TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

Informační systém pro kabelovou knihu TU

IS for TU Cablebook

Diplomová práce

Autor: **Bc. Tomáš Fejfar**

Vedoucí práce: Ing. Petr Kretschmer

V Liberci 18. 5. 2012

Originál zadání práce

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **souhlasím** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce.

Datum:

Podpis:

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu Ing. Petru Kretschmerovi za pomoc a podporu během vypracovávání mé práce. Také bych rád poděkoval své rodině a přátelům, že mě po celou dobu studia podporovali, a své přítelkyni za trpělivost. Dále pak Martinu Hujerovi za spolupráci při nastavování integračního serveru a uživatelském testování.

Abstrakt

Práce se zaměřuje na vytvoření webové aplikace – informačního systému pro kabelovou knihu Technické univerzity v Liberci. Hlavním cílem práce je vytvořit náhradu za současnou podobu kabelové knihy, která přestává vyhovovat současným potřebám. Výsledná aplikace umožní spolupráci více osob a zajistí kontrolu referenční integrity dat.

Teoretická část práce popisuje principy kabelové knihy a dále se zabývá jednotlivými technologiemi používanými pro tvorbu webových aplikací. Na závěr teoreticky rozebírá bezpečnostní hrozby, které se u webové aplikace mohou vyskytnout.

Druhá kapitola se soustředí na analýzu a návrh aplikace. Nejprve analyzuje současný stav kabelové knihy a následně se zabývá návrhem aplikace a databáze. Součástí této kapitoly je také návrh uživatelského rozhraní. Poslední podkapitola se zabývá bezpečností a způsoby, jakými lze předcházet vzniku bezpečnostních zranitelností.

Poslední kapitola popisuje praktické zpracování aplikace. Rozebírá jednotlivé části aplikace a jejich konkrétní implementaci. Také interpretuje výsledky uživatelského testování aplikace a uvádí chyby, které testování odhalilo. Kapitolu uzavírá popis způsobů automatizace některých činností a popis technik použitých pro optimalizaci rychlosti aplikace.

Klíčová slova:

webová aplikace, kabelová kniha, informační systém, PHP, Zend Framework

Abstract

This thesis focuses on the process of creating web application – information system for the cable book of Technical University of Liberec. The main goal of the thesis is to create a replacement for the current cable book implementation that starts to lag behind the current requirements. The application will allow collaboration of more users and it will also ensure data referential integrity.

The theoretical part of this thesis deals with the principles of cable book. Furthermore, it also presents different technologies used for web application development and describes security threats that can occur in a web application.

The second part focuses on analysis and design of the application. First, it analyses the current state of the cable book. Next, it deals with design of the application, database and user interface. Last, it describes security measurements that were used to prevent security breaches.

The last chapter concentrates on the practical implementation of the application design. It analyses each part of the application and its implementation. It presents the results of user testing and problems that were detected. The chapter ends with description of different automation and speed optimization strategies used.

Keywords:

web application, cable book, information system, PHP, Zend Framework

Obsah

Prohlášení ... 3
Poděkování ... 4
Abstrakt ... 5
Abstract ... 6
Obsah ... 7
Seznam symbolů, zkratek a termínů ... 10
Úvod ... 12
1 Teorie ... 13
1.1 Kabelová kniha a vnitřní telefonní síť TU ... 13
1.1.1 Motivace pro vytvoření webové aplikace ... 13
1.1.2 Strana budovy ... 14
1.1.3 Strana PBÚ ... 14
1.1.4 Hlavní rozvod ... 15
1.1.5 Vnitřní telefonní síť ... 16
1.2 Objektově orientované programování ... 16
1.3 PHP ... 17
1.3.1 Autoloading ... 18
1.4 Zend Framework ... 18
1.4.1 Oddělení aplikační logiky od vzhledu ... 19
1.4.2 Front Controller ... 20
1.5 HTML ... 20
1.5.1 HTML5 ... 21
1.6 Kaskádové styly ... 22
1.6.1 CSS frameworky ... 23
1.6.2 CSS kompilátory ... 24
1.7 JavaScript ... 24
1.7.1 AJAX ... 25
1.8 Správa zdrojového kódu ... 25
1.8.1 Git ... 26
1.9 Automatizace procesů ... 27
1.9.1 Apache Ant ... 28
1.9.2 Phing ... 28
1.10 Bezpečnost webových aplikací ... 28

1.10.1 XSS ... 29
1.10.2 SQL injection ... 30
1.10.3 Cross-site Request Forgery ... 31
1.10.4 Phishing ... 32
2 Analýza a návrh ... 33
2.1 Současná situace ... 34
2.2 Návrh struktury databáze ... 35
2.3 Návrh struktury aplikace ... 37
2.4 Uživatelské rozhraní ... 38
2.4.1 Výběr CSS/JS frameworku ... 38
2.4.2 Jednotný vzhled uživatelského rozhraní ... 39
2.4.3 Návrh wireframů ... 40
2.5 Bezpečnost ... 42
2.5.1 Přihlašování uživatelů ... 42
2.5.2 CSRF ... 42
2.5.3 XSS ... 43
2.5.4 SQL injection ... 43
2.5.5 Minimalizace rizikových faktorů ... 44
3 Praktické zpracování ... 46
3.1 FrontController pluginy ... 46
3.2 Controllery ... 47
3.2.1 CRUD controllery ... 47
3.3 Modely ... 49
3.4 View ... 50
3.4.1 Uživatelské rozhraní ... 51
3.5 Implementace formulářů ... 53
3.5.1 Vývoj decorátorů pro Twitter Bootstrap ... 55
3.6 Lightweight dispatch ... 55
3.7 Využití možností moderních prohlížečů ... 56
3.7.1 Využití localstorage ... 56
3.7.2 Využití content editable ... 56
3.7.3 Zrychlení načítání JavaScriptu ... 57
3.8 Filtrování ... 58
3.9 Napojení na API telefonního seznamu ... 61
3.10 Uživatelské testování ... 62
3.10.1 Výsledky testování ... 62
3.11 Automatizace některých procesů v kabelové knize ... 64

3.11.1 Historie úprav 64
3.11.2 Nastavení skriptu pro Phing 66
3.12 Použité techniky optimalizace rychlosti 67
3.12.1 Minimální množství vlastních routovacích pravidel 67
3.12.2 Classmap autoloader 67
3.12.3 Superluminal plugin 68
3.13 Instalace aplikace na server 69
3.13.1 Požadavky aplikace 69
3.13.2 Postup instalace 69
3.13.3 Nastavení databáze 70
3.13.4 Další nastavení 70
4 Závěr 71
5 Bibliografie 72

Seznam symbolů, zkratek a termínů

- **Action** – metoda controlleru, která zapouzdřuje jednu nebo více z jeho činností
- **ADN** – Additional Directory Number – další číslo linky
- **AJAX** – Asynchronous JavaScript and XML – nástroj pro asynchronní komunikaci webové stránky se serverem
- **Cookie** – textový soubor uložený u klienta, soužící k uchování informací o uživateli například pro budoucí identifikaci
- **CSS** – Cascading Style Sheet – jazyk určený k popisu způsobu zobrazení HTML, XHTML nebo XML kódu
- **CLI** – Command Line Interface – rozhraní příkazového řádku
- **Controller** – součást vzoru MVC, která zajišťuje zpracování požadavků
- **CRUD** – zkratka ze slov Create, Read, Update a Delete – používá se pro třídy, které zajišťují zmíněné akce nad modelem
- **Document root** – adresář webového serveru, kam směřují požadavky na určitou konkrétní doménu
- **DRY** – Don't Repeat Yourself – Neopakuj se! – způsob práce, kdy znovupoužíváme hotové části kódu
- **GPL** – GNU General Public License – licence určená pro svobodný software
- **Hash** – výsledek hashovací funkce, která jednosměrně převede velký objem dat do relativně malého řetězce, ukládání hashe zaručuje nemožnost dekódování původních dat
- **HTML** – HyperText Markup Language – značkovací jazyk pro tvorbu internetových stránek
- **HTTP** – HyperText Transfer Protocol – je součástí aplikační vrstvy ISO OSI modelu, umožňuje aplikacím přístup ke službám komunikačního systému
- **JS** – JavaScript
- **JSON** – JavaScript Object Notation – formát pro výměnu dat často používaný v internetových aplikacích
- **LESS** – CSS kompilátor
- **Model** – vrstva MVC vzoru obsahující doménovou logiku
- **MVC** – architektonický vzor Model-View-Controller

- **ODN** – Own Directory Number – primární číslo linky
- **Paralelka** – dvojité propojení čísla – jedno číslo je propojeno se dvěma zásuvkami
- **Phing** – PHing Is Not Gnu make – nástroj na automatizaci správy a sestavení aplikací v PHP
- **PHP** – PHP Hypertext Preprocessor – skriptovací jazyk
- **Ranžír** – propojení v hlavním rozvodu – propojuje zásuvku v budově a pozici na ústředně
- **SASS** – CSS kompilátor
- **SQL** – Structured Query Language – dotazovací jazyk pro přístup k databázi
- **SVG** – Scalable Vector Graphics – formát vektorové grafiky, založený na XML
- **TDD** – Test Driven Development – vývoj řízený testy
- **View** – prezentační vrstva MVC vzoru
- **Wireframe** – webová stránka bez funkčnosti, případně papírový model, který se používá při uživatelském testování
- **XML** – Extensible Markup Language – jazyk určený pro vytváření vlastních značkovacích jazyků využívaný také k přenosu dat mezi různými platformami
- **XSS** – Cross Site Scripting – bezpečnostní zranitelnost, která využívá načtení

Úvod

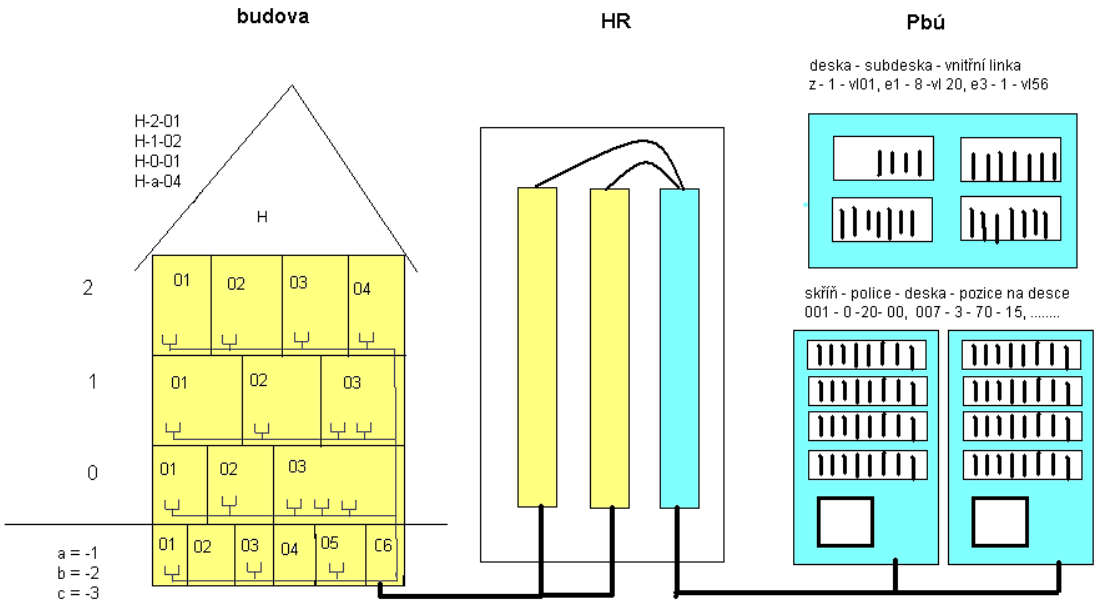
Informační systémy jsou v dnešní době nedílnou součástí správy každého komplexního systému. Mnohdy navíc vznikají iterativně a živelně, podle momentálních potřeb. S rozrůstajícím se systémem roste i složitost informačního systému. Do této situace se dostala i kabelová kniha Technické univerzity. Problémy plynoucí z nekonzistence dat a problematického sdílení byly motivací pro zadání této práce.

Cílem práce je vytvořit informační systém pro správu kabelové knihy Technické univerzity v Liberci ve formě webové aplikace. Vytvořením aplikace běžící na serveru bude vyřešen problém přístupu více osob k datům, současně lze také snáze řešit zálohování centrálně uložených dat. Aplikace umožní uživatelům v přehledném rozhraní vykonávat kroky potřebné k evidenci stavu vnitřní telefonní sítě Technické univerzity.

1 Teorie

1.1 Kabelová kniha a vnitřní telefonní síť TU

Kromě dat administrativní povahy (komu je v telefonním seznamu přiřazeno jaké číslo) je v rámci vnitřní telefonní sítě Technické univerzity nutné evidovat i další informace spíše technického rázu. K tomu slouží kabelová kniha. Ta eviduje veškerá fyzická (a i virtuální softwarová) propojení telefonních linek v rámci Technické univerzity.



Obrázek 1 Schéma vazeb v kabelové knize (zdroj: E. Augusta)

Obrázek 1 schematicky naznačuje, jak fungují základní vazby v kabelové knize. Žlutě je naznačena strana budovy a modře strana pobočkové ústředny. Ze schématu je patrné, že vedení z budovy a vedení z pobočkové ústředny se schází v hlavním rozvodu (HR).

1.1.1 Motivace pro vytvoření webové aplikace

V současné době je jedinou osobou pracující přímo s daty kabelové knihy referent telekomunikací Evžen Augusta. Tato situace se však v průběhu času může změnit a vyvstane požadavek na to, aby měly k aplikaci přístup i další osoby za účelem dohledu, kontroly nebo evidence.

Současná podoba kabelové knihy – tedy soubor ve formátu Excel – není pro spolupráci více pracovníků ideální. Neexistující referenční integrita vede k zanášení nekonzistentních dat a z kabelové knihy se tak namísto komplexního nástroje stává spíše orientační zdroj pro lidi znalé skutečné situace.

Hlavním cílem tedy je vytvoření centralizovaného úložiště dat, které bude odpovídajícím způsobem zálohováno, a bude možné, aby k němu přistupovalo více uživatelů zároveň. Jedním z pozitiv by měla být také vyšší konzistence zadávaných dat, vzhledem k vynuceným integritním omezením.

1.1.2 Strana budovy

Od zásuvek (v souboru kabelové knihy vedeny ve sloupci *hom.*) v jednotlivých budovách je propojeno vedení až do hlavního rozvodu (pozice na hlavním rozvodu je v souboru označena jako *HR celk.*). Toto vedení je neměnné a je dáno při montáži. Mění se pouze v případě, pokud dojde k přestavbám nebo rekonstrukcím.

Vedení prochází určitými místy, která jsou přesně označena. Průběh vedení je sledován po trase, aby bylo možné zjistit, pro která místa bude služba přerušena při poruše na určitém místě trasy. V souboru kabelové knihy je veden ve sloupci *mr průběh párů*.

1.1.3 Strana PBÚ

Pobočková ústředna má pevně daný maximální počet pozic. Jednotlivé ústředny se poměrně dost liší svou strukturou. Některé se dělí na desky a subdesky, jiné na skříně, police a desky. Každá pozice má však, nehledě na vnitřní strukturu, přiřazen unikátní identifikátor EQU (vedené v souboru taktéž jako *EQU*).

Vazba pozice na ústředně a pozice na hlavním rozvodu (v souboru vedena jako *MD hw. poz.*) je opět pevně dána při stavbě rozvodu a mění se prakticky pouze během rekonstrukcí nebo rozsáhlejších stavebních úprav.

Na straně PBÚ je také navázána linka. Každá pozice na ústředně může mít v každém okamžiku nastavenu maximálně jednu linku. Programování linky provádí technik softwarovým zásahem. Jedná se o jednu ze dvou proměnlivých vazeb.

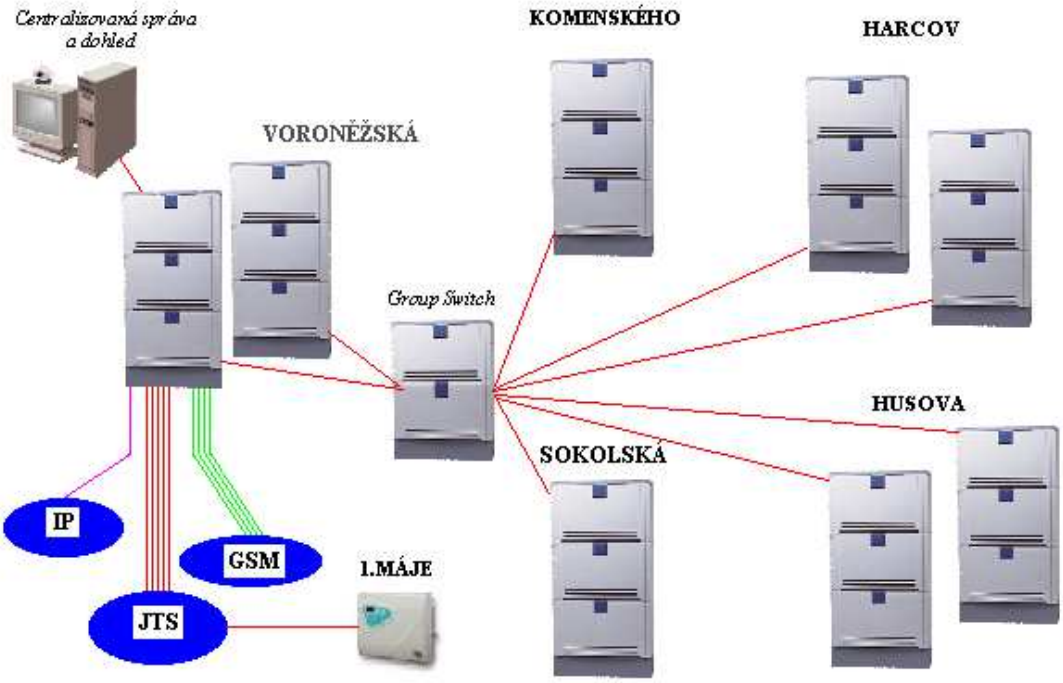
Existují také určité softwarové vazby mezi čísly. Tyto vazby se udržují ve sloupcích ADN a ODN. Linka má uvedeno ADN (Additional Directory Number) – další číslo linky, které jí odpovídá. Linka v ADN má pak vlastní záznam, u kterého je původní linka uvedena jako ODN (Own Directory Number).

1.1.4 Hlavní rozvod

Na hlavním rozvodu se střetávají vedení od budovy a od PBÚ. Jednotlivé zásuvky hlavního rozvodu se propojují tzv. *ranžírem* – na jedné straně je zásuvka od budovy a na druhé straně zásuvka od ústředny. Jedná se o druhou proměnlivou vazbu v rámci kabelové knihy. Propojení probíhá fyzickým přepojením spojovacího kabelu do jiné zásuvky.

