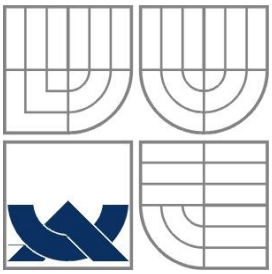


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

WEBOVÁ APLIKACE HELPDESK A SYNCHRONIZACE DAT

TITLE

DIPLOMOVÁ PRÁCE
DIPLOMA THESIS

AUTOR PRÁCE **BC. PAVEL BALOGH**
AUTHOR

VEDOUCÍ PRÁCE **PROF. RNDR. ING. JIŘÍ ŠŤASTNÝ, CSC.**
SUPERVISOR

BRNO 2012

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2011/12

ZADÁNÍ DIPLOMOVÉ PRÁCE

student(ka): Bc. Pavel Balogh

který/která studuje v **magisterském studijním programu**

obor: **Aplikovaná informatika a řízení (3902T001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Webová aplikace HelpDesk a synchronizace dat

v anglickém jazyce:

Web application HelpDesk and data synchronization

Stručná charakteristika problematiky úkolu:

Tvorba rozhraní (pomocí webových služeb) pro přenos dat mezi DB Postgre (software Goofer) a DB MySQL na webovém serveru. Webové služby budou realizovány v PHP pomocí třídy NuSOAP pro přenos dat na i z web serveru. Následovat bude tvorba generátoru těchto služeb pomocí xml a xsl transformace.
Tvorba vícevrstvé webové aplikace HelpDesk bude zahrnovat návrh datové struktury aplikace, vrstvu DAO (Database Access Object), aplikační vrstvu a vrstvu GUI.

Cíle diplomové práce:

Řešení synchronizace dat pomocí webových služeb pro přenos dat v prostředí PHP. Tvorba generátoru těchto služeb.
Tvorba webové aplikace HelpDesk bude obsahovat návrh datové struktury aplikace, implementaci vrstvy DAO (Database Access Object), implementaci aplikační vrstvy a vrstvy GUI v prostředí PHP.

Seznam odborné literatury:

[1] Darie, C. a kol.: AJAX a PHP - tvoříme interaktivní webové aplikace profesionálně, Zonerpress, 2006.
[2] Welling, L., Thomson, L.: PHP a MySQL - rozvoj webových aplikací, Softpress, Praha, 2002.
[3] Gutmans, A., Bakken, S. S., Rethans, D.: Mistrovství v PHP 5, Computer press, Brno, 2005.
[4] Urman, S., Hadman, R., McLaughlin, M.: ORACLE, Programování- v PL/SQL. CPress, Brno, 2007.

Vedoucí diplomové práce:doc. RNDr. Ing. Jiří Šťastný, CSc.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2011/12.

V Brně, dne 24. 11. 2011

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STROJNÍHO INŽENÝRSTVÍ DĚKANÁT

Ing. Jan Roupec, Ph.D.
Ředitel ústavu

prof. RNDr. Miroslav Doupovec, CSc.
Děkan

ABSTRAKT

Diplomová práce se zabývá vývojem webové aplikace Helpdesk, která zajišťuje komunikaci s internetovými uživateli. V první části práce je stručný popis jednotlivých prostředků a nástrojů, které byly použity. V praktické části je popsán návrh a vývoj jednotlivých vrstev webové aplikace. Je zde zmíněno i vytvoření a použití nástrojů pro synchronizaci a generování kódu.

ABSTRACT

The thesis deals with the development of the web-based Helpdesk application which ensures and supports the communication with Internet users. The first section of the thesis contains a brief description of the individual means and tools used by the developer. The practical part describes the design and development of the individual layers of the web-based application concerned. The latter section also mentions the development and usage of the required synchronization and code generation tools.

KLÍČOVÁ SLOVA

Helpdesk, synchronizace, generátor kódu, webová služba, PHP, MySQL, PostgreSQL.

KEYWORDS

Helpdesk, synchronization, code of generator, web service, PHP, MySQL, PostgreSQL.

BIBLIOGRAFICKÁ CITACE

BALOGH, P. Webová aplikace HelpDesk a synchronizace dat. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2012. 79 s. Vedoucí diplomové práce Prof. RNDr. Ing. Jiří Šťastný, CSc.

PROHLÁŠENÍ O ORIGINALITĚ

Prohlašuji, že jsem diplomovou práci zpracoval samostatně dle pokynů vedoucího práce s použitím uvedené literatury.

Datum: 25. 5. 2012 .....

PODĚKOVANI

Na tomto místě bych rád poděkoval vedoucímu práce Prof. RNDr. Ing. Jiřímu Šťastnému, CSc. za cenné rady a připomínky, které vedly k úspěšnému dokončení diplomové práce.

Obsah:

1 ÚVOD

V dnešní době plné informací má téměř každá organizace velkého i malého rozsahu své informační systémy. Pomocí informačních systému řídí své informace o zákaznících, o výrobě, zásobách, cashflow, workflow, atd. Bez těchto systémů a dat si již některé z nich nedokáží svojí existenci představit.

Stoupá jejich potřeba mít svá data kdykoliv dostupná a mít o nich přehled a možnost je zpracovávat kdekoliv na světě. K tomuto trendu přispívá masivní rozšíření internetu do všech oblastí. Softwarové firmy se začaly soustředit na tento trh, který díky internetu vznikl. Zaměřují se na možnosti, aby jejich software uměl komunikovat s účastníky na internetu, věnují pozornost zveřejňování určitých fragmentů dat např. produktů v e-shopech (možnost prodeje před internet), zpřístupňování dat pomocí portálů různých typů. Firmy začínají budovat internetové nástavby na své software a umožňují tak svým zákazníkům komunikovat přes internet.

Ani firma AbisTech s.r.o. nemohla zaostávat za ostatními a musela reagovat na rozmach internetu. Firma AbisTech s.r.o. je malá česká softwarová společnost, která se zaměřuje na řízení CRM oblasti firmy a jejím hlavním produktem je software Goofer.

Goofer je modulární systém postavený na moderních vývojových technologiích. Jedná se o celosvětově rozvinutou platformu .NET a přímo nástroj C#. Díky těmto nástrojům může být při vývoji využíváno velkého množství výhod a komfortu objektově orientovaného programování.

Slovem modulární je myšleno, že jednotlivé tématické oblasti software jsou rozřazeny do modulů a podle licenční politiky firmy si je lze zakoupit pro určité specializované oblasti např. kontakty, skladová evidence atd.

Dostalo se mi možnosti pro firmu AbisTech v rámci své diplomové práce zahájit vývoj nástroje pro komunikaci s webovým rozhraním a možnost vytvořit jeden z webových modulů.

V úvodu mé diplomové práce budete krátce seznámeni s technologiemi, které byly použity při tvorbě projektu a s průběhem a úskalími při vývoji. Dále bude následovat podrobná prezentace jednotlivých částí projektu. Na prvním místě bude seznámení se se způsobem přenosu data na webové rozhraní (synchronizace). Bude zmíněn nástroj pro generování kódu, o kterém se podrobněji dozvíme později. A v neposlední řadě bude řeč o webovém modulu HelpDesk, kde bude k vidění návrh a architektura samotné aplikace.

Při tvorbě diplomové práce jsem spolupracoval i na tvorbě částí systému Goofer a na určitých nástrojích. To bylo nezbytné pro zahájení vývoje webového modulu. Tyto úpravy a nástroje nejsou součástí diplomové práce, protože obsahují know how společnosti a ta si z pochopitelných důvodů nepřeje je zveřejňovat.

2 POUŽITÉ TECHNOLOGIE

V úvodu mé práce vás chci blíže seznámit s prostředky a technologiemi, které byly použity pro vývoj praktické části. Tato kapitola nám představí jednotlivé nástroje, bude zde zmínka nejen o jejich funkcionalitách ale krátce i historie. Určitým nástrojům budou věnovány jednotlivé podkapitoly.

2.1 Architektura klient – server

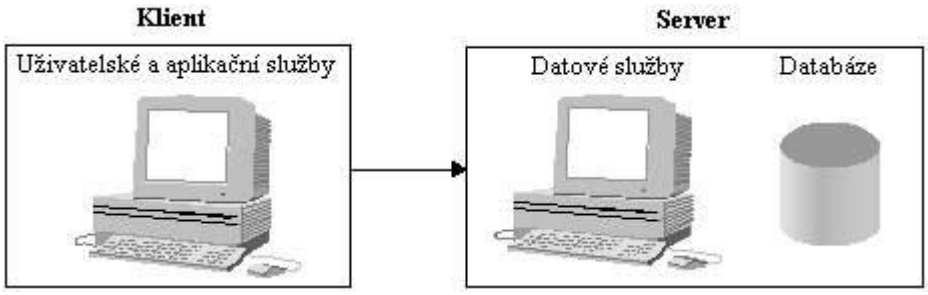
Architektura klient – server zajišťuje téměř všechny služby internetu a je zpravidla tvořena dvojicí programů:

- program typu *server* reaguje na požadavky ze strany klienta, požadavky zpracovává a odesílá výsledek zpět
- program typu klient se může dotazovat na jakýkoliv webový server a následně zobrazí výsledek

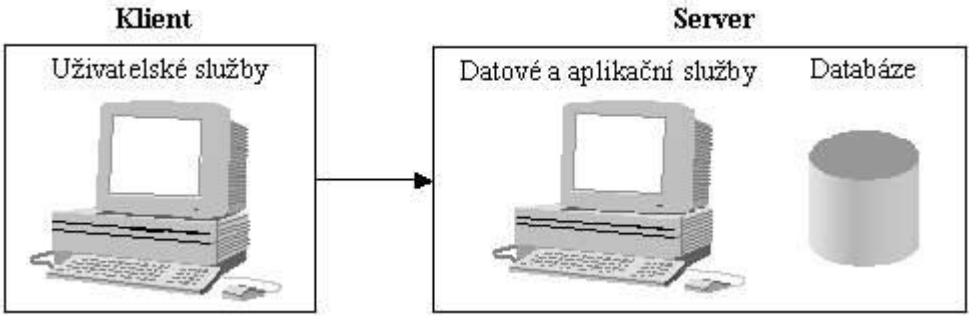
Komunikace mezi klientem a serverem se děje podle přesných formálních pravidel, pojmenovaných jako protokol HTTP (*HyperText Transfer Protocol*). Jedná se o internetový protokol, který byl určen k výměně hypertextových dokumentů mezi serverem a prohlížečem. HTTP funguje na bázi dotaz – odpověď, jednotlivé dotazy nejsou z pohledu serveru rozeznatelné. Z tohoto důvodu je HTTP protokol nazýván bezstavový. Tato vlastnost způsobuje problém při programování složitějších internetových aplikací, kde je potřeba si udržovat hodnoty proměnných. Vyplývající nedostatky je pak nutné obcházet pomocí tzv. *cookies* nebo *session* [1].

Architektura klient – server byla, jak již z jejího názvu vyplývá, původně koncipována jako dvouvrstvý model. Zmínka zde bude o dvou podtypech:

1. Architektura s výkonem soustředěným u klienta – veškeré požadavky na aplikační a uživatelské služby se zpracovávají u klienta. Slabinou tohoto řešení byla potřeba velké přenosové kapacity mezi klientem a serverem, kde probíhají velké počty datových procesů, viz obr. 1.
2. Architektura s výkonem soustředěným na server – klient dostává pouze požadované informace (tenký klient) na které se dotazuje na server viz obr. 2. Veškeré aplikační a datové služby probíhají na serveru [2].

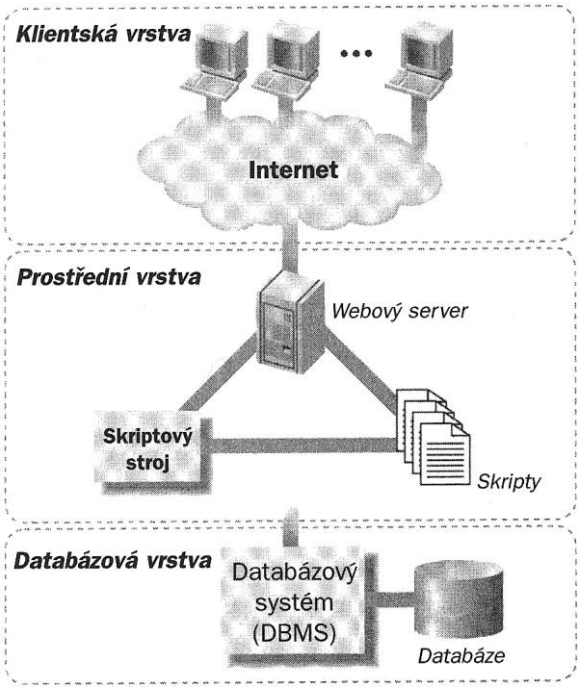


Obr. 1 Architektura s výkonem soustředěným u klienta



Obr. 2 Architektura s výkonem soustředěným na server

V současné době se často přechází z klasického klient – server modelu na model třívrstvý, znázorněný na obrázku 3. Tento model vychází z klasického dvouvrstvého řešení. Klientská vrstva zůstává beze změn, ale na straně serveru dochází k rozdělení aplikační a datové služby do samostatných logických modelů. Ty mohou být umístěny na jednom, popřípadě na dvou různých serverech. Třívrstvý model zajišťuje vyšší úroveň stability, protože umožňuje rozložení provozní zátěže ke zpracování na dvou samostatně běžících serverech [2]. U třívrstvého modelu si podrobněji popíšeme jednotlivé vrstvy a budou zde uvedeny i jednotlivé vývojové nástroje, které do vrstev patří.



Obr. 3 Třívrstvá architektura klient-server

2.2 Klientská vrstva

Klientskou vrstvu v třívrstvé architektuře tvoří obvykle webový prohlížeč. Tento software překládá zdroj informací ve formátu HTML, které následně zobrazí v dnešní době nejčastěji v grafické podobě. Dále vydává požadavky HTTP na potřebné zdroje a zpracovává odpovědi HTTP.

Webových prohlížečů je velká nabídka od různých firem (např. Microsoft – Internet explorer, Mozzilla – FireFox, Google – Chrome, atd.), přičemž každý z nich nabízí odlišné funkcionality a vlastnosti. Vlivem velké konkurence se vyvíjí nové a nové funkce a také

optimalizace výkonu a rychlosti. I když každá firma má odlišná řešení musí dodržovat normy, které stanovuje W3C. Základní společné vlastnosti prohlížečů:

- Všechny webové prohlížeče jsou klienty HTTP, které zasílají požadavky a zobrazují odpovědi z webových serverů
- Převážná většina prohlížečů dokáže zobrazovat grafické objekty, přehrávat videosekvence či zvukové záznamy
- Téměř všechny prohlížeče aplikují na stránky HTML, kaskádové šablony stylů CSS (Cascading Style Sheets), které definují způsob formátování elementů HTML

2.3 Prostřední vrstva

Ve většině třívrstvých webových databázových systémů se podstatná část aplikační logiky skrývá v prostřední vrstvě. Klientská vrstva pouze prezentuje data uživateli, zatímco databázová vrstva ukládá a načítá data. Prostřední vrstva zajišťuje zbývající role, které obě předešlé vrstvy spojují. Prostřední vrstva řídí strukturu a obsah dat zobrazených uživateli a zpracovává vstup od uživatele, na jehož základě formuluje dotazy pro zápis a čtení dat z databáze.

Prostřední vrstva je tvořena z komponent webového serveru, webových skriptovacích jazyků a stroje skriptovacího jazyku. Webový server zpracovává požadavky HTTP a vrací odpovědi na požadavky. V případě webových databázových aplikací směřuje tyto požadavky často k programům, jenž posléze komunikují s interním databázovým systémem.

2.3.1 Webový server – Apache

Webovým serverům se často říká také servery HTTP. Pojem „server HTTP“ vystihuje základní funkci webového serveru, a tou je naslouchat požadavkům HTTP ze sítě, přijmout příchozí požadavky HTTP od uživatele (nejčastěji od webového prohlížeče) a v neposlední řadě obsluha požadavku a následné vracení odpovědi HTTP s požadovaným zdrojem informací. Pro účely diplomové práce jsem zvolil téměř nejrozšířenější webový server a to Apache.

Apache patří mezi nejrobustnější implementace HTTP serveru, jedná se o řešení společnosti Apache Software Foundation. Vývoj Apache začal v roce 1993 v NCSA (*National Center for Supercomputing Aplications*) na Illinoiské univerzitě pod názvem NCSA HTTP. První veřejná verze s označením 0.6.2 byla vydána v dubnu 1995, od tohoto okamžiku rostla jeho oblíbenost a rozšířenost [3]. Apache se stává oblíbeným i proto, že se jedná o software s otevřeným kódem (open source) a pro svou modularitu. Pokud vám nějaká vlastnost chybí, můžete si ji sami naprogramovat jako samostatný modul a tento modul si do serveru přidat, to vše díky tomu, že je kód serveru volně dostupný [4]. Modularita je jednou z velkých výhod serveru a dává mu o ohromné možnosti. Moduly se natahují při startu serveru, jsou ve formátu DSO [*Dynamic Shared Object*]. Apache podporuje velkou škálu programovacích jazyku např. Perl, Python a v neposlední řadě PHP.

Přehled základních modulů Apache:

- mod_dir, mod_autoindes – pokud v adresáři existuje idnex.html nebo jiné soubory, které určíme, neprovede se standardní výpis adresáře, ale zobrazí se přímo tento soubor
- mod_cgi – zajišťuje spouštění CGI skriptů
- mod_userdir – umožňuje prezentovat webové stránky uživatelů
- mod_mine, mod_mine_magic – definuje základní typy dokumentů podle přípony nebo megic numbers
- mod_perl, mod_php – podpora různých jazyků přímo od serveru pro rychlejší zpracování
- mod_proxy – podpora pro proxy přístupy
- mod_rewrite – podpora přepisování URL

- mod_ssl – zajišťuje komunikaci přes SSL a TLS
- a další moduly [5].

2.3.2 Skriptovací jazyk PHP

Pro realizaci projektu diplomové práce byl zvolen pro svou rozšířenost a multiplatformovost skriptovací jazyk PHP (*Hypertext Preprocessor*). PHP je skriptovací jazyk určený pro vývoj dynamických webových stránek a aplikací. Jeho počátky sahají do roku 1994, kdy se programátor Rasmus Lerdorf rozhodl vytvořit počítadlo přístupů na svých webových stránkách. Původně ho napsal v Perlu, ale po čase zjistil, že tato aplikace příliš zatěžuje server. Následně byla sada těchto skriptů vydána pod názvem *Personal Home Page* zkratka PHP [6]. Další vývoj PHP převzal tým programátorů na Technion IIT, přepsali parser v roce 1997 a položili tak základ PHP 3. Začalo veřejné testování a k oficiálnímu vydání PHP 3 došlo v červnu 1998. V současné době je k dispozici verze PHP 5, která nabízí nečekané možnosti o proti předchozím verzím. V PHP 5 je již plně rozvinuto objektově orientované programování a lze tedy plně využívat výhod této programátorské techniky. Umožňuje psát svůj kód v html souborech, kde před odesláním ze serveru dochází ke zpracování kódu. Nebo se nabízí možnost psaní kódu v souborech s příponou php, který vždy vyhodnotí server a zpracovaný zašle k prohlížeči (u tohoto způsobu často dochází ke generování html). V PHP 5 lze díky různým šablonovým systémům zcela oddělit logický kód od zobrazovacího.

Výhody PHP:

- PHP je relativně jednoduché na pochopení a osvojení
- Syntax podobná jazyku C
- Podpora velké řady technologií, formátů a standardů
- Projekt s otevřeným kódem a s rozsáhlou podporou komunity
- Velká podpora ze strany webového serveru Apache
- Snadná komunikace s databázovými servery
- PHP je multiplatformní a díky tomu jej lze provozovat na většině webových serverů
- PHP podporuje velké množství poskytovatelů webhostingových služeb [7]

PHP má i nevýhody:

- Je to interpretovaný a ne kompilovaný jazyk, to znamená, že kdokoli, kdo má přístup na server může prohlížet vámi vytvořené skripty
- Ve standardní distribuci chybí ladící nástroj
- Po zpracování požadavku neudržuje kontext aplikace, dochází vždy k znovu vytvoření

2.3.3 XML

XML je zkratka z anglického *eXtesible Markup Language*, jedná se o rozšiřitelný značkovací jazyk. XML můžeme označit za tzv. metajazyk (nadřazený značkovací jazyk), v rámci něhož je možné vytvářet další vlastní jazyky definované pomocí DTP popisu (příklad takového jazyku je XHTML).

Vyznačuje se tím, že neobsahuje žádné striktně stanovené značky, kdokoli si může vytvořit vlastní značky (např. <auto></auto>). Díky této vlastnosti můžeme definovat přesnou strukturu každého XML dokumentu podle aktuálních potřeb. XML by se dal definovat jako určitý předěl mezi databázovou strukturou a textovým dokumentem. Je pro něj charakteristická naprostá absence informací o způsobu zobrazovaní, tím dochází k úplnému oddělení formy od obsahu. Pro svou flexibilitu umožňuje XML vlastní volbu zobrazení v každé aplikaci, které s XML dokumentem pracují. Používán je převážně ke snadné výměně dat, a nebo ke komunikaci nezávislé na konkrétní aplikaci či platformě. Z výše zmíněného tedy vyplývá, že jednou z hlavních výhod XML je nezávislost, standardizace, poměrně malá velikost a podpora národních

kódování [8]. Každý XML dokument je uveden tzv. *root tagu*, dále se skládá z *tagů*, které mohou obsahovat další vnořené tagy nebo text, dále každý s *tagů* může obsahovat atributy (určité vlastnosti).

2.3.4 HTML

Tato zkratka znamená *HyperText Markup Language*, jedná se o značkovací jazyk pro hypertext. HTML se používá pro vývoj stránek v systému World Wide Web (www), který umožňuje publikovat dokumenty na internetu. HTML se vyvinul z rozsáhlého univerzitního jazyku SGML (*Standard Generalized Markup Language*).

Počátky HTML se datují do roku 1989, kdy Tim Berners-Lee a Robert Caillau, kteří pracovali na propojení informačního systému CERN, potřebovali jednodušší nástroj pro zveřejňování a šíření informací. Z tohoto důvodu bylo v roce 1990 navrženo HTML a protokol pro jeho přenos v počítačové síti HTTP. V roce 1991 byl zprovozněn web CERN, následně byli osloveni Marca Andreessena a Erica Bina k vývoji prohlížeče Mosaaic, ten vznikl v 1993 a sklidil obrovský úspěch, jednalo se totiž o první prohlížeč s grafickým uživatelským rozhraním [9]. Od této doby se HTML neustále vyvíjí a vznikají nové a nové verze. Kód HTML je tvořen vesměs párovými značkami (syntaxe je vždy uvedena znakem <html>). Mezi značky se uzavírají části textu a tím se určuje význam (sémantika) obsaženého textu. Názvy jednotlivých značek se uzavírají mezi úhlové závorky. Jak již bylo zmíněno, kód začíná tzv. otevíracím elementem (Tag), následuje libovolný text a uzavírá se vesměs ukončovací značkou (tzv. ukončovacím elementem). Z důvodu velkého rozšíření HTML musely vzniknout určité standardy. Za tímto účelem vzniklo konsorcium W3C, která vytváří a stará se o standardy HTML.

Dokument jazyka HTML má předepsanou strukturu:

- Deklarace DTD – je uvedena direktivou *<!DOCTYPE*
- Kořenový element – element html (značka *<html></html>*)
- Hlavička elementu – obsahuje metadata, která se vztahují k celému dokumentu (definují např. název dokumentu, jazyk, kódování, klíčová slova, ...), značka *<head></head>*
- Tělo dokumentu – obsahuje vlastní text dokumentu, vymezují ho značky *<body></body>*[9]

2.3.5 CSS

Zkratkou CSS se označují kaskádové styly (*Cascading Style Sheets*), aneb jazyk pro popis způsobu zobrazení stránek napsaných v jazyce HTML. Jazyk byl navržen standardizační organizací W3C. Účelem jazyka je umožnit návrhářům oddělit vzhled dokumentu od jeho struktury a obsahu. Původně tuto roli měl zastávat jazyk HTML, ale vlivem nedostatečných standardů a nátlaku výrobců prohlížečů se vyvinul jinak. Pomocí CSS můžeme ovlivňovat vzhled HTML souboru a to např. barvu, rozmístnění prvků, velikost a styl písma atd. Soubor kde jsou umístněny kaskádové styly má příponu *css* a sám osobě je nepoužitelný, musí být propojený s HTML souborem, který ovlivňuje. I když CSS může libovolně měnit vzhled HTML souboru, je třeba dodržovat standardy HTML, protože kaskádové styly mohou být v prohlížeči vypnuty a pak nedojde ke korektnímu zobrazení [10]. Vlivem různých prohlížečů může dojít k tomu, že některé CSS styly nebudou podporovány a nebudou se zobrazovat korektně, proto je někdy nutné navrhnout CSS styly pro různé prohlížeče.

Výhody CSS:

- Rozsáhlé množství formátovaní (nabízí více než HTML)
- Jednodušší údržba webových stránek (chceme-li změnit vzhled, nemusíme hledat prvky v html, ale měnit pouze vlastnosti prvku v css)
- Oddělení struktury a stylu

- Cachování stylů nabízí zrychlení načtení webových stránek
- CSS vlastnosti můžeme dynamicky měnit pomocí JavaScriptu
- Pomocí různých stylů můžeme docílit odlišných vzhledů pro jednotlivá výstupní zařízení (tisk, mobil, PDA, ...)

2.3.6 JavaScript

JavaScript je skriptovací jazyk založený na jazyku JAVA. Může být zpracováván na straně klienta (v prohlížeči na PC), a nebo na straně serveru. Při zpracovávání na straně klienta se program odesílá se stránkou na klienta (do prohlížeče) a teprve zde se provede. Protikladem klientských skriptů jsou skripty serverové, ty se zpracovávají na serveru a odesílá se jen výsledek [11]. Pomocí skriptovacího jazyka lze přistupovat k jednotlivým prvkům stránky, např. k prvku formuláře, k obrázkům, odkazům atd. Tyto prvky můžeme pomocí skriptu měnit, a nebo ovlivňovat jejich vlastnosti. JavaScript reaguje na různé události (even handling), které vyvolává běh programu např. na kliknutí myši, na změnu, na refresh a další. JavaScript má i několik omezení, vyplývajících z toho, k čemu je určen: nelze pomocí něj vykonávat diskové operace (kromě *cookies*), protože málokdo by ocenil, kdyby kdokoli mohl pomocí webové stránky ovlivňovat soubory na jeho disku. Další omezení může plynout ze strany prohlížeče, který nemusí tento druh skriptu podporovat nebo dokonce může mít JavaScript zakázaný. Je třeba s touto možností počítat a zajistit funkčnost stránek i bez podpory JavaScriptu.

Charakteristika jazyka:

- Interpretovací – nemusí se kompilovat
- Objektový – využívá objektů prohlížeče a zabudovaných objektů
- Závislý na prohlížeči – funguje ve většině prohlížečů
- Case sensitivní – záleží na velikosti písmen v kódu
- Syntaxí podobný jazykům C a Java [12]

2.4 Databázová vrstva

V aplikaci s třívrstvou architekturou zajišťuje databázová vrstva veškerou správu dat. K základním úkonům správy dat patří jejich ukládání a načítání, a také správa aktualizací. Databázová vrstva musí umožňovat současný, neboli paralelní přístup více než jednoho procesu prostřední vrstvy, poskytovat zabezpečení, zajišťovat integritu dat a provádět různé podpůrné operace, jako je například zálohování dat. V řadě webových databázových aplikací zajišťuje tyto služby relační databázový systém a data jsou uložena v relační databázi.

Pro správu a úložiště dat jsem zvolil databázový systém MySQL, pro účel desktopové aplikace je určena databáze PostgreSQL.

2.4.1 Databáze MySQL

MySQL (*My Structure Query Language*) je databázový systém vytvořený firmou MySQL AB, které je v současné době vlastnictvím Sun Microsystems (dceřiná společnost Oracle Corporation). MySQL je průkopníkem tzv. dvojího licencování − je dostupné jak v bezplatné licenci GPL, tak pod komerční placenou licencí.

