITÉ TECHNOLOGIE

3.9.1. GlassFish V3 Open Source Edition

Aplikační server GlassFish V3 (8) je referenční implementací Java EE6 (1). S ohledem na vybrané technologie, které budou použity na serverové části, se jeví volba tak zvaného "plného" aplikačního server, který přináší například také podporu pro EJB, jako zbytečná. Jedním z důvodů volby GlassFish byly předchozí zkušenosti se serverem. Navíc s ohledem na serverové zdroje, které bude aplikace využívat, nebude příliš náročné použitý aplikační server změnit.

3.9.2. MySQL

MySQL je v dnešní době považována za již plnohodnotný databázový server, jedná se o jeden z nejrozšířenějších databázových serverů, to hlavně díky své jednoduchosti a nenáročnosti, verzím pro různé platformy a volné šiřitelnosti. Velmi často je dostupná společně s webovou aplikací phpMyAdmin (9) pro správu server. Ta umožňuje jednoduše přes webové rozhraní spravovat celý obsah databázového serveru společně s uživatelskými účty. (10)

KAPITOLA 4
REALIZACE

4.1. NASTAVENÍ

Protože aplikace neužívá příliš mnoho serverových zdrojů, jediné co bylo potřeba na aplikačním serveru nastavit je ConnectionPool a DataSource, tak aby aplikace mohla komunikovat s MySQL databází. V MySQL pak bylo potřeba založit novou databázi a nastavit k ní potřebná oprávnění.

4.2. DATOVÁ VRSTVA- FRAMEWORK NEPTUO-DATA

Framwork neptuo-data přináší vrstvu pro práci s databázovým poskytovatelem, uložiště pro nahrávání souborů a další.

4.2.1. DataStorage, QueryBuilder a AbstractEntity

DataStorage je rozhraní poskytující vrstvu abstrakce nad poskytovatelem databázového uložiště. Jeho implementace pro JPA, tedy třídy JpaDataStorage, pak obaluje použití EntityManageru z JPA frameworku. DataStorage přináší možnost sestavovat dotazy pomocí QueryBuilderu, který přináší vrstvu abstrakce nad tím, jak je dotaz nakonec sestaven, mimo to usnadňuje psaní dotazů díky kontextové nápovědě v IDE a díky tomu, že některé části dotazu sám doplňuje. Pokud vezmeme jeho JPA implementaci, tedy JpaQueryBuilder, a například chceme vybrat z databáze všechny uživatele jejichž křesní jméno odpovída obsahu proměnné firstname. Instance JpaQueryBuilderu, kterou získat z JpaDataStorage, máme v proměnné builder, pak stačí

```
1  builder
2      .select()
3      .from(User.class, "u")
4      .where("username", QueryComparator.EQ, firstname);
```

Další výhodnou vlastností je, že tento dotaz obsahuje pouze dvě řetezcové konstanty, které IDE není schopno zkontrolovat, zda jsou správně, první je alias pro entitu, kterou vybíráme, což ve většině případů je jeden, dva, maximálně několik znaků, ve kterých se člověk málo kdy splete. Druhá je jméno pole, podle kterého vybíráme. Navíc další výskyt aliasu je již doplňován automaticky. Tato třída tedy výrazně snižuje pravděpodobnost překlepu. A i když je možná kód pro dotaz delší než jeho přímé zapsání v podobě řetězce, většina znaků je vygenerována IDE.

AbstractEntity je abstraktní entita, od které může jakákoli další entita, která je identifikována pomocí ID dědit. Kromě metody equals, kterou implementuje porovnáním právě ID, což je ve většinou dostačující pro srovnání entit, obsahuje vlastnost isSystemValue, která při použití společně s DataStorage zaručuje, že pokud je true, pak tento objekt nelze smazat z databáze. Toto je užitečná vlastnost, která nám na nejnižší vrstvě aplikace zaručí, že systémové objekty,

jako může být třeba administrátorský účet, nebo důležité objekty, které jsou v databázi, aby bylo možné je konfigurovat, nicméně, které je důležité, aby v aplikaci byly a nedaly se smazat.

4.2.2. BaseDAO a AbstractDAO

BaseDAO je základní třída pro DAO třídy. Poskytuje metody pro nastavení a získání DataStorage a metodu pro vytvoření instance jiné dao třídy. Toto sice není vhodná vlastnost, protože vytváří vazby mezi jednotlivými dao třídami, nicméně se ukázala užitečnou a také v některých případech téměr až nezbytnou. Je tedy až na samotných vývojářích uživajících tuto metodu, aby ji použili správně a vhodně.

AbstractDAO je pak odvozena z BaseDAO, je parametrická a definuje několik základních metod pro práci s instancemi daného typu.

4.2.3. UploadStorage a UserSession

UploadStorage je rozhraní pro ukládání a získávání souborů uživatelem nahraných do dočasného uložiště. Využívá knihovnu commons-fileupload (11), z níž je i přebrána třída, se kterou pracuje, tedy FilteItem. Přístup k nahraným souborům je omezen, jediný způsob, jak získat záznam z UploadStorage, je poskytnutí metodě get správný klíč. Pokud tedy zvolíme správnou strategii generování klíčů, pak stačí používat jednu instanci implementace rozhraní UploadStorage v prostředí s více uživateli a máme zaručeno, že každý uživatel bude mít přístup pouze k záznamu, ke kterému zná klíč.

UserSession je pak rozhraní obdobné HttpSession z Java EE. Poskytuje metody pro vkládání a získávání objektů, lze ji vyčistit, odstraněním všech položek, případně odstranit z UserSessionContext.

4.3. SERVISNÍ VRSTVA- FRAMEWORK NEPTUO-SERVICE

Tato knihovna umožňuje snadno a rychle vytvářet třídy, které pomocí anotací získají veřejné rozhraní, které je přístupné přes webový prohlížeč. Jádrem knihovny je ServiceDispatcher. Jedná se o servlet, který je potřeba nakonfigurovat ve web.xml, typicky na adresu končící znakem *, znamenající, že další část url adresy je používaná jako adresa volané služby. Slouží jako vstupní bod do servisní vrstvy. Všechny požadavky, které budou zpracovávat servisní třídy jsou směrovány na tento servlet, ten pak pomocí části url požadavku a HTTP hlavičky nazvané *Method* zavolá správnou metodu ve správné službě.

Při prvním načtení ServiceDispatcheru, servlet získá mapu všech servisních tříd a jejich servisních metod. Taková třídy by mohl být realizována implementací nějakého rozhraní, které by mělo definované veřejné metody, ty by pak bylo možné v každé servisní třídě volat. Toto by mělo velkou nevýhodu v tom, že by každá taková třída mohla mít jen omezené, předem definované, množství veřejných metod.

Proto tato servisní vrstva používá refexi, která nám umožňuje daleko větší flexibilitu při implementaci servisních tříd. Za tímto účelem byly vytvořeny dvě anotace, ServiceClass a ServiceMethod.

4.3.1. ServiceClass a ServiceMethod

ServiceClass je použita nad definicí typu, tedy třídy. Tato anotace má jeden parametr, povinný, nazvaný url. Tento parametr definuje URL adresu, na které bude daná servisní třída přístupná.

Druhá anotace, ServiceMethod, je pak používána nad definicí veřejné metody, má 3 parametry, z nichž jsou všechny volitelné. První je name, definující veřejné jméno metody, toto jméno je pak porovnáno s hodnotou HTTP hlavičky *Method*, podle níž je volána správná servisní metoda, pokud však name není zadán, je považován za prázdný, tedy za výchozí metodu servisní třídy. Dalším parametrem je pole, říkající na které HTTP metody má metoda reagovat, výchozí hodnotou je POST, tedy že metoda je volána pouze, pokud uživatel použije HTTP metodu POST. Posledním parametrem je parametr require, který zaručuje, že metoda bude zavolána pokud aktuálně přihlášený uživatel má alespoň jednu ze zadaných rolí. Tento parametr nám umožňuje tedy některé servisní metody zpřístupnit pouze například administrátorům. Z bezpečnostního hlediska je toto velmi důležitý parametr, jak se tento parametr vyhodnocuje více popíši v dalších odstavcích.

4.3.2. MethodParamsProvider

Další velkou výhodu, kterou nám přináší použití reflexe je možnost, aby každá metody měla různé parametry. Tyto parametry nám nastaví buď servisní framework nebo nějaký další poskytovatel parametrů servisních metod. Vězměme například, že budeme chtít, aby naše servisní metoda měla přístup k parametrům HTTP požadavku, například k parametru *productId*, který uživatel odeslal společně s požadavkem. Díky reflexi, není problém aby naše metoda dostala jako jeden ze svých parametrů id produktu, který uživatel požadoval, a to dokonce velmi jednoduše, stačí mezi parametry metody přidat

```
1  @HttpParam("productId") String productId
```

a ServiceDispatcher se postará, aby tato hodnota byla správně předána. O to se starají tak zvaní *Poskytovatelé parametrů*. Jeden je přímo zabudovaný v servisním frameworku. Ten se stará o "základní" parametry spojené s HTTP požadavky, tedy o přístup k servletovému request a response objektu, HTTP parametrům, HTTP hlavičkám, případně serverovým zdrojům a také o načítání uživatelem odeslaných dat z požadavku do Java objektů. O tomto více v dalších odstavcích.

Registrace *Poskytovatele parametrů* je opět prováděna pomocí anotací. Několik málo anotací nám umožňuje jednoduše do aplikace přidat další třídu, která bude poskytovat parametry pro servisní metody. Životní cyklus poskytovatele parametru je spjat s životním cyklem servletu. Při prvním načtení ServiceDispatcheru se vytvoří instance všech poskytovatelů parametrů a tyto jsou znovu používany pro všechny požadavky až do zavolání metody na ServiceDispatcheru ukončující jeho život.

Třída se stává poskytovatelem parametrů, pokud nad svojí definicí obsahuje anotaci @ParamsProvider a definuje veřejnou metodu s anotací @ProviderMethod(ProviderMethodType.PROVIDE). Tato metoda musí mít tři parametry, a to typu Class, odkazující na typ parametru, který je hledán, typu MethodInfo, odkazující na

informace o aktuálně volané metodě, a typu int, udávající index parametru volané metody. Poslední argument je důležitý, pokud chceme získat informace o anotacích definovaných u parametru metody, jako je například anotace @HttpParam. Poslední podmínkou je, že metoda musí vracet typ ParamsProviderResult, který jedna obsahuje návratovou hodnotu, tedy hodnotu poskytnutého parametru a Boolean příznak idikující, zda byl poskytovatelem parameter poskytnut, nebo zda má být zavolán další z poskytovatelů. Třída splňující tyto dvě podmínky se stává poskytovatelem parametrů servisních metod.

Poskytovatel parameterů může mít definované další metody, ty jsou spjaty s životním cyklem. Pokud definuje metody s anotací @ProviderMethod(ProvideMethodType.INIT), pak je tato metoda volána při prvním načtení ServiceDispatcheru. Tato metoda nesmí mít žádné parametry a musí vracet void. Další metoda může být volána před a po každém požadavku, ta musí být uvozena anotací @ProviderMethod(ProvideMethodType.BEFORE), respektive @ProviderMethod(ProvideMethodType.AFTER), obě metody musí vracet void, první tři parametry mají stejné, tedy ServletContext, HttpServletRequest a HttpServletResponse, metoda volaná před požadavkem má navíc další dva parametry, objekt implementující rozhraní Serializer a druhý implementující rozhraní Deserializer. Tyto objekty slouží zápisu do výstupu k uživateli, respektive k náčítání objektů ze vstupu od uživatele. O nich více v dalších odstavcích.

Poskytovatel parametrů může mít ještě jednu důležitou roli, a to, že se může zapojit do vyhodnocování parametru require anotace ServiceMethod, toto lze zařídit vytvořením veřejné metody uvozené anotací @ProviderMethod(ProviderMethodType.AUTHORIZE), tato metoda musí vracet hodnotu typu boolean a mít jeden parametr, a to typu String, označující jméno role. Vrácená hodnota true znamená, že uživatel tuto roli má.

Logika pro autorizaci uživatele pak funguje následovně, pokud máme několik poskytovatelů parametrů s metodou na autorizaci a volaná servisní metoda vyžaduje, aby uživatel měl roli “A” nebo roli “B”, pak stačí, aby jeden z poskytovatelů vrátil true buď pro roli “A” nebo pro roli “B”. Tato logika umožňuje autorizaci uživatele rozdělit mezi několik poskytovatelů parametrů.

4.3.3. Zápis objektů do výstupu a načítání dat ze vstupu

Protože uživatel komunikuje se serverem po protokolu HTTP, je na serveru nutné text požadavku převést na Java objekty a naopak objekty vrácené servisními metodami převést na text. Totomu procesu se říká serializace a deserializace a slouží k tomu dvě definovaná rozhraní.

První je rozhraní Deserializer, které definuje dvě metody, jedna přebírá vstupní proud, ze kterého se bude text načítat, druhá pak přebírá implementaci rozhraní RequestParser, kam přijde vlastní logika pro načtení objektu z textu. RequestParser definuje tři metody, startElement, která je volána pokaždé, když Deserializer narazí na novým element, a přebírá jeho jméno, endElement, volaná nakonci každého elementu, která opět přebírá název elementu. A poslední, content, volanou vždy když Deserializer narazí na tělo elementu.

O to, zda se jedná o XML elementy nebo prvky JSON objektu nebo úplně jiný dataový formát, se již stará implementace rozhraní Deserializer, jejíž instanci vytváří ServiceDispatcher podle typu dat, která uživatel odeslal na server. Tento výběr je prováděn na základě hodnoty HTTP hlavičky pojmenované *Content-Type*, toto je standartní hlavička HTTP protokolu, která se běžně udává pro nastavení typu odesílaných dat.

Třída implementující rozhraní RequestParser pro přihlášení uživatele je vidět v příloze B1. Pomocí implementace RequestParseru není složíté jakkoli komplexní požadavek na server převést na Java objekt. Ukázka z přílohy B1 pro přihlášení uživatele očekává strukturu, která bude obsahovat informace o uživatelském jménu, heslu a doméně, ke které se uživatel příhlašuje. V případě XML formátu dat odeslaných na server by struktura měla vypadat nějak takto

```
1   <login>
2       <username>LOGIN</username>
3       <password>PASSWORD</password>
4       <domain>DOMAIN</domain>
5   </login>
```

Po zpracování tohoto požadavku LoginRequestParserem vrátí metody getUsername, getPassword a getDomain řetezce dodané místo zástupných symbolů *LOGIN*, *PASSWORD* a *DOMAIN*.

