Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky

Obor: informační a znalostní inženýrství

Diplomová práce

Komponentový framework pro vývoj webových aplikací

Diplomant: Bc. Jiří Staniševský
Vedoucí práce: Ing. Jiří Kosek

Praha 2011

Prohlášení

Prohlašuji, že jsem vypracoval samostatně diplomovou práci na téma *Komponentový framework pro vývoj webových aplikací*. Použitou literaturu a další podkladové materiály uvádím v přiloženém seznamu literatury.

V Praze dne 2. května 2011 .....

podpis diplomanta

Abstrakt

Cílem této práce bylo navrhnout a implementovat komponentový framework pro tvorbu webových aplikací v jazyce PHP. Základem pro návrh tohoto frameworku byla analýza výhod a nevýhod použití komponentových frameworků, srovnání dostupných frameworků a hodnocení jejich nedostatků. Hlavním požadavkem na nový framework a zároveň jeho největším přínosem je nezávislost na koncovém zařízení, tedy schopnost automaticky generovat rozdílný výstup pro mobilní zařízení a osobní počítače, při zachování pouze jedné aplikační logiky. Toho framework dosahuje pomocí vlastní originální implementace Model-View-Controller architektury, realizované pomocí tzv. renderovacích objektů. Veškerá aplikační logika je navíc řízená událostmi, které jsou generovány jednotlivými komponentami frameworku a kód se tak stává srozumitelnějším a lépe udržovatelným. Framework se snaží nabídnout uživatelům mobilních telefonů a jiných zařízení komfortní ovládání aplikací bez toho, aby vývojáře těchto aplikací nutil k vývoji jejich speciálních verzí pro tato zařízení.

Abstract

The aim of this work was to design and implement component framework for development of web applications written in PHP programming language. The design has been based on pros and cons analysis of component framework usage, existing framework solution comparison and analysis of drawbacks of these frameworks. End-user device independence is both the main requirement for this framework and also its greatest contribution to the field of web frameworks. To be able to generate different output for mobile and desktop devices and to keep the application logic at one place only, thus minimizing the amount of work required from the developer, it uses its own original implementation of Model-View-Controller architecture. The output is being generated by so called rendering objects associated with individual components. Applications based on this framework are purely event driven, thus having their code more readable and manageable. Framework tries to offer comfortable usage of web applications for mobile device users without forcing application developers to create additional versions of their applications.

Obsah

Úvod....1

kapitola 1:
Výhody a nevýhody použití frameworků

1.1 Pohled programátora....4
1.1.1 Eliminace stále se opakujícího kódu....4
1.1.2 Nižší dopad rychlosti vývoje na důkladnost....5
1.1.3 Bezpečnost....5
1.1.4 Knihovna funkcí....5
1.1.5 Abstraktní vrstva....6
1.1.6 Hardwarové nároky....8
1.1.7 Spolupráce v týmu....8
1.2 Pohled uživatele....9
1.2.1 Bonusy pro uživatele....9
1.2.2 Sjednocení prostředí....9
1.2.3 Rychlost odezvy....10
1.3 Různé techniky, aneb na co máte chuť?....12
1.3.1 Špagetový kód....12
1.3.2 Raviolový kód....15
1.3.3 Lasagnový kód....15
1.3.4 Framework....16
1.4 Shrnutí....17

kapitola 2:
Srovnání existujících frameworků

2.1 Porovnání frameworků....20
2.2 Srovnávané frameworky....20
2.2.1 Symfony....20
2.2.2 Zend Framework....21
2.2.3 PRADO....21
2.2.4 Yii....23
2.2.5 CodeIgniter....23
2.2.6 Nette....23
2.3 Hodnocení....24
2.3.1 Technické aspekty....24
2.3.2 Netechnické aspekty....26
2.3.3 Shrnutí....27
2.4 Motivace....27

kapitola 3:
Koncept ROAD frameworku

3.1 Základní koncept....30
3.2 Architektura frameworku....31
3.3 Integrace frameworku do aplikace....32

3.4 Události....33
3.5 Optimalizace pro různá koncová zařízení....34
3.5.1 Nezávislost na webovém prohlížeči....35
3.5.2 Nezávislost na platformě....35
3.6 Práce s databází....38
3.6.1 Abstraktní vrstva....38
3.6.2 Objektově-Relační Mapování....38
3.7 Podpora práce s formuláři....39
3.7.1 Aktivní prvky formuláře....40
3.7.2 Automatizace databázových operací....41
3.7.3 Více záznamové formuláře....41
3.8 Podpora více jazyků....42
3.9 Shrnutí....42

kapitola 4:

Implementace ROAD frameworku

4.1 Struktura frameworku....46
4.1.1 Struktura tříd....46
4.1.2 Struktura souborů....49
4.2 Požadavky na systém....50
4.3 Definice stromu komponent....50
4.4 Práce s událostmi....52
4.5 Jádro....55
4.5.1 Třída TRoad....55
4.6 Inicializační soubor....59
4.7 Načtení a spuštění frameworku....59
4.7.1 Zaváděcí soubor....59
4.7.2 Spuštění frameworku....60
4.8 Renderování stránky....61
4.8.1 Třída TCustomOutput....61
4.8.2 Třída TCustomRenderer....62
4.8.3 Výběr render-setu....63
4.8.4 Přizpůsobení výstupu....64
4.9 Konverze formátů....65
4.10 Práce s databází....66
4.10.1 Čtení dat....67
4.10.2 Zápis dat....67
4.11 Zpracování formulářů....68
4.11.1 Definice formuláře....68
4.11.2 Obsluha formuláře....68
4.11.3 Propojení s databází....69
4.12 Problémy a jejich řešení....70
4.12.1 Pomalé hledání komponent podle názvu....70
4.12.2 Rychlost odezvy aplikace....71
Závěr....75

příloha A:

Tvorba jednoduché aplikace

A.1 Popis aplikace....80
A.2 Instalace frameworku....80
A.3 První krůčky....82
A.4 Seznam výletů....86
A.5 Formulář pro úpravu výletu....90
A.6 Přidání nové destinace....96
A.7 Správa služeb....99
A.8 Přihlášení uživatele a omezení práv....102
A.9 Vlastní šablona....105
A.10 Vylepšení titulní stránky....108
A.11 Shrnutí....110

Seznam tabulek

tabulka č. 1 - Faktory určující rychlost odezvy aplikace....10
tabulka č. 2 - Srovnání programovacích technik....16
tabulka č. 3 - Test časové náročnosti běhu frameworku....72
tabulka č. 4 - Zrychlení odezvy s použitím nástroje Alternative PHP Cache....73
tabulka č. 5 - Vybrané možnosti zobrazení komponenty TMultiSelectBox....95

Seznam výpisů kódu

výpis kódu č. 1 - Špagetový kód....12
výpis kódu č. 2 - Strukturovaný kód....13
výpis kódu č. 3 - Špagetový PHP kód....13
výpis kódu č. 4 - Ukázka souboru .page frameworku PRADO....22
výpis kódu č. 5 - Ukázka souboru .php frameworku PRADO....22
výpis kódu č. 6 - Příklad integrace frameworku přes index.php....32
výpis kódu č. 7 - Příklad integrace frameworku přes načtení v souborech stránek....33
výpis kódu č. 8 - Procedurální obsluha aplikace....33
výpis kódu č. 9 - Událostmi obsluhovaná aplikace....34
výpis kódu č. 10 - Systém vlastností v Delphi (Object Pascal)....47
výpis kódu č. 11 - Tvorba stromu komponent....51
výpis kódu č. 12 - Nastavení vlastností komponent....51
výpis kódu č. 13 - Vyhledání existující komponenty....51
výpis kódu č. 14 - Tvorba stromu komponent s nastavením vlastností....52
výpis kódu č. 15 - Jednoduchá událost....53
výpis kódu č. 16 - Využití třídy TPanel....54

výpis kódu č. 17 - Zkrácený zápis obsluhy události....54
výpis kódu č. 18 - Neměnný obsah zaváděcího souboru....60
výpis kódu č. 19 - Obsluha událostí při inicializaci jádra....60
výpis kódu č. 20 - Načtení zaváděcího souboru ze souboru stránky....60
výpis kódu č. 21 - Spuštění frameworku v souboru stránky....61
výpis kódu č. 22 - Přizpůsobení vzhledu stránky....64
výpis kódu č. 23 - Převod formátů pomocí TCustomValue....66
výpis kódu č. 24 - Zjednodušený zápis převodu formátů....66
výpis kódu č. 25 - Modifikace databázového záznamu prostřednictvím ROAD frameworku....67
výpis kódu č. 26 - Obsluha formuláře....69
výpis kódu č. 27 - Propojení formuláře s databází....70
výpis kódu č. 28 - Konfigurace připojení k databázovému serveru....81
výpis kódu č. 29 - Konfigurační soubor....81
výpis kódu č. 30 - Zamezení výpisu chybových hlášek....81
výpis kódu č. 31 - Základní struktura souboru stránky....82
výpis kódu č. 32 - Ahoj světe....83
výpis kódu č. 33 - Automatické obnovení prvku....83
výpis kódu č. 34 - Převod formátů pomocí speciálních tříd....84
výpis kódu č. 35 - Převod formátů pomocí objektu $convertor....84
výpis kódu č. 36 - Popisek stránky....84
výpis kódu č. 37 - Odkazy na hlavní stránce....85
výpis kódu č. 38 - Hlavní stránka aplikace (index.php)....85
výpis kódu č. 39 - Seznam záznamů tabulky vylety....86
výpis kódu č. 40 - Seznam záznamů tabulky vylety....86
výpis kódu č. 41 - Selekce a pojmenování sloupců....87
výpis kódu č. 42 - Implicitní třídění....88
výpis kódu č. 43 - Odkaz na formulář pro úpravy....88
výpis kódu č. 44 - Zjednodušený zápis obsluhy události....88
výpis kódu č. 45 - Ruční vytvoření sloupce....89
výpis kódu č. 46 - Aktivování AJAXu....90
výpis kódu č. 47 - Stránka se seznamem výletů....90
výpis kódu č. 48 - Kostra formuláře....91
výpis kódu č. 49 - Výběrový seznam....91
výpis kódu č. 50 - Zkrácený zápis definice komponenty....92
výpis kódu č. 51 - Textová pole....92
výpis kódu č. 52 - Seznam pro výběr více položek....93
výpis kódu č. 53 - Víceřádkové vstupní pole....93
výpis kódu č. 54 - Tlačítko zpět....93
výpis kódu č. 55 - Provázání formuláře s databází....94
výpis kódu č. 56 - Tlačítko pro vytvoření nového výletu....96
výpis kódu č. 57 - Návrh formuláře pro přidání destinace....97
výpis kódu č. 58 - Otevření vnořeného okna....97
výpis kódu č. 59 - Uložení nové destinace....98
výpis kódu č. 60 - Formulář pro služby....99
výpis kódu č. 61 - Ukládání formuláře s více záznamy....100

výpis kódu č. 62 - Tlačítka pro přidání nového záznamu....101
výpis kódu č. 63 - Neprázdné prázdné textové pole....101
výpis kódu č. 64 - Tlačítko pro odstranění záznamu....102
výpis kódu č. 65 - Přihlášení uživatele....103
výpis kódu č. 66 - Vyhodnocení oprávnění....103
výpis kódu č. 67 - Omezení přístupu k formuláři pro úpravu výletu....103
výpis kódu č. 68 - Omezení přístupu ke správě služeb....104
výpis kódu č. 69 - Skrytí odkazu na úpravu výletu pro nepřihlášené návštěvníky....104
výpis kódu č. 70 - Vyžádání přihlášení....105
výpis kódu č. 71 - Záhlaví stránky....107
výpis kódu č. 72 - Zápatí stránky....107
výpis kódu č. 73 - Připojení stylového souboru....108
výpis kódu č. 74 - Vlastní komponenta....109
výpis kódu č. 75 - Renderování vlastní komponenty....109
výpis kódu č. 76 - Kontrola aktuálně používaného render-setu....110

Seznam obrazovek

obrazovka č. 1 - Základní formulář....51
obrazovka č. 2 - Formulář včetně popisků....52
obrazovka č. 3 - Reakce na událost....53
obrazovka č. 4 - Ověření instalace frameworku....82
obrazovka č. 5 - Ahoj světe....83
obrazovka č. 6 - Odkazy na hlavní stránce....85
obrazovka č. 7 - Seznam záznamů tabulky vylety....86
obrazovka č. 8 - Selekce a pojmenování sloupců....87
obrazovka č. 9 - Odkaz v seznamu výletů....89
obrazovka č. 10 - Základ formuláře pro úpravu výletu....93
obrazovka č. 11 - Formulář propojený s databází....94
obrazovka č. 12 - Formulář pro přidání nového výletu....96
obrazovka č. 13 - Vnořený formulář pro přidání destinace....98
obrazovka č. 14 - Formulář pro služby....100
obrazovka č. 15 - Ukládání formuláře s více záznamy....100
obrazovka č. 16 - Tlačítka pro přidání nového záznamu....101
obrazovka č. 17 - Kompletní správa služeb....102
obrazovka č. 18 - Omezení přístupu k formuláři pro úpravu výletu....104
obrazovka č. 19 - Vyžádání přihlášení....105
obrazovka č. 20 - Vzorová šablona....106
obrazovka č. 21 - Stránka se záhlavím, zápatím a připojeným stylovým souborem....108

Úvod

Vážení čtenáři, ve svých rukou právě držíte text mé diplomové práce, jejíž zadání zní: *Komponentový framework pro vývoj webových aplikací*. Samotná diplomová práce se skládá z tohoto textu a především z prototypové implementace frameworku **ROAD** (**R**apid **O**nline **A**pplication **D**evelopment). Její tvorba mi zabrala s přestávkami přibližně jeden rok.

Online slovník Merriam-Webster definuje slovo framework jako *basic conceptional structure (as of ideas)*[1], tedy jako základní strukturu konceptů (myšlenek), což odpovídá českému překladu *rámec*. Framework si tedy můžeme v jeho širším významu vyložit jako soubor ideí a postupů, které vedou k vytyčenému cíli. V oboru softwarového inženýrství si však slovo framework postupně získalo poněkud konkrétnější a hmatatelnější význam. Proto se na následujících přibližně sto stránkách budeme o frameworku bavit jako o „softwarové struktuře, sloužící jako podpora při programování a vývoji jiných softwarových projektů“[2]. Protože ROAD framework slouží pro usnadnění vývoje v jazyce PHP, omezím se ve valné většině textu na frameworky určené právě pro tuto platformu.

V následujících kapitolách bude mým úkolem analyzovat výhody a nevýhody frameworků, porovnat dostupné frameworky a navrhnout vlastní, alespoň v některých ohledech lepší řešení. Další kapitola bude vyhrazena popisu implementace navrženého řešení a jako přílohu uvedu i krátký tutoriál, který vás provede tvorbou jednoduché webové aplikace založené na ROAD frameworku.

Jelikož hlavní částí mé diplomové práce není tento text, ale samotný implementovaný framework, máte možnost si prohlédnout jeho zdrojové kódy, umístěné na přiloženém CD-ROM, případně si ho vyzkoušet v praxi na ukázkové aplikaci (viz *příloha A*).

kapitola 1:
Výhody a nevýhody použití frameworků

Jako každá programovací technika má i použití frameworků svoje klady a zápory. V této kapitole se budeme snažit na komponentový framework podívat nejen jako na nástroj usnadňující tvorbu aplikací, ale i jako na příčinu rozličných problémů.

1.1 Pohled programátora

Každý framework je primárně určen k usnadnění práce programátora. Proto je určitě názor programátora na výhody a nevýhody jeho použití nejdůležitější, ne-li přímo rozhodující.

1.1.1 Eliminace stále se opakujícího kódu

Jedním z hlavních důvodů, které vedou programátory k použití nejrůznějších frameworků, je automatizace nejčastěji se opakujících operací. Jestli v některém oboru neplatí pravidlo *„opakování matka moudrosti“*, pak je to právě v programování. Opakování podobného nebo nedejbože stejného kódu[1] na několika různých místech v programu je oprávněně všeobecně považováno za velmi špatný zlozvyk. Použitím frameworku se minimalizuje nutnost opakování a jsou tak omezeny jeho nepříznivé důsledky, avšak z pevně dané struktury frameworku plynou i jistá omezení.

Zvýšení produktivity programátora

Možná tím nejevidentnějším nedostatkem způsobu programování, při kterém se na více místech opakuje stejný kód, je z dlouhodobého hlediska snížení produktivity programátora. Ten musí při tvorbě nového programu, nebo jeho části, buď pracně prohledávat předchozí programy a hledat v nich kód, který by mohl znovu použít, nebo znovu příkaz po příkazu všechno naprogramovat. Framework programátorovi poskytuje znovu využitelné procedury nebo komponenty, které s minimálním množstvím dodatečného kódu vykonávají požadovanou funkci.

Menší pravděpodobnost chyby v programu

S opakováním kódu se v programu samozřejmě opakují i chyby, které v nich programátor napáchá. Pokud se nám chybu podaří v jedné části programu lokalizovat a opravit, je vysoce pravděpodobné, že se stejná chyba nachází na několika dalších místech v programu. Čím větší je projekt, na kterém programátor pracuje, tím menší je šance, že v případě nalezení chyby ji programátor odstraní ze všech míst v programu, ve kterých se vyskytuje. Hlavně z tohoto důvodu je žádoucí se opakování kódu pokud možno vyhnout.

Menší flexibilita

Asi jedinou výhodou opakování kódu v programu je možnost tento kód drobnou úpravou pokaždé přizpůsobit přímo požadavkům daného programu. Pokud program využívá nějaký framework, je funkce procedur a komponent tohoto frameworku více či méně pevně daná a přizpůsobení je realizováno pouze

1 Kódem v tomto kontextu myslíme soubor několika příkazů, které spolu dohromady vykonávají funkci programu.

parametrizací. Větší možnost přizpůsobení funkce prostřednictvím parametrizace je však vykoupena složitějším rozhraním a větším objemem frameworku, který musí počítat s větším množstvím stavů aplikace.

1.1.2 Nižší dopad rychlosti vývoje na důkladnost

Jelikož čas jsou peníze, snaží se každý racionálně uvažující jedinec placený za provedenou práci, tuto práci odvést co nejrychleji, aby mohl zbytek svého času věnovat jiné práci nebo nějaké příjemnější kratochvíli. S rostoucí rychlostí vývoje ale klesá pozornost, kterou programátor vývoji věnuje. Programátor přestává přemýšlet o mimořádných stavech programu a soustředí se čím dál více pouze na stav ideální. Náplní práce autorů frameworků je přemýšlet o všech možných stavech, které mohou nastat, a zajistit správnou funkci aplikace v každém z těchto stavů. Programátor aplikace se pak může soustředit výhradně na naplnění primární funkce programu. Aplikace se tak díky frameworkům stávají bezpečnějšími (viz níže) a kompatibilnějšími s různými koncovými zařízeními.

1.1.3 Bezpečnost

Z hlediska bezpečnosti framework přispívá především tím, že se snaží za programátora ošetřit všechny drobné skulinky, které při psaní aplikací ad-hoc vznikají. Programátor tak nemusí myslet na ošetření hodnot ve formulářích proti útokům typu SQL-injection[2][3] nebo na odhalení pokusů o hacknutí aplikace pomocí podstrčení ukradeného session id. Frameworky bohužel mají z hlediska bezpečnosti jednu velkou nevýhodu. A to tu, že každá bezpečnostní díra ve frameworku se vyskytuje i u všech ostatních instancí dané verze. Jakmile je tedy jednou díra odhalena, je proti útokům bezmocná i každá další aplikace na frameworku běžící. Navíc, čím známější je framework, tím více nenechavců bude určitě díry hledat a čím rozšířenější je, tím více aplikací bude pravděpodobně napadeno.

Z hlediska bezpečnosti je proto důležité, aby tvůrce frameworku rychle reagoval na nalezené bezpečnostní díry a aby uživatelé frameworku své instalace pravidelně aktualizovali.

1.1.4 Knihovna funkcí

Žádný programátor není vševědoucí. Díky frameworkům ale může uživatelům svých aplikací nabídnout i funkce, které by sám nebyl schopen naprogramovat, nebo by to pro něj představovalo příliš složitý a tudíž nákladný proces. Framework jako knihovna funkcí slouží jako černá schránka, z níž je vidět pouze rozhraní a po vykonání požadované operace i výsledek. Odpadá však možnost sledovat a modifikovat průběh této operace. Programátor se tak musí spolehnout na to, že tvůrce frameworku funkcionalitu implementoval kvalitně a pokud možno

2 Útočník vloží do textového údaje ukládaného do databáze škodlivý příkaz. Tento útok je možný, pokud aplikace správně neošetřuje znaky ' v údajích.

bezchybně. Pokud operace z nějakého důvodu selže, často si nemůžeme být jisti, zda jde o chybu v našem kódu, nebo přímo v frameworku. Jsme-li vůbec schopni takovouto chybu ve frameworku lokalizovat a opravit, musíme kvůli tomu modifikovat přímo soubory frameworku a odpadá nám tak možnost kdykoliv bezproblémově aktualizovat framework. Chybu a případně i s řešení je pak nutné nahlásit autorům frameworku a doufat, že její opravu zahrnou do oficiálním balíčku.

1.1.5 Abstraktní vrstva

Jak již bylo nastíněno na konci kapitoly 1.1.2, aplikace, které využívají frameworky, mají většinou širší kompatibilitu jak s koncovými zařízeními, tak i s operačními systémy, databázovými systémy a dalšími nástroji. K tomuto účelu využívají frameworky abstraktní vrstvy, které překládají programátorovy příkazy do formátu zpracovatelného konkrétním nástrojem. Skrz tyto abstraktní vrstvy zas nástroje zpětně interagují s programem tvořeným programátorem. Abstraktní vrstvy tedy zprostředkovávají vstup a výstup do a z programu.

Díky použití abstraktní vrstvy jsou na programátora kladeny nižší nároky co se týče znalostí dané technologie, protože všechny funkce které nástroj poskytuje, jsou zprostředkované frameworkem.

Tak jak jsme již psali v kapitole 1.1.4, každá černá schránka má i svoje podstatné stinné stránky. Abstraktní vrstva tak, jak o ní v této kapitole mluvíme, nepochybně černou stránkou se vším všudy je, a tudíž nese i všechny její nevýhody.

Koncové zařízení

Pokročilé frameworky ulehčují programátorovi práci tím, že detekují konfiguraci koncového zařízení[3], ze kterého uživatel k aplikaci přistupuje a přizpůsobují této konfiguraci běh aplikace. Programátor se pak může soustředit hlavně na to, čeho chce uživatel svými akcemi dosáhnout a ne na to, jak toho dosáhnout v daném prostředí. Kvalitní framework by tedy měl zajistit:

- **jednotný vstup:** nasměrování akcí provedených z různých koncových zařízeních do jednoho místa v kódu tak, aby mohl programátor tvořit aplikační logiku pouze jednou
- **jednotný výstup:** generovat výstupy pro jednotlivá koncová zařízení z dat předaných frameworku v jednotném formátu programátorem

Operační systém

Některé jazyky pro psaní webových aplikací (včetně PHP) jsou provozovatelné na více operačních systémech a jistá kompatibilita je tak zajištěna přímo zvolenou

3 Koncovým zařízením v případě webových aplikací myslíme buď prohlížeč webových stránek nebo typ hardwarového zařízení, na kterém je tento prohlížeč používán (např. osobní počítač, mobilní telefon, PDA).

technologií. Avšak i tak jsou mezi jednotlivými operačními systémy rozdíly, které znepříjemňují tvorbu aplikací. V PHP jsou například některé funkce dostupné pouze na systému Linux. Framework by měl nahradit nezbytné chybějící funkce ekvivalentními funkcemi napsanými například přímo v PHP. Programátor by měl mít možnost volat funkce jednotně, aby byl program jednoduše přenositelný mezi operačními systémy. Framework by tedy měl zajistit detekci operačního systému a poté volat funkce specifické pro příslušný operační systém.

Databázový systém

Další technologií, jejíž specifika je vhodné odstínit od programátora a aplikační logiky, je určitě databázový systém. Žádná kvalitní webová aplikace se dnes bez databázového systému podporujícího jazyk SQL neobejde. Abstraktní vrstva mezi programátorem a databázovým systémem nebo serverem musí zprostředkovávat vykonávání dotazů, správu transakcí a zpracovávání výsledků dotazu.

Abstrakce od databázového systému je kritická v případě, že nám jde o přenositelnost aplikace. Pokud je abstrakce kvalitní, není žádný problém databázi přesunout ze serveru jednoho typu na server jiného typu, aniž bychom jakkoliv upravovali samotnou aplikaci.

Databázové abstrakce použité ve frameworcích většinou nabízejí i množství nadstandardních funkcí usnadňujících práci s databází. Rovněž mohou odstraňovat nedostatky některých databázových systému. Nepodporuje-li například systém změnu vstupního a výstupního kódování, není problém pro abstraktní vrstvu dotazy do databáze převést do nativního kódování databázového serveru a po získání výsledků je převést zpět do kódování, které používá framework a přímo aplikace.

ORM

Některé frameworky navíc umožňují částečně abstrahovat i od samotného jazyka SQL. Například upravovat záznamy databáze na úrovni aplikační logiky:

- záznamy získané příkazem SELECT jsou abstraktní vrstvou uloženy v nějaké jazykové struktuře (např. objektu nebo poli)
- tato struktura je poté aplikací modifikována
- a abstraktní vrstva vygeneruje příkaz UPDATE s hodnotami které se změnily a podle polí obsažených v primárním klíči určí, který záznam v databázi má být změněn

Zde už pomalu vplouváme do tematiky Objektově Relačního Mapování[4] neboli ORM. Ideou ORM je zpřístupnit data z databáze formou objektů, reprezentujících entity, které má konkrétní tabulka databáze zachytit. ORM nadstavba nad databázovým serverem si bere za úkol synchronizovat data mezi databází a těmito objekty. Místo odesílání dotazů do databáze voláme funkce v nativním jazyce programovacího prostředí a měníme vlastnosti objektů. ORM vrstva se pak sama postará o promítnutí zpět do databáze.

Nevýhody

Stejně jako u každé abstraktní vrstvy je i použití databázové abstrakce vykoupeno neprůhledností logiky a snížením flexibility. Chceme-li abstrakci využívat dokonale a umožnit tak přenos aplikace na jinou technologii, musíme bohužel oželet některé funkce specifické pro konkrétní systém. Platí totiž pravidlo, že pokud má být abstraktní vrstva naprosto nezávisle použitelná, může ve svém rozhraní zpřístupňovat pouze ty funkce, které nabízí každý databázový systém, jehož služeb může aplikace využívat (na který existuje pro danou abstrakční vrstvu funkční ovladač).

1.1.6 Hardwarové nároky

Framework sám o sobě je souborem tříd, funkcí a dalších struktur potřebných pro jeho funkci, které jsou definovány v tom samém programovacím jazyce, pro který je framework určen. Kód frameworku tedy zabírá místo na disku a spotřebovává výpočetní čas procesoru stejně jako jakýkoli jiný kód. U nekompilovaných skriptovacích jazyků jako je PHP, přibývá ještě potřeba kód před vykonáním analyzovat a z obyčejného textového souboru převést do paměti na odpovídající programové struktury.

Čím více funkcí framework nabízí, tím je jeho kód složitější a tím náročnější je pro počítač analýza a vykonání jeho kódu. Je tedy evidentní, že pro zachování stejné rychlosti odezvy aplikace budeme muset při použití složitějšího frameworku použít silnější hardware. Záleží na nás, zda využijeme jednodušší framework nabízející méně funkcí a urychlíme tak chod aplikace, nebo obětujeme prostředky na koupi výkonnějšího hardware. Neekonomické je ale volit frameworky, které nabízejí množství zbytečných funkcí, jenž naše aplikace vůbec nevyužije.

Obecně můžeme říci, že použití frameworku zvyšuje nároky na hardware a zpomaluje běh aplikace. Zaměříme-li se ale jen na určité dílčí úkony, které aplikace provádí, mohou být tyto úkony vykonány i rychleji než bez použití frameworku. Jak již bylo mnohokrát řečeno, framework nabízí omezené množství omezeně flexibilních funkcí, které jsou ale dost často dotaženy do mnohem kvalitnější podoby než jaké by byl vůbec programátor schopen samostatně dosáhnout. V rámci tohoto zkvalitňování často tvůrci frameworku provádějí i optimalizaci kódu. Kód často opakovaných jednoduchých úkonů tak bývá ve frameworcích dobře optimalizován a jejich vykonání je díky tomu rychlejší.

1.1.7 Spolupráce v týmu

Představte si, že jste programátor přebírající částečně vyvinutou aplikaci po jiném programátorovi a snažíte se porozumět jejímu kódu. Bude se vám lépe přebírat kód plný proměnných, jejichž jména nedávají žádný smysl, funkcí, které sami o sobě nedělají vůbec nic a několika úrovňových cyklů, nebo program využívající Vám dobře známého (nebo alespoň dobře zdokumentovaného) frameworku? Frameworky už podle svého názvu (*„frame“ → „rámec“*) tlačí programátora k dodržování frameworkem předem určených principů. Orientace v kódu je pak

díky tomu mnohem snazší i pro další osoby.

Další vlastností frameworku je redukce kódu aplikační logiky. Díky frameworku je spousta řádků kódu potřebného pro vykonání určité akce nahrazeno jedním řádkem, ve kterém je volána funkce frameworku. Kód aplikační logiky je tak nejen o dost čitelnější, ale mnohem lépe se také spravuje. Pokud je potřeba program upravit, můžeme ho upravit změnou parametrů funkce frameworku, změnou pořadí volání jednotlivých funkcí nebo u komponentových frameworků nastavením vlastností jednotlivých komponent. Opět tedy měníme pouze to, *„co“* se má udělat, ne to, *„jak“* se to má udělat.

Používáme-li framework vyvíjený v rámci našeho týmu, mohou být v týmu vybráni zdatní programátoři, kteří mají na starosti správu kódu frameworku a ostatní framework pouze využívají jako černou skřínku. Tím, že úpravy do frameworku budou zanášet pouze pověřené osoby, je zajištěno, že programátoři nebudou kód frameworku měnit podle své potřeby a ztěžovat práci ostatním, kteří framework využívají na jiném místě programu.

Samozřejmým efektem používání frameworku v týmu je sjednocení způsobu, jakým programátoři využívají jednotlivé technologie. Tato skutečnost je dána tím, že ke konkrétním technologiím se většinou přistupuje pouze přes framework a způsob přístupu je tak vždy stejný.

1.2 Pohled uživatele

I když jsou frameworky primárně určeny pro ulehčení práce programátorů, zcela jistě ovlivňují i způsob, jakým mohou uživatelé pracovat s jejich aplikacemi. K čemu by nám byl komponentový framework, byť z programátorského hlediska úžasný, platný, když by bylo uživatelské prostředí jím zprostředkované nepoužitelné pro reálnou práci uživatele. Použitím frameworku, který ovlivňuje uživatelské prostředí, se uživateli otevírá i zavírá množství možností.

1.2.1 Bonusy pro uživatele

Jakmile je v aplikaci použit framework, přebírá aplikace nejen tu funkcionalitu frameworku, kvůli které byl konkrétní framework zvolen ale i veškerou ostatní funkcionalitu. Uživateli jsou tedy nabízeny možnosti, které by mu v dané situaci konvenční programátor za svou mzdu nenabídnout nemohl. Můžeme tedy říci, že uživatelé profitují z použití frameworků ve formě bonusových možností uživatelského prostředí.

1.2.2 Sjednocení prostředí

Stejně jako framework sjednocuje kód programu, sjednocuje (pokud ho ovlivňuje) i uživatelské prostředí. Uživatel si pak rychleji zvykne na způsob ovládání aplikace a je schopen ovládat nové části programu bez složitějšího zaučování.

- **sjednocení v rámci aplikace:** ovládání aplikace je jednotné nezávisle

na části aplikace, která je ovládána, a programátorovi, který tuto část napsal

- **sjednocení mezi aplikacemi:** všechny aplikace využívající stejný framework mají shodné nebo podobné ovládání

Máme-li aplikaci s klasickým prostředím (seznam položek, přidání, úprava, odstranění položky), je nám sjednocení ovládání jednoznačně ku prospěchu. To se ale nedá říci o netypických aplikacích s unikátním prostředím. Programátoři se v tomto případě musí rozhodnout, jestli využijí možností frameworku a seskládají prostředí z prvků, které framework nabízí, nebo věnují vývoji více úsilí a pro danou část aplikace vyvinou vlastní prostředí. V prvním případě je však většinou uživatel nucen kombinovat ovládání jednotlivých prvků obdobným způsobem, jak tyto prvky kombinoval programátor.