V určitých situacích je třeba, aby jedno číslo vyzvánělo na dvou různých místech. V tu chvíli lze vytvořit tzv. *paralelku*, tedy propojit jednu pozici na ústředně s dvěma zásuvkami v budově. Opačně (aby jedna zásuvka měla přiřazena dvě čísla) něco podobného možné není.

1.1.5 Vnitřní telefonní síť



Obrázek 2 Schéma vnitřní telefonní sítě (zdroj: E. Augusta)

Obrázek 2 ukazuje rozložení jednotlivých pobočkových ústředen. Z obrázku je patrné, že se od sebe jednotlivé ústředny liší – například ústředna v budově 1. Máje je zcela jiného typu. I přesto je každá pozice identifikována pomocí EQU.

1.2 Objektově orientované programování

Objektově orientované programování je paradigma vývoje softwaru stejně jako funkcionální nebo logické programování. Existuje několik základních charakteristik, které se objevují napříč různými objektovými programovacími jazyky nehledě na konkrétní implementaci objektového modelu. Můžeme je tedy označit za zásadní.

Polymorfismus v praxi znamená, že sám objekt určuje, jaký výkonný kód bude po zavolání metody vykonán. Dva různé objekty mohou tedy zareagovat na volání stejné metody se stejnými parametry zcela odlišně. Pokud bych chtěl toto chování ilustrovat konkrétním případem v Zend Frameworku, jeví se jako ideální jednotlivé implementace adapterů pro relační databáze (`Zend_Db_Adapter_*`). Konkrétně metoda `query()`, která provede SQL dotaz na serveru, přijímá vždy stejné vstupní parametry, ale vnitřní implementace volání se liší vzhledem k tomu, že i rozhraní pro různé relační databáze se v PHP liší.

Zapouzdření způsobuje, že vnitřní implementace objektu je z vnějšku skryta za rozhraní. Díky tomu nemůže při správném návrhu vnějším zásahem z vně objektu vzniknout nekonzistence. Toho se dá s výhodou využít například v situaci, kdy chceme, aby nebylo možné měnit hodnotu některé členské proměnné. Pokud bude viditelná pouze uvnitř objektu, nastavená v konstruktoru a nevytvoříme žádnou metodu, která by její změnu umožňovala, můžeme si být jisti, že nebude v žádném případě změněna z vnějšku. Různá nastavení viditelnosti (*private*, *protected*) umožňují omezení přístupu až na úroveň konkrétní třídy.

Dědičnost. Jedna třída může být potomkem jiné třídy. Pokud neimplementujeme některé metody ve zděděné třídě, budou se volat metody třídy rodičovské. Ne všechny jazyky ovšem mají dědičnost implementovanou pomocí tříd – například JavaScript implementuje tzv. prototypovou dědičnost přímo na úrovni objektů. Místo dědění se objekt naklonuje a stane se prototypem jiného objektu. Výsledek je ale velmi podobný – prototyp přebírá obsluhu volání neimplementovaných v potomkovi.

1.3 PHP

Jedná se o interpretovaný slabě typový skriptovací jazyk speciálně určený a vyvinutý pro vývoj webových stránek. Jeho název je rekurzivní zkratkou – PHP Hypertext Preprocessor. Syntaxí se velmi podobá jazyku C, ze kterého vychází. Je šířen pod speciální licencí nazvanou PHP License (momentálně v3.01), která je uznána sdružením OSI jako open source licence. V praxi se jedná o licenci podobnou BSD licenci – tedy licenci, která nenutí vydat odvozená díla pod stejnou licencí. Díky tomu je PHP vhodné jak pro tvorbu svobodného softwaru, tak pro komerční aplikace.

Jazyk PHP umožňuje využívat velkou část principů objektově orientovaného programování. Od verze 3 jsou k dispozici třídy a objekty a v dalších verzích postupně přibývala další rozšíření. V poslední stabilní verzi 5.4.3 poskytuje jazyk PHP již plně použitelné nástroje pro objektové programování – objekty, třídy, rozhraní, abstraktní třídy, dědičnost a různé viditelnosti členských proměnných. Z dalších zajímavých jazykových konstruktů budu jmenovat anonymní funkce a jmenné prostory (od 5.3) nebo traits (od 5.4), které významně zpřehledňují kód a umožňují jeho snadné znovupoužití.

1.3.1 Autoloading

Pokud je aplikace navržena objektově, je ideální oddělit jednotlivé třídy do různých souborů. Protože PHP správu tříd v souborech nijak neřeší, je třeba před použitím třídy soubor vložit do momentálně zpracovávaného skriptu. K tomu se využívají volání `include()`, `include_once()`, kdy nenalezený soubor vyvolá jen chybu `E_WARNING`. Druhou možností je použít `require()` a `require_once()`, kdy nenalezený soubor vyvolá chybu `E_FATAL_ERROR`. Direktivy končící na „once“ načtou soubor pouze jednou během celé doby běhu skriptu.

S rostoucí velikostí aplikace je však příliš časté volání těchto funkcí nepraktické, což vedlo k zavedení metody tzv. *autoloadingu*. Do PHP byla zavedena magická metoda `__autoload()` a posléze volání `spl_autoload_register()`, které umožňuje registrovat si vlastní *autoload* funkci. Tato funkce je zavolána s parametrem názvu právě zpracovávané třídy, pokud tato třída nebyla ještě načtena. Vývojář si pak může naimplementovat vlastní dynamické načítání třídy ze souboru. Vlastní logika převodu názvu třídy na soubor je zcela v režii vývojáře, což s sebou neslo problémy při použití tříd z různých balíčků a od různých vývojářů. Reakcí na tento stav bylo sepsání PSR-0.

PSR-0[1] je standard, který vzešel ze spolupráce širokého spektra vývojářů v rámci *Framework Interoperability Group*. Jeho základem je prostě přepsání hierarchie tříd do hierarchie souborů. Jednotlivé jmenné prostory – oddělené podtržítky (před PHP 5.3) nebo skutečnými oddělovači jmenných prostorů (od PHP 5.3) – jsou přeloženy na adresáře a samotný název třídy se stane názvem souboru připojením přípony `.php`. K dispozici je také referenční implementace *autoloaderu*[2]. Nevýhodou tohoto přístupu je to, že cesty jsou vyhodnocovány v rámci cest nastavených v `include_path`. Pokud je cest v `include_path` více, je nutné je projít postupně všechny, dokud není nalezen odpovídající soubor. To s sebou přináší množství přístupů na disk a s tím související pokles výkonnosti. Toto téma dále rozebírá část 3.12.

1.4 Zend Framework

Z množství PHP frameworků, které se v posledních několika letech objevily, jsem již dříve zvolil Zend Framework. Framework je licencován pod New BSD licencí. Ta

[1] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md
[2] https://gist.github.com/221634

podobně jako PHP licence umožňuje založit na Zend Frameworku komerční software bez obav z pozdějších soudních sporů. Navíc pro přispívání do kódu frameworku musí vývojář podepsat tzv. CLA – jedná se prakticky o smlouvu, ve které se vývojář dobrovolně zavazuje, že nebude nikdy vyžadovat žádnou náhradu za kód, který do frameworku poskytne.

Důvody, které mě vedly k výběru Zend Frameworku, jsou především praktické a souvisejí s vývojem aplikace. Ve stručnosti se jedná především o velkou flexibilitu danou důsledným oddělením jednotlivých komponent a využívání návrhových vzorů a možností objektového návrhu. Neméně důležitou výhodou je velmi předvídatelný a konzistentní vývoj. Mezi hlavními verzemi je důsledně dodržována zpětná kompatibilita a je tak možné s relativně malým úsilím nasazovat nové verze, což je důležité především v případě, že nějaká aplikace má fungovat dlouhodobě bez větších zásahů. Dobré udržovatelnosti napomáhá také fakt, že celý framework je pokryt jednotkovými testy a je vyvíjen metodikou TDD. Díky tomu nedochází ke zbytečnému zanášení nových chyb během vývoje. Na nalezené chyby je napsán ověřující test, který v budoucnu zabrání regresím, tedy stavu, kdy úpravou zdrojového kódu dojde k výskytu chyby ve funkcionalitě, která před úpravou fungovala bezchybně.

V současné době probíhá vývoj druhé verze Zend Frameworku. Přináší mimo jiné implementaci založenou na principu Dependency Injection a přepracovanou architekturu s ještě vyšší flexibilitou. Bohužel je stále ve stádiu vývoje a stavět na nestabilní verzi aplikaci, která by měla výhledově být používána delší dobu, není vhodné především proto, že se do vydání stabilní verze může zásadně měnit API, a to by zásadně znesnadnilo aplikaci bezpečnostních updatů v budoucnosti. Stabilní verze by měla být k dispozici na konci léta, což je bohužel příliš pozdě.

1.4.1 Oddělení aplikační logiky od vzhledu

V Zend Frameworku jsou data, aplikační logika a vzhled odděleny do tří částí. Jedná se o implementaci architektonického vzoru Model-View-Controller. To znamená, že přijde-li požadavek na *Controller*, ten vnitřní logikou rozhodne, jaká data jsou třeba, vyžádá si je od *Modelu* a předá je do *View*. Základním a nejdůležitějším prvkem je právě oddělení datové (Model) a prezentační (View) části (FOWLER, 2003, s. 331).

1.4.2 Front Controller

Návrhový vzor *Front Controller* boří zažitou praxi (nejen) PHP aplikací, kdy existuje množství oddělených vstupních bodů, které uživatel používá během práce s aplikací, jak uvádí (ZANDSTRA, 2010, s. 235). Aplikace používající *Front Controller* má jediný vstupní soubor a v něm aplikační logika rozhoduje o tom, jak bude probíhat zpracování. To je zásadní rozdíl proti častému PHP přístupu, kdy je pro každou akci v aplikaci vyhrazen jeden soubor (zobrazení hlavní stránky, přidání položky, přehled, atp.). Výhodou tohoto přístupu je, že je možné snadno provádět nastavení často používaných zdrojů (databáze, session, přihlášení uživatele) na jednom místě aplikace.

1.5 HTML

HyperText Markup Language je značkovací jazyk pro tvorbu webových stránek. Patří do rodiny jazyků založených na SGML. Základní stavební jednotkou HTML je tag. Tagy mohou obsahovat atributy a jim odpovídající hodnoty. Syntaxe HTML je velmi volná a prakticky jediným omezením je užití znaků *větší než* a *menší než* (`<tag>`) k ohraničení tagů. Velikost písmen, užití uvozovek, apostrofů, zákaz křížení tagů (`<b><i></b></i>`), použití ukončovacího tagu – to všechno patří mezi nepovinná pravidla. Následující dva zápisy jsou tedy v HTML ekvivalentní:

Kód 1 Ekvivalentní zápisy HTML

```
<!-- obecně přijímaný standard -->
<p style="color:blue"><b>Tento <i>odstavec</i></b> obsahuje <a
href="index.html">odkaz</a></p>
<ul>
<li>Odrážka</li>
</ul>

<!-- stále funkční-->
<P style=color:blue><b>Tento <i>odstavec</b></i> obsahuje <a
HREF=index.html>odkaz</A></p>
<UL>
<li>Odrážka
```

Tato vlastnost způsobuje množství problémů při strojovém zpracování HTML. Existuje snaha tuto volnost syntaxe odstranit – o to se snaží například jazyk XHTML, založený na XML. Naneštěstí se příliš neprosadil a ve výsledku zvítězil princip volnější syntaxe reprezentovaný HTML ve verzi 5. Ta existenci XHTML formálně uznává, avšak nevynucuje její použití ve všech případech a vnikají tak vlastně paralelní větve HTML5 a XHTML5 (HICKSON, 2012).

Volnost syntaxe s sebou kromě problémů přináší i výhody. Jednou ze zásadních výhod je, že parser HTML ignoruje neznámé tagy. Tato vlastnost umožňuje snadno přidávat do jazyka novinky za předpokladu, že je zachována základní zpětná kompatibilita. Díky tomu bylo možné začít v praxi používat HTML5 a postupně přidávat nové vlastnosti.

1.5.1 HTML5

Pojem HTML5 není úplně přesně definován a jeho význam závisí na kontextu, v jakém ho používáme. Obecně vžitý význam je, že HTML5 je rodina technologií – především těch, které začaly být šířeji implementované a přijímané v době draftu HTML5. Jedná se vlastně o značkovací jazyk HTML5, kaskádové styly – CSS3 (kaskádové styly rozebírá kapitola 1.6) a některá JavaScriptová API (o JavaScriptu pojednává kapitola 1.7).

Specifikace značkovacího jazyka HTML5 je ve stavu Working Draft (HICKSON, 2012). HTML5 draft není nijak závazný, a tudíž se jeho konkrétní implementace v prohlížečích mohou poměrně značně lišit. Je zpětně kompatibilní se staršími specifikacemi HTML, bere si to nejlepší ze současné implementace a zakazuje některé obskurní praktiky HTML4, které vycházely ze SGML (mj. procesní instrukce a shorthand zápis tagů (VAN KESTEREN et. al., 2012)). Minimální HTML5 dokument by mohl vypadat takto:

Kód 2 Minimální dokument (LAWSON, 2010)

```
<!doctype html>
<html lang=en>
<meta charset=utf-8>
<title>blah</title>
<body>
<p>I'm the content
```

Nejedná se přímo o minimální dokument, ale spíše o minimální smysluplný bezpečný dokument.

Rodina technologií HTML5 přináší množství funkčních vylepšení, především ve spojení s JavaScriptem. Mezi zajímavé vlastnosti patří například **localstorage** umožňující prakticky neomezeně dlouhé uchovávání často používaných dat u klienta bez nutnosti se pokaždé dotazovat na server. Mezi další vlastnosti patří **canvas**, který

umožňuje JavaScriptem vykreslovat rastrovou grafiku, a tagy **audio** a **video**, které umožňují přehrávání audiovizuálního obsahu na ve stránce. Zatím ale nepanuje shoda na kodeku pro zpracování videa – existuje opensource kodek Ogg Theora, licencovaný H.264 a Googlem vyvíjený WebM[3]. Další vlastností je **geolokační API**, které umožňuje stránce získat přístup ke geografické pozici – ať se jedná o pozici z připojené GPS nebo triangulaci pomocí síly signálu wifi sítí. **Sémantické tagy** – `header`, `footer`, `section`, `article` – nepřináší žádná zvláštní funkční vylepšení, ale měly by napomoci strojovému zpracování zavedením jednotné sémantiky do celkové struktury stránky. **Nové formulářové prvky** mají určitý, přesně daný typ obsahu a nahrazují používaná JavaScriptová rozšíření současných prvků – jmenovitě se jedná například o `datetime`, `number`, `email` nebo `url`. Součástí těchto prvků je i integrovaná validace. Dále **Inline SVG** umožňuje vykreslovat vektorové kresby (SVG) přímo ve stránce.

Ze zajímavých vlastností CSS3 zmíním **širokou podporu průhlednosti** prakticky u všech elementů a pro všechny stylovatelné části. Dále bych vyzdvihl **zadávání barev v různých barevných prostorech** včetně alfa kanálu (*RGB*, *HLS*, *RGBA*, *HLSA*) a **atributové selektory**, které umožňují ještě konkrétnější definici kaskádových stylů – např. `a[href^="http://"]`. Další vlastností CSS3 jsou **automaticky generované stíny a barevné přechody**, díky kterým lze snížit datovou náročnost stránek, jelikož minimalizují potřebu obrázků bez informační hodnoty. Problémem zůstává nesourodá implementace v prohlížečích, a tak hlavní oblastí použití jsou stejnorodé systémy – např. prohlížeč Safari na operačním systému iOS, používaném především na mobilních zařízeních od společnosti Apple. S použitím pro mobilní zařízení pak souvisí **media-queries** – aplikace jiné části stylopisů na základě vlastností zobrazovacího zařízení – rozlišení, orientace atp.

1.6 Kaskádové styly

Podle oficiální definice na webu konsorcia W3C (W3C, 2012) jsou kaskádové styly jazyk pro popis vzhledu webových stránek. Umožňují uživateli přizpůsobit zobrazení pro různé typy zařízení jako například velké obrazovky nebo naopak malé obrazovky nebo tiskárny. CSS je nezávislé na HTML a může být použito s libovolným značkovacím jazykem založeným na XML. Oddělení HTML od CSS umožňuje

[3] Zvláštní na této situaci je, že si Google a Microsoft vzájemně vyvíjí rozšíření – Google vyvíjí WebM rozšíření pro IE9 a Microsoft vyvíjí H.264 rozšíření pro Chrome.

jednodušší údržbu webů, sdílení stylopisů napříč více stránkami a úpravu stránek pro různá prostředí. Pro toto chování se vžil termín oddělení struktury (resp. obsahu) od vzhledu.

Kaskádové styly přiřazují jednotlivým kombinacím tagů, tříd, id a atributů určité vlastnosti – například velikost textu, barvu pozadí nebo odsazení. Jak již z názvu vyplývá, jsou aplikovány na obsah kaskádovitě. Pravidla jsou tedy nastavována od těch nejobecnějších k nejkonkrétnějším a od rodičů k dětem, přičemž později přiřazené pravidlo přepisuje pravidlo dřívější. Na příkladu (SWISHER, 2011) je patrné, jakým způsobem CSS funguje. Pravidla (Kód 3) aplikovaná na HTML (Kód 4) způsobí vypsání modrého podtrženého odstavce, přičemž první písmena jsou červená a také podtržená.

Kód 3 Ukázková CSS pravidla

```
strong {color: red;}
p {color: blue; text-decoration: underline;}
```

Kód 4 Ukázkový HTML kód

```
<p><strong>C</strong>ascading <strong>S</strong>tyle
<Strong>S</strong>heets</p>
```

Protože tag `strong` nemá definované pravidlo pro podtržení, použije se obecnější pravidlo z předka. Protože však má definované pravidlo pro barvu, tak se modrá barva nezdědí. Přidáním `text-decoration: none` do definice `strong` bychom mohli podtržení u červených písmen snadno odstranit.

1.6.1 CSS frameworky

Na rozdíl od programovacích jazyků není v CSS snadné vytvořit framework. Příčin je několik a nejmarkantnější z nich je neexistence nějaké složitější logiky, která by umožnila upravování definic uživateli frameworku. CSS frameworky a nástroje se tak omezovaly především na nízkoúrovňové chování – například na definici mřížky[4],

[4] http://960.gs/

nastavení jednotného výchozího vzhledu napříč prohlížeči[5] nebo na základní typografii[6].

S příchodem kompilátorů, jako je LESS nebo SASS, se ale situace zásadně mění, protože díky využití proměnných a dynamických výpočtů lze prakticky vytvořit celý vzhled webu. Lze například upravit vše od barev přes velikost písma až k obrázkům na pozadí, přičemž stačí několik snadných zásahů do kódu a není nutné přepisovat všechny výskyty ve všech souborech.