Druhé rozhraní, Serializer, se stará o správný zápis objektů do textové podoby. Toto rozhraní poskytuje několik metod, jak zapsat objekt do textové podoby. O vytvoření instance správné implementace rozhraní Serializer se opět stará ServiceDispatcher, který tak činí na základě hodnoty HTTP hlavičky *Accept*, která je opět běžně používána pro nastavení, jaké datové typy klient, který odesílá požadavek, je schopen přijmou a zpracovat.

Příhola B2 zobrazuje metodu, která přebírá kolekci uživatelů, tedy instancí třídy User, a instanci implementace rozhraní Serializer. Metoda vytvoří v serializeru obalový element *users*, zapíše začátek pole, a v cyklu pro každého uživatele z kolekce zapíše element *user* a pak ve formátu *klíč:hodnota* jednotlivé atributy. Jak bude vypadat výsledná struktura dat, již čistě záleží na tom, jak konkrétní implementace rozhraní Serializer zpracuje předaná data.

Tato implementace zaručuje, že servisní třídy nejsou nijak vázány na formát příchozích ani odchozích dat. Dokonce je možné pro data odesílaná na server používat jiný datový formát, než pro data přijímaná od serveru.

V příloze B2 stojí za povšimnutí použití třídy Options, která říká, v tomto případě JsonSerializeru, že má dané elementy zpracovat jiným způsobem, než v normálním případě. Toto vnáší do jinak čistého, tedy na výsledné struktuře nezávislého, návrhu jistý prvek závislosti na konktrétním serilizeru. Právě tento fakt a do jisté míry také fakt, že algoritmus pro načítání a zápis objektů je ve většině případů obdobný, stál za vznikem tříd AutoSerializer a AutoDeserializer. Tyto třídy, jak již jejich název napovídá, slouží k automatickému zápisu a čtení objektu.

4.3.4. AutoSerializer

Třída AutoSerializer poskytuje rozhraní pro nastavení objektu, který se má zapsat a metodu serialize, která zápíše objekt do předáného serializeru. Aby AutoSerializer věděl, jak má objekt zapsat, je nutné, aby třída objektu využívala anotaci Serializable, ta se dá použít jak pro třídu, tak pro metody dané třídy. Má jeden povinný parametr, a to název elementu, do kterého se má objekt, případně návratová hodnota metody, zapsat. Třída AccessType (v příloze B3) ukazuje

jednoduchost použití automatikého zápisu objektu do textu. Kód pro zápis kolekce uživatelů z ukázky B2 byl zobecněn a zjednodušen do třídy AutoSerializer, pro zápis kolekce objektů stačí zavolat tento jeden řádek

```
1  AutoSerializer.factory("accessTypes", accessTypes, serializer).serialize();
```

První parametr je název obalového elementu. Ten není definován nikde v třídě AccessType, protože ta se stará pouze o zápis jedné instance, nikoli celé kolekce, je nutné ho definovat při volání metody serialize, druhý je kolekce objektů typu AccessType, třetí pak instance implementace rozhraní Serilizer. Tímto způsobem je možné zapsat do výstupu celý strom objektů. Toto však může být také nebezpečné, pokud předaný objekt bude obsahovat smyčku, tedy pokud bude například obsahovat objekt, který bude ukazovate zpět na něj, zápis objektů do výstupu se může zacyklit. Je tedy nutné anotaci Serializable používat obezřetně. Následuje úkázka výstupu metody getUsers třídy UserService, která vrací seznam uživatelů, které má právo aktuální uživatel spravovat.

```
1  <neptuo-os generated="1304781662057">
2    <users>
3      <user>
4        <name>admin</name>
5        <displayName>admin: admin admin</displayName>
6        <enabled>true</enabled>
7        <surname>admin</surname>
8        <username>admin</username>
9        <created>2011-01-18 15:31:05.0</created>
10       <id>3</id>
11       <type>user</type>
12     </user>
13     <user>
14       <name>Marek</name>
15       <displayName>mara2: Marek Fisera</displayName>
16       <enabled>true</enabled>
17       <surname>Fisera</surname>
18       <username>mara2</username>
19       <created>2011-05-04 23:03:44.0</created>
20       <id>5</id>
21       <type>user</type>
22     </user>
23   </users>
24 </neptuo-os>
```

Servisní metody však nevolají ani metodu serialize třídy AutoSerializer na objekty, které chtějí vratit uživateli, o návratových typech servisních metod více v dalších odstavcích.

4.3.5. AutoDeserializer

Třída AutoDeserializer se pak na druhé straně stará o načtení objektů ze vstupu od uživatele. Logika této třídy je trochu komplexnější, protože pro efektivní zpracování uživatelova vstupu, umožňuje zadat více objektů, které chceme načíst. Díky tomuto přístupu nám stačí celý vstup od uživatele projít jen jednou. Důvod proč mít potřebu načítat uživatelův vstup do více různých objektů lze dobře vysvětlit na následujícím příkladu. Servisní třída FolderService a její metoda getItemsFromFolder, která má načíst informace o souborech a složkách z nějaké složky. Požadavek na tuto službu ve formátu XML vypadá následovně.

```
1  <neptuo-os>
2    <folders>
3      <folder>
4        <id>PARENT_FOLDER_ID</id>
5      </folder>
6    </folders>
7    <foldersOnly>TRUE/FALSE</foldersOnly>
8    <filterName>FILTER_NAME</filterName>
9    <filterExtension>FILTER_EXTENSION</filterExtension>
10 </neptuo-os>
```

První důležitá poznámka je, že všechny požadavky na serverou část jsou obaleny do elementu *neptuo-os* tak, aby každý požadavek měl jeden kořenový element, což je například u XML nutností. Druhá poznámka je ke struktuře dat, celá serverová část je implementována tak, že je možné provádět dávkové zpracování, tedy například pokud chci vytvořit uživatele, a chci jich vytvořit 100, neznamená to, že musím posílat 100 požadavků na server, nýbrž stačí jeden, který bude mít v sobě 100 struktur obsahující informace o užívateli. U služby FolderService nám tento princip přináší možnost stáhnout informace o více složkách najednou, proto není v požadavku jen jednoduše element *folderId*, ale celá struktura *folders*, *folder* a až v ní *id*.

Takto definovaný požadavek na server nám umožní do kolekce typu FolderItem načíst informace o složkách, jejich *id*, které uživatel požaduje. Dále je pak možné do jednoduchého boolean parametru načíst informaci z elementu *foldersOnly*, udávající, zda se mají vrátit jen složky, nebo složky i soubory. Obdobné je pak zpracování elementů *filterName* a *filterExtension*. Následující ukázka hlavičky metody getItemsFromFolder ukazuje jednoduchost použití

```
1  @ServiceMethod
2  public SerializerResult getItemsFromFolder(
3    User current, BaseDAO dao, Serializer s,
4    @RequestInput("folders") List<FolderItem> items,
5    @RequestInput("foldersOnly") Boolean foldersOnly,
6    @RequestInput("filterName") String filterName,
7    @RequestInput("filterExtension") String filterExtension
8  )
```

První tři parametry jsou standartní objekty poskytované některým z poskytovatelů. Čtvrtý parametr načítá do kolekce typu FolderItem obsah z element *folders*, další tři pak zpracovávají další elementy ze vstupu. Z ukázky je tedy vidět, že načtení uživatelova vstupu do několika objektů se může hodit.

AutoDeserializer poskytuje obdobné rozhraní jako AutoSerializer, také definuje factory metodu, kterou je možné vytvořit instanci a nastavit všechny potřebné parametry. Ta přebírá kromě instance implementace rozhraní Deserializer, vytvořené ServiceDispatcherem, a vstupního proudu z požadavku od uživatele, také pole instancí typu AutoDeserialiterItem, který nese informaci o tom, od jakého elementu a do jaké výsledné struktury se má vstup náčíst. Můžeme tedy načítat kolekce tříd využívající anotaci Deserializable, ke které se dostanu zanedlouho, jednotlivý prvek takové třídy, nebo primitivní typy z určitého elementu, jako int, long, String, Date. String a Date nejsou primitivní typy, ale v kontextu celých objektů typu User, je možné je nazvat primitivními. Ale spíše než načítání přímo primitivních typů, je vhodnější načítat jejich objektové reprezentace, tedy Int, Long, Boolean a poté testovat jejich hodnotu na null, tedy testovat zda patřičný element byl ve vstupu přítomen, nebo se jeho hodnotu nepovedlo převést. Příklad výše zníněné metody totiž nevyžaduje přítomnost elementů *foldersOnly* a dalších. V případě, že je jejich hodnota null, pak se prostě při vykonání metody ignorují.

Jak již jsem zmínil, klíč k načtení objektu ze vstupu je anotace Deserializable, jejíž použití je obdobné jako použití anotace Serializable, má tedy opět jeden povinný parametr udávající jméno elementu, ke kterému se váže. Opět se dá použít jak nad definicí třídy, tak nad definicí metody, tentokrát však očekává tak zvaný *setter*, tedy metodu která vrací void, respektive výstup metody je ignorován, a má jeden parametr.

Ač by na první pohled bylo možné obě anotace, tedy Serializable i Deserializable, sloučit, zůstávají separované, jeden z důvodů je, že dost často jsou prvky tříd, které jsou načítány různé od těch, které jsou zapisovány. Obdobné je to s povinným parametrem udávajícím jméno prvku, ten by opět mohl vycházet z názvu metody, ale takto je hned jasné, jak se bude výsledný element jmenovat.

4.3.6. Zpracování výstupu servisních metod

Nyní se ještě vrátím k tomu, jak se zpracovává návratová hodnota servisních tříd. I přes možnosti AutoSerializeru nestačí pouze vrátit kolekci ze servisní metody, což bylo původním cílem. Je nutné vracet komplexnější objekt, který poskytne dodatečné informace například o obalovacím elementu. Proto servisní třídy pokud chtějí něco vrátit, pak většinou vracejí instance konkrétní implementace rozhraní Result, který definuje jednu veřejnou metodu vracející data v podobě řetězce. Existuje několik implementací toho rozhraní, například CollectionResult, který přebírá kolekci elementů k zápisu a právě jméno obalovacího elemetu, EntityResult, který přebírá jednu instanci třídy, kterou lze zapsat, nebo RawResult, který přebíra pole typů byte, které se pak přímo zapisuje do výstupního poudu, RawResult je vhodný například pro odeslání obsahu obrázku nebo jiného souboru. CollectionResult a EntityResult neimplementují přímo rozhraní Result, ale rozhraní SerializerRequiredResult, které rozšiřuje rozhraní Result o metodu, která přejímá instanci implementace rozhraní Serializer, které pak vyžaduje volání metody serialize třídy AutoSerializer. Takto není nutné, aby servisní třídy přebírali serializer v parametrech a předávali ho pak CollectionResult nebo EntityResult, díky rozhraní SerializerRequiredResult, si o něj požádají ServiceDispatcher “resulty” sami.

V příloze B4 je finální implementace servisní třídy pro příhlášení uživatele. Má dvě veřejné metody, první pro příhlášení uživatele za pomoci uživatelského jména, hesla a domény, druhá zkontroluje, zda je uživatel přihlášen. Její tělo již nemusí nic kontrolvat, pouze zapíše do výstupu informace o aktuálním uživateli, celá kontrola je prováděna již v jednom z poskytovatelů parametrů. Pokud se metoda snaží získat entitu aktuálně přihlášeného uživatele a ten není přihlášen, vyhodí vyjímku HttpUnauthorizedException. Ze třídy je vidět, že velká většina kódu nutného pro zpracování dat od uživatele a zpracování dat směrem k uživateli byla abtrahována a psaní služeb poskytujících data je jednoduché a rychlé.

4.3.7. Zpracování chyb

Poslední důležitou funkcionalitu, kterou ServiceDispatcher zprostředkovává, je zpracování vyjímek. Protože všechny servisní metody jsou volány v rámci transakčního zpracování, pak pokud kterákoli komponenta vyhodí vyjímku, která není zachycena a doputuje až k ServiceDispatcheru, tak ten zavolá rollback na probíhající transakci, který způsobí, že všechny změny provedené voláním metody budou vráceny a všechny objekty nastaveny do původního stavu. Toto zaručuje korektní zpracování i v případě, kdy nastane chyba, ať už díky špatnému vstupu od uživatele nebo chybě některé z komponent. Pokud navíc bude možné výjímku zapsat do výstup, obdobně jako ostatní objekty v aplikaci, server odešle zpět informaci o tom, co způsobilo neúspěšné provedení. Pokud se například uživatel bude snažit pracovat s daty, ke kterým nemá oprávnění, dostane následující chybovou zprávu (za předpokladu, že byl použit XmlDeserializer)

```
1  <neptuo-os generated="1304783587818">
2    <exception>
3      <errorMessage></errorMessage>
4      <className>com.neptuo.service.HttpUnauthorizedException</className>
5      <errorCode>401</errorCode>
6    </exception>
7  </neptuo-os>
```

Dostane tedy ve většině případů dostačující informace o chybě. Protože celá aplikace je implementována tak, aby každá logická chyba měla svoji vlastní vyjímky, vetšinou postačí k indentifikaci kód chyby (element *errorCode*), který je v rámci aplikace jedinečný pro každou vyjímku, případně jméno třídy vyjímky nebo chybová zpráva.

Tímto způsobem by bylo možné zapsat do výstupu jakoukoli vyjímku, protože až na *errorCode* jsou ve výstupu informace, které obsahuje každá vyjímka. Z bezpečnostních důvodů se však zapisují pouze vyjímky definované v rámci aplikace.

Navíc pokud se jedná o vyjímku dědící z HttpException, pak je předán uživateli i patřičný standartní HTTP návratový kód, například pro servisní třídu nebo metodu, která nebyla nalezena, nebo také například pro neautorizovaného uživatele.

4.4. IMPLEMENTACE SERVEROVÉ ČÁSTI

Vlastní implementace serverové části je v projektu neptuo-server. Je rozdělena na jádro, tedy Core, a souborový systém, tedy Fsys. Každá tato část má obdobnou strukturu, skládá se

z kořenového balíku, tedy *com.neptuo.os.core* a *com.neptuo.os.fsys*, ten většinou obsahuje vyjímky používané danou částí, případně nějaké další třídy společné pro celou část. Dále pak balíku data.dao, tedy *com.neptuo.os.core.data.dao* a *com.neptuo.os.fsys.data.dao*, tyto balíky obsahují DAO třídy, pro práci s modelem, ten je v balících data.model. Dále pak balíky service, obsahující servisní třídy. Část Core pak definuje ještě v balíku service.annotation anotaci, kterou využívá poskytovatel parametrů pro předání autorizačního klíče servisní metodě.

4.4.1. Popis modelu

Jádro definuje 7 entit

IdentityBase je základní třída pro User a UserRole. Důvod, proč vznikla tato základní třída je, že do oprávnění ke složkám, souborům, případně dalším objektům, je možné zadávat jak uživatelské role, tak jednotlivé uživatele. Toto vyžaduje, aby uživatelé i role měli společný primární klíč, tedy identifikátor, kterým pak lze v oprávnění určit zda se jedná o uživatele nebo roli.