1.2.3 Rychlost odezvy

Problematika rychlosti odezvy aplikace je úzce spojena s problematikou hardwarových nároků popsanou v kapitole 1.1.6. S vyššími nároky se při zachování stejného hardware zvyšuje doba odezvy aplikace a uživatel tak musí čekat na vykonání úkolů, které aplikaci zadal.

V případě webových aplikací je rychlost odezvy stanovena časovou náročností těchto úkonů:

tabulka č. 1 - **Faktory určující rychlost odezvy aplikace**

Úkon	Faktory ovliňující délku trvání
analýza kódu skriptovacího jazyka[4]	délka a složitost kódu aplikace a frameworku
interpretace kódu	počet použitých komponent frameworku, počet a typ přístupů k databázi, přístup k souborům, interakce s operačním systémem, dotazy na jiný server atd.
vygenerování HTML kódu	celková složitost stránky, počet použitých komponent frameworku, počet zobrazovaných záznamů seznamů atd.
odeslání stránky do prohlížeče	objem HTML kódu vygenerovaného v předchozím kroku, velikost souborů stylů, velikost JavaScriptových knihoven
zpracování a zobrazení stránky	velikost a složitost všech součástí stránky (HTML, CSS, JS)
odeslání požadavku na server	počet prvků formuláře a objem dat v nich uložených

Jak je vidět, tak faktorů, které ovlivňují uživatelovo pohodlí při práci s aplikací, je mnoho a u všech je potřeba určit kompromisní poměr mezi rychlostí odezvy a množstvím nabízených funkcí (ať jsou to funkce poskytované vývojáři na straně serveru nebo uživateli na straně prohlížeče).

4 Odpadá pro kompilované jazyky.

Vyrovnávací paměti

Kromě optimalizace kódu a redukce složitosti stránky je v některých případech pro zrychlení odezvy aplikace možno využít i různé vyrovnávací paměti. Ty v principu dobu odezvy zmenšují tím, že uchovávají již dříve vygenerovaný obsah a aktualizují ho jen tehdy, pokud je opravdu potřeba. Ve chvíli kdy si uživatel chce stránku zobrazit, vyrovnávací paměť mu může ihned poskytnout obsah, který má uložený.

V rámci aplikace

Aplikace sama si může generovaný obsah ukládat hned na více úrovních:

Mění-li se nějaká část stránky pouze občas, může si aplikace uložit HTML kód této části buď do databáze nebo přímo do souboru a při dotazu na stránku nemusí tento obsah znovu generovat.

Aplikace ale nemusí do vyrovnávací paměti ukládat pouze HTML kód, stejně tak lze do ní uložit i datové struktury v různých formátech. To je vhodné hlavně v případě, že máme data, pro jejichž získání je potřeba mnoho časově náročných úkonů, a která chceme na různých místech prezentovat ve více formách. Díky tomu, že je PHP interpretovaný jazyk, můžeme si dokonce do vyrovnávací paměti ukládat i přímo zdrojový kód.

V rámci frameworku

Dokonce i framework sám může obsahovat vyrovnávací paměť. Framework si může stránky ukládat podle URL dotazu a kontrolovat aktuálnost podle časové značky předávané aplikací. Pokud bude čas změny dat obsažených na stránce (určených časovou značkou předanou frameworku) menší než čas stránky uložené ve vyrovnávací paměti, není zřejmě potřeba stránku znovu generovat a framework může do prohlížeče odeslat stránku z vyrovnávací paměti.

Pokud je framework komponentový, může být vyrovnávací paměť použita i na úrovni komponent. V tom případě si pak framework ukládá pouze vygenerovaný obsah komponent a výsledný kód stránky je tedy skládán z kusů čerstvě vygenerovaného kódu a úseků uložených ve vyrovnávací paměti.

V rámci webového serveru

Některé moderní webové servery (např. *Apache*) dnes nabízejí možnost ukládání stránek do vyrovnávací paměti. Stránky jsou stejně jako v předchozím případě uloženy podle URL a jejich vypršení je určeno buď pomocí doby platnosti, nebo data modifikace zdrojových dat.

V rámci webového prohlížeče

V předchozích případech jsme ukládáním do vyrovnávací paměti šetřili čas potřebný k vygenerování stránky. V tomto případě k němu musíme ještě přičíst

čas strávený přenosem stránky do prohlížeče. Protokol HTTP totiž v sobě zahrnuje hlavičky určené pro řízení vyrovnávací paměti. Server může se svými stránkami odesílat hlavičku `Expires`, která určuje, kdy platnost stránky vyprší. Do této doby bude prohlížeč uživateli pod stejnou URL nabízet vždy stránku z vyrovnávací paměti.

Další možnost je používat hlavičky `Last-Modified` a `If-Modified-Since`. Prohlížeč se nejdříve dotáže na stránku s hlavičkou `If-Modified-Since` obsahující čas modifikace stránky ve vyrovnávací paměti. Server pak zjistí, jestli byla data od té doby modifikována a odešle buď HTTP status `304 Not Modified` nebo kompletní stránku včetně hlavičky `Last-Modified` s časem poslední modifikace.

1.3 Různé techniky, aneb na co máte chuť?

Používání frameworků by se možná dalo klasifikovat jako jedna z mnoha programovacích technik. Zároveň by ale na framework mohlo být nahlíženo jako na kombinaci několika dalších programovacích technik. V této podkapitole se pokusím popsat několik základních technik programování a ukázat jejich vztah k frameworku.

S tím jak se vyvíjely nové programovací techniky[5], začalo se těm méně pokročilým pejorativně přezdívat podle pokrmů. Tyto pokrmy mají podobnou vizuální strukturu jako kód psaný technikou, kterou pojmenovávají. Postupem času se tyto přezdívky ustálily a začaly se používat jako téměř odborné termíny. [6]

1.3.1 Špagetový kód

Tou nejprostší technikou (nepočítáme-li kód assembleru) je tzv. špagetový kód. Špagetový kód je málo strukturovaný kód psaný hlavně pomocí základních jazykových konstruktů typu: podmínka, návěští, jdi na návěští.

výpis kódu č. 1 - **Špagetový kód**

```
<?php
  $i = 0;
  navesti:
  $i++;
  echo $i." na druhou jsou ".($i*$i)."\n";
  if($i < 10) {
    goto navesti;
  }
  echo "Program ukončen";
?>
```

Špagetový kód připomíná špagety svou komplikovaností a nečitelností. Velmi těžko se zjišťuje, kudy vedou jednotlivá vlákna programu, stejně jako se těžko odhaduje, jak jsou propleteny jednotlivé špagety na talíři.

S rozvojem programovacích jazyků přibývají nové jazykové konstrukty a kód se stejným efektem se díky tomu dá zaznamenat v mnohem přehlednějším tvaru.

V důsledku tohoto vývoje se návěští přestávají používat a pojem špagetového kódu se díky tomu posouvá. Čím dál tím častěji se setkáváme se situacemi, kdy je pojmem špagetový kód označován i strukturovanější kód obsahující cykly a volání funkcí. Tím se pojem *„špagetový kód“* pomalu přibližuje pojmu *„procedurální programování“*.

výpis kódu č. 2 - **Strukturovaný kód**

```
<?php
  for($i = 1; $i <= 10; $i++) {
    echo $i." na druhou jsou ".($i*$i)."\n";
  }
  echo "Program ukončen";
?>
```

PHP podporuje návěští a příkaz `goto` až od verze 5.3, budeme-li proto v následujících kapitolách používat pojem špagetový kód, vždy ho budeme používat v jeho širším významu.

Psaní aplikací formou špagetového kódu rozhodně není ideální technikou. Nepovažujeme-li špagetový kód sám o sobě za zlozvyk, musíme alespoň souhlasit s tím, že způsobuje spoustu problémů popsaných v předchozí podkapitole:

- programátor je nucen opakovat množství kódu na více místech programu a je tak vystaven všem problémům, které toto opakování přináší
- s tím, že špagetový kód je velmi slabě strukturovaný se pojí špatná čitelnost kódu a obtížnost orientace v něm
- najít a opravit chybu ve špagetovém kódu je několikrát náročnější než to udělat v silněji strukturovaném kódu (jsou-li v kódu použity návěští, náročnost této činnosti se ještě několikanásobně zvyšuje)

Špagety PHP chutnají nejlépe s omáčkou

Mluví-li se o špagetovém kódu v souvislosti s PHP nebo jiným skriptovacím jazykem pro generování HTML kódu, bývá orientace v tomto kódu navíc složitější ještě díky přítomnosti HTML značek[7] vložených mezi PHP kód. Na HTML značky se můžeme koukat jako na omáčku, která vyplňuje prostor mezi špagetami, částečně je zakrývá a činí tak orientaci v kódu ještě náročnější.

výpis kódu č. 3 - **Špagetový PHP kód**

```
<?php
  for($i = 1; $i <= 10; $i++) {
?>
  <p><?php echo $i; ?> na druhou jsou <?php echo ($i*$i); ?></p>
<?php
  }
?>
  <p style="font-weight: bold;">konec programu</p>
```

PHP bylo ve svých počátcích navrženo jako jednoduchý skriptovací jazyk obsahující pouze základní jazykové struktury a o objektově orientovaném programování jsme si mohli nechat pouze zdát. Jak se postupem času PHP

rozvíjelo, přibývala nová rozšíření a počet funkcí se zvětšoval. PHP začalo být díky jednoduché konfiguraci a absenci nutnosti kompilace kódu používáno v mnohem složitějších aplikacích a kód se díky omezenému množství jazykových struktur stával nesrozumitelným. Proto byla do PHP zakomponována podpora objektového programování (od verze 5 je podpora téměř srovnatelná s primárně objektově orientovanými jazyky).

Podpora objektového programování ale byla do PHP vnesena pouze jako rozšíření a PHP z důvodu zpětné kompatibility stále umožňuje tvořit aplikace na úrovni jednoduchého skriptovacího jazyka. Tato skutečnost umožňuje vznik hybridních aplikacích blíže popsaných v podkapitole Špagety s masovými kuličkami:

- aplikace programátorů, kteří jsou zvyklí psát špagetovým kódem a použijí komponenty nebo části kódu vytvořené zkušenějšími vývojáři, využívajícími objektové programování
- dlouhodobě vyvíjené aplikace procházející změnami programovacích technik z důvodu vývoje programovacího jazyka nebo programátorových schopností

Špagety s masovými kuličkami

Pojem „spaghetti with meat balls“ hanlivě označuje slabě objektově orientované programování. Masové kuličky reprezentují objekty a špagety vyjadřují málo strukturovanou programovou logiku, která tyto objekty spojuje. Kód skrývající se pod tímto termínem píší hlavně programátoři, kteří začínají s objektovým programováním (hlavně samouci) a jsou zvyklí programovat procedurálně. Takový kód je plný globálních proměnných, procedurálního kódu na nejvyšší úrovni (nezanořeného do žádné funkce ani metody) a sem tam volání metody nějakého objektu.

Špagety a komponentový framework

Špagetový kód je struktura, jejíž tvorbu se většina frameworků snaží minimalizovat tím, že velkou většinu programové logiky zapouzdřuje do svého jádra nebo přídavných komponent. Nikdy ale není možné tuto strukturu úplně eliminovat, protože špagetový kód ve svém širším smyslu je základním stavebním kamenem každé aplikace. I aplikace, která používá množství tříd, metod a funkcí, nemůže fungovat bez kódu obsaženého v těchto strukturách, a tento kód není nic jiného než špagetový kód. Záleží tedy na jakési úrovni přiblížení, kterou se na danou aplikaci díváme. V menších úrovních přiblížení se může jevit silně strukturovaná, jakmile se ale ponoříme hlouběji, zjistíme, že ve skutečnosti je řízena drobnými úseky málo strukturovaného kódu.

Dalším místem programu, ve kterém se kód může podobat špagetovému kódu, je nejvyšší úroveň. Tedy kód nezanořený do žádné funkce ani metody. Kód, který píše programátor tvořící koncovou aplikaci (uživatel frameworku). Převážně komponentové webové frameworky nechávají programátora definovat stránky a formuláře postupně přímo na úrovni souboru. Vytvoří se objekty zapouzdřující

jednotlivé komponenty, nastaví se jim vlastnosti, případně se zavolají funkce s určujícími parametry, které nadefinování komponent zařídí, a poté jsou komponenty voláním funkcí nebo metod aktivovány.

1.3.2 Raviolový kód

Ravioly připomínají svou strukturou (jsou to těstovinové kapsičky plněné masem nebo sýrem) použití relativně nezávislých komponent. Každá raviola představuje jednu komponentu, přičemž obsahuje náplň v podobě programového kódu a dat. V ideálním případě se tedy jedná o objekty zapouzdřující veškerou funkčnost těchto komponent a napojujících se na ostatní komponenty prostřednictvím pevně daného rozhraní.

Ravioly byly jako pojmenování tohoto způsobu programování zvoleny, kvůli jejich příbuznosti se špagetami. Na rozdíl od špagetového kódu ale tento termín není pejorativní. Technika skládání aplikací z komponent je v současné době velmi rozšířená a je většinou programátorů považována za srozumitelnou, jednoduchou a přesto velmi flexibilní.

Raviolový kód je pro komponentový framework ze své podstaty asi nejdůležitější programovou strukturou. Rámcově vystihuje vnitřní architekturu frameworku jako volitelný soubor volně pospojovaných relativně samostatných komponent. Pokud bychom chtěli komponentový framework zařadit do jedné z technik nastíněných v této kapitole, určitě by to byl raviolový kód.

1.3.3 Lasagnový kód

Po přečtení předchozích podkapitol si jistě odvodíte, jak zhruba vypadá architektura aplikací psaných lasagnovým kódem. Základními stavebními kameny lasagnového kódu jsou vrstvy. Každá vrstva je relativně nezávislá na ostatních a její funkce je ostatními řízena pouze přes pevně a přesně definované rozhraní[5]. Použití rozhraní skýtá několik výhod:

- ostatní vrstvy vždy vědí jak s vrstvou komunikovat
- možnosti jak ovlivnit chod vrstvy jsou omezené a díky tomu je menší šance, že se vrstva dostane do výjimečného nepřípustného stavu
- rozhraní zůstává stejné i když je vrstva upravena; ostatní vrstvy tak mohou využívat různé verze aniž by se musely přizpůsobovat; tato skutečnost umožňuje i snadnější práci v týmu, kdy se o vývoj jednotlivých vrstev starají jiní programátoři
- abstraktně definované rozhraní umožňuje k vrstvě přistupovat bez vědomosti toho, které konkrétní technologie využívá (více v kapitole 1.1.5)

Jestliže raviolový kód vyjadřuje vnitřní strukturu komponentového frameworku, lasagnový kód je pak vnějším pohledem na jakýkoli framework. Tento pohled

5 Nemusí se jednat přímo o rozhraní ve smyslu jazykového konstruktu `interface`, může jít například o abstraktní třídu nebo soubor objektů, funkcí a jiných struktur. Vždy ale musí být jednoznačně definované a zdokumentované.

můžeme ještě rozdělit podle úrovní přiblížení s jakou na framework nahlížíme. Úplně zvenčí můžeme vnímat celý framework jako jednu odstíněnou vrstvu, se kterou programátor komunikuje přes pevně dané rozhraní (třídy, metody, funkce atd.). Když se ale podíváme trochu blíže, zjistíme, že framework je sám o sobě souborem vrstev, z nichž některé využívá pouze framework sám, a některé mají svá rozhraní zpřístupněné přímo programátorovi (databázová vrstva, výstupní vrstva specifická pro koncové zařízení atd.).

1.3.4 Framework

Jak již bylo nastíněno v úvodu této kapitoly: s jistým odstupem můžeme používání frameworku vnímat jako samostatnou techniku psaní aplikací, pokud se ale podíváme pozorněji, zjistíme, že framework sám o sobě je souborem struktur vytvořených různými technikami. Pro snadnější orientaci si v přehledné tabulce shrneme jednotlivé techniky zmíněné v této kapitole a připomeneme si jejich klady zápory i vztah k frameworkům.

tabulka č. 2 - **Srovnání programovacích technik**

Technika	Klady	Zápory	Vztah k frameworku
Špagetový kód	• málo jazykových konstruktů	• nečitelnost • obtížná správa • opakování kódu	• snaha o eliminaci • vnitřní kód metod a funkcí
+ masové kuličky	• snaha o základní využití objektů	• přidáním strukturovaných prvků se čitelnost může ještě zhoršit	• kód programu využívajícího framework
Raviolový kód	• většinou plně OOP[6] • snazší čitelnost • lépe se spravuje • umožňuje práci v týmu • znovupoužitelné	• komponenty jsou pro vývojáře černé schránky • jsou vyžadována zdokumentovaná rozhraní komponent	• vnitřní struktura frameworku složeného z komponent
Lasagnový kód	• pevné rozdělení odpovědností jednotlivých vrstev • bezchybnost samostatných vrstev • lepší práce v týmu • možnost tvorby abstraktních vrstev • znovupoužitelné	• jsou vyžadována zdokumentovaná rozhraní vrstev • ovládání skrze rozhraní je omezené kvůli potřebnému zobecnění • vrstvy většinou musí počítat se množstvím zbytečných stavů programu	• framework jako jedna vrstva aplikace • vrstvy ovládající různé části aplikace • abstraktní vrstvy zprostředkovávající programátorům využití různých technologií

6 Objektově orientované programování.

1.4 Shrnutí

V předchozích podkapitolách jsem si ukázali mnohé výhody i nevýhody použití frameworku, provedli srovnání s ostatními programovacími technikami a ukázali si, jaký vztah mají tyto techniky k frameworku jako celku. Pro náplň shrnutí nám tedy zbývá už jen ukázat, ve kterých případech je vhodné framework použít a ve kterých nikoli. Pro hledání hranice mezi těmito dvěma variantami si vezmeme inspiraci v ekonomice. Budeme totiž hledat rovnovážný bod užitkové a nákladové funkce.

Užitek

Užitek je měřen spíše subjektivními veličinami (chybovost, univerzálnost, flexibilita) a roste s:

- opakovaným vykonáváním stejných nebo podobných úkonů
- rostoucí rozmanitostí očekávaných koncových zařízení
- s rostoucí potřebou distribuovat aplikaci na více hostingových serverů
- s rostoucím počtem vývojářů

Výše definovaný užitek je nepochybně závislý na rozsahu aplikace, který může být určen například počtem stránek, formulářů, výpisů záznamů atd. Užitek ale můžeme ještě vynásobit počtem aplikací jako celků, které společnost hodlá vytvářet. Je jasné, že pokud se v budoucnosti společnost rozhodne vytvořit obdobnou aplikaci bude přínos frameworku také obdobný jako v případě první aplikace.

Náklady

Náklady jsou měřeny objektivními veličinami (čas potřebný pro vývoj, finanční náklady) a jsou určeny těmito faktory:

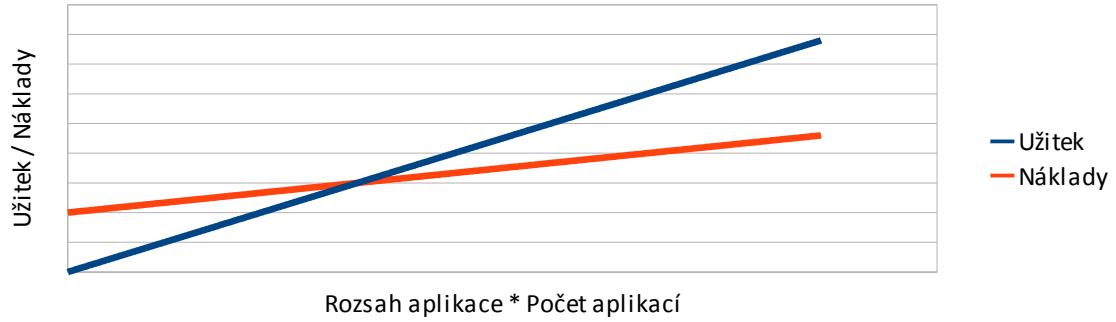
- pořízení frameworku
- nastavení parametrů
- vytvoření personifikovaných doplňků (šablony, styly, přídavné funkce)
- nákup hardware
- zaškolení vývojářů

Nakupuje-li společnost externí framework, rozumíme pod pořízením frameworku jeho pořizovací cenu, v opačném případě jde o čas strávený vývojem vlastního řešení. V obou variantách je náklad slabě závislý na rozsahu aplikace. V prvním případě bude framework tím dražší, čím více funkcí od něj požadujeme a v druhém případě nám zase vývoj zabere více času. Závislost ostatních faktorů na rozsahu aplikace je téměř zanedbatelná. Nákladová funkce bude tedy evidentně mnohem méně strmá než užitková.

Bod rovnováhy

Na následujícím grafu vidíte, že s rostoucím rozsahem začíná užitek převažovat náklady a framework se tak stává výhodnou investicí. Bod rovnováhy daný průsečíkem obou funkcí určuje hraniční rozsah aplikace, pro který je již vhodné využít framework jako základ pro vaši aplikaci.

diagram č. 1 - **Bod rovnováhy užitku a nákladů**



Jelikož při hledání bodu rovnováhy hrají velikou roli subjektivní veličiny a rozsah aplikace se předem špatně odhaduje, nelze pro nalezení bodu rovnováhy použít matematické vzorce. Každá společnost se rozhoduje sama na základě svých odhadů. Užitková funkce je ale podle mě natolik strmá, že pokud společnost vyvíjí webové aplikace častěji, nebo to je dokonce její hlavní náplň, měla by nějakou podobu frameworku pro své aplikace používat.

Kapitolu uzavřeme konstatováním, že čím rozsáhlejší je aplikace, kterou tvoříme, a čím více jich tvoříme, tím víc přichází v úvahu použití frameworku jako podpory pro její tvorbu.

kapitola 2:
Srovnání existujících frameworků

V této kapitole hodlám srovnat existující frameworky pro tvorbu webových aplikací v PHP. Porovnám jejich přínosy a pokusím se shrnout jejich nedostatky tak, abych se z nich mohl poučit při návrhu nového frameworku.

2.1 Porovnání frameworků

Abych obhájil své rozhodnutí pro tvorbu nového frameworku, když jich je, dalo by se říci, plný internet, pokusím se sestavit omezený seznam dostupných frameworků, u kterých budu hledat kladné i záporné stránky. Dopředu upozorňuji, že frameworky budu hodnotit podle svého názoru, a že vlastnost označená mnou za zápornou se může někomu jevit spíše jako kladná a naopak.

2.2 Srovnávané frameworky

Do srovnání jsem vybral PHP frameworky z několika skupin. První skupina obsahuje celosvětově rozšířené frameworky *Symfony*[8] a *Zend Framework*[9]. Druhá skupina je reprezentována pouze jedním zástupce a to frameworkem *PRADO*[10], kterému se povedlo do poměrně ustálených vod PHP frameworků vnést nové principy. Třetí skupinu tvoří frameworky *Yii*[11] a *CodeIgniter*[12], které sice nejsou rozšířené tolik, jako projekty z první skupiny, ale přesto jsou něčím zajímavé. Výběr frameworků jsem zakončil jedním povedeným českým zástupcem a to *Nette*[13] frameworkem, okolo kterého se v poslední době tvoří větší a větší komunita.

2.2.1 Symfony

Symfony je zajímavý open-source projekt, který je na trhu PHP frameworků již od roku 2005 a dosud je aktivně vyvíjen. Projekt je sponzorován francouzskou společností Sensio Labs a v současnosti je podle informací na oficiálních stránkách využíván ve více než 1000 aplikacích po celém světě. Stejně jako všechny ostatní frameworky, které budeme jmenovat, i Symfony dodržuje princip návrhového vzoru MVC[7].

Hned na začátek musím podotknout, že Symfony je o něco více než jen soubor PHP tříd, urychlujících vývoj aplikace. Pokud se rozhodnete využívat framework Symfony, budete si muset zvyknout na to, že více než kód PHP uvidíte soubory XML případně YAML, v nichž budete definovat struktury formulářů, databáze a celých modulů aplikací. Symfony totiž pro plné využití jeho možností vyžaduje interakci s řadou nástrojů určených pro generování PHP kódu, databází apod. Vzhledem k tomu, že tato interakce probíhá pomocí příkazové řádky, bude mnohem pohodlnější vyvíjet aplikaci v Symfony na počítači s operačním systémem Linux.

Právě díky těmto nástrojům je Symfony ideální pro rychlé generování back-endů webových stránek, kde nijak nezáleží na designu výsledné stránky. Programátor jen nadefinuje vrstvu Model a Controller i View nechá více méně vygenerovat automaticky. Framework tak vlastně automatizuje programování podle návrhového vzoru CRUD[8]. Symfony se se svou robustností hodí především pro

7 Model-View-Controller – rozdělení aplikací do tří relativně nezávislých vrstev (více viz kapitola 3.5.2)

8 Create-Read-Update-Delete – umožňuje vytvořit, zobrazit, upravit a smazat záznamy

větší projekty s rozsáhlým administračním rozhraním. Skutečnost, že nasazení frameworku a zvyknutí si na jeho ovládání přes příkazovou řádku, zabere nemálo času, ještě podtrhuje požadavek na rozsah projektu, který má být vyvíjen.

Jako poslední poznámku k tomuto frameworku bych dodal, že aplikace v něm psané nejsou plně událostmi řízené. Slovo „plně“ jsem použil proto, že programování pomocí frameworku není ani procedurálně řízené. Framework totiž stejně jako většina ostatních nabádá programátory, aby aplikační logiku soustředili do předem připravených metod jednotlivých Controllerů. Tyto metody jsou nazývány akce a jsou volány v případě, že uživatel požaduje jejich provedení pomocí kliknutí na odkaz apod.

2.2.2 Zend Framework

Již to, že open-source Zend Framework je spravovaný společností Zend Technologies, která se stará o vývoj samotného PHP interpreteru, dává jistou záruku kvality. Když k tomu připočteme, že od roku 2006, kdy byl framework poprvé uvolněn, nasbíral více než 500 vývojářů, kteří se aktivně podílejí na jeho vývoji, nemůžeme se divit, že čím dál více velkých firem od svých nových rekrutů požaduje znalost právě tohoto frameworku.

Zend Framework je striktně objektový soubor volně propojených komponent, z nichž ty nejjednodušší umožňují formátování čísel, nebo kontrolu správnosti URI adresy a ty nejpokročilejší implementují návrhový vzor MVC. Každý komponentový framework se pyšní tím, že jejich uživatelé mohou použít jen ty komponenty, které opravdu potřebují, ale dle mého názoru žádný z nich nemůže svému závazku dostát tak, jako Zend Framework.

Framework implementuje vzor MVC obdobně jako Symfony, za každou stránkou viditelnou pro uživatele, jsou tedy skryty 3 vrstvy a ve většině případů je každá z těchto vrstev v samostatném souboru. Oproti Symfony ale nabízí zajímavý bonus ve formě komponenty `Zend_Form_Decorator`, která se stará o generování HTML kódu ovládacích prvků formuláře. Vytvořením odvozené komponenty totiž můžeme generovaný kód zcela přizpůsobit našim potřebám.

Obrovskou výhodou Zend Frameworku je množství komponent, které nabízí. Už několikrát se mi stalo, že při hledání pomoci s implementací pokročilé funkce mi vyhledávač na prvních místech nabídl přímo stránku s popisem odpovídající komponenty Zend Frameworku. Zend Framework obsahuje nejen drobné komponenty a rámce pro dodržování návrhových vzorů ale i spoustu konkrétních řešení jako implementace napojení na nejrůznější webové služby a API populárních webových aplikací.

2.2.3 PRADO

Jako velký příznivec a několikaletý vývojář v Delphi, jsem si hned po prvním pohledu na PRADO framework všiml, že tyto dva nástroje mají nemálo společného. PRADO je čistě komponentový framework, který jako jediný z frameworků zde jmenovaných, není striktně založený na klasické MVC

architektuře. Stejně jako se v Delphi každý formulář skládá ze dvou souborů (`.dfm` a `.pas`)[14], tak i každá stránka v PRADU je zastoupena souborem pro definici vzhledu stránky (`.page`) a samotným PHP souborem obsluhujícím události generované komponentami. Na rozdíl od klasické MVC architektury, je ale v souboru `.page` definován nejen vzhled stránky a komponent, ale i chování těchto komponent.

PRADO je totiž plně událostmi ovládaný. Na rozdíl od ostatních frameworků, kde programátor soustřeďuje do obsluhy několika metod předdefinovaných v rodičovských třídách, PRADO nabízí množství událostí přímo u jednotlivých komponent. Programátor pak může velmi jednoduše v definičním souboru navázat libovolnou metodu na libovolnou událost libovolné komponenty.

Protože mě tento framework opravdu zaujal svou netradiční strukturou, dovolím si předvést ukázku jednoduché stránky:

výpis kódu č. 4 - **Ukázka souboru .page frameworku PRADO**

```
<com:TContent ID="body">

<h1>Textové pole a tlačítko</h1>

<div>
<com:TTextBox ID="TextBox" />
<com:TButton Text="Odeslat" OnClick="buttonClicked" />
</div>

</com:TContent>
```

Soubory pro definici stránky (`.page`) jsou evidentně XML soubory, kde se míchá (X)HTML kód s kódem pro definici komponent. Tento kód se zapisuje do definičního souboru pod vlastním XML name-space. Všimněte si atributu `OnClick` u komponenty `TButton`. Tento atribut určuje, jaká metoda bude volána po kliknutí na tlačítko „Odeslat“.

výpis kódu č. 5 - **Ukázka souboru .php frameworku PRADO**

```
<?php

class Stranka extends TPage
{
  public function buttonClicked($sender,$param)
  {
    $sender->Text = "Kliknul jsi na mě";
  }
}

?>
```

Soubor pro obsluhu událostí obsahuje pouze definici třídy odvozené od abstraktní třídy `TPage`. Třída obsluhuje jedinou metodu a tou je metoda obsluhující kliknutí na tlačítko „Odeslat“.

PRADO framework se vám nemusí líbit, ale musíte se mnou souhlasit, že se diametrálně odlišuje od všech ostatních zde zmiňovaných. První verze frameworku byla uvolněna v roce 2004 a do roku 2006, kdy byla vydána verze 3.0 procházel intenzivním vývojem. Po nahlédnutí do webového nástroje pro hlášení

chyb bych řekl, že je projekt dosud aktivně vyvíjen, o tom ale bohužel nesvědčí datum vydání poslední verze 3.1.7. Ta byla totiž uvolněna již v únoru roku 2010. Stagnace projektu může být také dána tím, že se většina vývojářů přesunula na nový projekt, „pokračovatele“ frameworku PRADO, framework Yii.

2.2.4 Yii

Yii je nováčkem na poli frameworků, který vznikl v roce 2008. I přes své mládí se ale především díky vztahu ke staršímu bratrovi PRADO dokázal dostat do popředí zájmu a nalákat spoustu příznivců. Yii má sice být neoficiální nástupce PRADO, odstraňující nedostatky ohledně jeho výkonu a malé přizpůsobitelnosti, po letmém prozkoumání a pročtení části dokumentace bych ale mezi těmito frameworky žádný vztah nenašel.

Yii podlehlo tlaku okolí a zavedlo plnou podporu MVC architektury se všemi výhodami i nevýhodami. Přesto si Yii zachovává komponentovou strukturu a umožňuje použití událostí, i když už ne pro zpracování esenciálních úkonů jako obsluha kliknutí na tlačítka.

Stejně jako Symfony obsahuje i Yii nástroje pro generování kódu ovládané přes příkazovou řádku. Kromě těchto nástrojů, ale od verze 1.1 nabízí i modul Gii ovládaný přes webové rozhraní, který generování Modelů, Controllerů a dalších součástí aplikací výrazně zjednodušuje. Nástroj pro generování CRUD modulů jde natolik daleko, že po správné definici Modelu vygeneruje kompletní systém pro zadávání, prohlížení, úpravu i mazání záznamů, které je možné dodatečně přizpůsobit podle libosti.

2.2.5 CodeIgniter

CodeIgniter se na svých oficiálních stránkách pyšní tím, že je jednoduchý Nenutí programátory dodržovat striktní, někdy omezující principy programování, a ve výsledné aplikaci zanechává pouze miniaturní stopu. Po zběžném prozkoumání frameworku a osahání práce s ním, musím souhlasit, že na programátory neklade žádné velké nároky.

Jádro frameworku totiž tvoří pouze implementace klasické MVC architektury a víceméně všechna další přidaná hodnota frameworku je rozdělena do samostatně použitelných knihoven, zvaných *helper*. Vrstva View je složena z HTML šablon doplněných sem tam o nějaký ten PHP kód, vypisující hodnoty předané Controllerem případně využívající nějakého *helperu*. CodeIgniter se tak může určitě stát výborným pomocníkem programátorů-designerů, kteří s PHP nemají příliš zkušeností a hlavní náplní jejich práce je spíš tvorba HTML kódu. Případně by se CodeIgniter mohl hodit pro malý tým složený z PHP vývojáře a HTML designera.