MySQL je multiplatformní databáze, pro svou snadnou implementaci lze instalovat na Linux, MS Windows a další operační systémy. Velmi častou a oblíbenou kombinací je instalace ve spojení Linux, MySQL, PHP a Apache, někdy označováno jako LAMP. MySQL se masivně rozšířilo právě díky bezplatné licenci a stoupající potřebě dynamických webových stránek s uloženým obsahem. Zmiňovaný databázový systém byl od počátku optimalizován na rychlost, a to i za cenu zjednodušení − jednoduchý způsob zálohování, donedávna absence *view*, *triggerů* a uložených procesů. Tyto nástroje začínají být v posledních verzích doplňovány [13]. Oblíbenost

a rozšířenost MySQL je rovněž díky jeho rychlosti − při optimálním množství dat je rychlejší než jeho robustnější konkurenti. Avšak rychlost klesá s rostoucím objemem dat např. PostgreSQL je o něco pomalejší při zpracování dotazu ale s rostoucím objemem dat je jeho rychlost stále stejná. Komunikace s databázovým strojem probíhá pomocí jazyka SQL, podobně jako u ostatních SQL databází se jedná o určitou odnož tohoto jazyka (rozšíření). V MySQL lze přistupovat pomocí ad-hoc dotazů, navíc umožňuje transakční zpracování, ale pouze u určitého typu tabulek a to InnoDB.

Výhody MySQL:

- Bezplatná licence
- Optimalizován na rychlost
- Velká rozšířenost v hostingových programech
- Snadné propojení s PHP – existence knihoven na komunikaci s MySQL
- Snadná administrace přes PHPMyAdmin (PHP klient pro správu)
- MySQL je opensource a lze upravovat jeho kód

2.4.2 Databázový systém PostgreSQL

Postgres je objektově relační databázový systém (ORDBMS), vydávaný pod licencí BSD, jedná se tedy o open source software, a jak je tomu u tohoto software zvykem, na vývoji se podílí globální komunita vývojařů a firem. Postgres je primárně vyvíjen pro unixové platformy, existuje však i pro Windows. Postgres prošel dlouhým vývojem, jeho počátky se datují od roku 1982, na Berkeleyho univerzitě tento projekt vedl Michael Stonebraker. Název tohoto projektu byl původně *Ingres*. V roce 1994 byl systém rozšířen o procesor jazyk SQL, který nahradil původní Quel.

Postgres používá dotazovací jazyk SQL pro výběr a modifikaci dat. Data jsou reprezentována jako množina tabulek spojená pomocí cizích klíčů [14]. Postgres umožňuje konverzi ze světa relačních databází do objektově orientovaného programování, to způsobuje problémy rozdílné organizace dat mezi těmito způsoby. Dále umožňuje důležitou techniku objektově orientovaného programování a to je dědičnost [14].

Postgres má i vlastní odnož programovacího jazyka PL/pgSQL, jedná se o jazyk pro vývoj uložených procedur. Syntaxe PL/pgSQL a PL/SQL si je hodně podobná, zásadní odlišnosti jsou pouze v implementaci. PL/pgSQL je velice jednoduchým interpretem abstraktního syntaktického stromu, který běží ve stejném procesu, ve kterém zároveň probíhá zpracování SQL. Je úzce integrován s Postgresem, proto má zanedbatelnou režii [15].

Výhody Postgres:

- Zvládá bez problému velké množství dat při zachování výkonu (narozdíl od MySQL)
- Nabízí rozsáhlé možnosti jazyka PL/pgSQL
- Do databáze lze ukládat velké soubory, obrázky a více
- Existence *triggerů*, *view* atd.
- Jedná se o robustní databázový systém (podobá se Oracle)
- Má vynikající vývojovou komunitu (velká a silná podpora)

3 GENEROVANÝ KÓD A GENERÁTOR KÓDU

V této kapitole bude objasněna výhoda generovaného kódu a v jakých oblastech diplomového projektu byla tato metoda uplatněna. Rovněž zde představím nástroj vytvořený pro generování. Budou zde popsány vstupy a výstupy nástroje pro generování.

3.1 Generovaný kód

Generování kódu je proces, při kterém se ze zadaných vstupních dat (struktury) a obrazu (šablony) výstupu sestaví plnohodnotný výkonný kód. Při tvorbě diplomové práce využívám generování kódu ve vybraných, předem definovaných oblastech. Pro generování není použit sofistikovaný evoluční algoritmus, který by sám reagoval na určité nepřesnosti a chyby. Využívá se vstupních dat a následně obrazu, na základě kterého budou vstupní data přetransformována na výstupní kód. Zjednodušeně řečeno, pomocí vstupů je striktně vygenerován výstup. Je-li v některém ze vstupů chyba, i výstupní kód obsahuje chyby a není zcela funkční. Generování, které je využíváno v rámci diplomové práce se dá využít pouze v oblastech, kde se vyskytuje velké množství opakujícího se kódu, a které se liší pouze v drobných nuancích, jako např. názvu vlastností, tříd, sloupů a tabulek a v tomto případě je výkonová logika stejná. Další možností je generovat kód, který se mění v určitých závislostech a tyto závislosti lze odchytit pomocí vstupních šablon a reagovat na ně daným způsobem. Příkladem oblasti typické pro využití generování kódu je například třída *bussines object,* tyto třídy mají stejnou strukturu, liší se pouze v určitých vlastnostech, a to v počtu a názvu polí. Pomocí procesu generování kódu lze vytvořit stovky řádků kódu, stačí daný kód napsat pouze jednou pro určitý případ. Následně je třeba daný kód otestovat, zda je plně funkční a poté zadat vstupní data, na která se vygeneruje funkční kód.

Výhody generovaného kódu:

- Stačí napsat kód pouze jednou
- Během krátkého okamžiku získáme stovky řádků kódu
- Pomocí generování se snižuje chybovost kódu, při opakovaném psaní kódu lze zavléct chybu
- Velká úspora času oproti motoricky přepisovanému kódu
- Pokud je potřeba kód upravit, upraví se pouze jednou a provede se nové generování
- Částečně zlevňuje vývoj
- Náročnost se vyskytuje pouze na začátku, v době kdy se definují vstupy a výstupy, posléze je údržba velmi snadná

Při tvorbě diplomové práce byl využit nástroj generování kódu u oblastí, které splnily podmínky pro generování. Jedná se o následující oblasti:

- Synchronizace dat (s touto oblastí budeme podrobně seznámeni v následující kapitole)
- Oblast *business object* a část tříd výkonové části *Database Access Object*
- Databázová struktura pro webové rozhraní (MySQL)

3.2 Nástroj Generátor kódu

Pro generování kódu byla vyvinuta aplikace pod platformou .NET a to z důvodu podpory XSLT (*Extensible Stylesheet Language Transformations*). Nabízí se možnost využít knihovny a pomocí platformy snadno vzniklou transformaci aplikovat. Je zde podporován velmi výkonný procesor XSLT, který je alfou a omegou samotné transformace. Tento procesor transformaci realizuje. XSLT přesně vyhovuje účelu našeho pojetí generování kódu. Základním principem XSLT transformace je vygenerování třetího dokumentu (nebo obecně souboru) a to na základě souboru se zdrojovými daty a šabloně).

Soubory nutné pro samotnou XSLT transformaci:

1. První soubor obsahuje zdrojová data, která budou transformována, tento soubor je formátu XML. XML je vstupem pro procesor, který na výstupu produkuje soubor předepsaného formátu.
2. Druhý soubor obsahuje vzorec pro transformaci. Zahrnuje předpis, podle kterého dochází k transformaci na výstupní soubor. Daný předpis je zapisován v jazyce XSL.

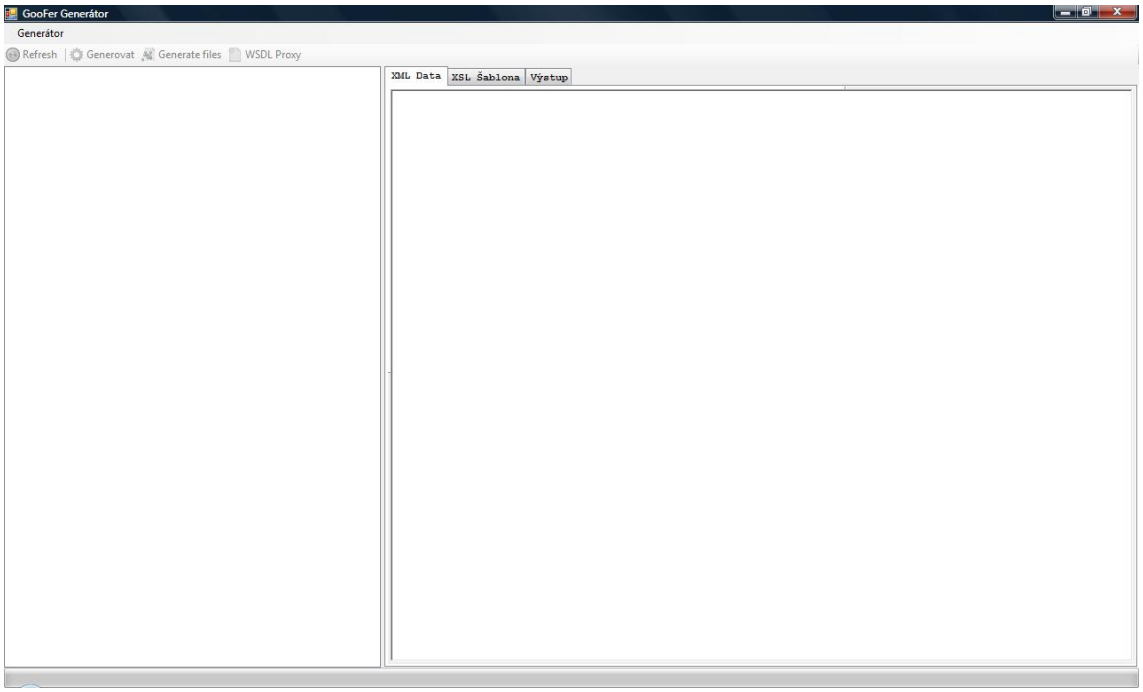
Výhoda XSLT transformace spočívá v tom, že na základě jednoho XML dokumentu můžeme vygenerovat několik různých druhu dokumentů. Mezi nejobvyklejší transformace patří:

- XML na HTML
- XML na jiné XML
- XML na prostý text (*txt* soubor)

Pro vyvinutý generátor se využívá generování prostého textu, pouze s rozdílem v příponě souboru. Soubor nemá příponu *.txt*, jelikož je výstup implementován pro webovou aplikaci jazyka PHP, má soubor s prostým textem příponu *php*.

3.3 Funkčnost aplikace generátoru kódu

Generátor kódu je desktopová aplikace, do níž je implementována XSLT transformace, která tvoří základní funkčnost generátoru. Před prvním spuštěním aplikace generátoru je třeba projít krátkým *Setup Wizardem*, který provede inicializaci aplikace. Při spuštění aplikace se otevře průvodní okno, viz obr. 4.



Obr. 4 Aplikace Generátoru po spuštění

Spuštěná aplikace není připojená, a proto se v ní nezobrazuje žádný výsledek. Na průvodní obrazovce lze vidět vzhled aplikace. Její okno je rozděleno do dvou hlavních panelů, které budou později podrobně popsány. V aplikaci se rovněž nachází důležitá komponenta nabídky menu označená *Generátor*. Nabídka *menu* nabízí potřebné komponenty pro inicializaci a funkcionality generátoru.

V menu se nachází následující položky:

- Připojení k DB
- Odpojení od DB
- Otevření souboru XML
- Otevření souboru XSL
- Uložení souboru XML
- Konec

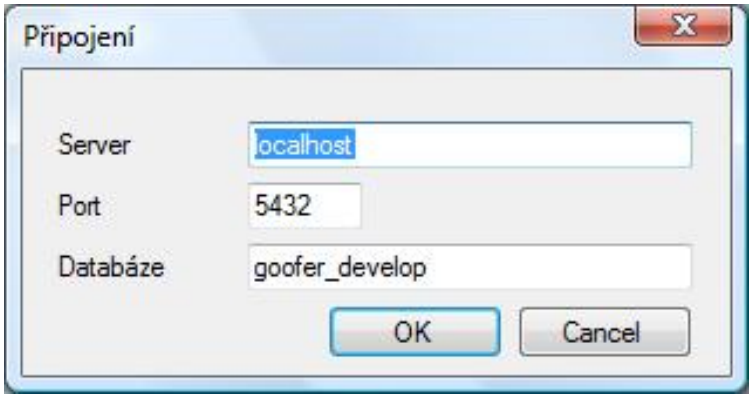
3.4 Funkcionality aplikace generátoru a nabídky menu

Pro přiblížení jednotlivých funkcionalit aplikace generátoru budou jednotlivé podkapitoly věnovány nabídkám v komponentě menu. Ve dvou podkapitolách se dotkneme problematiky XML a XSL, budou zde uvedeny příklady struktur a kódů návrhů souborů nezbytných pro generování.

3.4.1 Připojení k DB

Pod touto nabídkou je implementován konektor k databázi, která je zdrojem dat pro generátor. Na základě reprezentovaných dat v databázi vzniká zdrojový soubor pro transformaci. Implementovaný konektor je omezen pouze pro připojení PostGre databází. Důvod omezení je dán databází primárního systému *Goofer*. Veškerá data a informace jsou uvedena v tomto databázovém systému a je vždy považován za výchozí a primární.

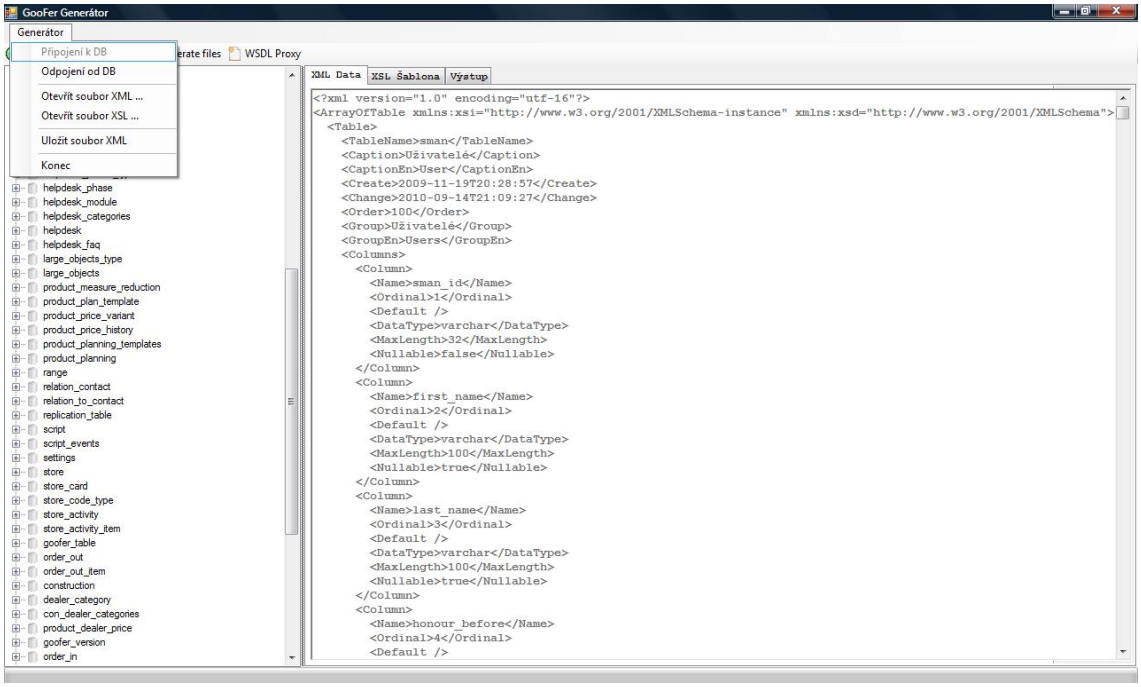
Při potvrzení nabídky pro připojení k databázi je uživatel dotázán na údaje, které jsou potřebné jako vstupní parametry pro konektor. Uživatel zadá potřebné údaje do dialogového okna na obr. 5.



Obr.5 Dialogové okno pro připojení k databázi

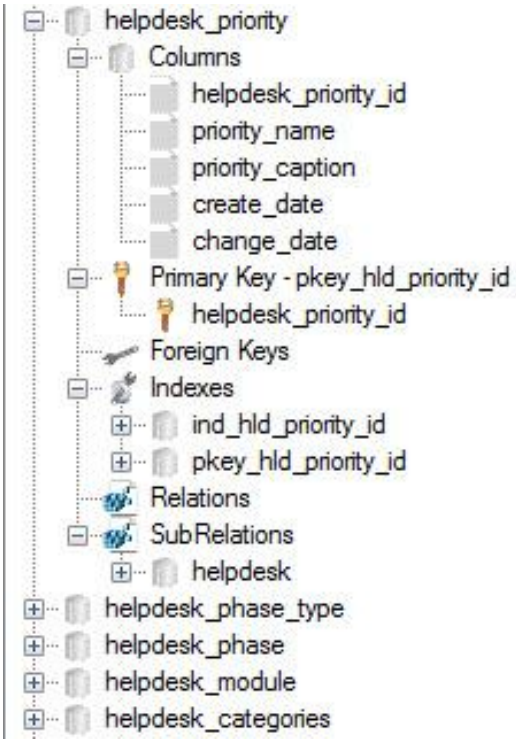
V aktuálně otevřeném dialogovém okně se provede zadání následujících parametrů: první parametr je adresace databázového serveru, v našem případě se jedná o připojení k databázovému serveru, umístěného na lokálním počítači. Generátor je defaultně určen pro připojení k lokálnímu databázovému systému na počítači, kde se vyskytuje instalace softwaru *Goofer* a tím je zajištěna existence zdrojové databáze. Dalším požadovaným parametrem je port pro komunikaci s Postgres serverem po síti. Tento port se volí při instalaci samotného databázového serveru. Posledním parametrem je zadání zdrojové databáze, nad kterou má být provedeno generování. Zadané informace zpracuje konektor, který provede připojení na databázový server a zajišťuje komunikaci mezi aplikací a serverem. Aplikace se hlásí do databáze podobně jako aplikace *Goofer*, tzn. pod administrátorským uživatelem, zbylé přihlašovací vlastnosti přebírá z modulu. Hlavní třídou zde je *DBloader*. Zde dochází k vytvoření připojení, ale i k plnění *business objectů* např. *Table, Column, Key*, atd. V okamžiku, kdy korektně proběhne připojení k databázovému serveru, potažmo k zadané databázi, dojde k

inicializaci aplikace generátoru po obou stranách již zmíněných panelů. Připojený generátor je vyobrazen na obr. 6.



Obr. 6 Připojená a zinicializovaná aplikace generátoru

Po přihlášení se levý panel okna aplikace zinicializuje a vytvoří se stromová struktura tabulek obsažených ve zdrojové databázi. Levý panel aplikace tvoří komponenta *TreeView* splňující požadavky na zobrazení stromové struktury připojené databáze. Ukázka rozvinuté stromové struktury je vyobrazena na obr. 7.



Obr. 7 Detailní pohled na levý panel s využitím komponenty TreeView

V první úrovni stromové struktury *TreeView* se vypisují názvy všech existujících tabulek v připojené databázi. Po rozvinutí další úrovně stromu se zobrazí další podvětve, které reprezentují jednotlivé vlastnosti tabulek. K nejdůležitějším vlastnostem patří uzel *Columns*, ten obsahuje informace o sloupcích, které tabulku tvoří. Do přehledu stromu jsou zahrnuty i údaje k primárním a cizím klíčům. Tyto údaje jsou nezbytné pro relační vazby mezi tabulkami. K dalším vlastnostem patří i přehled indexovaných polí v určité tabulce. V neposlední řadě jsou zde uvedeny informace o relacích mezi jednotlivými tabulkami, a to vazby nadřazené a podřazené. Stromová nabídka reprezentující přehled datové struktury je spíše informativní a slouží k snadnému prohlížení vstupních dat z připojené databáze. Informace v *TreeView* mají totožnou strukturu jako XML. Na základě těchto informací se generuje XML kód nad datovou strukturou pomocí třídy *XMLbuilder*. Třída *XMLbuilder* pomocí údajů z připojené databáze vytvoří XML strukturu a výsledek se zobrazí v pravém panelu okna. Pravý panel je tvořen komponentou *RichTextBox* a je tímto inicializován. Konponenta *RichTextBox* je začleněna do tří záložek (*TabControl*). Záložka, do které třída vloží výslednou XML strukturu, se nazývá *XML Data*. O výsledné XML struktuře a jejím významu se více zmíním v následující podkapitole.

3.4.2 XML struktura a její popis

Tato kapitola obsahuje ukázky fragmentů XML struktury, ke každé části bude uveden stručný popis. XML kód v pravém panelu slouží přímo jako vstup pro XSLT transformaci. Dále se budeme věnovat samotné struktuře XML v příkladech.

Jako první je třeba představit nerozvinutou XML strukturu, která reprezentuje pouze základní elementy a jejich vnořené elementy nebo atributy jsou skryté. Nerozvinutá struktura je znázorněna na obr. 8.

```
<?xml version="1.0" encoding="utf-16"?>
<ArrayOfTable xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<Table>
   <TableName>helpdesk_priority</TableName>
   <Caption>Help desk priorita</Caption>
   <CaptionEn>Help desk priority</CaptionEn>
   <Create>2011-12-19T20:28:57</Create>
   <Change>2011-12-29T20:15:04</Change>
   <Order>2700</Order>
   <Group>Help desk</Group>
   <GroupEn>Help desk</GroupEn>
   <Columns>
   ...
   </Columns>

   <PrimaryKeys>
   ...
   </PrimaryKeys>

   <ForeignKeys />
   <Indexes>
   ...
   </Indexes>
   <Relations>
   ...
   <Relations>
   <SubRelations>
   </SubRelations>
</Table>

<Table>
   ...
</Table>
</ArrayOfTable>
```

Obr. 8 Struktura XML dokumentu bez vnořených elementů a atributů

Každý XML soubor musí být uvozen hlavičku, kde je uvedena verze jazyka a dále použité kódování (v našem případě se používá kódování *utf-16*). Každý XML dokument je tvořen elementy, ty mohou být navzájem vnořené, a tím dle potřeby zachycovat strukturu informaci. XML kód začíná tzv. „kořenový elementem“, který uzavírá celý dokument. V uvedené struktuře je kořenovým elementem *ArrayOfTable*.

Element *Table* je hlavním elementem, který reprezentuje objekty tabulek a obsahuje další vnořené elementy a atributy představující jednotlivé vlastnosti tabulek. Tento element je opakující se, jeho výskyt ve struktuře je dán počtem tabulek ve zdrojové databázi. Následují elementy, vnořené pod element *Table*:

Element *TableName* je vnořen pod *Table*, uchovává se v něm název tabulky pro daný uzel. Patří k nejpoužívanějším elementům.

Element *Caption* nese informace o popisu tabulku. Je-li tento popis v databázi vytvořen, přenese se do tohoto elementu. Tento element se vyskytuje i v další jazykové mutaci.

Element *Group* je určen pro sdružování tabulek do určitých skupin, např. samotný Postgres má skupinu *Sys*, *Admin*, atd. Rovněž název skupiny má i jinou jazykovou variantu.

Elementy *Create* a *Change* jsou ve struktuře obsaženy, aby bylo zřejmé, kdy tabulka byla vytvořena a v jaký okamžik došlo k její poslední modifikaci.

Element *Columns* je přímo podřazený hlavnímu elementu *Table* a sám je ještě složen z vlastní podstruktury. Nese více opakujících se vlastnosti. Reprezentuje množinu polí dané tabulky. Podstruktura *Columns* je zachycena na obr. 9.

```
<Columns>
   <Column>
     <Name>helpdesk_priority_id</Name>
     <Ordinal>1</Ordinal>
     <Default />
     <DataType>varchar</DataType>
     <MaxLength>32</MaxLength>
     <Nullable>false</Nullable>
   </Column>
     ...
   <Column>
    ...
   </Column>
</Columns>
```

Obr. 9 Struktura elementu Columns

Element *Column* je vnořený do struktury *Columns*, jedná se o opakující se element, nachází se ve struktuře tolikrát, kolik má daná tabulku sloupců. Tento element nese informace o sloupcích tabulky. Dále si představíme elementy, které jsou vnořené do elementu *Column*.

Element *Name* vnořený do *Column* vyjadřuje název sloupce tabulky. Element *Ordinal* udává pořadí sloupce v tabulce, tedy na jaké pozici se vyskytuje. Element *Default* (je-li vyplněn) stanovuje výchozí hodnotu, které sloupec nabývá, není-li tato zadána uživatelem. Element *DataType* popisuje datový typ sloupce v tabulce, nabývá hodnot názvů datových typů databáze Postgres. Element *MaxLength* určuje maximální velikost sloupce např. u typu *varchar(20)*. Element NullAble nese údaj o tom, zda sloupec v tabulce může nést *null* hodnotu nebo musí být naplněn hodnotou. Všechny výše uvedené elementy v tomto odstavci jsou vnořeny do struktury *Column*.

Element *PrimaryKeys* a *ForeignKeys* patří pod přímo vnořené elementy *Table*. Tyto elementy nesou informace o sloupcích tabulky, kde mají roli primárních a cizích klíčů. Elementy striktně definují vazby mezi jednotlivými tabulkami. Obsahují další vnořené informace o sloupcích. K vnořeným elementů patří *Name*, který nese označení primárního, popřípadě cizího klíče. Dalším vnořeným elementem je již známá struktura *Columns (*i se svými vnořenými elementy). Počet elementů vnořených do *Column* odpovídá počtu cizích a primárních klíčů, které jsou v tabulce obsaženy. Posledním elementem těchto dvou elementů je

IsUnique, který udává, zda jsou primární a cizí klíče unikátní. Vnořený element *PrimaryKeys* je znázorněn na obr. 10. Element *ForeignKeys* je téměř totožný jako element *PrimaryKeys*.

```
<PrimaryKeys>
 <PrimaryKey>
 <Name>pkey_hld_priority_id</Name>
 <Columns>
   <Column>
     <Name>helpdesk_priority_id</Name>
     <Ordinal>1</Ordinal>
     <Default />
     <DataType>varchar</DataType>
     <MaxLength>32</MaxLength>
     <Nullable>false</Nullable>
   </Column>
 </Columns>
 <IsUnique>false</IsUnique>
 </PrimaryKey>
</PrimaryKeys>
```

Obr. 10 Vnořený element PrimaryKey

Element *Indexes* patří mezi podřazené elementy *Table* a doplňuje další vlastnosti tabulky. Pokud jsou na určitých sloupcích v tabulce navrženy a realizovány indexy budou zobrazeny v tomto elementu. Indexy jsou zde zaznamenány podobně jako cizí klíče, mezi elementy *Indexes* je obsažen opakující se element *Key*. Ten se (podobně jako u elementu primárních klíčů) opakuje pro všechny indexy tabulky. Struktura je pojmenováná pomocí elementu *Name*, který nese název indexu v databázi. Navíc obsahuje již známou strukturu pro sloupce *Column*, včetně všech jeho vlastností. Ukázka struktury elementu *Indexes* je znázorněna na obr. 11.

```
 <Indexes>
 <Key>
   <Name>ind_hld_priority_id</Name>
   <Columns>
     <Column>
       <Name>priority_name</Name>
       <Ordinal>2</Ordinal>
       <Default />
       <DataType>varchar</DataType>
       <MaxLength>20</MaxLength>
       <Nullable>false</Nullable>
     </Column>
   </Columns>
   <IsUnique>true</IsUnique>
 </Key>
 ...
 </Indexes>
```

Obr. 11 Ukázka detailu elementu Indexes

Posledními elementy ve struktuře jsou *Relations* a *SubRelations*, které obsahují údaje o vzájemných vazbách mezi jednotlivými tabulkami. Rozdílnost mezi elementy je následující: element *Relations* uchovává vazbu k tabulkám, které jsou v daném uzlu dané tabulce nadřazené (odkazuje na tabulky přes cizí klíče), ve vazbě se postupuje zdola nahoru. Element *SubRelations* eviduje vazbu na tabulky podřazené, po vazbě se postupuje svrchu dolů. Obsahuje opakující se element *Relation*, který zahrnuje jednotlivé vazby v tabulkách. Protože tabulka může mít více vazeb, je vazba zaobalena do opakujících se elementů *Relation*. V elementu jsou vnořeny další elementy. A to třeba *TableName*, který obsahuje název tabulky, na kterou vazba směřuje. Podstatným vnořeným elementem je *Columns*, který se liší od struktury zmíněné výše. Obsahuje element *ColumnRelation*, který se může opakovat a to např. z důvodu složených klíčů. Element *ColumnRelation* má i další vnořené vlastnosti. Vnořené elementy *PrimaryColumnName* a *ForeignColumnName* obsahují názvy sloupců reprezentujících cizí a primární klíče a provázanost těchto tabulek. Poslední elementy jsou potřebné pro zachování datové integrity a jedná se o *UpdateRule* a *DeleteRule*. Pro každou relaci jsou uvedena integritní

omezení např. *restrict, cascade, setnull*.

```
<Relations />
  <Relation>
    <TableName>helpdesk</TableName>
    <Columns>
      <ColumnRelation>
        <PrimaryColumnName>helpdesk_priority_id</PrimaryColumnName>
        <ForeignColumnName>helpdesk_priority_id</ForeignColumnName>
        <UpdateRule>CASCADE</UpdateRule>
        <DeleteRule>SET NULL</DeleteRule>
      </ColumnRelation>
    </Columns>
  </Relation>
<Relations />
```

Obr. 12 Rozvinutý element Relations

3.4.3 Otevření XSL šablony a popis šablony

Otevření souboru šablony XSL provedeme v nabídce *menu*, po potvrzení nabídky *Otevření souboru XSL* se otevře okno pro vyhledání souboru s příponou *xsl*. Soubor se šablonou se otevře do druhé záložky „XSL šablona“ v pravém panelu okna. Panel s XSL šablonou patří k velmi důležitému dokumentu pro generátor, na základě této šablony se generuje samotný kód a tvoří formát a vzhled výstupního dokumentu.