User je odvozená třída z IdentityBase a definuje základní vlastnosti uživatele. Ty jsou vidět v konceptuálním datovém modelu na Obrázku A.4. Uživatel se může přihlásit pouze pokud jeho účet má příznak enabled nastaven na true.

UserRole je opět odvozená třída z IdentityBase. Definuje své jméno a odkaz na předka. Uživatelské role jsou organizovány do stromové struktury. Pokud pak uživatel má přiřazenu nějakou roli, pak je to stejné, jako by měl přiřazeny všechny "pod role", tedy všechny role, které jsou definovány ve stromové struktuře pod touto rolí.

UserLog je záznam o každém přihlášení uživatele. Je evidován čas přihlášení a odhlášení a autorizační klíč. Tato entita je použita při kontrole autorizačního klíče, podle nějž je pak nalezena entita aktuálního uživatele. Při každém požadavku přihlášeného uživatele na server je pak tento záznam upraven a nastaven čas poslední aktivity uživatele na aktuální čas. Více o kontrole přihlášeného uživatele v části o poskytovatelích parametrů implementovaných v aplikaci.

Permission je základní třída pro oprávnění k nějakému objektu. Definuje typ oprávnění, tedy objekt typu AccessType, dále pak IdentityBase, tedy odkaz na roli nebo uživatele, k němuž se toto oprávnění vztahuje.

AccessType je typ přístupu k objektu, většinou jsou to systémové objekty, mají tedy nastavený příznak isSystemValue na true. Tento typ přístupu pouze definuje svůj název, id a kategorii, ke které patří.

Property je entita pro uložení uživatelských nastavení, to se vždy vztahuje k nějakému uživateli a jménu třídy, které toto nastavení používá, a samozřejmé klíč a hodnotu. Na klientovi se pak tato entita používá pro ukládání například velikosti oken, apod.

Souborový systém pak definuje pět dalších pro práci se soubory a složkami

FileSystemItem je základní třída pro prvek souborového systému, vlastnosti této entity jsou opět vidět na Obrázku A.4. Zajímavé jsou pak atributy publicId a isPublic, které v budoucnu umožní přístup k vybraným prvkům souborového systému z externích instancí aplikací. Toto

umožní pak uživateli ze vzdáleného systému pracovat s těmito soubory, případně si složku nastavit jako svoji jednotku, obdobně jako fungují sdílené složky v operačních systémech.

File je odvozená entita z FileSystemItem, která přidává pouze atribut type, obsahující typ souboru, který reprezentuje. Dále pak definuje nepersistentní vlastnost pro velikost souboru, tato se pak vždy nastavuje na aktuální velikost souboru při načítání z databáze v FolderDAO, případně FileDAO.

Directory je entita pro reprezentaci složky v souborovém systému, je odvozená od FileSystemItem. V tuto chvíli nepřidává žádný další atribut, ale i tak má svůj význam, říká nám, že daný prvek je složka a v níže zmíněné entitě Drive, lze pak navázat pouze pouze složky.

FileSystemPermission dědí ze základní třídy pro oprávnění, navíc definuje vlastnosti pro uchování ke kterému prvku souborového systému se oprávnění váže. Správa souborového systému využívá tři typy přístupu k prvkům. Základní oprávnění, nazvané *Read*, opravňuje uživatele pouze soubor nebo složku vidět ve správci souborů a zobrazit jeho obsah. Další je oprávnění, které navíc přidává uživateli právo zápisu do souboru nebo složky, nazvané *ReadWrite*. U složek toto oprávnění umožňuje uživateli vytvářet v dané složce podsložky nebo nahrávat soubory. U souboru má uživatel možnost zapisovat obsah, nebo ho případně nahradit nově nahraným souborem. Oprávnění *Manage* pak umožňuje uživateli prvek přejmenovat, smazat nebo nastavovat oprávnění přístupu k němu. Toto oprávnění automaticky dostane uživatel při vytvoření nové složky nebo nahrání nového souboru.

Drive je pak entita reprezentující jednotku v systému. Dělí se na dva druhy, jednak systémové jednotky a jednak uživatelské jednotky. Ta je navázána na nějakou složku v systému, pokud má nastavenu physicalPath na nějakou cestu, pak to znamená, že tato jednotka je již přímo navázána na fyzickou cestu v souborovém systému. Tuto cestu mají nastavenou pouze systémové jednotky, složka na kterou je navázána má pak svého rodiče, vlastnost parent, nastavenou na null. Uživatelská jednotka na druhé straně je pak navázána na libovolnou složku, je přiřazena ke konkrétnímu uživateli a je pouze jakousi zkratkou v souborovém systému.

4.4.2. Použití neptuo-data

Téměr celá logika serverové části je uložena v DAO třídách, tedy ve třídách, které jinak nejsou navázané na webovou vrstvu. Jsou tedy velmi snadno a rychle znovupoužitelné. Tímto jsme získali v aplikaci vrstvu, která se dá znovu použít. Všechny DAO třídy dědí z AbstractDAO nebo z BaseDAO, tedy poskytují metody pro práci s jednotlivými entitami.

Jádro obsahuje DAO třídy pro správu uživatelských účtů a rolí. Dále pak pro entitu AccessType, která je používána v přístupových právech, kde definuje akce, které dané oprávnění umožňuje. Další DAO je pro entitu Property.

Část spravující souborový systém pak poskytuje DAO třídy pro složky, soubory a jednotky. Společným předkem DAO třídy pro soubory a složky je třída FsysItemDAO, která definuje několik společných metod pro práci s těmito dvěmi entitami. Dále definuje několik hlaviček metod, které je nutné v potomcích přepsat, mezi ně patří například metody, které se starají o přesouvání a kopírování souborů a složek.

FolderDAO umožňuje kromě získání obsahu složky, složky vytvářet, pokud vytvářená složka nemá zadána žádná oprávnění, pak se převádějí oprávnění z nadřazené složky. Přitom také

provádí kontrolu, zda má aktuální uživatel možnost zápisu do nadřazené složky. Dále pak umožňuje smazat složku, včetně rekurzivního mazání všech svých podadresářů a souborů, přejmenovat, přidat nebo odebrat oprávnění. Ve všech těchto případech je nutné mít k dané složce oprávnění *Manage*. Dále umožňuje zapsat celou složku do dočasného zip souboru a ten vrátit.

FileDAO poskytuje obdobné rozhraní jako FolderDAO, přídává možnost nahrání souboru za použití výše zmíněného objektu typu FileItem z commons-fileupload.

Při vytváření nového prvku pomocí FileDAO a FolderDAO jsou oprávnění k nové složce nastavena podle rodičovské složky. Pokud v těchto zkopírovaných oprávnění není oprávnění *Manage* pro aktuálního uživatele, toto oprávnění je do nich přidáno.

DriveDAO pak poskytuje rozhraní pro vytváření a mazání uživatelských jednotek. K systémovým jednotkám má uživatel přístup pouze, pokud má na cílové složce právo *Read*.

4.4.3. Použití neptuo-service

Výsledná aplikace obsahuje 8 servisních tříd v části Core, které umožňují příhlášení a odhlášení uživatele, pracovat s uživatelskými účty a rolemi. Je zde služba, reprezentovaná třídou IdentityLookupService, pro vyhledání uživatele nebo role podle části jména. Pak jsou zde pomocné služby pro získání přístupových typů, tedy AccessTypeService, a pro uživatelské nastavení, tedy PropertyService. Dále je zde služba pro odeslání feedbacku, která umožňuje odeslat email s feedbackem.

V druhé části aplikace jsou dvě hlavní služby pro práci s prvky souborového systému, tedy FileService a FolderService. Protože nahrání souboru na server není možné pomocí technologie AJAX, ale pouze klasickým odesláním formuláře, není tedy možné nastavit potřebnou HTTP hlavičku *Method* pro zavolání patřičné metody na servisní třídě. A protože v celé servisní vrstvě byla použita konvence, kdy výchozí metoda, tedy s nevyplněným jménem, servisní třídy jejíž URL končí množným číslem, například */fsys/folders* nebo */fsys/files*, vrací seznam záznamů, byla vytvořit vlastní službu pro nahrávání souborů. Tato služba nevrací data ve formátu podle toho, který klient nastavuje v hlavičce *Accept*, ale vždy vrací typ *text/plain* obsahující pouze klíč k nahranému souboru. Toto bylo vynuceno opět možnostmi internetových prohlížečů, kdy jediný způsob jak odeslat formulář klasickým způsobem, ale bez znovu načtení celé stránky, je nastavit mu atribut target na jméno vloženého rámce ve stránce. Bohužel pak vrácený obsah je zpracován jako klasický HTML dokument a ostatní formáty dat, jako například XML, nebylo možné korektně zpracovat.

Další dvě služby, tedy FileContentService a ZipFolderService, opět nevrací format dat podle hlavičky *Accept*, ale přímo binárně zapsaný obsah požadovaného souboru, respektive složky. Tyto dvě služby je možné vyvolat přímo z adresního řádku prohlížeče a pokud zadáme správné *publicId* k souboru, respektive složce, vrátí nám požadovaný obsah, takto je tedy možné i externě odkazovat na soubory v aplikaci.

Aplikace pak definuje dva poskytovatele parametrů, CoreParamsProvider poskytuje přístup servisním třídám k DAO objektům z jádra. Udržuje instanci UploadStorage používané pro uchování uživateli nahraných souborů. A poskytuje přístup k entitě aktuálně přihlášeného uživatele. Po přihlášení uživatele, kdy je vytvořen záznam v UserLogu a je k němu přiřazen unikátní *AuthToken*, řetězec znaků používaný k autentizaci uživatele. Tento řetězec pak musí

uživatel vždy připojit ke svému požadavku na server formou HTTP hlavičky, pojmenované *AuthToken*. Také definuje autorizační metodu pro ověření oprávnění uživatele volat danou metodu. Druhý poskytovatel pak umožňuje přístup k DAO objektům pro správu souborového systému.

4.5. KLIENTSKÁ ČÁST

4.5.1. Obecně

Klientská část poskytuje webové rozhraní pro práci se serverovou částí. Je rozdělena do několika vrstev, nejspodnější vrstva definuje třídy pro komunikaci se serverovou částí aplikace, nad ní jsou pak komponenty, nejčastěji dědící ze třídy Window, které vytváří uživatelské rozhraní a zpracovávají události od uživatele.

4.5.2. Struktura

Uživatelské rozhraní aplikace se skládá z pracovní plochy a dvou lišt. Horní lišta, nazvaná hlavní menu a reprezentovaná třídou MainMenu, slouží jako menu aplikace, spodní lišta, nazvaná taskbar a reprezentovaná třídou Taskbar, zobrazuje spuštěné aplikace. Pracovní plocha, reprezentovaná třídou Desktop, pak zobrazuje jednotlivá okna spuštěných aplikací. Jednotlivé agendy klientské části aplikace, jako je například Správce uživatelů nebo Prohlížeč, jsou nazývány aplikace. Každá taková aplikace má hlavní třídu, která implementuje rozhraní Application. Tato třída je vstupním bodem do aplikace a slouží pro spuštění aplikace, například z hlavního menu. Této třídě pak možné předat argumenty pří spouštění aplikace.

Třída implementující rozhraní Application pak většinou zobrazuje hlavní okno dané aplikace. Toto okno je reprezentováno abstraktní třídou Window, která zaručuje zaregistrování okna do WindowManageru, který se stará o přepínání mezi jednotlivými okny pracovní plochy, o zaregistrování prvku do taskbaru a dalši. Dále pak pokud není nastaveno jinak v potomkovy, třída Window automaticky načte ze serveru vlastnosti daného okna, jako je jeho výška a šírka. Při změně velikosti uživatelem, jsou pak nové hodnoty uloženy zpět na serverové části, každý uživatel si může takto přizpůsobit každé okno aplikace. Množina automaticky načítaných vlastností se dá rozšírit překrytím metody addPropertiesToLoad v odvozené třídě. Překrytím metody onPropertiesLoaded se pak dají tyto načtené vlastnosti zpracovat. Vlastnosti jsou načítány v konstruktoru třídy Window a okno není vykresleno na plochu dokud nejsou všechny vlastnosti aplikovány, uživateli se tedy zobrazí okno až když je plně inicializované. Všechny vlastnosti okna jsou načítány pouze při prvním vytvoření okna, dále jsou pak již využívány lokálně uložené hodnoty, které jsou aktualizovány při jejich změně.

4.5.3. Lokalizace a ostatní zdroje aplikace

Celá aplikace je přeložena do českého a anglického jazyka. Pro toto bylo použito standartní rozhraní frameworku GWT pro tvorbu lokalizovaných aplikací. Společné texty používané celou aplikací jsou ve třídě WebdeskConstants, každá aplikace pak se definuje svůj vlastní balík lokalizovaných textů, klíč jednotlivých textů je nejčastěji odvozen od názvu komponenty, která ho využívá. Pro jednoduchost použití jsou pak ve třídě Contants vytvořeny statické instance všech jazykových balíků. Většina aplikace také využívá ikony z balíku Icons16, jehož instance je také v Constants. Tento přístup zajistí, že při kompilaci je ze všech obrázků vytvořen jeden velký, který uživatel při načtení aplikace stáhne ze serveru, a celá aplikace ho pak již jen využívá. Tento

přístup bohužel nebylo možné využít úplně na všech místech aplikace, protože rozhraní ImageBundle, které Icons16 implementuje, nepodporuje vyhledání obrázku podle klíče.

4.5.4. Komunikace se serverovou částí

Třídy pro komunikaci se serverovou částí mají příponu „Service", jejich účelem je vzít předaná data, převést je do formy, kterou je možné poslat po HTTP protokolu, odeslat na server a odpověď od serveru opět převést na data, se kterými bude možné v aplikaci dále pracovat. Protože komunikace po HTTP protokolu je prováděna pomocí technologie AJAX, tedy asynchroního volání serveru, není možné data vrácená od serveru přímo vrátit jako návratovou hodnotu metody v servisní třídě, ale je nutné metodě předat takzvaný „callback", tedy objekt implementující nějaké rozhraní, kterému budou data předána. Tento způsob umožňuje dále s uživatelským rozhraním pracovat, zatímco se čeká na zpracování požadavku serverem. Následuje ukázka typické hlavičky metody v servisní třídě

```
1  public static void getFolderItems(
2      int parentId, boolean foldersOnly, String nameFilter, String extension,
3      AsyncCallback<FsysItem> callback)
```

Poslední argument je zmiňovaný objekt, implementující rozhraní AsyncCallback, které definuje metody pro zpracování dat, a metody pro zpracování případné chyby při vykonávání požadavku.