2.2.6 Nette

Nette je jediný český zástupce v našem výběru, ale musím říci, že se nijak nebojí

velkých zahraničních konkurentů. Vzhledem k tomu, že komunitu, která se okolo něj sdružuje nelze ani srovnávat s komunitami okolo gigantických projektů jako Zend Framework, nemůže Nette konkurovat rozsahem. Proto se soustředí na implementaci základních prvků co nejelegantnějším způsobem. Framework je postaven na architektuře MVC, kterou drobně modifikuje zaměněním pojmu Controller za Presenter.

Nette má čistou a dobře navrženou objektovou strukturu, která využívá k definici rozhraní tříd struktury typu `interface` i přes to, že PHP jako interpretovaný jazyk je dostatečně měkký, aby se bez těchto struktur obešel. Na první pohled je vidět, že Nette není vytvořeno a dokonce ani určeno pro programátory začátečníky nebo „PHP skriptaře“, pro které je PHP nejstriktnější jazyk, který znají.

Nette používá abstraktní databázovou dibi[15], která je vyvíjena jako samostatný nástroj stejnou komunitou jako samotný framework. Dibi je stejně jako Nette jednoduchým a čistým nástrojem, který není obalen nadbytečnými funkcemi, ale zato je spolehlivý a dbalý standardů.

Framework Nette se pyšní vysokou úrovní bezpečnosti aplikací, kterou zajišťuje právě omezeným množstvím do detailu promyšlených funkcí a tříd. Kromě databázové vrstvy, která se sama stará o správné formátování a ošetřování hodnot proti útokům, je zde promyšlený systém šablon, který umožňuje tzv. kontextově citlivé ošetřování hodnot. V praxi to znamená, že pokud programátor vypisuje hodnotu přímo do viditelného elementu, je ta ošetřena pomocí funkce `htmlspecialchars()`, a pokud vkládá hodnotu do JavaScriptu, je převedena do nativní formy pro JavaScript (text, číslo, pole atd.).

2.3 Hodnocení

2.3.1 Technické aspekty

Architektura

Všechny zde jmenované frameworky až na PRADO spojuje jeden prvek, MVC architektura ve své nejstriktnější formě, tedy nutnost implementace všech tří vrstev jako samostatných celků aplikace. Evidentně se architektura MVC natolik rozšířila a standardizovala, že už ani sami inovativní tvůrci frameworku PRADO ve svém novém projektu Yii nedokázali vymyslet a implementovat jinou strategii. Vzhledem k tomu, že místo pokusu o byť drobné pozměnění této architektury, se frameworky zaměřují na nabídku nástrojů, které ulehčují generování jednotlivých vrstev, usuzuji, že architektura MVC je navržena téměř dokonale a jediný její neduh je větší objem kódu, který je potřeba napsat.

Nezávislost na platformě

Když se podíváte na hodnocené frameworky, zjistíte, že žádný z nich explicitně

nepodporuje odlišné zobrazení aplikací na mobilních zařízeních. Nezbývá mi než zeptat se proč, když MVC architektura k tomu přímo vybízí. Zaráží mě, že jsem nenarazil ani na podporu vícenásobných View pro odlišná koncová zařízení. Je pravda, že u frameworků, kde programátor v rámci Controlleru explicitně vybírá odpovídající View, může být tato funkce implementována poměrně snadno. Programátor jen musí zjistit, zda je stránka prohlížena mobilním zařízením či nikoliv, a načíst odpovídající View. Stejně ale nechápu, proč není výběr vhodné verze View zapracován přímo do frameworku.

Doposud jsme jako o přizpůsobení výstupu konkrétnímu koncovému zařízení mluvili pouze jako o výběru vhodné šablony. Vzhledem k tomu, že frameworky obsahují množství komponent a widgetů, které generují svůj vlastní HTML kód, jenž je následně do šablony dosazen, bude realizace nezávislosti na koncovém zařízení ještě o něco složitější. Neprošel jsem dokumentaci frameworků do posledního slova, takže následující tvrzení je spíše spekulací, ale dle mého názoru žádný framework kromě Zend Frameworku neumožňuje změnit kód generovaný integrovanými komponentami bez zásahu přímo do kódu frameworku. Bez možnosti změnit generovaný kód můžeme šablonu ohýbat jak chceme, ale stránku přizpůsobenou pro mobilní zařízení z ní nedostaneme.

Události

Většina zmíněných frameworků umožňuje použít události, ale prakticky pouze PRADO, český zástupce Nette a do jisté míry i Yii jsou opravdu událostmi řízené. Možná mi nedáte za pravdu, ale dle mého názoru nejsou události jen přežitkem z desktopových aplikací. Vykonávání akcí v rámci obsluhy zajišťuje, že bude akce provedena ve chvíli, kdy k ní opravdu má dojít. Události minimalizují počet používaných podmínek, kód se tak stává přehlednějším a přitom je zachována ne-li přímo rozšířena flexibilita aplikace. Navíc se díky událostem dá modifikovat chování komponent bez toho, aby bylo zasahováno přímo do jejich kódu. Umožňuje-li některá komponenta více variant svého chování a není-li správná varianta známa dopředu, není nic jednoduššího, než v případě potřeby vyvolat událost v jejíž obsluze se preferovaná varianta zvolí. Komponenty se tak mohou chovat standardně do doby, než programátor obslouží danou událost a chování změní.

Tíha frameworku

Tento titulek se snaží uvést podkapitolu srovnávající hodnocené frameworky podle toho, jestli jsou to jednoduché, snadno nasaditelné frameworky (*light-weight*), nebo robustní strukturované soubory všemožných funkcí (*heavy-weight*). Jednoznačně nejtěžším hodnoceným frameworkem je Zend Framework, který obsahuje obrovské množství komponent, tříd a funkcí. Abyste tento fakt poznali, nemusíte dokonce ani procházet kód nebo dokumentaci frameworku, stačí si pouze stáhnout jeho aktuální distribuci a podívat se, jakou velikost má stahovaný archiv. Ten má totiž pro kompletní verzi frameworku více než 25 MB. Díky tomu, že je struktura Zend Frameworku striktně komponentová,

budete ale vždy využívat a do operační paměti načítat jen ty komponenty, které opravdu použijete. Bez této vlastnosti by Zend Framework nebyl pro reálné nasazení vůbec použitelný. Na druhé straně stojí drobné frameworky jako Nette, CodeIgniter a Yii, jejichž distribuční balíčky mají něco přes 2 MB.

Zvolit vítěze v této kategorii není snadné, protože vždy závisí na daném projektu. Budete-li chtít vytvořit malou aplikaci o několika formulářích bez napojení na servery třetích stran, zvolíte drobný framework. Pokud budete vyvíjet mamutí projekt propojený s několika webovými službami, exportující a importující z nejrůznějších formátů a očekávat, že se na něm bude podílet 30 vývojářů, zvolíte rozšířený a robustní framework typu Zend Framework nebo Symfony.

2.3.2 Netechnické aspekty

Licence a cena

Vzhledem k tomu, že PHP zpracovává nekompilovaný zdrojový kód, by bylo pro výrobce frameworků velmi těžké uhlídat zneužití kódu nebo modifikaci kódu jejich díla. Možná i proto jsou všechny hodnocené frameworky šířeny jako open-source software. Přímé náklady pořízení frameworku jsou díky tomu nulové.

Vývoj

Z hlediska množství vývojářů podílejících se na tvorbě frameworku je jednoznačným vítězem Zend Framework. Je pravděpodobné, že proto budou případné chyby ve frameworku odstraňovány rychleji než u ostatních. Dále se dá předpokládat i častější přísun nových komponent implementujících nové funkce. Méně časté naopak budou zásahy do jádra frameworku, protože takový zásah by pravděpodobně ovlivnil velkou část participující vývojářů a jejich komponent. Otázkou zůstává, zda je tato skutečnost pro koncového programátora výhodou, či naopak. Ustálený a funkční framework může být výhodnou v případě souběžné práce na více projektech započatých v jiném časovém okamžiku. Důležité ale je, aby se vývoj nezastavil úplně, neboť pak by byl čas vložený do seznámení s frameworkem poměrně krátkodobou investicí. Z tohoto důvodu bych se obával využít pro svůj vývoj PRADO framework, jehož vývoj dle mého názoru již delší dobu stagnuje.

Dokumentace

Široká vývojářská základna nemusí být zárukou kvalitní dokumentace. S rostoucím počtem vývojářů a jimi vyvíjených komponent se již dokumentace nedá tvořit centrálně a i o tuto činnost se pak musí starat jednotliví vývojáři. Záleží pak jen na nich a na vedení komunity, jestli dokumentaci zpracují kvalitně nebo ne. Dle mého názoru byla ke všem hodnoceným frameworkům dostupná poměrně detailní dokumentace, i když právě u Zend Frameworku nebyla API dokumentace, popisující framework až do úrovně metod a vlastností, kompletní.

Tato dokumentace je většinou generovaná přímo z kódu a někteří přispívající vývojáři ji evidentně do kódu nevložili.

Podpora

Na rozdíl od dvou předchozích faktorů nezávisí dostupná podpora u open-source projektů ani tak na vývojářích, jako spíš na komunitě uživatelů, tedy programátorů pracujících s frameworkem. Většina šikovných programátorů se stává součástí komunity kolem frameworku a ráda poskytuje rady svým méně zkušeným kolegům ať už v podobě tutoriálů nebo odpovědí na nejrůznější dotazy. Řekl bych, že v tomto aspektu znovu vítězí Zend Framework, díky jeho rozšíření po celém světě.

2.3.3 Shrnutí

Vybrat jednoho vítěze je těžké, ale pokud bych se opravdu měl rozhodnout pro jeden z hodnocených frameworků, asi bych vzhledem k rozsahu mých současných projektů (vesměs drobné projekty), zvolil Yii framework. U Yii mě zaujala především komponentová struktura a skutečnost, že jej vyvíjí stejní lidé jako PRADO framework, který, ač má svá omezení, mě prostě zaujal.

Pokud bych měl myslet ve velkém a počítat s tím, že bych se teoreticky nechal zaměstnat ve větší firmě, začal bych se učit pracovat se Zend Frameworkem, protože ten je díky své rozšířenosti a rozsáhlosti stále nejsilnějším koněm na poli PHP frameworků.

Závěrem bych chtěl ještě pochválit Nette framework, který mě zajímá více a více s každým přečteným řádkem jeho dokumentace.

2.4 Motivace

V této kapitole bych chtěl stručně shrnout jaké faktory mě vedly k myšlence vytvořit si vlastní framework. Dopředu bych chtěl podotknout, že jsem za dobu, kterou vyvíjím v PHP, vytvořil již dvě verze „frameworků“, z nichž oba jsou v současnosti použité v reálně nasazených aplikacích. Ani jeden z těchto „frameworků“ sice ani zdaleka nedosahoval rozsahu ani kvalit ROAD frameworku, ale i tak jsem měl dopředu alespoň matnou představu o tom, jak dlouhou dobu mi asi vývoj frameworku zabere.

Nedostatky u existujících frameworků

V současné době se čím dál více rozvíjí trh s mobilními zařízeními s přístupem k internetu, počínaje mobilními telefony a konče kapesními tablety. S tím jak roste množství prodaných mobilních zařízení se zvyšuje i počet potenciálních uživatelů webových aplikací přistupujících k těmto aplikacím prostřednictvím svých mobilních zařízení. Proto se mi zdá důležité nabídnout těmto uživatelům alespoň srovnatelný komfort, jaký mají uživatelé přistupující přes stolní počítač

nebo notebook. Vzhledem k tomu, že jsem u žádného z hodnocených frameworků tuto možnost nenašel, stala se tato potřeba jedním z důvodů pro vytvoření nového frameworku.

Z mého pohledu byla dalším drobným nedostatkem skutečnost, že některé frameworky plně nevyužily potenciál událostí jako nástroje pro interakci uživatele s aplikací. Posledním faktorem, který mě alespoň ze začátku odrazoval, byl fakt, že veškerý provoz je směrován do souboru `index.php`, kde teprve probíhá routování do jednotlivých Controllerů. Osobně mám radši přímý přístup k jednotlivým souborům (viz 3.3). Vzhledem k tomu, že tímto způsobem fungovaly všechny frameworky, nemohu ale tuto skutečnost brát jako nedostatek.

Pohodlí

V rámci své bakalářské práce jsem vytvořil prototyp drag&drop vývojového prostředí (IDE[9]) pro tvorbu aplikací v PHP, který dosud pro některé své projekty používám. Toto IDE využívalo na serveru běžící „framework“, který jsem za tímto účelem vyvinul. Jak už jsem zmínil v úvodu kapitoly, nebyl tento framework příliš propracovaný, ale i přesto byla tvorba aplikací v drag&drop vývojovém prostředí velmi pohodlná. Rozhlížel jsem se po ostatních frameworcích a hledal nějaký, který bych mohl do svého drag&drop IDE integrovat. Jelikož, kromě frameworku PRADO, který má zase jiné nedostatky, nebyl žádný framework vhodný, přibyl mi další důvod, proč implementovat nový framework.

Flexibilita

Ať jsou frameworky třetích stran sebe lépe konfigurovatelné, vždy nakonec narazíme na nějaké omezení, které nelze bez zásahu do jádra frameworku překonat. I kdyby byl programátor s frameworkem natolik dobře seznámený, že by byl schopen zásah do jádra provést, při příští aktualizaci frameworku by o změny nejspíš přišel. Stejně je tomu s případnými chybami ve frameworku. Pokud budete vyvíjet aplikaci a odhalíte nějakou závažnější chybu, určitě se vám nebude chtít čekat až jí někdo z vývojářů odstraní a vy budete moct framework aktualizovat. Pokud opravu provedete sami a komunita okolo frameworku ji, například na váš popud, nezahrne do distribučního balíčku, musíte jí po každé aktualizaci frameworku opravovat znovu. Řešením, i když poněkud náročným, je opět vytvořit si vlastní framework, který bude plně pod vaší kontrolou.

Dobrodružství

V neposlední řadě mě k tvorbě vlastního frameworku motivoval i fakt, že rád experimentuji a programování na obecné úrovni je pro mě velmi zábavné. Berme však tento důvod jako neoficiální.

9 Integrated Development Enviroment

kapitola 3:
Koncept ROAD frameworku

V této kapitole bude představen koncept nového frameworku pro tvorbu webových aplikací v jazyce PHP.

3.1 Základní koncept

ROAD je zkratkou pro **R**apid **O**nline **A**pplication Development, jestli tedy chce tento framework dostát svému jménu, musí umožňovat rychlý vývoj online aplikací. Chceme-li tedy shrnout základní požadavky na nově vznikající framework do pár hesel, budou to nejspíš tyto:

pevné a spolehlivé jádro

základní funkce spojené do jednoho pevně uzavřeného jádra s minimem potřebných nastavení a dodatečného kódu

komponentová architektura

ostatní funkce frameworku rozdělené do samostatných komponent, které je možné podle potřeby použít nezávisle na ostatních

událostmi řízená aplikační logika

samotná aplikační logika spouštěná vždy, když je potřeba, událostmi definovanými frameworkem

abstraktní přístup k různým koncovým zařízením

uživatelské prostředí definované pouze jednou nezávisle na používaném koncovém zařízením, pro nějž je prostředí vygenerováno přímo frameworkem

abstraktní přístup k databázi

abstraktní vrstva odstiňující konkrétní databázové systémy od programátora aplikace

rozšířená podpora práce s databází

třídy a funkce usnadňující práci s databází a nabízející dodatečné možnosti

integrovaná podpora JavaScriptu a AJAXu

uživatelské prostředí nabízející použití moderních technologií jako JavaScript a AJAX[16]

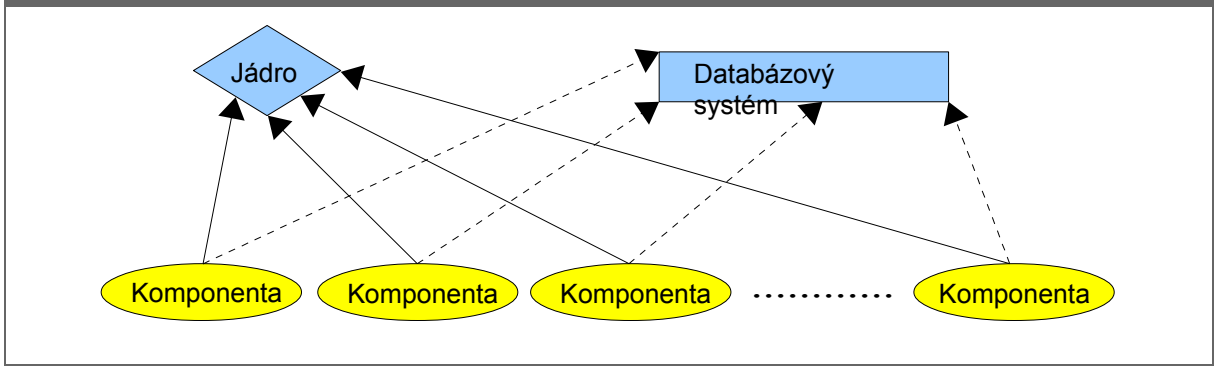
nezávislost na podporovaných technologiích

zajištění funkčnosti aplikace nezávisle na tom, jestli koncové zařízení podporuje JavaScript, AJAX, nebo třeba Cookies

3.2 Architektura frameworku

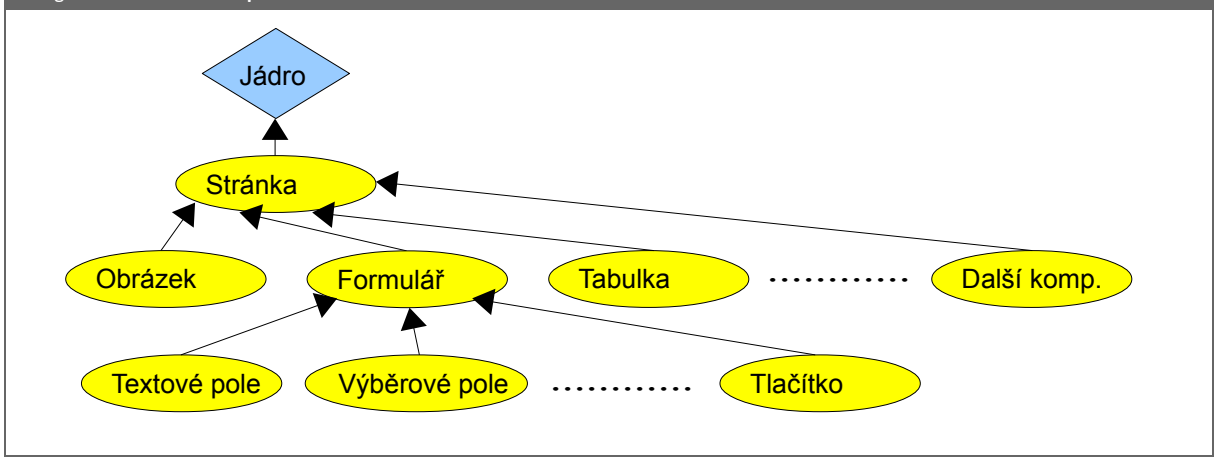
Základní rámec architektury je více méně předem daný, neboť se má jednat o framework komponentový. Primárním stavebním kamenem tedy bude komponenta reprezentující objektem obalenou relativně samostatnou funkci frameworku. Komponenty budou navázány na jádro, které bude provoz celého frameworku řídit. Paralelně s jádrem budou komponenty propojeny i s abstraktní databázovou vrstvou. Každá komponenta tedy bude moci využívat jinou databázi či přímo databázový server.

diagram č. 2 - **Konceptuální architektura frameworku**



Jádro bude neměnnou součástí frameworku a jeho běh bude modifikovatelný pouze pomocí parametrů (konstanty, vlastnosti) a předem definovaných událostí. Komponenta muže být jakýkoliv objekt odvozený od společné předem definované třídy. Každá komponenta bude moci obsahovat libovolné množství dalších komponent a takto propojené komponenty pak budou tvořit strom komponent.

diagram č. 3 - **Strom komponent**



Musíme si uvědomit, že každá stránka bude mít svůj strom komponent, který bude vypadat podle toho, co bude ta konkrétní stránka vykonávat. Programátor tedy bude muset pro každou stránku nadefinovat strom komponent znovu. Framework by s tím měl počítat a stavbu stromu komponent co nejvíce zjednodušit.

3.3 Integrace frameworku do aplikace

Mnoho PHP frameworků a rozšiřitelných webových aplikací používá pro přístup ke svým stránkám jeden soubor. Pro jednoduchost řekněme, že se tento soubor jmenuje `index.php` a je umístěn v kořenovém adresáři webové aplikace. Dotaz na jakoukoliv stránku je pak směrován do tohoto souboru a kód v tomto souboru pak rozhoduje, jaká stránka bude generována. Framework (nebo jádro aplikace) pak rozhodne, jaké další soubory bude potřebovat a načte je do paměti. V těchto souborech jsou pak obsaženy informace o tom, co má být provedeno a jak má být stránka vygenerována.

výpis kódu č. 6 - **Příklad integrace frameworku přes index.php**

```
<?php
  // Toto je výpis souboru index.php

  setlocale(LC_ALL,'cs_CZ');
  // ... a další incializační kód frameworku

  require('stranky/'.$_GET['stranka'].'.php');

  // ukončovací kód frameworku
?>
```

Já osobně jsem si tento přístup příliš neoblíbil především z těchto několika důvodů:

- bez použití mod_rewrite nebo jiného nástroje pro přepisování URL se bohužel pro přístup ke stránkám takovéto aplikace nevyhneme nepěkným URL (`/index.php?stranka=produkty`); díky tomu jsou URL:
  - špatně zapamatovatelné
  - nevhodné pro vyhledávače
  - nepřehledné i pro vývojáře
- před tím, než je zavolán konkrétní soubor, framework vykoná množství akcí, které nemáte možnost nijak ovlivnit
- u některých frameworků a aplikací je dokonce v době, kdy je volán soubor stránky, již vypsána část výstupního HTML kódu, což znemožňuje úpravu HTTP hlaviček a změnu výstupního formátu
- URL skriptu uložené v globálních proměnných je u každé stránky stejné
- zkrátka jsem raději, když mám věci pod kontrolou

Druhou možností, jak integrovat framework do aplikace, je volat v každém souboru, který reprezentuje konkrétní stránku, několik řádek kódu. Tímto kódem načteme zaváděcí soubor frameworku, případně ho donastavíme a následně spustíme. Nevýhodami tohoto přístupu je opakování oněch pár řádek kódu a potřeba vědět, ve kterém adresáři se aktuálně nacházíme, abychom byli schopni zavolat zaváděcí soubor frameworku. V okamžiku zavádění totiž ještě nejsou nadefinovány konstanty a proměnné, do kterých si framework ukládá absolutní

cestu ke svým souborům.

výpis kódu č. 7 - **Příklad integrace frameworku přes načtení v souborech stránek**

```
<?php
  // Toto je výpis souboru stranka.php
  require_once('road/run.php'); // načtení jádra frameworku
  $page->title = 'Vítejte'; // dodatečná konfigurace frameworku

  // vlastní kód stránky: definice stromu komponent, obsluha událostí...

  $road->run(); // spuštění frameworku
?>
```

ROAD framework bude do aplikací integrován druhým způsobem, tedy načítáním zaváděcího souboru a manuálním spouštěním. Každý soubor stránky bude načítat zaváděcí soubor, který se postará o načtení všech potřebných souborů jádra a základních komponent, vytvoří základní objekty frameworku a podle konfiguračního souboru provede příslušná nastavení. Poté budou v souboru stránky definovány dodatečné parametry a bude vytvořen strom komponent. Voláním metody jádra frameworku pak bude framework spuštěn a dle potřeby vygenerována požadovaná stránka.

3.4 Události

ROAD framework bude ve svém jádru i v jednotlivých komponentách zapouzdřovat množství z vnějšku nepříliš transparentní programové logiky a pro programátora by bylo složité řídit běh aplikace klasickým procedurálním způsobem. Z tohoto důvodu jsem se při návrhu frameworku inspiroval moderními programovacími jazyky a jejich knihovnami vizuálních komponent a rozhodl jsem se nechat programátory řídit provoz frameworku pomocí obsluhování událostí.

Jádro i komponenty frameworku budou mít definované události, ke kterým při běhu aplikace dochází. Uživatel bude mít možnost k těmto událostem navázat funkce, jenž framework zavolá v případě, že událost nastane. V rámci těchto funkcí pak bude zavolán programátorův kód a budou vykonány příslušné akce.

Při použití standardního procedurálního přístupu musel programátor kontrolovat stav aplikace pomocí podmínek, aby věděl, co se od jeho kódu v konkrétní okamžik vlastně očekává. V událostmi řízené aplikaci pouze napíše kód a řekne, kdy se má tento kód zavolat.

výpis kódu č. 8 - **Procedurální obsluha aplikace**

```
<?php

// podminka zjišťující stávající situaci
if(!empty($_POST['jmeno']) && !empty($_POST['prijmeni'])) {
  // kód k vykonání
  mysql_query('INSERT INTO osoby ('…
}

?>
```

výpis kódu č. 9 - **Událostmi obsluhovaná aplikace**

```
<?php

// funkce obsluhující událost
function ulozit() {
  // kód k vykonání
  mysql_query('INSERT INTO osoby ('…
}

// přiřazení funkce k události
$formular->onSave = 'ulozit';

?>
```

Jak je vidět na první pohled, použitím událostí je kód více strukturovaný a mnohem přehlednější. Použití událostí má i další výhody:

- programátor nepotřebuje znát technická specifika jazyka, aby určil, kdy má být kód vykonán
- událost je vždy volána pouze v okamžiku, kdy jsou všechny předpoklady pro její volání splněny
- události mohou nést parametry blíže specifikující danou situaci
- použití událostí eliminuje opakování kódu:
  - událost může být vyvolána na více místech programu aniž by se tomu musel programátor přizpůsobovat
  - jedna funkce může být přiřazena více událostem
  - jedna funkce může být přiřazena stejné události u více komponent
- kód obsluhující událost může být umístěn kdekoliv ve zdrojovém kódu (samozřejmě ale musí být definován dříve než dojde k vyvolání události)

Stejně jako každá část frameworku si i mechanismus událostí s sebou nese problém menší flexibility. Flexibilita je u událostí omezena hlavně tím, že programátor může využít pouze ty události, které jsou k jednotlivým komponentám nadefinované. Pokud jsou komponenty ve vztahu k požadované funkcionalitě navržené špatně, mohou některé důležité události chybět a programátorovi pak nezbývá než se vrátit k procedurálnímu přístupu, případně oba principy nějak kombinovat.

3.5 Optimalizace pro různá koncová zařízení

Webové aplikace jsou trendem současné doby a díky rozvoji technologií jako JavaScript a AJAX často nahrazují klasické desktopové aplikace i v oborech, ve kterých by to dříve nepřipadalo v úvahu. Příčiny zvyšující se popularity webových aplikací jsou evidentní:

- absence instalace
- bezproblémový přístup z jakéhokoliv k internetu připojeného počítače

- není potřeba žádná synchronizace mezi jednotlivými zařízeními

Jednoduše řečeno webové aplikace přinášejí nezávislost; nezávislost na hardwaru, operačním systému, typu připojení a zeměpisné poloze. Bohužel tato nezávislost není dokonalá a je vždy podmíněna konfigurací zařízení, kterým k webové aplikaci přistupujete. Čím pokročilejší a specifičtější technologie při vývoji aplikace použijete, tím více se tyto odchylky projeví.

3.5.1 Nezávislost na webovém prohlížeči

V roce 1994 bylo založeno World Wide Web Consortium (W3C)[17], které se od té doby stará o navrhování a vydávání webových standardů. Standardy, které vydává, jsou dobře zdokumentované a dávají vývojářům pevný rámec, ve kterém se mohou při vývoji pohybovat. Co už ale bohužel není tak slavné, je podpora těchto standardů v nejrůznějších moderních prohlížečích. Je pochopitelné, že se vývojář nemůže spolehnout na to, že uživatel aplikace bude používat vždy aktuální verzi prohlížeče a bude tak schopen správně „zobrazit“ stránku využívající nejnovější verzi standardů. Zarážející ale je, že například prohlížeč Internet Explorer firmy Microsoft ani ve verzi 8 neodstranil se standardy spojené nedostatky, které znepříjemňovaly vývoj aplikací pro Internet Explorer verze 5. Jestliže jsou prohlížeče většiny ostatních výrobců o krok pozadu za standardy, pak Internet Explorer ztrácí minimálně půl kola.

Jelikož framework bude využívat pokročilé technologie jako JavaScript a AJAX, bude detekce prohlížeče a jeho konfigurace klíčovou funkcí ke správnému provozu aplikace. Framework by měl nejdříve zjistit o jaký prohlížeč se jedná a poté vykonat test, na jehož základě zjistí reálné aktuální nastavení prohlížeče. Zjistí tedy, zda může aplikace využívat JavaScript, zda se může na server dotazovat asynchronními dotazy (AJAX) a zda může využívat Cookies. Po detekci konfigurace jí musí framework automaticky přizpůsobit svůj běh. Všechny komponenty tedy musí fungovat jak s JavaScriptem, tak bez něj, jádro musí umět udržet sezení i bez použití Cookies a v případě, že prohlížeč neumožňuje asynchronní dotazy, musí být všechny dynamické komponenty schopny generovat svůj obsah již při generování celé stránky.

3.5.2 Nezávislost na platformě

Jestliže minulé roky byly dobou rychlého rozvoje osobních počítačů co do počtu i výkonu, pak v současnosti můžeme největší vývoj zaznamenat u mobilních zařízení[10]. S tímto rozvojem je spojena i snaha většiny vývojářů webových aplikací, zpřístupnit jejich aplikace držitelům těchto mobilních zařízení. Většina nových zařízení dnes umí zobrazit klasickou webovou stránku určenou pro osobní počítače, ale tato stránka se většinou na malém displayi stává nepřehlednou a díky použití JavaScriptu, reakcím na klávesy a podobně, také hůře ovladatelnou. Cílem frameworku bude přizpůsobit výstup aplikace pro

10 Mobilními zařízeními v tomto kontextu myslíme PDA, tablety a dnes především mobilní telefony. Přenosné počítače typu notebook (laptop) bychom řadili spíše do kategorie osobních počítačů.

platformu, ze které uživatel k aplikaci přistupuje, aniž by programátor musel tvořit dvě verze stránky. Tento princip vyžaduje použití architektury *Model-View-Controller*.

Model-View-Controller architektura

Architektura MVC je princip rozdělení aplikace do tří téměř nezávislých vrstev, z nichž každá se stará o jinou úroveň aplikace.[18] Požadavkem na tyto tři vrstvy je, aby šla každá vrstva změnit s co nejmenším dopadem na ostatní vrstvy. Funkce, které jednotlivé vrstvy v aplikaci vykonávají jsou:

- **Model:** Skladuje, uchovává a zpřístupňuje data[11] aplikace v definované a srozumitelné struktuře.
- **View:** Zpřístupňuje data uživateli a tvoří prostředí pro interakci s ním.
- **Controller:** Reaguje na vnější podněty (většinou akce uživatele) a příslušným způsobem upravuje Model.

Klademe-li požadavek nezávislosti vrstev, musíme být samozřejmě shovívaví a realističtí, neboť, kdyby byly vrstvy naprosto nezávislé, nemohly by spolu nikterak komunikovat a aplikace by nikdy nemohla fungovat. Závislost vrstev je určena jednoznačně a pevně definovaným rozhraním, přes které spolu vrstvy komunikují. Jakmile změna v jedné vrstvě vyžaduje změnu rozhraní, jedná se o zásah i do sousedních vrstev, neboť i ty musí být přizpůsobeny tak, aby s novým rozhraním uměly pracovat. Návrhu rozhraní by mělo být věnováno hodně pozornosti, abychom se do budoucna vyhnuli potřebě ho měnit.

Ve většině aplikací je MVC architektura použita pro umožnění snadné změny nebo dokonce výměny Model nebo View, přičemž Controller je více méně to, co zůstává pevné a neměnné. Za změnu Modelu můžeme považovat například změnu databázového systému a pod změnou či výměnou View si můžeme představit výběr jiného uživatelského prostředí. Nás bude v této kapitole zajímat zejména změna View.