1.6.2 CSS kompilátory

Kompilátory CSS jsou poměrně novým fenoménem. Jejich principem je rozšíření jazyka CSS o další konstrukty. Tyto konstrukty jsou pak ve fázi kompilace nahrazeny platnou CSS syntaxí. To umožňuje zavedení proměnných, aparátu základních matematických operací nebo funkcí pro úpravu barev. Také umožňují pokročilou manipulaci se samotnými pravidly – jejich zanořování, parametrizace nebo vytvoření tzv. mixinů. To zásadním způsobem usnadňuje práci s rozsáhlými CSS definicemi.

Mezi nejznámější kompilátory patří LESS, SASS, HSS nebo Stylus. LESS a Stylus mají referenční implementaci napsanou v JavaScriptu, což s sebou nese výhodu v podobě platformní nezávislosti. A dokonce lze tyto nástroje používat za pomocí JavaScriptového kompilátoru přímo ve stránce – to s sebou však nese zvýšené nároky na výkon. Ale pokud je na serveru k dispozici interpret JavaScriptu (např. NodeJS), tak je možné kompilaci provádět před odesláním souboru klientovi. SASS je implementován v Ruby a HSS v nepříliš známém jazyku Neko. I pro SASS však existuje implementace v JavaScriptu[7], zjevnou nevýhodou však je, že je vždy o něco pozadu za aktuální verzí a není jisté, že se bude vždy chovat stejně jako referenční implementace. Proto mi její použití nepřijde vhodné.

1.7 JavaScript

Jedná se o slabě typový skriptovací jazyk s prototypovou dědičností. Jeho nejčastější použití je pro dynamickou manipulaci s HTML dokumentem, avšak je používán i pro jiné účely. V poslední době se jedná především o jeho použití na

[5] http://meyerweb.com/eric/tools/css/reset/
[6] http://baselinecss.com/
[7] https://github.com/visionmedia/sass.js/

serveru[8]. Vychází ze specifikace jazyka ECMAScript podobně jako například ActionScript používaný pro Flash.

1.7.1 AJAX

S využitím JavaScriptu úzce souvisí termín AJAX. Jedná se o programovací techniku, kdy pomocí JavaScriptu vytváříme a odesíláme asynchronní požadavky na server a vrácená data opět zpracujeme pomocí JavaScriptu. Pojmenování *Asynchronous JavaScript and XML* není úplně přesné, protože pro přenos dat mezi serverem a JavaScriptem u klienta můžeme používat i jiné formáty dat než XML. Často je používán JSON nebo dokonce přímo HTML. AJAX umožňuje načíst ze serveru data nebo část HTML stránky bez nutnosti přenášet celou stránku. Podle (GARRETT, 2005) *„běžný web funguje na principu start-stop-start-stop, což AJAX nabourává vytvořením prostředníka – AJAX enginu, který je vřazen mezi uživatele a server. I když by se mohlo zdát, že další vrstva aplikaci musí nutně zpomalit, opak je pravdou*“ (vlastní překlad). AJAX se masově rozšířil spolu se službami společnosti Google (Maps, Gmail, …).

1.8 Správa zdrojového kódu

Při vývoji rozsáhlého systému vývojář časem narazí na potřebu sledování změn ve zdrojovém kódu, aby bylo možné se v případě potřeby vrátit k některé z předchozích verzí. Ještě markantnější to je, pokud se na vývoji podílí celý tým vývojářů – je třeba zajistit nejlépe automatizovaným procesem, aby si navzájem nepřepisovali změny v souborech a měli k dispozici vždy aktuální verzi.

Nejjednodušším způsobem správy změn je verzování pomocí tzv. snapshotů. To v praxi znamená, že vývojář ručně nebo automaticky v určitých intervalech vytváří zálohy své pracovní kopie projektu. Tento přístup má hned několik nevýhod. Jednak je poměrně náchylný k selhání, pokud nejsou zálohy vytvářeny automaticky. Pokud si vývojář v důležitém okamžiku zapomene vytvořit zálohu, není již možnost jak se vrátit k původní verzi. Mimo to je tento postup značně neefektivní, neboť jsou stejná data uložena několikrát – v každé kopii. V neposlední řadě tento systém není vhodný, pokud na projektu spolupracuje více vývojářů.

[8] http://nodejs.org/

Další možností, která byla v minulosti využívána, je sdílené úložiště zdrojových kódů – typicky na síťovém disku. To ovšem také není ideální, především proto, že se může snadno stát, že si vývojáři navzájem přepíší úpravy. Navíc není možné se vrátit ke starší verzi, což je jeden ze základních požadavků.

Tento problém řeší různé SCM (Source Code Management) nástroje. Ty sledují změny v souborech a ukládají jejich jednotlivé revize. Měly by také poskytovat další nástroje pro složitější vývojové scénáře jako například dělení na vývojové větve, vzájemné slučování větví, zamykání souborů nebo tagování význačných verzí (typicky produkčních).

Pro nástroje sledující verze souborů se vžila zkratka VCS (Version Control Systems). Mohou být centrální, nebo distribuované – podle toho rozlišujeme centralizované VCS (CVCS), nebo distribuované VCS (DVCS). Mezi CVCS patří například nástroje Concurrent Versions System[9] (CVS) nebo Subversion[10] (SVN), mezi DVCS patří například Git[11] nebo Mercurial[12]. Pro správu zdrojového kódu aplikace kabelové knihy jsem použil DVCS Git.

1.8.1 Git

Git je distribuovaný systém na správu zdrojového kódu. Jeho hlavní předností – kromě toho, že je distribuovaný – je velmi snadný vývoj ve větvích. Umožňuje pro každou funkcionalitu vytvořit zvláštní větev a větve následně slučovat, přesouvat mezi nimi kód nebo je aktualizovat (*rebase*, *merge*) změnami v jiných větvích. Zjistil jsem, že pro vývoj softwaru na zakázku je vhodnější než SVN. Především proto, že není závislý na jednom centrálním poskytovateli repositáře. Každý uživatel má u sebe vlastní repositář. Ze svého repositáře může změny odesílat to jednoho nebo více dalších repositářů (*push*), případně od nich změny přijímat (*fetch*, *pull*). Tyto repositáře mohou být jak sdílené vzdálené repositáře na serveru, tak repositáře konkrétních vývojářů v místní síti.

Tímto způsobem lze snadno překonat problém, se kterým jsem se potýkal při vytvoření telefonního seznamu. SVN repositář aplikace chtěl mít Ing. Kretschmer pod

[9] http://www.nongnu.org/cvs/
[10] http://subversion.apache.org/
[11] http://git-scm.com/
[12] http://mercurial.selenic.com/

kontrolou, a proto byl přesunut na počítač v síti univerzity. To způsobilo množství problémů s přístupem do sítě TU z Internetu. S Gitem je možné revize vytvářet lokálně, bez připojení k internetu, případně si je odesílat na vlastní server s repositářem a do repositáře v síti TU je odesílat hromadně po dokončení nějakých logických celků. To velmi zjednoduší následný vývoj aplikace po nasazení.

1.9 Automatizace procesů

Udržování procesů souvisejících se správou a vývojem aplikace je vždy pro administrátory velkou výzvou, neboť se často jedná o opakující se práci, která je náchylná na chybu lidského faktoru. Jedná se například o různé čištění logů, zálohování databáze, kontroly konzistence nebo spouštění testů.

Řešením tohoto problému je specifikace těchto často se opakujících činností tak, aby je bylo možné spouštět automaticky, bez zásahu administrátora. Jak píše (CAMPI et. al., 2008, s. 11), nejdůležitějším aspektem automatizovaného řešení je jeho reprodukovatelnost. Skript musí být napsaný tak, aby fungoval nejen na vývojovém serveru, ale abychom s jeho pomocí byli schopni ty samé úkony provést i na produkčním serveru. Campi zmiňuje i další vlastnosti ideálního řešení – ověřitelnost stavu, ve kterém se systém nachází, automatické opravy problémů, bezpečnost automatického řešení a další.

V případě operačního systému unixového typu bývají často řešením shellové skripty. Jsou napsané na míru projektu a typicky ztrácí jednu důležitou vlastnost – přenositelnost. Nelze je snadno spustit na jiném operačním systému (např. na vývojovém stroji s MS Windows). Tím vzniká nutnost oddělit od sebe skripty pro produkci a pro vývoj, nebo sjednotit platformy. To není ideální. Navíc často nejsou shell skripty nijak zvlášť dokumentovány a pro neznalého člověka nejsou čitelné.

Dalším častým řešením jsou tzv. make skripty, které zmiňuje (ZANDSTRA, 2010, s. 407), tedy konfigurační skripty využívané programem make. Jedná se vlastně o zabalení předchozího řešení do utility určené k sestavování projektů. Ale stejně jako předchozí řešení neumožňuje snadnou přenositelnost (i když možnosti existují[13]).

[13] http://cs.nyu.edu/rgrimm/teaching/fa05-oop/windows-make.html

1.9.1 Apache Ant

Apache Ant je nástroj napsaný v jazyku Java (a tedy platformě nezávislý), který konfigurace sestavení ukládá ve formátu XML. Kromě multiplatformnosti je výhoda také v tom, že existuje množství jednotlivých nástrojů, které se s Apache Ant dají použít. Jedná se o nástroje pro přejmenovávání, balení, kopírování na vzdálená úložiště atp. Navíc soubor XML umožňuje generovat dokumentaci částečně automaticky pomocí XSLT transformace. Přesto všechno není Ant ideální pro nasazení společně s aplikací kabelové knihy. Ke svému běhu vyžaduje běhové prostředí jazyka Java (JRE), které nemusí být na serveru dostupné.

1.9.2 Phing

Jako ideální řešení se jeví Phing. Jedná se o systém vytvořený po vzoru Antu, avšak implementovaný v jazyce PHP. Díky tomu lze i samotné dílčí úkoly psát v PHP. A jak uvádí (ZANDSTRA, 2010, s. 408) – pokud máme na serveru spuštěnou aplikaci v PHP, tak přítomnost řádkového interpretu PHP je sázka na jistotu. Phing navíc zapadá do celkové infrastruktury PHP. Je to poznat i na samotné instalaci tohoto nástroje, kterou je možné provést pomocí nástroje PEAR, který bývá součástí většiny instalací PHP.

Kód 5 Instalace nástroje Phing

```
pear channel-discover pear.phing.info
pear install phing/phing
```

Phing je řízen *build skriptem*. Jedná se o běžný XML soubor, který obsahuje definice jednotlivých úkolů (*targety*), jejich závislostí a parametrů. Jednotlivé *targety* jsou pak volány jako parametry volání z příkazové řádky.

Kód 6 Ukázka volání Phingu pro sestavení CSS z LESS

```
phing buildcss
```

1.10 Bezpečnost webových aplikací

Tak, jak se rozvíjí internet a webové aplikace, rozvíjí se i kriminalita v této oblasti. Nelze proto přehlížet otázku bezpečnosti webových aplikací. V době, kdy

existují automatizované nástroje, kterými lze testovat XSS[14] nebo SQL injection[15] zranitelnosti, dokáže s pomocí Google a běžné znalosti internetu nezabezpečený web napadnout i zručnější student střední školy.

Vzhledem k faktu, že aplikace kabelové knihy běží v omezeném režimu a bez autorizace je práce s ní nemožná, není nebezpečí útoků tak markantní. Než může útočník využít XSS nebo SQL injection zranitelnosti, musí nejprve proniknout do systému. I přesto jsem se snažil bezpečnost aplikace nepodcenit.

1.10.1 XSS

Zranitelnost XSS umožní útočníkovi upravit kód HTML stránky a tím pádem také vykonat libovolný kód nebo jen snížit důvěryhodnost webu[16]. Může se jednat buď o změnu dočasnou, nebo trvalou. Dočasné změny je možné využít např. k obejití tzv. *Same Origin Policy*[17]. Pokud je změna dočasná, ale projeví se například změnou URL, dá se to využít k útoku typu *Session Hijacking*[18] (takovému útoku byl vystaven např. i server AbcLinuxu[19]). Oběť může dostat inkriminovaný odkaz například emailem, okamžitě po otevření stránky je útočníkovi odeslán obsah cookie oběti – pokud je oběť přihlášena, tak může útočník vytvořením stejné cookie posílat požadavky jménem uživatele, aniž by musel zjišťovat přihlašovací heslo oběti.

V případě trvalé změny je situace ještě nebezpečnější, neboť není třeba nalákat oběť, aby navštívila nějak speciálně upravenou stránku. Kód na stránce je trvalý a může tak snadno útočit na každého návštěvníka.

Obranou proti XSS je dodržování poučky *„filter input escape output“* – filtrovat vstup, ošetřovat výstup. Problémem však zůstává správné určení kontextu. A to jak u vstupních, tak u výstupních dat. Například HTML kód ve jménu uživatele je nežádoucí, avšak ve formátovaném obsahu poznámky je naopak žádoucí. To samé platí pro výstup, protože jiné ošetření je nutné pro výstup do HTML, jiné pro výstup do JS. Dvojnásobné ošetření je poté třeba pro výstup do JS, který vytváří HTML kód.

[14] http://a4apphack.com/featured/secfox-xssme-automated-xss-detection-in-firefoxpart-3
[15] http://nosec.org/en/productservice/pangolin/
[16] Příklad využití dočasné XSS zranitelnosti ke snížení důvěryhodnosti webu http://zpravy.idnes.cz/video-web-cssd-napadli-utocnici-vnutili-mu-loupezniky-z-pohadky-p8a-/domaci.aspx?c=A090522_214640_domaci_anv
[17] http://www.w3.org/Security/wiki/Same_Origin_Policy
[18] https://www.owasp.org/index.php/Session_hijacking_attack
[19] http://www.abclinuxu.cz/blog/zatial_bez_mena/2007/7/xss-plus-session-hijacking-hack-abclinuxu.cz

1.10.2 SQL injection

Zranitelnost SQL injection využívá speciálního významu některých znaků v jazyku SQL a umožňuje útočníkovi do legitimního dotazu podstrčit škodlivý kód. Rozlišujeme zranitelnosti *inline* (tj. přímo v dotazu) a *terminating* (tj. předčasným ukončením dotazu) (CLARKE et. al., 2009, s. 62,68). Pokud není na výstupu patrná změna (i pokud SQL injection proběhlo úspěšně), nazýváme takový útok *blind SQL injection* (CLARKE et. al., 2009, s. 219).

Na následujícím bloku kódu se pokusím demonstrovat, jak tato zranitelnost funguje. Předpokládejme, že se jedná například o přihlašovací pole.

Kód 7 Kód náchylný k SQL injection

```
$user = $_GET['username'];
$password = $_GET['password'];
$select = "SELECT COUNT(*) FROM admins WHERE username = '$user'
    AND password = '$password'";
```

Kód kontroluje, jestli v databázi existují záznamy, které odpovídají zadaným hodnotám. Výsledný dotaz v běžném případě bude odpovídat očekávanému chování.

```
SELECT COUNT(*) FROM admins WHERE username = 'karel@novak.cz'
    AND password = 'tajneheslo'
```

Předpokládejme však následující situaci: Do pole uživatele vyplníme následující řetězec a do pole hesla dáme libovolný řetězec:

```
' OR 1 --
```

Výsledný SQL dotaz tedy bude:

```
SELECT COUNT(*) FROM admins WHERE username = '' OR 1 -- ' AND password =
'heslo'
```

Sekvence dvou pomlček za sebou má v MySQL speciální význam a označuje komentář. Server tak vykonávání dotazu zastaví předčasně a vrací všechny řádky tabulky `admins`. Právě jsme využili zranitelnosti k provedení *terminating* SQL injection útoku. Obranou proti tomuto útoku je buď ošetřování vstupních dat, která vkládáme do

dotazů nebo použití parametrizovaných dotazů, kdy dochází k ošetření na straně serveru. Do hloubky se tématu SQL injection věnuje například již zmiňovaná kniha *SQL Injection Attacks and Defense* (CLARKE et. al., 2009).

1.10.3 Cross-site Request Forgery

CSRF je typ útoku na webovou aplikaci, který využívá samotného principu fungování HTTP komunikace. Principem útoku je vytvoření HTTP požadavku neočekávaným způsobem (například při zobrazení obrázku na úplně jiném webu) a provedení akce, aniž by ji mohl uživatel ovlivnit. Problematické na tomto útoku je, že server není schopen odhalit, zda uživatel opravdu požadavek inicioval, nebo zda byl odeslán bez jeho vědomí.

K úspěšnému útoku je potřeba splnit několik podmínek:

- útočník zná (případně vhodně odhadne) strukturu webu, na který útočí
- aplikace nekontroluje, zda byl požadavek iniciován zevnitř, či vně aplikace
- pokud aplikace vyžaduje přihlášení, tak musí být uživatel přihlášen (tj. pracuje s aplikací nebo se po skončení práce neodhlásil)
- uživatel navštíví web, který je využívaný k iniciaci útoku – přičemž může jít o:
    - za tím účelem vytvořený web (pornografie, warez, atp)
    - běžný web napadený pomocí jiné zranitelnosti (XSS, SQL injection)
- případně v dobré víře klikne na odkaz v emailu nebo v souboru

Velkým nebezpečím je i to, že CSRF zranitelnosti lze využít i k útoku na intranetové systémy.

Zranitelnost pomocí CSRF byla dokonce v roce 2008 nalezena i na webu banky ING Direct (ZELLER et. al., 2008, s. 5) navzdory faktu, že banky jsou považovány za instituce dbající na bezpečnost nejvíce. O tom, že CSRF je i dnes stále rozšířenou hrozbou svědčí nedávno uveřejněný článek (HOMAKOV, 2012), který upozorňuje na CSRF zranitelnosti v množství webových aplikací. Autor článku je známý především díky odhalení zásadní bezpečnostní chyby v systému na správu zdrojových kódů GitHub, který byl mnohými považován za vzor správně napsané webové aplikace.

Obrana před CSRF není snadná, protože se nejedná o žádnou nestandardní činnost. Jde o běžný HTTP požadavek s tím rozdílem, že se vykonal bez vědomí

uživatele. Často je jako obrana před CSRF uváděno řešení v podobě kontroly odkazující stránky (tzv. referer). To ovšem není dostatečné – pole lze podvrhnout – a navíc to může zkomplikovat použití aplikace za restriktivním firewallem, který toto pole z bezpečnostních důvodů neposílá. Jedinou dostatečnou obranou proti CSRF je tak posílání bezpečnostního tokenu. Token je vygenerován při požadavku na zobrazení formuláře, při odeslání formuláře je odeslán ve skrytém poli a na serveru proběhne kontrola, zda odeslaný token odpovídá vygenerovanému. Pokud chybí nebo neodpovídá, nepřišel uživatel ze stránky, která token vygenerovala, ale odjinud, a požadavek je zablokován.

Bohužel tato ochrana může vést i k netypickému chování – například pokud je při zpracování formuláře přerušeno spojení a uživatel později okno obnoví s novým odesláním formuláře, tak token již není platný a požadavek je zablokován. To je bohužel daň za bezpečnost. Z mého pohledu se jako nejrozumnější ochrana proti CSRF jeví vysouvací lišta v prohlížeči „skutečně chcete odeslat požadavek na server XY?“, jak to navrhuje Homakov (HOMAKOV, 2012).