Klíčová třída používaná v servisních metodách se jmenuje ServiceRequestBuilder. Ta umožňuje předat data, URL serverové servisní třídy a název metody, která se má zavolat, callback a informace pro DataEventBus (o tomto konceptu později). ServiceRequestBuilder je třída, která slouží pro nastavení parametrů požadavku na server, sama neobsahuje příliš logiky, ale usnadňuje vytváření požadavků na server a zabaluje části kódu, který se opakuje v každém nebo téměř každém požadavku. Definuje několik "setter" metod, tedy metod, která nastavují jednolivé parametry. Protože vytvoření instance této třídy slouží pouze k jednorázovému nastavení parametrů požadavku a jeho odeslání, každá metoda navíc místo typu void, který vrací běžné setter metody, vrací danou instanci ServiceRequestBuilderu, je tedy možné vytvoření požadavku zapsat do jednoho řádku, jak je vidět na následující ukázce

```
1  ServiceRequestBuilder.factory(FsysItem.class)
2      .setCallback(callback)
3      .setDataEvent(DataEvents.FsysItems, DataEventType.UPDATE)
4      .setData(model)
5      .setMethod(ServiceMethods.FileSystem.Folders.Rename)
6      .toRequestSent();
```

Poslední volaná metoda je toRequestSent, která celý požadavek odešle na server. Pro zjednodušení nastavování cílových serverových metod a serverových servisních tříd, byla vytvořena třída ServiceMethods, která definuje pro každou serverovou servisní třídu jednu statickou třídu, v předchozím příkladu je to třída Folders, navíc jsou všechny třídy ještě rozděleny podle modulů na serverové části, tedy Core a FileSystem. Dále pak pro každou metodu servisní třídy je jedna statická instance třídy ServiceMethod. Protože navíc třída ServiceMethod

přebírá i instance ServiceUrl, stačí v ServiceRequestBuilderu nastavit pouze metodu a o jakou serverovou servisní třídu se jedná se již doplní právě z předané ServiceMethod.

Obdobně jako na serverové části, i na té klienstké je nutné převést data aplikace do textové podoby tak, aby bylo možné je po HTTP protokolu poslat na server. Bohužel i přes velké možnosti GWT, je zde absence reflexe pomocí které by bylo možné objekty zapsat do textové podoby. Zřejmě i proto přinest framework ExtGWT třídu ModelType, která obsahuje definici, jak se má objekt zapsat a zpět načíst z textové podoby. Aby tento princip mohl fungovat, je nutností aby objekt, který chceme zapsat nebo načíst, měl všechny své vlastnosti uložené ve speciální mapě, která je definována ve třídě BaseModelData. Celé to pak funguje tak, že třída dědící z BaseModelData ukládá své proměnné do této mapy, v ModelType je pak definováno, které prvky této mapy se mají do výstupu zapsat a společně s názvem elementu pro jednu instance dané třídy a názvem obalujícího elementu. V aplikaci jsem pak toto spojil v jedno, vytvořit vlastní třídu BaseModelData, od které dědí všechny třídy jejichž instance jsou posílány na server. Moje třída BaseModelData definuje proměnnou modelType typu ModelType, a každá odvozená třída do této proměnné nastavuje, jak má být zapsána a načtena do respektive z textu. Protože v některých případech modelType, definovaný v definici třídy, nemusí přesně vyhovovat tomu, co v dané požadavku potřebujeme odeslat, třída ServiceRequestBuilder definuje setter pro nastavení modelType, který se má použít, toto chování je například využité při odesílání požadavku na smazání uživatelského účtu, kde je zbytečné na server odesílat celou entitu User, když serveru stačí pro smazání uživatele pouze jeho id.

Protože od serveru přijmeme opět pouze textovou odpověď, je ve většině případů nutné tento text převést zpět na požadované objekty, které je možné dále používat v aplikaci. Pro předání dat zpět aplikaci slouží dvě rozhraní, RequestCallback a AsyncCallback, obě definují tři metody, které musí být implementovány, metodu onSuccess, která je volána v případě, že při odeslání, serverovém zpracování a přijetí odpovědi nesnastavala žádná chyba. Metodu onClientError, která je volána, pokud při odesílání požadavku nastane nějaká chyba a metodu onServerError, která je volána, pokud nastane nějaká chyba při serverovém zpracování odesleného požadavku. Tedy například pokud se uživatel snaží smazat složku, ke které nemá patřičná oprávnění. Jediné, v čem se tyto dvě rozhraní liší, jsou parametry předávané metodě onSuccess. U rozhraní RequestCallback jsou předány objekty typu Request a Response, které obsahují informace o HTTP požadavku a odpovědi serveru, včetně textové podoby odpovědi serveru. Rozhraní AsyncCallback naopak obdrží seznam objektů, navíc pokud je zadán generický typ, pak jsou tyto objekty přetypovány na tento typ. Pokud je ServiceRequestBuilderu předán objekt implementující RequestCallback, pak ServiceRequestBuilder funguje pouze jako zpostředkovatel. Pokud je však předána instance implementující AsyncCallback, při přijetí odpovědi od serveru je zavolána metoda parse třídy BaseService, která provede převedení textu získaného od serveru na objekty, které jsou pak předány metodě onSuccess.

Protože API frameworku GWT, které zprostředkovává vlastní AJAX zavolání serveru nepodporuje rozdělení na chyby způsobené na straně klienta, tedy například výpadekem připojení k serveru, a na chyby způsobené při zpracování požadavku na straně serveru, je ještě pod třídou ServiceRequestBuilder třída RequestBuilder, která rozlišuje klientské chyby a chyby serveru, a podle druhu chyby volá metodu onClientError, které předává vyjímku, která chybu způsobila, nebo metodu onServerError, které předává objekt obsahující informace o serverové chybě.

Poslední část ServiceRequestBuilderu umožňuje nastavit informace o datevé události, která je vyvolána při úspěšném požadavku.

Třída DataEventBus, jak její název napovídá, realizuje sběrnici datových událostí. Pokud s jedním typem dat, například s obsahem složky, pracujeme na několika různých místech aplikace, například obsah domovského adresáře, který je zobrazen na ploše, a obsah téhož adresáře máme otevřený v okně aplikace Prohlížeč, pak je nutné, aby obsah byl v obou zobrazení stejný. Tuto synchornizaci zaručuje DataEventBus. DataEventBus umožňuje zaregistrovat posluchače na učitý typ datové události. Daný posluchač je pak vždy notifikován, pokud aplikace přijme data takového typu. Druhy těchto událostí jsou definovány v třídě DataEvents. Producent události pak ještě předává, jestli se jednalo o přidání nových dat, aktualizaci již exitujících dat, odebrání dat a nebo o nahrazení stávajících dat novými. Navíc sběrnici může ještě předat libovolný objekt, jako parametr události, ten je například použit při načítání nového obsahu složky, kdy je sběrnici předán objekt typu ExplorerFilterParams, který uchovává stav aktuálního stav filtru. V tomto případě je tento objekt klíčový, protože pokud máme otevřenou jednu složku ve dvou oknem aplikace Prohlížeč, avšak v jednom okně některé položky odfiltrujeme, pak je důležité, aby se v druhém okně aktualizovaly pouze položky, které vyhovují nastavení tohoto filtru, a ostatní zůstaly beze změny. Aby se pak aplikace jako je Prohlížeč nebo správce uživatelů nemusely starat o zaregistrování do DataEventBus, byla vytvořena třída ListStore, která rozšiřuje třídu ListStore z frameworku ExtGWT o automatické zaregistrování na předanou událost. Třída ListStore je používána v celém frameworku ExtGWT jako uložistě dat pro různé komponenty uživatelského rozhraní. Pokud chceme vyvolat datovou událost, což většinou děla třída ServiceRequestBuilder, zavoláme na DataEventBus metodu fireEvent, které předáme typ datové události, o jakou změnu dat se jedná a nová data. DataEventBus se pak postará o zavolání vševh posluchačů registrovaných na danou datovou událost. Tento princip umožňuje synchronizovat dat používaná celou aplikací.

4.5.5. Aplikace Explorer (Prohlížeč)

Aplikace pro procházení a práci se souborovým systémem. Narozdíl od ostatních aplikací, kde je uživatelské rozhraní definované přímo v hlavním okně, jádro prohlážeče je vytvořené jako panel, který lze pak dle potřeby vložit do různých oken.

Veřejné rozhraní, které panel poskytuje je prezentováno rozhraním ExplorerPanel. To definuje veřejné metody, které jsou panelem poskytovány. O vytvoření instance se stará ExplorerPanelFactory. Vlastní implementace panelu je rozdělena do několika tříd, tou základní je ExplorerPanelBase, která dědí od LayoutContainer, což je obecný předek všech panelů, tedy komponent, které mohou obsahovat další prvky, ve frameworku ExtGWT. ExplorerPanelBase definuje základní metody a proměnné používáné dalšími třídami, včetně definice posluchačů na uživatelské události v panelu. Od ExplorerPanelBase dedí abstraktní třída ExplorerPanelController, která definuje prvky uživatelského rozhraní panelu. To se skládá ze čtyř panelů. První je NavigateBar, který obsahuje komponenty pro navigaci. Druhý pak ActionsBar, který definuje komponenty pro uživatelské akce s vybraným záznamem, případně pro připdání nových zazánamů. Dále také komponenty pro filtrování obsahu zobrazené složky. Hlavní panel je komponenta MainContentPanel, která se skládá ze dvou Grid komponent, jedna zobrazuje jednotky dostupné uživateli a druhá obsah aktuální složky. Poslední panel, který je v některých oknech, využívajících ExplorerPanel skryt, je DetailsBar, který zobrazuje informace o vybraném záznamu. Dále ExplorerPanelController definuje abtraktní metody, které jsou volány

při jednotlivých akcích uživatele. Třída ExplorerPanelImpl implementuje rozhraní ExplorerPanel a dědí od ExplorerPanelController. Tato třída realizuje logiku uživatelského rozhraní, stará se o načítání dat, navigaci mezi jednotlivými složkami, zobrazování dialogů pro vytvoření složky, upload souboru a další. Poslední třídou přímo spjatou s panelem je ExlporerPanelHelper, který obsahuje některé pomocné metody pro vytvoření uživatelského rozhraní a jeho nastavení.

ExplorerPanel pro uchovávání seznam položek zobrazené složky nevyužívá standartní ListStore, ale používá vlastní třídu nazvanou FileSystemListStore. Protože logika, zda přijaté objekty od DataEventBus patří do aktuální složky nebo ne, je složitější než v ostatních případech, kdy se ListStore rozhoduje zda přijaté objekty zařadit do seznamu nebo ne, jen podle hodnoty volitelného parametru a typu události. U FileSystemListStore ještě záleží, zda je zobrazovaná složka načtena pomocí adresy složky, nebo podle id. Další roli také hraje stav filtru v daném okně.

Hlavní okno aplikace Explorer slouží pro prohlížení obsahu, pro výběr souboru slouží okno FileSelectWindow, které přidává pod panel pro prohlížení ještě tlačítka pro potvrzení vybraného souboru a pro zrušení výběru a zavření okna. Pro výběr složky pak slouží okno obdobné oknu FileSelectWindow nazvané FolderSelectWindow. Tyto a některá další okna lze zobrazit jako dialogy, tedy okna která jsou vždy na popředí dokud nejsou zavřena, jednoduše pomocí třídy Dialogs, která definuje statické metody pro jejich vyvolání.

4.5.6. Ostatní aplikace

DriveManager, nebo-li Správce jednotek, je aplikace pro správu uživatelských a systémových jednotek. Uživatel zde vidí své uživatelské jednotky a systémové jednotky ke kterým má oprávnění. U systémových jednotek, stejně tak jako u své domovské, *Home* jednotky, má uživatel pouze právo měnit oprávnění k její kořenové složce, a to jen pokud k dané složce má oprávnění *Manage*. Ke své domovské jednotce má uživatel vždy právo *Manage*, u systémových jednotek mají toto právo zpravidla uživatelé role *Admins*. Uživatel si zde může definovat vlastní jednotku a navázit ji na nějaký adresár v aplikaci. Tato jednotka slouží jako jaká si zkratka pro otevření dané složky v Exploreru. Tato aplikace, stejně jako UserManager zmíněný níže, používá standartní ListStore a je registrovaná do DataEventBus.

UserManager, nebo-li Správce uživatelských účtů, obsahuje rozhraní pro vytváření a správu uživatelských účtů a rolí. Uživatel zde vidí pouze účty, které mají přiřazenou jednu z rolí uživatele nebo, kteří mají přiřazenou jednu z rolí, jejíž nepřímým předkem, je jedna z rolí uživatele. Obdobná logika je použita i u rolí, které uživatel vidí v editaci.

Textový editor, tedy TextEditor, je aplikace pro editaci textových souborů a to buď ve formě čistě textového souboru nebo ve formě HTML souboru, který je pak zobrazován jako formátovaný text, tedy obdobně jako text ve Windows aplikaci Word. Toto jsou také prozatím jediné dva druhy souborů, které je možné otevřít přímo z aplikace Explorer a není uživatel nucen je stáhnout na lokální disk a po úpravě zpět nahrát na server.

4.6. SHRNUTÍ VÝVOJE APLIKACE

4.6.1. Serverová část

Fakt, že jsem v aplikaci nepoužíval žádný aplikační framework, přinesl hodně práce navíc. Serverová část aplikace obsahuje 156 tříd, z toho je 20 v projektu neptuo-data a 57 v projektu

neptuo-service. Použitím nějakého vhodného frameworku jsem mohl ušetřit polovinu zdrojových kódů.

Ovšem výsledné rozhraní knihovny neptuo-service je velmi flexibilní. Umožňuje komunikovat s různými datovými formáty. Vytvoření servisní třídy je velmi jednoduché a umožňuje se výhradně zaměřit na vlastní aplikační logiku. Kódu, který je potřeba napsat pro zpracování vstupu a výstupu, je již opravdu minimum.

4.6.2. Klientská část

Framework GWT vývoj klientské části aplikace velmi ulehčil, také díky svému „Hosted mode“, který při vývoji aplikace umožňuje používat přímo bytecode zkompilované aplikace. Pomocí pluginu v prohlížeči pak umožňuje generovat výslednou Javascriptovou aplikaci po částech. Toto umožňuje při změně zdrojových kódů jen znovu načíst stránku s aplikací a změny jsou aplikovány na bytecode používaný pluginem prohlížeče a bez nutnosti znovu kompilace celé aplikace jsou změny aplikovány na výslednou aplikaci.

Framework ExtGWT přenesl všechny potřebné komponenty uživatelského rozhraní. A tak jsem se mohl hlavně zaměřit na sestavení výsledného rozhraní.