Renderovací objekty

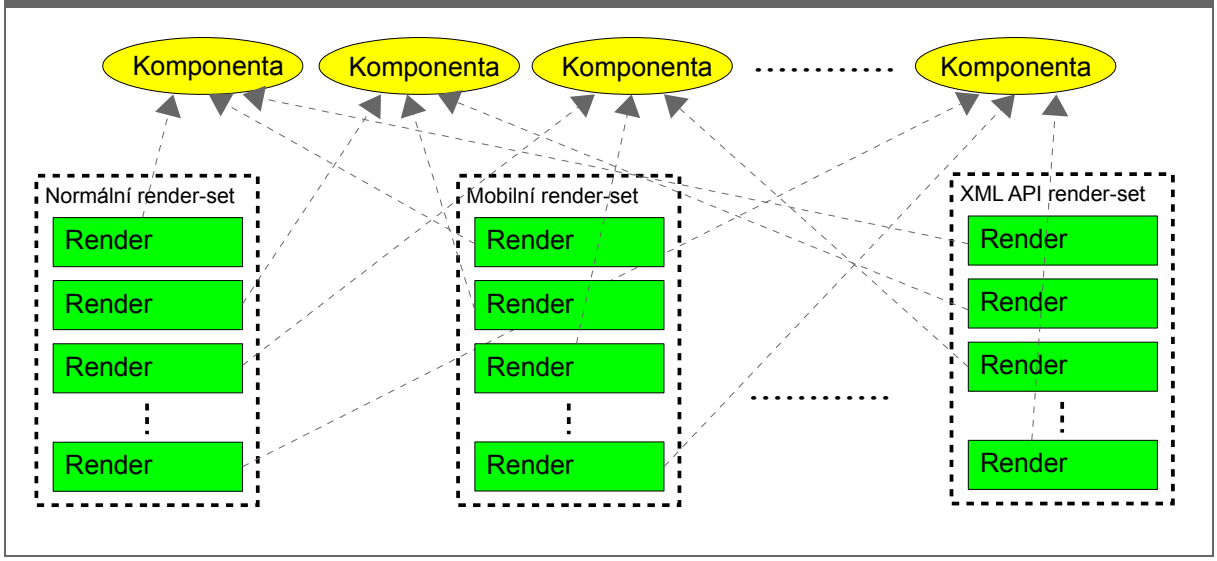
Hojně webových aplikací a frameworků nabízí použití rozličných šablon a stylů a tím jejich modifikovatelnost končí. ROAD jde v tomto smyslu mnohem dále a jeho cílem je umožnit opravdovou *výměnu* celého View za jiné bez jakéhokoliv vlivu na ostatní vrstvy. Kvůli tomu ROAD představuje princip renderovacích objektů. Ve zkratce půjde o to, že komponenty samy o sobě nebudou obsahovat kód určený ke generování výstupu. Tento kód bude obsažen v tzv. *renderech*, třídách, které budou generování výstupu komponent obsluhovat. Každá komponenta tedy bude mít jeden nebo více renderů v závislosti na tom, pro kolik typů uživatelského rozhraní bude komponenta určena.

Rendery budou sdružovány do tzv. *render-setů* obsahujících rendery generující

11 Důležité je podotknout, že daty nejsou myšleny pouze řádky uložené v databázi, ale i například nastavení frameworku, nebo při malém přimhouření oka i vlastnosti komponent.

jedno konkrétní uživatelské rozhraní. Pouhou výměnou render-setu pak framework docílí generování výstupu v diametrálně odlišné formě. Framework pak bude schopen bez zásahu programátora generovat rozdílné uživatelské rozhraní nejen pro osobní počítače a mobilní zařízení, ale i například pro aplikace třetích stran formou XML API.

diagram č. 4 - **Rendery a render-sety**



MVC v ROAD

Z minulé podkapitoly vyplývá, jak je v ROAD frameworku koncipována vrstva View. Zbývá otázka, jaká část frameworku reprezentuje vrstvu Model a jaká vrstvu Controller. Pro přehlednost si všechny vrstvy i jejich vztah k frameworku ještě jednou vyjmenujeme:

Model

Model je ve frameworku reprezentován především abstraktní databázovou vrstvou, databázovým ovladačem a samotným databázovým systémem. Další data pak budou uložena v konfiguračním souboru a v souborech jednotlivých stránek, kde jsou reprezentovány hodnotami přiřazenými vlastnostem komponent.

View

Vrstva View je reprezentována renderovacími objekty, které jsou seskupeny do render-setů. Více o této vrstvě naleznete v předchozí podkapitole.

Controller

Controller je v ROAD zastoupen jádrem frameworku a jeho jednotlivými komponentami. Ty se starají o reakce na uživatelovy podněty a přizpůsobují Model podle jeho požadavků. Chování komponent a jádra je navíc ovlivněno

přímo programátorem pomocí obsluhy událostí, takže do Controlleru můžeme zahrnout i kód obsažený v jednotlivých souborech stránek.

3.6 Práce s databází

Tvoříme-li desktopovou aplikaci, většinou o databázovém systému začneme uvažovat až tehdy, když zjistíme, že budeme potřebovat ukládat veliké množství záznamů. Desktopové aplikace mají k dispozici spousty operační paměti, ve které mohou data shromažďovat po celou dobu jejich běhu. Aplikace před svým ukončením potřebná data uloží do souboru na disku a pří příštím spuštění je zas načte. Při tvorbě webové aplikace (především v PHP) je ale v paměti uložena pouze instance aplikace, která právě generuje aktuální stránku. Ostatní data nashromážděná při interakci by byla ztracena, kdyby nebyla uložena právě pomocí databázového systému.

3.6.1 Abstraktní vrstva

Důležitost abstraktní databázové vrstvy byla vysvětlena v kapitole 1.1.5, a nyní se proto můžeme rovnou soustředit na popis této vrstvy tak, jak bude zkonstruována v ROAD frameworku. Základem abstrakce je rozhraní, přes které bude programátor k databázi přistupovat. To musí obsahovat především funkce pro vykonání SQL příkazu, získání záznamů pomocí SQL dotazu a práci s transakcemi. Dále musí mít nástroj pro skladování a procházení získaných záznamů.

Abstraktní databázová vrstva v ROAD frameworku bude rozdělena na dvě hlavní části. První částí bude tzv. připojení. To má, jak již název napovídá, za úkol připojit se k databázovému serveru a předávat mu dotazy a příkazy zadané aplikací v jazyce SQL. Po volání dotazu, navracejícího záznamy z databáze, přijde na řadu druhá hlavní část, kterou je tzv. data-set, tedy soubor dat. Data-set bude programátorovi aplikace nabízet funkce pro pohyb v záznamech a bude umožňovat převedení záznamů do požadované struktury (např. pole, objekt, pole objektů atd.).

Implementace těchto dvou hlavních částí bude řešena pomoc dvou abstraktních tříd definujících jejich rozhraní, přes které k nim bude aplikace a programátor přistupovat. Z těchto dvou tříd pak budou dědit jednotlivé *„ovladače“*, tedy třídy ovládající konkrétní databázový systém.

3.6.2 Objektově-Relační Mapování

ORM je obecně velmi vágně definovaný termín. V některých článcích se ORM zmiňuje ve spojení s ORDBMS, tedy objektově-relačními databázovými systémy, které díky své architektuře skutečně umožňují uložit libovolné objekty i jejich propojení se světem ostatních objektů přímo do databáze a poté tyto objekty načíst zpět do paměti. V jiných článcích se zas za ORM považuje implementace nadstavby nad relačním databázovým systémem, která automaticky načítá

hodnoty polí do příslušných proměnných předem definovaných objektů, případně zapouzdřuje formulaci dotazů generovaných při volání určitých funkcí těchto objektů. V jistém slova smyslu můžeme za ORM považovat i PHP funkci `mysql_fetch_object()`, která vrací objekt vygenerovaný z aktuálního záznamu v databázi.

Frameworky jsou navrhovány jako obecné podpůrné nástroje. Proto ani ROAD nemůže dopředu vědět, jaká bude struktura objektů aplikace, která bude na frameworku postavena. Z tohoto důvodu nemůžeme počítat s tím, že bude možné úspěšně automaticky mapovat objekty na data v databázi. Vzhledem k tomu, že implementovat ORDBMS v jazyce PHP by bylo bláznovství, nemůžeme od ROAD frameworku očekávat ani to, že bude umožňovat ukládání objektových struktur do databáze. Převod dat záznamu do objektu je v PHP integrovaný přímo, takže touto funkcí bychom nepřinesli nic nového.

ROAD framework se proto ubírá trochu jiným směrem a jako v mnoha jiných aspektech se i zde inspiruje u knihoven komponent složitějších kompilovaných jazyků. Ze zkušeností s tvorbou webových aplikací vím, že je občas otravné znovu a znovu konstruovat SQL dotazy pro upravení, případně přidání záznamu. Vždy je potřeba ohlídat správné ošetření hodnot proti útokům, převádět hodnoty do řetězců odpovídajících datovým typům konkrétních polí, správně formulovat podmínku pro vyhledání aktualizovaného záznamu a hlídat samotný syntax SQL dotazu.

ROAD proto přichází s alternativou v podobě implementace návrhového vzoru Active Record[19] jako rozšíření třídy data-set. Ta kromě klasického převodu záznamů do nejrůznějších struktur bude obsahovat i možnost záznam otevřít v modifikovatelné struktuře vyjádřené polem objektových reprezentací jednotlivých datových polí vráceného záznamu. Každé pole načtené ze záznamu pak obsahuje hodnotu i informaci o jejím datovém typu. Programátor pak může hodnotu několika polí změnit a po volání příslušné funkce záznam uložit. Třída data-set pak sama automaticky vygeneruje příslušný SQL dotaz a odešle ho do databáze. Sama se tedy postará o převod a ošetření hodnot i o stanovení podmínky pro vyhledání záznamu, který má být aktualizován.

Kromě evidentního ulehčení práce s databází pro samotného programátora tato nadstavba na databázovým ovladačem umožňuje i automatizaci načítání a ukládání záznamů. Díky tomu může programátor využívat některé komponenty pro práci s databází, které jsou po pouhém nastavení názvu tabulky schopné načíst hodnoty do formuláře a v případě, že si to uživatel vyžádá i uložit provedené změny ve formuláři. Jak bude zmíněno v dalších kapitole, podpora formulářů je jednou z nejdůležitějších funkcí ROAD frameworku.

3.7 Podpora práce s formuláři

Bez debaty nejzdlouhavější a zároveň nejzranitelnější činností, co se bezpečnosti aplikace týče, je práce s formuláři. Bez použití frameworku se programátor u každého formuláře nevyhne těmto krokům:

- načtení hodnot z databáze a převedení do uživatelem čitelného formátu
- samotná tvorba formuláře v HTML kódu
- kontrola hodnot zadaných uživatelem (povolený rozsah a obsah)
- případné vyrozumění uživatele o chybách ve formuláři
- převod hodnot do formátu použitelného v SQL
- odeslání do databáze a prezentování úspěchu či neúspěchu

ROAD framework se snaží množinu těchto činností minimalizovat a práci s formuláři co nejvíce automatizovat. V ideálním případě pak na programátora zbude pouze jeden úkon:

- definice struktury formuláře a provázání polí s databází

3.7.1 Aktivní prvky formuláře

Veškeré aktivní prvky na formuláři budou v ROAD frameworku reprezentovány samostatnými komponentami. Celý formulář tedy bude v kódu definovaný jako podstrom komponent, jehož kořenem bude komponenta *„formulář“*. K dispozici budou komponenty reprezentující nejen textová pole, výběrové seznamy, zaškrtávací pole a další standardní HTML prvky, ale i složitější prvky jako dynamické výběrové seznamy, seznamy zaškrtávacích polí atd. Komponenty budou obsahovat množství konfiguračních vlastností, které diverzitu prvků na stránce ještě rozšíří. Mezi další vlastnosti budou patřit i parametry pro kontrolu zadaných hodnot. Například u textového vstupního pole půjde určit, zda je hodnota povinná, v jakém formátu má hodnota být a v jakém rozmezí se má pohybovat (délka pro text a meze pro číselné hodnoty a pro datum). Kromě této základní kontroly bude framework automaticky schopen zkontrolovat i duplicitu zadávané hodnoty. Na každý vstupní ovládací prvek bude možno navázat událost pro validaci, takže programátor bude moci v případě potřeby hodnotu zkontrolovat důkladněji.

Ve standardech jazyka HTML není definováno nijak závratné množství vstupních ovládacích prvků a mnoho webových aplikací řeší tento nedostatek použitím JavaScriptu. Pokud prohlížeč řádně podporuje JavaScript, je programátor schopen nabídnout uživateli mnohem interaktivnější prostředí. ROAD se proto snaží držet krok a nabízí u některých svých komponent obohacení JavaScriptem. Použitím JavaScriptu ROAD umožňuje nejen otevírat a zavírat skupiny ovládacích prvků, vybírat zaškrtávací pole v seznamech a filtrovat v nich, ale i například dynamicky obnovit část formuláře, nebo pomocí AJAX odeslat formulář bez obnovení stránky.

Princip využití JavaScriptu a AJAXu uvedeme na jedné z nejkomplikovanějších komponent ve frameworku. Komponenta slouží pro dynamické vyhledání a připojení položky k aktuálnímu záznamu. Uživateli je nabídnuto textové pole, do něhož může zadat kus textu identifikující položku, kterou chce připojit. Jakmile tak učiní, stránka si pomocí AJAX dotazu stáhne podle zadaného textu

vyfiltrované položky a nabídne je uživateli ve vyskakovací nabídce. Potom, co uživatel jednu z položek vybere, je tato položka připojena k aktuálnímu záznamu. Z posledního odstavce můžete mít pocit, že tato komponenta bude vyžadovat spoustu dodatečného kódu, který bude filtrovat položky a odesílat je v definovaném formátu stránce. ROAD framework však všechny tyto akce provádí sám. Ve skutečnosti bude definice této komponenty stejná jako definice seznamu s možností výběru více položek.

Jak již bylo zmíněno dříve, ROAD framework si zakládá na kompatibilitě se všemi možnými koncovými zařízeními, proto zajišťuje, že každý ovládací prvek bude ovladatelný i bez podpory JavaScriptu a AJAXu. Stručně řečeno, uživatelé s prohlížečem podporujícím JavaScript a AJAX budou z těchto technologií profitovat a ostatní budou co nejméně penalizováni.

3.7.2 Automatizace databázových operací

Díky komponentám pro zadávání hodnot, odpadá programátorovi starost o interakci s uživatelem, stále je to ale on, kdo musí načíst záznam z databáze, hodnoty jeho polí nastavit komponentám a provést fyzické uložení změněných hodnot do databáze. Naštěstí ROAD framework přichází s komponentou, která právě tyto akce automatizuje. Tato komponenta reprezentuje vazbu na tabulku v databázi. Jediné co je potřeba nastavit je název tabulky (resp. SQL dotaz), ze které se má záznam načíst a u komponent reprezentujících jednotlivé vstupní ovládací prvky zadat název pole, ze kterého se čerpá jeho hodnota. Vše ostatní provede komponenta automatizovaně sama. Při otevření formuláře načte pole příslušného záznamu do jednotlivých ovládacích prvků a při uložení formuláře zase zjistí všechny změny a záznam v databázi zaktualizuje.

Díky událostmi řízené architektuře je možné do průběhu načítání nebo uložení zasáhnout a přizpůsobit tyto činnosti konkrétní situaci.

3.7.3 Více záznamové formuláře

Zavedeným standardem webových aplikací je něco, čemu by se dalo říkat *„architektura dvou stránek“*. Aplikace pod jednou stránkou nabízí seznam existujících záznamů a po kliknutí na odkaz u vybraného záznamu nás přenese na formulář pro jeho úpravy, tedy druhou stránku. Tato architektura se může jevit jako dostačující, ale existují situace, kdy uživatel potřebuje upravit více záznamů najednou a přepínání mezi seznamem a formulářem ho zdržuje. ROAD pro tento problém nabízí řešení ve formě formuláře pro úpravu více záznamů najednou. Vše, co je potřeba udělat, je nadefinovat formulář pro jeden záznam a použít komponentu pro stránkování. U stránkovací komponenty se nastaví z jaké tabulky (resp. SQL dotazu) se mají záznamy čerpat a komponenta sama pak zařídí, že formulář bude zduplikovaný tolikrát, kolik záznamů je zrovna v tabulce (resp. vráceno na SQL dotaz). Načtení i uložení hodnot je samozřejmě plně automatizované tak, jak je popsáno v předchozí podkapitole.

Formulář nadefinovaný tímto způsobem může obsahovat všechny typy vstupních

ovládacích prvků od prostých textových polí až ty nejsložitější JavaScriptem obohacené komponenty. Díky tomu můžeme vytvořit formulář pro úpravu téměř neomezeného množství záznamů najednou stejně rychle, jako jakýkoliv jiný formulář.

3.8 Podpora více jazyků

Velké množství významných desktopových aplikací bylo za roky jejich existence distribuováno ve více zemích a více jazykových verzích. Většina těchto aplikací byla primárně vytvořena v jednom jazyce a poté lokalizována pro další země. Každé instalační médium takovéto aplikace bylo většinou už předem určeno pro jednu zemi (skupinu zemí) a tomu odpovídala i jazyková lokalizace.

S přesunem aplikací na web se ale tento způsob lokalizace jeví jako nepoužitelný. Proto se čím dál více aplikací mění ve více-jazykové. Taková aplikace má aplikační logiku zcela oddělenou od řetězcových konstant a formátů specifických pro danou zemi. Při dotazu na stránku, aplikace vybere vhodný jazykový soubor podle HTTP hlaviček, které zasílá webový prohlížeč, a vrátí stránku lokalizovanou do příslušného jazyka. Tento princip je příjemný v tom, že veškerá aplikační logika a také data v aplikaci existují jen jednou. Velkým plusem je také snadné přidání další lokalizace. Stačí přeložit soubory s texty, zaregistrovat nový jazyk a aplikace je rázem dostupná i pro uživatele z další země.

V ROAD frameworku komponenty ani jádro neobsahují žádné jazykově specifické texty, je tedy možné framework bez obav použít pro více-jazykové aplikace. Všechny texty, formáty a dokonce i texty chybových hlášek lze snadno lokalizovat díky tomu, že se vyskytují v jednom samostatném souboru. Kromě textů, které jsou umísťovány na generovanou stránku, jsou v tomto souboru i texty, kterými s uživatelem komunikuje prostřednictvím JavaScriptu.

Kromě frameworku samotného je ale potřeba lokalizovat i aplikaci, která je na něm vyvíjena. K tomu slouží mechanismus *„překladu“*. Některé komponenty mají jednu nebo více textových vlastností, jako třeba popisek nebo nápovědu. Jakmile těmto vlastnostem uživatel nastaví natvrdo nějaký jazykově specifický text, stává se tato komponenta nelokalizovatelnou. Proto ROAD nabízí možnost zadat hodnoty těchto vlastností jako název konstanty. Všechny textové vlastnosti začínající řetězcem `LNG_` jsou brány jako názvy konstant a při překladu, který je (pokud je aplikace nastavena jako více-jazyková) volán automaticky, dojde k nahrazením hodnotou této konstanty. Programátor tak může pomocí ROAD frameworku bez problému vytvořit zcela lokalizovatelnou aplikaci.

3.9 Shrnutí

V předchozích podkapitolách jsme si stručně představili koncept ROAD frameworku jako použitelného a v lecčem inovativního nástroje pro snadnou tvorbu webových aplikací. Ať tvoříte jakoukoliv aplikaci, framework, nebo jiný nástroj, vždy narazíte na problém nepřímé úměry univerzálnosti a snadnosti

použití. Je jasné, že čím bude framework univerzálnější, tím více kódu bude muset programátor napsat, aby vše fungovalo, tak jak si představuje. Na druhou stranu, budeme-li chtít framework, který provede většinu operací sám téměř bez zásahu programátora, bude tento framework evidentně méně flexibilní.

Osobně se v posledních letech mého programování snažím ve svých aplikacích razit ideu jakési *„Just-In-Time konfigurace“*. Základní požadavky kladené na takovouto aplikaci (resp. její část) jsou:

- okamžitá funkčnost bez potřeby nastavení; aplikace běží v režimu, v jakém jí nejspíš bude využívat většina uživatelů po většinu času
- možnost pozměnit nastavení chování aplikace přímo z části aplikace, ve které se požadovaná funkce spouští (tedy minimalizace potřeby vstupů do předvoleb)

Ke konfiguraci aplikace tedy dochází ve chvíli, kdy uživateli nedostačuje standardní chování (proto *Just-In-Time*). Aplikace se pak trochu vymyká dříve definovanému pravidlu o nepřímé úměře mezi univerzálností a jednoduchostí. Pro občasného uživatele není problém aplikaci ovládat a pro pokročilého uživatele nabízí množství parametrů ovlivňujících její běh.

Vztáhneme-li ideu *Just-In-Time konfigurace* na webový framework, budeme za uživatele frameworku považovat programátora aplikace. Jako takovému se mu ROAD snaží při jeho minimálním úsilí nabídnout jednoduše fungující komponenty, které lze v případě potřeby pomocí všemožných vlastností přizpůsobit k obrazu svému. Pokud programátorovi nestačí ani vlastnosti komponent, přichází na řadu události. Díky této promyšlené technice popsané v kapitole 3.4 je chování frameworku a jeho komponent možno pozměnit více, než by to dokázaly stovky vlastností a parametrů.

kapitola 4:
Implementace ROAD frameworku

V této kapitole si popíšeme implementaci frameworku na technické úrovni. Popíšeme si některé problémy, na které jsem během implementace narazil, a jak jsem tyto problémy řešil.

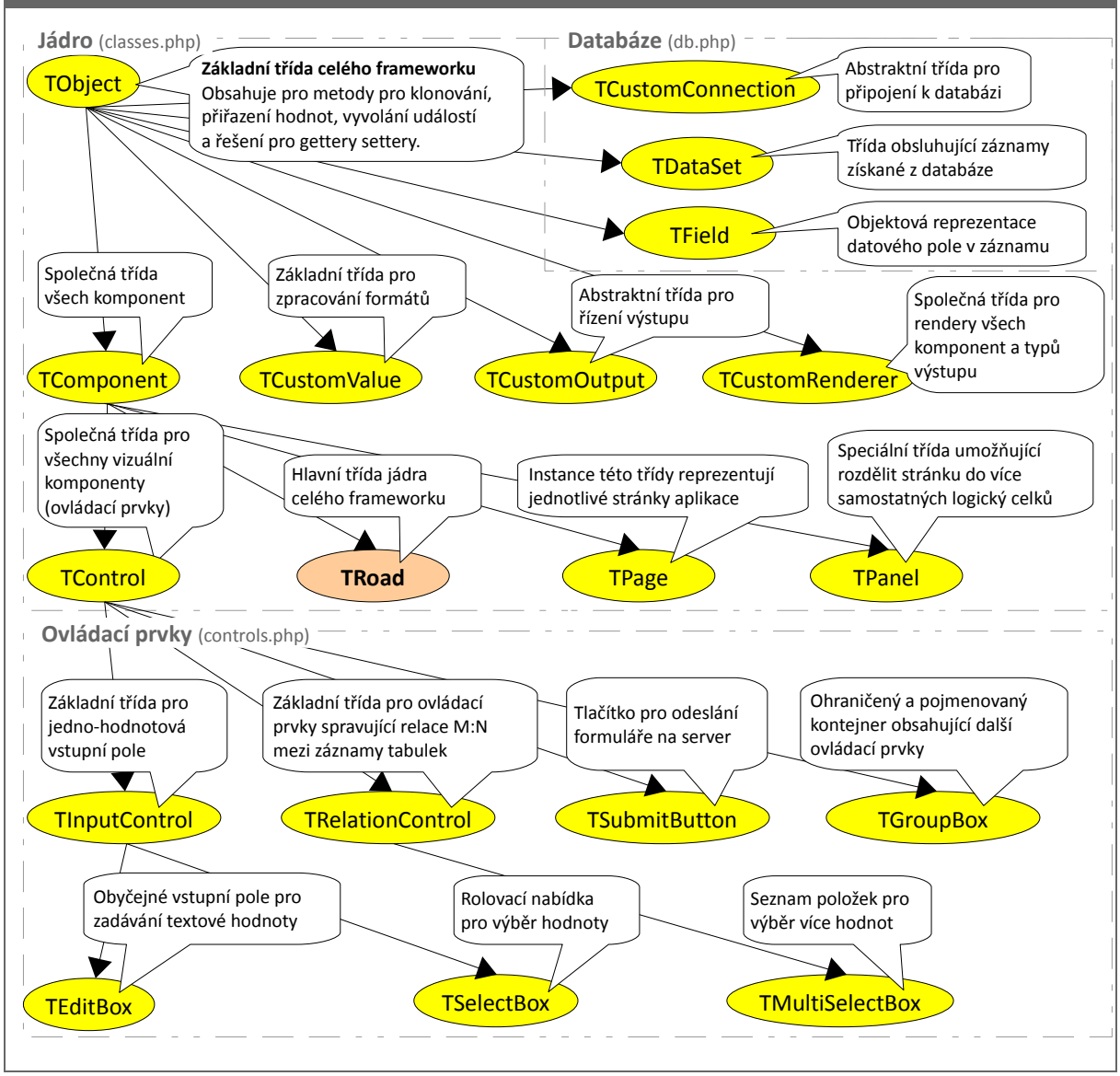
4.1 Struktura frameworku

V úvodu této kapitoly si nastíníme základní strukturu frameworku, abychom se v něm při dalším výkladu lépe zorientovali.

4.1.1 Struktura tříd

Pro snadnější orientaci v prostoru všech tříd frameworku můžete použít tento následující diagram tříd. Jelikož strom všech komponent a ovládacích prvků, které framework obsahuje, by se na papír formátu A4 určitě nevešel, je tento diagram značně zjednodušený (resp. prořezaný). Třídy v něm zobrazené jsou určeny buď přímo pro použití při vývoji aplikací, nebo slouží jako abstraktní třídy, ze kterých by měl vývojář aplikace nebo tvůrce komponent dědit.

diagram č. 5 - **Zjednodušený diagram tříd**



Šipky spojující jednotlivé třídy vyjadřují vztah dědičnosti mezi těmito třídami, přičemž šipka vždy směřuje od předka k potomkovi.

Základní třídy

Aby mohl programátor správně využívat ROAD framework, musí pochopit některé jeho základní principy. Proto si zde popíšeme některé důležité základní třídy.

TObject

Třída `TObject` je základní třídou, ze které dědí všechny ostatní třídy celého frameworku. Součástí této třídy jsou metody pro:

- načtení hodnot vlastností z jiného objektu nebo asociativního pole: `assign()`, `assignArray()`
- klonování objektu (v případě komponent i celého podstromu): `cloneMe()`
- standardní práci s objekty a třídami (integrované funkce PHP převedené do OOP modelu): `getClass()`, `getParentClass()`, `isInstanceOf()`, `isImplementorOf()`, `getMethods()`, `getVars()`, `getClassVars()`
- vyvolání události: `fireEvent()`

Kromě nabídky těchto metod třída řeší i přístup k nedefinovaným vlastnostem. Za dobu mé práce s Delphi a experimentování s C# jsem si oblíbil systém vlastností[20], který tyto jazyky implementují. Ve zkratce se jde o to, že programátor přistupuje k vlastnostem přímo jako k proměnným, ale v případě, že to třída vyžaduje, jsou při zápisu a/nebo čtení volány metody zvané *„settery“* a *„gettery“*.

výpis kódu č. 10 - **Systém vlastností v Delphi (Object Pascal)**

```
TTrida = class(TRodic)
private

  // proměnná s uloženou hodnotou
  FNazev: String;

  // gettery a settery
  function GetCastka: Double;
  function GetPocet: Integer;
  procedure SetCastka(const Value: Double);
  procedure SetNazev(const Value: String);
public

  // vlastnost s přímým čtením z proměnné
  // a zápisem přes setter
  property Nazev: String read FNazev write SetNazev;

  // vlastnost jen pro čtení přes getter
  property Pocet: Integer read GetPocet;

  // vlastnost čtená přes getter
```

```
  // a zapisovaná přes setter
  property Castka: Double read GetCastka write SetCastka;
end;

...

objekt := TTrida.Create();

objekt.Nazev := 'Název';
pocet := objekt.Pocet;
```

Tento systém je příjemný především v tom, že programátor nemusí volat metody getterů a setterů. Díky tomu je použití vlastností mnohem lépe čitelné a lépe se integruje do ostatního kódu.

V PHP jsou všechny vlastnosti objektů jen obyčejné proměnné. O nějakém pokročilém systému vlastností tedy nelze mluvit. PHP ale díky tomu, že je interpretovaným jazykem, nabízí možnost zpracování přístupu k nedefinovaným proměnným. Všechny pokusy o čtení nedefinované proměnné jsou směrovány do metody `__get()` a všechny pokusy o zápis do metody `__set()`. Tyto dvě metody můžeme brát jako tzv. univerzální gettery a settery.

Při experimentování s těmito metodami jsem se rozhodl vytvořit obecný systém vlastností tím, že ho implementuji do univerzálních getterů a setterů třídy `TObject`. Vlastnost je v takovém případě vlastně virtuální proměnnou. Když pak programátor přistupuje například k proměnné `nazev` a tato proměnná neexistuje, jsou volány metody `__get()` nebo `__set()`. Kód těchto metod pak zjišťuje, zda existují metody `_getNazev()` nebo `_setNazev()`, případně přímo proměnná `_nazev`. Následně je příslušná metoda volána, nebo je nastavena hodnota (resp. získána) do (resp. z) vnitřní (privátní nebo chráněné) proměnné prefixované podtržítkem.

Tento systém splňuje všechny požadavky, které jsou na přístup k vlastnostem kladeny. Objekt si sám může pomocí setterů kontrolovat hodnoty svých proměnných a pomocí getterů dokonce jejich hodnoty dynamicky generovat. Programátor využívající objekt nemá možnost číst ani zapisovat do proměnných, ke kterým by neměl mít přímý přístup. Problém ale tkví v náročnosti výše zmíněných operací. Srovnal jsem dobu přístupu k vlastnosti přímým způsobem a přes `__get()` nebo `__set()` pomocí průměru z 100 000 čtení a zápisu. Výsledek testu nebyl příliš optimistický, neboť přístup přes univerzální getter a setter byl časově přibližně 10x náročnější. Díky tomu není možné reálně využívat tento princip přístupu u velkého množství tříd a jejich vlastností. V ROAD frameworku je tento princip použit pouze v několika málo případech, ve kterých je to vyžadováno kvůli potřebě vykonat důležitou akci při zapsání hodnoty do vlastnosti.

TComponent

Třída `TComponent` je základní třídou pro všechny komponenty v celém frameworku. Definuje princip tvorby stromu, zpracování vstupu uživatele a jeho interakci s databází. Třída obsahuje vlastnosti a metody určené pro:

- práci se stromem komponent:
  `components`, `parent`, `addComponent()`, `delete()`, `findComponent()`
- načítání a validaci hodnot odeslaných uživatelem prostřednictví formuláře:
  `loadInput()`, `validate()` a událost `onValidate`
- načítání a ukládání hodnot databázového záznamu:
  `setDataset()`, `loadFromDataset()`, `saveToDataset()`
- a samotné renderování komponenty:
  `visible`, `setRenderer()`, `getRenderer()`, `render()`

Podrobnější užití třídy `TComponent` a jejích potomků bude popsáno v kapitole 4.3.

TControl

`TControl` je společnou třídou pro většinu ovládacích prvků na stránce. Rozšiřuje `TComponent` o vlastnosti určující vizuální atributy ovládacího prvku:

`width`, `height`, `cssStyle`, `visible`, `caption`, `title`, `cssClass`, `enabled`, `readOnly`, `dynamic`, `focused`

4.1.2 Struktura souborů

Programátor využívající framework se bez znalosti adresářové struktury frameworku objede pouze do té doby, než bude mít potřebu detailněji prozkoumat kód jeho jednotlivých komponent a funkcí. Proto přikládám alespoň jednoduchý diagram znázorňující adresářovou strukturu frameworku v podobě stromu.

diagram č. 6 - **Adresářová struktura ROAD frameworku**

/	kořenový adresář celé webové aplikace
road	hlavní adresář ROAD frameworku
core	soubory tříd, konstant a soubor jádra frameworku
drivers	soubory především databázových ovladačů
lang	soubory s jazykovými konstantami
lib	knihovny třetích stran využívané frameworkem
log	adresář pro ukládání logovacích souborů
output	obsahuje adresáře pro jednotlivé render-sety
full	render-set pro normální výstup
mobile	render-set pro mobilní výstup
…	
template	obsahuje uživatelské šablony pro jednotlivé render-sety
full	šablona pro normální výstup
mobile	šablona pro mobilní výstup
…	
utils	soubory s podpůrnými funkcemi

Jak je vidět z diagramu, celý framework je obsažen v jednom adresáři, který je umístěn přímo v kořenovém adresáři webové aplikace. ROAD framework si sám zjišťuje, jestli aplikace operuje přímo v kořenovém adresáři webového serveru, nebo v nějakém jeho podadresáři a přizpůsobuje tomu všechny generované URL. Díky tomu můžeme ROAD framework nasadit i jen do části webové prezentace, případně na servery, u kterých nemáme přístup do kořenového adresáře.

Vzhledem k tomu, že framework je stále ve vývoji, je složen z mnoha menších souborů. Tím se orientace v něm stává jednodušší a hledání chyby mnohem rychlejší. Poté, co bude framework odladěn, se chystám uvolnit minimalizovanou verzi bez komentářů a dalších bezvýznamných znaků. Tato verze se bude skládat z mnohem méně souborů a bude určena pro nasazení do produkčních systémů.