1.10.4 Phishing

I v případě, že je aplikace velmi dobře zabezpečena, je od uživatelů vyžadováno složité a dlouhé heslo a v aplikaci nejsou zjevné bezpečnostní chyby, stále existuje jedna další zranitelnost. Nezkušený uživatel.

Phishing cílí právě na takové uživatele. Jedná se o praktiku, při které útočník získá od uživatelů přihlašovací údaje, který je poskytne v dobré víře, že se jedná o legitimní požadavek. A to buď v návaznosti na planou výhružku „pošlete nám jméno a heslo, abychom mohli ověřit, že jste skutečným majitelem účtu“ nebo vytvořením kopie původního webu na podobné doméně, kam uživatel údaje zadá, aniž by cokoli tušil.

Řešením je informování uživatelů o takové hrozbě a hlavně opakované ujišťování, že nikdy nepožadujeme posílání údajů mailem, neposíláme přímé odkazy na přihlášení atp. Aplikace kabelové knihy však tímto typem útoku asi nebude příliš ohrožena. Cílem jsou častěji velké projekty s mnoha uživateli, kde je větší šance na nalezení důvěřivého uživatele.

2 Analýza a návrh

Aplikace telefonního seznamu, kterou jsem zpracovával v rámci své bakalářské práce, vyžadovala z výkonnostních, ale i funkčních důvodů použití návrhu databáze, který nerespektuje skutečné fungování přiřazování linek a osob. Osobě je přiřazena kancelář, funkce a linka. Ve skutečnosti však existuje mezi kanceláří a linkou na úrovni kabelové knihy transitivní závislost.

V kapitole 1.1 jsem zmínil, že kabelová kniha obsahuje záznam pro všechny existující propojení mezi pobočkovými ústřednami a zásuvkami v jednotlivých budovách. Celkově se může jednat o tisíce položek. Při vytváření telefonního seznamu jsem s propojením na kabelovou knihu počítal, a proto jsem vytvořil zvláštní tabulku, která obsahuje pouze čísla linek. Tento krok se během analýzy pro zpracování kabelové knihy ukázal jako zbytečný. Propojení linek přes kabelovou knihu by vedlo k nahrazení této jedné vazby komplikovanější vazbou vázanou pomocí identifikačního kódu kanceláře M:N vazbou hlavního rozvodu na konkrétní pozici na pobočkové ústředně. Každý databázový dotaz by se kvůli tomu výrazně zkomplikoval a zpomalil. Navíc by neexistovala možnost, jak uživateli přiřadit číslo, které není vedené v kabelové knize (např. mobilní telefon).

Existuje jisté riziko, že údaje v současné verzi kabelové knihy nejsou přesné, neboť nebyla nijak ověřována jejich referenční integrita. Proto by nasazení této změny prakticky znemožnilo běžné používání aplikace telefonního seznamu, který je již na Technické univerzitě nasazen a používán. A to nejméně do doby, než budou data telefonního seznamu ověřena a opravena.

Při prvotní analýze jsem se tedy rozhodl kabelovou knihu od telefonního seznamu oddělit. Díky tomu se dají data z kabelové knihy ověřovat postupně a nijak neovlivní současnou funkčnost telefonního seznamu. Přestože budou oddělené na úrovni aplikačního i databázového návrhu, některá data jsem se rozhodl sdílet – především data, která se složitě synchronizují a často mění (konkrétně číselníky s kódy kanceláří). V případě změn ovlivňujících telefonní seznam využiji existující API telefonního seznamu k iniciování změny.

2.1 Současná situace

Návrhu aplikace předcházela analýza současného stavu kabelové knihy. Data kabelové knihy jsou vedena v jednom souboru ve formátu Microsoft Excel. V listu *kabelová kniha* jsou uvedeny informace o propojení přes hlavní rozvod a informace o zásuvkách na straně budovy, v ostatních listech je vedena část na straně pobočkových ústředen. Nejsou použity žádné odkazy na jiné listy, a tak například číslo linky uvedené v listu *kabelová kniha* není nijak synchronizované s číslem linky uvedeným na listu pobočkové ústředny. To podle mého názoru musí velmi ztěžovat jakékoli úpravy a především vést k zanášení chyb.

Analýza současného fungování systému kabelové knihy byla velmi složitá, zejména proto, že nemám dostatečně hluboké teoretické znalosti principů fungování vnitřní telefonní sítě, resp. její konkrétní implementace na Technické univerzitě. Většinu informací jsem tedy získal komunikací s Evženem Augustou z Referátu telekomunikací rektorátu Technické univerzity a analýzou současné verze kabelové knihy, kterou mi i s doplňujícími vysvětlujícími údaji poskytl.

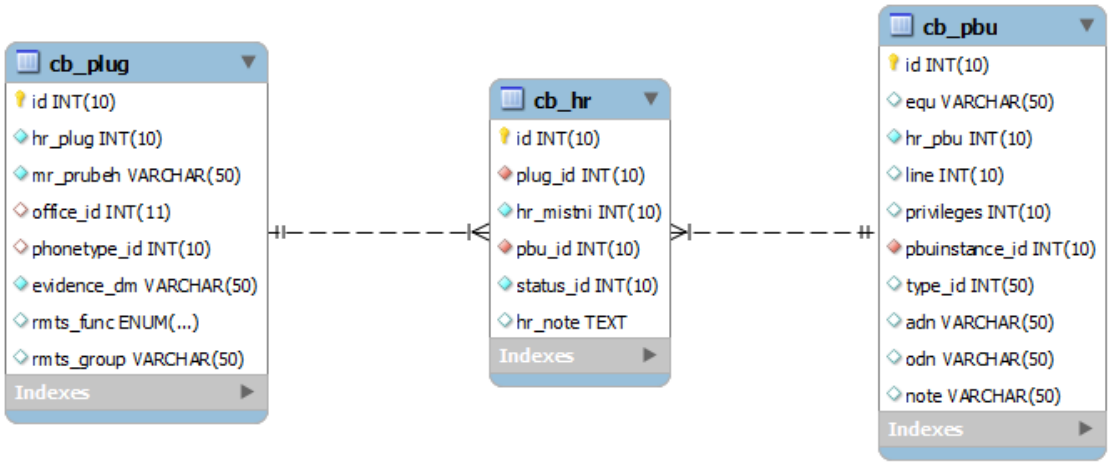
I přesto, že výsledný návrh nepůsobí na první pohled příliš složitě, jednotlivé iterace návrhu specifikace, které k němu vedly, byly velmi komplikované. Rád bych to ilustroval na jedné konkrétní situaci. V jednom z dokumentů, které jsme si během analýzy vyměnili, bylo napsáno: „*Na zásuvku v místnosti se SW propojuje číslo linky. Může nastat situace, kdy člověk odejde z kanceláře a vezme si číslo s sebou = přeprogramuje se vazba mezi kanceláří a číslem linky. A dotyčnému se jeho číslo nastaví v nové kanceláři = naprogramuje se vazba nová místnost => linka*“. Z této věty by se mohlo zdát, že existuje *funkční závislost linky na zásuvce*. Není tomu tak. Pokud je třeba přesunout linku do jiné místnosti, tak mohou nastat dvě situace. Softwarově se přeprogramuje pozice na ústředně na jinou linku – tedy změní se vazba *pozice na ústředně* a *linky*. Druhou možností je pak fyzicky změnit *ranžír* na hlavním rozvodu tak, aby byla zvolená pozice na ústředně napojena do jiné kanceláře. Toto je pouze jeden z mnoha případů, kdy bylo velmi problematické specifikovat správné chování aplikace v konkrétních případech.

2.2 Návrh struktury databáze

Během návrhu databáze jsem narážel často na to, že některé vazby nebyly na první pohled patrné, nebo naopak na situaci, kdy například hodnota, která se zdála být primárním klíčem, nemohla být jako primární klíč použita. Nakonec jsem šel cestou vytvoření zvláštního sloupce `id` (tzv. umělý primární klíč), který funguje jako primární klíč a nemá žádný smysl v návaznosti na ostatní data. To s sebou nese i určitý přínos, co se týče bezpečnosti – více se tomu věnuji v kapitole 2.5.5.

Hlavní část databáze částečně vychází ze současné kabelové knihy a dělí se na tři základní části. První část je strana budovy (`cb_plug`). Představuje jednotlivé zásuvky v jednotlivých budovách. Každá zásuvka je identifikována svou pozicí na hlavním rozvodu (`hr_plug`) – to je určeno napevno, při fyzickém přivedení drátu od zásuvky do hlavního rozvodu. Dále je ke každé pozici vedeno, jaký typ telefonu je použit, evidenční číslo telefonu, průběh páru a také softwarové vazby mezi přístroji.

Druhá část je strana pobočkové ústředny (`cb_pbu`). Pozice na pobočkové ústředně je v aplikaci identifikována pomocí své pozice na hlavním rozvodu (`hr_pbu`). Dále je u každé pozice evidován kód pozice (`equ`), linka, oprávnění (např. možnost volat do veřejné telefonní sítě), pobočková ústředna ke které přísluší, typ pozice a softwarové vazby (ADN a ODN).

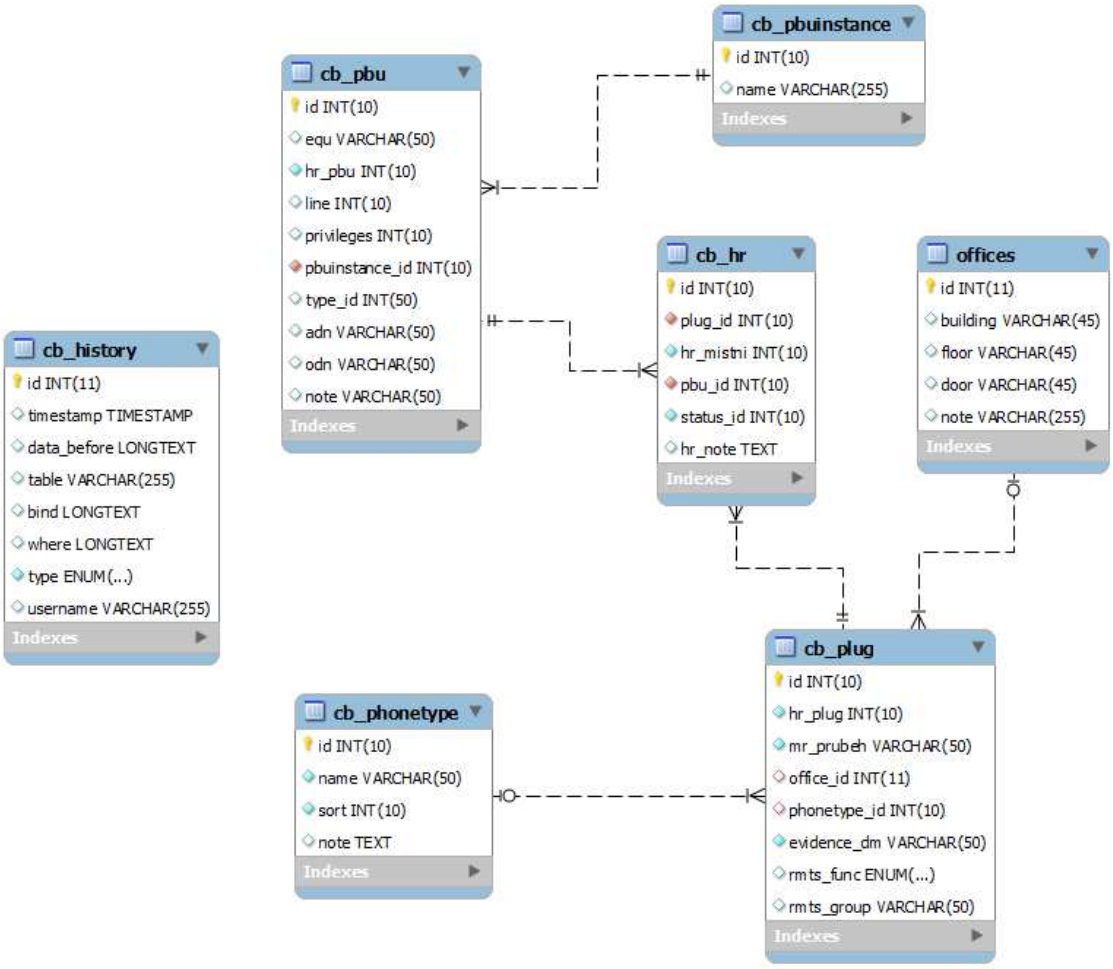


Obrázek 3 Návrh hlavní části kabelové knihy (zdroj: autor)

Třetí část, hlavní rozvod (`cb_hr`), spojuje mezi sebou zásuvky a pozice na pobočkové ústředně. Zde při analýze opět vyvstal problém – na jedné zásuvce je propojeno vždy nejvíce jedno číslo. Avšak jedno číslo lze propojit tzv. paralelkou na více zásuvek. Z toho by se mohlo zdát, že `cb_plug` a `cb_pbu` jsou spojeny N:1 vazbou.

A taková také byla původní struktura. Následně se nicméně ukázalo, že tato struktura neumožňuje pojmout veškerá potřebná data – obsahem kabelové knihy totiž nejsou jen existující propojení, ale také plánované spoje a z evidenčních důvodů také některé odstraněné spoje. V tu chvíli mohou existovat trojice (`hr_plug=1234`, `hr_pbu=4321`, `plánované`) a (`hr_plug=1234`, `hr_pbu=9876`, `odstraněné`). To však nesplňuje předpoklad, že hodnota `hr_pbu` by měla být v rámci vazeb unikátní. Vazba se tím mění na M:N.

Veškeré tyto problémy plynuly především z neexistence konkrétní specifikace. Zadáním této diplomové práce bylo prakticky replikovat současnou funkčnost kabelové knihy do formy webové aplikace. Měl jsem snahu vytvořit nějakou abstraktní specifikaci jednotlivých vazeb kabelové knihy, ale většinou komunikace uvázla a v kruhu se vrátila k tomu, že soubor kabelové knihy je sám o sobě specifikací. Předpokládám, že to v budoucnu povede k potřebě dalších úprav v aplikaci, tak jak se budou objevovat nové případy použití, které současná struktura nedokáže pokrýt.



Obrázek 4 Schéma databáze (zdroj: autor)

2.3 Návrh struktury aplikace

Prvním krokem návrhu struktury aplikace bylo stanovit nejčastější činnosti, které budou v rámci kabelové knihy vykonávány. Vzhledem k faktu, že vazby a většina údajů na straně budov a pobočkových ústředen je pevně daná, zvolil jsem jako hlavní část aplikace (`IndexController`) přehled propojení na hlavním rozvodu. Pro účely přehledu jsou však zobrazena i veškerá data na straně budov a ústředen. To umožňuje uživateli procházet jednotlivá propojení na hlavním rozvodu a současně mít přehled o propojených místech.

V jednom controlleru je zpracován přehled a úpravy zásuvek v budovách a v druhém jsou zpracovány pozice na pobočkových ústřednách. Při tvorbě jsem musel několikrát některé části návrhu přepracovat. Například původně jsem předpokládal, že položky se budou mazat pouhým odkazem. Během tvorby se však ukázalo, že samotný

odkaz je náchylný na útok typu CSRF, a proto jsem byl nucen původní návrh upravit a přidat novou akci na smazání, která je proti CSRF chráněna.

Každý z hlavních controllerů aplikace (`IndexController`, `PbuController`, `ZasuvkyController`) obsahuje základní akce pro vytvoření, smazání, úpravu a vypsaní položek. Navíc obsahují akce pro specifické potřeby každého z nich – například načítání propojených položek, ukládání poznámek, atp. Controllery číselníků mají podobnou strukturu. Navíc se generují automaticky za pomoci několika málo řídících proměnných – hlouběji se tomu věnuji v kapitole 3.2.1.

V případě controllerů vědomě porušuji jedno ze základních pravidel programování – tj. nemíchat v kódu pojmenování v různých jazycích (čeština a angličtina). Důvody, které mě k tomu vedly, podrobně vysvětluji v kapitole 3.12.1. Ve zkratce je to proto, že název controlleru ovlivňuje adresu URL, na které bude dostupný.

2.4 Uživatelské rozhraní

2.4.1 Výběr CSS/JS frameworku

Pro svou práci jsem se rozhodl použít některý z dostupných CSS/JS frameworků především proto, že sám nejsem v CSS definicích příliš zručný a framework umožňuje něco, co se blíží komponentovému vývoji software. Do užšího výběru jsem zařadil následující:

- Skeleton[20] + HTML Kickstart[21]
- HTML Boilerplate[22]
- Compass[23]
- Bootstrap[24]

Z výběru jsem hned na začátku vyřadil Compass, který sice poskytuje skutečně široké možnosti, ale je postavený na SASS, který mi vyhovuje mnohem méně než LESS, a vyžaduje ke kompilaci Ruby, které jsem nechtěl do systému instalovat.

[20] http://getskeleton.com/
[21] http://www.99lime.com/
[22] http://html5boilerplate.com/
[23] http://compass-style.org/
[24] http://twitter.github.com/bootstrap/index.html

Případně umožňuje též online kompilaci přes vložený JS soubor, což není ideální, neboť to negativně ovlivňuje rychlost výsledné aplikace.

Kombinace Skeleton + Kickstart se mi jeví jako poměrně rozumně použitelná, avšak odrazuje mne od ní fakt, že se jedná o dva oddělené projekty. Může se tak snadno stát, že některé definice budou kolidovat nebo nebudou kompatibilní.

HTML Boilerplate oproti ostatním působí velmi komplexně. Spíše než o CSS framework se jedná o komplexní nástroj na správu veřejně dostupných souborů aplikace. Nabízí nástroje pro efektivní cachování a celkovou optimalizace CSS a JS. Pro potřeby aplikace mi však přijde jeho komplexnost zbytečná. Zvolil bych ho tedy spíš pro veřejně dostupnou aplikaci, kde bych musel podporovat široké spektrum uživatelských prostředí s různými verzemi prohlížečů a vysokou zátěží. Přesto některé jeho části – především vypracovanou podporu cachování pomocí `.htaccess` nastavení – použiji.

Nakonec jsem pro realizaci kabelové knihy zvolil CSS framework Bootstrap od Twitteru. Je psaný v LESS syntaxi. Obsahuje graficky zdařilé a přehledné styly pro většinu potřebných elementů, včetně inline formulářů, chybových zpráv nebo obrázkových galerií. Dále obsahuje několik JavaScriptových pluginů postavených nad jQuery, což dále usnadňuje použití. Ve své práci bych ho rád používal v co nejširší míře.

V návaznosti na volbu Bootstrapu jako frameworku pro CSS jsem se rozhodl, vzhledem k existující integraci, využívat jako JavaScriptový framework jQuery. Pro jQuery existuje množství pluginů, které lze do aplikace snadno dodat, což je velmi užitečné. Hlavní nevýhodou při použití jQuery je neexistující podpora pro jednoduchou práci se jmennými prostory[25]. Naštěstí je na internetu k dispozici množství způsobů, jak tuto funkčnost emulovat[26].