Jedinou nevýhodou použitého GWT je, že existuje několik situací, kdy kód spuštěný v “Hosted mode” funguje, ale po překompilování do výsledného Javascriptu již ne. Naštěstí všechny tyto situace, na které jsem narazil, jsou dobře zdokumentované a i odůvodněné proč se tak děje.

KAPITOLA 5
TESTOVÁNÍ

Testování bylo rozděleno na testování klientské a serverové části. Testování klientské části, protože bylo většinou prováděno formou manuálního testování, pak v sobě zahrnovalo integritní testování spolupráce jednotlivých částí aplikace.

5.1. SERVEROVÉ TESTOVÁNÍ

Testování serverové části aplikace bylo prováděno v krátkých cyklech. Při přidání nové metody do DAO vrstvy a servisní vrstvy byla vždy otestována korektní funkcionalita, až teprve poté bylo v klientské části aplikace vytvořeno patřičné rozhraní a testována vzájemná komunikace.

5.1.1. Testování doménového modelu

Protože o většinu logiky ukládání a získávání dat z databáze se stará framework JPA, u doménového modelu se tak jednalo pouze o testování přítomnosti a korektnosti dat po komitu transakce vyvolané servisní vrstvou. K tomuto ověřování byl nejčastěji použit nástroj phpMyAdmin, který umožňuje zobrazovat obsah databáze.

5.1.2. Testování DAO vrstvy

Testování DAO vrstvy bylo spojeno s první fází testování servisní vrstvy, kdy byly generovány požadavky na server tak, aby program prošel všechny možné cesty v dané DAO třídě. V této fázi testování bylo nejčastěji využíváno krokování programu, které umožňovalo projít celé zpracování požadavku.

5.1.3. Testování servisní vrstvy

Testování servisní vrstvy bylo prováděno ve dvou krocích. Nejdříve pomocí testovacího rozhraní, pomocí souboru test-service.html z kořenového adresáře aplikace. Toto rozhraní umožňuje odeslat na servisní vrstvu HTTP požadavek. Navíc je zde možné manuálně nastavit obsah požadavku. Díky tomu je možné otestovat jak chování v případě korektních vstupů od uživatele, tak i otestovat zpracování nesprávných vstupů. Testování bylo prováděna v prohlížeči Firefox a pro kontrolu výsledků byl použit plugin Firebug (11), který umožňuje analyvat odeslané požadavky.

Ve druhém kroku se pak již jednalo o testování integrace s klientským rozhraním, kdy byly požadavky na servisní vrstvu generovány již přímo z klienského rozhraní. V této fázi již struktura požadavků odeslaných na server byla korektně vygenerována klientem, a testovala se tak pouze správná reakce na validní a nevalidní hodnoty zadané uživatelem, které nezachytila klientská validace vstupů.

5.2. KLIENTSKÉ TESTOVÁNÍ

Ačkoli se jedná o webovou aplikaci, server neodesílá klientovi téměř žádné HTML a veškerý obsah, který se uživateli zobrazuje je generován dynamicky pomocí javascriptu. Žádný z faktorů,

které jsou pro klasické webové aplikace důležité, jako validita HTML a optimalizace pro vyhledávače, zde není tak důležitý nebo není téměř vůbec důležitý. S ohledem na to, tedy záleží pouze na korektním zobrazení v jednotlivých prohlížečích. Toto máme z velké většiny zaručené díky použitým technologiím, tedy GWT, které odstraňuje většinu rozdílů mezi jednotlivými prohlížeči. Dále také díky knihovně ExtGWT, která je již vyvíjena a testována pro různé prohlížeče. Aplikaci byla testována pro prohlížeče Google Chrome, Firefox a Interner Explorer, tedy 3 nejčastěji používané prohlížeče.

Většina testování klienské části byla prováděna formou manuálního testování. Zde byl opět často využíván plugin Firebug pro prohlížeč Firefox, který umožňoval kontrolu odesílaných dat na server.

5.3. ALFA TESTOVÁNÍ

Alfa testování probíhalo na lokálním počítači. Aplikace byla vystavena na GlassFish server na počítači s operačním systémem Windows 7. Na aplikaci se již pohlíženo jako na celek. Testování simulovano reálné použití aplikace.

5.4. BETA TESTOVÁNÍ

Pro účely beta testování je aplikace vystavena na adrese http://os.neptuo.com/webdesk/. Do aplikaci je možné se přihlásit pomocí následujících účtů:

- Uživatelské jméno: **admin**, heslo: **admin**
  Jedná se o vestavěný administrátorský účet, má přiřazenou role admins. Tento účet tedy může spravovat veškerý obsah, s vyjímkou domovských adresářů jednotlivých uživatelů.

- Uživatelské jméno: **tester**, heslo: **tester**
  Jedná se o uživatele s přiřazenou rolí users. Tato role má oprávnění *ReadWrite* do systémové jednotky *System://*.

KAPITOLA 6
ZÁVĚR

Hlavním cílem práce bylo vytvořit systém pro správu souborového systému. Výsledná aplikace umožňuje všechny běžné operace se soubory a složkami známé z operačních systémů.

Před počáteční analýzou bylo důležité nahlédnout do již existujících řešení a získat tak co nejvíce informací jak by aplikace měla fungovat. Řešení volně dostupná na internetu poskytnula dostatečné informace a inspiraci pro vlastní řešení.

Dalším úkolem bylo vybrat vhodný programovací jazyk a prostředí pro aplikaci. Prozkoumal jsem programovací jazyky pro webové aplikace od těch nejskromnějších až po jazyky jako je Java a C#. Zde při výběru hrály roli také zkušenosti z předchozích projektů.

Po výběru programovacího jazyka přisla volba na vhodné frameworky. Pro klientskou část volba GWT vývoj velmi usnadnila a zrychlila. Výsledná aplikace je optimalizovány do takové míry, které by s jiným nástrojem šlo jen velmi těžko dosáhnout. Na serverové straně volba vlastní implementace rozhraní pro komunikaci naopak množství práce přidělala. Nakonec však knihovna neptuo-service přinesla rozhraní, které lze jednoduše použít a šetří řádky kódu při psaní jednotlivých servisních tříd.

Testování pak odhalilo řadu chyb, počínaje chybami v neptuo-service, zejména při serializaci a deserializaci objektů, přes chyby v aplikační logice vlastní aplikace, tedy projektu neptuo-server, po chyby v uživatelském rozhraní, kde se jednalo také o několik chyb při zobrazení v jednotlivých prohlížečích.

Celá práce pak splnila očekávání. Přináší bohaté uživatelské rozhraní, které správně funguje v nejpoužívanějších prohlížečích. Serverová část pak přináší široké API, toto API navíc umí komunikovat jak pomocí XML, tak pomocí JSON. Díky struktuře a vhodné implementaci navíc pro další formáty dat lze dodatečně přidat podporu.

KAPITOLA 7
LITERATURA

1. **Oracle.** *Java EE.* [Online] http://download.oracle.com/javaee/6/tutorial/doc/docinfo.html.

2. **Microsoft.** *ASP.NET.* http://www.asp.net/.

3. **Microsoft**. *Silverlight.* http://www.silverlight.net/.

4. **Adobe.** *Flash.* http://en.wikipedia.org/wiki/Adobe_Flash.

5. **Google.** *Google Web Toolkit.* http://code.google.com/webtoolkit/.

6. **Sencha.** *Ext GWT.* [Online] http://www.sencha.com/products/extgwt/.

7. **Isomorphic Software.** *SmartGWT.* [Online] http://www.smartclient.com/.

8. **Oracle.** *GlassFish.* [Online] http://glassfish.java.net/.

9. **phpMyAdmin dev team.** *phpMyAdmin.* [Online] http://www.phpmyadmin.net/.

10. **Oracle.** *MySQL.* http://www.mysql.com/.

11. **Mozilla.** *Firebug.* http://getfirebug.com/.

12. **Apache.** *Commons FileUpload.* [Online] http://commons.apache.org/fileupload/.

13. **Mamo, Ron.** *Reflections.* [Online] http://code.google.com/p/reflections/.

14. **Crane, D., Pascarello, E. a James, D.** *Ajax in Action.* Manning Publications, 2005.

KAPITOLA 8
SEZNAM POUŽITÝCH ZKRATEK

AJAX – Asynchronous Javascript and XML

ASP – Active Server Pages

API - Application Programming Interface

DAO – Data Access Object

HTML - HyperText Markup Language

HTTP - Hypertext Transfer Protocol

IDE - Integrated development environment

Java EE – Java Enterprise Edition

JPA – Java Persistence API

JSON - JavaScript Object Notation

ORM - Object-relational mapping

PHP – Hypertext Preprocessor

SQL - Structured Query Language

UI – User Interface

UML – Unified Modeling Language

URL – Uniform Resource Locator

XHTML - Extensible HyperText Markup Language

XML - Extensible Markup Language

XSLT - Extensible Stylesheet Language Transformations

KAPITOLA 9
OBSAH CD

/kořen
- /dist
  - /neptuo-server.war
  - /neptuo-webdesk.war
- /doc
- /install
  - /glassfish-3.0.1-windows.exe
  - /xampp-win32-1.7.3.exe
- /script
  - /neptuo_default.sql
- /source
  - /neptuo-data
  - /neptuo-server
  - /neptuo-service
  - /neptuo-webdesk
- /uml

Složka **dist** obsahuje zkompilovanou verzi klientské a serverová části aplikace.

Složka **doc** obsahuje tento dokument.

Složka **install** obsahuje instalační soubory k ostatním aplikacím nutných pro spuštění aplikace.

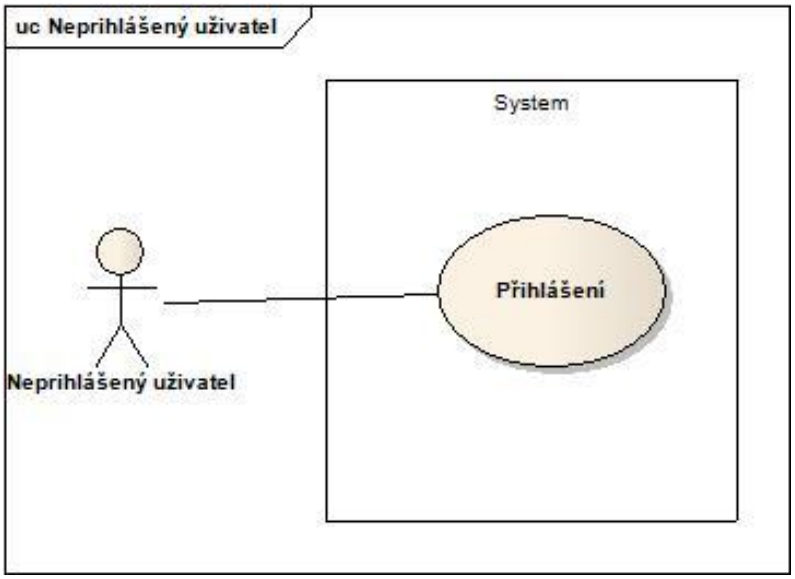
Složka **script** obsahuje SQL skript s výchozími daty pro novou databázi, bez kterých nebude aplikace korektně fungovat. V tomto skriptu je případně potřeba změnit nastavení kořenových složek, více *Příloha D: Instalační příručka*.

Složka **source** obsahuje zdrojové kódy a projekty pro NetBeans IDE ke všem čtyřem projektům.

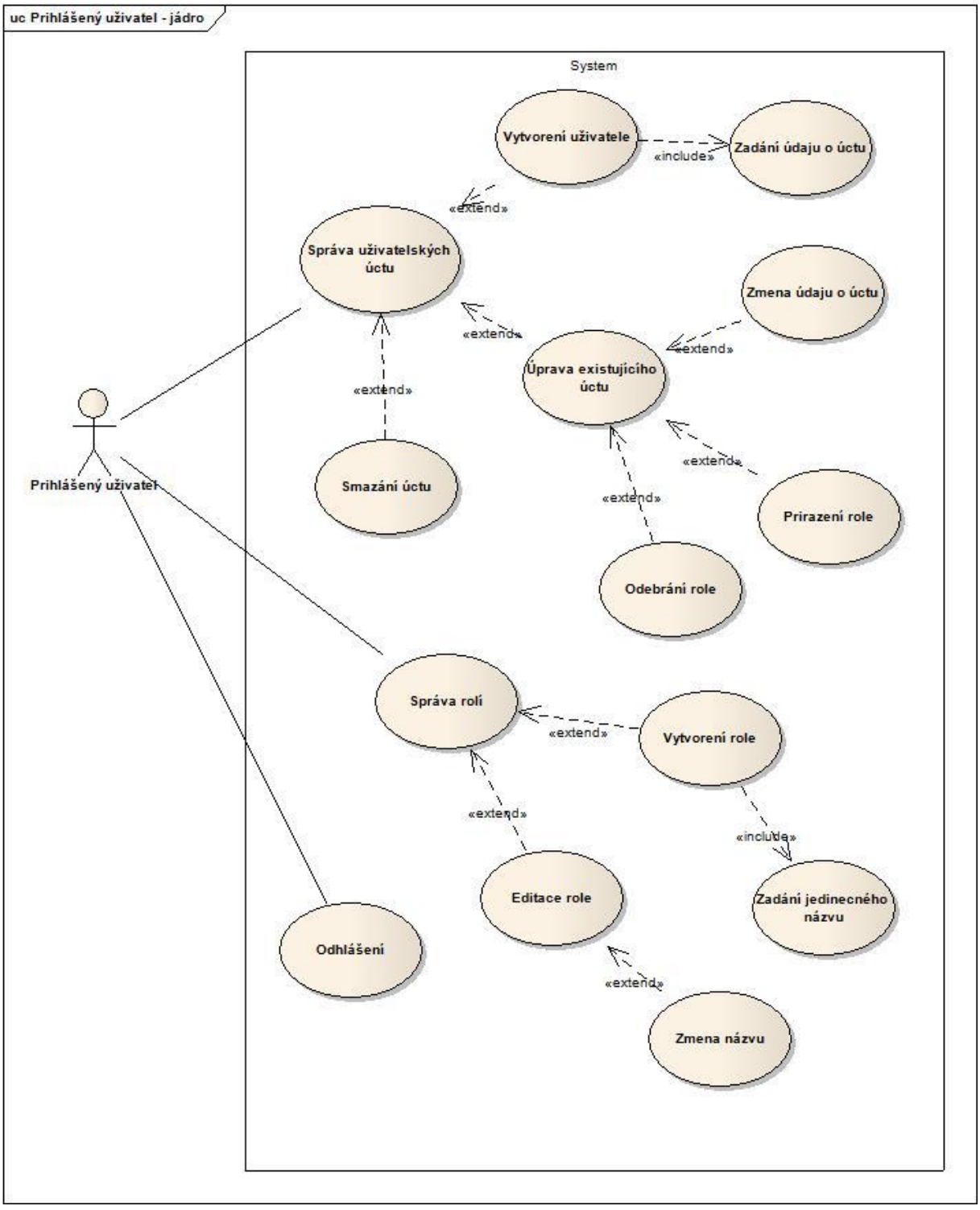
Složka **uml** obsahuje exportované UML diagramy vytvořené při analýze aplikace.