4.2 Požadavky na systém

ROAD framework je určen pro aplikace vyvíjené v jazyce PHP. Protože framework intenzivně využívá podporu OOP, je vyžadováno PHP ve verzi 5. K jeho provozu je tedy potřeba webový server s podporou tohoto skriptovacího jazyka. Typ webového serveru by neměl být důležitý, protože ROAD by měl běžet stejně dobře jako na Apache i na Microsoft IIS. Osobně doporučuji webový server Apache s PHP verze 5.3 neboť na této konfiguraci byl framework vyvíjen. Dále byl framework testován na PHP 5.2. Pro ROAD není rozhodující, jestli je PHP zavedeno jako modul webového serveru nebo běží v režimu CGI.

Vyžadovaná rozšíření PHP:

- `iconv` práce s řetězci v libovolném kódování a jejich převod do jiného
- `mbstring` pro práci s řetězci ve více bytovém kódování (UTF-8, UTF-16)
- `json` generování JSON pro zpracování JavaScriptem (protože tento modul podporuje pouze UTF-8, plánuji ho nahradit vlastní obecnou funkcí)

Databázový server není pro běh frameworku vyžadován, ale značné množství jeho funkcí je na databázi navázáno a byla by škoda je nevyužít. Jako databázový server lze použít jakýkoliv server, ke kterému existuje ovladač pro ROAD. Prozatím existuje spolehlivý ovladač pro MySQL, využívající `mysqli` rozšíření pro PHP a ve fázi ladění je i ovladač pro PostgreSQL.

4.3 Definice stromu komponent

V kapitole 3.2 jsme se bavili o stromu komponent jako o reprezentaci zanoření jednotlivých ovládacích prvků na stránce. Nyní si ukážeme jak se tento strom skládá v ROAD frameworku. Veškerý kód, který strom komponent pro danou stránku definuje, bude ve většině případů umístěn v jednom souboru. Tento soubor můžeme budeme nazývat podle kapitoly 3.3 *„soubor stránky“*. Komponenty jsou v něm definovány pouhým vytvořením příslušných objektů,

přičemž objekt komponenty jenž má být nové komponentě nadřazená, je předáván jako parametr konstruktoru.

výpis kódu č. 11 - **Tvorba stromu komponent**

```
// formulář umístěný přímo na stránku
$formular = new TForm($page);
// rámeček umístěný do formuláře
$ramecek = new TGroupBox($formular);
// textové vstupní pole v rámečku
$vstupniPole = new TEditBox($ramecek);
// tlačítko pro odeslání formuláře pod rámečkem
$tlacitko = new TSubmitButton($formular);
```

Strom komponent nadefinovaný výše, bude na obrazovce klasického webového prohlížeče vypadat následovně:

obrazovka č. 1 - **Základní formulář**

Všimněte si, že framework sám doplnil popisky komponent, podle jejich názvů. Jelikož jsme při tvorbě komponent nezadali ani tyto názvy, byly automaticky vygenerovány podle typu komponenty a jejího pořadí. Chceme-li popisky a názvy nastavit ručně, musíme změnit příslušné vlastnosti těchto komponent:

výpis kódu č. 12 - **Nastavení vlastností komponent**

```
$ramecek->caption = 'Zadejte text';
$vstupniPole->caption = 'Text';
$tlacitko->caption = 'Odeslat';

$formular->name = 'formular';
$ramecek->name = 'ramecek';
$vstupniPole->name = 'vstupniPole';
$tlacitko->name = 'tlacitko';
```

Název komponenty je důležitý pro jedinečnou identifikaci komponenty v rámci frameworku. Díky němu může framework komponentu vyhledat v případě, že je s ní potřeba provést nějaké operace. Navíc framework umožňuje programátorovi vyhledat komponentu stejným způsobem, jako by se přistupovalo k vlastnosti objektu, přičemž i v tomto případě vyhledává podle jejího názvu. Pokud je správně zadán název komponent, jsou následující zápisy ekvivalentní[12]:

výpis kódu č. 13 - **Vyhledání existující komponenty**

```
// 1) přímý přístup k objektu komponenty
$tlacitko->caption = 'Odeslat';
// 2) dohledání komponenty pomocí speciální funkce
$formular->findComponent('tlacitko',true)->caption = 'Odeslat';
// 3) přístup ke komponentě jako k vlastnosti
$formular->tlacitko->caption = 'Odeslat';
```

12 Zápisy jsou ekvivalentní co do výsledku. Bohužel co do rychlosti ekvivalentní nejsou. Viz kapitola 4.12.1

ROAD framework sice sám komponentám přiřazuje jedinečné názvy podle toho jak vznikají, ale vzhledem k tomu, že při generování složitých stránek může být jedna nebo více komponent z nějakého důvodu úmyslně vynechána, názvy komponent by pak neodpovídaly předchozímu stavu. Navíc programátor neví, jaký název byl které komponentě přiřazen a nemůže kvůli tomu využívat mechanizmus hledání komponent. Z těchto důvodů je dobré alespoň důležitým komponentám přiřazovat jedinečný název manuálně.

Pro jednoduchost zápisu je možné název komponenty předat přímo jako druhý parametr konstruktoru. Finální zkrácený zápis definice stromu by tedy vypadal následovně:

výpis kódu č. 14 - **Tvorba stromu komponent s nastavením vlastností**

```
$formular = new TForm($page,'formular');

$ramecek = new TGroupBox($formular,'ramecek');
$ramecek->caption = 'Zadejte text';

$vstupniPole = new TEditBox($ramecek,'vstupniPole');
$vstupniPole->caption = 'Text';

$tlacitko = new TSubmitButton($formular,'tlacitko');
$tlacitko->caption = 'Odeslat';
```

Výstup samozřejmě reflektuje pouze změnu popisků, neboť změnu názvu komponenty uživatel aplikace nijak přímo nepocítí:

obrazovka č. 2 - **Formulář včetně popisků**

Komponenty samozřejmě obsahují množství dalších vlastností, které modifikují nejen jejich vzhled, ale i jejich chování. Ke všem těmto vlastnostem se přistupuje obdobně jako k vlastnosti `caption`.

4.4 Práce s událostmi

V předchozí kapitole jsme si uvedli, jak se snadno nadefinuje strom komponent tak, aby vytvořil jednoduchý formulář. Nyní si ukážeme jak ovládat chování formuláře pomocí obsluhy událostí. Nejprve je potřeba nadefinovat funkci, která bude událost obsluhovat a pak tuto funkci k události přiřadit. Vezměme si formulář z předchozí kapitoly a řekněme, že chceme reagovat na kliknutí na tlačítko „Odeslat“. Nadefinujeme tedy funkci, která nám o tom dá vědět prostřednictvím výpisu textu. V reálné aplikaci by tento postup byl špatný, neboť výpis textů by měl být svěřen vždy pouze jednotlivým renderům.

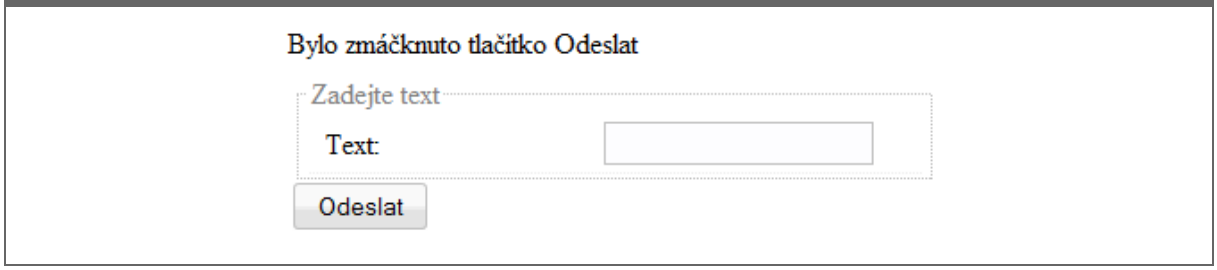
výpis kódu č. 15 - **Jednoduchá událost**

```
function odeslano($sender) {
  echo 'Bylo zmáčknuto tlačítko '.$sender->caption;
}

$tlacitko->onClick = 'odeslano';
```

Všimněte si, že každé události je vždy předán jako první parametr původce této události. Další parametry se různí podle toho o jakou událost a jakou komponentu jde. Po zmáčknutí tlačítka na formuláři by měl uživatel vidět následující výstup:

obrazovka č. 3 - **Reakce na událost**



Důležité je, že k vyvolání události `onClick` dochází až potom, co jsou do komponent načteny všechny hodnoty zadané uživatelem. Programátor tak může v obsluze události tyto hodnoty podle potřeb zpracovat. Událost `onClick` samozřejmě není jedinou událostí, kterou může programátor využít, a komponenta `TSubmitButton` není jediná komponenta, která nějaké události vyvolává. S popisem všech komponent a událostí bychom se ale do rozsahu diplomové práce určitě nevešli. Pro detailnější seznámení s událostmi proto doporučuji prostudovat přílohu ukazující tvorbu vzorové aplikace.

Třída TPanel

Protože ROAD framework se pyšní tím, že je objektově orientovaný, nemůže být spokojený s událostmi obsluhovanými globálními funkcemi. Proto framework nabízí třídu `TPanel`. Třída `TPanel` je abstraktní třídou, která neslouží pro přímé využití ve stromu komponent, ale nabízí programátorovi vytvořit a použít ve stromu její potomky. `TPanel` je speciální proto, že všechny události, v něm a v jeho pod-komponentách vyvolané, jsou směrovány do jeho metod namísto do globálních funkcí. Umístíme-li tedy celý formulář z našeho příkladu do potomka třídy `TPanel`, bude při stisknutí tlačítka namísto globální funkce `odeslano()` volána metoda `$panel->odeslano()`.

výpis kódu č. 16 - **Využití třídy TPanel**

```
class TVlastniPanel extends TPanel {
  function odeslano($sender) {
    echo 'Uživatel napsal: "'.$this->vstupniPole->value.'"';
  }
}

$panel = new TVlastniPanel($page,'panel');

$formular = new TForm($panel,'formular');

...

$tlacitko->onClick = 'odeslano';
```

Použitím potomka třídy `TPanel`, lze stránku rozdělit na několik oddělených logických celků, které spolu nekolidují názvy. Obsluha událostí je odstíněna od globálních funkcí a díky tomu, že se při ní pohybujeme v prostředí konkrétní části stránky, snadněji se dostaneme k požadovaným komponentám (viz přístup ke komponentě *vstupniPole*). Jednotlivé panely se navíc stávají znovupoužitelnými fragmenty aplikace. Například formulář pro přihlášení uživatele pak můžeme bez problému opakovat na každé stránce.

Kromě obsluhy událostí je samozřejmě možné přímo do třídy panelu vložit i tvorbu stromu jeho pod-komponent. Kód tvorby stromu by v tomto případě patřil nejspíše do konstruktoru třídy. S jistou dávkou šikovnosti tak programátor může definovat fragment stránky, z něhož může vytvořit více instancí i na jediné stránce.

Zkrácený zápis obsluhy události

S rostoucím množstvím různých událostí ve frameworku roste i počet událostí, které lze obsloužit jedním nebo dvěma příkazy. Definovat pro obsluhu takovéto události globální funkci, nebo dokonce metodu třídy `TPanel`, by bylo z hlediska objemu potřebného kódu poněkud neefektivní. Díky tomu, že je PHP interpretovaný jazyk, mohl jsem do ROAD frameworku přidat podporu tzv. inline obsluhy událostí. Jedná se vlastně o to, že místo názvu funkce nebo metody přiřadí programátor události přímo kód k vykonání. Nastane-li daná událost, je zavolána funkce `eval()`, která daný kód vyvolá.

Aby byl tento způsob jednoznačně odlišen od standardního způsobu, musí programátor jako první znak kódu umístit znak >. Díky tomuto znaku framework pozná, že se jedná o kód k vykonání a ne název metody nebo funkce.

výpis kódu č. 17 - **Zkrácený zápis obsluhy události**

```
$tlacitko->onClick = 'echo "Bylo zmáčknuto tlačítko ".$this->caption;';
```

Všimněte si, že na rozdíl od předchozích zápisů, nepřistupujeme k instanci tlačítka přes proměnnou `$sender` ale přímo přes `$this`. Proměnná `$this` je nám k dispozici proto, že kód je vykonávaný přímo v rámci komponenty `TSubmitButton`.

4.5 Jádro

Jádro frameworku je tvořeno především třídou `TRoad`, která dědí vlastnosti a metody třídy `TComponent`, a souborem `road.php`, který se stará o inicializaci celého frameworku.

4.5.1 Třída TRoad

`TRoad` je potomkem `TComponent` především proto, že `TComponent` definuje vlastnosti a metody pro práci s pod-komponentami a `TRoad` díky tomu může sloužit jako komponenta nejvyšší úrovně. Aplikace by při svém běhu měla obsahovat pouze jednu instanci třídy `TRoad` uloženou v globální proměnné `$road`. Třída `TRoad` obsahuje mechanismy pro detekci webového prohlížeče, udržování sezení, kontrolu přihlášení a práv uživatele a také systém pro vyvolávání akcí.

Detekce prohlížeče

Detekce prohlížeče probíhá primárně pomocí funkce `get_browser()`. Tato funkce zjišťuje podle HTTP hlavičky *User-Agent*, jaký prohlížeč uživatel používá, a pomocí jakési databáze webových prohlížečů, uložené v souboru `browsecap.ini`,[13] rozhoduje, které technologie daný prohlížeč podporuje a jestli se jedná o desktopový nebo mobilní prohlížeč. Jelikož je tato databáze obsáhlá, trvá detekce prohlížeče poměrně dlouhou dobu na to, aby byla prováděna při každém generování stránky. Z toho důvodu framework detekovaný prohlížeč ukládá v rámci sezení, detekce tak probíhá pouze při prvním přístupu ke stránce.

Princip detekce prohlížeče podle hlavičky *User-Agent* je dostačující pouze do té doby, dokud uživatel prohlížeče nezmění jeho nastavení. Když uživatel například vypne podporu JavaScriptu, aplikace by o tom nebyla nijak informována a předkládala by uživateli verzi stránek, která JavaScript vyžaduje. Z toho důvodu bylo potřeba umožnit skutečnou detekci aktuální konfigurace prohlížeče. Průběh detekce, kterou lze v případě potřeby v konfiguračním souboru vypnout, je následující:

- prohlížeč se poprvé dotáže na stránku aplikace
- aplikace vrátí prázdnou stránku, ve které je obsažen:
  - JavaScript, který ihned po zjištění aktuálního nastavení podpory Cookies a AJAXu, automaticky předá informace o konfiguraci aplikaci a obnoví stránku
  - alternativní způsob obnovení stránky pomocí meta hlavičky refresh
- aplikace si konfiguraci prohlížeče uloží do proměnné sezení a při příštích žádostech o stránky je rovnou generuje v příslušné formě

13 Vzhledem k rychlému rozvoji v oblasti webových prohlížečů je potřeba mít `browsecap.ini` neustále aktualizovaný.

Programátor může v konfiguračním souboru nastavit, aby se místo prázdné stránky, před detekcí odesílala stránka s minimálními požadavky. Tedy stránka bez podpory JavaScriptu a AJAXu. Tím lze zajistit, že se stránka správně zobrazí i v prohlížečích se zakázaným automatickým přesměrováním. V opačném případě by se uživateli zobrazila zpráva o skutečnosti, že jeho prohlížeč nepodporuje automatické přesměrování a je potřeba kliknout na přiložený odkaz.

Programátorovi je detekovaná konfigurace prohlížeče zpřístupněna pomocí speciální třídy `TBrowser` jejíž instance je uložena v `$road->browser`.

Autentizace a autorizace

Bezpečnost aplikací je velmi důležitým faktorem a nejprostší bezpečnostní dírou je bezesporu opomenutí omezení přístup k určité funkci aplikace jen vybraným uživatelům. Aby se programátor této chyby dopouštěl co nejméně, nabízí ROAD framework propracovaný systém přihlašování uživatelů a správu jejich uživatelských práv. Mechanizmus autentizace a autorizace je skrytý ve třídě `TSecurity` jejíž instance je uložena v `$road->security`.

ROAD framework udržuje přihlášení uživatele buď pomocí HTTP autentizace, nebo pomocí sezení. Jelikož ROAD framework je schopen pomocí modifikace URL udržet sezení i bez podpory Cookies, je použití druhé možnosti zcela bezproblémové. Mezi oběma možnostmi se dá přepínat v konfiguračním souboru.

HTTP autentizace

Framework při přístupu na stránku vrací prohlížeči status *401 Unauthorized*, který ho donutí požádat uživatele o přihlašovací údaje. Dotaz na jakoukoliv stránku aplikace je pak doprovázen odesláním těchto přihlašovacích údajů. Framework jako reakci na takovýto dotaz vyvolá událost `onHttpLogin`, které jsou jako parametry předány přihlašovací údaje. Programátor v obsluze této události uživatele fyzicky přihlásí voláním metody `login()` viz níže.

Session autentizace

Autentizace prostřednictvím proměnných sezení, je realizována prostřednictvím metody `login()`. Programátor musí vytvořit a uživateli zpřístupnit formulář pro zadání přihlašovacích údajů. Po odeslání tohoto formuláře programátor vyhledá příslušného uživatele v databázi a pokud ho nalezne, zavolá metodu `login()`, které může předat dva parametry:

- libovolnou hodnotu reprezentující daného uživatele (řetězec, pole, objekt); na základě této hodnoty pak bude programátor uživatele identifikovat v průběhu jeho práce s aplikací
- pole nebo čárkami oddělený seznam práv daného uživatele

Hodnoty předané metodou `login()` jsou automaticky uloženy do proměnné sezení a při příštím přístupu k aplikaci jsou zas automaticky načteny do třídy

`TSecurity`.

Kontrola uživatelských práv

V průběhu generování stránky má programátor dvě možnosti jak kontrolovat oprávněnost uživatele k určitým úkonům:

- metodou `isAllowed()` zjistí, zda má uživatel dostatečné oprávnění
- metodou `requireRight()` zajistí, že generovaní stránky bude pokračovat pouze tehdy, bude-li mít uživatel dostatečné oprávnění; v opačném případě bude vyvolána událost `onDenied`, ve které může programátor uživatele vyrozumět, nebo to jednoduchou hláškou udělá přímo framework

Jednotlivá oprávnění mohou být dána:

- **celočíselnou hodnotou:** uživatel je k akci oprávněn tehdy, je-li hodnota vyžadovaného oprávnění nižší nebo rovna než hodnota jeho oprávnění
- **pseudo-adresářovou strukturou:** uživatel je k akci oprávněn tehdy, existuje-li „adresář“ vyžadovaného oprávnění v „adresářové struktuře“ dané jemu přidělenými oprávněními

Příklad vyhodnocování oprávnění pomocí pseudo-adresářové struktury

Jsou-li oprávnění uživatele tato:

`/clanky/prohlednout`,`/komentar/pridat`,`/komentar/smazat`

pak je uživatel oprávněn například k:

`/clanky` a `/komentar/pridat`

ale není oprávněn k:

`/clanky/upravit` a `/uzivatele`

Systém akcí

Doposud jsme se bavili pouze o programování v rámci serveru, tedy vytváření a úpravu souborů `.php`. Nyní si ukážeme jak lze běh aplikace ovládat přímo z prohlížeče a to dokonce bez použití JavaScriptu. Programovat budeme pomocí adresního řádku.

Ve standardních webových aplikacích je pomocí adresního řádku možné předat omezené množství předem definovaných parametrů, které ovlivňují běh aplikace a generování vrácené stránky. Tento princip ale není příliš flexibilní, a aplikace navíc musí vždy testovat existenci všech parametrů a procedurálně je zpracovávat.

ROAD framework proto přichází s tzv. systémem akcí. Představte si, že při volání stránky můžete pomocí adresního řádku přímo zavolat metodu některé z komponent a dokonce jí předat nějaké parametry. ROAD to umožňuje prostřednictvím speciálního parametru `rAction`. Tento parametr může být buď pole příkazů nebo samostatný příkaz. Pro rychlé pochopení principu si ukážeme

příklad.

Příklad

Voláním naší ukázkové stránky adresou:
`/stranka.php?rAction=vstupniPole-render()`
získáte stránku, na které bude vidět pouze textové pole, protože metoda `render()` vykreslí tuto komponentu a pak generování stránky skončí.

Právě metodu `render()` volá pomocí akcí komponenta, když je jí pomocí JavaScriptu nakázáno, aby se obnovila. Komponenta požádá aplikaci o stejnou stránku, na které se právě nachází, ale řekne jí, ať vyrenderuje pouze tu konkrétní komponentu. Díky tomu se může HTML kód všech dynamických komponent generovat na straně serveru, aniž by musel programátor jakkoliv přizpůsobovat aplikaci. Velká výhoda toho, že jsou komponenty generovány výhradně na straně serveru je ta, že bude nezměněný JavaScript bez problému operovat s programátorem vytvořenými nebo přizpůsobenými komponentami.

Všimněte si, že syntaxe příkazu je podobná syntaxi PHP s tím rozdílem, že se vynechává dolar uvozující proměnnou a oddělovač objektu a metody je redukován na pouhé znaménko mínus. První část příkazu tedy udává název komponenty, jejíž metody mají být volány (pokud tato část chybí, budou volány metody na objektu `$page`, tedy na celé stránce). Část za znamínkem mínus udává metodu, která má být být volána. Metodě lze předat i textové a numerické parametry. Jelikož parametr `rAction` může být pole, je možné za sebou řadit i více příkazů. Adresa stránky by pak vypadala následovně:

```
/stranka.php?rAction[]=komp1-metoda()&rAction[]=komp2-metoda()
```

Pro zjednodušení zápisu je povolená i zkrácená forma uvozená dvojtečkou:

```
/stranka.php?:vstupniPole-render()
```

Vyvolávat akce navíc lze nejen přes GET dotaz, ale i přes POST. Příkaz akce tak není problém zakomponovat do polí formuláře.

Možná vás napadne, jestli není tento systém poněkud nebezpečný, když takhle bez ostychu vystavuje metody komponent útokům. I k tomuto faktu bylo samozřejmě přihlédnuto a proto nejsou systémem akcí přístupné všechny metody komponent, ale jen ty, které jsou k tomu předem určené. Zda je metoda přístupná přes akce nebo ne, určuje chráněná vlastnost každé komponenty s názvem `actions`. Jedná se o asociativní pole, jehož klíči jsou názvy povolených metod a hodnotami jsou konstanty určující, kdy v průběhu generování stránky má být metoda zavolána.

Jak framework během generování stránky provádí mnoho operací, v každém okamžiku je stav aplikace odlišný. Každá metoda volaná prostřednictvím akcí tedy potřebuje být zavolána v jiném okamžiku. Například metoda `render()` vyžaduje, aby byly před jejím voláním dokončeny všechny předchozí úkony, tedy aby byly načteny záznamy z databáze a případně načten vstup uživatele.

Na druhou stranu metoda `insert()` u databázového propojení (komponenta `TTableLink`) musí být volána ještě předtím, než dojde k načtení z databáze, protože jejím úkolem je vytvořit nový záznam, který bude editován. Volby okamžiku, kdy mají být metody volány jsou následující:

- `ACTION_BEFORE_INIT` před inicializací komponent metodou `init()`
- `ACTION_BEFORE_INPUT` před načtením hodnot z formuláře
- `ACTION_AFTER_INPUT` po načtením hodnot z formuláře

4.6 Inicializační soubor

Cesta k inicializačnímu souboru je `/road/core/road.php` a tento soubor slouží k inicializaci celého frameworku. Jeho úkolem je:

- detekovat platformu serveru, na kterém aplikace běží
- definovat důležité konstanty frameworku jako: `CFG_ROAD_DIR`, `CFG_ROOT_DIR`, `CFG_WEB_ROOT`, `CFG_SITE` atd.
- načíst všechny soubory frameworku do paměti
- nastavit ovladače chyb (ROAD framework používá vlastní způsob hlášení a logování chyb a varování)
- zjistit preferovaný jazyk uživatele a načíst příslušný jazykový soubor
- zjistit podporovaná kódování
- a především vytvořit instanci třídy `TRoad` (jádro frameworku) a `TPage` (aktuální stránka)

Soubor `road.php` je vlastně hlavním programem celého frameworku a jako takový by neměl být programátorem vůbec modifikován.

4.7 Načtení a spuštění frameworku

Doposud jsme se bavili pouze o tom, jak framework pracuje a jak se má pracovat s ním, ale stále jsme si neukázali, jak vlastně framework integrovat do aplikace a jak ho spustit. Jak již bylo řečeno v kapitole 3.3, ROAD framework bude do aplikace integrován na úrovni souborů stránek. V každém souboru reprezentujícím stránku tak bude potřeba framework načíst a spustit.

4.7.1 Zaváděcí soubor

Načtení frameworku by v ideálním případě mělo probíhat přes zaváděcí soubor `/road/run.php`. Obsah toho souboru je více-méně pod kontrolou programátora. Jediné dva řádky, které tento soubor musí obsahovat jsou:

výpis kódu č. 18 - **Neměnný obsah zaváděcího souboru**

```
// načtení inicializačního souboru
require_once('core/road.php');

// inicializace jádra
$road->init();
```

Možná se ptáte, proč není inicializace jádra spuštěna již v rámci inicializačního souboru. Je to proto, že během inicializace mohou být vyvolány některé události, které vyžadují programátorovu interakci. Ten má právě v zaváděcím souboru možnost mezi načtením inicializačního souboru a spuštěním inicializace jádra definovat funkce obsluhující tyto události.

výpis kódu č. 19 - **Obsluha událostí při inicializaci jádra**

```
require_once('core/road.php');

function checkLogin(TSecurity $sender) {
  // například kontrola doby platnosti přihlášení
}
$road->security->onCheckLogin = 'checkLogin';

$road->init();
```

Dalším důležitým krokem, který by měl být v rámci zaváděcího souboru realizován, je vytvoření připojení k databázi. Díky tomu, že je připojení k databázi vytvořeno již v zaváděcím souboru, budou mít přístup k databázi všechny stránky, které tento soubor načtou, tedy všechny stránky aplikace.

výpis kódu č. 20 - **Načtení zaváděcího souboru ze souboru stránky**

```
<?php
  require_once('road/run.php');

  ...

?>
```

Jak vidíte, použití zaváděcího souboru je realizováno jen jeho prostým načtením do paměti.

4.7.2 Spuštění frameworku

Skutečnost, že jsme framework načetli do paměti a inicializovali, ale nestačí k tomu, aby byla vygenerována naše požadovaná stránka. V rámci souboru stránky nás čeká především ještě definice stromu komponent tak, jak byla popsaná v kapitole 4.3, a poté ještě samotné spuštění frameworku. Spuštění frameworku se provádí voláním metody jádra `run()`. Po zavolání této metody jsou komponenty inicializovány, vykonány akce (pokud byly nějaké zadány), načteny hodnoty z databáze a z odeslaného formuláře, vyvolány příslušné metody a jako poslední jsou komponenty vyrenderovány. S dokončením všech těchto operací je běh skriptu ukončen.

výpis kódu č. 21 - **Spuštění frameworku v souboru stránky**

```
<?php
  require_once('road/run.php');

  ... // definice stromu komponent

  $road->run();
?>
```

Alternativou k volání metody `run()` je volání metod `start()` a `stop()`. V metodě `start()` framework vykoná všechny úkony, které vykonává metoda `run()`, až na renderování komponent a ukončení běhu skriptu. Metoda pouze odešle hlavičky stránky a vyrenderuje záhlaví. Pak je na programátorovi aby vygeneroval obsah stránky buď pomocí metod `render()` u jednotlivých komponent, nebo přímým výpisem HTML kódu[14]. Po vygenerování obsahu stránky musí programátor zavolat ještě metodu `stop()`, která vygeneruje zápatí stránky a ukončí běh skriptu.

Jak vidíte, integrace frameworku do aplikace je jednoduchou záležitostí a díky metodám `start()` a `stop()` lze framework integrovat i do stránek, které již existují a generují obsah stránky přímým výpisem HTML kódu. Pokud bude ale programátor dodržovat principy MVC architektury a vystačí si s metodou `run()`, pak bude odměněn v podobě absolutní nezávislosti na platformě uživatele, která je tématem především následující kapitoly.

4.8 Renderování stránky

Renderování stránky probíhá plně dle principu, který byl definovaný v kapitole 3.5.2. Komponenty jsou renderovány pomocí objektů zvaných rendery, přičemž všechny rendery jsou potomky třídy `TCustomRenderer`. Rendery jsou seskupeny do render-setů, z nichž každý je určen pro jednu koncovou platformu. Oproti konceptuálnímu návrhu z kapitoly 3.5.2 přibyl ještě objekt *output*. Ten slouží jako vrstva mezi výstupním bufferem a rendery. Rendery skrze tuto vrstvu generují výstupní kód komponent. Každý render-set pak má svůj vlastní objekt *output*.

4.8.1 Třída TCustomOutput

Instance potomků třídy `TCustomOutput` reprezentují výše zmíněné objekty typu *output*. Obecně řečeno, potomci této třídy zprostředkovávají renderům zápis do výstupního bufferu webového serveru. Kromě toho se starají o nastavení správného typu MIME pro odesílaná data a také o převod jejich kódování do prohlížečem podporovaného. Objekt *output* navíc slouží jako registr jednotlivých renderů v daném render-setu. S jistou dávkou nadsázky můžeme vlastně říci, že potomci třídy `TCustomOutput` reprezentují jednotlivé render-sety. Pro jistotu si ještě jako příklad potomka této třídy uvedeme třídu pro standardní

14 Tento způsob generování není doporučován, protože obchází renderovací objekty, čímž porušuje MVC architekturu a ruší tak nezávislost aplikace na koncovém zařízení uživatele. Aby byla nezávislost alespoň trochu zachována, programátor by musel pomocí podmínek rozlišovat, pro které zařízení má zrovna výstup generovat.

výstup ve formě XHTML značek.

TXhtmlOutput

Třída `TXHtmlOutput` nabízí metody pro generování XHTML kódu jako: `rCode()`, `rText()`, `rTagOpen()`, `rTagClose()`, `rTagText()`, `rLinkText()` atd. Většina těchto metod očekává název tagu elementu a asociativní pole atributů. Výhody toho, že je veškeré generování výstupu směrováno přes tuto třídu jsou:

- texty mohou být převedeny do prohlížečem podporovaného kódování
- všechny texty i hodnoty atributů jsou automaticky ošetřeny proti speciálním znakům
- třída automaticky kontroluje validitu generovaného kódu a případně vyrozumí programátora přes upozornění[15]
- třída automaticky odsazuje vnořené elementy, aby byl generovaný výstup lépe čitelný pro vývojáře[16]
- generovaný výstup nemusí být rovnou vypisován do výstupního bufferu, ale třída ho může:
  - před odesláním zkomprimovat metodou *gzip* nebo *deflate*
  - uložit (celý nebo jen část) do vyrovnávací paměti a později ho načíst bez opakovaného generování[17]

XHTML Rendery všech komponent pak místo vypisování kódu pomocí příkazu **`echo`** nebo funkce `print()`, volají metody třídy `TXhtmlOutput`.

4.8.2 Třída TCustomRenderer

Jestli-že potomci třídy `TCustomOutput` reprezentují jednotlivé rendery, pak potomci třídy `TCustomRenderer` reprezentují jednotlivé rendery. `TCustomRender` je jednoduchá třída abstraktně definující několik málo metod, z nichž nejdůležitější je metoda `render()`. Tuto metodu volá framework, když je třeba vygenerovat výstup komponenty, která je této metodě předána jako parametr. Přímo ve třídě `TCustomRenderer` je metoda `render()` definována jako pouhý rozcestník, ve kterém je voláno renderování záhlaví komponenty (`renderHeader()`), jejích pod-komponent (`renderChildren()`) a nakonec zápatí komponenty (`renderFooter()`). Jedině metoda pro renderování pod-komponent je přímo v této třídě implementována, ostatní očekávají, že budou implementovány potomky třídy.

Každý potomek třídy `TCustomRenderer` má tedy za úkol se postarat o renderování jednoho typu komponenty pro jedno koncové zařízení. Celkový

15 Po odladění aplikace je vhodné kontrola validity a urychlit tak trochu generování stránky.

16 Po odladění aplikace je vhodné vypnout i automatické odsazování, neboť se tím znatelně zmenší objem přenášených dat.

17 Podpora vyrovnávací paměti prozatím není implementována, ale plánuji ji do budoucích verzí zahrnout.

počet potřebných renderů bude roven přibližně počtu komponent krát počet koncových zařízení. Každý render je potřeba zaregistrovat u příslušného potomka třídy `TCustomOutput` voláním metody `registerRenderer()`, která očekává dva parametry. Prvním je instance třídy `TCustomRenderer` a druhým je název třídy komponenty, pro kterou je render určen. Při registraci platí, že každý render, který umí renderovat nějakou třídu, umí renderovat i její potomky. K renderování však bude použit pouze pokud nebude nalezen žádný způsobilejší render (tedy bližší k dané komponentě).