Podle formátu šablony je zřejmé, že uvnitř souboru musí být obsažen XSL kód. Proto je nezbytné osvojit si pravidla a syntax tohoto jazyka pro psaní šablon. Pro každý konkrétní případ generování kódu musela být napsána a vytvořena zvláštní šablona. Oblastí, kde je využíváno generování kódu je oblast synchronizace, dále pro přenos dat, pro vytvoření datové struktury pro MySQL databázy a nakonec pro *Database Access Object* vrstvu. Pro ukázku a popis XSL šablony jsem zvolil jednodušší šablonu pro generování datové struktury pro MySQL (obr. 13).

Šablona je uvozena hlavičkou, podobně jak u XML dokument. Hlavička obsahuje informace o verzi jazyka, kódování výstupu. Dále je třeba dostat se k datům v XML souboru, abychom s nimi mohli pracovat. K tomuto úkonu nám slouží příkaz *CDATA*. *CDATA* je mechanismus sloužící k přímému vložení dat obsahujících znaky vyhrazené pro XML syntaxi: tj. <>\&"' .Tyto znaky je nutné mimo sekvenci *CDATA* zapisovat pomocí HTML entit např. *< > & " a '*.

Postup tvorby šablon byl následující: nejprve bylo nezbytné napsat daný kód a otestovat jeho funkčnost, v uvedeném příkladu šablony je to kód v SQL příkazu pro vytvoření datové struktury tabulky. Napsaný kód se zkopíruje do sekvence *CDATA*, *sql* kód bude vnímán jako čistý text. Pasáže kódu, které chceme doplnit zdrojovými daty, nahradíme příkazy XSL. V případě SQL šablony pro tvorbu datové struktury se příkazy XSL nahradí (např. název tabulky, názvy sloupců). Při psaní XSL šablon bylo využito základních příkazů a sekvencí jazyka XSL. A to řídící struktury *IF*, pomocí níž se zjišťuje, zda danou hodnotu vypsat nebo ne, popřípadě zda vypsat v závislosti na datech jinou sekvenci. Sekvence *IF* byla využita např. v části, kdy se rozhoduje, jestli sloupec vůbec může být bez hodnoty. Pro části kódu, které se opakují, ale liší se jen v názvu, se využívá cyklů, a to především *<xsl:for-each select="Table">*. Pomocí smyčky *<xsl:for-each select="Table">* projdeme všechny tabulky v XML souboru. K hodnotám v určité části zanoření v souboru XML se lze dostat s pomocí příkazu *<xsl:value-of select="../../TableName"/>*, tento případ vrátí název tabulky. V atributu *select* uvádíme název entity v XML, jsme-li v určité úrovni zanoření v XML entitách, pomocí ../ se vrátíme o úroveň výše.

Při vývoji XSL šablony se nabízí možnost využívat i vlastní naprogramované funkce. Tyto funkce lze vyvolat pomocí příkazu *<xsl:value-of select="dmx:nazevFunkce(parametr)"/>*. Pro účely generátoru byly vytvořeny funkce pro konvertování datových struktur (např. převod mezi datovými typy z PostGres do MySQL), pro úpravu textu. Definice funkcí jsou uloženy v souboru *_utility.xslt*, ten musí být namapován v šabloně, ve které chceme funkce používat.

Při tvorbě samotné šablony je nutné dbát na to, aby se do kódu nepozorností nevnesla chyba. Odladění proti chybovosti se týká i samotných šablon. U šablon se musí kontrolovat chyby v samotné struktuře šablony, ale i chyby, které mohou nastat při přepisování původního kódu a mohla by tak být pozměněna funkcionalita, kód by v tomto případě nemusel pracovat korektně.

```
<xsl:import href="_utility.xslt"/>

<xsl:output method="text" encoding="utf-16"/>

<xsl:template match="ArrayOfTable">
  <xsl:variable name="version" select="'1.0.0.0'" />
  <xsl:variable name="smallcase" select="'abcdefghijklmnopqrstuvwxyz'" />
  <xsl:variable name="uppercase" select="'ABCDEFGHIJKLMNOPQRSTUVWXYZ'" />
  &lt;?xml version="1.0" encoding="utf-16"?&gt;
  &lt;ArrayOfFileObject xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"&gt; &lt;FileObject>
  &lt;![CDATA[
  <xsl:for-each select="Table">
    CREATE TABLE `<xsl:value-of select="TableName"/>` (
    <xsl:for-each select="Columns/Column">
      `<xsl:value-of select="Name"/>` <xsl:value-of
       select="dmx:GetConvertTypeForMySql(DataType,MaxLength)"/>
      <xsl:if test="Nullable='false'">
        <xsl:text> NOT</xsl:text>
      </xsl:if> NULL <xsl:value-of select="dmx:GetDefaultValueForMySql(Default)"/>
       <xsl:if test="starts-with(Default,'nextval(')">
        <xsl:text>AUTO_INCREMENT</xsl:text>
      </xsl:if>,
    </xsl:for-each>PRIMARY KEY (<xsl:for-each select="PrimaryKey/Columns/Column">
      `<xsl:value-of select="Name"/>`<xsl:if test="position()!=last()">
        <xsl:text>, </xsl:text>
      </xsl:if>
    </xsl:for-each>)
    ) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
    <xsl:for-each select="Indexes/Key">
      CREATE <xsl:if test="IsUnique='true'">
        <xsl:text>UNIQUE</xsl:text>
      </xsl:if> INDEX `<xsl:value-of select="Name"/>` ON `<xsl:value-of
      select="../../TableName"/>` (<xsl:for-each select="Columns/Column">
        <xsl:value-of select="Name"/>
        <xsl:if test="position()!=last()">
          <xsl:text>, </xsl:text>
        </xsl:if>
      </xsl:for-each>);
    </xsl:for-each>

  </xsl:for-each>
  ]]&gt;
  &lt;/Content&gt;
  &lt;/FileObject>
  &lt;/ArrayOfFileObject&gt;
</xsl:template>
```

Obr. 13 Ukázka XSL šablony pro generování DB struktury pro MySQL

3.4.4 Další funkcionality v nabídce menu generátoru

V nabídce menu se skrývají další funkcionality aplikace generátoru, o kterých zatím nebyla zmínka. Patří sem:

- Nabídka menu *Odpojení od DB* – dojde k ukončení spojení generátoru s databází a přejde se do offline stavu. Je spuštěna metoda destruktoru pro databázové připojení. I v tomto stavu lze provádět generování kódu.
- Nabídka menu *Uložení XML souboru* – XML dokument, vytvořený na základě databázové struktury, bude uložen do souboru. Název a cesta souboru se volí přes klasické WIN okno průzkumníka pro uložení. Využívá se zde metoda pro operace vytváření a uložení souboru.

- Nabídka menu – *Otevření souboru XML* – po potvrzení této nabídky se otevře klasické okno s průzkumníkem pro vyhledaní souboru s příponou *xml*. Po otevření *xml* souboru se načte XML struktura do záložky XML data do panelu v pravé části okna aplikace, což slouží jako zdroj dat pro generování offline, bez nutnosti připojení k databázi. Díky této funkcionalitě je nástroj univerzální, můžeme načíst jakoukoliv XML strukturu a na základě odpovídající šabloně tomuto zdroji můžeme vygenerovat nový dokument různého tvaru.
- Nabídka menu *Konec* – slouží pro ukončení běhu programu. Po stisknutí se uvolní všechny instance a ukončí se celá aplikace.

3.4.5 Ovládací prvky pro generování

Aplikace generátoru obsahuje lištu vytvořenou pomocí komponenty *ToolStrip*, na níž jsou umístněny jednotlivá tlačítka (*buttons*). Pod těmito tlačítky je zavěšený kód, který je při události kliku spuštěn.

Tlačítko *Refresh* zajišťuje obnovu načtení všech objektů v komponentě *TreeView*, kde jsou reprezentovány jednotlivé tabulky. Dojde-li za běhu generátoru ke změně v datové struktuře např. odebrání nebo vytvoření tabulky, pomocí této funkce se provede znovunačtení dat.

Obr. 14 Tlačítko Refresh generátoru

Potvrzením tlačítka *Generovat* se spustí hlavní funkcionalita generátoru – samotný proces generování kódu. Proběhne proces XSLT transformace, který na základě XSL šablony s odkazem na zdrojová data v XML vytvoří třetí dokument podle zadaného formátu v šabloně. Výsledek se přiřadí do komponenty záložky *Výstup* v *RichTextBox*. V záložce se objeví výstupní vygenerovaný kód, který lze přikopírovat do jakéhokoli souboru.

Obr. 15 Tlačítko Generovat

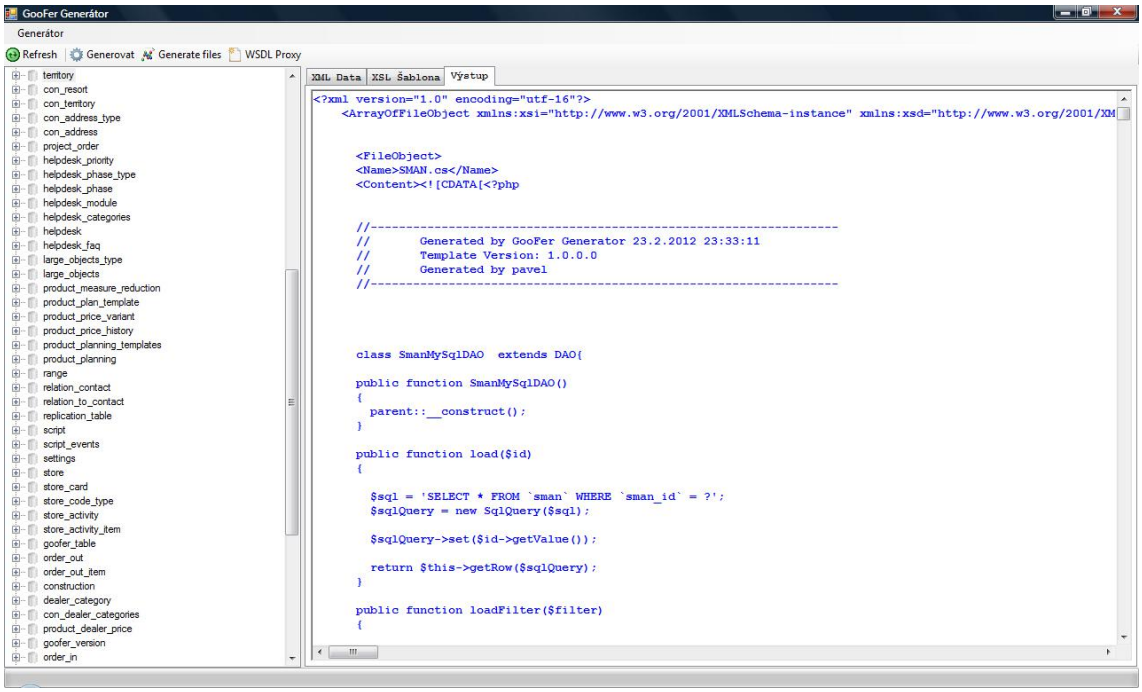
Po spuštění procesu generování dojde k inicializaci poslední záložky pravého panelu, ukázka okna aplikace s vygenerovaným kódem je na obr. 16. Vygenerovaný kód v pravém panelu je určen spíše pro vizuální kontrolu správnosti kódu, je zde ještě obsažena část XML kódu a proto je třeba výsledek uložit. Ve vytvořených souborech se vyskytuje pouze vygenerovaný kód.

V *Generate File* je implementován proces pro uložení výsledku vytvořeného generováním. Proces provede uložení do souboru podle názvu a přípony, definovaných v XSL šabloně. Podle definice v XSL šablony se provede uložení do jednoho souboru pod zadaným názvem a příponou, popřípadě do více souborů, je-li název souboru definován dynamicky a při průchodu smyčkou dojde k uložení do souboru.

Obr. 16 Tlačítko Generate files

Po potvrzení tlačítka *Generate files* je uživatel vyzván pomoci průzkumníka k zadání

cílového adresáře, kam budou soubory uloženy.



Obr. 16 Okno aplikace po spuštění procesu generování

3.4.6 Přínos nástroje pro generování

Nástroj pro generování kódu byl vyvinut za účelem urychlit vývoj jednotlivých částí aplikací. S jeho využitím se nejprve počítalo jen pro nástroj synchronizace dat a pro tvorbu datové struktury pro webové rozhraní. Při vývoji *Database Access Object* se využití tohoto nástroje samo nabízelo. Čas vynaložený na vývoj uvedeného nástroje se již téměř vrátil. Umožňuje vygenerovat tisíce řádků kódu, které by se neustále opakovaly a lišily by se pouze v určítých v parametrech. Nástroj pomáhal i při odlaďování různých částí projektů, např. když se musela měnit logika výpočtu některých funkcí. Pomocí nástroje stačilo upravit pouze šablonu a vše znovu vygenerovat. Nástroj byl od jeho vývoje už mnohokrát využit a nadále bude využíván při vydání nových verzí software, kdy se změní datová struktura a změny se musí projevit i v internetových modulech. Nástroj bude využíván při release.

Seznam šablon vyvinutých pro účel nástroje pro generování:

- *WebService.PHP.xsl* – určena pro vygenerování nástrojů pro synchronizace. Obsahuje soubory s výkonovými funkcemi samotné synchronizace. Pomocí této šablony jsou generovány i struktury a registrace samotné webové služby. Všechny tyto nástroje jsou zahrnuty v jedné šabloně.
- *DataStructure.MySql.xsl* – je určena pro vygenerování datové struktury pro MySQL. Šablona byla využita jako vzorová ukázka, protože není rozsáhlá
- *MySqlDAO.xsl* – šablona je určena pro vrstvu DAO aplikace Helpdesk, a to pro výkonovou část objektů, které obsahují metody pro načítání dat, pro vkládání.
- *DaoFactory.xsl* – šablona pro generování třídy vrstvy DAO, která obsahuje metody, které vrací instance objektů.
- *BO.xsl* – šablona pro vrstvu DAO, s jejíž pomocí se generuje struktura *Business Object*.
- *Include_class_dao.xsl* – šablona zařazena opět k vrstvě DAO, pomocí ní je prováděna implementace jednotlivých tříd do projektu.

4 SYNCHRONIZACE DAT

V této kapitole bude zmíněn řešení synchronizace dat s webovým rozhraním. Krátce se zmíníme i o jiných uvažovaných řešeních a úskalích, které nastaly při vývoji. Je zde uvedena i funkcionalita a popis jednotlivých funkcí využívaných pro synchronizaci dat.

4.1 Návrh synchronizačního nástroje

Synchronizace dat je nezbytná k výstavbě webové nástavby nad aplikací *Goofer* a konkrétně na jeho modul *Helpdesk*. Je třeba nějakým způsobem přenést data z PostGres do databáze na webovém rozhraní *MySQL*. Pro tento účel bylo nutné vytvořit nástroj, který tuto výměnu dat mezi tlustým klientem a webovým rozhraním zajistí. Synchronizační nástroj musí umět zajišťovat obousměrnou komunikaci. Pod obousměrnou komunikací je zahrnut přenos z aplikace *Goofer*, tzn.: pokud uživatel vytvoří nový požadavek, v tlustém klientovi se musí tato akce promítnout na webovém rozhraní a následně, dojde-li k založení nového požadavku na webovém rozhraní, musí být tyto informace promítnuty na tlustém klientovi. Tento synchronizační nástroj zajišťuje, aby databáze na webovém rozhraní byla přesným obrazem databáze tlustého klienta a naopak. Je zřejmá snaha vyvarovat se duplicitě jednotlivých záznamů v databázích.

4.1.1 Možnosti řešení synchronizačního nástroje

Na začátku řešení nástroje pro synchronizaci dat bylo nutné definovat způsob, jakým se bude nástroj realizovat. Uvažovalo se nad následujícími možnostmi:

1. Přenos *cvs* souboru – tento způsob by vyžadoval generování souboru formátu *csv* tj. souboru s hlavičkou, kde jsou jednotlivá data (sloupce) odděleny středníkem. Tyto soubory by se přenášely a stahovaly z FTP, obsahujícího webové moduly. Tento způsob by vyžadoval na obou stranách funkcionalitu pro tvorbu *csv* dokumentů se změnovými a novými daty. Vyžadoval by funkcionalitu, která tyto soubory zpracuje. Tento způsob jsem po zvážení zavrhl z důvodu těžkopádnosti řešení a nedostatečné flexibility. Muselo by pokaždé dojít k rozložení tohoto souboru parserem a data z něj složitě vybírat.

2. Přenos *sql* souboru – přenos *sql* souboru je velmi podobný prvnímu způsobu. Soubor by se obdobně přenášel a stahoval z FTP, s tím rozdílem, že data ze souboru by se nemusela vůbec rozkládat a vybírat. Soubor dle návrhu by obsahoval rovnou SQL příkazy typu *insert*, *update* a *delete*. Tento způsob je rychlejší, protože *sql* soubor se jen provede a změny se promítnou bez jakéhokoli dalšího zásahu. Synchronizace by se odehrávala na nejnižší úrovni prostřednictvím databáze. Ale ani tento způsob se nezdál být zcela optimálním, a proto bylo namístě hledat další způsob.

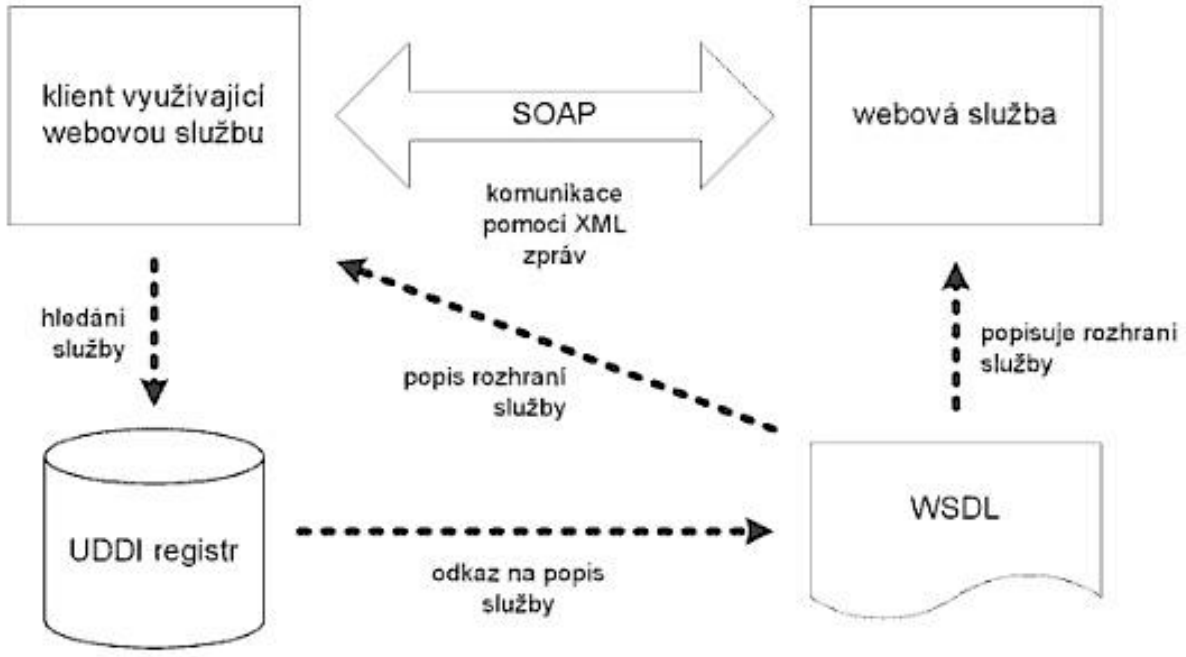
3. Komunikace a přenos pomocí webových služeb – pomocí této metody mohou mezi sebou komunikovat aplikace v různých jazycích na různých platformách, komunikace probíhá pomocí XML, což umožňuje společnou komunikaci standardním způsobem. Tento způsob je dostatečně flexibilní a rozšiřitelný, proto jsem ho zvolil pro vývoj nástroje pro nástroj synchronizace dat. Dalším důvodem výběru řešení prostřednictvím webových služeb je jejich využitelnost, třeba v oblasti technologií na mobilních telefonech.

4.1.2 Webové služby

Webová služba umožňuje komunikaci mezi aplikacemi odlišných vývojových prostředí. Komunikace je založena na nezávislých technologiích, jako je jazyk XML a přenos přes protokol HTTP. Různorodé aplikace si mezi sebou posílají XML zprávy, které přenáší dotazy a odpovědi jednotlivých aplikací. Základní struktura webových služeb je založena na třech elementárních technologiích:

1. SOAP (*Simple Object Access Protocol*) – obálka a způsob kódování dat používaných pro komunikaci
2. WSDL (*Web Services Description Language*) – formát pro popis rozhraní webové služby
3. UDDI (*Universal Description, Discovery and Integration*) – standardní mechanismus umožňující registraci a vyhledávání webových služeb [16]

Výše uvedené vlastnosti webových služeb jsou přesně ten důvod, proč byl zvolen zmiňovaný nástroj řešení synchronizace dat pro naše účely. Všechny tři výše uvedené technologie se využívají pro řešení webové synchronizační služby. Pro lepší představu funkčnosti a významu jednotlivých technologií následuje ilustrační obr. 17.



Obr. 17 Princip funkčnosti webových služeb

4.2 Funkcionality synchronizačního nástroje

Pro samotný přenos dat se využívá webová služba. V případě diplomového projektu je používán nástroj *NuSOAP*. *NuSOAP* není modul PHP, jedná se o skupinu tříd, které umožňují vytvářet a konzumovat služby *SOAP*, tyto třídy nevyžadují žádné speciální rozšíření PHP. Pomocí tříd se vystaví webová služba. V průběhu této časti jsou uvedeny příklady řešení všech technologií, potřebné pro realizaci webové služby (synchronizace dat).

Samotná webová služba je realizována v souboru *igoofer_webservice*, ukázka obr. 18. V tomto souboru jsou vloženy všechny údaje o službě. Je tu implementována třída *NuSOAP* a následně vytvořena instance této třídy. Instancí třídy se vytvoří *SOAP* server prostřednictvím

něhož bude zajištěna komunikace. Následně je třeba definovat jmenný prostor (*namespace*) webové služby, aby webová služba byla jedinečná a identifikovatelná. V rámci našeho řešení webové služby je pojmenování *namespace igoofer*. Po inicializaci je pomocí metod třídy nastaveno kódování přenosu, bez tohoto nastavení dochází k nedostatečnému zobrazení dat na obou stranách synchronizovaných databází, může se totiž automaticky nastavit do některého z kódování nepodporujícího české znaky a vzniká problém s diakritikou. Následuje vložení souboru, ve kterém jsou uvedeny odkazy na všechny soubory reprezentující jednotlivé nástroje webové služby. Ty jsou umístněny v jednom definovaném souboru, jenž je posléze vložen do souboru *igoofer_webservice*, kde využívá definici struktury služby. Nakonec je vložen soubor *function.php*, který obsahuje pomocné funkce, jež jsou využívány v rámci funkcí zajišťujících samotnou synchronizaci.

```
<?php
   // Pull in the NuSOAP code
   require_once('lib/nusoap.php');
   // Create the server instance
   $server = new soap_server();
   $server->soap_defencoding = 'UTF-8';
   $server->decode_utf8 = false;

   $server->configureWSDL('iGooFer', 'urn:iGooFer');
   include('config.php');

   require_once('config.class.php');
   require_once('config.php');
   require_once('log/class/error.class.php');
   require_once('log/class/file.class.php');
   require_once('log/class/logerr.class.php');
   include('common.php');
?>
```

Obr. 18 Ukázka souboru igoofer_webservice.php

4.2.1 Pomocné funkce synchronizace

V rámci synchronizace byly vyvinuty funkce, které jsou používány funkcemi zajišťujícími samotný přenos dat. Jsou to ty, které zajišťují kontroly nebo transformaci vstupních dat. Nejsou registrovány v rámci služby, jsou volány pouze za účelem podpory synchronizace.

Na obr. 19 je zobrazena část funkce *NormalizeValue* pro úpravu formy dat přenášených v rámci synchronizace. Obsahuje dva vstupní parametry. První, *value*, reprezentuje hodnotu, která bude převedena, podle druhého parametru *type*, na hodnotu normalizované tvaru. Na základě datového typu se převede hodnota do správného tvaru. Pokud např. odešleme přes synchronizační nástroj pole typu *datum*, má formát dle *PostGres* a je třeba ho převést do masky dle *MySQL*, nebo, je-li poslána hodnota typu *string*, je následně opatřena uvozovkami. Návratová hodnota funkce je hodnota ve správném databázovém formátu a typu. Funkce zajišťuje, aby nedošlo k chybě při vykonání SQL dotazu na základě špatné datové formy. Funkce je využívána pro data vstupující do *MySQL*.

```
function NormalizeValue($value,$type)
{
  switch (strtolower($type))
  {
    ...
  }
  return $ret;
}
```

Obr. 19 Část funkce NormalizeValue

Funkce *ToNullValue* na obr. 20 má jeden vstupní parametr *value* a obsahuje hodnotu předanou ze synchronizační funkce. Návratová hodnota funkce se určuje podle datového typu

php z parametru *value*. Funkce se používá převážně ve směru z webového prostředí.

```
function ToNullValue($value)
{
  $ret='';
  switch(gettype($value))
  {
    ...
  }
  return $ret;
}
```

Obr. 20 Část funkce ToNullValue

Funkce *Decrypt* a *Encrypt* (obr. 21) zajišťují zašifrování a dešifrování hesla pro databázi *MySQL*. Heslo je přenášeno v parametru synchronizačních funkcí. Funkce má jeden vstupní parametr typu *string*. Návratová hodnota se liší podle typu funkce, u dešifrovací je návratová hodnota původní heslo a u šifrovací se vrací zakódované heslo. Funkce se používají k zabezpečení předávaného hesla v rámci synchronizace, aby nedošlo k jeho zachycení a zneužití. Pro šifrování hesel se využívá algoritmu *MCRYPT_MODE_CBC*.

```
function Decrypt($strToDecrypt)
{
...
}

function Encrypt($strToEncrypt)
{
...
}
```

Obr. 21 Hlavička funkcí Encrypt a Decrypt

4.2.2 Synchronizační funkce

Synchronizační funkce jsou samotným srdcem synchronizace, na základě jejich provolávání dochází k předávání dat. Pro vývoj této hlavní části webové služby je využit nástroj generátor kódu. Pro jednotlivé tabule jsou vygenerovány funkce pro jejich synchronizaci. Při vývoji těchto funkcí byla zvolena tabulka, na jejím základě byly napsány funkce, které pracují a modifikují data. Tyto funkce se pro danou tabulku otestovaly a odladily. Po testech funkční správnosti byly dané přepsány pomocí jazyka XSL do šablony. Na základě šablony a pomocí generátoru byly funkce a struktury webové služby vygenerovány pro všechny tabule databáze. Generátor nám poskytuje následující výhodu: v okamžiku, kdy se vytvoří nová tabulka (nové pole) ve stávající tabulce, provede se pouze nové generování synchronizačních funkcí a jejich struktur a synchronizace je během pár vteřin rozšířena. Dále následuje seznam jednotlivých synchronizačních funkcí a jejich stručný popis. Názvy jednotlivých funkci se skládají z názvu tabulky, jako je prefix, a z označení činnosti funkce.