PŘÍLOHA A
UML DIAGRAMY

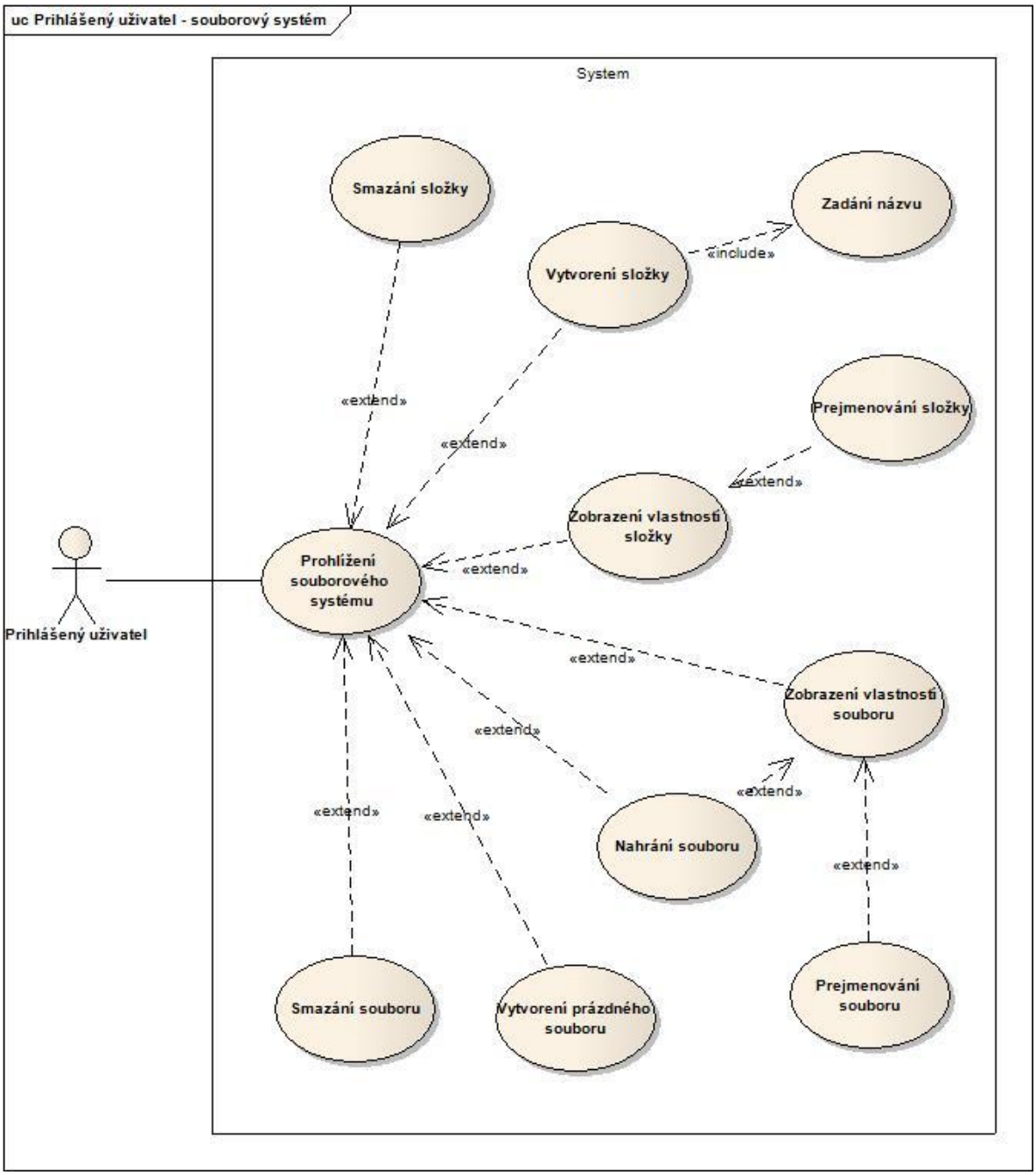
Tato příloha obsahuje UML diagramy modelované při analýze a návrhu aplikace. Jsou také obsaženy na CD ve složce uml.



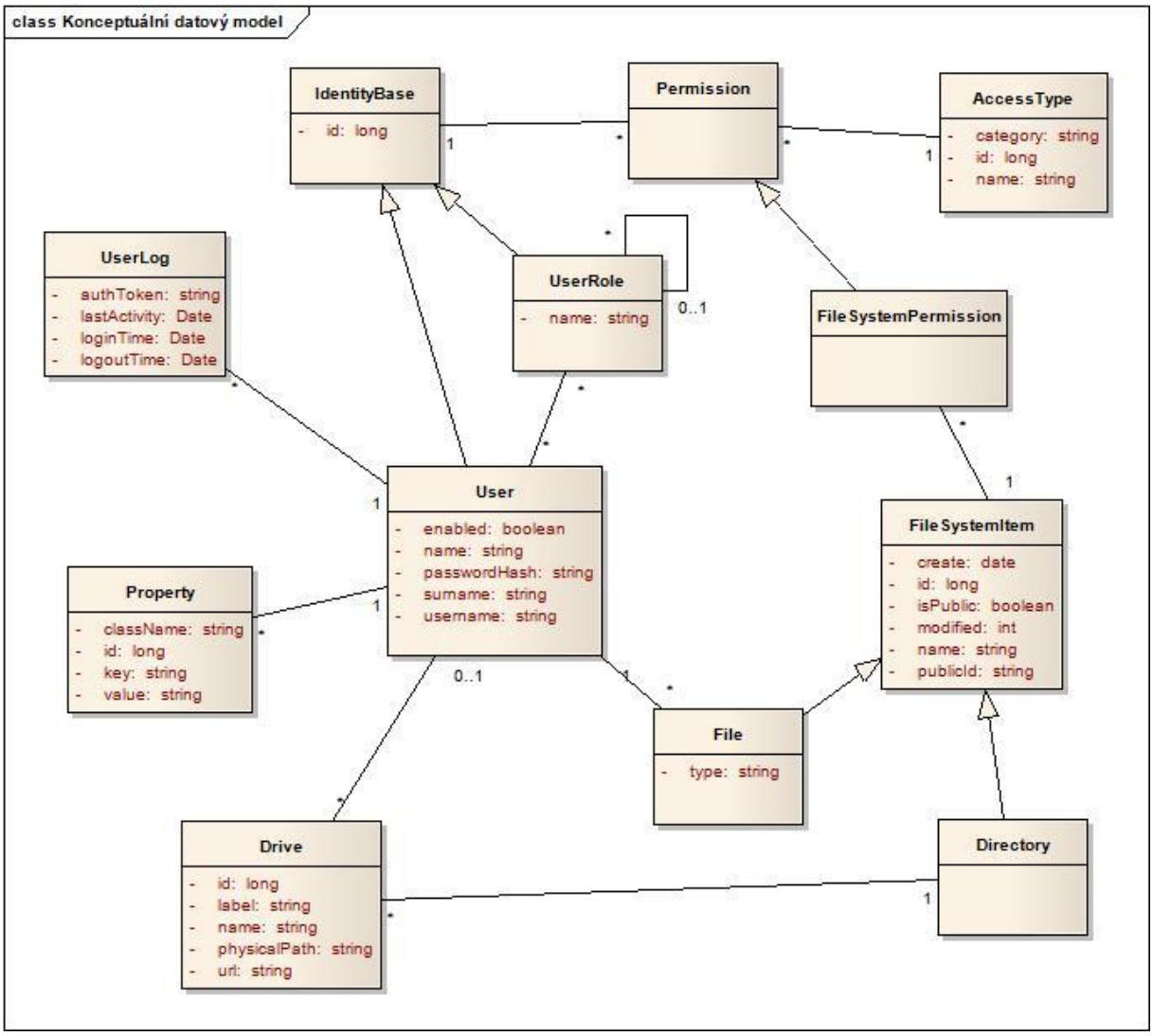
Obrázek A.1 Případy užití - nepřihlášený uživatel



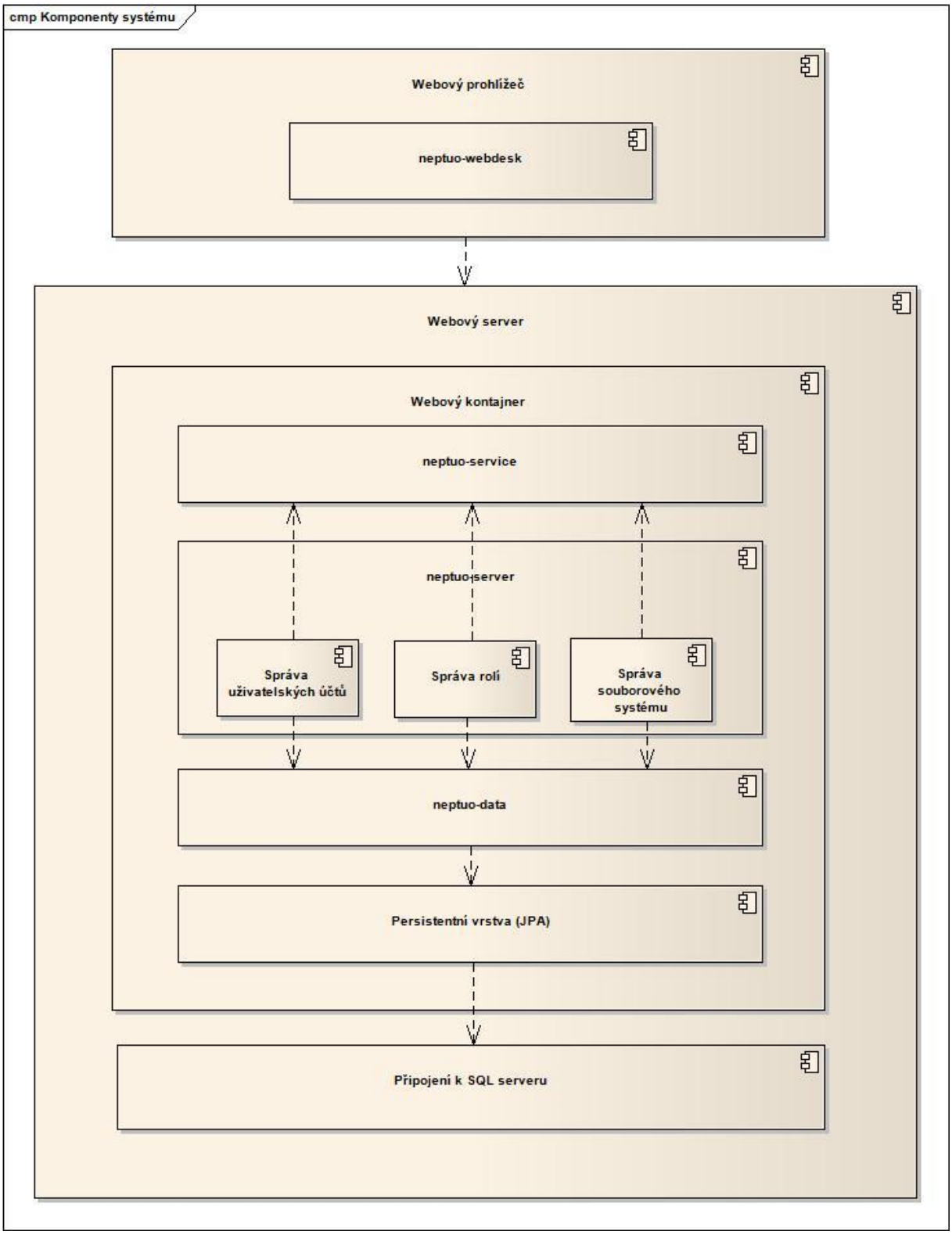
Obrázek A.2 Případy užití - přihlášený uživatel - jádro



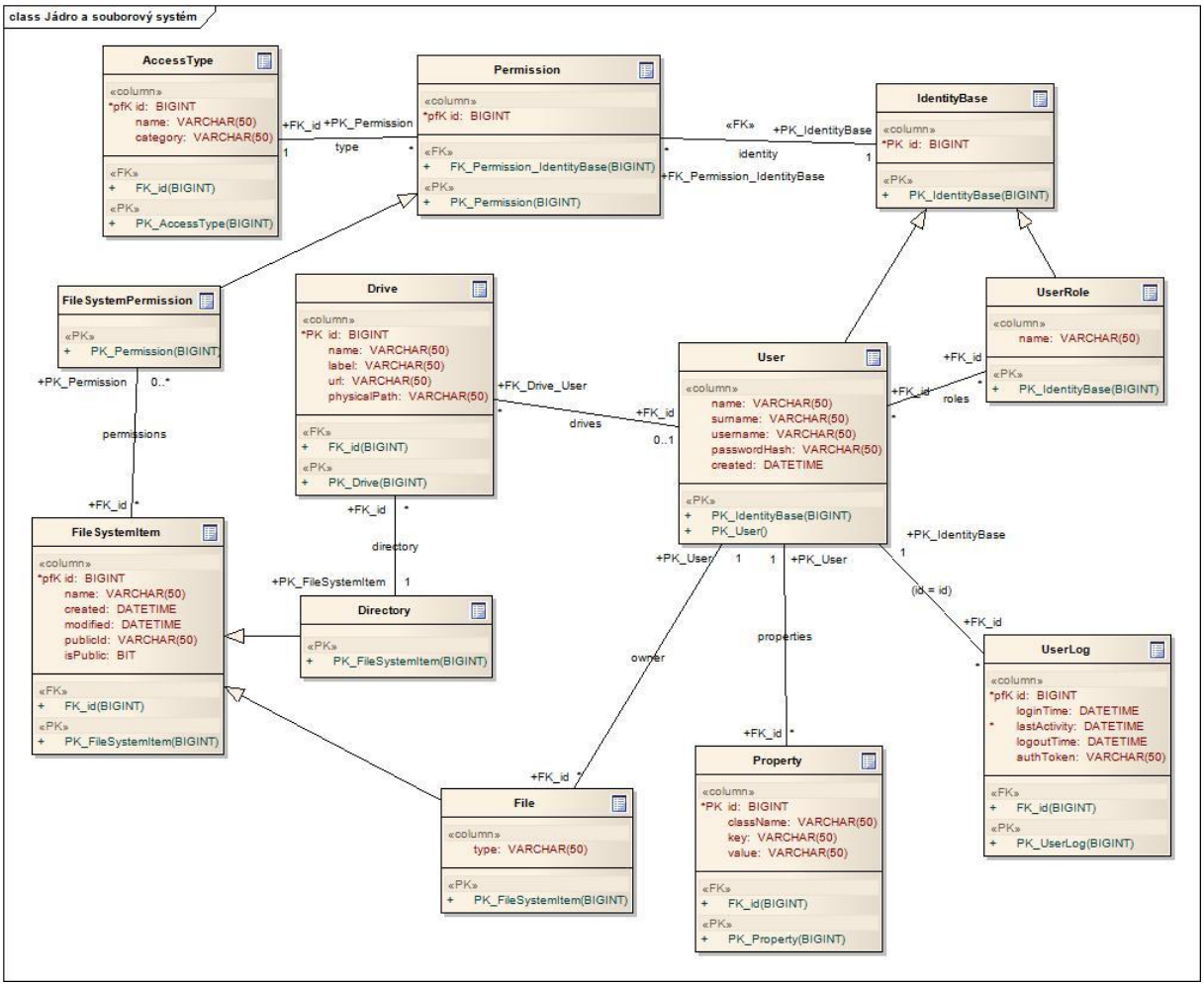
Obrázek A.3 Přihlášený uživatel - souborový systém