4.8.3 Výběr render-setu

Je vidět, že architektura render-setů je poměrně propracovaná, ale sebevíc propracované render-sety nebudou generovat správný výstup, když by byl vybrán nevhodný render-set. Framework vybírá render-set primárně podle detekovaného prohlížeče. V souboru `browser.ini` (viz kapitola 4.5.1), je uchována i informace, zda je daný prohlížeč provozován na mobilním zařízení nebo na osobním počítači. Díky tomu může framework rozhodnout, zda použít render-set pro mobilní zařízení nebo standardní render-set pro osobní počítače. Standardní distribuce ROAD frameworku ale obsahuje i render-set pro XML API[18] a také render-set pro serializaci do formátu JSON, který je využíván JavaScriptovými knihovnami frameworku. Jak lze tedy vybrat jiný render-set než je implicitní (ať už desktopový nebo mobilní)? Framework reaguje na GET (resp. POST) parametr `rOutput` a podle jeho hodnoty nastaví odpovídající render-set jako aktivní.

Parametr `rOutput` může nabývat libovolných hodnot, pokud existuje odpovídající render-set. Render-sety jsou umístěny v adresáři `/road/output`, kde jsou reprezentovány podadresáři pojmenovanými podle názvu render-setu. Ve standardní distribuci frameworku jsou obsaženy tyto render-sety (tzn. existují tyto adresáře):

- `full` pro klasický prohlížeč osobního počítače
- `mobile` pro mobilní zařízení
- `xml` pro XML API
- `json` pro podporu dynamických komponent skrz JavaScript a AJAX

V rámci těchto adresářů se nachází soubor `render-set.php`, který v sobě obsahuje definici celého render-setu. Tedy definici všech potomků třídy `TCustomRenderer` a také potomka třídy `TCustomOutput`.

Parametrem `rOutput` lze zvolit aktivní render-set pro aktuální stránku, kdežto parametrem `rDefaultOutput` bude zvolen aktivní render-set pro celé sezení. Díky parametru `rDefaultOutput` může programátor na stránku jednoduše umístit odkaz pro přepínání mezi mobilní a plnou verzí stránky.

18 Render-set pro XML API generuje strojově čitelnou reprezentaci stromu komponent pro aplikace třetích stran.

4.8.4 Přizpůsobení výstupu

Principem ROAD frameworku je pokud možno nechat generování výstupního HTML přímo na jednotlivých renderech a framework obsahuje všechny potřebné rendery pro komponenty, které programátorům nabízí k použití. Může se zdát, že kromě CSS stylů programátorovi nezbývají žádné prostředky, jak si vzhled aplikace přizpůsobit. ROAD framework ale nabízí podporu vlastních renderů. Každý programátor tak může přizpůsobit rendery jednotlivých komponent svým potřebám.

Přizpůsobení renderů a vytvoření nových probíhá obdobně jako u definice standardních renderů v jednotlivých render-setech. Programátor vytvoří potomka třídy `TCustomRenderer` a poté voláním metody `registerRenderer()`, přiřadí instanci této třídy k určité komponentě. Touto registrací je přepsána původní vazba na standardní render a při generování stránky bude použit tento nový. Stejně tak probíhá i registrace renderů vytvořených pro renderování nově vytvořených uživatelských komponent.

Obecné uživatelské rendery by měly být uloženy v souboru `/road/template/`**`render-set`**`/template.php`, kde **`render-set`** je název render-setu, tedy `full`, `mobile`, `xml`, `json` nebo jiné, pokud jsou definované.

Příklad

Chceme-li aplikaci přidat pěkné záhlaví, hlavní nabídku a zápatí s copyrightem, musíme přepsat render komponenty `TPage`.
V souboru `/road/template/full/template.php` vytvoříme potomka třídy `TXhtmlPage` a nazveme ho například `TMyXhtmlPage`. Přepíšeme metody `renderHeader()` a `renderFooter()` tak, aby se v nich generovalo záhlaví a zápatí stránek, a nový render zaregistrujeme voláním metody `registerRenderer()`.
Souboru `template.php` by tedy vypadal následovně:

výpis kódu č. 22 - **Přizpůsobení vzhledu stránky**

```
<?php

  class TMyXhtmlPage extends TXHTMLPage {

    public function renderHeader(TComponent $component,$flags = null)  {
      parent::renderHeader($component); // vygenerovat <head> element

      // vygenerování menu
      $this->output->rTagOpen('div',array('class'=>'menu'));

      ...

    }
    public function renderFooter(TComponent $component,$flags = null)  {
      // zápatí s copyrightem
      $this->output->rTagCode('div','2011 &copy; Firma &amp; spol. s.r.o.',array('class' =>
'footer'));

      ...

      parent::renderFooter($component); // ukončit <body> a <html> tag
```

```
    }
    public function __construct($road) {
      parent::__construct($road);

      // přilinkovat soubor style.css ve stejném adresáři jako je tento soubor
      $this->metaAddStyle('/'.relativePath(dirname(__FILE__)).'/style.css');
    }
  }

  // zaregistrovat render do aktuálního render-setu
  $this->output->registerRenderer(new TMyXhtmlPage($this),'TPage');
?>
```

Proměnná `$this`, je při registraci renderu použitá proto, že celý soubor `template.php`, je spouštěn v metodě třídy `TRoad`. `$this` tedy odkazuje na instanci této třídy, tedy na jádro frameworku.

4.9 Konverze formátů

Další nepříjemnou a stále se opakující akcí je převod hodnot mezi jednotlivými formáty. To se týká hlavně datových typů jako *datum-čas* a *číslo*. Pro každého datový typ budeme při vývoji aplikace potřebovat přinejmenším 4 formáty:

- **SQL** formát – formát, ve kterém jsou hodnoty vraceny z databáze a ve kterém jsou očekávány v rámci příkazů `INSERT` a `UPDATE`
- **nativní PHP** formát – čistý datový typ určený pro zpracování v kódu (`timestamp`, `integer`, `float` atd.)
- formát **pro zobrazení** – formát v jakém budou data prezentována v prohlížeči
- formát **pro úpravu** – od uživatele se očekává že zadá data právě v tomto formátu (takto naformátovaná data budou obsahem vstupních polí)

Používání nízkoúrovňových funkcí jako `date()`, `strtotime()`, `strftime()` nebo `formatnumber()` bezpochyby vede k radikálnímu snížení flexibility aplikace. Chceme-li například k aplikaci dodatečně přidat další jazykovou mutaci, musíme projít všechny soubory, ve kterých jsou tyto funkce volány a parametry funkcí pozměnit aby odpovídali danému jazyku.

Řešením může být vytvoření funkcí pro převod mezi formáty, které nebudou očekávat parametry závisející na dané jazykové mutaci. Vezmeme-li ale v potaz počty datových typů a jejich formátů, zjistíme, že množství převodních funkcí by bylo neúnosné ($pocetDatovychTypu \cdot pocetFormatu^2$). Z toho důvodu jsem se rozhodl soustředit veškeré možné převody v rámci jednoho datového typu do samostatných tříd. Pro většinu standardních datových typů a jejich podruhů je tak v ROAD frameworku definována samostatná třída obsluhující převod mezi jednotlivými formáty těchto datových typů. Rozhraní pro práci s těmito třídami je definované v abstraktní třídě `TCustomValue`.

Třída `TCustomValue` neslouží jen pro převod formátů, ale i pro uchování hodnoty. Díky tomu můžeme silně zredukovat počet převodních funkcí. Pro převod

z jednoho formátu do druhého totiž programátor nejdříve zavolá metodu pro nastavení hodnoty a poté metodu pro její získání. První metoda očekává hodnotu ve zdrojovém formátu a druhá metoda jí vrací v cílovém formátu. Třída `TCustomValue` tedy nabízí 8 metod pro 4 formáty:

- SQL formát – `setAsSql()`, `getAsSql()`
- nativní PHP formát – `setRaw()`, `getRaw()`
- pro zobrazení – `setAsDisplay()`, `getAsDisplay()`
- pro úpravu – `setAsInput()`, `getAsInput()`

výpis kódu č. 23 - **Převod formátů pomocí TCustomValue**

```
$hodnota = new TDateTime();
$hodnota->setAsSql('2011-04-02 12:32:01');
echo $hodnota->getAsDisplay();
// vypíše 02.04.2011 12:32
```

Jak je vidět, kód, který je potřeba napsat, je poměrně dlouhý a pokud chceme pouze převést jeden formát na druhý, je tento zápis neefektivní. Z toho důvodu ukládá ROAD framework do globální proměnné `$convertor` instanci třídy `TConvertor`, která převod formátů z hlediska objemu kódu zjednodušuje.

výpis kódu č. 24 - **Zjednodušený zápis převodu formátů**

```
echo $convertor->sqlToDisplay('dateTime','2011-04-02 12:32:01');
```

Jedná se opravdu jen o zjednodušení zápisu, protože ve skutečnosti je tak jako tak vytvořena instance třídy `TDateTime` a zavolány její metody `setAsSql()` a `getAsDisplay()`.

4.10 Práce s databází

Mechanizmus pro práci s databází v rámci ROAD frameworku lze rozdělit do tří úrovní, přičemž každá z těchto úrovní je obsluhována jednou třídou.

- úroveň **databáze** – potomek třídy `TCustomConnection`
  - Třída reprezentuje připojení k databázovému serveru, odesílá do něj SQL dotazy a zpracovává jejich výsledky.
- úroveň **záznamu**(ů) – `TDataSet`
  - Reprezentuje seznam záznamů, vrácený po odeslání dotazu `SELECT` do databázového severu. Umožňuje procházení těmito záznamy a jejich ukládání do různých datových struktur.
- úroveň **datového pole** – `TField`
  - Instance této třídy reprezentují jednotlivá datová pole a zpřístupňují informace o jejich datovém typu, výskytu v primárním klíči atd.

Komponenty frameworku, které jsou určeny pro práci s databází, počítají s tím, že programátor nadefinuje připojení k databázovému serveru v rámci globální proměnné `$db`.

4.10.1 Čtení dat

K získání dat z databáze zavoláme metodu `query()` třídy obsluhující vybraný databázový server (potomek `TCustomConnection`). Tato metoda vrátí instanci třídy `TDataSet`, která zpřístupňuje data záznamů odpovídajících podmínkám v dotazu. V asociativním poli `$fields` třídy `TDataSet` jsou pod jmény datových polí uloženy[19] instance třídy `TField`, díky kterým může programátor a hlavně jednotlivé komponenty přistupovat k datům v nich uložených.

Hodnoty polí nejsou uloženy jako prostá data, ale jsou obaleny instancemi potomků třídy `TCustomValue`. Díky tomu, může programátor získávat data již správně naformátovaná pro konkrétní akci, kterou realizuje.

4.10.2 Zápis dat

Zápis dat do databáze probíhá vždy pomocí odeslání příkazu `UPDATE` resp. `INSERT` a stejně tak tomu je i při použití ROAD frameworku. Konstrukce takového dotazu je dle mého názoru velmi zdlouhavá činnost, kterou bohužel vývojář webové aplikace musí neustále opakovat znovu a znovu. V ROAD frameworku jsem proto do třídy `TDataSet` implementoval mechanizmus, který generování těchto příkazů i jejich odesílání do databázového serveru plně automatizuje.

Pro modifikaci záznamu stačí požadovaný záznam načíst do třídy `TDataSet` (viz předchozí podkapitola) a poté upravit hodnoty jeho jednotlivých polí. Vygenerování příkazu `UPDATE` a jeho odeslání do databázového serveru je provedeno po zavolání metody `post()`.

výpis kódu č. 25 - **Modifikace databázového záznamu prostřednictvím ROAD frameworku**

```
$data = $db->query('SELECT * FROM sluzby WHERE id = 1');
$data->fetchFields();
$data->fields['nazev']->value->setRaw('Snídaně');

// nebo zkráceným zápisem, který v případě
// potřeby sám zavolá i metodu fetchFields()
$data->setRaw('nazev','Snídaně');
$data->post();
```

Pro přidání nového záznamu slouží metoda `insert()`, která musí být zavolána předtím, než začnou být upravovány hodnoty jednotlivých polí.

19 Aby nebyla zbytečně prodlužována doba odezvy aplikace, je pole `$fields` naplněno až po zavolání metody `fetchFields()`.

4.11 Zpracování formulářů

Formulář je bezpochyby hlavním nástrojem aplikace pro získání údajů od uživatele. Zároveň je to jedno z nejzranitelnější míst aplikace. Protože špatně ošetřené hodnoty mohou vést k narušení dat a běhu aplikace, počínaje poškozenou integritou dat a konče jejich ztrátou či krádeží. Je zarážející, kolik „na koleně“ psaných aplikací stále při ukládání do databáze neošetřuje hodnoty proti útokům.

Tato podkapitola obsahuje jen stručný popis základních principů, více informací o zpracování formulářů najdete v příloze této diplomové práce.

4.11.1 Definice formuláře

Formulář v ROAD frameworku není nic jiného než podstrom komponent jehož kořenem je komponenta `TForm`. Tato komponenta se stará o vygenerování HTML prvku `<form>` s příslušnými atributy a o načítání dat přicházejících z prohlížeče do jednotlivých pod-komponent. Vzhled a obsah formuláře je pak definován jednotlivými ovládacími prvky, které jsou v ROAD reprezentovány samostatnými komponentami. Framework ve své knihovně komponent zahrnuje všechny běžné ovládací prvky z nichž většinu ještě obohacuje.

Každá komponenta, reprezentující ovládací prvek, obsahuje množství vlastností, přes které lze nastavit nejen její vzhled ale i většinu faktorů ovlivňujících validaci zadané hodnoty. Například pomocí vlastnosti `$required` lze určit, zda je hodnota vyžadovaná, vlastnost `$unique` zaručuje jedinečnost hodnoty v rámci tabulky nebo její zvolené podmnožiny (`$uniqueScope`) a vlastností `$format` typu `TCustomValue` může programátor určit datový typ hodnoty a jeho náležitosti.

4.11.2 Obsluha formuláře

Obsluha formuláře a dat přes něj zadaných není v ROAD frameworku realizována na úrovni komponenty `TForm`, ale na úrovni tlačítka `TSubmitButton` ve formuláři umístěného. Programátor prostě využije událost `onClick` této komponenty a po stisknutí tlačítka uživatelem, provede příslušné akce. Aby si mohl být při zpracování údajů jistý že odpovídají jeho představám, zavolá jako v rámci obsluhy události nejdříve metodu `validate()` na komponentě `TForm`, která postupně projde všechny ovládací prvky, zkontroluje platnost zadaných hodnot, vyrozumí uživatele o případných nesrovnalostech a poté vrátí logickou hodnotu určující, zda je formulář správně vyplněn. Na základě této hodnoty se pak programátor rozhodne, jestli bude se zpracováním hodnot pokračovat.

Pro ukázku obsluhy formuláře použiji rozšířenou verzi příkladu z kapitoly 4.4.

výpis kódu č. 26 - **Obsluha formuláře**

```
class TVlastniPanel extends TPanel {
  function odeslano($sender) {
    if($this->formular->validate()) {
      echo 'Uživatel napsal: "'.$this->vstupniPole->format->getAsDisplay().'"';
    }
  }
}
$panel = new TVlastniPanel($page,'panel');
$formular = new TForm($panel,'formular');
$ramecek = new TGroupBox($formular,'ramecek');
$ramecek->caption = 'Zadejte text';

$vstupniPole = new TEditBox($ramecek,'vstupniPole');
$vstupniPole->caption = 'Číslo';
$vstupniPole->required = true;        // uživatel musí zadat hodnotu
$vstupniPole->format = new TNumber(); // uživatel je nucen zadat platné číslo

$tlacitko = new TSubmitButton($formular,'tlacitko');
$tlacitko->caption = 'Odeslat';
$tlacitko->onClick = 'odeslano';
```

4.11.3 Propojení s databází

Každý ovládací prvek v ROAD frameworku lze jednoduše propojit s datovým polem tabulky pomocí vlastnosti `$field`, která očekává název tohoto pole. Pomocí metody `setDataset()`, která očekává instanci třídy `TDataSet`, lze poté komponentě přiřadit data získaná z databáze pomocí metody `query()`. Metoda `setDataset()` je rekurzivní a stačí jí proto zavolat přímo na komponentě `TForm`. Stejně tak je rekurzivní i metoda `loadFromDataset()`, která se postará o načtení hodnot z objektu `TDataSet` do samotných komponent.

Při ukládání hodnot zpět do databáze postupujeme analogicky. Nejprve zavoláme metodu `saveToDataset()`[20], která hodnoty uloží zpět do objektu `TDataSet` a na tomto objektu pak zavoláme metodu `post()`, která data modifikovaný záznam odešle do databáze.

Všimněte si, že i přesto, že odstraníme definici formátu a informaci o povinnosti pole, formulář stále vyžaduje jako vstupní číslo a nepovolí uložení bez jeho vyplnění. ROAD framework totiž sám zjišťuje datový typ, povinnost a dokonce i požadavek unikátnosti datového pole přímo ze záznamů získaných z databáze. Pokud však programátor z nějakého důvodu chce, může vlastnosti pole získané z databáze předefinovat, protože vlastnosti `$required`, `$unique` a `$format` jsou nadřazené.

20 Metoda si sama zajistí validaci vstupních polí a není proto potřeba volat metodu `validate()` samostatně.

výpis kódu č. 27 - **Propojení formuláře s databází**

```
class TVlastniPanel extends TPanel {
  function odeslano($sender) {
    if($this->formular->saveToDataSet()) {
      $this->formular->dataset->post();
    }
  }
}
...
$vstupniPole->field = 'cena';
$formular->setDataset($db->query('SELECT * FROM vylety'));
$formular->loadFromDataset();
```

Propojení pomocí komponenty TTableLink

Ruční obsluha načtení a uložení hodnot formuláře z a do databáze může vypadat poměrně jednoduše, pokud se ale pustíme do pokročilejších úkonů, jako ošetření výjimek, reakce na změnu primárního klíče a ukládání souvisejících záznamů jiných tabulek, kód, který bude potřeba pokaždé napsat, značně naroste. ROAD framework proto nabízí komponentu `TTableLink`, která všechny tyto akce automatizuje. `TTableLink` je potomkem komponenty `TSubmitButton` a proto se na stránce zobrazí jako obyčejné tlačítko. Z pohledu programátora ale není vůbec obyčejné, neboť se samo postará o načtení všech hodnot do formuláře a o jejich zpětné uložení do databáze. Vše co k tomu potřebuje je nastavení vlastnosti `$query` (očekává SQL dotaz vracející záznamy, které mohou být upraveny) nebo `$table` (očekává název tabulky jejíž záznamy mohou být upraveny) a umístění do formuláře.

Aby komponenta věděla, který záznam z dané množiny má právě upravovat, musíme jí předat parametr pomocí `$_GET` nebo `$_POST` proměnné `nazevKomponenty-keys`. Proměnná může obsahovat asociativní pole všech datových polí spadajících do primárního klíče a jejich hodnot, nebo řetězec identifikující záznam tak, jak byl vygenerován metodou `getRowId()` třídy `TDataSet`. Pokud nechceme upravovat existující záznam, ale přidat nový, stačí stránku zavolat bez tohoto parametru. Nastavením vlastnosti `$mode` můžeme přidávání nebo editaci záznamu zakázat.

Více informací o komponentě TTableLink a ukázku jejího použití naleznete v příloze.

4.12 Problémy a jejich řešení

Převedení konceptu do reálné implementace se samozřejmě neobešlo bez překážek a následných kompromisů. Pro ilustraci uvádím několik z nich.

4.12.1 Pomalé hledání komponent podle názvu

Vyhledávání komponent podle názvu bylo primárně určeno pro vnitřní potřeby frameworku, kde bylo používáno jen občasně. Když jsem ale do frameworku zapracoval možnost přistupovat ke komponentám jako k vlastnostem jejich

nadřazených komponent, uvědomil jsem si, že rychlost tohoto prohledávání už bude hrát jistou roli.

Udělal jsem proto test rychlosti o rozsahu 100 000 čtení z vlastnosti vyhledávané komponenty a výsledky byly katastrofální. Hledání komponenty podle názvu bylo ve velmi jednoduchém formuláři (definovaném v kapitole 4.3) 400x pomalejší než přímý přístup k objektu komponenty a přístup přes vlastnost nadřazené ještě o něco málo pomalejší. Uvědomil jsem si, že takový rozdíl v rychlosti by se při větší míře používání mohl znatelně projevit na době odezvy celé aplikace, proto jsem začal přemýšlet, jak tento problém obejít, aniž bych musel omezit programátora.

Řešení

Rozhodl jsem se, že odkazy na všechny vytvořené komponenty uložím do asociativního pole indexovaného jejich názvy. V tomto poli pak bude možno komponentu vyhledat mnohem rychleji, než by to bylo možné pomocí procházení. Výsledky testu tohoto vylepšení byly mnohem příjemnější než ty předchozí. Hledání podle názvu nyní bylo přibližně 10x pomalejší a přístup pomocí vlastnosti přibližně 20x (s rostoucí složitostí stromu komponent se rozdíl mezi klasickým hledáním a přístupem přes vlastnost bude stírat). Bohužel i toto řešení má některé podstatné nedostatky:

- v asociativním poli se rozlišují malá a velká písmena
- jakmile programátor po vytvoření komponenty změní její název, musela by se změna promítnout i do asociativního pole; to by vyžadovalo přístup přes setter, což by opět poněkud zpomalilo inicializaci aplikace

Prozatím jsem se rozhodl tyto nedostatky nijak neřešit a nechat programátora profitovat z dodržování principu pojmenování komponent již při vytváření jejich instancí (pomocí parametru konstruktoru) a správného používání malých a velkých písmen. Programátor, který tyto dvě zásady nedodrží, bude bohužel penalizován časově náročnějším vyhledáváním. Pokud totiž vyhledávání pomocí asociativního pole selže, použije se klasické prohledávání do hloubky.

4.12.2 Rychlost odezvy aplikace

Jak se můžete dočíst v kapitole 1.2.3, rychlost odezvy aplikace je jedním z nejdůležitějších faktorů určujících komfort uživatele. Proto je důležité, aby ROAD framework příliš nepřidával na době odezvy. Faktory určující dobu odezvy můžeme rozdělit do dvou skupin podle toho, jestli jsou závislé na počtu uživatelů, kteří právě aplikaci používají, nebo ne.

Čas generování stránky

Jedinou veličinou spadající do první skupiny, kterou framework přímo ovlivňuje, je čas potřebný k exekuci PHP kódu a vygenerování HTML kódu stránky. Framework prozatím nebyl nasazen do náročného provozu, takže nevím, jak

se zatížení projeví na rychlosti odezvy, ale předpokládám, že právě čas potřebný k vygenerování stránky bude růst nejrychleji. Když jsem se při nízkém zatížení snažil dobrat k tomu, jaké akce nejvíce přidávají na době odezvy, zjistil jsem, že nejvíce času spotřebuje načítání a parsování souborů frameworku. Abych si svou teorii ověřil, implementoval jsem do ROAD frameworku jakýsi měřič výkonu, který automaticky shromažďuje údaje o časové náročnosti jednotlivých úkonů v rámci běhu frameworku. K testování jsem použil ukázkovou aplikaci cestovka, který je popsaná v příloze této diplomové práce. Snažil jsem se napodobit chování běžného uživatele a provedl celkem 30 měřitelných akcí. Průměrné výsledky testu ukazuje následující tabulka:

tabulka č. 3 - **Test časové náročnosti běhu frameworku**

Skupina	činnost	čas	ze skupiny	z celku
Zavedení jádra při načítání souboru `run.php`		**0,081**		**77,67%**
	Načtení souborů jádra, komponent a renderů	0,074	91,56%	71,11%
	Ostatní činnosti inicializace jádra	0,007	8,44%	6,56%
Běh frameworku po zavolání metody `run()`		**0,021**		**20,57%**
	Inicializace komponent	0,007	28,39%	6,34%
	Zpracování dat z formuláře	0,000	0,51%	0,11%
	Renderování komponent	0,015	63,01%	14,07%
Ostatní činnosti prováděné v souborech stránek		**0,002**		**1,76%**
	Definice komponent, kontrola práv atd.	0,002	100,00%	1,76%
Celkem		**0,104**		**100,00%**
z toho dotazy do **Databáze**[21]	Zahrnuje pouze vykonání dotazu, ne nadstavbu přidanou ROAD frameworkem	0,006		5,71%

Jak vidíte, zavedení jádra zabere v běžných podmínkách téměř 80% výpočetního času, přičemž téměř všechen tento čas zabere načítání a parsování souborů se zdrojovými kódy frameworku.

Řešení pomocí modularizace

Jedním možným řešením je rozdělit zdrojový kód frameworku do více relativně samostatných částí, které budou frameworkem načítány pouze, pokud budou v rámci prováděné akce zrovna potřeba. Nejlépe půjde tento zásah realizovat v rámci knihovny komponent. PHP dokonce nabízí speciální funkci `__autoload()`, která je volána v případě, že se programátor pokouší vytvořit instanci třídy, jejíž definice ještě není načtená v paměti. V rámci obsluhy této funkce by framework mohl automaticky načíst příslušný soubor s definicí třídy.

Toto řešení sice vyžaduje zásah do stávající struktury kódu, ale bude účinné na jakékoliv konfiguraci webového serveru. V tomto okamžiku však nejsem

21 Dotazy do databáze bohužel nejde odečíst od některé z ostatních skupin, protože práce s databází se prolíná všemi skupinami.

schopen ani odhadnout, jak moc by se framework dal modularizovat a tedy ani to, jak by byl provoz aplikace urychlen.

Řešení pomocí vyrovnávací paměti

Vzhledem k tomu, že se PHP, díky jednoduchosti jeho nasazení a jeho rozšíření, začalo používat i v aplikacích s rozsahem, který pro něj vůbec nebyl zamýšlen, bylo jen otázkou času, než vývojáři přijdou s nástroji pro urychlení běhu aplikací psaných v PHP. Mezi těmito nástroji lze najít jeden velmi užitečný, nástroj, který bychom mohli nazývat *loader*, *pre-loader*, *code-cache* atd. Ve zkratce jde o nástroj, který často používané soubory se zdrojovými kódy ukládá do operační paměti, tak, aby mohli být příště rychleji načtené. Zrychlení je více než znatelné, protože do operační paměti není zdrojový kód ukládán jako text, ale jako předkompilovaný kód.

Kromě komerčních nástrojů jako:

- *ZendOptimizer+*
- nebo *IonCube PHP Accelerator*

existuje i několik open-source projektů, z nichž za zmínku stojí například:

- *Alternative PHP Cache*
- *eAccelerator*
- *XCache*

Pro ukázku efektu tohoto řešení, jsem do své distribuce PHP doinstaloval rozšíření *Alternative PHP Cache* a zopakoval test doby odezvy s těmito výsledky:

tabulka č. 4 - **Zrychlení odezvy s použitím nástroje Alternative PHP Cache**

Skupina	činnost	čas	ze skupiny	z celku
Zavedení jádra při načítání souboru `run.php`		**0,024**		**48,52%**
	Načtení souborů jádra, komponent a renderů	0,016	68,26%	33,12%
	Ostatní činnosti inicializace jádra	0,008	31,74%	15,40%
Běh frameworku po zavolání metody `run()`		**0,024**		**48,37%**
	Inicializace komponent	0,009	34,82%	17,93%
	Zpracování dat z formuláře	0,000	0,50%	0,25%
	Renderování komponent	0,015	58,46%	30,10%
Ostatní činnosti prováděné v souborech stránek		**0,002**		**3,11%**
	Definice komponent, kontrola práv atd.	0,002	100,00%	3,11%
Celkem		**0,049**		**100,00%**
z toho dotazy do **Databáze**	Zahrnuje pouze vykonání dotazu, ne nadstavbu přidanou ROAD frameworkem	0,008		16,15%

Všimněte si, že čas potřebný k načtení souborů frameworku se zmenšil z 0,074 na 0,016 vteřiny to jest na 20%. Díky tomu se průměrný čas generování stránky

zmenšil o 50%, což je znatelné zlepšení. Prozatím jsem nezkoušel ostatní nástroje, ale předpokládám, že vzhledem k principu vyrovnávací paměti, budou výsledky srovnatelné.

Upovídanost kódu

Dalším faktorem, který může hlavně u pomalých připojení ovlivnit rychlost odezvy je objem dat, které je potřeba přenést do počítače uživatele. Mezi tato data patří připojené soubory (obrázky, styly a skripty) a hlavně HTML kód stránky. Připojené soubory jsou většinou prohlížečem ukládány do vyrovnávací paměti, takže jejich načítání příliš nezdržuje. V této podkapitole se tedy zaměříme pouze na problém objemného HTML kódu stránky.

Když si zobrazíte zdrojový kód stránky vygenerované ROAD frameworkem, určitě si všimnete, že je celkem „upovídaný“. Každá komponenta má vždy přiřazený atribut `id` a přinejmenším jednu třídu v rámci atributu `class`. Některé komponenty mohou být v závislosti na jejich konfiguraci popsány i pěti třídami. Ve spojení s tím, že názvy tříd nejsou nijak zkracovány, může tato skutečnost znamenat klidně i o 100% objemnější kód oproti kódu bez popisu tříd.

Důvod, proč je generovaný kód tak upovídaný je jednoduchý:

> Větší množství tříd a lepší čitelnost názvů tříd přispívá k snadnějšímu přizpůsobení stránky pomocí stylových souborů a snadnější ovládání vygenerované stránky pomocí skriptů.

V případě, že se během nasazení do reálného provozu ukáže objem kódu jako neúnosný, budu asi framework muset přizpůsobit tak, aby generoval:

- kratší názvy tříd
- a/nebo méně různých tříd

Závěr

Závěrem hlavního textu své diplomové práce bych chtěl především zhodnotit úspěšnost prototypové implementace ROAD frameworku a také vytyčit směr, kterým by se jeho další vývoj mohl ubírat.

Před tím, než jsem začal psát text této diplomové práce, jsem již měl velkou část frameworku implementovanou, což ale neznamená, že během psaní textu neprošel žádnými změnami. I přesto, že jsem si návrh frameworku celkem pečlivě připravil ještě před tím, než jsem napsal první řádek kódu, byl tento návrh, tak jak bylo potřeba, ohýbán a měněn. Při psaní textu a hlavně jeho přílohy mě poměrně často napadaly myšlenky, které jsem prostě nemohl do frameworku neimplementovat. Diplomovou práci je ale bohužel potřeba odevzdat, vývoj pozastavit a stav frameworku zmrazit k dnešnímu datu. Skutečnost, že je vývoj frameworku stále otevřený, ale zároveň musí být pozastaven, sama o sobě značí, že v něm bude množství chyb a nedostatků, které odhalí až další vývoj a nasazení do reálného provozu.

Již dnes ale můžeme zhodnotit, zda ROAD framework dostál svým předsevzetím a stal se tak alespoň trochu konkurenceschopným vůči ostatním frameworkům na trhu. Dle mého názoru mohu s klidným svědomím tvrdit, že všechny požadavky, jmenované v kapitole 2.4, ROAD framework úspěšně plní. Jednoznačně největším přínosem frameworku je architektura renderovacích objektů, která díky vícenásobným zcela odděleným prezentačním vrstvám umožňuje pohodlné používání aplikace na různých koncových zařízeních. Za další plus bych si dovolil považovat striktně událostmi ovládanou komponentovou strukturu frameworku a integrovanou podporu dynamických ovládacích prvků využívajících AJAX. Fakt, že jednotlivé komponenty jsou většinou plně konfigurovatelné pomocí vlastností, umožňuje budoucí implementaci drag&drop nástroje pro uživatelsky příjemnější definici formulářů a dalších komplexních stránek.

Když vynechám drobné chybky a nedodělky, vidím na ROAD frameworku hned několik větších nedostatků. Z hlediska použitelnosti v reálném nasazení je to určitě jeho výkon. Tento problém je ale celkem podrobně popsán v kapitole 4.12.2 a proto se jím nebudu dále zaobírat. S relativně krátkou dobou strávenou vývojem frameworku je spojen i nepříliš velký počet jednotlivých komponent. Díky tomu framework prozatím postrádá některé důležité funkce jako například podporu uploadu souborů. Potom, co bude framework úspěšně nasazen do několika reálných aplikací, bude tento nedostatek odstraněn, nebo alespoň minimalizován, neboť budu kvůli těmto projektům nucen chybějící funkce implementovat.