Funkce *Nazev_tabulkyListData* (obr. 22) má čtyři vstupní parametry. První dva reprezentují login a heslo pro přihlášení do databáze na webovém rozhraní. Na základě přihlašovacích údajů se otevře připojení do databáze. Tato funkce slouží pro přenos dat do systému *Goofer*. *Goofer* se přes webovou službu dotáže této funkce na určitý výsledek dat. S dotazováním na určitou skupinu souvisí třetí a čtvrtý parametr *LimitFrom* a *LimitTo*. Pomocí zmíněných parametrů se bere daný fragment dat. Tato funkce obsahuje *SELECT* příkaz, který je napojený přes *join* na tzv. replikační tabulku, kde je nastavena *WHERE* fráze přes parametry *LimitFrom* a *LimitTo*. Pomocí *SELECT* příkazu se provede naplnění proměnné typu *pole output_list*. Jedná se o asociační pole podle struktury tabulky. Pole *output_list* je návratovou hodnotou funkce a při jejím dotázání přes webovou službu je toto pole odesláno. Návratové pole funkce má striktně danou strukturu, tato podmínka je zajištěna funkcí generátoru, který asociační pole vygeneruje přesně podle struktury dané tabulky, pro kterou je funkce určena.

Následně je návratové pole zpracováno na straně desktopové aplikace. Ještě před odesláním výsledku se provede vyvolání funkce *deleteRowReplication*, ta bude zmíněna později.

```
function HELPDESK_PHASEListData($UserName, $Password, $LimitFrom, $LimitTo)
{
  global $database;
  global $server_name;
  $conn = mysql_connect($server_name, Decrypt($UserName), Decrypt($Password));
  $db = mysql_select_db($database, $conn);
  mysql_query("SET CHARACTER SET utf8, NAMES utf8", $conn);

   try
   {

  if (!$conn)
  {
    throw new log(mysql_error(),22);
  }

  $SQL = "SELECT t.`phase_order`,t.`phase_name` ...
  FROM `helpdesk_phase` t
  JOIN `replication_table` r ON r.`new_value` =
  CONCAT(t.`helpdesk_phase_id`,';',t.`helpdesk_phase_type_id`)
  WHERE r.`table_name` = 'helpdesk_phase' ".(($LimitTo)? " order by
  r.replication_table_id
  LIMIT ".$LimitFrom." , ".$LimitTo."":'').";

  $result = mysql_query($SQL,$conn);
  if (!$result)
  {
    return null;
  }

  $output_list = Array();
  while($records = mysql_fetch_array($result))
  {
     $output_list[] = Array(
      "phase_order" => NormalizeValue($records["phase_order"],"int"),
      "phase_name" => NormalizeValue($records["phase_name"],"string"));
     ...
  }
  }
   catch(Exception $except)
   {
     $except->erroMessage($strsql);
   }

  return $output_list;
}
```

Obr. 22 Část funkce HELPDESK_PHASEList_date

Funkce *Nazev_tabulkySetData* (obr. 23) má čtyři vstupní parametry. První dva, stejně jak u předchozí funkce, zahrnují login a heslo, sloužící pro vytvoření připojení k databázi. Předposlední parametr *input_list* je nejdůležitější, parametr je podobně jak v předchozí funkci tvořen asociačním polem. Funkce umožňuje přenášet data z *Gooferu* do webové databáze. V těle funkce následuje přepínací struktura *case*, která vyhodnocuje poslední parametr *Action*. Podle hodnoty parametru *Action* se provede SQL příkaz, a to buď příkaz *INSERT*, pro vložení záznamu, příkaz *UPDATE* pro modifikaci nebo poslední možnost, příkaz *DELETE* pro smazání záznamu. Funkce má návratovou hodnotu typu *boolean*, se dvěma možnými výstupy: *True*, pokud funkce proběhla korektně nebo *False* při chybném průběhu funkce. Rozhodovací struktura *case* je umístěna ve smyčce *foreach*, která vykoná jednotlivé *sql* příkazy přes celé vstupní pole *input_list*. V každém průběhu cyklu se zjišťuje, jestli byl *sql* příkaz proveden bez chyby. Nastane-li chyba při zpracování dotazu, je funkce ukončena a veškeré změny, které před chybou proběhly, jsou vráceny zpět pomocí *rollback*. SQL příkazy probíhají v transakci, tzn. pokud nebudou vykonány všechny dotazy v rámci transakce, nastane vrácení všech změn v dané transakci. Když je funkce vykonána bez jakékoli chyby, provede se *commit* a následně se všechny změny v transakci projeví. Očekávaná struktura vstupu musí odpovídat struktuře

tabulky, pro kterou je funkce volaná. Je to nutná podmínka, jelikož SQL dotazy jsou generovány přesně na strukturu tabulky a obsahujů jejich jednotlivá pole. Podle návratové hodnoty funkce se řídí mazání záznamu z replikační tabulky.

```
function HELPDESK_PHASESetData($UserName, $Password, $input_list, $Action)
{
  $commit = true;

  ...

  return $err;
}
```

Obr. 23 Část funkce HELPDESK_PHASESetData

Funkce *Nazev_tabulkyInitData* (obr. 24) má podobný účel jak funkce *SetData*. Na vstupu jsou rovněž čtyři parametry. První dva slouží jako v předchozích případech pro připojení k databázi. Parametr *input_list* je opět pole hodnot dle striktně dané struktury podle jednotlivých tabulek. Význam funkce je následující: pokud je nově vytvořena databáze a je bez jakéhokoliv záznamu, provede se tzv. inicializační import, kde se vytvoří úplný obraz dat z databáze desktopové aplikace na webovou databázi. Díky této funkci se přenesou veškerá data na webové rozhraní a inicializují se. Poslední parametr *Delete* určuje, zda má být tabulka před inicializačním importem promazána. Dotazy spuštěné ve funkci probíhají v transakci, nedojde-li ke správnému zpracování všech dotazů, veškeré změny budou vráceny zpět pomocí *rollback*. Informaci o chybném průběhu nese návratová hodnota funkce *err*.

```
function HELPDESK_PHASESetData($UserName, $Password, $input_list, $dction)
{
  $commit = true;

  ...

  return $err;
}
```

Obr. 24 Inicializační funkce HELPDESK_PHASEInitData

Funkce *nazev_tabulkyListDataConfirm* na obr. 25 je volána po úspěšném procesu synchronizační funkce. Funkce má čtyři vstupní parametry: první dva slouží pro vytvoření databázového připojení. Další parametr *ReplicationId* nese hodnotu klíče pro určité tabulky, nad kterou synchronizační funkce probíhala. Posledním parametrem je *Action* a nabývá hodnot (U, I, D), jedná se o změnové operace, které proběhly nad daty v databázích obou stran. Na základě parametru *ReplicationId* a *Action* se provede promazání dat z replikační tabulky, aby nemohlo dojít k duplicitnímu přenosu záznamů. Návratovou hodnotou funkce je hodnota typu *boolean*, určuje zda funkce proběhla bez chyby.

```
function HELPDESK_PHASEListDataConfirm($UserName, $Password, $ReplicationId,
$Action)
{

   ...

   return deleteRowsReplication('helpdesk_phase', $ReplicationId, $Action);

}
```

Obr25 Funkce pro smazání záznamu replikační tabulky

Funkce *FileProcessor* (obr. 26) vykonává přenášení objektů souborů, které jsou uložené v databázi desktopové aplikace nebo na pevném disku. Jedná se o soubory typu pdf, jpeg, atd.. Objekty jsou v databázi uloženy jako *binary large object* v binární podobě, tyto soubory jsou poslány v parametru *ContenFile*. Pomocí funkce PHP *base64_decode* je převeden a následně uložen na FTP do adresáře, který je nastaven v souboru *config*. První dva parametry slouží k

připojení do databáze (*UserName*, *Password*). Parametr *OriginalName* nese název souboru, včetně přípony typu souboru. Podle parametru *Init* se jen určuje, jestli se má adresář pro ukládání objektů vytvořit a zda se mají nastavit práva pro zápis. Parametrem *Action* je parametr změnových operací pro soubory, tzn. daný soubor vytváří, přepíše nebo smaže. Pro objekty uložené na pevném disku probíhá proces obdobně, jen se využívá tabulky *large_object*, kde jsou uloženy odkazy na přenášené soubory.

```
function FileProcessor($UserName, $Password, $NewName, $OriginalName,
$ContentFile, $Action, $Init)
{
   ...
}
```

Obr. 26 Funkce pro přenos souboru FileProcessor

Funkce *CountReplicationRowns* a *CountReplicationTables* jsou pouze zjišťovací, neprovádí žádné replikační procesy. Při jejich volání se zjišťuje počet záznamů. U obou funkci jsou parametry pro přihlášení k databázi (*UserName*, *Password*). Funkce *CountReplicationRowns* má ještě dva další parametry a to parametr *Table*, který určuje, nad jakou tabulkou se počet záznamů k replikaci zjišťuje. Pomocí parametru *Action* určuje další omezující podmínku pro zjišťování počtu záznamu. Počet záznamů je omezen na jméno tabulky a typ změny záznamu. Funkce *CountReplicationTables* nemá žádné jiné parametry než pro připojení k databázi. Pomocí funkce se vrací pole názvů tabulek, které mají záznam v replikační tabulce. Tyto funkce mají optimalizační charakter a urychlují dotazování na replikační funkce.

Funkce *Ping* (obr. 27) kontroluje připojení k databázi. Před zahájením replikačního procesu se pomocí této funkce testuje připojení k databázi. Její další důležitou rolí je návratová hodnota. Funkce vrací verzi databáze (aplikace). Dojde-li k rozpoznání verzí mezi databází a aplikací, proces replikace není zahájen.

```
function Ping($UserName, $Password)
{
  global $database;
  global $server_name;
  $conn = mysql_connect($server_name, $UserName, $Password);
  $db = mysql_select_db($database, $conn);
  mysql_query("SET CHARACTER SET utf8, NAMES utf8", $conn);
  if (!$conn)
  {
    return "error connect";
  }

  $sql = "SELECT `version_id` FROM `goofer_version` ORDER BY `caption` DESC";
  $result = mysql_query($sql);
  $records = mysql_fetch_array($result);

  if($records['version_id']){
        $return = $records['version_id'];
  }
  else{
        $return = 'noversion';
  }
  return $return;
}
```

Obr. 27 Kontrolní funkce Ping

4.2.3 WSDL a registrace funkcí

Jazyk WSDL slouží k popisu webových služeb jako množiny koncových bodů zpracovávajících zprávy. Jednotlivé zprávy a operace jsou přitom popisovány na abstraktní úrovni, až následně jsou svázány s konkrétním síťovým protokolem a datovým formátem. WSDL umožňuje snadné vytvoření popisu rozhraní, nejčastěji se popisují služby, které si posílají zprávy pomocí formátu SOAP a protokolu HTTP [16], využitím třídy NuSOAP a schopnosti WSDL zapouzdřit jedno nebo více XML schémat k popisu datové struktury

využívané služby. WSDL je využíváno pro definování nové datové struktury a k jejímu popisu. Pro funkčnost synchronizace bylo třeba vytvořit popis a registrovat strukturu jednotlivých funkcí určených pro synchronizaci v rámci služby. V rámci služby je třeba definovat vstupní a výstupní parametry jednotlivých synchronizačních funkci do datových struktur a provést jejich registraci. Úkon definice a registrace je třeba provést pro každou synchronizační funkci, s kterou se pracuje prostřednictvím webové služby. Pro vývoj registrací a definování struktury byla zvolena jedna tabulka a všechny její synchronizační funkce. Pro jednotlivé funkce byl napsán kód pro registraci a definici struktury parametru. Příklady registrace a definice struktur budou k vidění níže.

Pro jednotlivé funkce webové služby dané tabulky byla vytvořena XSL šablona pro definici nové datové struktury a registraci funkce určené pro generátor. V šabloně se provedlo přesání původního kódu prostřednictvím XSL. Jednalo se o nahrazení hodnot, např. názvu tabulek, funkcí či sloupu, měnící se v závislosti na dané tabulce. Načtením šablony do generátoru bylo provedeno vygenerování těchto nových struktur pro jednotlivé tabulky v databázi. Následovat budou jednotlivé příklady datových struktur pro určité případy funkcí webové služby potřebné pro synchronizaci dat. S využitím třídy *NuSOAP* se přidají nové definice struktury prostřednictvím metody *addComplexType*. Příklad definice struktury výstupu pro synchronizační funkci *HELPDESKDataList* je na obr. 28. Obdobným způsobem se provede definice i pro funkci *HELPDESKInitData*. Tyto datové struktury definují přenášená pole, která využívají synchronizační funkce a odkazují se na ně při registraci do webové služby.

```
$server->wsdl->addComplexType(
'HELPDESKDataList',
'complexType',
'struct',
'all',
'',
array(
'hd_deadline' => array('hd_deadline' => 'hd_deadline', 'type' => 'xsd:dateTime',
'nillable' => 'true')
 ...
, 'hd_caption' => array('hd_caption' => 'hd_caption', 'type' => 'xsd:string',
'nillable' => 'true')
));
```

Obr. 28 Definice výstupní struktury funkce DataList

Ukázka definice datové struktury *array types* (obr. 29) přes WSDL pro SOAP funkce *DataList*. Využívá se výše definované struktury.

```
$server->wsdl->addComplexType(
'helpdeskDataListRs',
'complexType',
'array',
'',
'SOAP-ENC:Array',
array(),
array(
array('ref'=>'SOAP-ENC:arrayType','wsdl:arrayType'=>'tns:HELPDESKDataList[]')
), 'tns:HELPDESKDataList'
);
```

Obr. 29 Definice výstupní struktury array funkce DataList

Popis parametrů metody *addComplexType*:

1. Jméno
2. Typová třída (complexType | simpleType | atribut)
3. PhpType (podpora array a struct)
4. Compositor (all | sequence | choice)
5. Namespace (jméno jmenného prostoru)
6. Prvky = array(name = array(name => “, type”))
7. attrs = array(array(“ref“ => “, “arrayType” ,“arrayType” => []))

Zatím se provedla definice typů dat (struktur), které budou přes webovou službu vstupovat a vystupovat. Dále se definuje vlastní provoz webové služby. Provede se registrace a definice jednotlivých funkcí, včetně vstupních parametrů a návratové hodnoty. Určí se datové typy parametru a jako výstup použijeme již definované struktury pro výstup. Registrování funkcí se realizuje pomocí metody *register*. Registrovaná funkce je funkce psaná v PHP (představeno v podkapitole 4.2.2).

Příklad způsobu registrace funkce *HELPDESKListData* je uveden na obr. 30, krok registrace je nutný pro viditelnost funkce uvnitř webové služby.

```
$server->register("HELPDESKListData",
array(
'UserName' => 'xsd:string',
'Password' => 'xsd:string',
'LimitFrom' => 'xsd:int',
'LimitTo' => 'xsd:int'
),
array('return' => 'tns:helpdeskDataListRs'),
'urn:iGooFer',
'urn:iGooFer#HELPDESKListData',
'rpc',
'encoded',
'Returns a list of data'
);
```

Obr. 30 Registrace funkce ListData

Popis parametrů metody *register*:

1. Název
2. Pole vstupních parametrů
3. Návratová hodnota
4. Namespace
5. Název s namespace
6. Styl
7. Použití
8. Popis

4.3 Princip synchronizace

Princip synchronizace dat může být rozdělen do dvou částí. První část je jednodušší a nemusí se u ní řešit problém, který by vznikl při duplicitě v datech. Tato část je pouze jednosměrná − a to vždy z databáze v *PostgreSQL* desktopové aplikace na webové rozhraní do databáze např. MySQL. Tuto část můžeme nazvat jako inicializační import a probíhá následujícím způsobem: ze všech vybraných tabulek obsažených v databázi *Postgre* se přenesou veškeré záznamy na stranu webového rozhraní a vytvoří se jejich přesný obraz. Po inicializačním importu jsou obě databáze datově identické bez jakýchkoli duplicit. Pro synchronizaci dat je primární databáze v *Postgre.* Z této databáze se vždy vychází, a když dojde ke konfliktu, pravda řešení je vždy na straně *Postgre* databáze desktopové aplikace. V první části synchronizace tedy dojde k přesunu veškerých dat podle pravidel *Postgre.*

Druhá část je již trochu složitější, na obou dvou stranách máme identická data a úkolem druhé části je provádět přesun pouze rozdílových dat mezi jednotlivými databázemi. Je třeba přenášet jen záznamy, které byly nově přidány nebo upraveny na *Postgre* databázi a tyto změny se projevily v databázi na webovém rozhraní. To samé musí fungovat i v druhém směru. Druhá část principu synchronizace zajišťuje datovou stejnorodost i po změnách, mazání a přidávání záznamů v obou databázích. Principem synchronizace je, aby se změny projevily na obou databázích a byl zachován přesný obraz *Postgre* databáze na webovém rozhraní a naopak. Tohoto kroku se docílí přenášením změněných záznamů na obou stranách, tak aby bylo možné přenášet pouze změnové záznamy rychlým způsobem bez toho, aby docházelo k porovnávání

jednotlivých záznamů v daných tabulkách jestli došlo ke změně. Porovnávání jednotlivých záznamů by bylo velmi časově i výkonově náročné.

Je třeba zavést určitá pravidla pro změnové záznamy a také je nějakým způsobem uchovávat. Pro tyto účely byla vytvořena tabulka *replication_table*, kde se ukládají veškeré změny provedené v databázi. Pomocí této tabulky se zajišťuje celá replikace změněných a nově vytvořených dat. Při odesílání dat přes webovou službu se pro každou tabulku berou pouze záznamy, které jsou v této obsaženy.

Dalším nutným krokem bylo zajištění ukládání změněných záznamů do replikační tabulky. U kroku plnění replikační tabulky na straně databázového serveru *PostgreSQL* se využívá funkcionality *Triggeru*. U všech tabulek, kde bylo třeba zajistit replikaci dat byl vytvořen *Trigger*. Tyto *triggery* reagují na všechny typy události změny a to *UPDATE, INSERT* a *DELETE*. Při každé změně se vykoná část kódu pro jednotlivou událost změny a propíše záznam do replikační tabulky. Na straně webového rozhraní funkcionalitu *Triggeru* supluje vrstva umístněná nad databází DAO. V DAO je metoda, která provádí zápis do replikační tabulky na webovém rozhraní pod příslušnou změnovou událostí.

4.3.1 Replikační tabulka

Replikační tabulka je nezbytným prvkem pro synchronizaci. Nastane-li v aplikaci změna, v tabulce se vytvoří záznam o změně. V tabulce se neukládají celé řádky záznamu, ve kterých byla provedena změna, ale uchovává se pouze odkaz na změněný záznam. Ukázka struktury tabulky *replication_table* potřebné pro replikaci je zachycena na obr. 31.

Sloupce	Typ
record_id	varchar(250)
table_name	varchar(50)
change_type	varchar(1)
create_date	datetime
new_value	text
old_value	text

Obr. 31 Struktura replication_table

Popis jednotlivých polí *replication_table* na obr. 31:

1. *record_id* – do tohoto pole se ukládá název primárního klíče záznamu, kde je změna prováděna. Na základě tohoto atributu se napojuje přímo záznam z tabulky, kde byla změna provedena.
2. *table_name* – v tomto atributu je uveden název tabulky, ve které se daný záznam měnil. Název tabulky pak slouží jako parametr u přenosu změn, kdy dochází k prokřížení dat s uvedenou tabulkou a replikační tabulkou za pomoci *id_record*. Tímto postupem dostaneme fragment změn.
3. *change_type* – tento atribut uchovává, jaká změna byla u daného záznamu provedena. Nabývá následujících hodnot U – změna hodnoty v záznamu, I – vložení nového záznamu do tabulky, D – odstranění záznamu z uvedené tabulky.
4. *create_date* – zde se uchovává datum, kdy byla daná změna provedena.
5. *new_value* – tento atribut uchovává samotnou hodnotu primárního klíče záznamu kde byla provedena změna.
6. *old_value* – zde se ukládá původní hodnota primárního klíče a to jen v případě, kdy nastane změna v hodnotě primárního klíče. V ostatních případech se opět uloží hodnota primárního klíče.

Replikační tabulky se nachází v *PostgreSQL* databázi i v databázi na webovém prostředí. Tyto tabulky jsou navzájem nezávislé a obsahují odlišná změnová data. Replikační tabulka desktopové aplikace zaznamenává změny uvnitř systému, která se následně přenáší na webové rozhraní. Replikační tabulka na webovém rozhraní udržuje informace o změnách, které proběhly na webové aplikaci a změny zde uvedené se propisují do systému. Replikační tabulky na obou rozhraních zajišťují datovou konzistentnost a udržují zrcadlový obraz primární databáze v *PostgreSQL* na webovém rozhraní.

4.3.2 Způsoby přidávání záznamů pro replikaci

Na straně *Postgre* databáze zajišťuje funkcionalitu inicializace replikační tabulky speciální funkce *Trigger*. Ta se spustí pod určitou událostí nad tabulkou. Jedná se o změnové operace *insert, update, delete*. U funkce *Trigger* se volí, kdy má být spuštěna, možnosti jsou před provedením nebo po provedení dané operace.

Trigger, který zajišťuje přidávání změnových záznamu do replikační tabulky, volá funkci *replication*. V této funkci jsou větve *IF* pro jednotlivé tabulky a každá větev je dané tabulce přizpůsobena. V jednotlivých větvích se provádí nastavení parametrů pro záznam do replikační tabulky pro tabulku, která *trigger* vyvolala. Na obr. 32 je část kódu funkce s větví pro tabulku *Helpdesk* provádějící zápis do replikační tabulky.

```
CREATE OR REPLACE FUNCTION replication() RETURNS trigger AS
DECLARE
ids text;
vals text;
vals_old text;
myquery text;
keys text;
sets text;
typ character varying;
rec RECORD;
BEGIN
ids='';
vals='';
IF TG_OP = 'INSERT' THEN
    typ='I';
ELSEIF TG_OP = 'UPDATE' THEN
    typ='U';
ELSEIF TG_OP = 'DELETE' THEN
    typ='D';
END IF;

IF TG_RELNAME = 'helpdesk' THEN ids = 'helpdesk_id';
IF TG_OP <> 'DELETE' THEN vals = NEW.helpdesk_id; END IF;
IF TG_OP = 'DELETE' OR TG_OP = 'UPDATE' THEN vals_old = OLD.helpdesk_id; END IF;
IF TG_OP = 'UPDATE' THEN
     keys = 'helpdesk_id=''' || OLD.helpdesk_id || '''';
     sets = 'helpdesk_id=helpdesk_id';
END IF;
ELSEIF
...
END IF;
INSERT INTO replication_table (
record_id,new_value,old_value,table_name,change_type,create_date ) VALUES (
ids,vals,vals_old,TG_RELNAME,typ,CURRENT_TIMESTAMP );
--Replikace i podrizenych zaznamu pri Update
IF TG_OP = 'UPDATE' THEN
SELECT INTO myquery child_tables_joins FROM goofer_table WHERE table_name =
TG_RELNAME;
IF (myquery is not null OR myquery <> '') AND (keys <> '' OR keys is not null)
THEN
 EXECUTE 'UPDATE ' || substr(myquery,0,strpos(myquery,' ')) || ' SET ' || sets ||
' WHERE ' || keys;
END IF;
END IF;
```

Obr. 32 Příklad funkce pro přidávání záznamu do replikační tabulky

Na straně webového rozhraní zajišťuje přidávání záznamu metoda *writeToReplication* třídy *DAO (*na obr. 32). Metoda je volána na konci každé metody vrstvy DAO, která provádí manipulaci s daty a má tři parametry, pomocí kterých se inicializují parametry pro zápis do replikační tabulky. Parametr *tableName* obsahuje název tabulky, ve které změna probíhá. Podle typu metody, která provádí změny, předává parametr *Action* a nabývá hodnot (I, U, D). Poslední parametr t*able* reprezentuje *business object* záznamu, kde probíhá změna. Z *business objectu* se zjistí hodnoty primárních klíčů.

```
function writeToReplication($tableName, $Action, $table){

  $dateCreate = date("Y-m-d H:i:s");
  $arrPk = $this->getPrimaryKey('variable');

  foreach ($arrPk as $key => $val) {
    $keyValue[] = $table->$val->getValue();
    $keyName[] = $table->$val->Name;
  }

  $key = implode(';', $keyName);
  $keyV = implode(';', $keyValue);

  $sql = "INSERT INTO `replication_table` (`record_id`, `table_name`,
  `change_type`, `create_date`, `new_value`, `old_value`)
  VALUES (?, ?, ?, ?, ?, ?);";

  $sqlQuery = new SqlQuery($sql);

  $sqlQuery->set($key);
  $sqlQuery->set($tableName);
  $sqlQuery->set($Action);
  $sqlQuery->set($dateCreate);
  $sqlQuery->set($keyV);
  $sqlQuery->set($keyV);

  $this->executeInsert($sqlQuery);
}
```

Obr. 32 Metoda třídy DAO pro zajištění replikace

4.3.3 Problém při řešení fáze synchronizace a využití

Při řešení fáze synchronizace se vyskytly určité problémy, které nebyly předpokládány na začátku vývoje a musely být provedeny značné úpravy. Webové služby byly nachystány k provozu, a když se začal testovat jejich chod a cvičně se provedl inicializační import nad velkým objemem dat, došlo k chybě běhu skriptu. Chyba byla vyvolána samotným Apache serverem, a to na délku běhu skriptu. Zpracování výkonové části skriptu spadlo na *timeout*, který je přednastaven na hodnotu 5 minut. Díky tomuto zátěžovému testu s velkým objemem dat byly nedostatky zavčas odhaleny a následně tak mohly být přijaty úpravy funkcí do podoby, které jsou zde uvedeny. K chybě docházelo z toho důvodu, že všechny funkce pro inicializaci jednotlivých tabulek se vykonávaly jako jeden velký balík a při velkém objemu dat došlo k ukončení skriptu. Proto byly upraveny do takové podoby, aby mohly být spouštěny postupně po jedné. Každá funkce pro jednotlivou tabulku je vykonávaná samostatně a i při velkém objemu dat zatím proběhlo její vykonání v časovém limitu. Navíc došlo k rozdělení rozhraní webové služby, místo jednoho velkého řešení byl proveden rozpad na dílčí webové služby.

Pro realizaci samotné synchronizace dat bylo třeba proniknout do oblasti webových služeb a prozkoumat možnosti třídy *NuSOAP*. Na základě těchto informací bylo navrženo řešení, které je tvořeno dvěmi částmi, a to přenosem dat pomocí webových služeb a pak samotná část replikací změn přes tabulky replikací.