2.4.2 Jednotný vzhled uživatelského rozhraní

Za účelem jednotného vzhledu aplikace používám `Zend_Layout`. Jedná se o zjednodušenou implementaci vzoru *Two Step View* (FOWLER, 2003, s. 365). Fowler

[25] JavaScript podporu projmenné prostory nemá, ale využívá se prefixování proměnných, podobně jako v případě PHP před verzí 5.3, nebo zanořování objektů
[26] http://stackoverflow.com/questions/527089/is-it-possible-to-create-a-namespace-in-jquery

uvádí, že by mezikrok zpracování měl obsahovat logickou prezentaci dat bez jakéhokoli formátování, která je poté transformována do HTML. V `Zend_Layout` je však HTML generováno již v prvním mezikroku. V praxi je ale skutečně tím, co ovlivňuje vzhled, až druhý krok – layout, protože v něm jsou nastaveny kaskádové styly, které nejvíce ovlivňují vzhled. V prvním kroku je vykreslen obsah určité části stránky do tzv. *response segmentu*. Počet *response segmentů* není nijak omezený. Následně je *response segment* vložen do layoutu.

Kód 8 Obsahová část layoutu

```
<div class="container-fluid" id="page">
    <?php echo $this->messages();?>
    <?php echo $this->layout()->content;?>
    <footer class="footer">
        <p class="pull-right">
            <a href="#">Back to top</a>
        </p>
    </footer>
</div>
```

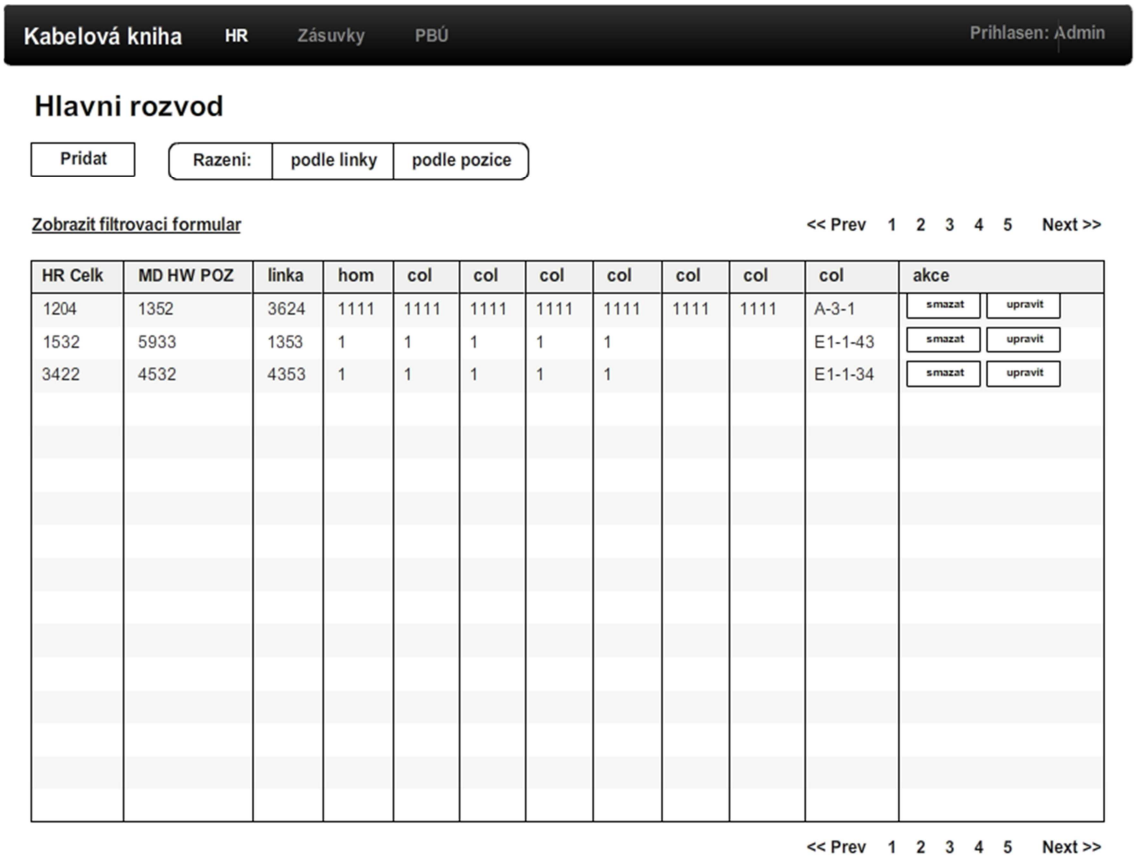
Kód 8 ukazuje hlavní, obsahovou část layoutu. Volání metody `messages()` vykreslí chybové, případně potvrzovací zprávy. Pod ním je vidět vykreslení kódu vzniklého v mezikroku – *response segmentu* `content`.

Díky použití layoutu a kaskádových stylů je docíleno jednotného vzhledu napříč aplikací. Zásadní změny je tak třeba provádět pouze na jednom místě kódu a nedochází k nekonzistencím v různých částech aplikace, co se týče vzhledu.

2.4.3 Návrh wireframů

Před tím, než jsem vytvořil funkční HTML, jsem si pomocí nástroje *MockFlow*[27] vytvořil jednoduché wireframy. Snažil jsem se v nich zachytit nejdůležitější činnosti, které bude uživatel provádět, a vytvořit k nim odpovídající reprezentaci v uživatelském rozhraní. Už předem jsem věděl, že chci používat Bootstrap (kapitola 2.4.1), a proto jsem tomu přizpůsobil i vzhled wireframů.

[27] http://www.mockflow.com/



Obrázek 5 Návrh rozhraní pro přehled vložených položek (zdroj: autor)

Wireframy jsem vytvářel iterativně. Jednotlivé iterace jsem zkoušel předkládat různým lidem a ptát se jich na jejich názory. Na některém z prvních návrhů jsem například umístil filtrovací formulář nad tabulku. Ukázalo se však, že je to nevhodné, protože zabírá příliš mnoho místa a hlavní obsah – samotná data – byl odsunut až dolů.

Obrázek 6 Návrh formuláře (zdroj: autor)

K návrhu formuláře neměly dotazované osoby žádné připomínky, a proto zůstal v prakticky nezměněné podobě – jen došlo k barevnému odlišení odesílacího tlačítka.

2.5 Bezpečnost

2.5.1 Přihlašování uživatelů

Uživatelé se k systému přihlašují přes přihlašovací formulář. Hlavním způsobem přihlašování je – jako v případě telefonního seznamu – autentizace a autorizace proti školnímu serveru LDAP. Nebylo proto třeba implementovat celou přihlašovací infrastrukturu, obnovení zapomenutého hesla, blokování účtu při více neúspěšných pokusech atp.

Specifickou úpravou přihlašování je tzv. *localhost mód*. Vývoj aplikace probíhá typicky na lokálním serveru, který běží v počítači vývojáře. Takový server typicky nemá přístup ke všem zdrojům, ke kterým má přístup server produkční. Stejný případ je i s přihlašováním ke školnímu LDAP serveru. Z mého vývojového PC není k tomuto serveru přístup. Byl jsem proto nucen dočasně vypnout logiku, která se stará o přihlášení. Takový postup však není ideální. Především proto, že spoléhá na to, že se neprojeví chyba lidského faktoru – lidově řečeno „že to vývojář nezapomene zapnout“. A nemusí se jednat přímo o zapomenutí, vývojové prostředí může sdílet více vývojářů a někteří z nich nemusí být se zmíněnou úpravou seznámeni a mohou ji omylem nasadit na produkční server.

Abych předešel možným chybám, vytvořil jsem zmíněný *localhost mód*. Pokud server naslouchá jen na lokálních IP (127.0.0.1), pak ho považuji za vývojový, a to způsobí zapnutí druhé verze přihlašování – *automatické*. Do přihlašovacího formuláře je přidán přepínač, který umožní přihlásit se automaticky místo přes LDAP – výsledkem je vždy úspěšné přihlášení nehledě na jméno a heslo. To umožňuje snadno testovat například přihlášení více uživatelů. A díky automatické kontrole, zda se jedná o vývojový server, se nestane, že by se vypnutá kontrola přihlašování omylem dostala na produkční server.

2.5.2 CSRF

V kapitole 1.10.3 pojednávající o CSRF z teoretického pohledu bylo již zmíněno, jaké jsou podmínky vzniku CSRF. Ideálním způsobem, jak tuto zranitelnost ošetřit, je předávat v každém požadavku unikátní klíč – token. Vykonáním některých požadavků se nemění žádná data, například se pouze nějaká data vypisují. Pro takové požadavky je

předávání tokenu zbytečné a jen komplikuje kód a znesnadňuje úpravy. Objektová struktura Zend Frameworku mi umožnila nasadit předávání tokenu na nejnižší úrovni a zcela transparentně pro vývojáře.

Předpokládám, že pokud bude aplikaci kabelové knihy vyvíjet ještě někdo jiný než já, bude dodržovat nastavenou konvenci. Tou konvencí je, že pokud jsou jakkoli měněna data, provádí se změna pomocí metody POST – tedy formulářem.

Pro formuláře jsou k dispozici dvě základní třídy. Všechny formuláře jsou potomky třídy `Tul_Form`. Ta nastavuje dekorátory ovlivňující vzhled formulářů a stará se o to, aby formuláře vypadaly a chovaly se konzistentně napříč aplikací. Druhou třídou je `Tul_Form_Mutating`, která je – jak název napovídá – určena pro vytváření formulářů měnících data. Do formuláře je automaticky přidán skrytý prvek, který obsahuje hodnotu tokenu. Jeho hodnota je po odeslání zkontrolována a porovnána s hodnotou uloženou na serveru. Navíc je token na serveru zneplatněn, aby již nemohl být použit.

2.5.3 XSS

Je velmi nepravděpodobné, že by se někdo pokusil zneužít XSS zranitelnosti v aplikaci kabelové knihy. Muselo by se jednat o některého z autorizovaných uživatelů, což je velmi nepravděpodobné. I přesto se snažím toto nebezpečí nepodcenit. Především jde o ošetřování vstupů a výstupů. Vstupy lze velmi snadno ošetřovat pomocí filtrů použitých ve formulářích (kapitola 3.5).

2.5.4 SQL injection

Obrana proti SQL injection je součástí Zend Frameworku. Metody modelů (implementující *table data gateway* a *row data gateway*) mají parametry ošetřeny automaticky a třídy pro ruční skládání dotazů obsahují nástroje pro parametrizaci dotazů a ošetřování vstupních proměnných. Tento postup ukazuje Kód 9. V metodě `where()` je použit parametr *id* pro omezení dotazu. Ten je ošetřen a vložen na místo otazníku v dotazu.

Kód 9 Skládání dotazu pomocí Zend_Db_Select

```
$sql = $this->getAdapter()
            ->select()
            ->from($this->_name, array())
            ->joinLeft('cb_hr', 'cb_hr.plug_id = cb_plug.id')
            ->joinLeft('cb_pbu', 'cb_pbu.id = cb_hr.pbu_id')
            ->where('cb_hr.plug_id = ?', $id);
```

Pokud je použit tento způsob, je zaručeno, že se do dotazu nemohou dostat závadná data, protože jsou vždy předem ošetřena. Tento způsob používám důsledně ve všech dotazech do databáze.

Bezpečnostní problém by však vznikl, pokud by vývojář nebyl s tímto způsobem práce seznámen a vložil by proměnnou přímo do řetězce bez ošetření. Takovému chování bohužel nelze zabránit.

2.5.5 Minimalizace rizikových faktorů

I při sebevětší snaze o zabezpečení aplikace nelze zaručit, že žádná bezpečnostní chyba neexistuje. Nezanedbatelnou součástí bezpečnosti je proto minimalizace rizik, které by případná bezpečnostní chyba představovala. Příkladem takového přístupu je **vytvoření nezávislého primárního klíče** – v aplikaci kabelové knihy se jedná o sloupec `id`. V případě existující CSRF zranitelnosti v aplikaci nemůže útočník odhadovat `id` ze znalosti dat, neboť skutečná data nejsou na hodnotě `id`, které se používá, nijak závislá.

V aplikaci kabelové knihy působí tato snaha poněkud směšně, protože je nepravděpodobné, že by cílem útočníka bylo například přepojit linku s pěkným číslem do své kanceláře. Ale pokud nastane podobná situace v případě banky, je vše jasnější.

Kód 10 Zranitelnost odhadnutím parametrů

```
https://secure.bank.cz/transfer/amount/1000/from/1521315422-
0800/to/2463120200-1300

https://secure.bank.cz/transfer/amount/1000/from/7/to/13
```

V druhém případě musí útočník nejprve nějakým způsobem získat interní čísla účtů, aby mohl převést prostředky na svůj účet. Tento typ opatření je nazýván

bezpečnost skrze utajení[28] a je uváděn jako nefunkční opatření pro zabezpečení (PFLEEGER et. al., 2011, s. 194), pokud však není jediným prostředkem bezpečnosti, ale její podpůrnou složkou, pak má smysl (AMOROSO, 2010, s. 133).

Součástí omezení rizika je také **umístění souborů aplikace mimo adresáře přístupné z internetu**. V ideálním případě není možné PHP skript na serveru nijak zobrazit, protože nejprve projde interpretem PHP. Avšak pokud dojde k chybě v nastavení serveru a PHP soubory jsou odesílány jako čistý text, tak vzniká bezpečnostní riziko – PHP soubor může obsahovat přihlašovací jména nebo hesla ke službám nebo jiné citlivé údaje. Řešením je umístění co možná největší části aplikace mimo veřejný kořenový adresář (*document_root*). Díky tomu v případě chyby bude veřejně přístupný jen soubor `index.php`, který ale neobsahuje žádná citlivá data.

[28] Security throught Obscurity

3 Praktické zpracování

Prvním krokem ke zprovoznění aplikace je stažení Zend Frameworku[29] a jeho umístění do složky, která je obsažená v konfigurační direktivě `include_path`. Použil jsem nejnovější vydání dostupné v době psaní práce – verzi 1.11.11.

Nejprve bylo třeba vytvořit základní kostru aplikace. Jak jsem již uvedl v kapitole 1.4.2, aplikace postavené nad Zend Frameworkem využívají architektonický vzor *Front Controller*. Prvním krokem tedy bylo vytvořit soubor `index.php` v kořenovém veřejném adresáři serveru. Z bezpečnostního hlediska je vhodné, aby tento soubor obsahoval pouze nejnutnější kód a aby adresář byl o jednu úroveň hlouběji ve struktuře, než je adresář s aplikací. Obsahem souboru `index.php` je tak prakticky pouze definice veřejného adresáře, vyžádání souboru s třídou `Bootstrap` a zavolání základní startovací metody.

Třída `Bootstrap` provádí nutné kroky ke spuštění *Front Controlleru*, načítání tříd a dalších potřebných procedur. Díky objektovému návrhu je možné ji využívat opakovaně. To jsem využil v tzv. *lightweight dispatcheru* (více v kapitole 3.6).

3.1 FrontController pluginy

Celý Zend Framework se snaží propagovat princip DRY. Proto existuje mnoho míst, které umožňují vytvoření znovupoužitelných komponent. Jedním z příkladů jsou *Front Controller* pluginy. Sám jich mnoho používám ve svých aplikacích, které vyvíjím. Příkladem takové třídy je například `Tf_Controller_Plugin_DbConnection`, což je třída, která zabezpečuje připojení k databázi, což je typická činnost v každé webové aplikaci. Místo abych tento často se opakující kód psal pokaždé znovu, stačí pouze následujícím voláním připojit plugin k *Front Controlleru*.

Kód 11 Připojení pluginu k Front Controlleru

```
Zend_Controller_Front::getInstance()
    ->registerPlugin(new Tf_Controller_Plugin_DbConnection());
```

[29] http://framework.zend.com/download

Dalším příkladem takové třídy je `Tul_Acl`. To je pro potřeby kabelové knihy lehce upravená verze mé třídy `Tf_Acl`, která se stará o řízení přístupových práv k aplikaci. Zajišťuje také přesměrování uživatele na autentizační formulář. Podobným příkladem je také třída `Tf_Controller_Plugin_Superluminal`, která využívá načítání více tříd do jednoho souboru k urychlení aplikace – více se o ní zmiňuji v kapitole 3.12.3.

3.2 Controllery

V Zend Frameworku jsou jednotlivé controllery potomky základní třídy `Zend_Controller_Action`, která se stará o základní funkce, které musí mít každý controller – tedy zpracování požadavku, vrácení odpovědi, přístup k view, načítání helperů a další. Zajímavostí je, že v ZF verzi 2 již pro implementaci controlleru stačí pouze implementovat rozhraní `Dispatchable` obsahující předpis pro pouhé dvě metody. To umožňuje čistější návrh vlastního řešení controlleru (například za využití kompozice), protože přílišné zanoření tříd může představovat problém z hlediska návrhu a následně také výkonnostní problém. Verze 1 toto bohužel neumožňuje.

Využil jsem tedy dědičnost – třídu `Tul_Controller_Base` – ta implementuje některé užitečné a často používané funkce. Například metoda `_crudId()` automaticky řeší načítání ID položky z URL a její přiřazení do proměnné – tedy kód, který se vyskytuje prakticky v každém controlleru. Existuje možnost tuto funkcionalitu přesunout do znovupoužitelných action helperů – některé funkce již samotné action helpery používají (např. `flashMessenger` pro vytváření zpráv pro uživatele). V tomto případě jsem se však rozhodl tak neučinit, a to především proto, že by to mohlo negativně ovlivnit výkon. Načítání action helperů má totiž určité výkonnostní rezervy, jelikož načítání probíhá postupným procházením několika složek, dokud není nalezena odpovídající třída.

3.2.1 CRUD controllery

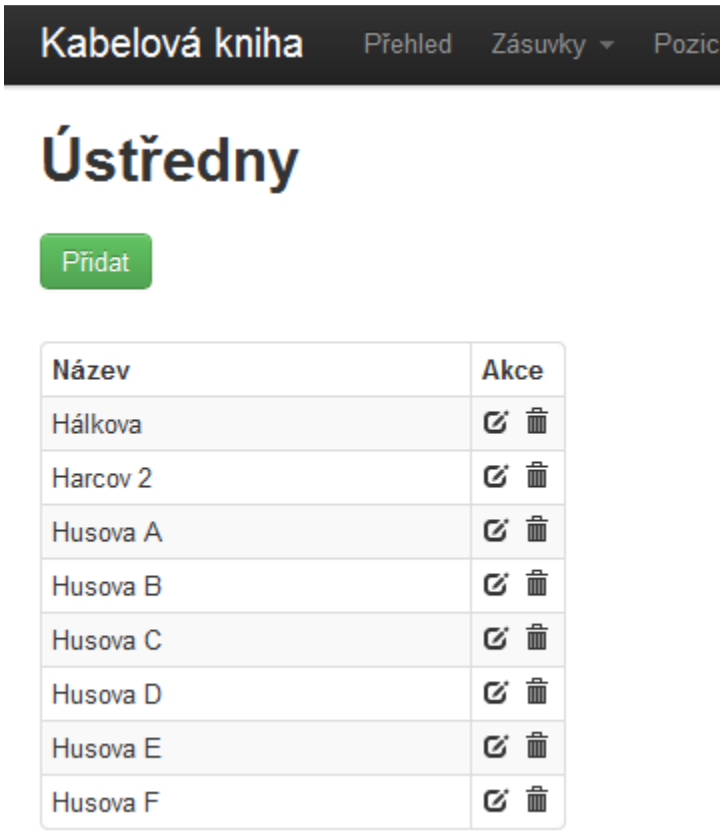
Pro číselníky, které obsahují často se opakující jednoduché úkony (vložit, smazat, upravit, vypsat seznam), jsem vytvořil speciální třídu controlleru, která tyto problémy řeší. Konkrétní controllery pak od této třídy dědí a obsahují pouze minimální nastavení.