Dalším velkým záporným faktorem frameworku je nedostatek možností, jak přizpůsobit uspořádání komponent na jednotlivých stránkách. V současné verzi může programátor pouze skládat komponenty za sebou a případně určit, které ovládací prvky mají být spolu na jednom řádku a které ne. Na svou obhajobu musím říci, že právě malé množství prvků pro změnu uspořádání stránky,

umožňuje generovat zcela odlišné výstupy pro jednotlivá koncová zařízení. Pro vyřešení tohoto problému přemýšlím o implementaci „layoutů“ pro jednotlivé komponenty. Pro každý render-set by si pak programátor mohl vytvořit jakousi šablonu z (X)HTML kódu, do které by rozmisťoval jednotlivé komponenty.

V předchozích odstavcích jsem se pokusil shrnout kladné i záporné stránky ROAD frameworku a poukázat na hlavní prvky, které ho odlišují od ostatních dostupných produktů. U většiny jmenovaných problémů jsem zároveň navrhl jejich řešení a tato řešení se pokusím v blízké době implementovat. Nyní je jen na vás, abyste posoudili, zda kladné stránky převyšují a implementaci ROAD frameworku lze považovat za úspěšnou, či nikoliv.

Jak jsem zmínil dříve, mám kromě odstranění stávajících nedostatků, v plánu i implementaci rozšíření pro jedno z existujících IDE, které by umožnilo snadnější definici stromu komponent a případně i jejich rozvržení na stránce.

Děkuji vám za čas, který jste věnovali čtení mé diplomové práce, a doufám, že jste v jejím textu nalezli něco, co vás zaujalo. Pokud se vám ROAD framework líbí a chtěli byste se o něm dozvědět více, přečtěte si i *přílohu A* a nechte si ukázat jak snadno se dá pomocí ROAD frameworku vytvořit jednoduchá aplikace.

Bc. Jiří Staniševský

Seznam použité literatury

[1] Merriam-Webster Incorporated. *Definition of framework* [online]. 2011 [cit. 2011-04-21]. Dostupný z WWW: <http://www.merriam-webster.com/dictionary/framework>.

[2] RŮZNÍ. *Framework* [online]. 2006 [cit. 2011-04-21]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Framework>.

[3] FRIEDL, Stephen J.. SQL Injection Attacks by Example. *Steve Friedl's Unixwiz.net Tech Tips* [online]. 2007 [cit. 2011-02-01]. Dostupný z WWW: <http://www.unixwiz.net/techtips/sql-injection.html>.

[4] FERSCHMANN, Petr. *Objektově-relační mapování* [online]. 2005 [cit. 2011-02-15]. Dostupný z WWW: <http://blog.softeu.cz/prednasky/2005/orm/index.html>.

[5] PÖSINGER, Martin. Komponentový framework pro vývoj webových aplikací. [online]. 2007 [cit.]. Dostupný z WWW: <http://isis.vse.cz>.

[6] RŮZNÍ. *Spaghetti code* [online]. 2001-2011 [cit. 2011-01-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Spaghetti_code>.

[7] WELLING, Luke, THOMSONOVÁ, Laura. *PHP a MySQL - rozvoj webových aplikací*. 2. vyd. Praha : Softpress 2004. 910 s. ISBN 80-86497-60-7.

[8] POTENCIER, Fabien. *Getting Started with symfony* [online]. 2009. [cit. 2011-02-05]. Dostupné z WWW: <http://www.symfony-project.org/getting-started/1_4/en/>.

[9] Zend Technologies Ltd.. *Programmer's Reference Guide* [online]. Zend Technologies Ltd. 2010. [cit. 2011-02-05]. Dostupné z WWW: <http://framework.zend.com/manual/en/learning.quickstart.html>.

[10] PradoSoft. *Prado QuickStart Tutorial* [online]. PradoSoft 2007. [cit. 2011-02-06]. Dostupné z WWW: <http://www.pradosoft.com/demos/quickstart/>.

[11] Yii Software LLC.. *The Definitive Guide to Yii* [online]. Yii Software LLC. 2010. [cit. 2011-02-06]. Dostupné z WWW: <http://www.yiiframework.com/doc/guide/>.

[12] EllisLab, Inc.. *CodeIgniter User Guide* [online]. EllisLab, Inc. 2009. [cit. 2011-02-08]. Dostupné z WWW: <http://codeigniter.com/user_guide/>.

[13] Nette Foundation. *Vytvořte si první aplikaci!* [online]. Nette Foundation 2010. [cit. 2011-02-08]. Dostupné z WWW: <http://doc.nette.org/cs/quickstart>.

[14] PACHECO, Xavier, TEIXEIRA, Steve. *Delphi průvodce vývojáře*. 1. vyd. Brno : UNIS 1996. 882 s. .

[15] Nette Foundation. *dibi - Quick Start* [online]. Nette Foundation 2010. [cit. 2011-02-08]. Dostupné z WWW: <http://dibiphp.com/cs/quick-start>.

[16] ZAKAS Nicholas C., McPEAK Jeremy, FAWCETT Joe . *Ajax Profesionálně*. 1. vyd. Brno : Zoner Press 2007. ISBN 978-80-86815-77-0. Kap. 2,4,8; s. 37-87, 125-154, 271-308.

[17] World Wide Web Consortium. *Stránky World Wide Web Consortium* [online]. [cit. 2011-03-02]. Dostupný z WWW: <http://www.w3.org>.

[18] BERNARD, Borek. Úvod do architektury MVC. *zdroják.cz* [online]. 2009 [cit. 2011-01-25]. Dostupný z WWW: <http://zdrojak.root.cz/clanky/uvod-do-architektury-mvc/>. ISSN 1803-5620

[19] TICHÝ, Jan. Pět vrstev modelu. *PHP Guru* [online]. 2010 [cit. 2011-02-15]. Dostupný z WWW: <http://www.phpguru.cz/clanky/pet-vrstev-modelu>.

[20] TEIXEIRA, Steve, PACHECO, Xavier. *Mistrovství v Delphi 6*. 1. vyd. Praha : Computer Press 2002. 822 s. ISBN 80-7226-627-6.

příloha A:
Tvorba jednoduché aplikace

Dle mého názoru bude pro ukázku toho, co všechno ROAD framework dokáže, nejlepší vytvořit společně jednoduchou aplikaci. Na rozdíl od převážně teoretického popisu v samotné diplomové práci není úkolem této přílohy ukázat, jak framework funguje, a prozradit vám principy, na jejichž základech pracuje. Účelem této přílohy je nechat vás posoudit jak snadné je aplikaci pomocí ROAD frameworku vytvořit a jak komfortně jí lze poté díky němu používat.

A.1 Popis aplikace

Pro předvedení vybraného souboru základní i pokročilých funkcí frameworku jsem zvolil jakýsi webový katalog výletů fiktivní cestovní kanceláře. Kromě toho, že půjde katalogem procházet, bude možné ho i plně spravovat. V rámci katalogu jsou kromě samotných výletů evidovány i jejich destinace a služby, které jsou do jednotlivých výletů zahrnuty.

A.2 Instalace frameworku

A.2.1 Požadavky naší aplikace

Pro provoz aplikace, kterou hodláme vytvořit budeme potřebovat následující software:

- **webový server s podporou PHP 5.2** – nejlépe server Apache
- **databázový server MySQL** – v brzké době je plánována implementace podpory PostgreSQL
- **utf-8 kompatibilní textový editor** – od všech souborů aplikace se očekává, že budou v kódování utf-8

A.2.2 Soubory frameworku

V přiloženém archivu `road-instalace.zip` naleznete zkomprimovaný adresář `road` obsahující kompletní instalaci aktuální verze frameworku. Archiv můžete rozbalit buď přímo do kořenového adresáře webového serveru nebo do jeho libovolného podadresáře. ROAD framework automaticky detekuje, jestli aplikace běží v kořenovém adresáři, nebo v některém jeho podadresáři a automaticky upravuje všechny generované odkazy. Pro správnou funkci frameworku je ještě potřeba umožnit webovému serveru zápis do podadresáře `road/cache`.

A.2.3 Založení databáze

Prostřednictvím nástroje pro správu vašeho databázového serveru založte databázi s názvem *cestovka* a pro vytvoření její struktury a naplnění daty, použijte soubor `mysql.sql` z archivu `cestovka.zip`.

A.2.4 Konfigurace připojení k databázi

Po založení nové databáze je potřeba nakonfigurovat ROAD tak, aby byly jeho komponenty schopny s touto databází pracovat. Otevřete soubor zaváděcí soubor `run.php` umístěný v podadresáři *road*, odkomentujte část s definicí databáze

a nastavte jednotlivé parametry tak, aby odpovídaly konfiguraci vašeho databázového serveru.

výpis kódu č. 28 - **Konfigurace připojení k databázovému serveru**

```
<?php
  require_once('core/road.php');

/*
  If you want to use database support in your application,
  please uncomment following lines an configure connection
  parameters for appropriate database driver
*/

  $db = new TMySqlImprovedConnection();
// or
// $db = new TPostreSqlConnection();

  $db->debug = true;
  $db->server->username = 'root';
  $db->server->password = '';
  $db->connect();
  $db->selectDatabase('cestovka');


  $road->init();
?>
```

A.2.5 Konfigurační soubor

Dalším důležitým souborem, který je přímo určen k tomu, aby v něm programátor dělal změny, je konfigurační soubor `road/config.php`. Soubor obsahuje množství konstant, kterými lze běh aplikace přizpůsobit. Nás budou zajímat především konstanty `CFG_APPLICATION_NAME`, obsahující název naší aplikace, a `CFG_PAGE_TITLE_FORMAT`, která určuje, jak bude vypadat popisek stránky zobrazený jako nadpis okna nebo panelu prohlížeče.

výpis kódu č. 29 - **Konfigurační soubor**

```
  define('CFG_APPLICATION_NAME','Katalog výletů');
  define('CFG_PAGE_TITLE_FORMAT','%s - Katalog výletů');
```

Další významnou skupinou konstant je sekce *debugging*, která určuje chování aplikace při výskytu chyby. ROAD je implicitně nastaven tak, aby hlásil všechny chyby a upozornění[22]. Po odladění aplikace bude tedy žádoucí reportování chyb vypnout nebo alespoň omezit a k tomu slouží konstanta `CFG_ERROR_REPORTING`, která očekává stejné hodnoty jako vestavěná funkce PHP `error_reporting()`.

výpis kódu č. 30 - **Zamezení výpisu chybových hlášek**

```
  define('CFG_ERROR_REPORTING',E_ALL);
```

A.2.6 Test úspěšnosti instalace

Zda-li byla instalace úspěšná si můžete ověřit tím, že do adresního řádku vašeho

22 ROAD obsahuje vlastní systém pro zobrazování a logování chyb.

webového prohlížeče zadáte adresu: `http://server/road/` kde **server** zastupuje IP adresu nebo hostname počítače, na kterém aplikaci provozujete, případně doplněnou lomítkem a názvem podadresáře, do kterého jste aplikaci nainstalovali.

obrazovka č. 4 - **Ověření instalace frameworku**

ROAD framework successfuly installed

Database status:

successfuly connected

Your browser:

Firefox version 3.6
Mozilla/5.0 (Windows; U; Windows NT 6.1; cs; rv:1.9.2.15) Gecko/20110303 Firefox/3.6.15
browser running on *Win7* connected from *127.0.0.1* at *Mon, 21 Mar 2011 09:36:45 +0100*

- Javascript: **OK**
- Css: **3**
- Ajax: **OK**
- ActiveX: N/A
- Cookies: **OK**
- Session: **OK**
- Frames: **OK**
- IFrames: **OK**
- Tables: **OK**
- Java: **OK**
- VBScript: N/A

```
http://server/road/
```

Po úspěšném ověření instalace doporučuji soubor `index.php` z podadresáře *road* odstranit.

A.3 První krůčky

Než se pustíme do operací s databází a formuláři, ukážeme si v několika krocích, jak vytvořit tu nejzákladnější stránku použitím ROAD frameworku.

A.3.1 Vytvoření souboru index.php a načtení frameworku

V kořenovém adresáři naší aplikace (tedy tam, kde se nachází podadresář *road*) vytvořte soubor `index.php`. Jako každý správný PHP soubor bude i tento soubor obsahovat uvozovací a zakončovací značky jazyka PHP (`<?php` a `?>`). Za určitou analogii k těmto značkám můžeme u ROAD považovat příkazy pro načtení a spuštění frameworku. Většina stránek v aplikaci bude totiž definovaná soubory s následující strukturou:

výpis kódu č. 31 - **Základní struktura souboru stránky**

```
<?php
  require_once('road/run.php'); //načtení zaváděcího souboru
  // ... zde bude definice stromu komponent a obsluha událostí
  $road->run(); //spuštění frameworku
?>
```

Pokud teď nasměrujete webový prohlížeč na adresu `http://server/`, obdržíte prázdnou stránku, která již ale obsahuje HTML hlavičku a díky tomu jsou do ní načtené potřebné skripty a styly.

A.3.2 Ahoj světe

Abychom naší titulní stránce dodali alespoň trochu informační hodnoty, necháme jí vygenerovat jednoduchý pozdrav. K tomu je vhodné vytvořit objekt třídy `TStaticText`, který slouží k zobrazení pevně zadaného textu, či hodnoty načtené z databáze. Mezi příkazy pro načtení a spuštění frameworku vložte následující kód:

výpis kódu č. 32 - **Ahoj světe**

```
$ahoj = new TStaticText($page,'ahoj'); //vytvoření prvku
$ahoj->caption = 'Ahoj světe! Právě je '.strftime('%H:%M:%S'); //nastavení textu
$ahoj->kind = TEXT_HEADER_1; //určuje styl prvku - TEXT_HEADER = nadpis nejvyšší úrovně
```

obrazovka č. 5 - **Ahoj světe**

Ahoj světe! Právě je 13:14:40

```
http://server/
```

Možná se ptáte, proč byste měli kvůli pouhému vypsání pozdravu na stránku vytvářet objekt. Není snad jednodušší vypsat pozdrav přímo pomocí příkazu **`echo`**? Má odpověď je jasná, ano je. ALE...

A.3.3 Ahoj aktuální světe

...kdybychom pozdrav vypsali použitím příkazu **`echo`**, mohli bychom pak takto snadno vytvořit stránku s automaticky se obnovujícím nadpisem?

výpis kódu č. 33 - Automatické obnovení prvku

```
$ahoj->refreshInterval = 5000;
```

Pokud si nyní obnovíte okno prohlížeče s pozdravem, spustí se na stránce automaticky generovaný JavaScript, který zařídí, že nadpis obsahující náš pozdrav bude za 5 vteřin automaticky obnoven. Obnovení je prováděno pomocí AJAX dotazu, takže je vždy obnoven pouze obsah daného prvku a ne celé stránky.

Automatické obnovení je jen drobnou ukázkou toho, proč se i pro tak jednoduchý prvek, jako je nadpis, vyplatí používat pro generování obsahu stránky komponenty z ROAD frameworku.

A.3.4 Formáty

Při tvorbě generování uvítacího textu jsme se z hlediska principů ROAD frameworku dopustili jedné chyby. Vypisujeme totiž čas v pevně definovaném

formátu a tím narušujeme vícejazykovou strukturu ROAD frameworku. Při vývoji reálné aplikace bude nutné použít robustní standardizovaný nástroj pro práci s formáty.

Převod formátů pomocí tříd

Nejprve vytvoříme instanci třídy `TLongTime`, která obsluhuje čas s přesností na vteřiny. Poté pomocí metody `setRaw()` nastavíme aktuální čas vrácení funkcí `time()`. Potomci třídy `TDateTime` nabízí mnoho nadstandardních metod pro práci s hodnotou v nich uloženou a jednou z nich je i metoda `setNow()`, která nahrazuje použití kombinace `serRaw(time())`. Jako poslední přichází převod hodnoty do výstupního formátu. Jelikož hodnotu chceme jen zobrazit, použijeme metodu `getAsDisplay()`.

výpis kódu č. 34 - **Převod formátů pomocí speciálních tříd**

```
$cas = new TLongTime();
$cas->setNow(); // $cas->setRaw(time());

$ahoj = new TStaticText($page,'ahoj');
$ahoj->caption = 'Ahoj světe! Právě je '.$cas->getAsDisplay();
```

Převod formátů pomocí objektu $convertor

Protože opakované vytváření objektů pro převod formátů bezpochyby programátora zdržuje, nabízí mu ROAD možnost tento proces urychlit pomocí objektu třídy `TConvertor` uloženém v globální proměnné `$convertor`. Převod formátu je realizován pomocí volání převodní metody, jíž je jako první parametr předán řetězec identifikující daný datový typ a pak hodnota, kterou je potřeba převést. Jelikož v našem příkladě převádíme z časového razítka (nativního typu) do výstupního formátu, použijeme metodu `rawToDisplay()`.

výpis kódu č. 35 - **Převod formátů pomocí objektu $convertor**

```
$ahoj->caption = 'Ahoj světe! Právě je '.$convertor->rawToDisplay('longtime',time());
```

A.3.5 Popisek stránky

Také máte při brouzdání internetem otevřeno spoustu oken/panelů/záložek? Pak jistě víte, jak lehce se v nich lze ztratit, když na nich zobrazené stránky nemají správně vyplněné popisky. V ROAD frameworku je naštěstí nastavení popisku stránky úplná hračka:

výpis kódu č. 36 - **Popisek stránky**

```
$page->title = 'Vítejte';
```

A.3.6 Kam teď?

Umět pozdravit je nepochybně důležité, ale pokud by mělo zůstat jen u toho, moc bychom si toho neukázali. Pro demonstraci dalších funkcí frameworku budeme zakládat nové soubory PHP a proto je nutné dát umístit na hlavní stránku odkazy, které nás na stránky reprezentované těmito soubory nasměrují. Pro vytvoření odkazu můžete použit komponenty `TLinkTag` nebo `TLinkButton`, které se od sebe liší pouze vzhledem.

výpis kódu č. 37 - **Odkazy na hlavní stránce**

```
$odkaz = new TLinkButton($page,'odkaz');
$odkaz->href = 'seznam-vyletu.php';

$odkaz = new TLinkButton($page,'sluzby');
$odkaz->href = 'sluzby.php';
// vlastností caption můžete nastavit popisek odkazu / tlačítka
// $odkaz->caption = 'Nabízené služby';
```

Všimněte si, že si obě instance komponenty `TLinkButton` postupně odkládáme do jedné proměnné s názvem `$odkaz`. Tento způsob je použitelný, pokud nepotřebujeme pozdější přístup k jednotlivým komponentám a nechce se nám pokaždé přepisovat zkopírované kusy kódu.

obrazovka č. 6 - **Odkazy na hlavní stránce**

```
http://server/
```

A.3.7 Teď už umíme chodit

V této kapitole jsme si ukázali tvorbu velmi jednoduché stránky jejíž kód si pro jistotu ukážeme vcelku, abychom viděli, jak do sebe jednotlivé kusy kódu zapadají:

výpis kódu č. 38 - **Hlavní stránka aplikace (index.php)**

```
<?php
  require_once('road/run.php');
  $page->title = 'Vítejte';

  $ahoj = new TStaticText($page,'ahoj');
  $ahoj->caption = 'Ahoj světe! Právě je '.$convertor->rawToDisplay('longtime',time());
  $ahoj->kind = TEXT_HEADER_1;
  $ahoj->refreshInterval = 5000;

  $odkaz = new TLinkButton($page,'odkaz');
  $odkaz->href = 'seznam-vyletu.php';

  $odkaz = new TLinkButton($page,'destinace');
  $odkaz->href = 'destinace.php';

  $odkaz = new TLinkButton($page,'sluzby');
  $odkaz->href = 'sluzby.php';
```

```
  $road->run();
?>
```

A.4 Seznam výletů

V této kapitole vytvoříme stránku s jednoduchým seznamem stránkujícím všechny nabízené výlety.

Nejprve založíme nový soubor stránky s názvem `seznam-vyletu.php`. Na tento soubor již odkazujeme přímo z hlavní stránky, takže ani nemusíme do webového prohlížeče psát celou adresu.

A.4.1 Samotný seznam

Pro začátek vytvoříme instanci komponenty `TLister`, která se stará o zobrazování výpisu záznamů čerpaných z databáze nebo z proměnné typu pole. Protože chceme vypisovat položky z databázové tabulky *vylety*, komponentě nastavit vlastnost `$query` na požadovaný SQL dotaz.

výpis kódu č. 39 - **Seznam záznamů tabulky vylety**

```
  $seznam = new TLister($page,'seznam');
  $seznam->query = 'SELECT * FROM vylety';
```

obrazovka č. 7 - **Seznam záznamů tabulky vylety**

◄ 1 ►

id	cil	vedouci	odjezd	navrat	cena	popis
1	6	2	25.04.2011 08:00	25.04.2011 19:00	800	Stejně jako vždy i tentokrát vyrážíme do Hluboké celkem brzy, abychom si stihli všechno prohlédnout.
2	2	3	12.04.2011 09:00	12.04.2011 16:00	600	Prohlídka hradu včetně Chrámu Sv. Víta a obědu s prezidentem České Republiky.
3	9	3	19.04.2011 09:00	19.04.2011 18:00	700	Již samotná cesta Kokořínkým údolím je sklělým zážitek a když se k tomu přidá ještě prohlídka hradu, je z toho úžasný den strávený na čistém vzduchu.

zobrazeno 3 záznamů z celkem 3

```
http://server/seznam-vyletu.php
```

Jak vidíte na obrazovce, ROAD sám vygeneroval výpis dat, do kterého zahrnul všechny pole a záznamy vrácené na zvolený SQL dotaz. Komponenta je již v tomto stavu schopná postarat se o stránkování a třídění záznamů. K setřídění dojde po kliknutí na popisek sloupce a stránkovat lze pomocí tlačítek v horní části seznamu.

Čistě pomocí SQL jazyka můžeme zvýšit informační hodnotu tohoto výpisu tím, že nahradíme identifikátory spřažených záznamů jim odpovídajícími názvy; tedy propojíme tabulku s tabulkou *destinace* a *uzivatele*.

výpis kódu č. 40 - **Seznam záznamů tabulky vylety**

```
  $seznam->query = 'SELECT v.*, d.nazev, u.jmeno FROM vylety AS v LEFT JOIN destinace AS d
ON(v.cil = d.id) LEFT JOIN uzivatele AS u ON(v.vedouci = u.id)';
```

Teď už nám do výpisu vstupují i čitelná data. Sloupce obsahující identifikátory jsou ale pořád vidět a pro běžného uživatele budou určitě matoucí. Snadným řešením tohoto problému by bylo vyselektovat sloupce přímo na úrovni jazyka

SQL. Tímto řešením bychom se ale připravili o identifikaci záznamu pro další použití (odkaz na formulář pro úpravu záznamu apod.). Stejně snadným ne-li snadnějším řešením je použít vlastnost `$captions` komponenty `TLister`. Tato vlastnost očekává přiřazení asociativního pole, obsahujícího názvy sloupců jako klíče a popisky sloupců jako hodnoty. Pokud toto pole popisek nějakého sloupce neobsahuje, nebude tento sloupec zobrazen, takže vlastně zabijeme dvě mouchy jednou ranou.

výpis kódu č. 41 - **Selekce a pojmenování sloupců**

```
$seznam->captions = array(
  'nazev' => 'Destinace',
  'odjezd' => 'Čas. odjezdu',
  'cena' => 'Cena',
  'jmeno' => 'Vedoucí'
);
```

obrazovka č. 8 - **Selekce a pojmenování sloupců**

◄ 1 ►

Destinace	Čas. odjezdu	Cena	Vedoucí
Hluboká	25.04.2011 08:00	800	Karel Vomáčka
Kokořín	19.04.2011 09:00	700	Lucie Svíčková
Pražský Hrad	12.04.2011 09:00	600	Lucie Svíčková

zobrazeno 3 záznamů z celkem 3

```
http://server/seznam-vyletu.php
```

A.4.2 Další vlastnosti sloupců

Selekci požadovaných sloupců a nastavení jejich popisku máme za sebou. Co když ale budeme chtít nastavit další vlastnosti sloupců? V ROAD frameworku jsou jednotlivé sloupce reprezentovány instancemi třídy `TListerColumn`, která nabízí množství vlastností, metod a událostí, umožnujících přizpůsobit si sloupec podle vlastní potřeby.

Problém je ale v tom, že zatím nevíme jak k těmto objektům přistupovat. Jelikož `TListerColumn` je potomek třídy `TComponent`, lze k ní stejně jako k jakékoliv jiné komponentě přistupovat přes metodu `findComponent()` libovolné nadřazené komponenty, nebo přes zkrácený zápis: `$nadrazena->hledana` využívající univerzální *getter*.

V okamžiku, kdy potřebujeme změnit vlastnosti sloupců, tedy při definování stromu komponent, ale ještě neproběhl samotný dotaz do databáze a objekty sloupců tedy nemohou být ještě vytvořené. Třída `TLister` naštěstí nabízí speciální metodu `createColumns()`, která okamžitě provede dotaz do databáze, zjistí strukturu vrácených záznamů a vytvoří příslušné sloupce. Tuto metodu tedy musíme zavolat po nastavení vlastností `$query` a `$captions` a před modifikací vlastností jednotlivých sloupců.

Třídění

Pojďme tedy společně nastavit seznam výletů tak, aby byl implicitně setříděn podle času odjezdu. Vlastnost, která třídění ovládá, se jmenuje `$sortOrder` a může nabývat hodnot daných konstantami `SORT_*`.

výpis kódu č. 42 - **Implicitní třídění**

```
$seznam = new TLister($page,'seznam');
$seznam->query = 'SELECT v.*, ...';
$seznam->captions = array(
  'nazev' => 'Destinace',
  'odjezd' => 'Čas. odjezdu',
  'cena' => 'Cena',
  'jmeno' => 'Vedoucí'
);
$seznam->createColumns(false);    // předčasně vytvoříme sloupce podle struktury záznamů
// false v argumentu určuje, že názvy sloupců nemají být prefixované názvem seznamu

$seznam->odjezd->sortOrder = SORT_ASCENDING; // nastavíme sloupci odjezd vzest. třídení
```

Odkaz

Teď když máme seznam výletů tak jak jsem ho chtěli, zbývá jen otázka „co dál?“. Tuto otázku si budou klást především uživatelé vaší aplikace. Ti budou sice spokojeni se seznamem výletů, ale nebudou již vědět, jak vybraný výlet upravit. Z toho důvodu přidáme upravíme sloupec *nazev* tak, aby po kliknutí na něj byl uživatel přesměrován na formulář pro úpravu vybraného výletu.

Jelikož adresa, na kterou má být uživatel přesměrován, je pro každý záznam jiná, musíme použít dynamické nastavení této adresy. Díky systému obsluhy událostí je to hračka. Použijeme událost `onGetHref` sloupce *nazev*, která od programátora očekává, že v rámci její obsluhy pomocí příkazu **`return`** navrátí adresu odkazu. Událost předává parametr `$data`, který obsahuje instanci třídy `TDataSet`, případně objekt nebo pole reprezentující daný záznam. Jelikož v našem případě načítáme data do seznamu přímo z databáze, můžeme se spolehnout na to, že v tomto parametru je opravdu instance třídy `TDataSet`. Tato třída nabízí velmi užitečnou metodu `getRowId()`, která vrací jedinečný identifikátor záznamu sestavený z polí patřících do primárního klíče.

výpis kódu č. 43 - **Odkaz na formulář pro úpravy**

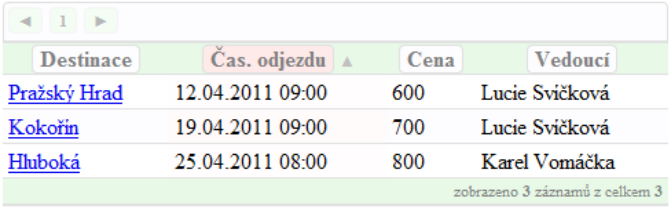
```
function nazevGetHref($sender, $data) {
  return "/upravit-vylet.php?vylet-key=".$data->getRowId();
}
$seznam->nazev->onGetHref = 'nazevGetHref';
```

Pro zjednodušení kódu lze použít i zkrácený zápis obsluhy metody. V případě, že je řetězec identifikující obsluhu události prefixován znakem >, není tento řetězec považován za název funkce, ale za kód, který má být vykonán.

výpis kódu č. 44 - **Zjednodušený zápis obsluhy události**

```
$seznam->nazev->onGetHref = '> return "/upravit-vylet.php?vylet-key=".$data->getRowId();';
```

obrazovka č. 9 - **Odkaz v seznamu výletů**



Destinace	Čas. odjezdu	Cena	Vedoucí
Pražský Hrad	12.04.2011 09:00	600	Lucie Svičková
Kokořín	19.04.2011 09:00	700	Lucie Svičková
Hluboká	25.04.2011 08:00	800	Karel Vomáčka

zobrazeno 3 záznamů z celkem 3

```
http://server/seznam-vyletu.php
```

A.4.3 Ruční definice sloupců

Jestli máte při vývoji aplikací radši všechno pevně ve svých rukách a možnosti, jak nakonfigurovat objekt třídy `TLister` k obrazu svému, se vám zdají omezené, vězte, že ruční definice sloupců je jednou z možností, jak flexibilitu této komponenty zvýšit.

Jak již bylo řečeno, třída `TListerColumn` je potomkem třídy `TComponent` a díky tomu lze jejími instancemi zacházet stejně jako se všemi ostatními komponentami. Sloupce se tedy vytváří stejně jako jakékoliv jiné komponenty a stejně tak se jim i nastavují vlastnosti. Vlastnost `$field` určuje, z jakého pole tabulky mají být načítány data do tohoto sloupce.

výpis kódu č. 45 - **Ruční vytvoření sloupce**

```
$nazev = new TListerColumn($seznam,'nazev');
$nazev->field = 'nazev';
$nazev->caption = 'Destinace';
```

Sami vidíte, že definovat jeden sloupec vyžaduje poměrně hodně kódu. Právě proto jsem ROAD postupně obohatil o další způsoby (zmíněné v předchozích podkapitolách) jak si sloupce přizpůsobit. Pokud se ale rozhodnete používat vlastní sloupce odvozené od `TListerColumn`, načítat data z jednoho pole do více sloupců nebo jinak složitě modifikovat strukturu sloupců, ruční definici se bohužel nevyhnete.

Naštěstí ROAD přichází s drobným nástrojem, který vám tuto práci ulehčí. Jedná se o speciální render-set s názvem *php*, který u komponent se složitější strukturou vypisuje na obrazovku jejich zdrojový kód. V našem případě nám tento render-set pomůže takto:

- nejprve necháme komponentu seznamu téměř bez nastavení (nastavíme pouze `$query` a případně i `$captions`)
- v prohlížeči zadáme adresu stránky s naším render-setem a přidáme parametr *?rOutput=php*
- v okně prohlížeče bychom měli nyní mít vypsaný kód pro definici sloupců podle těch, které byly automaticky vytvořené
- definice zkopírujeme a vložíme pod definici seznamu ve zdrojovém souboru

A.4.4 Trocha dynamiky

Ve stavu, v jakém se naše aplikace právě nachází, nemusí být v načítání celé stránky a pouze jedné komponenty příliš markantní rozdíl. Ovšem s rostoucí robustností stránky a nabalováním různých stylových souborů se tento rozdíl může celkem rapidně zvětšovat. Opepříme tedy náš seznam trochou AJAXu. V ROAD frameworku je to jeden z nejjednodušších úkonů. Prostě jen nastavíme vlastnost `$dynamic` komponenty `TLister` na hodnotu **`true`**.

výpis kódu č. 46 - **Aktivování AJAXu**

```
$seznam->dynamic = true;
```

Když si teď stránku obnovíte a kliknete na záhlaví sloupce nebo na některé z tlačítek pro stránkování, uvidíte, že se obsah seznamu změní bez toho, aby byla stránka znovu celá načítána ze serveru.

A.4.5 Shrnutí

Pro shrnutí úprav na stránce seznamu výletů uvádím celý zdrojový kód:

výpis kódu č. 47 - **Stránka se seznamem výletů**

```
<?php
  require_once('road/run.php');
  $page->title = 'Seznam výletů';

  $seznam = new TLister($page,'seznam');
  $seznam->query = 'SELECT v.*, d.nazev, u.jmeno FROM vylety AS v LEFT JOIN destinace AS d
ON(v.cil = d.id) LEFT JOIN uzivatele AS u ON(v.vedouci = u.id)';
  $seznam->captions = array(
    'nazev' => 'Destinace',
    'odjezd' => 'Čas. odjezdu',
    'cena' => 'Cena',
    'jmeno' => 'Vedoucí'
  );
  $seznam->createColumns(false);
  $seznam->odjezd->sortOrder = SORT_ASCENDING;
  $seznam->nazev->onGetHref = '>return "/upravit-vylet.php?vylet-key=".$data->getRowId();';
  $seznam->dynamic = true;

  $road->run();
?>
```

Když se na takhle zpětně podíváme na celý soubor, musíte se mnou souhlasit, že vypadá celkem přehledně a navíc opravdu úsporně.