V současné době je synchronizační nástroj aktivně využíván a je plně funkční. Je nasazen na odlišnou webovou aplikaci, než jaká je součástí diplomové práce.

5 APLIKACE HELPDESK

Na úvod kapitoly bude zmíněn návrh a architektura webové aplikace Helpdesk a jaké byly na aplikaci požadavky. V podkapitolách budou podrobně popsány jednotlivé vrstvy aplikace. U každé vrstvy budou uvedeny příklady jejich činnosti.

5.1 Návrh a architektura aplikace

Při návrhu samotné aplikace Helpdesk (dále už jen Helpdesk) byly striktně dodržovány požadavky ze strany zadavatele. Ty se týkaly jednotlivých funkcionalit Helpdesku, ale i vývojového nástroje, v kterém má být Helpdesk vyhotoven. Zadavatel nastínil představu, jak si představuje funkčnost a podobu Helpdesku. Ve fázi návrhu aplikace se konalo velké množství schůzek, kde se řešily jednotlivé požadavky a schvalovala se již navržená řešení v rámci diplomové práce. Součástí schůzek o návrhu byl i modul v aplikaci *Goofer*, který byl souběžně vyvíjen na základě řešení diplomového projektu. Diplomová práce se týká pouze webového řešení. Webové řešení je nástavbou modulu Helpdesk v aplikaci *Goofer* a pomocí synchronizace spolu navzájem komunikují. Webová nástavba Helpdesk je zaměřena na uživatele (zadavatele). Přes aplikaci Helpdesk je umožněna interaktivní komunikace s uživatelem z aplikace *Goofer*.

Hlavním úkolem a činností webové nástavby Helpdesk je zajistit obousměrnou komunikaci s uživatelem prostřednictvím webového rozhraní do aplikace *Goofer*. Webová aplikace přijímá dotazy od uživatele, které se přesunou do desktopové aplikace ke zpracování a následně se interaktivně zobrazí výsledky pro uživatele. Webová nástavba by měla být obrazem modulu v aplikaci *Goofer* a zajišťovat činnosti spojené s uživatelem (zadavatelem). Modul aplikace *Goofer* obsahuje činnosti spojené z řešitelem požadavku, zadané přes webového uživatele.

5.1.1 Požadavky na aplikaci Helpdesk

Od zadavatele bylo dodáno velké množství požadavků, které bylo potřeba rozřadit do jednotlivých oblastí a zobecnit na nadřazené požadavky. Rozhodovalo se, zda požadavek se týká přímo návrhu aplikace, nebo datové struktury

Následuje krátký přehled zobecněných požadavků:

- Vytvořit prostředek pro přenášení dat na webové rozhraní
- Navrhnout datovou strukturu pro *Postgre* a MySQL, včetně všech procedur a funkcí
- Možnost změnit datové úložiště Helpdesku
- Realizace webové nástavby Helpdesk
- Provést grafické řešení a umožnit snadnou změnu grafického vzhledu

Přehled požadovaných nástrojů pro vývoj:

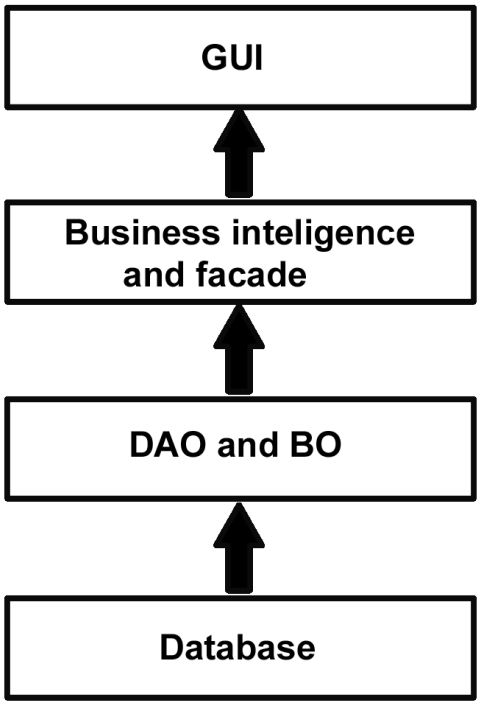
- Jazyk PHP5 byl zadán zadavatelem pro svou nezávislost na platformě a pro svou rozšířenost v odvětví webových aplikací
- Datové uložiště v *Postgre* bylo určeno vývojem aplikace *Goofer*
- Pro webové rozhraní byla určena databáze MySQL

Výše byl zmíněn seznam obecných požadavků na vývoj Helpdesku. Další požadavky se již týkaly přímo funkcionalit Helpdesku. Byl proveden sběr informací na položky, které mají být součástí jednotlivých požadavků (*task*) webového modulu. Jedná se o informace typu *název požadavku*, *termín řešení*, atd. Rozřazení těchto informací bylo již na drobnější dílčí části, které se týkají problematiky při vývoji Helpdesku. Uvedené informace byly využity i pro návrh a vývoj datové struktury. Více informací o návrhu datové struktury bude řečeno později.

Po tvorbě datové struktury a navržení struktury a funkcionalit webové aplikace proběhla realizace modulu *Helpdesk* v aplikaci *Goofer* fimou AbisTech s.r.o. Na základě vytvořeného modulu vznikla webová nástavba, která je předmětem diplomového projektu. V průběhu vývoje modulu v aplikaci *Goofer* firmou AbisTech s.r.o. probíhaly četné konzultace funkcionalit, které budou realizovány na webovém rozhraní.

5.1.2 Návrh a model webové aplikace Helpdesk

Podle požadavků od zadavatele byl navržen model architektury Helpdesk. Protože zadavatel požadoval možnost změnit datové uložiště nebo snadno a rychle upravit grafický vzhled, byl navržen vícevrstvy model. Ukázka modelu architektury je na obr. 33.



Obr. 33 Model architektury webové aplikace Helpdesk

Pro Helpdesk byl navržen čtyřvrstvý model, aby bylo docíleno nezávislosti na datové vrstvě a oddělení grafického prostředí. Model obsahuje následující vrstvy.

1. *Database* – vrstva je tvořena databází, která slouží pro ukládání dat webové aplikace. Databázový systém je možno měnit, protože aplikace ve vícevrstvém modelu nepřistupuje přímo k datu v databázi, ale prostřednictvím vrstvy umístěné nad ní. U jednovrstvé aplikace by uvedeného efektu nemohlo být dosaženo, protože výkonová logika je úzce spojena s databází.

2. *DAO* a *BO* – (celý název vrstvy *Database Access Object* a *Business Object*) vrstva je určena pro přímou komunikaci s databází. Prostřednictvím vrstvy se připojuje k databázi, následně se nad ní vykonávají požadované dotazy vyšší vrstvy. Výstupem vrstvy jsou inicializované objekty, které reprezentují data z databáze (tzv. *Business Objects*). Na datové objekty se dotazuje vyšší vrstva.

3. *Business Inteligece and Facade* – vrstva je tvořena výkonovou logikou aplikace.

Zde jsou realizovány jednotlivé funkcionality aplikace. Vrstva zpracovává *Business Object* nebo posílá inicializované datové objekty zpět ke zpracování. K předávání objektu pomáhá část *Facade*.

4. *GUI* – (*Graphical User Interface*) tvoří grafický vzhled aplikace a uživatelské prostředí. Jedná se pouze o grafické zobrazení aplikace, kde jsou odkazy na proměnné. Pomocí této vrstvy došlo k oddělení grafického vzhledu od výkonové logiky aplikace.

Jednotlivé vrstvy modelu aplikace budou podrobněji popsány v následujících podkapitolách. Pomocí vícevrstvého modelu bylo docíleno nezávislosti na databázi a možnosti do budoucna měnit uložiště dat.

5.2 Vrstva *Database*

Vrstva zajišťuje veškeré ukládání dat z aplikace a následně jejich předávání. Funkci datového uložiště plní pro aplikaci *Goofer* databázový server *PostgreSQL* a pro webové rozhrání MySQL. Vrstva databáze je nezbytnou součástí každé dynamické aplikace pracující s daty. Aplikace potřebuje přijatá data zpracovat a následně uchovat. Druhá část je prezentace uložených dat. K těmto účelům je určena probíraná vrstva. Rozdílnost v databázových systémech využívaných aplikací *Goofer* a webovým modulem Helpdesk bude řešena ve vyšší vrstvě DAO. Do této vrstvy patří databázová struktura Helpdesk a její vývoj.

5.2.1 Postup návrhu databázové struktury

Jako první krok před samotným vývojem aplikace HelpDesk byl proveden návrh datové struktury. Se zadavatelem probíhaly dlouhé konzultace jednotlivých funkcionalit a na základě těchto informací byl sestaven seznam entit. Díky seznamu byl realizován datový model aplikace. Protože tato aplikace se začleňovala do systému s již vytvořenou databázovou strukturou, bylo nezbytné si nastudovat pravidla a jmenné konvence existující struktury a řídit se jimi při návrhu samotných tabulek Hepldesku. Jedná se především o konvence pojmenovávání polí a jednotlivých tabulí, aby nedocházelo k nestejnorodosti v pojmenování, např. konvence pro pojmenování tabulek je název modulu a pak samotný název, který vystihuje účel tabulky. K dalším konvencím patří i pojmenování cizích klíčů a taky existence polí v každé tabulce, kde se uchovává datum vložení, změny záznamu a jiné. Při tvorbě samotné datové struktury byly využity znalosti z předmětu *Databázové systémy* a normální formy při tvorbě modelu. Pro samotnou realizaci databázové struktury byly použity dva databázové servery. Databázový server *PostgreSQL*, který používá aplikace zadavatele jako datové uložiště. Protože bylo potřeba vytvořit totožnou datovou strukturu i pro webové rozhraní, byl zvolen databázový systém MySQL, a to především pro svou velkou rozšířenost a popularitu u webových aplikací. Jedná se o relační databáze, proto bylo potřeba vytvořit relace mezi jednotlivými tabulkami, aby byla zajištěna datová integrita. Pro funkčnost aplikace bylo nutné vytvořit tzv. *triggery* (procedury, které proběhnou při určité události nad tabulkou např. událost typu *update*). Jednalo se jak o *triggery*, které se musely přebírat z již existujících konvencí tabulek ostatních modulů aplikace (např. úprava polí s datem vložení a změny záznamu, atd.), tak i dalších nových *triggerů,* (např. plnění replikací, ukládání historizujících záznamů atd). Na závěr bylo třeba optimalizovat výkon databáze a to pomocí indexů. Přes indexy se urychluje běh dotazu, protože pomocí indexů se již pracuje jen se záznamy, které odpovídají údajům, nad kterými sql dotaz probíhá. Bylo třeba určit, která pole oindexovat. V průběhu vývoje aplikace byly indexy doplňovány. U textových polí, kde se předpokládá průběh vyhledávání, byly vytvořeny *fulltext* indexy.

Všechny výše zmíněné operace byly realizovány při tvorbě databázové struktury na *Postgre* serveru, pro vytvoření databázové struktury na webovém rozhraní byl využit nástroj pro generování kódu, kdy se na základě struktury *Postgre* vygenerovala struktura jednotlivých tabulek pomocí SQL příkazů *create table*.

Teď bude následovat krátký seznam databázových tabulek vytvořených v datovém návrhu, doplněn stručným popisem na co jsou určeny.

5.2.2 Seznam a popis tabulek databázové struktur

Zde je uveden seznam databázových tabulek HelpDesk vytvořených při návrhu databázové struktury. Ke každé tabulce je přidán krátký popis, k čemu je tabulka určena.

1. *Helpdesk* – jedná se o hlavní tabulku aplikace, od které se vše odvíjí. Tato tabulka obsahuje všechny údaje o určitém požadavku např. samotné znění, datum splnění atd. Vyskytuje se zde velká řada cizích klíčů, které směřují nejen do číselníků a souvisejících tabulek, ale také navíc obsahují vazby do již existujících tabulí systému *Goofer*.

2. *Helpdesk_categories* – každý požadavek lze kategorizovat do různých skupin, a to například kategorie *chyba*, *dotaz* atd. Jedná se o vazební tabulku.

3. *Helpdesk_categories_language* – v této tabulce jsou uloženy samotné názvy kategorií, obsahuje odkaz na tabulku *helpdesk_categories* a tabulku *language*, složením cizích klíčů těchto dvou tabulí se dotazuje do této tabule. Tabulka je určena pro variaci jazykových mutací systému.

4. *Helpdesk_module* – tabulka obsahuje údaje o tom, k jakému modulu se požadavek vztahuje. Tabulka je určena pro číselník modulů.

5. *Helpdesk_module_language* – tabule obsahuje jazykové mutace pro názvy a popisy modulů v číselníku.

6. *Helpdesk_history* – tato tabulka uchovává v historii změny nad tabulkou *helpdesk*, tyto změny se propisují pomocí *triggeru*.

7. *Helpdesk_phase* – tabulka obsahuje číselník fází požadavku a grafické zobrazení fáze.

8. *Helpdesk_phase_language* – v tabulce jsou uloženy jazykové mutace pro všechny slovní popisky a názvy fází požadavku.

9. *Helpdesk_phase_type* – tabulka rozřazuje fáze podle typu požadavku.

10. *Helpdesk_phase_type_language* – v tabulce jsou uloženy jazykové varianty pro typy fází požadavku.

11. *Helpdesk_phase_lifecycle* – jedná se o tabulku, která určuje závislost fáze na přechod na jinou fázi. Určuje návaznost jednotlivých fází (životní cyklus požadavku).

12. *Helpdesk_phase_history* – tato tabule uchovává historické údaje o tom, v jakých stavech se požadavek nacházel, a jaký uživatel tyto změny provedl.

13. *Helpdesk_priority* – zde jsou uloženy nadefinované priority, s jakými se má požadavek řešit. Požadavek se řeší podle toho, jakou má priorita váhu.

14. *Helpdesk_priority_language* – různojazyčné popisky priorit požadavku.
15. *Helpdesk_product* – jedná se o vazební tabulku na tabulku produktů, protože

požadavek se může vztahovat na určitý produkt.

16. *Discussion _forum* – v tabulce jsou uložena diskusní fóra, vztahující se k požadavku v Helpdesku. Tabulka nezačíná prefixem helpdesk, protože tabulka nebude využita pouze pro aplikaci Helpdesk, ale pro všechna diskusní fóra v systému.

17. *Discussion* – zde jsou uloženy jednotlivé příspěvky a odpovědi v danému fóru. V této tabulce existuje pole, které odkazuje zpět na záznam v tabulce *Discussion*.

18. *Users* – tato tabulka vznikla pro zřízení uživatelů, kteří mohou přistupovat k internetové aplikaci. Je zde návaznost na tabulku *Contacts* a *Sman*.

19. *User_group* – zde jsou evidovány skupiny uživatelů, každý uživatel je zařazen do určité skupiny.

20. *User_in_group* – jedná se o vazební tabulku mezi tab. *user* a tab. *user_group*, pomocí této tabulky se zjišťuje, v jakých skupinách se uživatel nachází.

21. *User_project* – požadavky jsou vztaženy na určité projekty. Tabulka tvoří vazbu mezi uživateli, skupinami a projekty, na které jsou uživatelé přiřazeni.

22. *User_role* – zde jsou uložena práva (právo zápisu, čtení, ...) jednotlivých uživatelů anebo skupin uživatelů.

23. *User_role_language* – obsahuje různojazyčná vyjádření popisů práv.

24. *Replication_table* – tabulka určena pro replikaci, plní se pomocí *triggerů* umístněných nad jednotlivými tabulkami anebo metodou ve vrstvě DAO.

Tabulky využivané v aplikaci Helpdesk v již existující v systému:

1. *Order_in* – tabulka uchovává údaje o objednávkách, požadavek může ukazovat na určitou objednávku

2. *Project_order* – tabulka projektů, ke kterým se požadavek Helpdesku vztahuje

3. *Contacts* – tabulka je navázaná na uživatele, každý webový uživatel se musí nacházet v této tabulce

4. *Sman* – jedná se o speciální tabulku uživatelů

5. *Large_object* – tabulka odkazuje na objekty (dokumenty) navázané např. na produkt, nyní bude využita i pro Helpdesk

5.3 Vrstva *Database Access Object* a její funkcionalita

Vrstva *Database Access Object* (dále jen DAO), zajišťuje veškerou komunikaci mezi databází a webovou aplikací Helpdesk. Vrstva má zajistit nezávislost aplikace na databázovém serveru. Vrstva DAO má následující funkcionality:

- Připojení k databázi
- Zpracovávání SQL příkazů v transakcích nebo *ad-hoc*
- Inicializace *Business Object* (dále jen BO)

- Zajišťování datové integrity
- Realizace zápisu změnových záznamů do replikační tabulky
- Metody s předdefinovanými SQL dotazy

Pomocí této vrstvy již nedochází k přímé komunikaci mezi databází a výkonovou logikou aplikace. Pokud výkonová logika nepracuje přímo s databází, je možné zaměnit databázový server a vytvořit novou DAO vrstvu, upravenou na nově zvolený databázový systém a aplikace by měla bez další zásahu pracovat.

DAO vrstva musí udržovat datovou integritu dat na straně MySQL. MySQL tuto vlastnost nepodporuje, nejsou zde přesně definovány restrikce mezi záznamy v tabulkách, navíc může dojít ke smazání záznamů i když má podřazené záznamy (např. záznam *objednávka* a *položky objednávky*). DAO tuto vlastnost s *Postgre* supluje za MySQL.

Vrstva DAO může být dále členěna na další části:

- Část zajišťující připojení k databázi a realizující zapouzdření funkcí PHP pro komunikaci s databází a vykonávání SQL dotazů
- Část, kde se provádí volání zapouzdřených funkcí PHP a samotné zpracování dotazů na databázi
- Poslední část se týká předávaných dat prostřednictvím BO

5.3.1 Popis tříd pro komunikaci s databází

Byla vytvořena statická třída *QueryExecutor*, která do svých metod zaobalila PHP funkce pro zpracování SQL dotazu. Zaobalením interních funkcí PHP pro MySQL se docílilo toho, že už nebudou pro zpracování volány přímo funkce PHP, ale metody této třídy. Jedná se o nezbytný krok k tomu, aby mohlo nastat oddělení zbývajících vrstev od databáze. Dále zde budou zmíněny jednotlivé metody třídy.

Metoda *Execute (*obr. 34) je nejdůležitější metodou třídy *QueryExecutor*, která provede vykonání samotného SQL dotazu. Uvnitř metody nastane vytvoření objektu databázového připojení, přičemž objekt připojení k databázi je vytvořen pouze jednou, při volání jinou metodou je už objekt předáván. Při zpracování dotazu je přednostně zahájeno transakční zpracování dotazu. SQL dotaz je do metody dodán prostřednictvím třídy *sqlQuery*, prostřednictvím její instance v parametru metody. Návratovou hodnotou metody je pole záznamu vrácené z databáze na základě vstupního dotazu. Metoda zaobaluje funkci PHP *mysql_fetch_array*.

```
public static function execute($sQuery){
  $trans = Transaction::getTransaction();
  if(!$trans){
     $connection = new Connection();
  }else{
    $connection = $trans->getConnection();
  }
  $query = $sQuery->getQuery();
  $result = $connection->executeQuery($query);
  if(!$result){
    throw new Exception(mysql_error());
  }
  $i=0;
  $tab = array();
  while ($row = mysql_fetch_array($result)){
    $tab[$i++] = $row;
  }
  mysql_free_result($result);
  if(!$trans){
    $connection->close();
  }
  return $tab;
}
```

Obr. 34 Metoda Execute pro vykonání dotazu

Další metody třídy *QueryExecutor* jsou velmi podobné, liší se především v návratové hodnotě metody a v typu zaobalené interní funkce PHP.

Seznam dalších metod třídy *QueryExecutor*:

- Metoda *fetchField* obsahuje funkci *mysql_fetch_field*, návratovou hodnotou je pole hodnot nesoucí informaci o sloupcích tabulky
- Metoda *numField* zaobaluje funkci *mysql_num_field* a vrací hodnotu počtu sloupců
- Metoda *fieldLen* reprezentuje funkci *mysql_field_len* a vrací hodnotu délky polí
- Metoda *getResult* vrací přímo výsledek dotazu z databáze
- Metoda *numRow* reprezentuje funkci *mysql_num_rows* a vyžádá počet řádků na základě dotazu, metoda se využívá např. pro stránkování

V PHP existuje ještě velká řada funkcí pro komunikaci s MySQL databází. Uvnitř třídy *QueryExecutor* je zaobaleno jen několik nejvíce využívaných pro zpracování a komunikaci s databází. Pokud by bylo třeba i dalších funkcí v PHP, lze třídu jednoduše rozšířit o potřebné funkce.

Potřebnou součástí vrstvy bylo vytvoření připojení k databázi. Pro tento účel byla zavedena třída *Connection*, která zajišťuje připojení k databázi. Využívá dalších statických tříd, nezbytných pro připojení k databázi. První je statická třída *ConnectFactory,* zde je zaobalena interní funkce PHP *mysql_connect* pro vytvoření připojení.

Pro vytvoření připojení je třeba přihlašovacích údajů k databázi, tyto informace nese statická třída *ConnectionProprety*. Obsahuje následující vlastnosti, které jsou volány třídou *ConnectFactory*.

- *host* – vlastnost obsahuje adresaci k serveru
- *user* – uživatelské jméno pro připojení
- *password* – heslo pro připojení daného uživatele
- *database* – název databáze pro připojení

Vrstva podporuje transakční zpracování dotazu, v případě, že je pro daný typ tabulky podporován. Transakční provedení dotazu zajišťuje třída *Transaction*. Třída ve svém konstruktoru využívá třídy *Connection,* pomocí které vytvoří připojení. Uvnitř konstruktoru je vyvolána transakce příkazem *begin*. Třída má další metody, které řídí transakční zpracování dotazu.

- Metoda *commit* – provede se při korektním zpracování všech dotazů v rámci transakce
- Metoda rollback – vyvolává se případě, kdy dojde k chybě vykonání dotazu v transakci a všechny provedené změny v rámci transakce budou vráceny zpět

Poslední třídou na úrovni přímé komunikace s databází je *sqlQuery*. Ve vrstvě DAO se využívá tzv. parametrického zadávání SQL dotazů, příklad uveden na obr. 35. Třída zajišťuje nahrazení parametrů dotazu ve správném pořadí a nahradí příslušný parametr hodnotou.

```
$sql = 'SELECT * FROM `helpdesk_phase` WHERE `helpdesk_phase_id` = ?';
$sqlQuery = new SqlQuery($sql);
$sqlQuery->set($id->getValue());
$sqlQuery->getQuery();
```

Obr. 35 Zpracování parametrického SQL dotazu

Parametr je reprezentován symbolem *?* a třída *sqlQuery* vykonává nahrazení parametrů přes metodu *set*. Správné pořadí použití metody *set* je zajištěno pomocí částečného generování DAO vrstvy, kde se generují třídy a jejich metody, které mají vygenerované dotazy a přesnou posloupnost parametrů nastavovaných prostřednictvím těchto metod.

Výsledný SQL dotaz vrací metoda *getQuery,* dotaz je bez parametru a obsahuje příslušné hodnoty. Dotaz se zpracovává přes metodu *Execute* třídy *QueryExecutor.*

5.3.2 Popis třídy pro vykonávání dotazů a inicializaci BO

Třída obsahuje přímo parametrizované SQL dotazy a zajišťuje inicializaci BO anebo s ní pracuje. Tato třída je generována pomocí nástroje pro generování. Každá tabulka v databázi má vygenerovanou třídu pod svým názvem a prostřednictvím metod tříd se realizuje inicializace BO, popřípadě je zpracovává a data z nich předává do databáze. Uvedená část DAO vrstvy je typickým příkladem využití generátoru. Třídy obsahují pořád stejné metody, jen se mění obsah metod, kde se pracuje s údaji o dané tabulce

Název třídy pro zpracování a inicializaci BO se tvoří z prefixu názvu tabulky a *MySQLDAO*. Dále bude uveden příklad a popis konkrétní třídy pro tabulku *helpdesk_phase* obr. 36.

```
class HelpdeskPhaseMySqDAO  extends DAO
{
  public function HelpdeskPhaseMySqlDAO()
  {
    parent::__construct();
  }
...
}
```

Obr. 36 Třída pro inicializaci BO HelpdeskPhaseMySQLDAO

Třída *HelpdeskPhaseMySQLDAO* je potomkem třídy *DAO*. Třída *DAO (*obr. 37), obsahuje metody, které jsou obecné pro všechny tabulky. Je přímým potomkem třídy *QueryExecutor* a dědí všechny vlastnosti pro vykonávání dotazů.

```
class DAO extends QueryExecutor
{

   public function __construct()
   {
    ...
   }

   function getGuid()
   {
    ...
   }

   function writeToReplication($tableName, $Action, $table)
   {
   ...
   }

   public function getPrimaryKey($type, $objTab)
   {
    ...
   }

}
```

Obr. 37 Struktura třídy DAO

Metody třídy *DAO*

- Metoda *getGuid* je určena pro generování jedinečného klíče, který slouží jako primární klíč záznamu v tabulce
- Metoda *writeToReplication* byla již zmíněna v kapitole synchronizace dat
- Metoda *getPrimaryKey* vrací pole obsahující primární klíč tabulky, v případě složeného klíče všechny názvy klíčů. Metoda má dva parametry, první parametr *type* určuje, v jaké formě má být hodnota názvu primárního klíče předána. Varianty forem jsou tři − v databázové formě: *název_pole* (pouze text), ve formě proměnné třídy (objekt pole primárních klíčů), formě názvu proměnné třídy (*nazevPole* pouze text.)

Výše bylo zmíněno, jaké vlastnosti dědí třída *HelpdeskPhaseMySQLDAO* od rodičovské třídy *DAO*. Dále budeme pokračovat s popisem metod třídy *HelpdeskPhaseMySQLDAO*.

Metoda *loadFilter* (na obr. 38) na základě vstupního parametru *filter* provede vytvoření *where* fráze SQL dotaz. Parametr *filter* je objekt, který bude popsán níže v této podkapitole. Po zpracování SQL dotazu metoda vrátí list BO.

```
public function loadFilter($filter)
{

  $where = $filter->getFilter();
  $sql = 'SELECT * FROM `helpdesk_phase`'.$where;
  $sqlQuery = new SqlQuery($sql);
  $arrValue = $filter->whereValue;

  foreach ($arrValue as $key => $val)
  {
    $sqlQuery->set($val->getValue());
  }
  return $this->getList($sqlQuery);
}
```

Obr.38 Metoda LoadFilter vrací pole BO

Za pomocí metody *readRow* se provede inicializace BO. *Business Object* je pojmenován po tabulce, nad kterou je dotaz vykonán. Návratová hodnota metody představuje jeden záznam v tabulce.

```
protected function readRow($row)
{
  $helpdeskPhase = new HelpdeskPhase();

  $helpdeskPhase->helpdeskPhaseId->setValue($row['helpdesk_phase_id']);

  $helpdeskPhase->createDate->setValue($row['create_date']);
  ...

  return $helpdeskPhase;
}
```

Obr. 39 Metoda readRow inicializující BO

Metoda *getList* (na obr. 40) vrací pole BO, které reprezentují jednotlivé řádky tabulky dle zadaných podmínek.

```
protected function getList($sqlQuery)
{
  $tab = $this->execute($sqlQuery);
  $ret = array();

  for($i=0;$i<count($tab);$i++)
  {
          $ret[$i] = $this->readRow($tab[$i]);
  }

  return $ret;
}
```

Obr. 40 Metoda getList vracející pole BO

Metoda *insertRow* (obr. 41) slouží pro vložení záznamu do databáze prostřednictvím BO. Metoda má jeden parametr *helpdeskPhase* a reprezentuje BO pro tabulku *helpdesk_phase*. Pro každou tabulku je vygenerována vlastní třída, *nazev_tabulkyMySQLDAO*, která je přizpůsobena konkrétní tabulce. Všechny dotazy, BO, názvy proměnných se řídí tabulkou, kterou jsou určeny. Přes vstupní BO se provede nahrazení parametru v SQL dotazu na hodnoty, které obsahuje BO. U primárních klíčů se provede automatické přidání jedinečných hodnot, aby nedošlo k duplicitám. Provede se vložení záznamu do tabulky *helpdesk_phase*. Po úspěšném

vložení záznamu do tabulky nastane vytvoření odkazu na vložený záznam v replikační tabulce, aby mohlo v rámci synchronizace nastat přenesení záznamu do *Postgre* databáze. Na závěr metody se vrátí pole obsahující hodnotu primárního klíče (v případě složených klíčů, hodnoty všech klíčů).