Tento přístup jsem použil při tvorbě telefonního seznamu, ale musel jsem ho upravit, aby odpovídal specifikům implementace kabelové knihy.

Kód 12 Konkrétní CRUD controller pro úpravu jednotlivých ústředen

```
<?php
class UstrednyController extends Tul_Controller_Crud
{
    protected $_modelClass = 'Model_PbuInstance';
    protected $_formClass = 'Form_Codebook_PbuInstance';
    protected $_viewKey = 'ustredny';
    protected $_title = 'Ústředny';
}
```

Je patrné, že struktura jednotlivých konkrétních implementací je skutečně minimální. Jediná další věc, kterou je třeba pro jednotlivé controllery číselníků dopsat, je výpis jednotlivých položek, protože ten se mezi jednotlivými controllery zásadně liší. Výsledkem je plně funkční controller pro úpravu, výpis, vkládání a mazání (Obrázek 7).



Obrázek 7 Výsledek implementace (zdroj: autor)

3.3 Modely

Pro přístup k databázi jsem využil kombinované řešení za použití `Zend_Db_Table` a `Zend_Db_Select`. Abych omezil opakující se kód, vytvořil jsem vlastní třídu `Tul_Db_Table_Abstract`, která je potomkem třídy `Zend_Db_Table_Abstract`. Ta poskytuje často používané metody jako například `fetchPairs()`, která vrací pole, jehož klíči jsou *id* položek v databázi a hodnotami jsou lidsky čitelné výrazy jim odpovídající.

Zajímavostí je statická metoda `getReadableRowSql()`. Tato metoda vrací SQL výraz, který odpovídá lidsky čitelné reprezentaci položky v databázi – často je to název, ale může to být teoreticky jakýkoli platný výraz. Tato metoda využívá novou vlastnost PHP 5.3 – *late static binding*. Do nedávna nebylo možné používat metody třídy, které používají nějaké proměnné nebo konstanty třídy, pokud tyto byly ve zděděné třídě přetíženy. Odkazy na takové proměnné byly vyhodnoceny v místě definice třídy a obsahovaly tudíž hodnotu z rodičovské třídy.

Kód 13 Ilustrace late static binding

```
class Tul_Db_Table_Abstract extends Zend_Db_Table_Abstract
{
    protected static $_readableRowSql = 'name';
    public static function getReadableRowSql($includingTable = false)
    {
        if ($includingTable) {
            return static::TABLE . '.' . static::$_readableRowSql;
        }
        return static::$_readableRowSql;
    }
}
```

V PHP 5.3 se díky *late static bindingu* vyhodnocuje kontext až při samotném volání metody. Aby byla zajištěna zpětná kompatibilita, bylo zavedeno nové klíčové slovo `static`, které na rozdíl od `self` používaného v předchozích verzích tuto vlastnost jazyka používá. Kód 13 ukazuje, jak jsem toho využil pro vytvoření statické metody, která používá proměnnou i konstantu třídy. Následující kód pak zobrazuje zděděnou třídu.

Kód 14 Ukázka zděděné třídy

```
class Model_Hr extends Tul_Db_Table_Abstract
{
    const TABLE = 'cb_hr';
}
```

Jak je patrné, tak konstanta `TABLE` bude použita s hodnotou z třídy `Model_Hr` a statická proměnná `$_readableRowSql`, která v zděděné třídě není, bude použita s hodnotou z rodičovské třídy.

Late static bindingu jsem využil také v metodě `joinThis()` sloužící k manipulaci se `Zend_Db_Select` – objektové implementace databázových dotazů.

3.4 View

Vzhledem k tomu, že Zend Framework nepoužívá žádný zvláštní šablonovací systém, jsou views (šablony) čisté PHP soubory s příponou `phtml`. Opět zde existuje možnost vytvoření znovupoužitelných tříd – tzv. view helperů. Této možnosti jsem využil a vytvořil třídu `Tf_View_Helper_UrlHelper`, což je jednoduchá abstrakce nad URL, která mi umožnila implementaci filtrovacích funkce, které generují pro každé

určité nastavení filtru určité URL. To umožňuje velmi snadné sdílení – například lze takové URL poslat kolegovi nebo si ho přidat do záložek.

Pro vizuální reprezentaci jsem použil, jak jsem již zmínil v kapitole 2.4.1, CSS framework Bootstrap. I když se jedná o řešení, které je nasazeno na mnoha webech a je velmi oblíbené, bylo nutné některé části upravit. Díky využití CSS kompilátoru LESS jsem mohl napsat vlastní LESS soubor – `custom.less`, který byl zaintegrován do stylopisu Bootstrapu a minifikován.

Kód 15 Ukázka vytvoření CSS s využitím LESS syntaxe

```
.sorter {
    a {
        padding: 0 1px;
    }
    a.sorter-add {
        margin-right: 8px;
    }
    a:hover {
        background-color: desaturate(lighten(@linkColor, 50%), 50%);
        text-decoration: none;
    }
}
```

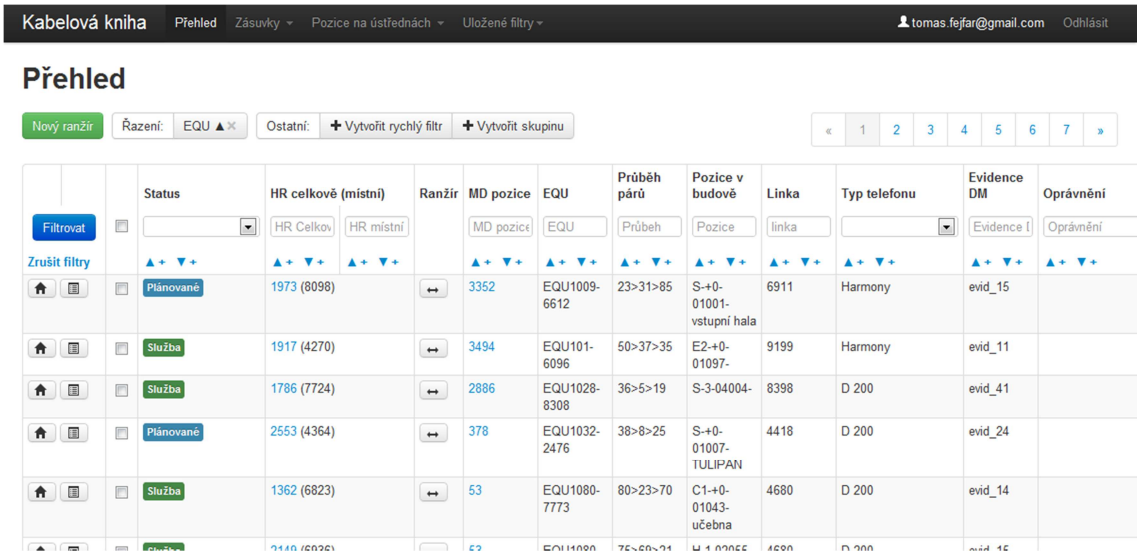
Na tomto kusu kódu je vidět využití skvělých vlastností LESS kompilátoru. Kód představuje definici šipek pro řazení výsledků ve výpisu. Jak je vidět, pomocí zanoření se podařilo zápis značně zjednodušit a zpřehlednit. Navíc je u nastavení barvy pozadí při najetí myši využito výpočetní logiky, kterou LESS disponuje – barva pozadí je dopočítána z barvy odkazů pomocí zesvětlení a desaturace. Díky tomu bude mít pozadí odpovídající barvu i v případě, že se později rozhodneme změnit barvu odkazů. Především z pohledu konzistence je to veskrze pozitivní.

Problém při nasazení Bootstrapu představovalo použití formulářů. Ty vyžadují pro správné zobrazení velmi specifický HTML kód, který třída obsažená v Zend Frameworku není schopna vytvořit. Dále se o tom zmíním v kapitole 3.5.1.

3.4.1 Uživatelské rozhraní

Při tvorbě reprezentace uživatelského rozhraní v HTML jsem vycházel z wireframů (o tvorbě wireframů pojednává kapitola 2.4.3) a snažil jsem se jich co možná nejvíce držet. Během implementace se však ukázalo, že mnoho věcí zůstalo

během návrhu wireframů nedořešených a mnohé problémy se během tvorby wireframů nepodařilo odhalit.



Obrázek 8 Jedna z posledních iterací uživatelského rozhraní (zdroj: autor)

Je patrné, že výsledné uživatelské rozhraní se v mnohém liší od wireframu (Obrázek 5). Nyní se zaměřím na jednotlivé změny rozhraní oproti wireframům.

Filtrování je jednou z nejčastějších činností, které uživatel v kabelové knize vykonává. Standardní formulář nad tabulkou, který je použit v případě aplikace telefonního seznamu (Obrázek 9) není vhodný, protože filtrovaných parametrů je velké množství a rozsáhlý formulář tak odsouvá samotná data mimo obrazovku. Nutnost posouvat stránku při každém zobrazení je zásadní nevýhoda a znepříjemňuje používání aplikace.

Obrázek 9 Filtrování v telefonním seznamu (zdroj: autor)

Ve wireframech jsem tuto situaci vyřešil odkazem, který zobrazoval a skrýval formulář. Pokud je však filtrování využíváno často, skrytý formulář není vhodný. Nakonec jsem tedy zvolil variantu, kdy jsou **filtrovací políčka integrována do tabulky**, což šetří místo a navíc je to pochopitelnější.

Další velmi zásadní změnou proti wireframům je **umístění tlačítek akcí (smazat, upravit) na levou stranu**. Z wireframu nebylo patrné, jak se bude tabulka chovat, pokud bude obsahovat reálná data. Z obrázku to sice není patrné, ale stránka je širší než 1280 pixelů. I na běžném notebooku nebo menším LCD panelu je tedy nutno posouvat stránku do stran. Je proto nevhodné mít tak důležitý ovládací prvek jako tlačítka akcí za okrajem obrazovky. Tabulku jsem proto přeskládal tak, aby nejdůležitější informace byly co nejvíce vlevo a byly vidět hned po načtení stránky.

Třetí změnou je **přesunutí tlačítka pro úpravu ranžíru** mezi „ranžírované“ položky: HR celkové a MD pozice. I když takové umístění není na první pohled logické, myslím, že by mohlo fungovat, protože lépe ilustruje existenci vazby mezi zmíněnými dvěma položkami. Nevýhodou by mohlo být, že se tlačítko v seznamu „ztratí“, protože nebude dostatečně vizuálně výrazné. V tomto případě si výsledkem nejsem jistý a bude asi třeba počkat na nasazení aplikace do produkčního prostředí.

3.5 Implementace formulářů

V Zend Frameworku jsou formuláře implementovány jako objektová abstrakce nad výsledným HTML kódem formuláře. A to včetně validace, filtrování a získávání hodnot jednotlivých elementů.

Tento postup s sebou nese mnoho výhod. Například zjednodušuje načítání hodnot do modelu. Pokud správně nastavíme filtrování a validátory, tak jsou výsledkem volání metody `getValues()` čistá data, která je přímo možné vložit do databáze. Objektová podstata formulářů pak také umožňuje dědičnost, čehož jsem využil při ochraně formulářů proti CSRF (více v kapitole 2.5.2).

Kód 16 Zdrojový kód formuláře pro přihlášení

```
class Form_Login extends Tul_Form
{
    public function init()
    {
        $this->addElement(new Zend_Form_Element_Text(array(
            'name' => 'username',
            'label' => 'Přihlašovací jméno',
            'required' => true,
        )));
        $this->addElement(new Zend_Form_Element_Password(array(
            'name' => 'password',
            'label' => 'Heslo',
```

```
            'required' => true,
        )));
        if (Config::isLocalhost()) {
            $this->addElement(new Zend_Form_Element_Radio(array(
                'name' => 'type',
                'label' => 'Přihlášení',
                'required' => false,
                'multiOptions' => array(
                    'auto' => 'Automatické (pouze localhost)',
                    'ldap' => 'LDAP',
                ),
                'value' => 'auto',
            )));
        }
        $this->addElement(new Zend_Form_Element_Submit(array(
            'name' => 'submit',
            'label' => 'Přihlásit',
            'class' => 'btn btn-primary btn-large',
            'ignore' => true,
        )));
    }
}
```

Jak je patrné, kód formuláře neobsahuje žádné informace o vykreslování (kromě CSS třídy uvedené pod klíčem `class`). Tyto informace jsou zděděny od třídy `Tul_Form`.

Pro některé položky formulářů bylo třeba vytvořit vlastní validační třídy, protože jednoduchá validační pravidla obsažená ve výchozím nastavení nepostačovala. Ukázku takového validačního pravidla ukazuje následující kód.

Kód 17 Validátor průběhu

```
class Tul_Validate_Prubeh extends Zend_Validate_Abstract
{
    const INVALID = 'invalid';

    protected $_messageTemplates = array(
        self::INVALID => "Toto není platný průběh"
    );

    public function isValid($value)
    {
        $numMatches = preg_match('/^[>]{0,1}[0-9A-Za-z]+([>]{0,1}[0-9A-Za-
z])*$/', $value);
        if ($numMatches < 1) {
            $this->_error(self::INVALID);
            return false;
        }
        return true;
    }
}
```

Tento validátor slouží pro kontrolu, že je v poli *průběh* vložena platná hodnota. Je definována zpráva pro případ, že hodnota nebude platná, a je implementována metoda `isValid()`, která provádí kontrolu. Z hodnoty regulárního výrazu vidíme, že se jedná

o řetězec, který může začínat znakem „>“, a za ním následuje několik písmen nebo číslic. Tato skupina se může opakovat.

3.5.1 Vývoj decorátorů pro Twitter Bootstrap

Díky velké flexibilitě vykreslování formulářů v Zend Frameworku jsem mohl vytvořit nadstavbu nad standardními třídami formulářů. Ta umožňuje bez větší práce vytvořit vlastní formulář, který se při výpisu vypíše se syntaxí odpovídající potřebám Bootstrapu. Protože se jedná o znovupoužitelné řešení, rozhodl jsem se ho zveřejnit pod svobodnou licencí, aby ho mohli použít i další vývojáři. K tomu jsem využil platformu pro sociální programování GitHub[30]. To se ukázalo jako správný krok, protože několik vývojářů začalo projekt sledovat, oznamovat chyby[31] a někteří se dokonce přímo zapojili do vývoje[32].

3.6 Lightweight dispatch

I když Zend Framework exceluje v mnoha ohledech, rychlost mezi ně nepatří. Flexibilní řešení vyžaduje množství kontrol a míst, kde může být napojeno určité rozšíření. S rostoucí flexibilitou však stejně narůstá komplexita. Velká část z času požadavku je využita na tzv. *dispatch* proces. To jsou veškeré činnosti, které je třeba provést před zpracováním samotného výkonného kódu – například routování, překlad routy na třídu, kontrola existence controlleru a odpovídající třídy, vytváření tříd odpovědi, pohledu a mnoho dalších. Tento kód sice poskytuje vývojáři množství usnadnění, ale pokud je nutné zpracovat požadavek co nejrychleji, je překážkou.

Přikročil jsem proto k vytvoření vlastního *dispatch* procesu. Ten je zjednodušen na minimální možnou míru. Vyvolá se požadavkem na `ajax.php`. Ten podobně jako `index.php` obsahuje pouze to nejnutnější. Další část je znovupoužita ze třídy `Bootstrap` s jediným malým rozdílem – místo metody `Bootstrap::start()` je volána metoda `Boostrap::startLightweight()`. Ta prakticky jen zaregistruje třídy pro autoloading, nastaví parametry pro databázi a přeloží URL ve formátu `/controller/akce` na odpovídající metodu třídy controlleru a té okamžitě předá řízení vyvoláním metody `dispatch()` controlleru.

[30] https://github.com/tomasfejfar/Zend-Form-Bootstrap
[31] https://github.com/tomasfejfar/Zend-Form-Bootstrap/issues/5
[32] https://github.com/tomasfejfar/Zend-Form-Bootstrap/pull/2

Controller je opět minimální. Buď vrátí odpověď přímo na výstup, nebo pokud se jedná o pole, odešle ho jako JSON s odpovídajícími hlavičkami. Celý tento proces je oproti výchozí situaci velmi rychlý.

Rychlosti *lightweight dispatche* využívám především pro asynchronní volání iniciovaná JavaScriptem (AJAX popisuje kapitola 1.7.1). Díky tomu působí aplikace rychlejším dojmem a přitom jsou pro běžné ne-JavaScriptové požadavky zachovány možnosti, které poskytuje výchozí řešení.

3.7 Využití možností moderních prohlížečů

3.7.1 Využití localstorage

Lokální úložiště jsem v aplikaci využil pro rychlé ukládání filtrů. V libovolném výpisu si uživatel může tento výpis včetně všech aktuálně nastavených filtrů uložit pro pozdější použití. Protože nastavení filtrů je reflektováno v URL, tak je ukládání poměrně snadné – stačí pouze uložit momentální adresu.

Pro práci s lokálním úložištěm jsem použil knihovnu Lawnchair[33], kterou jsem zvolil především pro její jednoduchost – nechlubí se žádnými zvláštními schopnostmi. Umí vybírat záznamy, velmi jednoduše vyhledávat záznamy podle klíčů a vracet všechny záznamy. Nic víc ani není potřeba. Její síla však spočívá ve využití adaptérů – podobně jako `Zend_Db` umí využít více úložišť – není tak přímo závislá na použití *localstorage*, ale umožňuje používat také např. pomocí *userdata* ve starších verzích Internet Exploreru nebo Webkit SQLite dostupné ve starších verzích prohlížečích s jádrem Webkit (Chrome, Safari).

Skript na stránce navíc naslouchá události *storage* okna prohlížeče. Díky tomu se uložené filtry aktualizují napříč okny bez nutnosti obnovování okna.

3.7.2 Využití content editable

Jak jsem uvedl v kapitole 1.5.1, jednou z nových vlastností HTML5 je možnost editovat obsah přímo v prohlížeči a přímo manipulovat jednotlivými prvky stránky. Této schopnosti jsem využil proto, aby bylo možné velmi snadno a rychle upravovat poznámky pro jednotlivá propojení.