Obrázek A.4 Konceptuální datový model



Obrázek A.5 Komponenty systému



Obrázek A.6 Databázový model

PŘÍLOHA B
ZDROJOVÉ KÓDY

Tato příloha obsahuje ukázky zdrojových kódů, většina je obsažena na CD ve složce source, některé byly v průběhu implementace nahrazeny jiným řešením.

```
1  package com.neptuo.os.core.service;
2
3  import com.neptuo.os.service.serialization.RequestParser;
4
5  public class LoginRequestParser implements RequestParser {
6      private final String LOGIN_KEY = "login";
7      private final String USERNAME_KEY = "username";
8      private final String PASSWORD_KEY = "password";
9      private final String DOMAIN_KEY = "domain";
10
11     private String current = "";
12     private String username;
13     private String password;
14     private String domain;
15
16     @Override
17     public void startElement(String name) {
18         if(USERNAME_KEY.equals(name)
19             || PASSWORD_KEY.equals(name) || DOMAIN_KEY.equals(name)) {
20             current = name;
21         }
22     }
23
24     @Override
25     public void content(String content) {
26         if(USERNAME_KEY.equals(current)) {
27             username = content;
28         } else if(PASSWORD_KEY.equals(current)) {
29             password = content;
30         } else if(DOMAIN_KEY.equals(current)) {
31             domain = content;
32         }
33     }
34
35     @Override
36     public void endEmenet(String name) {
37         return;
38     }
39
40     public String getUsername() { return username; }
41
42     public String getDomain() { return domain; }
43
44     public String getPassword() { return password; }
45 }
```

B1 Zdrojový kód pro třídu LoginRequestParser, která byla později nahrazena třídou LoginItem

```
1  ...
2
3  protected void write(List<User> users, Serializer s) {
4       Options o1 = new Options();
5       o1.add(JsonSerializer.ESCAPE_SEPARATOR_KEY);
6       Options o2 = new Options();
7       o2.add(JsonSerializer.ESCAPE_ELEMENT_KEY);
8
9       s.writeElementStart("users", o1);
10      s.writeArrayStart();
11      for (User u : users) {
12          s.writeElementStart("user", o2);
13          s.writeKeyValue("id", u.getId());
14          s.writeKeyValue("name", u.getName());
15          s.writeKeyValue("surname", u.getSurname());
16          s.writeKeyValue("username", u.getUsername());
17          s.writeKeyValue("passwordHash", u.getPasswordHash());
18          s.writeKeyValue(
19              "created",
20              u.getCreated() == null ? "" : u.getCreated().toString()
21          );
22          s.writeKeyValue("enabled", u.getEnabled());
23          s.writeElementEnd("user", o2);
24      }
25      s.writeArrayEnd();
26      s.writeElementEnd("users", o1);
27  }
28
29  ...
```

B2 Část zdrojového kódu, který se stará o zapsání kolekce uživatelů do výstupního serializeru

```
1  package com.neptuo.os.core.data.model;
2
3  import ...;
4
5  @Entity
6  @Table(name = "core_accesstype")
7  @Serializable(name="accessType")
8  @Deserializable(name="accessType")
9  public class AccessType extends AbstractEntity
10     implements java.io.Serializable {
11
12     private static final long serialVersionUID = 5182060875486476822L;
13     private Long id;
14     private String name;
15     private String category;
16
17     public AccessType() {
18     }
19
20     public AccessType(String name) {
21         this.name = name;
22     }
23
24     @Id
25     @GeneratedValue
26     @Override
27     @Serializable(name="id", primary=true)
28     public Long getId() {
29         return this.id;
30     }
31
32     @Deserializable(name="id")
33     public void setId(Long id) {
34         this.id = id;
35     }
36
37     @Column(name = "name", length = 20, unique = true)
38     @Length(max = 20)
39     @Serializable(name="name")
40     public String getName() {
41         return this.name;
42     }
43
44     public void setName(String name) {
45         this.name = name;
46     }
47
```

```
48      @Column(name = "category", length = 20)
49      @Length(max = 20)
50      @Serializable(name="category")
51      public String getCategory() {
52          return category;
53      }
54
55      @Deserializable(name="category")
56      public void setCategory(String category) {
57          this.category = category;
58      }
59  }
```

B3 Zdrojový kód databázové entity AcceType využívající anotace pro automatický zápis a načtení objektu třídami AutoSerializer, respektive AutoDeserializer

```
1  package com.neptuo.os.core.service;
2
3  import ...;
4
5  @ServiceClass(url = "/core/login")
6  public class LoginService {
7
8      @ServiceMethod
9      public CollectionResult<User> doLogin(UserDAO dao, DriveDAO drives,
10             @RequestInput("login") LoginItem item,
11             HttpServletResponse response)
12         throws DataStorageException, HttpException {
13
14         UserLog log = dao.login(
15             item.getUsername(),
16             item.getPassword(),
17             item.getDomain()
18         );
19         ResponseHelper.setAuthToken(response, log.getAuthToken());
20
21         if(drives.findUserHome(log.getUser()) == null) {
22             drives.createUserHome(log.getUser());
23         }
24
25         List<User> list = new ArrayList<User>();
26         list.add(log.getUser());
27         return new CollectionResult<User>("users", list);
28     }
29
30     @ServiceMethod(name = "isLogged", transactional = false)
31     public CollectionResult<User> checkLogged(User current)
32         throws ServiceException, IOException {
33
34         List<User> list = new ArrayList<User>();
35         list.add(current);
36         return new CollectionResult<User>("users", list);
37     }
38 }
```

B4 Finální implementace servisní třídy pro příhlášení uživatele. Má dvě veřejné metody, první pro příhlášení uživatele zapomocí uživatelského jména, hesla a domény, druhá zkontrolu, zda je uživatel přihlášen, její tělo již nemusí nic kontrolvat, pouze zapíše do výstupu informace o aktuálním uživateli, celá kontrola je províděna již v jednom z poskytovatelů parametrů, pokud se metoda snaží získat entitu aktuální uživatele a ten není přihlášen.

PŘÍLOHA C

C1. UŽIVATELSKÁ PŘÍRUČKA

Přihlašovací obrazovka – Obrazovka pro přihlášení je jednoduchá a stadartní, obsahuje tři pole pro zadání uživatelského jména, hesla a domény. Tlačítkem *Přihlásit* případně stiskem klávesy enter se lze přihlásit. V levém horním rohu je navíc možnost výběru jazyku, zatím jsou podporovány čeština a angličtina.

Rozhraní - Uživatelské rozhraní se skládá ze tří částí, viz obrázek C2.2. Horního hlavního menu, obsahující tlačítka pro spouštění jednotlivých aplikací a tlačítko pro odhlášení uživatele. Spodní lišty, nazvané Taskbar, která zobrazuje aktuálně spuštěné aplikace a umožňuje jejich přenesení do popředí. A vlastní plochy, do které jsou umísťována jednotlivá okna aplikací.

Hlavní menu - Hlavní menu v levé části obsahuje tlačítka pro spuštění dostupných aplikací. V pravé části pak tlačítko pro odhlášení uživatele a tlačitko pro obnovené aktuálního přihlášení uživatele. Pokud je uživatel 30 minut neaktivní v komunikaci se serverem, jeho přihlášení vyprší je nucen se znovu přihlásit, toho tlačítko zobrazuje čas do vypršení přihlášení a jeho stisk odešle požadavek na server, čímž je automaticky přihlášení prodloužena opět na 30 minut.

Plocha – Plocha zabírá nejvíce místa, slouží k zobrazování oken jednotlivých spuštěných aplikací. Navíc pokud jsou v domovské jednotce nějaké složky nebo soubory, tak se tyto zde zobrazí. Dvojklikem pak lze zobrazit obsah vybrané složky v aplikaci Explorer nebo vybraný soubor stáhnout, případně otevřít v TextEditoru.

Taskbar - Taskbar zobrazuje z seznam spuštěných aplikací. Úplně na levé straně je tlačítko pro minimalizaci všech spuštěných aplikací. Na pravé straně je pak zobrazen aktuální čas a tlačítka pro zobrazení zpráv systému a zaslání feedbacku.

Okna – Okna mají základní funkčnost jako okna ve Windows, lze je posouvat, roztahovat, minimalizovat, maximalozovat, případně zavřít. Některá okna jsou tak zvané dialogy, které znemožňují práci s dalšími okny dokud nejsou potvrzena nebo zavřena. Vetšina oken krom základních tlačítek v záhlaví pro manipulaci s oknem, ještě obsahuje tlačítko, pro znovu načtení dat zobrazených v okně.

Explorer – Tato aplikace, viz Obrázek C2.3, slouží k procházení souborového systému. Hlavní okno má v horní části dva panely.

V horním jsou vlevo tři tlačítka, dvě pro navigaci vpřed a zpět v historii a prostřední pro přesunutí u úroveň výše. Vedle je adresní řádek, kam lze přímo zadat adresu složky, kterou chceme otevřít, to pak lze potvrdit stiknutím klávesy enter nebo kliknutím na tlačítko úplně vpravo. Pod tímto panelem je panel s tlačítky pro různé akce. Tlačítko úplně vlevo obsahuje nabídku na vytvoření složky, prázdného souboru, nebo pro nahrání souboru z disku. Poslední položka otevře aplikaci pro nahrání více souborů. Tato aplikace umožňuje soubor případně před uložením na serveru přejmenovat. Další tlačítko pak umožňuje vybranou položku stáhnout. Pokud je vybrána složka, pak se stáhne celá složka, se všemi podsložkami a soubory ve formátu zip. Další tlačítko umožňuje vybraný soubor nebo složku přesunout nebo zkopírovat na jiné místo. Tato funkce funguje obdobně jako schránka ve Windows, pokud máme vybraný nějaký soubor nebo složku, stačí stisknout kopírovat nebo vyjmout, přejít do složky, kam chceme

soubor nebo složku vložit a stiknout vložit. Lze také kopírovat nebo přesouvat mezi jednotlivými okny aplikace Explorer. Pokud si v jednom okně něco vyjmeme, pak ve druhém můžeme dát vložit. Předposlední tlačítko umožňuje zobrazit a editovat podrobnosti o souboru či složce, položku je možné přejmenovat a v případě souboru, lze stávající soubor nahradit nahraním nového. Na další záložce okna s podrobnostmi se dají nastavit přístupová práva pro daný objekt. Tlačítko *Přidat* otevře okno pro vyhledání uživatele nebo role, viz Obrázek C2.4, stačí do textového pole zadat část uživatelského jména nebo jména role a stisknout *Hledej*, pak jen vpravo vybrat jaké oprávnění chceme přidat, ze seznamu vybrat uživatele nebo roli a stiknout *Přidat*. Oprávnění jsou následující, základní oprávnění je *Read*, které umožňuje uživateli zobrazit obsah dané složky nebo souboru, další oprávnění je *ReadWrite*, které umožňuje do dané složky nebo souboru i zapisovat. Nejvyšší oprávnění je pak *Manage*, které umožňuje uživatel i daný soubor nebo složku smazat, případně nastavovat jeho oprávnění. Poslední tlačítko vlevé části pak umožňuje soubor smazat. Na pravé části tohoto panelu jsou prvky pro filtrování zobrazeného obsahu, první tlačítko umožňuje zobrazovat jen složky, následuje textové pole pro zadání začátku názvu a textové pole pro filtrování podle koncovky souboru, tyto dvě textové pole se aplikují jen na soubory.

Hlavní obsah tvoří dva seznamy, nalevo je seznam jednotek, ke kterým máme přístup. Otevřít ji lze kliknutím, případně dvojkliknutím. Jednotky jsou rozděleny na systémové a uživatelské, zde mají odelišnou ikonu. Tento seznam lze minimalizovat kliknutím na tlačítko v záhlaví seznamu. Napravo je pak obsah zobrazené složky. Krom sloupců, kterou jsou vidět, se dají zobrazit další, stačí po najetí nad hlavičku sloupce kliknout levým tlačítkem na zobrazenou šipku, přejet na *Sloupce* a zaškrtat požadované slopce. Celý seznam lze seřadit kliknutím na hlavičku sloupce. Ve spodní části okna je panel, který zobrazuje některé další vlastnosti vybraného záznamu.