Vstupní parametr je inicializován ve vyšší vrstvě v aplikační logice, kde je metoda třídy volána.

```
public function insertRow($helpdeskPhase)
{
  $sql = 'INSERT INTO `helpdesk_phase` (`helpdesk_phase_id`, `create_date`,
`change_date`, `phase_icon`, `helpdesk_phase_type_id`, `phase_order`,
`phase_name`)
  VALUES (?, ?, ?, ?, ?, ?, ?)';
  $sqlQuery = new SqlQuery($sql);

  $helpdeskPhase->helpdeskPhaseId->setValue($this->getGuid());
  $helpdeskPhase->helpdeskPhaseTypeId->setValue($this->getGuid());
  $sqlQuery->set($helpdeskPhase->helpdeskPhaseId->getValue());
  $sqlQuery->set($helpdeskPhase->createDate->getValue());
  $sqlQuery->set($helpdeskPhase->changeDate->getValue());
   ...

  if($this->execute($sqlQuery))
  {
  $this->writeToReplication("helpdesk_phase", "I", $helpdeskPhase);
  }
  $arrPkey['helpdeskPhaseId'] = $helpdeskPhase->helpdeskPhaseId->getValue();
  $arrPkey['helpdeskPhaseTypeId'] = $helpdeskPhase->helpdeskPhaseTypeId-
>getValue();

  return $arrPkey;

}
```

Obr. 41 Metoda insertRow pro vložení záznamu do databáze

Dále bude uveden seznam s popisem dalších metod třídy *HelpdeskPhaseMySQLDAO (MySQLDAO).*

- Metoda *load* provede načtení pouze jednoho záznamu z tabulky na základě hodnoty primárního klíče. Vstupní parametr *Id* reprezentuje objekt sloupce primárního klíče. Metoda je podobná metodě *loadFilter*.
- Metoda *checkRowFilter* provede test, zda při zadaných omezujících podmínkách existují v tabulce záznamy vyhovující těmto podmínkám. Vstupní parametr *filter* je objekt obsahující omezující podmínky. Metoda vrací hodnotu typu *boolean* (*treu* při nalezení záznamů, *false* nebyl nalezen žádný záznam).
- Metoda *queryAll* načte celý obsah tabulky bez jakéhokoliv omezení. Návratovou hodnotou je list BO.
- Metoda *updateMulti*, jak je z názvu patrné, vykoná editaci záznamu v tabulce. Vstupním parametrem je obdobně jako u metody *insertRow* inicializovaný BO. Změna se provede na všech sloupcích záznamu v tabulce. Je zde parametrizovaný SQL příkaz *update*, který se následně provede. Údaje o změně určitého záznamu jsou propsány do replikační tabulky.
- Metoda *delete* zajišťuje smazání záznamu z tabulky. Vstupním parametrem je opět BO a na základě primárního klíče ze vstupního parametru se provede smazání záznamu z tabulky. Po úspěšném smazání záznamu nastane zápis do replikační tabulky o této změně.

U metod *updateMulti* a *delete* je třeba zajistit datovou integritu. Datovou integritu zajišťuje uvnitř zmíněných metod instance třídy *Relationship*. Třída *Relationship* obsahuje dvě metody.

První metoda *checkRelation* třídy Relationship kontroluje, zda nad měněným záznamem

existují integritní omezení (např. *restrict, set null, cascade*). Je-li nalezeno integritní omezení, je běh změny záznamu zastaven, a nebo je vykonán dle typu integritního omezení. Vykonání integritního omezení provede druhá metoda *relation* třídy *Relationship*. Metoda *Relation* provede rekurzivní volání provedené změny nad všemi záznamy souvisejícími s integritním omezením. V případě integritního omezení *cascade* se změnovou operací *delete* dojde pomocí metody *relation* ke smazání všech souvisejících záznamů. Na obr. 42 je uvedena část kódu, která zajišťuje u metod *delete* a *updateMulti* integritní omezení.

```
if(Relationship::checkRelation($helpdeskPhase, 'D'))
{
  $efect = $this->execute($sqlQuery);
  if($efect)
  {
    $this->writeToReplication("helpdesk_phase", "D", $helpdeskPhase);
  }
}
```

Obr. 42 Část kodu pro zajištění datové integrity

5.3.3 Popis vlastností a metod třídy pro BO

Business object jsou třídy, které jsou generovány pomocí šablony nástrojem pro generování. Každá tabulka v databázi má vygenerovanou třídu pro *business object*, která vyjadřuje strukturu dané tabulky. Třídy jednotlivých BO reprezentují jednotlivé tabulky ve vyšších vrstvách a umožní práci s nimi. BO třídy jsou pojmenovány podle názvu tabulky, např. tabulka *helpdesk_phase* má svůj obraz v BO třídě HelpdeskPhase (na obr. 43).

```
class HelpdeskPhase extends Table
{
var $helpdeskPhaseId;
var $createDate;
var $changeDate;
var $phaseIcon;
var $helpdeskPhaseTypeId;
var $phaseOrder;
var $phaseName;¨

function construct()
{

  $this->helpdeskPhaseId = new Field(array(0 => 'helpdesk_phase_id',
  'varchar(32)','32', '10'));
  $this->createDate = new Field(array(0 => 'create_date', 'datetime', '0', '0'));
  $this->changeDate = new Field(array(0 => 'change_date', 'datetime', '0', '0'));
  $this->phaseIcon = new Field(array(0 => 'phase_icon', 'smallint', '0', '0'));
  $this->helpdeskPhaseTypeId = new Field(array(0 => 'helpdesk_phase_type_id',
  'varchar(32)', '32', '10'));
  $this->phaseOrder = new Field(array(0 => 'phase_order', 'smallint', '0', '0'));
  $this->phaseName = new Field(array(0 => 'phase_name', 'varchar(32)','32','0'));

  $this->TableName = 'helpdesk_phase';
}
}
```

Obr. 43 BO pro tabulku helpdesk_phase

Třída pro BO obsahuje vlastnosti, které představují jednotlivé sloupce tabulky. Vlastnosti (sloupce) jsou instancí třídy *Field*. Třída pro BO obsahuje tolik vlastností typu Field, kolik má tabulka sloupců. Třída *Field* (obr. 44) tvoří strukturu jednotlivých sloupců tabulky. Třída Field obsahuje následující vlastnosti:

- *Name* – vyjadřuje název sloupce
- *Type* – zde je uložen datový typ sloupce
- *Size* – určuje velikost sloupce (např. *varchar(20)*)
- *isPrimary* – nabývá hodnot 1 když sloupec je primární klíč, 0 standardní sloupec
- *Value* – hodnota daného slupce

Paremetrem konstruktoru třídy *Field* je pole, které obsahuje všechny výše uvedené vlastnosti, kromě *Value*. Pro inicializaci vlastnosti *Value* se používá následující metody třídy *Field*. Pro inicializaci vlastnosti *Value* se používá ve většině případu metoda *setValue*, u hodnot typu *datum* má využití metoda *setValueDate*. Pro získání hodnoty jednotlivých vlastností (sloupců) se využívá metoda *getValue*, u varianty vlastnosti typu *datum*, vrací metoda *datum* ve správném fromátu.

```
class Field
{
var $Name;
var $Type;
var $Size;
var $isPrimary;
var $Value;

function construct($param = array())
{
  $this->Name = $param[0];
  $this->Type = $param[1];
  $this->Size = $param[2];
  $this->isPrimary = $param[3];
}

public function getValue()
{
  return $this->Value;
}

public function getValueDate()
{
  return date("d.m.Y", strtotime($this->Value));
}

public function setValue($val)
{
  $this->Value = $val;
}

public function setValueDate($val)
{
  $val = date("Y-m-d", strtotime($val));
  $this->Value = $val;
}
}
```

Obr. 44 Ukázka třídy Field

Při vytvoření instance třídy HelpdeskPhase (BO), vznikne objekt, který je obrazem struktury tabulky. Objekt (BO) je třeba naplnit hodnotami jednotlivých slouců. K této inicializaci dojde buď při načítání dat z databáze a nebo od uživatele, prostřednictvím komponent. Třída *HelpdeskPhase* je potomkem třídy *Table*. Třída *Table* obsahuje vlastnosti a metody, které jsou obecné pro BO všech tabulek. Třída Table obsahuje následující vlastnosti a metody:

- Vlastnost *TableName* – představuje název tabulky, pro kterou je BO určen
- Vlastnost *Relation* – obsahuje pole tabulek, které jsou v relaci s tabulkou BO
- Metoda *setRelation* – provede vytvoření pole *Relation* a naplní ho hodnotami

5.3.4 Popis vlastností a metod třídy Filter

Třída *Filter (obr. 45)* byla vyvinuta za účelem tvorby omezujících podmínek v SQL dotaze. Na základě jejich metody a vlastností se doplní *where* fráze dotazu. Instance třídy *Filter* je vstupním parametrem funkcí po inicializaci BO ve vrstvě DAO např. *loadFilter*, *load*, *updateMulti*. Pomocí třídy *Filter* se nastavuje i *order by* fráze, kterou se řídí, podle jakých sloupců tabulky bude výsledek dotazu řadit. Další činností třídy je umožnit stránkování záznamu. V dotazu se pomocí třídy *Filter* nastaví, od jaké pozice budou záznamy vráceny a

v jaké rozsahu.

Dále bude následovat seznámení s jednotlivými vlastnostmi a metody třídy *Filter*. Omezující podmínky pro výběrový dotaz jsou uloženy ve vlastnostech *arrCondition* , *arrCondGroups* a *arrSorting*. U všech tří případů se jedná o pole, které obsahuje instance třídy *FilterItem*. Třída *FilterItem* byla vytvořena pro jednotlivé položky filtru, s kterými se pracuje uvnitř třídy *Filter*. Třída *FilterItem* udržuje následující vlastnosti:

- *Column* – je instance třídy *Field*, tato vlastnost obsahuje veškeré informace o sloupci tabulky
- *Operator* – obsahuje operátor pro porovnání s hodnotou sloupce (není-li hodnota operátoru zadána, uvede se defaultně operátor rovnosti)
- *logOperator* – zde je uvedeno, s jakým logickým operátorem (AND, OR) byla podmínka do třídy *Filter* přidána
- *Group* – reprezentuje, do jaké skupiny byla podmínka zařazena
- *SortType* – obsahuje typ řazení záznamu v SQL dotaze (ASC, DESC)

```
class Filter
{
var $arrCondition = array();
var $arrCondGroups = array();
var $arrSorting = array();
var $pageLimit;

function construct()
{
...
}

function addCondAnd($cond, $operator = '', $group = '')
{
  $condition = new FilterItem($cond, 'AND', $operator, $group);

  if($group == '')
  {
    $this->arrCondition[] = $condition;
  }
  else
  {
    if(!isset($this->arrCondGroups[$group]->groupName))
    {
      $this->createGroup($group);
    }
    $this->arrCondGroups[$group]->addItem($condition);
  }
}

function addCondOr($cond, $operator = '', $group = '')
{
...
}

...
}
```

Obr. 45 Část struktury třídy Filter

Přidání omezující podmínky se provede metodou *addCondAnd* (obr. 45) třídy *Filter*. Metoda *addCondAnd* přidá novou omezující podmínku s logickým operátorem *and* do proměnné *arrCodition*. Metoda má tři vstupní parametry. První parametr *cond* je objekt třídy *Field*, který obsahuje všechny potřebné informace o sloupci tabulky (název, hodnotu, atd.). Parametr *operator* je nepovinný a jeho prostřednictvím se zadává operátor pro srovnání hodnot. Není-li parametr zadán, použije se předem nastavený operátor rovnosti. Přes parametr *group* lze omezující podmínku zařadit do skupiny. Přes skupinu omezujících podmínek se zajišťuje priorita vykonávání skupin podmínek (jedná se o uzávorkování podmínek v rámci dotazu). Skupiny se ukládají do vlastnosti *arrCondGroup* a vytváří se metodou *createGroup*. Obdobným způsobem se vytvoří podmínka s logickým operátorem přes metodu *addCondOr*.

Řazení záznamů zajišťují metody *addSortASC* a *addSortDESC*, které přidají objekt instance *FilterItem* do pole vlastnosti *arrSorting*. Objekt filtrační položky nastaví vlastnost SortType na hodnotu dle způsobu řazení záznamů (hodnoty *asc* a *desc*).

Poslední vlastnost *pageLimit* třídy *Filter* nabízí možnost stránkování záznamů. Vlastnost *pageLimit* je instancí třídy *PageLimit*, která nabízí následující vlastnosti a metody:

- vlastnost *limit* udává maximální počet záznamů, které má dotaz vrátit
- vlastnost *countPage* – na základě této hodnoty se určuje, od jakého záznamu má být výsledek dotazu vrácen
- metoda *getLimit* sestaví z výše uvedených vlastností frázi *limit* SQL dotazu pro omezení počtů záznamů

Inicializace vlastnosti *pageLimit* (obr. 46) třídy *Filter* provede metoda *setLimit*. Metoda *setLimit* pracuje se dvěma vstupními parametry. Pomocí parametru *countPage* se nastaví hodnota záznamu, od kterého má být výsledek vrácen. Druhý parametr *limit* je nepovinný, není-li zadán, bere se hodnota z konfigurační třídy, kde je nastaven limit počtu záznamů na stránce. V případě zadání parametru *limit* se jeho hodnota bere přednostně.

```
function setLimit($countPage, $limit = '')
{
  $this->pageLimit = new LimitPage($countPage, $limit);
}
```

Obr. 46 Inicializace vlastnosti pageLimit

Prostřednictvím třídy *Filter* se realizuje dotvoření SQL dotazu v DAO vrstvě. Samotné vytvoření části dotazu zajišťuje metoda *getFilter* (obr. 47). Metoda vrací pouze ty části SQL dotazu, které jsou ve třídy nastaveny. Není-li nastavena žádná omezující podmínka, řazení nebo stránkování, je vrácen prázdný výsledek.

```
function getFilter()
{
  $filter = $this->getWhere().$this->getSort().$this->getLimit();

  return $filter;
}
```

Obr. 47 Metoda getFilter vracející SQL dotaz

Výše uvedené třídy jsou nejdůležitějšími třídami vrstvy DAO a zajišťují hlavní činnost vrstvy. Vývoj DAO vrstvy byl náročnou částí realizace aplikace Helpdesk. Bez této vrstvy by nebylo možné oddělit aplikační vrstvu od datové. Je nezbytnou součásti aplikace a tvoří alfa-omegu s realizací všech operací s databází. Při vývoji vrstvy bylo třeba důkladně navrhnout architekturu jednotlivých tříd, aby byly zajištěné veškeré operace, které požaduje aplikace Helpdesk na databázi.

5.4 Vrstva *Business Inteligence* a *Facade*

Součástí vrstvy jsou přímo jednotlivé části aplikace Helpdesk, které zde budou představeny v grafické podobě. V této části bude zahrnuta i vrstva GUI, při prezentování jednotlivých činností aplikace. Vrstva *Business Inteligence* bude dále v této podkapitole označována jako aplikační vrstva.

Aplikační vrstva je řešena částečně podle MVC (*Model View Controller*) návrhového vzoru, oddělujícího uživatelské rozhraní od logiky aplikace. Jednotlivé části aplikace byly rozděleny na komponenty, které obsahují aplikační logiku pro svou činnost. Uvnitř komponenty se mohou nacházet subkomponenty, jedná se o komponenty menšího rozsahu, které doplňují určité činnosti (např. stránkování, progresbar, atd.) a mají vlastní grafický vzhled. Jednotlivé komponenty komunikují s DAO vrstvou přes tzv. *Facade*. *Facade* je nástavba nad vrstvou

DAO. Prostřednictvím *Facade* tříd jsou volány jednotlivé funkce vrstvy DAO (např. načtení dat z tabulky, atd.) a slouží ke komunikaci mezi DAO a komponentou. Ve *Facade* se řeší i propojení jednotlivých tabulek v rámci objektu. Při vývoji *Facade* tříd je využito vzoru *Singleton*. Jak je již z názvu je patrné, jedná se o návrhový vzor, které má zajistit, aby existovala pouze jedna instance určité třídy. Příklad *facade* třídy s využitím vzoru *singleton* lze vidět na obr. 48. Třída má konstruktor deklarovaný jako *private*, ten může být volán pouze uvnitř třídy. Pro vytvoření objektu se využívá statická metoda třídy *getInstance*. Pokud neexistuje žádná instance třídy, je volán konstruktor a vrací se objekt do statické proměnné *Instance*. V opačném případě metoda *getInstance* vrací odkaz na statický datový člen *instance*. Prostřednictvím tříd *facade* se vykonávájí dotazy z vrstvy DAO. V konstruktoru *facade* třídy se vytvoří instance třídy DAO určité tabulky a prostřednictvím jednotlivých metod se k objektu DAO přistupuje. Dotazování do objektu DAO se provede například metodou *getPhaseList*, kde se provede vyvolání metody *loadFilter*, která vrátí pole BO obsahující jednotlivé fáze helpdesku. S metodami *facade* tříd se pracuje uvnitř jednotlivých komponent. Ve třídě *facade* lze vytvořit i metody pro volání libovolných funkcí z vrstvy DAO různých tabulek.

```
class helpdeskPhaseFacade{

   private static $instance = false;
   var $phase;

   private function __construct()
   {
      $this->phase = DAOFactory::getHelpdeskPhaseDAO();
   }

   public static function getInstance()
   {
      if(self::$instance === false)
      {
        self::$instance = new helpdeskPhaseFacade();
      }
      return self::$instance;
   }

   function getPhaseList($filter)
   {
      return $this->phase->loadFilter($filter);
   }

  ...
}
```

Obr. 48 Ukázka facade třídy pro helpdeskPhase

Aplikační vrstva je tvořena jednotlivými komponenty. Pro každou komponentu je vytvořená vlastní třída, která obsahuje výkonovou logiku pro zajištění činnosti komponenty. Komponenty komunikují s vrstvou DAO prostřednictvím tříd *facade*, kde se vyvolávájí jednotlivé metody vrstvy DAO. V rámci aplikační vrstvy se již pracuje jen s BO. Pro třídy komponent bylo nutné zavést strukturu, podle které je třeba se řídit a dodržovat. Do předepsané struktury patří dvě metody, které jsou určeny pro zobrazovací účely. Každá třída komponenty obsahuje metodu *displayTempl*, která zajistí zobrazení šablony (zobrazení grafického rozhraní) napsané pro danou komponentu. Nabývá-li komponenta více stavů a pro každý stav je jiné grafické zobrazení, jsou uvnitř metody *desplayTempl* i podmínky, které vyhodnocují, jak šablonu zobrazit. Druhou nezbytnou metodou v určené struktuře je metoda *setTemplValue*, která je určena pro zajištění odkazů na jednotlivé proměnné komponenty, aby mohly být zobrazeny uvnitř šablony. Tato metoda nemusí být vždy v komponentě přítomna, protože u menších komponent může být tato funkce zahrnuta již v metodě *desplayTempl*.

Prostřednictvím komponent se inicializují BO, které jsou posílány přes vrstvu DAO do databáze. Komponenty za pomoci svých metod zajišťují, aby data v BO byla korektní a mohla být uložena do databáze. Uvnitř komponenty se vytváří i instance třídy *Filter*, na základě údajů získaných od uživatele. Komponenta zpracuje údaje od uživatele a následně provede nastavení

objektu třídy *Filter*. Objekt třídy *Filter* je zpracováván uvnitř vrstvy DAO, a na základě jeho nastavených vlastností je vrácen odpovídající výsledek zpět do komponenty. Komponenta následně vrácený výsledek zpracuje a provede zobrazení přes grafické rozhraní.

```
public function displayTempl()
{
   $smarty->display("helpdeskFile.tpl");
}

public function setTemplValue()
{
   $smarty->assign(helpdeskFile,$this->helpdeskFile);
}
```

Obr. 49 Metody displayTempl a setTemplValue

V další části této kapitoly budou zobrazeny grafické ukázky aplikace Helpdesk, které tvoří jednotlivé komponenty. Ke každému grafickému zobrazení bude uveden krátký popis, co komponenta uživateli nabízí a její činnost.

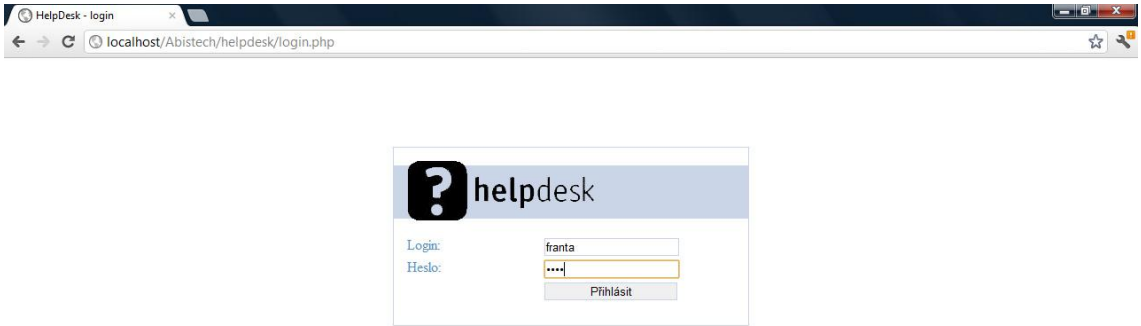
5.4.1 Komponenta pro přihlášení uživatele

Přihlášení uživatele zajišťuje komponenta *authentication*. Uživatel zadá přihlašovací jméno a heslo přes formulář vyobrazený na obr. 50 a potvrzením tlačítka *Přihlásit* spustí proces ověření přihlašovacích údajů. Objekt po přihlášení prostřednictvím metody *loginMethod* ověří, zda je uživatel zaznamenán v databázi uživatelů v tabulce *user*. Před samotným ověřením uživatele musí být zadané heslo zašifrováno, aby mohla nastat shoda s heslem uloženým v databázi. Hesla jsou v databázi uložena a zašifrováná algoritmem *MCRYPT_3DES*. Proces ověření uživatele přes databázi může skončit ve stavech:

- uživatel nebyl nalezen nebo přihlašovací údaje byly špatně zadány uživatelem, neprovede se přihlášení
- uživatel prošel úspěšně procesem ověření a proběhne přihlášení

V případě chybného přihlášení proces ověřování skončí a uživatel je vyzván k novému přihlášení. Komponenta uživatele upozorní na nesprávný pokus o přihlášení do aplikace.

Když se uskuteční úspěšný proces ověření, nastane přihlášení uživatele do aplikace Helpdesk a obrazovka pro přihlášení bude přesměrována na úvodní stránku Helpdesk.



Obr. 50 Printscreen obrazovky pro přihlášení uživatele

Na konci procesu úspěšného ověření se vytvoří instance třídy *User*, která obsahuje všechny potřebné údaje o přihlášeném uživateli a používá se pro identifikaci uživatele v celé aplikaci Helpdesk.

Objekt o uživateli obsahuje informace, do jaké skupiny patří a jaká má práva. Součástí

procesu ověřování je zjištění, do jaké uživatelské skupiny je uživatel přiřazen. Na základě skupiny se zjišťuje role uživatele. Helpdesk podporuje zatím dvě uživatelské role:

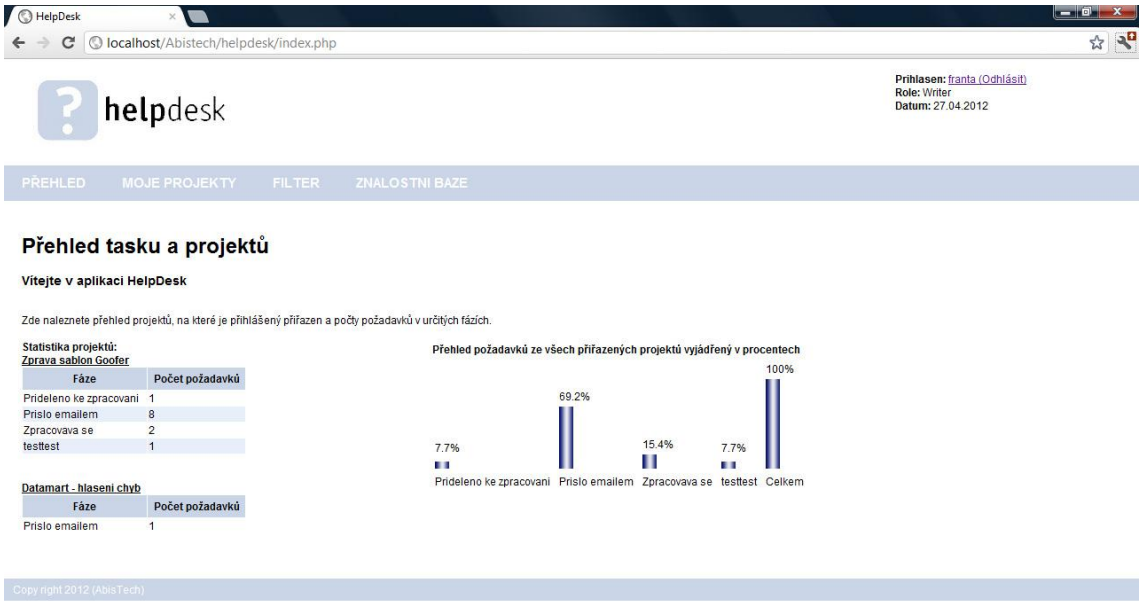
- role *reader* – uživatel s touto rolí může pouze procházet jednotlivé záznamy a nemůže je žádným způsobem ovlivňovat, jedná se pouze o tzv. čtenáře
- role *writter* – tato uživatelská role umožňuje jak jednotlivé záznamy číst a zobrazovat, tak má oprávnění záznamy měnit nebo přidávat

Na základě role obsažené v instanci třídy *User* se ovlivňuje chod aplikace Helpdesk, z hlediska možnosti manipulace s daty. Při úspěšném procesu ověření se zjišťují jednotlivé projekty, na které je uživatel přiřazen. Do objektu uživatele se přiřadí všechny projekty, které jsou u uživatele nastaveny. Uživatel vidí pouze své projekty.

Po úspěšném přihlášení je objekt uživatele uložen do globální proměnné typu *session*. Aby mohlo nastat uložení objektu do proměnné typu *session*, musí nastat *serializace* objektu. Pomocí proměnné typu *session* je možné k objektu přistupovat kdekoliv v aplikaci Helpdesk a udržuje se v paměti po celou dobu práce s aplikací Helpdesk.

5.4.2 Komponenta statistického přehledu

Po úspěšném přihlášení uživatele je tento přesměrován na úvodní stránku aplikace Helpdesk (obr. 51). Průvodní okno aplikace obsahuje statistický přehled, který má pro uživatele informační charakter. Průvodní okno statistiky je tvořeno komponentou *projectStatisticComp*. Komponentou *statistický přehled* je uživateli prezentován přehled o projektech, na které je přiřazen. Pro identifikaci projektu se používá vlastnost objektu *User*. Přehledové tabulky v levé části obrazovky se vztahují ke každému projektu a informují uživatele, kolik požadavků má pod daným projektem zadaných, a v jaké jsou fázi řešení. Druhý způsob prezentace statistických hodnot umožňuje komponenta grafické vyjádření dat pomocí sloupcového grafu. Zatímco údaje v tabulkách se vztahují na jednotlivé projekty uživatele, hodnoty v grafu se vztahují na jednotlivé fáze řešení požadavku ve všech přidělených projektech. Graf vyjadřuje procentuální podíl jednotlivých fází požadavku k celkovému počtu všech požadavků. Úvodní stránka aplikace Helpdesk slouží pouze pro informování uživatele o jeho požadavcích a projektech.