[33] http://westcoastlogic.com/lawnchair/

K editování stačí kliknout do pole s poznámkou a začít psát. Jakmile klikne uživatel mimo pole, hodnota je automaticky na pozadí přenesena asynchronním požadavkem na server, kde je uložena. Pokud vše proběhne dobře, uživatel nepozoruje nic. Pokud dojde k chybě, zobrazí se okno s výstrahou, aby byl uživatel informován o tom, že se data nepodařilo uložit a tím pádem dojde k jejich ztrátě, pokud opustí současnou stránku. Rychlost, s jakou je možné úpravy provádět, by měla přispět ke snadnému používání aplikace.

Podobné chování jsem zvažoval i pro samotnou editaci údajů v přehledu položek. Nakonec se mi to ale jevilo jako nevhodné, neboť uživatel by mohl snadno nějaké hodnoty změnit omylem. Také by nebylo snadné vytvořit uživatelské rozhraní, které by uživatele informovalo o tom, že určitá hodnota je například neplatná, nebo mu nabízet možné varianty. Proto je úprava jednotlivých položek umístěna na zvláštní stránce.

3.7.3 Zrychlení načítání JavaScriptu

Při otevření stránky se načte HTML hlavička, ve které jsou odkazovány nejrůznější soubory. Může se jednat třeba o CSS stylopisy, ikony, RSS feedy a v neposlední řadě také JavaScript. Ten má mezi načítanými soubory výsadní postavení. Veškeré stahované soubory jsou zpracovávány paralelně, kromě JavaScriptu. Načítání takového souboru zastaví vykreslování stránky a stahování dalších zdrojů. Je to proto, že načtený soubor může razantně ovlivnit výslednou podobu stránky. Může například odebrat některé stylopisy nebo zcela odstranit části stránky. Toto chování je pochopitelné pro generický JavaScript. Ale při použití knihovny JQuery, která je použita v aplikaci kabelové knihy, se stejně žádný kód neprovádí, dokud není vytvořena v paměti celá HTML stránka (událost `DOMContentLoaded` objektu `document`). Proto je toto chování nežádoucí. Možné řešení je velmi malým kusem kódu vytvořit dynamicky HTML element `script`, případně načtením obsahu souboru asynchronním požadavkem a jeho zpracování funkcí `eval`, což je ovšem poměrně složité a navíc to může představovat různé bezpečnostní problémy.

Na řešení tohoto problému však existuje knihovna RequireJS[34], která tento problém řeší asynchronním načítáním a zpracováváním skriptů a je dostatečně vyzkoušená, protože ji používá mnoho existujících projektů, takže je možnost

[34] http://requirejs.org/

bezpečnostních chyb přeci jen menší než u nového kódu napsaného od nuly. Navíc u RequireJS probíhá další vývoj (poslední změna byla v době psaní práce před patnácti dny).

Kód 18 Použití RequireJS pro načtení skriptů

```
<script type="text/javascript">
require([
    "index/load-connections",
    "jquery",
    "bootstrap-dropdown",
    "common/filter-persistence"
], function(){});
</script>
```

Kód 18 ukazuje, jak vypadá volání knihovny RequireJS v aplikaci kabelové knihy. Každé volání má k sobě přiřazenou funkci, která je zavolána po načtení a zpracování souboru (tzv. callback). Je tak možné psát kód s mnoha závislostmi a získávat je za běhu až když jsou potřeba.

Ve většině případů však stačí, že načítání JavaScriptu nebrzdí vykreslování stránky. Proto jsem vytvořil view helper `Tf_View_Helper_RequireJs`, který mi umožňuje třídy, o nichž vím, že je určitě budu potřebovat, nastavovat přímo v PHP kódu. Čekat s načítáním souborů až do chvíle, kdy jsou přímo potřeba, nemusí být ideální. Kdybychom čekali s načtením skriptu obsluhujícího kliknutí do poslední chvíle, způsobilo by to nepříjemnou prodlevu mezi kliknutím a samotnou obsluhou. V určitých případech to naopak smysl má. Konkrétně pokud skript přímo neobsluhuje událost, ale vykonává nějakou složitější činnost, a také pokud je velmi velký a jeho stahování by zbytečně zdržovalo načítání každé stránky. Příkladem takového skriptu by bylo například ořezávání obrázků – běžné požadavky takový skript nepotřebují, a pokud je tento skript třeba, není nutná okamžitá odezva.

3.8 Filtrování

Nejsložitější komponentou kabelové knihy je modulární mikroframework pro filtrování a řazení výpisů dat. Využívá objektové principy a byl vyvíjen s důrazem na snadnou rozšiřitelnost v případě potřeby.

Kód 19 Třída filtru na hlavní stránce

```
<?php
class Crud_Filter_Index extends Crud_Filter
{
    public function init()
    {
        $this->addColumn('status_id', 'status_id', true, 'exact');
        $this->addColumn('hr_plug', 'hr_plug', false, 'int');
        $this->addColumn('hr_mistni', 'hr_mistni', false, 'int');
        $this->addColumn('hr_pbu', 'hr_pbu', false, 'int');
        $this->addColumn('mr_prubeh', 'mr_prubeh', false, 'text');
        $this->addColumn('office', new
Zend_Db_Expr(Model_Office::getReadableRowSql()), false, 'text');
        $this->addColumn('line', 'line', false, 'int');
        $this->addColumn('phonetype_id', 'phonetype_id', true, 'exact');
        $this->addColumn('equ', 'equ', false, 'text');
        $this->addColumn('evidence_dm', 'evidence_dm', false, 'text');
        $this->addColumn('pbuinstance_name', new
Zend_Db_Expr(Model_PbuInstance::getReadableRowSql(true)), false, 'text');
        $this->addColumn('rmts_func', 'rmts_func', false, 'exact');
        $this->addColumn('rmts_group', 'rmts_group', false, 'text');
    }
}
```

Základem je třída `Crud_Filter`, která představuje objektovou abstrakci nad celým konceptem řazení a filtrování. Třída zpracovává vstupní parametry předané v URL a upravuje podle nich databázový dotaz. Aby nebylo možné parametry v URL podvrhovat, má každý filtr přiřazené povolené parametry.

Jak ukazuje Kód 19, do třídy jsou přidávány jednotlivé sloupce – prvním parametrem je název filtrovacího parametru, druhým je SQL výraz, který bude použit pro omezení dotazu, třetí určuje, zda musí být použita přesná shoda a čtvrtý označuje typ sloupce. Typ sloupce určuje, jakým způsobem bude filtrování probíhat. I když je zde uveden jako řetězec, tak uvnitř metody je převeden na název třídy odpovídající danému typu sloupce.

Sloupec je definován pouze svým rozhraním `Crud_Filter_Column_Interface`. Jakákoli třída implementující toto rozhraní může být použita jako sloupec. Je připravena také abstraktní třída `Crud_Filter_Column_Abstract`, která obsahuje metody vhodné pro typické použití. Není však nutno ji používat a je možné využít vlastní implementaci. Třídy, které od ní dědí, mohou využít množství předpřipravených pravidel. Pravidla jsou implementována jako třídy, které implementují `Crud_Filter_ValueProcessor_Interface`. Jednotlivá pravidla ve frontě jsou vyhodnocována, a pokud hodnota odpovídá očekávanému formátu, je na ní pravidlo použito.

Kód 20 Třída pravidla pro celá čísla

```
<?php
class Crud_Filter_Column_Int extends Crud_Filter_Column_Abstract
{
    public function process(Zend_Db_Select $select, $value)
    {
        return $this->_process($select, $value);
    }

    protected function _loadProcessors()
    {
        $this->_processors = array();
        $this->_processors[] = new Crud_Filter_ValueProcessor_IsEqual();
        $this->_processors[] = new Crud_Filter_ValueProcessor_Between();
        $this->_processors[] = new
Crud_Filter_ValueProcessor_ComparedTo();
        $this->_processors[] = new Crud_Filter_ValueProcessor_Number();
    }
}
```

Kód 20 ukazuje třídu sloupce pro celočíselné hodnoty. V metodě `_loadProcessors()` jsou načítány jednotlivé třídy, které zpracovávají hodnoty. Je vidět, že sloupce typu `Int` podporuje formátování parametrů `IsEqual` (`==hodnota`), `Between` (`10--25`), `ComparedTo` (`>5, <10`) a `Number` (libovolné číslo). Kód 21 popisuje průběh výběru. Jak je fronta postupně zpracovávána, je na jednotlivých třídách volána metoda `process()`. Pokud hodnota odpovídá regulárnímu výrazu uvedenému v konstantě třídy, je z hodnoty vytvořen `ValueProcessor_Result` (stačí, aby třída implementovala `ValueProcessor_Result_Interface`), který je vrácen třídě sloupce ke zpracování. V případě, že hodnota neodpovídá, metoda vrací `false` a zpracování přebírá další `ValueProcessor`.

Kód 21 Třída zpracovávající hodnoty filtru

```
<?php
class Crud_Filter_ValueProcessor_Between implements
Crud_Filter_ValueProcessor_Interface
{
    const REGEX = '/^([!]{0,1})([0-9]+)--([0-9]+)$/';
    public function isValidFormat($value) {
        return (preg_match(self::REGEX, $value) > 0);
    }

    public function process($column, $value)
    {
        if (!$this->isValidFormat($value)) {
            return false;
        }
        $values = array();
        preg_match(self::REGEX, $value, $values);
        if ($values[1] == '!') {
            return new Crud_Filter_ValueProcessor_Result_Between($column
                ' NOT BETWEEN %s AND %s', array($values[2], $values[3]));
        }
        return new Crud_Filter_ValueProcessor_Result_Between($column .
            ' BETWEEN %s AND %s', array($values[2], $values[3]));
    }
}
```

Na vráceném výsledku je zavolána metoda `applyTo()`, která konečně provede samotnou úpravu dotazu podle svých parametrů. Díky využití objektového přístupu je možné vytvářet i velmi složité výsledky, které do dotazu přidají například spojení s jinou tabulkou nebo ho jinak zásadním způsobem změní.

Výsledkem je, že přidání nových způsobů zpracování nebo nových typů sloupců je otázkou přidání několika tříd s přesně stanoveným rozhraním. To by mělo také přispět ke snadnějším úpravám aplikace v budoucnu.

3.9 Napojení na API telefonního seznamu

Protože kabelová kniha v sobě neudržuje informace o osobách a je to většinu času zbytná informace, využil jsem API telefonního seznamu pro získání osoby, která danou linku používá. Po kliknutí na ikonku postavy v přehledu HR je vytvořen asynchronní požadavek, který provede komunikaci s API telefonního seznamu a vrátí požadovaná data o osobách, které linku používají. Za pozornost stojí, že akce zpracovávající volání sama inteligentně rozpozná podle hlaviček požadavku, zda se jedná o volání AJAXem nebo o běžný požadavek, a podle toho vrátí buď běžnou odpověď, nebo pouze relevantní data.

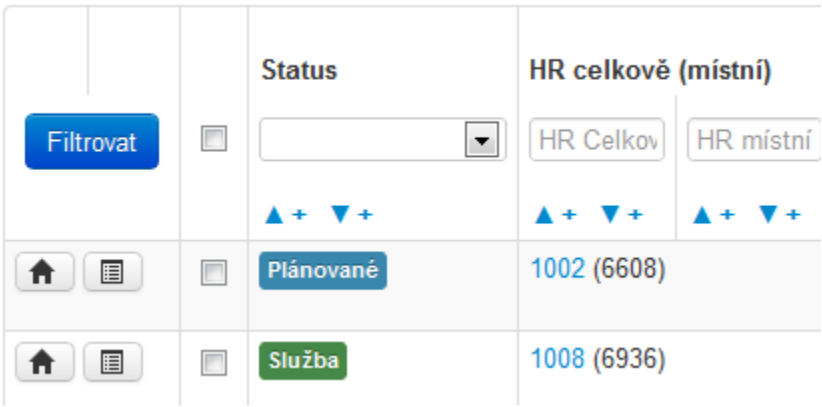
3.10 Uživatelské testování

Pokud člověk vyvíjí aplikaci samostatně bez spolupráce s jinými lidmi, lehce se může stát, že zapomene na to, aby byla aplikace dobře použitelná i pro nového uživatele. Vývojář, znalý struktury aplikace, kterou sám programoval, ví, co který ovládací prvek dělá a používá je automaticky. Nový uživatel ale musí nejprve funkce ovládacích prvků odhadnout (například podle popisku, barvy nebo tvaru), vyzkoušet, zapamatovat si je a teprve poté se je naučí používat automaticky. Skutečné uživatelské testování s více uživateli, předpřipravenými scénáři a rozsáhlou analýzou je mimo rozsah této práce. Rozhodl jsem se i přesto provést alespoň jednoduché testování s několika málo uživateli. Provést druhé referenční testování s jinými uživateli se mi již nepodařilo zorganizovat.

Připravil jsem několik jednoduchých scénářů, které představovaly typické činnosti při správě kabelové knihy – vyfiltrovat nějaké záznamy, vytvořit propojení atd. Ty jsem poté předložil testovacím uživatelům k vypracování. Během vypracovávání jsem uživatele sledoval a vyhodnocoval, co jim dělá potíže. Důležitým aspektem uživatelského testování je nikdy uživatelům neradit. Testujeme aplikaci, ne jejich schopnosti. Pokud nevidí tlačítko, které přímo před nimi, není to jejich chyba, nýbrž chyba návrháře, protože tlačítko není dost výrazné nebo není na místě, kde by ho uživatel očekával.

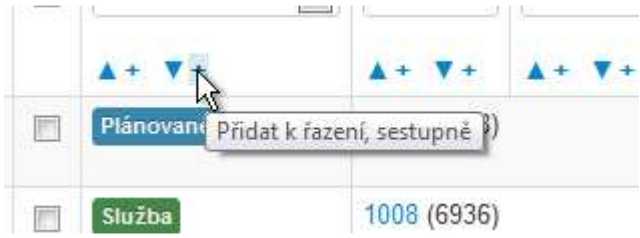
3.10.1 Výsledky testování

Testování odhalilo některé zdánlivé detaily, které výrazně ovlivňovaly práci s aplikací. Uživatelé reagovali velmi pozitivně na filtrovací políčka přímo v tabulce, které si ihned asociovali s filtrováním, a bez problémů je používali. Trochu nezvykle na ně působilo umístění tlačítka pro odeslání vlevo a hledali ho namísto toho vpravo. Umístění vpravo je nevhodné vzhledem k důvodům popsaným v kapitole 3.4.1, a proto jsem se pokusil tento problém vyřešit přebarvením tlačítka ze šedé na modrou barvu (Obrázek 10). V ideálním případě by bylo vhodné provést test znovu s jinými uživateli a ověřit, že přebarvení pomohlo, ale to bohužel nebylo možné.



Obrázek 10 Přebarvené filtrovací tlačítko (zdroj: autor)

Dalším problémem, který se objevil, byly nepochopitelné značky pro přidání do řazení (malé plus vedle šipky pro řazení). V původním stavu tlačítka neměla titulek a byla vykreslena jako horní index u šipky. Bylo ihned jasné, že plus v horním indexu je příliš malé na to, aby se do něj uživatel trefoval myší, a proto jsem ho přesunul vedle šipky. Navíc jsem přidal světlemodré pozadí, které zprostředkovává uživateli zpětnou vazbu, že už je v aktivní oblasti tlačítka. Zvažoval jsem ještě zvětšit celkovou aktivní plochu tlačítka, ale to kvůli stísněným prostorám užších sloupců nebylo možné. Pro ještě lepší použitelnost jsem přidat k tlačítkům titulky s textovou informací o funkci tlačítka (Obrázek 11).



Obrázek 11 Upravená tlačítka řazení (zdroj: autor)

Poslední výraznou změnou bylo prolinkování hlavních hodnot na filtry v odpovídajících přehledech. Nyní je tedy možné se z přehledu propojení na hlavním rozvodu přejít ze sloupce *HR celkově* na přehled pozicí na budovách s vyfiltrovaným odpovídajícím záznamem. To samé platí i pro seznam propojených míst v seznamech pozic na budovách a ústřednách.

Celkově myslím, že vyplatilo uživatelské testování provést, protože odhalilo některé zjevné nedostatky, které ztěžovaly práci s aplikací.

3.11 Automatizace některých procesů v kabelové knize

3.11.1 Historie úprav

U původní verze kabelové knihy neexistuje žádná historie úprav. V současnosti to není příliš vážný problém, neboť s kabelovou knihou pracuje jediný člověk. I přesto sledování změn může přinést pozitivní výsledky – lze například obnovit omylem smazaná data.

Vedení historie změn lze řešit na několika úrovních – na úrovni databáze, na úrovni aplikace nebo na úrovni uživatele. Na úrovni uživatele lze například zobrazovat poslední hodnoty (případně několik posledních hodnot) ve formuláři a ukládat je v localstorage – pro případ, že uživatel omylem uloží změny a bude se chtít vrátit k minulým datům. Na úrovni aplikace se jedná o logování zápisů a změněných dat na rozhraní aplikace a databáze. Na úrovni databáze se pak jedná o inkrementální zálohy, případně specializované nástroje, které úzce spolupracují s databází (např. Oracle GoldenGate[35]).

Z pohledu aplikace kabelové knihy jsou zásadní jen změny provedené zevnitř aplikace – tedy ne servisní zásahy přímo v databázi. Proto jsem se rozhodl, že verzování budu řešit pouze na úrovni aplikace a ne na úrovni databáze. Pokud by to bylo žádoucí, lze časem nasadit nějaké serverové řešení verzování databáze.

K abstrakci databáze v aplikaci je používána třída `Zend_Db` a částečně je použit také návrhový vzor *Table Row Data Gateway* implementovaný pomocí `Zend_Db_Table` a `Zend_Db_Table_Row`, interně využívající opět `Zend_Db`. To umožňuje navázat určité události na velmi nízké úrovni abstrakce.

Kód 22 ukazuje přetížení metody `Zend_Db_Adapter::update()` k zápisu dat do tabulky s historií.

[35] http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html

Kód 22 DbAdapter_Mysql::update()

```
public function update ($table, array $bind, $where = '')
{
    $select = $this->select()->from($table);
    foreach ($where as $onewhere) {
        $select->where($onewhere);
    }
    $before = $this->fetchAll($select, null, Zend_Db::FETCH_ASSOC);
    $res = parent::update($table, $bind, $where);
    $this->_addToHistory('update', $before, $table, $bind, $where);
    return $res;
}
```

Samotný zápis do tabulky historie pak ukazuje Kód 23. V metodě `update()` adaptéru lze pomocí proměnné `$where` zjistit, které řádky budou ovlivněny, a vybrat je, aby byly později uloženy do historie spolu se změněnými daty jako záloha. Veškerá data jsou serializována pomocí standardní serializační funkce v PHP. Je také uloženo uživatelské jméno uživatele, který změnu provedl. V aplikaci je pak k dispozici jednoduché rozhraní, které umožňuje procházet provedené změny.