UserManager – UserManager slouží ke správě uživatelských účtů a rolí. Rozhraní se skládá ze dvou záložek. Na první lze spravovat uživatelské účty. Horní část zobrazuje dostupné uživatelské účty. Spodní pak formulář pro zadání údajů, případně při překliknutí na “Seznam rolí ...” se zobrazí seznam dostupných rolí a žaškrtané jsou role, které uživatel má přiřazené. Při editaci uživatele je možné nevyplňovat heslo,v takovém případě zůstane heslo nezměné. Při vytváření je však nutné heslo vyplnit. Uživatelský účet je možné smazat pouze před prvním přihlášením uživatele, pak je možné tento účet jen zablokovat, toho lze dosáhnout odškrtnutím zaškrtávátka *Povolený.*

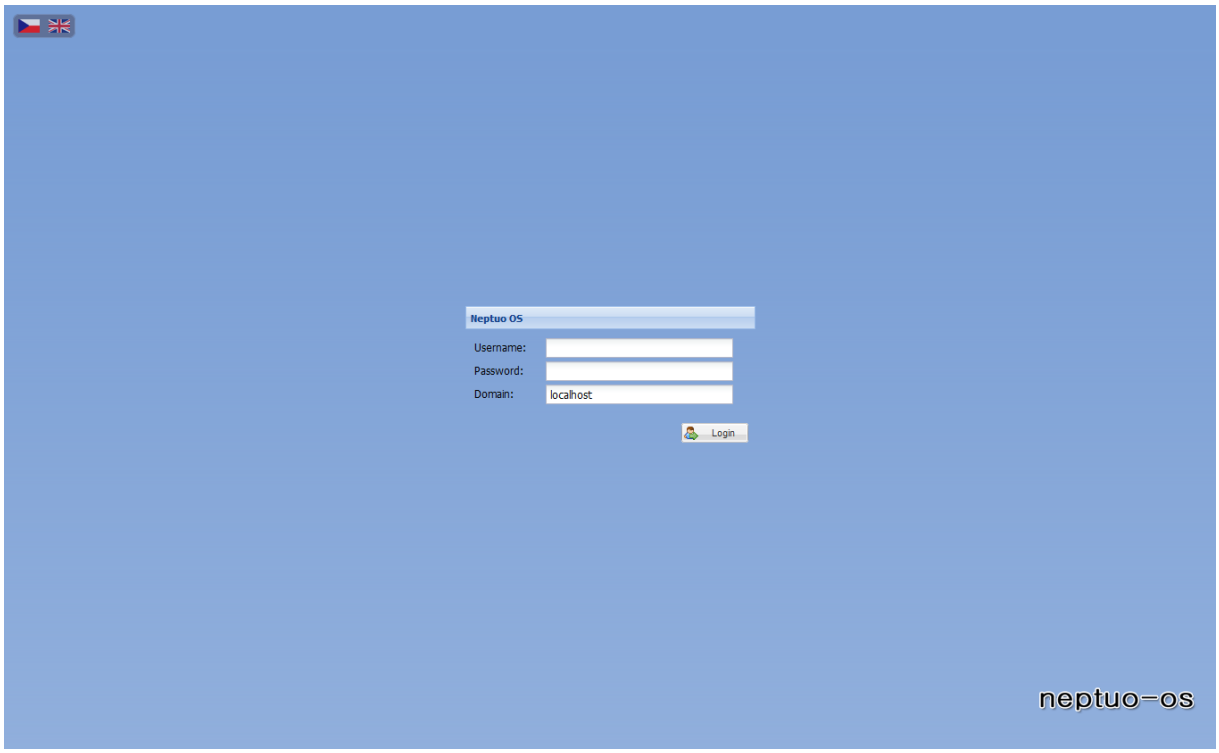
Na druhé záložce je pak editace rolí. Role musí mít jedinečný název a rodičovskou roli. Uživatel má právo editovat a přiřazovat pouze role, jejichž nepřímým předkem je jedna z rolí, které má přiřazené.

DriveManager – Tato aplikace umožňuje spravovat jednotky. U systémových jednotek umožňuje nastavovat oprávnění ke kořenové složce, to je samozřejmě podmíněné oprávněním *Manage* pro danou složku. Dále umožňuje vytvářet a spravovat uživatelské jednotky. Ty slouží jako zkratky v souborém systému. Pokliknutí na *Vytvořit* nebo *Editovat* se zobrazí okno s jednoduchým formulářem, ten obsahuje jméno jednotky, popisek a tlačítko pro výběr kořenové složky jednotky, stiknutím tohoto tlačítka se otevře aplikace Explorer, ve které je potřeba vybrat složku, která bude kořenovou pro tuto jednotku. Stiskem uložit se pak změny potvrdí.

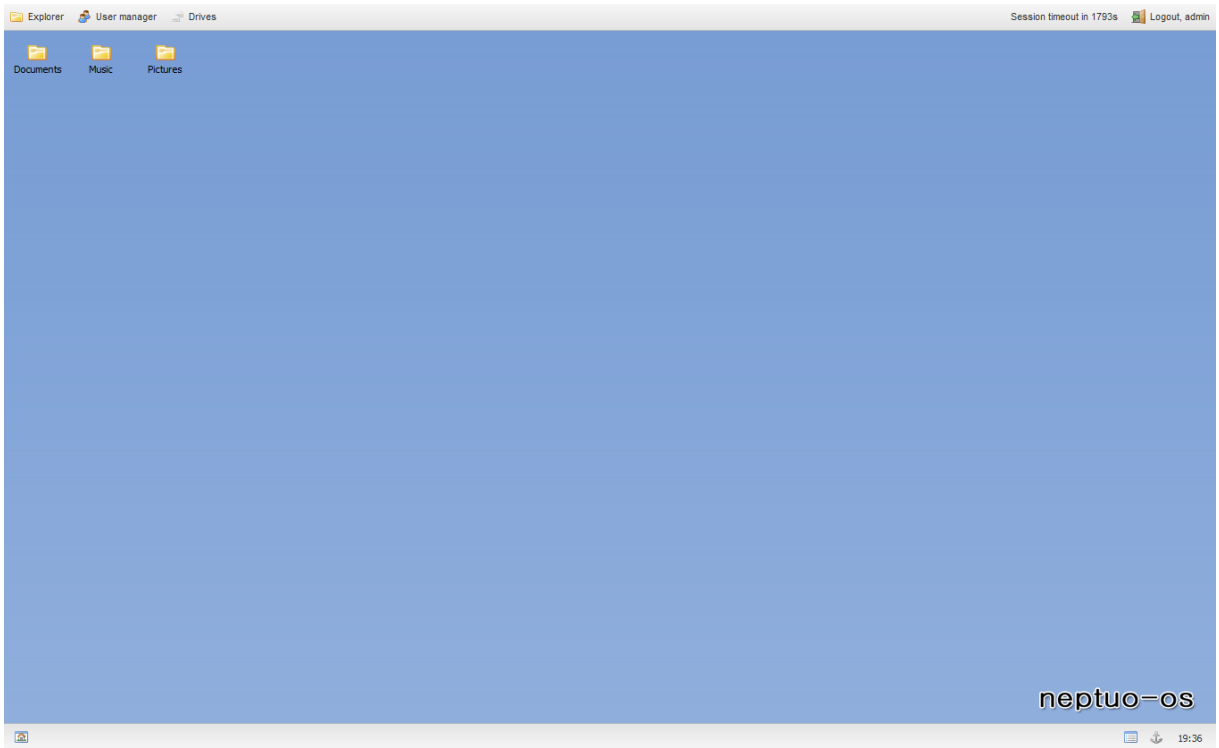
TextEditor – TextEditor je aplikace pro editaci textu, a to buď čistého textu nebo formátovaného textu. Dá se spustit jedině z aplikace Explorer dvojklikem na textový nebo HTML

soubor. Pokud chceme vytvořit nový soubor, je nutné v aplikaci Explorer v požadované složce dát *Vytvořit* > *Vytvořit prázdný soubor*, zadat mu jméno a koncovku .txt, pro textový soubor, nebo .html pro HTML soubor, tedy formátovaný soubor. Rozhraní aplikace je pak velmi jednoduché. Obsahuje tři tlačítka pro *Otevření*, *Uložení* a *Uložení jako* a vlastní editační pole. Při editaci formátovaného textu pak ještě obsahuje lištu s tlačítky pro formátování textu.

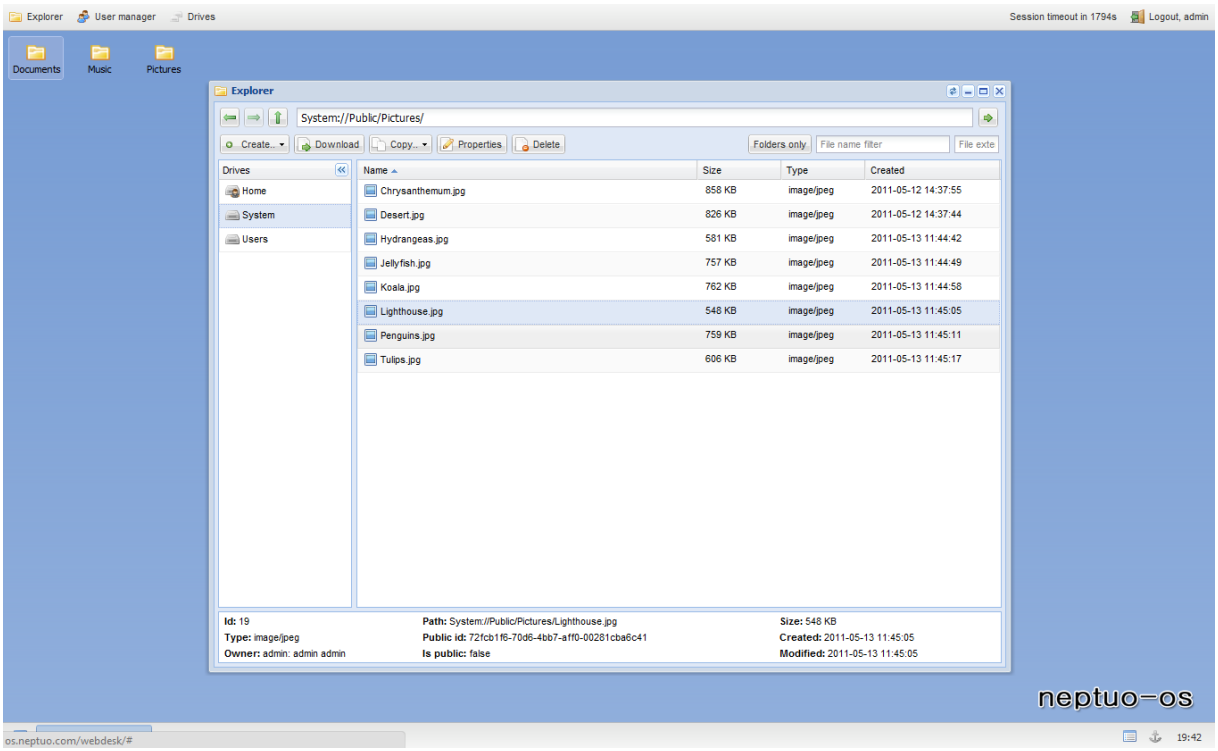
C2. NÁHLEDY APLIKACE



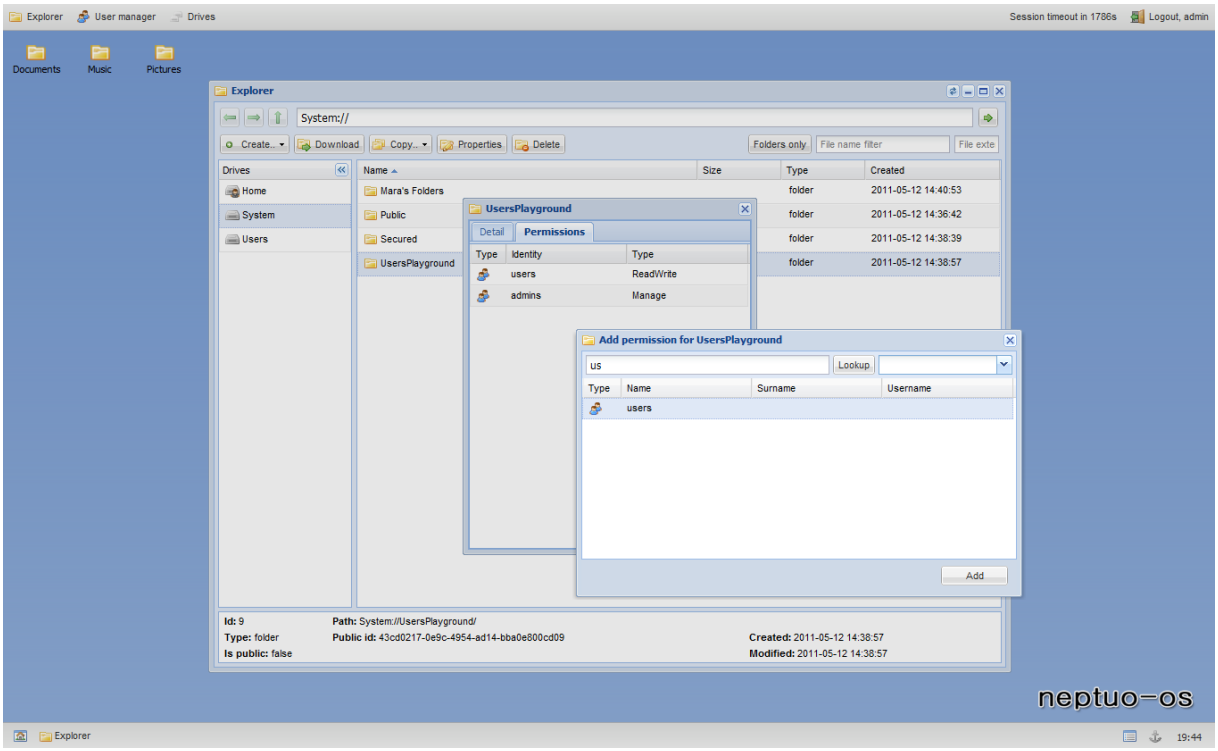
Obrázek C2.1 Přihlašovací obrazovka



Obrázek C2.2 Plocha po přihlášení



Obrázek C2.3 Aplikace Explorer zobrazující obsah složky System://Public/Pictures/



Obrázek C2.4 Nastavení oprávnění ke složce v Exploreru

PŘÍLOHA D
INSTALAČNÍ PŘÍRUČKA

1) Instalace aplikačního serveru GlassFish V3
   - z přiloženého CD nainstalovat aplikační server GlassFish (pro Windows uživatele)
2) Instalace databázového serveru MySQL
   - z přiloženého CD nainstalovat databázový server MySQL (pro Windows uživatele)
3) Konfigurace serverových zdrojů
   - V databázovém stroji je nutné vytvořit prázdnou databázi a nastavit přístupová práva pro účet, kterým se bude aplikační server k databázi připojovat
   - Nad databází spustit skript neptuo_default.sql (ze složky script na přiloženém CD)
     - Aplikace vyžaduje dva fyzické adresáře, jeden pro uživatelské účty a druhý pro systémovou jednotku, v tomto skriptu jsou jejich cesty nastaveny na *D:/Temp/FileBrowser/Users*, respektive *D:/Temp/FileBrowser/System*. V případě použití jiných adresářů, je nutné tyto cesty změnit.
   - V aplikačnám serveru GlassFish nakonfigurovat
     - ConnectionPool, který se bude připojovat k vytvořené databázi
     - DataSource, který se bude jmenovat NeptuoOSDefault a bude využívat vytvořený ConnectionPool
   - Nahrát na server aplikaci neptuo-server.war z přiloženého CD (složka dist), nastavit její ContextPath na /server (pokud je potřeba mít serverovou část aplikace na jiné ContextPath, pak je nutné změnit nastavení v Configuration.properties v projektu neptuo-webdesk a tento projekt znovu sestavit)
   - Nahrát na server aplikaci neptuo-webdesk.war