Obr. 51 Úvodní obrazovka Helpdesk s popisnou statistikou

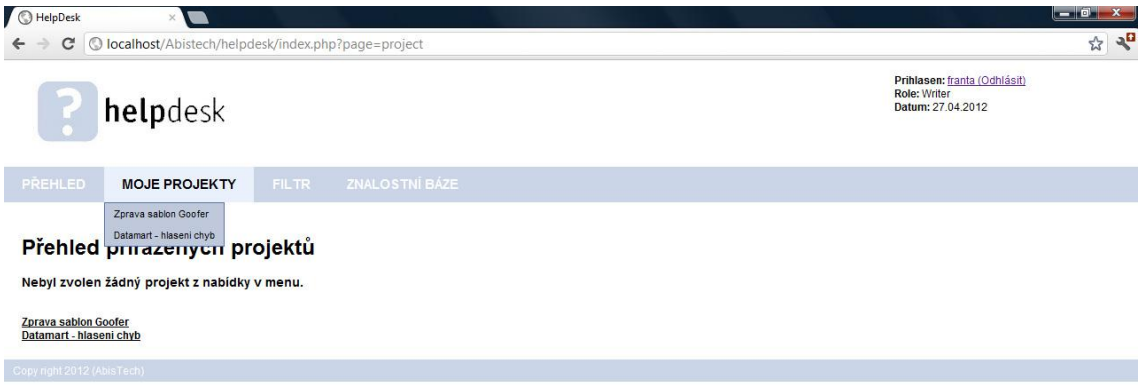
Na úvodním okně lze dále vidět hlavní navigační menu, které se zobrazuje ve všech částech aplikace Helpdesk, a za jehož pomoci se uživatel může přesouvat mezi jednotlivými

stránkami aplikace.

V pravém horním rohu je grafický výstup komponenty *eUserComp*, která pracuje s objektem *User*, vytvořeným po přihlášení. Komponenta zobrazuje údaje o uživateli, získané z objektu *User*. K dalším činnostem této komponenty je možnost odhlášení uživatele. Dále poskytuje informace o projektech, které má uživatel k dispozici a pod jakou uživatelskou rolí je přihlášen.

5.4.3 Komponenta přehledu požadavku v Helpdesku

Uživatel může z úvodní obrazovky přejít na přehled požadavků pomocí navigačního menu v horní části obrazovky potvrzením volby *Moje projekty* (obr. 52). Po najetí kurzoru myši na nabídku *Moje Projekty* se provede vysunutí javascriptového *Popup* menu, které nabízí jednotlivé projekty, na které je uživatel přiřazen. Pro zajištění správné funkčnosti aplikace i při vypnutí podpory *Javascript*, lze potvrzením nabídky *Moje Projekty* zobrazit seznam přiřazených projektů v hlavním panelu ve formě odkazů. Uživatel si zvolí projekt z *Popup* menu nebo z nabídkové stránky a bude přesunut do detailu projektu.



Obr. 52 Volba projektu prostřednictvím nabídky menu

Projekty pro uživatele jsou velmi důležité, protože jednotlivé požadavky v Helpdesku se tvoří pod projekty, proto aby uživatel mohl s Helpdeskem pracovat a musí tedy mít přiřazený alespoň jeden projekt. Po zvolení projektu bude uživateli zobrazen detail projektu s jednotlivými požadavky zadanými uživatelem (obr. 53). Na obrazovce má uživatel přehled o jednotlivých požadavcích, které pro daný projekt zadal. V komponentě seznamu požadavků je uživatel zkráceně informován o stavu požadavku, o termínu do kdy má být požadavek vyřešen, kdo jeho požadavek řeší.

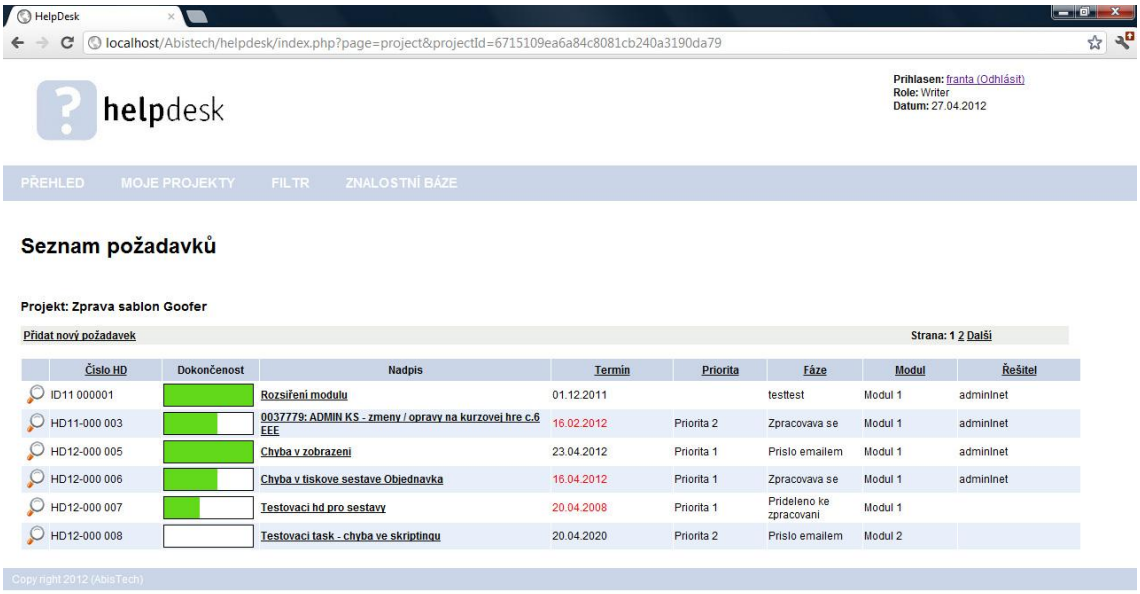
Komponenta seznamu požadavků umožňuje uživateli požadavky řadit podle jednotlivých sloupců tabulky. Řadit záznamy lze podle sloupců, které mají v záhlaví podtržením vyznačený popis. Jedná se o hypertextový odkaz a jeho potvrzení se provede řazením. Při prvním kliknutí na odkaz sloupce se provede sestupné řazení dle sloupce, při druhém kliknutí na stejný odkaz se provede vzestupné řazení záznamu. Stejné funkce nabízejí všechny označené sloupce. Další vlastností komponenty je stránkování seznamu požadavků. V aplikaci je nastaveno v konfiguračním souboru kolik může být zobrazeno maximálně záznamů na stránku. Uživateli je umožněno přecházet mezi jednotlivými stranami, buď rovnou přes odkaz čísla stránky, anebo prostřednictvím odkazů *Další* nebo *Předchozí*. Pro vlastnosti seznamu řazení a stránkování záznamů se využívá inicializace třídy *Filter* uvnitř komponenty. V komponentě se nastavením vlastnosti *Paging* třídy *Filter* docílí zobrazení správného množství záznamů, vlastností *arrSorting* třídy *Filter* se realizuje seřazení záznamů.

Pro každý požadavek v seznamu požadavků se vyskytuje volba pro zobrazení detailu požadavku. Zobrazení detailu požadavku se provede kliknutím na ikonu lupy nebo přes odkaz umístněný pod názvem požadavku.

Na obrazovce seznamu požadavků je uveden název projektu, aby uživatel věděl, kde se

nachází. Pod názvem projektu je nabídnuta uživateli možnost přidání nového požadavku přes volbu *Přidat nový požadavek*.

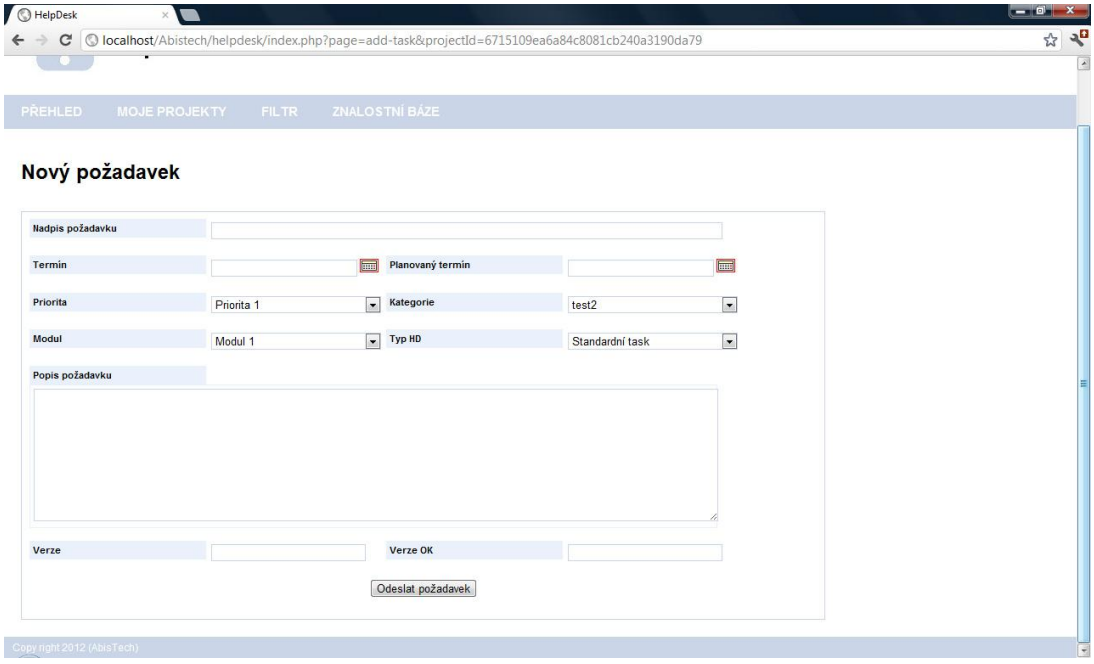
Hlavní činností komponenty je zajistit uživateli přehled o všech jeho požadavcích, členěných podle jednotlivých projektů.



Obr. 53 Přehled požadavků uživatele pro daný projekt

5.4.4 Komponenta přidání nového požadavku

Přidání nového požadavku může uživatel provést pro jednotlivé projekty při zvolení projektu. Potvrzením možnosti přidání požadavku je uživateli komponentou nabídnut formulář pro zadání požadavku přes grafické rozhraní. Komponenta zajišťuje pro prvky formuláře typu *select* inicializaci hodnot v seznamu, ze kterého uživatel vybírá hodnotu. U prvku, kde se vyplňuje datum, je uživateli nabídnuto zadání prostřednictvím javascriptové komponenty kalendáře.

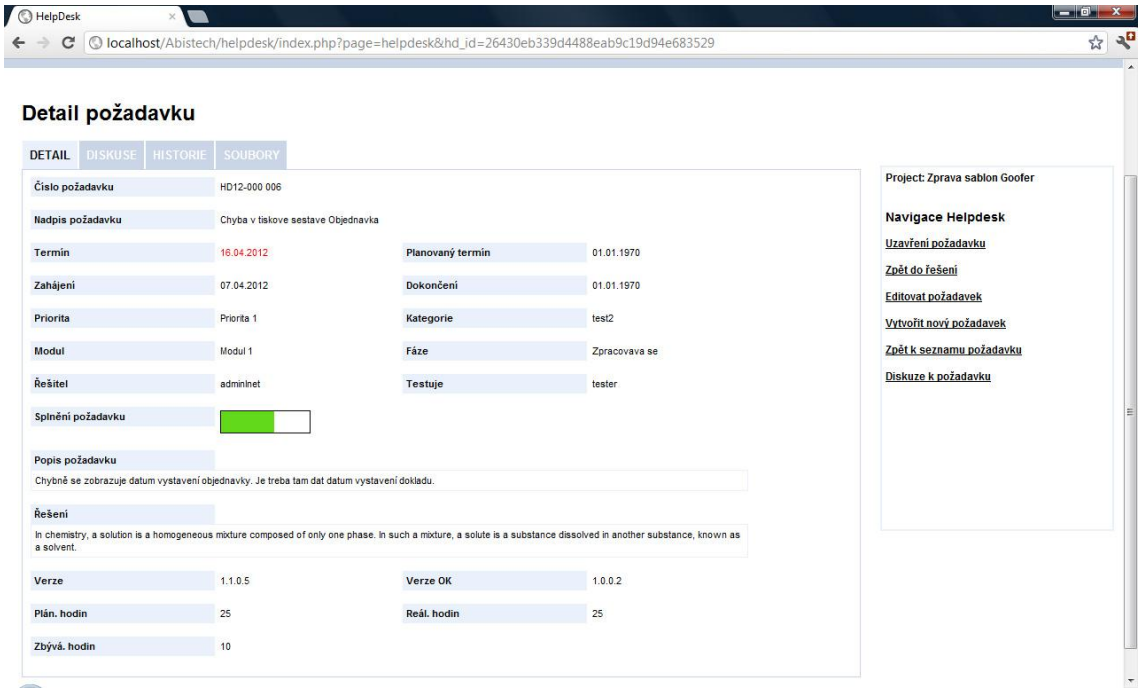


Obr. 54 Komponenta pro přidání nového požadavku

Uživatel postupně vyplní nebo zvolí hodnoty polí ve formuláři a přidání se uskuteční potvrzením tlačítka *Odeslat požadavek*. Uživatel vyplňuje např. název či datum, do kdy má být požadavek vyřešen, samotné znění požadavku, atd. Komponenta zajišťuje korektní zadání dat. Probíhají zde dvě kontroly dat odeslané formulářem. První se uskutečňuje již na straně prohlížeče prostřednictvím Javascriptu, kde se kontroluje vyplnění povinných polí. Je-li hodnota povinného pole nesprávně zadána, je uživatel upozorněn prostřednictvím *messagebox* o nesprávně vyplněném povinném poli a na upozorňované pole je nastaven *focus*. Při kontrole Javascriptem nastane odeslání dat formuláře teprve ve chvíli, když jsou všechna povinná pole správně vyplněna. Druhá kontrola probíhá prostřednictvím metody komponenty. Tato kontrola zajišťuje správnost údajů, v případě, kdy by nebyl v prohlížeči povolen Javascript. Komponenta na základě vyhodnocení správnosti údajů zadaných od uživatele provede vytvoření BO. Při vytváření BO se inicializují i data, která nejsou zadaná uživatelem, např. číslo požadavku se zachováním řady pro určitý rok, generuje se primární klíč, datum vytvoření a v neposlední řadě se hodnoty, které nejsou vyplňovány uživatelem, nastaví do výchozí hodnoty, třeba první fáze požadavku, atd. Následně je BO předán přes DAO vrstvu ke zpracování a data jsou uložena do databáze. Po úspěšném odeslání požadavku uživatelem je mu zobrazen druhý stav komponenty, který mu nabízí možnost nového založení požadavku v projektu, nebo může přejít na seznam požadavků k danému projektu. Hlavní náplní komponenty je vytváření požadavků uživatelem v Helpdesku.

5.4.5 Komponenta detail požadavku – úvodní přehled

Komponenta detail úvodní přehled (obr. 55) zobrazuje veškeré údaje, které jsou pro daný požadavek k dispozici přes grafické rozhraní. V této komponentě jsou uvedeny informace zadané uživatelem, především se však zde zobrazují informace od řešitele určitého požadavku. Uživatel se přes úvodní obrazovku detailu požadavku dozví, je-li jeho požadavek už přiřazen řešiteli, popř. v jaké fázi řešení se požadavek nachází a na kolik procent je již zpracován. V detailu požadavku se nachází i text o řešení, uvedený řešitelem (pokud je požadavek vyřešen). Pomocí této komponenty má uživatel úplný přehled o tom, jak postupuje řešení jeho požadavku.



Obr. 55 Detail požadavku úvodní přehled

Součástí komponenty jsou dvě subkomponenty, které zajišťují dílčí funkce a zobrazení.

Pro účel grafického znázornění procentuálního zpracování požadavku byla vytvořena komponenta *Progressbar*, která se v entitách HTML nevyskytuje. Komponenta byla realizována pomocí HTML, CSS a Javascriptu. Vstupním parametrem, který reprezentuje procenta, se v komponentě vyhodnotí data pro vykreslení hodnoty v této komponentě. Grafickou podobu subkomponenty lze vidět na obr. 55.

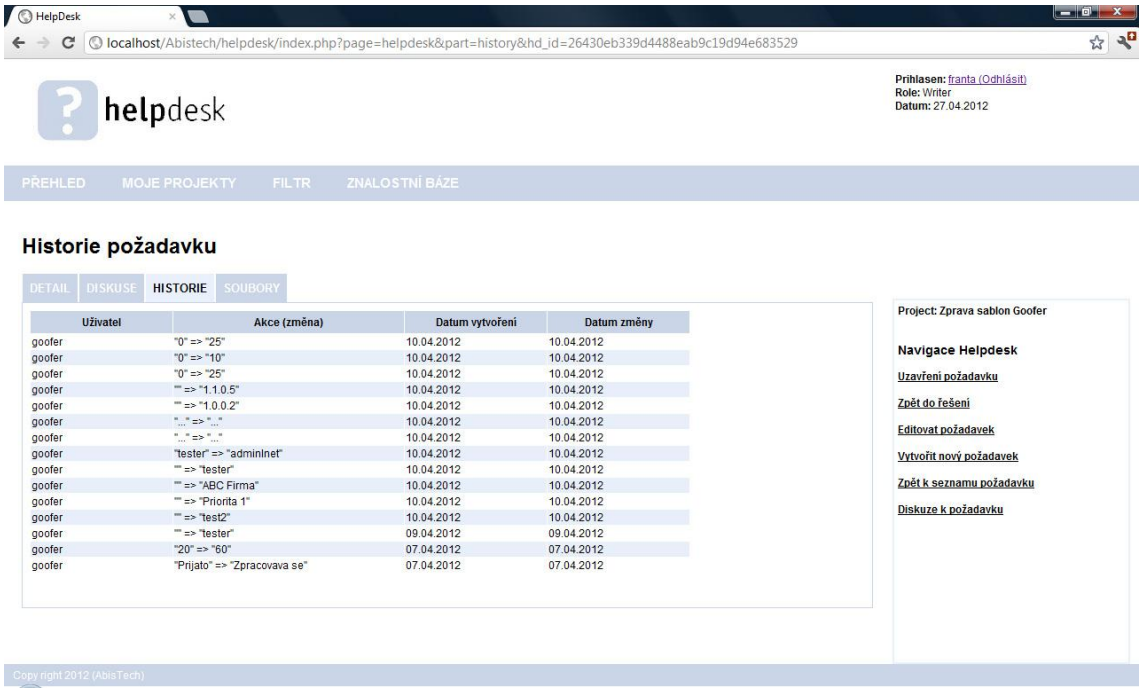
Další subkomponentou je vypsání a označení termínu řešení. Komponenta vyhodnocuje termín, do kdy má být požadavek vyřešen. Na základě zadaného termínu od uživatele se vyhodnotí, zda je požadavek po termínu či ne. Není-li požadavek vyřešen a je po termínu, hodnota data signalizuje červenou barvou, že požadavek nebyl splněný včas. Uživatel má pak optickou kontrolu a nemusí pracně zjišťovat, jestli má nějaké požadavky po termínu vyřešení.

Uživateli jsou v detailu požadavku umožněny operace s požadavkem. Zmíněné operace se vyskytují na pravé straně obrazovky detailu v pomocném navigačním menu. V pomocném navigačním menu jsou umožněny uživateli následující operace:

- *Uzavření požadavku* – uživatel může požadavek uzavřít nebo přerušit zpracování požadavku. Uzavření se provede, je-li požadavek vyřešen, ukončit lze v průběhu fází požadavku
- *Zpět do řešení* – je-li požadavek již vyřešen a uživateli řešení nevyhovuje, může vrátit požadavek zpět do řešení
- *Editovat požadavek* – uživateli je umožněno měnit hodnoty v již zadaném požadavku. Ovšem editaci záznamů může být provedena pouze nad hodnotami, které byly zadány uživatelem
- *Vytvořit nový požadavek* – v navigačním menu je rovněž možnost vytvořit nový požadavek do projektu, kde se nachází požadavek, kterého si uživatel prohlíží detail
- *Zpět na seznam požadavků* – potvrzením této volby se uživatel přesune zpět na seznam požadavků v rámci projektu

5.4.6 Komponenta detail – historie událostí

V detailu požadavku se nachází členění v podobě záložek, které lze vidět nad ohraničením oblasti detailu požadavku. Pro přechod mezi jednotlivými záložkami provede uživatel kliknutí na libovolnou záložku, která není aktuálně zvolena.



Obr. 56 Detail požadavku část historie událostí

Zvolí-li uživatel záložku *historie*, nastane zobrazení dat, která se týkají zvolené záložky. V záložce *historie* jsou uvedeny veškeré informace, které se v průběhu života požadavku měnily. Uživatel zde má přehled, v jakých časových úsecích se měnily fáze požadavku. Má přehled, kolik uživatelů se podílelo na řešení požadavku, když si požadavek předávali. Uživatel v této záložce vidí veškeré události, které se nad požadavkem odehrály. Tato funkcionalita je zajištěna na úrovni databáze pomocí *triggerů*, které provedou zápis do historizující tabulky pro tabulku *helpdesk*.

5.4.7 Komponenta detail požadavku – část přiložené soubory

Na komponentu pro přiložené soubory se uživatel přesune přes záložku *soubory*. Po přepnutí záložky se zobrazí grafické rozhraní komponenty (obr. 57), pro přidávání souboru ke zvolenému požadavku.

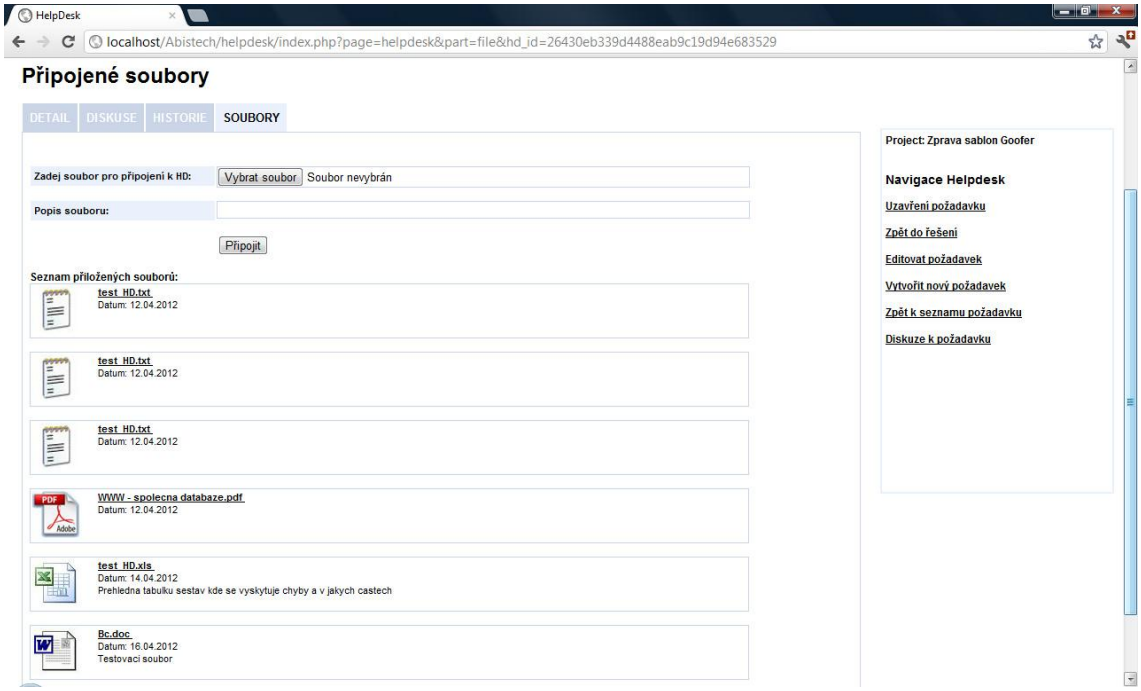
Komponenta pro připojení souboru zobrazí formulář pro možnost připojení souboru (objektu). Uživatel přes tlačítko *Vybrat soubor* zadá přes průzkumníka cestu k souboru umístněného na jeho lokálním disku. Uživatel může k přikládanému souboru doplnit krátký popis, např. o obsahu souboru, k čemu je určen, atd. Připojení souboru k požadavku provede uživatel přes tlačítko *Připojit*.

Komponenta při odeslání souboru provádí dvě kontroly souboru:

1. Kontrola velikosti souboru – v konfiguračním souboru je uvedena maximální velikost souboru v MB, které lze připojit k požadavku. V případě překročení tohoto limitu je uživatel informován o překročení limitu pro velikost přiložených souborů, a uložení souboru nenastane.
2. Kontrola na typ souboru – aplikace podporuje pouze připojení některých typů souborů. Když přiložený soubor neodpovídá podporovanému formátu, je uživatel upozorněn na to, že přípona přiloženého souboru není podporována.

Komponenta podporuje následující přípony souboru:

- *txt, csv, xm,, html*
- *doc, docx, xls, xlsx, pps*
- *pdf*



Obr. 57 Detail požadavku část pro připojení souboru

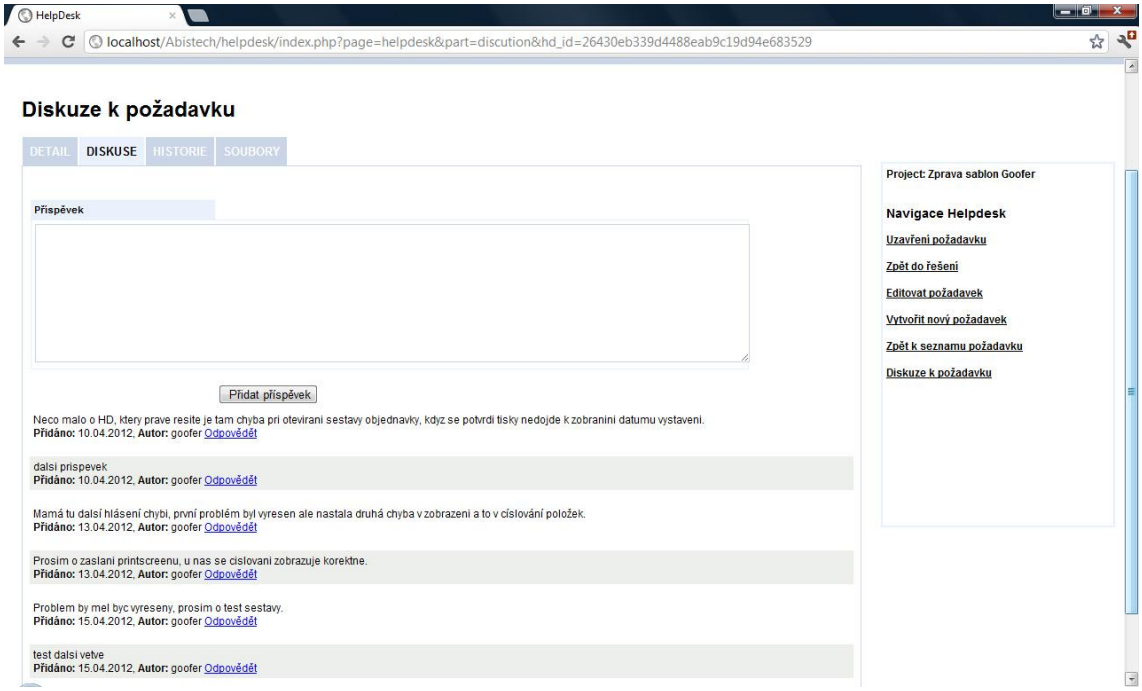
Po úspěšném průchodu přes zmíněné kontroly nastane samotné uložení souboru do adresáře umístěného na FTP webové aplikace. Cesta k adresáři je uvedena v konfiguračním souboru. Není-li adresář na FTP vytvořen, je komponentou vytvořen a nastaven na práva úplného řízení. Komponenta po přenesení souboru provede zápis přes vrstvu DAO do tabulky *large_object*, aby mohl být soubor přenesen přes synchronizační nástroj do desktopové aplikace.