Kód 23 DbAdapter_Mysql::_addToHistory()

```
protected function _addToHistory ($type, $before, $table, $bind = null,
$where = null)
{
    $username = (Zend_Auth::getInstance()->hasIdentity() ?
Zend_Auth::getInstance()->getIdentity()->username : '');
    if (count($before) === 1) {
        $before = array_pop($before);
    }
    $row = array(
        'data_before' => serialize($before),
        'table' => serialize($table),
        'bind' => serialize((null === $bind) ? array() : $bind),
        'where' => serialize((null === $where) ? '' : $where),
        'username' => $username,
        'type' => $type
    );
    $this->_historyQueue[] = $row;
    if (! $this->_isTransaction) {
        $this->_flushHistoryQueue();
    }
}
```

Implementace je provedena na úrovni databázové abstrakce (Kód 23). Tím je docíleno toho, že je transparentní jak pro uživatele používajícího aplikaci, tak pro vývojáře, který pracuje s kódem aplikace. To by mělo přispět k snazším úpravám aplikace v budoucnu, neboť ukládání historie bude fungovat bez úprav dále.

Na první pohled je patrné, že toto řešení není ideální, co se týče konzistence původních dat. V čase mezi výběrem dat, která odpovídají podmínce, a samotnou úpravou dat může dojít k zápisu do databáze a výsledek filtru `$where` může být jiný. Této vlastnosti jsem si vědom, ale nepovažuji ji za kritickou, protože aplikace nebude provozována ve vysoce konkurenčním prostředí a současně ji budou používat spíše jednotky uživatelů. Šance, že dva uživatelé budou ve stejný čas pracovat se stejnými daty a navíc tato data budou oba i upravovat, je minimální. Situaci lze řešit například pomocí zamykání tabulek[36] nebo transakcemi[37]. Ale zamykání tabulek je problematické v situaci, kdy dojde k uváznutí skriptu nebo jiné dlouhotrvající operaci, neboť tím budou blokována všechna další spojení. Transakce jsou problematické v tom ohledu, že existuje možnost, že je již transakce aktivní (iniciovaná v nějaké části aplikace), a nové založení transakce poté selže. Logika zápisu do historie a vytváření transakcí by tedy musela být o mnoho složitější. Ani jeden z přístupů tedy není ideální, a proto by podle mě bylo třeba v případě vyšší zátěže a konfliktů nasadit řešení přímo na straně databázového serveru.

3.11.2 Nastavení skriptu pro Phing

V build skriptu (*build.xml*) jsem vytvořil několik úkolů pro často opakované činnosti. Jedná se o **vyčištění cache** (*clear*), které znamená prosté smazání a vytvoření složky cache. Dále obsahuje **sestavení CSS** (*buildcss*) ze zdrojových souborů ve formátu LESS, které automaticky výsledné CSS zmenší odstraněním nepotřebných mezer a nových řádků. Dalším úkolem je **vytvoření exportu struktury databáze** (*migrate*). Příkazy potřebné pro vytvoření struktury databáze jsou uloženy do souboru *migration.sql*, který je verzován. Díky tomu je možné sledovat změny v databázi v jednotlivých revizích. Posledním úkolem je **vytvoření zálohy databáze** (*db-backup*), které využívá nástroje `mysqldump` dostupného spolu s běžnou instalací MySQL. Automaticky vytváří datovaný soubor ve složce *db-backup*.

Součástí build skriptu jsou také definice úkolů pro kontinuální integraci, které mi poskytl Martin Hujer a které využívám, abych mohl sledovat kvalitu kódu aplikace na integračním serveru. Jednotlivá nastavení Hujer hlouběji rozebírá ve své bakalářské práci (HUJER, 2012).

[36] http://dev.mysql.com/doc/refman/5.0/en/lock-tables.html
[37] http://dev.mysql.com/doc/refman/5.0/en/ansi-diff-transactions.html

Parametry, které se mohou lišit mezi různými systémy, jsou uloženy v souboru *build.properties*, který může být upraven pro konkrétní nasazení. Není tak nutné upravovat soubor *build.xml*.

3.12 Použité techniky optimalizace rychlosti

3.12.1 Minimální množství vlastních routovacích pravidel

Jak jsem již předestřel v kapitole 2.3, porušil jsem záměrně pravidlo nemíchat v rámci pojmenování v kódu různé jazyky. Aby bylo jasné, proč tak činím, je nutno nejprve vysvětlit, jak funguje v Zend Frameworku překlad z URL na konkrétní metodu controlleru. Ve výchozím stavu je očekávána URL (pojmem URL rozumíme v tomto případě tu část adresy, která reprezentuje cestu na webovém serveru – tedy obsah odpovídající klíči `PHP_URL_PATH` ve výstupu funkce `parse_url()`) ve formátu `/:controller/:action/`, přičemž hodnoty `index` lze vynechat[38]. Zend Framework však umožňuje tvorbu pravidel, podle kterých se jednotlivá URL na metody překládají. Problémem je, že s rostoucím množstvím pravidel se prodlužuje doba zpracování. V rámci optimalizace rychlosti jsem se kompletně této funkcionalitě vyhnul. Protože jsou metody controlleru pojmenovány česky, lze s výhodou využít výchozího chování a není nutné porovnávat jednotlivá pravidla. Motivací pro česká pojmenování v URL byla především pochopitelnost pro koncového uživatele.

3.12.2 Classmap autoloader

Jak jsem již zmínil v kapitole 1.3.1, v PHP je k načítání tříd ze souborů používán princip autoloadingu. V případě Zend Frameworku (a tedy i aplikace kabelové knihy) jde o autoloading implementovaný podle PSR-0 standardu. Problémem PSR-0 je však fakt, že jsou prohledávány veškeré cesty uvedené v konfigurační direktivě `include_path`. Pokud jich je více a jednotlivé knihovny jsou umístěny z historických nebo objektivních důvodů v různých složkách, dochází k velkému zpomalení. S tímto problémem jsem se setkal i v případě aplikace kabelové knihy, byť jsem již na základě zkušeností z předchozích projektů množství různých cest v `include_path` minimalizoval.

[38] Proto je výchozí URL aplikace prázdné (/) i když je ve skutečnosti interně přeloženo na */index/index/* a tím pádem na *IndexController::indexAction*

Jako zdroj relevantních dat pro porovnání různých metod načítání tříd jsem použil Autoloading Benchmark (O'PHINNEY, 2010), který jde velmi do hloubky a testuje jak variantu s opcode cache, tak bez ní. Z výsledků jasně vyplývá, že nejdelší čas zabere nalezení správného souboru, pokud není cesta k třídě absolutní, ale pouze relativní k `include_path`. Nejvýhodnějším řešením se tak ukázalo použití tzv. *classmap* – tedy pole, ve kterém jsou uloženy všechny třídy a jim odpovídající absolutní cesty.

Již dříve jsem se pokusil takový systém naimplementovat, avšak hlavním problémem byla nutnost dopisovat nově vytvořené třídy do classmapy. V Zend Frameworku verze 2 je tento princip použit jako jeden z hlavních. Ale je k dispozici i verze pro Zend Framework verze 1[39] (o tom, proč nepoužívám přímo ZF 2 se zmiňuji v kapitole 1.4). Výhodou této implementace je, že je možné ji provozovat paralelně s klasickým PSR-0 autoloadingem. To umožní větší flexibilitu a rychlý vývoj za pomoci automatizovaného PSR-0 autoloadingu a použití vygenerované classmapy pro vyšší výkon v produkčním prostředí.

3.12.3 Superluminal plugin

Tento plugin řeší stav, kdy je potřeba mít třídy oddělené do jednotlivých souborů a přitom množství přístupů na disk snižuje výkonnost aplikace. Jedná se o přepsání Zend Framework 2 modulu Superluminal od Evana Couryho[40] pro použití v Zend Frameworku 1. Vydal jsem ho pod velmi volnou MIT licencí na GitHubu[41].

Plugin v metodě `dispatchLoopShutdown()` načte všechny použité třídy a rozhraní, sestaví z nich graf a následně je naskládá podle potřeby za sebe do jednoho souboru. To sníží počet dotazů na disk, které jsou třeba k provedení skriptu. Výsledný soubor se všemi třídami se vloží na začátek skriptu a jedním požadavkem jsou načteny všechny třídy najednou.

Nejmarkantnější změna rychlosti je patrná u disků s velkou přístupovou dobou, kdy vyhledání každého ze stovek souboru může trvat třeba i 10 ms. Na produkčním serveru je vždy třeba otestovat, zda přináší zrychlení, nebo naopak zpomalení. Například v případě SSD disku, kde je přístupová doba minimální, může tento plugin naopak oproti načítání *Just-In-Time* představovat zpomalení.

[39] https://github.com/weierophinney/zf-examples/tree/feature%2Fzf1-classmap/zf1-classmap
[40] https://github.com/EvanDotPro/EdpSuperluminal
[41] https://github.com/tomasfejfar/ZF1-superluminal

3.13 Instalace aplikace na server

Vzhledem k faktu, že aplikace zatím nebyla nasazena na server univerzity, zaměřím se na závěr na postup instalace aplikace na server.

3.13.1 Požadavky aplikace

Vždy uvádím aplikaci, doporučenou verzi a v závorce je uvedena verze používaná při vývoji.

- Server Apache: 2.0 (2.0)
- PHP: 5.3 (5.3.8)

  openssl, short_open_tag: on, safe_mode: off, json, pdo_mysql, pcre
- MySQL: 5 (5.5.16)
- Zend Framework: 1.11 (1.11.11)[42]
- Zend_Form Boostrap[43]

3.13.2 Postup instalace

Aplikace předpokládá, že bude spuštěna v kořenovém adresáři libovolné domény. Nejprve je tedy nutné vytvořit (nejspíše) doménu třetího řádu. Dále je potřeba nasměrovat kořenový adresář domény do složky *public* a nastavit odpovídajícím způsobem na serveru direktivu `AllowOverride` tak, aby fungovala přepisovací pravidla a nastavení v souboru `.htaccess`.

Dalším krokem je nakopírování složky *Zend Frameworku* a *Zend_Form Bootstrapu* do složky, která je součástí `include_path`. Je důležité, aby složka byla v seznamu `include_path` co nejvýše. Ideálně na prvním místě. Výrazně to sníží zátěž serveru a zvýší rychlost aplikace. Jako ideální se jeví symbolický odkaz do složky `library`.

Aplikace potřebuje mít práva pro zápis do adresáře `application/other/cache`, kam se ukládají dočasné soubory.

[42] http://framework.zend.com/releases/ZendFramework-1.11.11/ZendFramework-1.11.11-minimal.zip
[43] https://github.com/tomasfejfar/Zend-Form-Bootstrap/

Pro správnou komunikaci s telefonním seznamem je třeba vytvořit pár RSA klíčů pro asymetrickou šifru ve formátu PEM. Klíče, které jsem používal při vývoji, jsem vytvořil následujícím způsobem:

Kód 24 Vytvoření klíčů

```
openssl genrsa -out private.pem 1024
openssl rsa -in privatekey.pem -pubout -out public.pem
```

Výsledkem jsou dva soubory (`private.pem` a `public.pem`). Soubor s veřejným klíčem je třeba nahrát do telefonního seznamu a nastavit ho do tabulky `api_gateway` odpovídajícím položkám.

Kód 25 Tabulka s přístupovými údaji k API

```
INSERT INTO `api_gateway`
    (`id`, `api_key`, `api_class`, `public_key_path`)
VALUES
    (3, 'vTerrRQU5vfOvj02grm5BeMCIQ86LiQJ', 'ReportChange', '/keys/
public.pem');
```

Pro produkční nasazení bude vhodné testovací klíče nahradit novými klíči tak, aby privátní klíč zůstal neodhalen. Současný privátní klíč je součástí repositáře a jeho bezpečnost tak pro produkční nasazení není zaručena.

3.13.3 Nastavení databáze

Ve vývojovém prostředí běží MySQL server na stejném stroji jako samotná aplikace (s uživatelem `root` a bez hesla). Pro produkční nasazení je třeba upravit přihlašovací jméno, heslo a adresu serveru, kde se databáze nachází – tedy upravit v souboru `application/Config.php` hodnotu v poli `Config:: $_db`.

3.13.4 Další nastavení

Dále je třeba v telefonním seznamu nastavit API klíče (sloupec `api_key`) pro `UsersOnLine` a `ReportChange` (sloupec `api_class`). A tyto klíče pak vyplnit v `Config.php` do proměnné `$_tsApiKeys`.

Nakonec je třeba nastavit v aplikaci momentální URL telefonního seznamu do konstanty `Config::TS_URL`.

4 Závěr

Aplikace kabelové knihy je vytvořena v jazyce PHP za pomoci Zend Frameworku. Pokud je to možné, využívá principů objektového programování k zaručení snadné rozšiřitelnosti. Je verzována nástrojem pro správu zdrojového kódu Git. Veškeré nástroje použité pro vývoj aplikace jsou volně dostupné pod open source licencí. Významnou výhodou je tedy fakt, že nasazení systému nepředstavuje, kromě lidských zdrojů, žádné další náklady na hardware nebo software. V rámci práce bylo vytvořeno několik open source knihoven a ty jsou úspěšně používány dalšími vývojáři.

V rámci diplomové práce se mi podařilo vytvořit aplikaci, která může nahradit současnou verzi kabelové knihy. Oproti současné verzi kontroluje aplikace integritu dat. Díky centrálnímu úložišti dat umožňuje snadnou spolupráci více osob najednou a zaručuje jim, že budou pracovat s aktuálními daty.

Aplikace zatím není nasazena do provozu. Podle plánu by měla být nasazena během následujících měsíců, až budou zkonvertována současná data do databáze. Předpokládám, že bude nutné, abych v aplikaci provedl ještě nějaké úpravy podle toho, jaké požadavky se projeví při skutečném provozu, podobně jako tomu bylo u aplikace telefonního seznamu.

5 Bibliografie

AMOROSO, E., 2010. *Cyber Attacks: Protecting National Infrastructure* [online]. Burlington: Elsevier Science [cit. 2012-02-07]. ISBN 0123849179. Dostupné z: http://books.google.cz/books?id=N0qXBzV1mpkC

CAMPI, N. a K. BAUER, 2008. *Automating Linux and Unix System Administration* [online]. 2, ilustrované vydání. Apress [cit. 2012-04-10]. ISBN 1430210591. Dostupné z: http://books.google.cz/books?id=HUzWZN4Z_HsC

CARR, J., 2011. *Inside Cyber Warfare: Mapping the Cyber Underworld* [online]. 2. vyd. Sebastopol: O'Reilly & Associates [cit. 2012-04-20]. ISBN 1449310044. Dostupné z: http://books.google.cz/books?id=nFP9wrNmGhcC

CLARKE, J. R. M. ALVAREZ a D. HARTLEY, 2009. *SQL Injection Attacks and Defense* [online]. Syngress Pub [cit. 2012-04-12]. ISBN 1597494240. Dostupné z: http://books.google.cz/books?id=yfP9e8k6sIUC

DASWANI, N. C. KERN a A. KESAVAN, 2007. *Foundations of Security: What Every Programmer Needs to Know* [online]. Berkeley: Apress [cit. 2012-03-25]. ISBN 1590597842. Dostupné z: http://books.google.cz/books?id=zwAT_dcL84YC

FOWLER, M., 2003. *Patterns of Enterprise Application Architecture* [online]. 3.vydání. Boston, Massachusetts: Addison-Wesley [cit. 2012-04-29]. ISBN 9780321127426. Dostupné z: http://books.google.cz/books?id=FyWZt5DdvFkC

GARRETT, J. J., 2005. Adaptive Path. *Ajax: A New Approach to Web Applications* [online]. 18. 02. 2005 [cit. 2012-03-03]. Dostupné z: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

HICKSON, I., 2012. World Wide Web Consortium. *HTML5: Edition for Web Authors* [online]. 2. 3. 2012 [cit. 2012-03-03]. Dostupné z: http://dev.w3.org/html5/spec-author-view/spec.html#html-vs-xhtml

HOMAKOV, E., 2012. A few CSRF-like vulnerable examples. [online]. 2. 4. 2012 [cit. 2012-04-07]. Dostupné z: http://homakov.blogspot.com/2012/03/hacking-skrillformer-moneybookers.html

HOMAKOV, E., 2012. CSRF Is A Vulnerability In All Browsers - You Should Do Something. [online]. 30. 3. 2012 [cit. 2012-05-04]. Dostupné z: http://homakov.blogspot.com/2012/03/1-csrf-is-vulnerability-in-all-browsers.html

HUJER, M., 2012. *Kontinuální integrace při vývoji webových aplikací v PHP*. Praha. Bakalářská práce. Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky [cit. 2012-05-01]. Dostupné z: http://www.scribd.com/mhujer/d/90668202

LAWSON, B., 2010. Bruce Lawson's personal site. *A Minimal HTML5 Document* [online]. 22. 7. 2010 [cit. 2012-03-03]. Dostupné z: http://www.brucelawson.co.uk/2010/a-minimal-html5-document/

O'PHINNEY, M. W., 2010. phly, boy, phly. *Autoloading Benchmarks* [online]. 17. 08. 2010 [cit. 2012-04-07]. Dostupné z: http://mwop.net/blog/245-Autoloading-Benchmarks.html

PFLEEGER, C. P. a S. L. PFLEEGER, 2011. *Analyzing Computer Security: A Threat/Vulnerability/Countermeasure Approach* [online]. přepracované vydání. Prentice Hall Professional [cit. 2012-03-18]. ISBN 0132789469. Dostupné z: http://books.google.cz/books?id=nVaCwXp_S8wC

SWISHER, J., 2011. Mozilla Developer Center. *Cascading and inheritance* [online]. 06. 06. 2011 [cit. 2012-03-03]. Dostupné z: https://developer.mozilla.org/en/CSS/Getting_Started/Cascading_and_inheritance

VAN KESTEREN, A. a S. PIETERS, 2012. World Wide Web Consortium. *HTML5 differences from HTML4* [online]. 5. 2. 2012 [cit. 2012-03-03]. Dostupné z: http://dev.w3.org/html5/html4-differences/

W3C, 2012. World Wide Web Consortium. *HTML & CSS* [online]. 02. 03. 2012 [cit. 2012-03-03]. Dostupné z: http://www.w3.org/standards/webdesign/htmlcss#whatcss

ZANDSTRA, M., 2010. *Php Objects, Patterns and Practice* [online]. 3. ilustrované vydání. Apress [cit. 2012-04-10]. 143022925X. Dostupné z: http://www.google.cz/books?id=hE9Qf-tqR0oC

ZELLER, W. a E. W. FELTEN, 2008.*Cross-Site Request Forgeries: Exploitation and Prevention* [online]. Princeton University, verze 10/15/2008. Dostupné také z: http://www.cs.utexas.edu/users/shmat/courses/library/zeller.pdf