Po úspěšném přiložení souborů, se následně vytvoří seznam jednotlivých souborů, které byly k požadavku připojeny. V seznamu je zobrazena ikona, která reprezentuje příponu souboru. Dále se zde nachází název souboru, včetně jeho přípony. Další informace jsou popis k souboru zadaný od uživatele a datum, kdy byl soubor připojen. Seznam přiložených souborů je vyobrazen na obr. 57. Uživateli je v seznamu umožněno jednotlivé soubory otevřít a to přes ikonu typu soubor, a nebo přes název souboru.

Hlavní náplní komponenty je zajistit uživateli možnost připojit soubory k požadavku a jejich následnou evidenci a zobrazení v požadavku.

5.4.8 Komponenta detail požadavku – část diskuse

Uživatel se do části diskuse přesune přes záložku *diskuze* (obr. 58). V této části byla vystavěna možnost komunikace mezi zadavatelem a řešitelem prostřednictvím diskusního fóra. Diskusní forum k požadavku bylo realizováno za účelem urychlit komunikaci mezi zadavatelem a řešitelem, aby bylo docíleno co nejrychlejšího a nejefektivnějšího vyřešení problému.



Obr. 58 Detail požadavku část diskuse

Komponenta se skládá ze dvou částí, a to z části k přidávání příspěvku, a části pro zobrazení příspěvků. V části pro přidání příspěvku je uživateli zobrazen formulář s polem, kde může zadat svůj příspěvek (otázku, konstatování, atd.) a odeslání příspěvku učiní tlačítkem *Přidat příspěvek*. Nový příspěvek se zobrazí v části pro přehled diskuse pod formulářem.

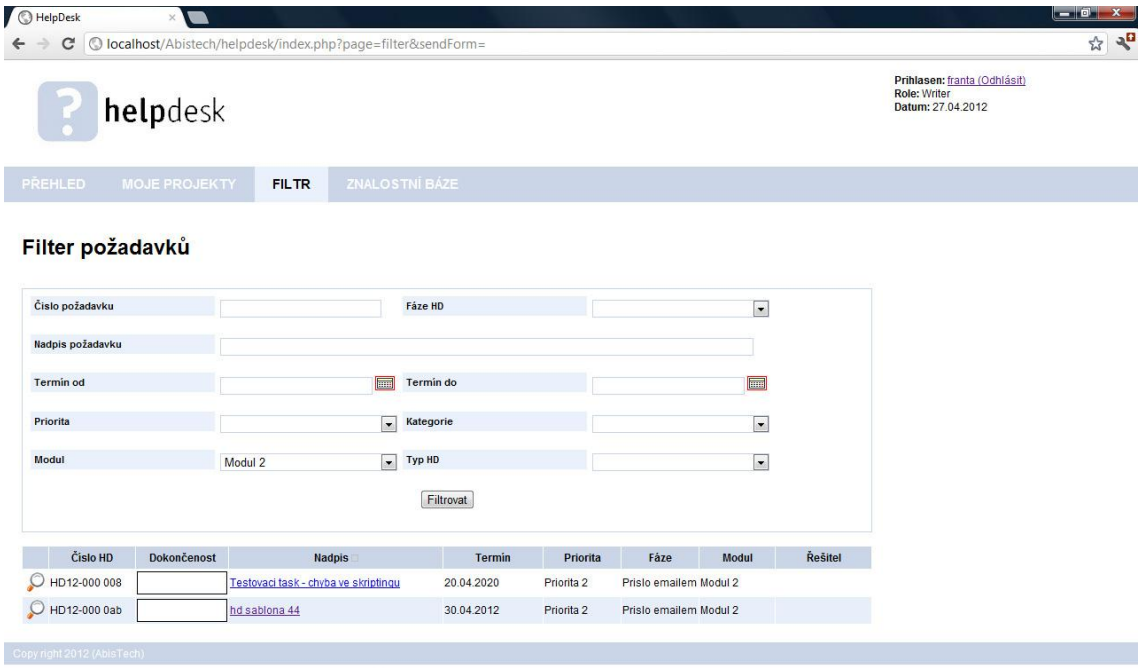
Diskuse k požadavku má stromovitou strukturu, tzn. každý příspěvek může mít na sebe navázány další příspěvky a ty zase mohou obsahovat další reakce na příspěvek. V databázi je tohoto efektu docíleno vazbou tabulky *discussion*, sama na sebe, pomocí pole *discussion_master_id*. V kódu metody se využívá rekurzivního volání metody až k listům (koncovým bodům) stromové struktury. Pomocí rekurze se docílilo toho efektu, kdy se související příspěvky zobrazují následně po sobě. V diskusi se může vyskytovat více

samostatných stromových struktur (vláken). Nové vlákno se založí právě přidáním nového příspěvku přes vyplnění příspěvku a stisknutí tlačítka *Přidat příspěvek*. Další úrovně stromu se tvoří pomocí odkazu *Odpovědět*. Při potvrzení tohoto odkazu bude uživatel odkázán na formulář odpovědi. Přidaná reakce bude zařazena pod příspěvek, kterého se reakce týkala. Pomocí odpovědí se provede zanořování do úrovní stromu a příspěvky se udržují na stejné úrovni spolu. Při zadání prvního příspěvku k požadavku, komponenta vykoná založení diskusního fóra k požadavku, ke kterému se vztahují jednotlivé příspěvky.

Komponenta *diskuse* slouží uživateli ke komunikaci s řešitelem, k objasnění problému a následně umožňuje komunikaci archivovat. Už nemusí probíhat komunikace skrze e-mail a vše se řeší v rámci Hepldesku a daného požadavku (nemusí se již uchovávat komunikace přes e-mail).

5.4.9 Komponenta filtr požadavků

Uživatel se přesune na komponentu *Filtr* (obr. 59) přes hlavní navigační menu položkou *FILTR*. Uživateli je zde umožněno provést vyhledávání mezi všemi jeho vytvořenými požadavky. Komponenta je rozdělena na dvě části. První část je tvořena formulářem, kde uživatel zadává nebo volí filtrační omezení, podle kterých chce vyhledat požadavky vyhovující zadaným kritériím.



Obr. 59 Komponenta filtračního nástroje

Potvrzením tlačítka *Filtrovat* jsou jednotlivá kritéria odeslána ke zpracování. Na základě odeslaných filtračních omezení komponenta provede inicializaci vlastností instance třídy *Filter*, která je vstupním parametrem metod ve vrstvě DAO. U komponenty filtrace záznamů se pro přidávání položek do objektu *Filter* využívají nejen položky s porovnávacím operátorem rovnosti, ale i operátory nerovností (menší, větší) pro určení intervalu, operátor *like* pro hledání v řetězci a další. Na základě objektu *Filter* vrátí vrstva DAO list BO odpovídající zadaným omezením od uživatele. Vráceny výsledek požadavků, vyhovující omezením komponenta, se zobrazí pod filtračním formulářem. Výsledek je uživateli vypsán přehlednou tabulkou, kde jsou uvedeny vybrané informace o požadavku. Uživateli je ve výsledku vyhledávání umožněno přejít na detail jednotlivých požadavků. Když není na základě zvolených výběrových kriterií nalezen žádný výsledek, uživatel je informován o nenalezení záznamu podle odeslaných kritérií.

Komponenta má uživateli usnadnit vyhledávání požadavků ve všech jeho přiřazených

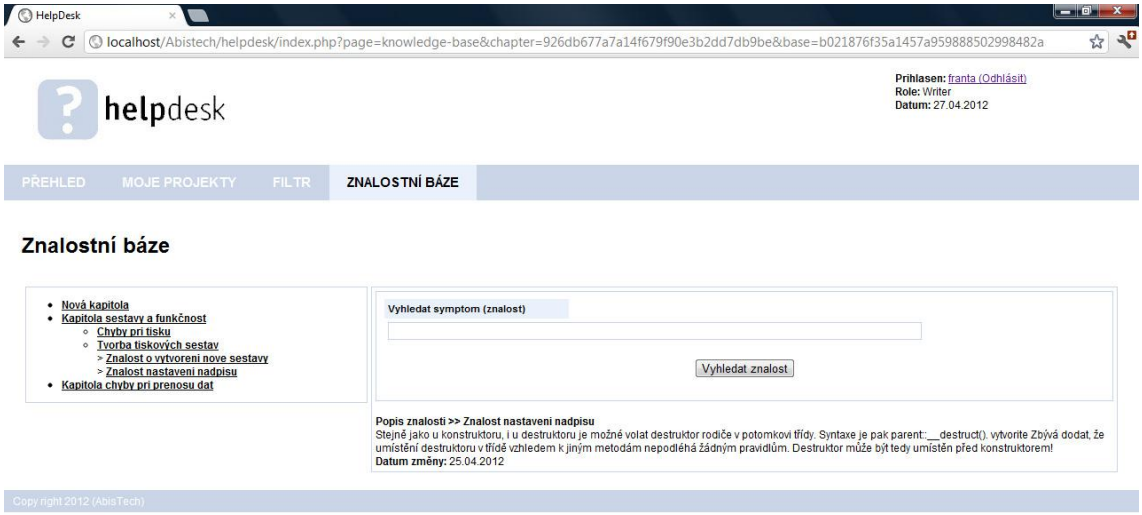
projektech. Nabízí mu možnost si zvolit např. pouze požadavky v určitém časovém intervalu, v určité fázi, atd. Komponenta usnadňuje uživateli orientaci mezi jeho požadavky, aby si nemusel přesně pamatovat pod jaký projekt daný požadavek zadal.

Výhodou vícevrstvé aplikace je snadná rozšiřitelnost formuláře pro vyhledávání, pouze se rozšíří grafický vzhled a rozšířené položky se přidají do objektu *Filter*.

5.4.10 Komponenta znalostní báze

Znalostní báze obsahuje informace, pomocí kterých by mohl uživatel vyřešit svůj problém bez zadání požadavku do Helpdesku. Ve znalostní bázi uživatel může vyhledat, zda jeho problém nebyl již řešen a vyřešen. Mohou zde být uloženy i postupy, nastavení a jednotlivé symptomy (problém v požadavku) a jeho řešení. Znalostní báze je tvořena na straně uživatele v *Goofer*. Uživatel desktopové aplikace rozčlení jednotlivé znalosti. Může využít i již vytvořených požadavků přes webové rozhraní a přidat je do báze. V *Goofer* se vytváří kapitoly a podkapitoly, které člení jednotlivé znalosti pro uživatele. Webové rozhraní má zajistit prezentaci znalostí a možnost ve znalostech snadno vyhledávat, a nebo jimi procházet.

Uživatel komponentu znalostní báze (obr. 60) nalezne v hlavním navigačním menu pod volbou *ZNALOSTNÍ BÁZE*. Když se uživatel přepne do komponenty znalostní báze, zobrazí se grafické rozvržení komponenty.



Obr. 60 Komponenta znalostní báze

Komponenta je rozdělena do dvou sloupcových částí. V levé části obrazovky je umístěn rejstřík znalostí. Znalosti jsou rozčleněny do kapitol a podkapitol. Rejstřík reprezentuje stromovou strukturu, do níž se uživatel postupně zanořuje až ke konkrétní znalosti. Uživateli se postupně otevírají jednotlivé podkapitoly a znalosti, podle toho jakou zvolí kapitolu, o úroveň výše. Pod jednotlivými kapitolami mohou být přiřazeny znalosti. Když si uživatel přes rejstřík dohledá znalost, která ho zajímá a rozklikne ji, nastane zobrazení v pravé části obrazovky. Pro jednotlivé znalosti se zobrazí název, pod kterým je znalost vedena v rejstříku a samotný text znalosti. U každé je uvedeno i datum poslední změny. Rejstřík je jeden ze způsobu pohybů uživatele po jednotlivých znalostech v bázi.

Druhou možností vyhledávání ve znalostní bázi je přes formulář pro vyhledávání v bázi, přes klíčová slova. Uživateli je umožněno vyplnit do formuláře symptom nebo slovo, které chce aby znalosti obsahovaly. Po odeslání formuláře nastane fulltextové vyhledávání v názvech a textu znalostí. Po skončení vyhledávání jsou vráceny pouze znalosti, které uživatelem zadaná klíčová slova obsahují. Vyhledané znalosti jsou vypsány v pravé části obrazovky pod vyhledávacím formulářem. Uživatel si může jednotlivé znalosti prohlédnout a zjistí, jestli vyhovují jeho problému nebo zda nezkusí zadat do vyhledání problém pomocí jiných klíčových

slov. Není-li klíčové slovo obsaženo ve znalosti, je vrácen prázdný výsledek a uživatel je informován o nenalezení znalosti podle zadaných klíčových slov. Komponenta je především určena pro ty uživatele, kteří by byli schopni vyřešit svůj problém na základě znalostí získaných z báze.

5.5 Grafické rozhraní - GUI

Vrstva GUI (*Graphical User Interface*), dále jen rozhraní, tvoří samotný vzhled aplikace. Rozhraní rohoduje jak bude stránka vypadat, určuje vzhledy formulářů a jednotlivých komponent v Helpdesku. Vytvořením vrstvy má být docíleno oddělení vzhledu aplikace od výkonových kódů jednotlivých komponent. Webové aplikace lze psát i způsobem, kdy HTML kód je vypisován přímo ve výkonovém kódu v PHP. Vazba mezi výkonovým kódem a HTML vzhledem je pevně spojena, v tomto případě je velmi obtížné měnit grafický vzhled aplikace. Obtížnost změny grafického vzhledu je z důvodu existence HTML kódu v jednotlivých kódech PHP, kdy se určitá část stránky vykresluje dynamicky v PHP. Musí se zasahovat přímo do výkonové logiky a to není žádoucí. Proto byla navržena další vrstva grafického rozhraní, která obsahuje pouze značky jazyka HTML a odkazy na proměnné z výkonové logiky.

Pro řešení výše uvedeného problému byl pro Helpdesk zvolen nástroj *Smarty*, který tento problém vyřešil. *Smarty* je šablonovací systém určený pro PHP, který umožňuje oddělit aplikační logiku (kód PHP) od zobrazovací logiky (HTML). Šablonový systém *Smarty* byl zvolen, proto, že se jedná o komplexní systém nabízející velké množství funkčností a existuje k němu podrobná dokumentace. Dalším kritériem výběru bylo, že patří k volně šiřitelným systémům, u kterých lze jednotlivé funkce přebírat a rozšiřovat. Pomocí šablon *Smarty* bylo realizováno grafické rozhraní Helpdesku.

Ve vrstvě GUI jsou realizovány všechny prvky, které vidí uživatel na obrazovce. Jsou zde všechny formuláře, zobrazení komponent atd. Oddělením grafického vzhledu od aplikační logiky bylo docíleno kýženého výsledku snadné změny grafického zobrazení. Snadná a rychlá změna grafického zobrazení byl jeden z požadavků na Helpdesk.

5.5.1 Tvorba šablon grafického rozhraní

Smarty pracuje se soubory šablon s příponou *tpl*, kde je psán HTML kód, který má na starosti formátování grafického rozhraní. Při tvorbě grafického rozhraní bylo třeba vytvořit pro každou komponentu soubor typu šablona, do kterých byl napsán HTML kód, který utváří jejich grafický vzhled a rozmístění na stránce. Některé komponenty mají i více šablonových souborů, protože mohou měnit v některých stavech vzhled (např. komponenta pro přidání nového požadavku má stav, kdy zobrazuje formulář pro zadání požadavku, po odeslání formuláře nabývá druhého stavu uspěšného přidání požadavku a zobrazuje nadefinované možnosti). Na obr. 61 je ukázka realizace šablony pro komponentu *eUse*r. Komponenta má na starosti zobrazení informací o přihlášeném uživateli a nachází se v pravém horním rohu stránek Helpdesku.

```
<div id="header-user">
   <strong>Prihlasen:</strong> <a href="index.php?logout=logout">{$userName}
  (Odhlásit)</a>
   <br /><strong>Role:</strong> {$userRole}
   <br /><strong>Datum:</strong> {$actulDate}
</div>
```

Obr. 61 Ukazka šablony komponenty eUser informace o uživateli

Hodnoty proměnných z jednotlivých komponent se v šabloně zobrazí pomocí odkazů na proměnnou, které jsou na obr. 61 vloženy mezi složenými závorkami (např. {$userName}). Při kompilaci šablony bude na místo odkazu vložena hodnota proměnné předané šabloně přes komponentu. Jednotlivé komponenty zajišťují volání své šablony a předání proměnných do

systému *Smarty*, který s nimi dále pracuje. Ve výkonové logice komponenty je vyhodnoceno, jakou šablonu má zobrazit podle stavu, ve kterém se komponenta nachází. Při prvním volání šablony komponentou systém *Smarty* šablonu zkompiluje, kompilovaná verze je výjadřena *php* kódem, který vykoná zobrazení grafického vzhledu. Při dalším volání šablony komponentou již ke kompilaci nedochází. Šablonový systém nabízí velké množství funkcí a řidících struktur. Při tvorbě šablon Helpdesku bylo využito cyklů a řidících struktur, které šablonový systém nabízí. Cyklů v šablonách se používá především pro proměnné typu pole předané komponentou do šablony. Typickým příkladem využití cyklu je zobrazení v šabloně, třeba u komponenty zobrazení požadavků v jednotlivých projektech. Zde se cyklu využívá pro zobrazení řádků tabulky, kde každý řádek je jeden prvek pole s BO. Ukázkou využití struktury cyklu v šabloně je obr. 62.

```
{section name=dis loop=$discussionArr}
<div class="{cycle values="dis-row-l,dis-row-s"}">

{$discussionArr[dis]->discussionText->getValue()}<br />
<strong>Přidáno:</strong> {$discussionArr[dis]->createDate->getValueDate()},
<strong>Autor:</strong>
{$discussionArr[dis]->userName->getValue()}
<a href="{$discussionArr[dis]->url}">Odpovědět</a>
</div><br />
{/section}
```

Obr. 62 Ukázka využití cyklu v šabloně u komponenty pro zobrazení diskuse

5.5.2 Grafický vzhled aplikace Helpdesk

Pro grafický vzhled byl vytvořen dvousloupcový layout, který se přizpůsobuje celé šíři obrazovky. Navržený layout se přizpůsobuje velikosti okna prohlížeče. Layout je vytvořen pomocí HTML a *css* stylů. V layout jsou odkazy na *css* styly, které zajištují rozmístění jednotlivých elementů v layoutu. Přes kaskádové styly se řídí i používaný styl písma, jaké pozadí element bude mít, jestli bude použito ohraničení a jaké šířky a barvy. Kaskádové styly dotváří grafický vzhled stránek. Navržený layout byl rozložen do tří šablon. Šablony layoutu jsou *header*, *body* a *footer*. Tyto šablony zajišťují rozmístění stránky Helpdesku. Vzhled komponent je vkládán do těchto tří šablon. Šablonový systém *Smarty* umožňuje vkládání šablon do šablony. Spojením všechy šablon a prvků se docílí výsledného vzhledu stránky Helpdesku, tato část se řídí v souboru *index.php*, který se načte při zadání adresy Helpdesku.

Pro změnu grafického vzhledu aplikace stačí pouze úpravy jednotlivých souborů šablon a změna hodnoty v souboru kaskádových stylů. Tyto změny může udělat i kodér HTML stránek bez zásahu programátora.

Výsledné vzhledy šablon layoutu a jednotlivých komponent byly představeny v podkapitolách aplikační vrstvy.

6 ZÁVĚR

Práce se zabývala vytvořením webové nástavby Helpdesk nad modulem desktopové aplikace, která zajišťuje část pro webového uživatele (zadavatele požadavků), zatímco část pro řešitele zajišťuje desktopová aplikace, která není součástí diplomové práce. Účelem práce bylo zhotovení nástroje, který umožní komunikaci desktopové aplikace s uživateli, pomocí internetového rozhraní.

V první části je krátké seznámení s vývojovými a softwarovými prostředky, které byly použity při realizaci diplomového projektu. U některých nástrojů byla zmíněna zkráceně jejich historie. Hlavním záměrem této části bylo stručné představení prostředků k vývoji praktické části a objastnění jejich významu.

Praktická část se skládá ze tří částí, které bylo nezbytné vytvořit pro funkčnost celého projektu.

Při návrhu a analýze projektu bylo zjištěno, že bude třeba vyvinout velké množství funkcí a tříd, které se z velké části opakují a liší se pouze v jistých částech, závislých na určitých vlastnostech. Pro tyto účely byl v první praktické části zhotoven nástroj pro generování kódu. Generátor kódu zajistil flexibilitu a rychlost při vývoji velkého množství funkcí a tříd, které by musely být mnohokrát přepisovány. Nezbytnou součástí generování kódu bylo i vytvoření šablon, na základě kterých bylo generování kódu realizováno, a které dávají podobu jednotlivým třídám a funkcím. Nástroj se stal nedílnou součástí vývoje a při dalších rozšířeních projektu bude zcela jistě využíván. Generátor kódu se využívá v obou následujících částech.

V další části projektu bylo třeba zajistit přenos dat mezi desktopovou aplikací a webovou aplikací Helpdesk a naopak. Pro tyto účely bylo nezbytné vytvořit nástroj, který umožní přenos dat. Pro synchronizaci dat bylo využito webových služeb v *php* a využití třídy NuSOAP pro webové služby. Pro synchronizaci byly vytvořeny funkce, které jsou v rámci webových služeb volány a provádí datové změnové operace. Přes rozhraní webových služeb již probíhá komunikace mezi desktopovou aplikací a webovou aplikací.

Poslední část práce je věnována návrhu a vývoji webové nástavby Helpdesk. Zde je zmíněna architektura, která byla pro projekt navržena a jsou zde popsány jednotlivé vrstvy modelu aplikace Helpdesk. Při práci se využívá vícevrstvý model aplikace, který má zajistit snadný vývoj a rozvoj webové aplikace do budoucna. Při této části byla navrhnuta databázová struktura pro modul Helpdesk v desktopové aplikaci *Goofer*. Na základě navržené struktury byl realizován modul Helpdesk v systému *Goofer* firmou AbisTech s.r.o. Jednotlivé podkapitoly jsou věnovány funkčnosti a činnosti jednotlivých vrstev, které tvoří webovou nástavbu Helpdesk. Jsou popsány v pořadí, ve kterém byly vyvinuty. Tato chronologie je dána jako při stavbě budovy, kde se začíná od základů a postupuje se až po střechu. Výsledkem poslední kapitoly je funkční webová aplikace Helpdesk (uživatelská část pro zadavatele), která s využitím synchronizačního nástroje komunikuje s desktopovou aplikací.

Výsledná webová aplikace Helpdesk bude teprve vstupovat do fáze uživatelský testů, po skončení fáze testů se předpokládá její nasazení pro interní využití v průběhu měsíce srpna roku 2012.

SEZNAM POUŽITÉ LITERATURY

[1] ADAPTIC. *Znalosti HTTP* [online]. 2010, [cit. 2012-04-30]. Dostupné z: <http://www.adaptic.cz/znalosti/slovnicek/http>.

[2] FS.VSB. *Architektura databází* [online]. 2010, [cit. 2012-05-01]. Dostupné z: <http://books.fs.vsb.cz/MSSQLServer/MSSQL_soubory/SQL_index3.htm>.

[3] SCHAUER, Petr. *Apache* [online]. 2006, [cit. 2012-05-01]. Dostupné z: <http://www.petr.isibrno.cz/professional/pocuvahy.php>.

[4] BROŽEK, Erik. *Server Apache* [online]. 1998, [cit. 2012-05-01]. Dostupné z: <http://www.zive.sk/server-apache-co-je-vlastne-zac-a-co-dokaze/sc-3-a-153277/default.aspx>.

[5] RYCHNOVSKÝ, Lukáš. *Apache* [online]. 2003, [cit. 2012-05-01]. Dostupné z: <http://www.fi.muni.cz/~kas/p090/referaty/2003-jaro/skupina10/apache.html>.

[6] ZAJÍC, Petr. *Serial PHP* [online]. 2003, [cit. 2012-05-01]. Dostupné z: <http://www.linuxsoft.cz/php/Serial-PHP.pdf>.

[7] ZAJÍC, Petr. Historie jazyka *PHP* [online]. 2004, [cit. 2012-05-01]. Dostupné z: <http://www.linuxsoft.cz/article.php?id_article=171>.

[8] ADAPTIC. *Znalosti XML* [online]. 2010, [cit. 2012-04-30]. Dostupné z: <http://www.adaptic.cz/znalosti/slovnicek/xml/>.

[9] LÁTAL, P. Elektronická podpora výuky předmětu AI. Zlín, 2009. 60 s., 2 s. příloh. Bakalářská práce na fakultě Aplikované informatiky Univerzitě Tomáše Bati ve Zlíně na Ústavu aplikované informatiky. Vedoucí diplomové práce Ing. Roman Šenkeřik, Ph.D..

[10] STOHWASSER, Petr. *Co je css* [online]. 2010, [cit. 2012-05-01]. Dostupné z: <http://www.pestujemeweb.cz/obsah/css/co-je-css.php>.

[11] MARKO, Rišo. *JavaScript 1 uvod* [online]. 2004, [cit. 2012-05-01]. Dostupné z: <http://www.linuxsoft.cz/article.php?id_article=237>.

[12] JANOVSKÝ, Dušan. *Javascrip uvod* [online]. 2010, [cit. 2012-05-01]. Dostupné z: <http://www.jakpsatweb.cz/javascript/javascript-uvod.html>.

[13] KIMPL, L. Prostorové nadstavby nekomerčních databází – vstup a správá geoobjektů. Olomouc, 2010. 66 s., 3 s. příloh. Bakalářská práce na fakultě Přírodovědecké Univerzity Palackého v Olomouci na Katedře geoinformatiky. Vedoucí diplomové práce RNDr. Vilém Pechanec, Ph.D.

[14] I15. *Postresql* [online]. 2011, [cit. 2012-05-02]. Dostupné z: <http://www.i15.cz/postgre-sql/>.

[15] STĚHULE, Pavel. *PL/pgSQL efektivně* [online]. 2011, [cit. 2012-05-02]. Dostupné z: <http://www.postgres.cz/index.php/PL/pgSQL_efektivn%C4%9B>.

[16] KOSEK, J. Inteligentí podpora navigace na WWW s využitím XML. Praha, 2002. 68 s., 4 s. příloh. Diplomová práce na fakultě Informatiky a statistiky Vysoké školy ekonomické v Praze na Katedře informačního a znalostního inženýrství. Vedoucí diplomové práce Ing. Vojtěch Svátek, Dr.

[17] HUGH, E. Williams; Lane, David. Programujeme webové aplikace pomocí PHP a MySQL. Praha : Computer Press, 2002. 530 s. ISBN 80-7226-760-4.

[18] GROFF, R. James; Weinberg, N. Paul. SQL – kompletní průvodce. Brno : Computer Press, 2005. 923 s. ISBN 80-251-0369-2.

[19] LAVIN, Peter. PHP objektově orientované koncepty, techniky a kód. Praha : Grada, 2009. 201 s. ISBN 978-80-2137-8.

[20] GUTMANS, A., Bakken, S. S., Rethans, D.: Mistrovství v PHP 5, Computer press, Brno, 2005. 645 s. ISBN 978-80-251-1519-0

[21] DARIE, C. a kol.: AJAX a PHP - tvoříme interaktivní webové aplikace profesionálně, Zonerpress, 2006. 320 s. ISBN 80-86815-47-1

[22] URMAN, S., Hadman, R., McLaughlin, M.: ORACLE, Programování- v PL/SQL. CPress, Brno, 2007. 720 s. ISBN 978-80-251-1870-2

[23] SCHLOSSNAGLE, G. Pokročilé programování v PHP5. Zoner press, Brno, 2004. 640 s. ISBN 80-86815-14-5
[24] Oficiální stránky PHP [on-line]. Elektronická adresa http://www.php.net
[25] Oficiální stránky MySQL [on-line]. Elektronická adresa http://dev.mysql.com
[26] Oficiální stránky Apache [on-line]. Elektronická adresa http://apr.apache.org
[27] Oficiální stránky Smarty [on-line]. Elektronická adresa http://www.smarty.net/
[28] Oficiální stránky Postgresql [on-line]. Elektronická adresa http://www.postgresql.org/