A.5 Formulář pro úpravu výletu

V této kapitole vytvoříme formulář, ve kterém bude uživatel moci upravovat existující a přidávat nové výlety.

Pro tento účel založíme soubor `upravit-formular.php`.

A.5.1 Návrh formuláře

Protože budeme v budoucnu na tuto stránku přidávat ještě další formulář, obalíme ten, co budeme vytvářet teď do vlastního potoka komponenty `TPanel`. Ten se nám postará o odstínění událostí a zobrazování jednotlivých formulářů. Do tohoto panelu vložíme komponentu `TForm`, reprezentující HTML formulář. Formulář opticky rozdělíme do několika skupin pomocí komponenty `TGroupBox` a nakonec vložíme komponentu `TSubmitToolBar` jako lištu tlačítek (lze použít i `TToolBar`, která se liší pouze vzhledem).

výpis kódu č. 48 - **Kostra formuláře**

```
$panVylet = new TPanel($page,'panVylet');

$frmVylet = new TForm($panVylet,'frmVylet');


$grpObecne = new TGroupBox($frmVylet,'grpObecne');
$grpObecne->caption = 'Obecné';

$grpSluzby = new TGroupBox($frmVylet,'grpSluzby');
$grpSluzby->caption = 'Služby';

$grpPopis = new TGroupBox($frmVylet,'grpPopis');
$grpPopis->caption = 'Popis';
$grpPopis->expandable = true; // skupinu lze sbalit a rozbalit kliknutím na nadpis
$grpPopis->expanded = false;  // skupina je implicitně sbalená

$tlbVylet = new TSubmitToolBar($frmVylet,'tlbVylet');

$tlbTlacitka = new TSubmitToolBar($frmVylet,'tlbTlacitka');
```

Po tom, co jsme formulář pomocí skupin rozdělili do několika částí, nadefinujeme komponenty pro úpravu jednotlivých polí editovaného záznamu. Pro pole *cil* použijeme komponentu `TLookupSelectBox`, která reprezentuje standardní výběrový seznam obohacený o automatizaci načítání položek z databáze. Obdobně jako u ostatních komponent spolupracujících s databází, je i u této komponenty její provázání s databází dáno hodnotou vlastnosti `$query`. Od dotazu se očekává, že vrátí data-set s 1 – 3 poli, přičemž první udává hodnotu, druhý název a třetí delší popis položky seznamu.

U komponent odkazujících se na pole databáze je stejně jako u sloupců v seznamech rozhodující vlastnost `$field`, která určuje ze/do kterého pole budou hodnoty načítány/ukládány. Abychom uživateli nenabízeli první položku v seznamu jako předvybranou, přidáme do výběrového seznamu fiktivní položku s prázdnou hodnotou pomocí metody `addOption()`.

výpis kódu č. 49 - **Výběrový seznam**

```
$lstCil = new TLookupSelectBox($grpObecne,'lstCil');
$lstCil->caption = 'Destinace';
$lstCil->query = 'SELECT id,nazev,popis FROM destinace';
$lstCil->field = 'cil';
$lstCil->addOption('','- vyberte destinaci -');
```

Předpokládám, že časem se vám, stejně jako mě, zprotiví neustálé opakování zápisu `$nazevKomponenty->vlastnost`, nehledě na to, že tento zápis

znemožňuje snadnou znovupoužitelnost. Chcete-li vytvořit další instanci komponenty s maličko odlišnými vlastnostmi, musíte po zkopírování kódu upravit každý řádek tak, aby název komponenty odpovídal nově vytvářené komponentě. Z tohoto důvodu jsem v ROAD zavedl možnost zkráceného zápisu při definici komponenty. Jako třetí parametr konstruktor komponenty očekává asociativní pole, kde indexy určují názvy vlastností. Jedním příkazem tak lze najednou nastavit většinu potřebných vlastností.

výpis kódu č. 50 - **Zkrácený zápis definice komponenty**

```
$lstCil = new TLookupSelectBox($grpObecne,'lstCil',array(
  'caption' => 'Destinace',
  'query' => 'SELECT id,nazev,popis FROM destinace',
  'field' => 'cil', // PHP 5 toleruje čárku i za poslední hodnotou pole
));
$lstCil->addOption('','- vyberte destinaci -'); // volání metod zůstává nadále stejné
```

Pro pole *cena*, *odjezd* a *navrat* použijeme klasický jednořádkový vstupní prvek, který je v ROAD implementován v komponentě `TEditBox`. Komponenta se sama stará o kontrolu formátu zadávaných dat a dokonce o vyloučení duplicity u polí patřících do *unique* klíče. Navíc u polí typu *datum* nebo *datum a čas* je uživateli automaticky nabízen kalendář.

výpis kódu č. 51 - **Textová pole**

```
$lstCena = new TEditBox($grpObecne,'lstCena',array(
  'caption' => 'Cena výletu',
  'field' => 'cena',
));

$lstOdjezd = new TEditBox($grpObecne,'lstOdjezd',array(
  'caption' => 'Odjezd',
  'field' => 'odjezd',
));

$lstNavrat = new TEditBox($grpObecne,'lstNavrat',array(
  'caption' => 'Návrat',
  'field' => 'navrat',
  'newRow' => false, // umístit doprava za předchozí prvek
));
```

Další funkcí, kterou od naší ukázkové aplikace očekáváme, je možnost přiřadit k jednotlivým výletům služby, které jsou v ceně tohoto výletu zahrnuty. K tomuto účelu je v databázi zavedena relační tabulka `sluzby_vyletu` obsahující sloupce `sluzba` a `vylet`. Vytvořit v PHP stránku, která spravuje relace M:N, už je celkem složité. Díky ROAD je to ale otázka pár vteřin. Komponenta `TLookupMultiSelectBox` totiž veškerou práci programátora omezuje na nastavení několika vlastností:

- `query` – stejně jako u `TLookupSelectBox` určuje odkud se čerpá seznam povolených položek
- `relationTable` – určuje název relační tabulky
- `fieldThis` – určuje pole relační tabulky identifikující právě upravovaný záznam

- `fieldRelated` – určuje pole relační tabulky identifikující svázaný záznam

výpis kódu č. 52 - **Seznam pro výběr více položek**

```
$lstSluzby = new TLookupMultiSelectBox($grpSluzby,'lstSluzby',array(
  'caption' => 'Zahrnuté služby',
  'query' => 'SELECT id, nazev, popis FROM sluzby',
  'relationTable' => 'sluzby_vyletu',
  'fieldThis' => 'vylet',
  'fieldRelated' => 'sluzba',
));
```

Další pole, které musíme uživateli zprostředkovat je pole `popis`. Uživatel by měl mít možnost popsat výlet na více řádcích, proto použijeme komponentu `TTextArea`, kterou umístíme do implicitně sbalené skupiny `$grpPopis`.

výpis kódu č. 53 - **Víceřádkové vstupní pole**

```
<?php
  $memPopis = new TTextArea($grpPopis,'memPopis',array(
    'caption'=>'Popis výletu',
    'field'=>'popis',
  ));
```

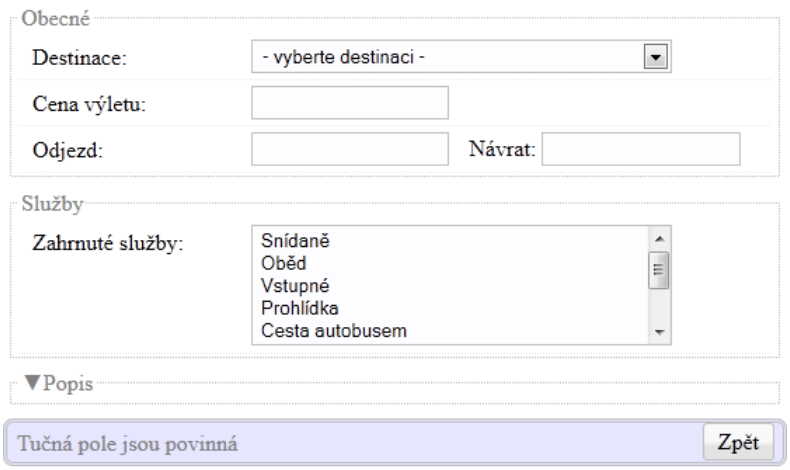
Do lišty tlačítek prozatím vložíme pouze tlačítko *Zpět*, které nás přenese zpět na seznam výletů. U URL adres se nemusíte bát v používat absolutní cesty (v rámci aplikace). ROAD URL sám doplní o případné podadresáře před tím, než odkaz odešle uživateli do prohlížeče.

výpis kódu č. 54 - **Tlačítko zpět**

```
$btnZpet = new TLinkButton($tlbVylet,'btnZpet',array(
  'caption' => 'Zpět',
  'href' => '/seznam-vyletu.php',
));
```

Pokud nyní v seznamu výletů, který jsme vytvořili v minulé podkapitole, klikneme na název některé z destinací, měli bychom vidět následující formulář:

obrazovka č. 10 - **Základ formuláře pro úpravu výletu**




```
http://server/upravit-vylet.php?vylet-keys=2
```

Jak je vidět, formulář je zatím „mrtvý“. Nejsou zobrazeny hodnoty editovaného záznamu ani vyznačená povinná pole.

A.5.2 Propojení s databází

Abychom formulář oživili, musíme ho nějak navázat na tabulku výletů. K tomuto účelu slouží komponenta `TTableLink`. Tato komponenta vypadá na formuláři jako obyčejné tlačítko. Za tímto tlačítkem se ale skrývá mechanizmus, který načítá a ukládá hodnoty do/z formuláře. Komponenta je na formuláři reprezentována tlačítkem, aby byla automatizována i interakce s uživatelem. Po kliknutí na toto tlačítko totiž dojde k uložení formuláře do databáze. Abychom měli správné pořadí tlačítek, nadefinujeme komponentu před definicí tlačítka „Zpět“.

výpis kódu č. 55 - **Provázání formuláře s databází**

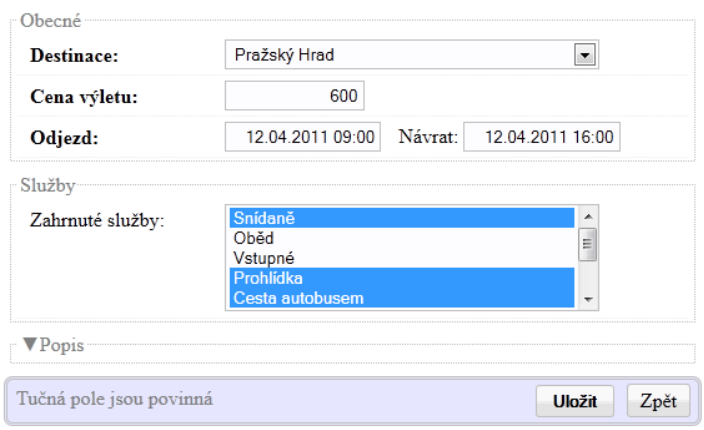
```
$vylet = new TTableLink($tlbVylet,'vylet',array(
  'caption' => 'Uložit',
  'query' => 'SELECT * FROM vylety',
  'default' => true, // reaguje na stisk kláves ENTER
  'onBeforeSave' => '>
    if($this->dataset->isInserting()) {
      $this->dataset->setRaw("vedouci",$this->road->security->userInfo->id);
    }
  ',
));
```

Vlastnost `$query` jako vždy určuje zdroj (nyní i cíl) dat. V obsluze události `onBeforeSave` nastavujeme v případě, že je přidáván nový záznam, hodnotu pole *vedouci* výletu na *id* aktuálně přihlášeného uživatele. Více o uživatelích najdete v kapitole A.8.

Komponentu `TTableLink` musíme vždy umístit dovnitř formuláře, neboť tato komponenta s databází propojuje ovládací prvky právě a pouze v rámci jednoho formuláře.

Po obnovení stránky by se nám formulář měl naplnit hodnotami z databáze:

obrazovka č. 11 - **Formulář propojený s databází**



Schválně si nyní zkuste změnit hodnoty některých polí a formulář uložit. Pokud

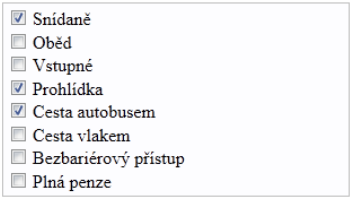
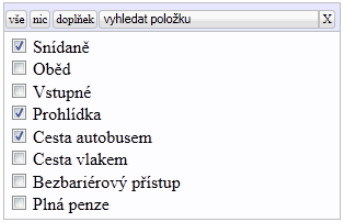
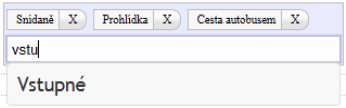
jste postupovali správně, všechno funguje a formulář je relativně připravený k použití.

A.5.3 Vylepšení seznamu služeb

Osobně nechápu, jak mohl někdo určit, že standardní výběrový seznam s možností výběru více položek je v prohlížečích reprezentován pouhým list-boxem. Většina běžných uživatelů webu nemá o ovládání list-boxů za pomocí kombinací *Ctrl+Klik* a *Shift+Klik* ani ponětí a primární účel výběrového seznamu je tak ztracen.

Již při návrhu komponenty `TSelectBox` (resp. `TMultiSelectBox`) jsem počítal s tím, že budou vzhled i chování této komponenty značně přizpůsobitelné pomocí vlastností. Nakonec se více méně vše smrsklo do jedné vlastnosti s názvem `$displayStyle`. Tato vlastnost očekává hodnotu nebo kombinaci hodnot konstant `SELECTBOX_*`.

tabulka č. 5 - **Vybrané možnosti zobrazení komponenty TMultiSelectBox**

zaškrtávátka	**zaškrtávátka s nástr. lištou**	**„připojovač“**
`SELECTBOX_RADIOGROUP_VERTICAL`	`SELECTBOX_RADIOGROUP_VERTICAL | SELECTBOX_COMBOBOX`	`SELECTBOX_LINKER`
		
• obyčejný seznam zaškrtávacích polí	• nástroje pro výběr a vyhledání položek	• ideální při velkém množství položek • uživatel zadá část textu a jsou mu nabídnuty odpovídající položky

Jednotlivé konstanty lze pomocí operátoru `|` kombinovat dle libosti:

```
$lstSluzby->displayStyle = SELECTBOX_RADIOGROUP_VERTICAL | SELECTBOX_LINKER;
```

Tento zápis například zajistí, že když nebude uživatelův prohlížeč podporovat JavaScript a nebude tak moci zobrazit „připojovač“, zobrazí namísto něj seznam zaškrtávátek. Konstantu `SELECTBOX_LINKER` je vhodné kombinovat s vlastností `$dynamic` nastavenou na **`true`**, pomocí AJAXu tak budou jednotlivé položky načítány do prohlížeče až potom, co uživatel zadá část jejich názvu. Komponenta pak nejen, že nezabírá zbytečně moc místa na obrazovce, ale navíc ještě nezdržuje při načítání stránky.

Konstanty `SELECTBOX_*` lze s obdobným efektem použít i pro komponentu `TSelectBox`, která dovoluje vybrat pouze jednu hodnotu.

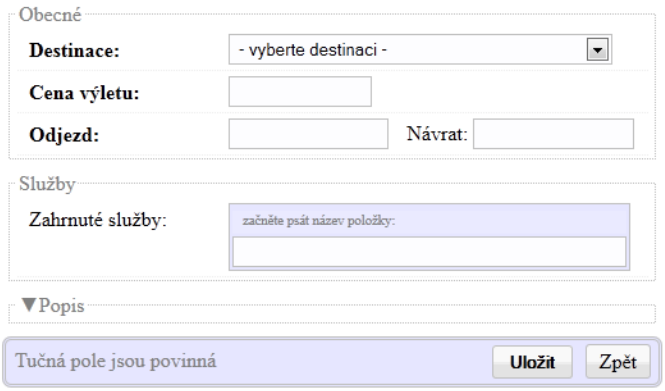
A.5.4 Nový výlet

Prozatím jsme se zaobírali pouze úpravou již existujícího záznamu. Náš katalog by ale měl určitě umožnit i založení nového výletu. V ROAD je založení nového záznamu realizováno zavoláním stránky pro úpravy bez parametru identifikujícího existující záznam. Pro nás to znamená umístit na stránku se seznamem výletů odkaz s následující adresou: `/upravit-vylet.php` Odkaz vytvoříme jako tlačítko na tlačítkové liště.

výpis kódu č. 56 - **Tlačítko pro vytvoření nového výletu**

```
$tlb = new TSubmitToolBar($page,'tlb');
$btnPridat = new TLinkButton($tlb,'btnPridat',array(
  'href' => '/upravit-vylet.php',
  'caption' => '+ Přidat',
));
```

obrazovka č. 12 - **Formulář pro přidání nového výletu**




```
http://server/upravit-vylet.php
```

A.6 Přidání nové destinace

Správu destinací lze samozřejmě realizovat standardním způsobem tak, jak jsme realizovali správu výletů. Ovšem díky JavaScriptu a AJAXu se nám otevírají nové možnosti. Jednou z těchto možností je i dynamický formulář.

Protože jsme stránku pro úpravu výletu obalili do komponenty `TPanel`, můžeme se pokusit vytvořit formulář pro přidání nové destinace tak, aby se zobrazil po kliknutí na tlačítko, přímo přes tuto stránku a po uložení nové destinace se zase zavřel. Jelikož jsme se v rámci této přílohy již s frameworkem do jisté míry sžili, budeme nyní postupovat trochu rychleji a hlavně po větších kusech.

A.6.1 Návrh formuláře

Celý formulář i všechnu jeho aplikační logiku umístíme do souboru stránky pro úpravu výletu. Jelikož budeme při obsluze událostí na tomto formuláři vykonávat složitější úkony, obalíme celý formulář do vlastního potomka komponenty

TPanel. Po vytvoření instance tohoto potomka mu nastavíme vlastnost $renderComponent na **false**, čímž určíme, aby se uživateli implicitně vůbec nezobrazoval. Pro přehlednost uvádím pouze část kódu, kompletní zdrojový kód naleznete v souboru upravit-vylet.php z přiloženého archivu cestovka.zip.

výpis kódu č. 57 - **Návrh formuláře pro přidání destinace**

```
class TDestinacePanel extends TPanel {
}

$panDestinace = new TDestinacePanel($page,'panDestinace',array(
  'renderComponent' => false,
));

$frmDestinace = new TForm($panDestinace,'frmDestinace');

$grpDestinace = new TGroupBox($frmDestinace,'grpDestinace',array(
  'caption' => 'Nová destinace',
));

... celý zdrojový kód je v souboru upravit-vylet.php ...

$destinace = new TTableLink($tlbDestinace,'destinace',array(
  'caption' => 'Uložit',
  'query' => 'SELECT * FROM destinace',
));
```

A.6.2 Zobrazení formuláře

Nyní máme nadefinovaný formulář, který je ale skrytý. Zobrazení formuláře vyvoláme pomocí komponenty TPanelOpener, která je na stránce reprezentovaná odkazem. Jakmile uživatel klikne na tento odkaz, komponenta zobrazí pomocí knihovny Fancybox panel, určený vlastností $panelName. Jelikož potřebujeme zachovat uživatelem pozměněné hodnoty, musíme pomocí vlastnosti $dynamic donutit komponentu, aby formulář otevírala pomocí AJAXového dotazu. Vlastností $exclusive nabádáme komponentu, aby při otevírání panelu zavřela všechny ostatní. Tato vlastnost má efekt pouze pokud prohlížeč nepodporuje JavaScript. Vlastnost $windowWidth je jen kosmetická a určuje šířku otevíraného Fancybox okna.

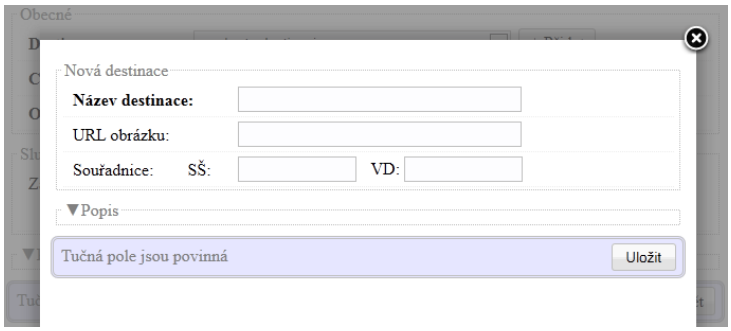
Komponentu umístíme za výběrový seznam pro destinace:

výpis kódu č. 58 - **Otevření vnořeného okna**

```
$btnDestinace = new TPanelOpener($grpObecne,'btnDestinace',array(
  'caption' => '+ Přidat',
  'cssClass' => 'rButton',
  'newRow' => false,

  'panelName' => 'panDestinace',
  'exclusive' => true,
  'dynamic' => true,
  'windowWidth' => 600,
));
```

obrazovka č. 13 - **Vnořený formulář pro přidání destinace**




```
http://server/upravit-vylet.php
```

Pokud se nyní pokusíte novou destinaci uložit, bude celá stránka znovu načtena a tím se ztratí všechny hodnoty zadané uživatelem do formuláře pro úpravu výletu. Pro správnou funkčnost našeho řešení tomuto jevu musíme zabránit. Nastavíme proto u komponenty `TTableLink` vlastnost `$dynamic` na **`true`** a tím zajistíme, že formulář bude odeslán prostřednictvím AJAXu. V případě dynamického odesílání formuláře se framework sám stará o prezentaci výsledků uživateli.

Nám už zbývá jen obsloužit událost `onSaved`, která nastává v okamžiku, kdy je formulář úspěšně uložen do databáze. Tuto událost navážeme na metodu `destinaceUlozena()` našeho potomka komponenty `TPanel`. V rámci této funkce nejdříve donutíme komponentu `TLookupSelectBox` k obnovení sama sebe pomocí AJAXu. Díky tomu do jejího seznamu přibude nově zadaná destinace. V rámci tohoto obnovení můžeme dokonce nastavit nově přidanou destinaci jako vybranou. Poslední akcí, kterou je třeba provést, je zavřít okno Fancyboxu.

výpis kódu č. 59 - **Uložení nové destinace**

```
  class TDestinacePanel extends TPanel {
    function destinaceUlozena($sender)  {
      $sender->road->queueMessage('Nová destinace byla úspěšně přidána',MSG_SUCCESS);
      if($this->road->isAjaxed()) {
        $sender->javaScript = '
$("#'.$this->page->lstCil->getId().'").refresh({
  leaveOut: ["'.$this->page->lstCil->getId().'"],
  data: [{name: "'.$this->page->lstCil->getId().'", value: "'
                                                  .$sender->getDataset()->getRowId().'"}]
});
$.fancybox.close();';
      } else {
        $this->road->redirect(changeUrlQuery(absoluteUri(':'),array(
          'rAction' => false
        ),URL_QUERY_REMOVE));
      }
      return true;
    }
  }

  ...

  $destinace = new TTableLink($tlbDestinace,'destinace',array(
    ...
    'onSaved' => 'destinaceUlozena',
    'dynamic' => true,
  ));
```

Jak je vidět z ukázky kódu, takto pokročilé techniky už se neobejdou bez přímého použití JavaScriptu. V současné době neobsahuje ROAD komponentu nebo mechanizmus, který by tuto akci plně automatizoval.

A.7 Správa služeb

Tabulka *sluzby* v naší databáze je ukázkovým příkladem tabulky s předpokladem velkého množství záznamů při malém počtu sloupců. Pro tento typ tabulky je poměrně nevhodné použít konvenční princip seznamu záznamů a formuláře pro úpravu na samostatných stránkách. Uživatel tak totiž zbytečně tráví většinu času tím, že se pomocí odkazů nebo navigačních tlačítek prohlížeče přepíná mezi stránkou se seznamem a stránkou s formulářem.

V ROAD můžeme díky komponentě `TControlPager` spojit formulář pro úpravu i seznam záznamů nejen v jednu stránku, ale i v jeden ovládací prvek. Komponenta `TControlPager` totiž replikuje vnořené ovládací prvky právě tolikrát, kolik je v tabulce záznamů.

A.7.1 Tvorba formuláře

Pro správu služeb založíme soubor `sluzby.php`. Jedinou vlastností, kterou musíme u komponenty `TControlPage` nastavit, je vlastnost `$query`. Ta očekává SQL dotaz vracející záznamy, které mají být zobrazeny/upraveny. Do této komponenty musíme vložit všechny ovládací prvky pracující s daným záznamem. V našem případě půjde o dvě komponenty `TEditBox` pro úpravu polí *nazev* a *popis*. Aby se u každého záznamu neustále neopakoval popisek těchto komponent, nastavíme jim vlastnost `$displayCaption` na **`false`**. Jelikož očekáváme, že uživatel bude odesílat upravené hodnoty zpět na server, budeme muset celou stránku obalit do komponenty `TForm`.

výpis kódu č. 60 - **Formulář pro služby**

```
$frm = new TForm($page,'frm');
$lstSluzby = new TControlPager($frm,'lstSluzby',array(
  'query' => 'SELECT * FROM sluzby',
  'dynamic' => true,
));
$edtNazev = new TEditBox($lstSluzby,'edtNazev',array(
  'caption' => 'Název',
  'field' => 'nazev',
  'width' => 260,
  'displayCaption' => false,
));
$edtPopis = new TEditBox($lstSluzby,'edtPopis',array(
  'caption' =>'Popis',
  'field' =>'popis',
  'newRow' => false,
  'width'=>400,
  'displayCaption' => false,
));
```

obrazovka č. 14 - **Formulář pro služby**

◄ 1 ►

Snídaně	Po cestě do destinace je zdarma podávána studená snídaně.
Oběd	V restauračním zařízení destinace je smluven hromadný oběd s moż
Vstupné	V ceně výletu je i vstupné do hlavních částí památky.
Prohlídka	V ceně výletu je společná prohlídka s průvodcem.
Cesta autobusem	Doprava je zdarma zajištěna autobusem.
Cesta vlakem	Do destinace se dopravíme vlakem projíždějícím zdejší malebnou kr
Bezbariérový přístup	Autobus i destinace jsou připraveny pro handicapované osoby.
Plná penze	Na každý den výletu je pro vás připravena snídaně, oběd i večeře.

```
http://server/sluzby.php
```

A.7.2 Ukládání formuláře

Na stránce jsou teď zobrazeny všechny záznamy přímo v textových vstupních polích a uživatel tak může upravovat více položek najednou. Bohužel uživatel zatím nemá jak upravené záznamy uložit. Pro zprostředkování ukládání záznamů v `TControlPager` slouží komponenta `TMultiTableLink`. Ta je analogická ke komponentě `TTableLink` s tím rozdílem, že místo SQL dotazu ve vlastnosti `$query` se u ní nastavuje pouze název komponenty `TControlPager` pomocí vlastnosti `$pagerControl`. Po kliknutí na tlačítko, které je automaticky umístěno na stránku, projde komponenta všechny záznamy a případné změny uloží.

výpis kódu č. 61 - **Ukládání formuláře s více záznamy**

```
$tlb = new TSubmitToolBar($frm,'tlb');
$sluzby = new TMultiTableLink($tlb,'sluzby',array(
  'caption' => 'Uložit',
  'default' => true,
  'pagerControl' => 'lstSluzby',
));
$btnZpet = new TLinkButton($tlb,'btnZpet',array(
  'caption' => 'Zpět',
  'href' => '/index.php',
));
```

obrazovka č. 15 - **Ukládání formuláře s více záznamy**

Tučná pole jsou povinná Uložit Zpět

A.7.3 Přidání a odebrání záznamu

Jak vidíte, změna více záznamů najednou není pro ROAD žádný velký problém. V katalogu výletů ale určitě bude potřeba služby nejen upravovat, ale i vytvářet a mazat. ROAD framework nabízí dvě komponenty, které se o tyto akce ve spojení s komponentou `TControlPager` starají. Obě dvě komponenty jsou v prohlížeči reprezentovány klasickými tlačítky.

Přidání záznamu

Pokud komponentu `TInsertButton` umístíme do formuláře a propojíme jí s komponentou `TControlPager`, přidá po kliknutí uživatele jeden prázdný záznam. Záznam je přidán pouze v paměti a dokud uživatel formulář neuloží, změny se nijak neprojeví do databáze. Komponenta `TInsertButton` nabízí i vlastnost `$howMany`, přes kterou lze nastavit, kolik nových záznamů se má po kliknutí přidat. Vytvoříme tedy dvě tlačítka, z nichž jedno bude přidávat jeden záznam a druhé pět záznamů. Obě tato tlačítka umístíme do lišty tlačítek k tlačítkům *Uložit* a *Zpět*.

výpis kódu č. 62 - **Tlačítka pro přidání nového záznamu**

```
$btnPridat = new TInsertButton($tlb,'btnPridat',array(
  'caption' => '+ 1',
  'title' => 'přidat 1 službu',
  'pagerControl' => 'lstSluzby',
));
$btnPridat5 = new TInsertButton($tlb,'btnPridat5',array(
  'caption' => '+ 5',
  'title' => 'přidat 5 služeb',
  'pagerControl' => 'lstSluzby',
  'howMany' => 5,
));
```

obrazovka č. 16 - **Tlačítka pro přidání nového záznamu**

```
http://server/sluzby.php
```

Po přidání je sice logické, že textová pole tohoto záznamu jsou prázdná, ale zároveň díky tomu poněkud ztrácíme informaci o tom, jaké do těchto polí patří hodnoty. I s tím ale ROAD počítá a tak stačí komponentám `TEditBox` přidat jednu CSS třídu, díky níž pak JavaScriptové jádro frameworku automaticky vyplní prázdná pole jejich popisky informativním šedým písmem.

výpis kódu č. 63 - **Neprázdné prázdné textové pole**

```
$edtNazev = new TEditBox($lstSluzby,'edtNazev',array(
  ...
  'cssClass' => 'rNotBlank',
));
$edtPopis = new TEditBox($lstSluzby,'edtPopis',array(
  ...
  'cssClass' => 'rNotBlank',
));
```

Odstranění záznamu

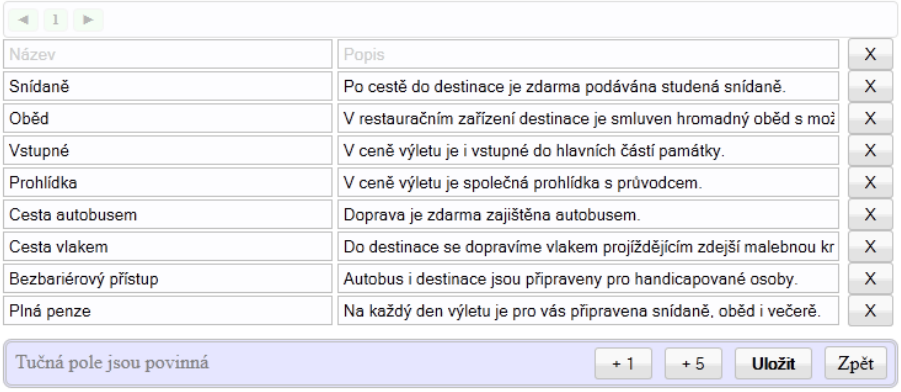
Odstranění záznamu lze implementovat stejně rychle jako jeho vložení. Stačí nám k tomu komponenta `TDeleteButton`, kterou vložíme za textová pole do komponenty `TControlPager` tak, aby byla zreplikována s každým záznamem. U komponenty `TDeleteButton` se nám bude velmi hodit vlastnost, kterou tato

komponenta dědí z komponenty `TSubmitButton`, a to vlastnost `$confirmText`. Ta totiž, pokud obsahuje libovolný dotaz, zajistí, že uživatel bude muset nejprve potvrdit své rozhodnutí o odstranění záznamu.

výpis kódu č. 64 - **Tlačítko pro odstranění záznamu**

```
$btnSmazat = new TDeleteButton($lstSluzby,'btnSmazat',array(
  'caption' => 'X',
  'title'=> 'odstranit službu',
  'confirmText' => 'Opravdu chcete odstranit tuto službu?',
));
```

obrazovka č. 17 - **Kompletní správa služeb**



`http://server/sluzby.php`

A.8 Přihlášení uživatele a omezení práv

Katalog výletů tvoříme jako internetovou aplikaci a u každé internetové aplikace musíme počítat s tím, že si k ní může najít cestu téměř kdokoliv. Je tedy důležité rozlišit, zda k aplikaci přistupuje zákazník či jiný „veřejný“ uživatel, nebo zaměstnanec cestovní kanceláře, který výlety spravuje. ROAD framework poskytuje intuitivní mechanizmus pro identifikaci přihlášeného uživatele v rámci třídy `TSecurity`, jejíž instance je trvale uložena ve vlastnosti `$security` jádra frameworku.

A.8.1 Formulář pro přihlášení uživatele

Z hlediska práce s ROAD frameworkem není samotný návrh formuláře ani kontrola zadaných přihlašovacích údajů nijak zajímavá. Proto se zaměříme pouze na malý útržek kódu a zbytek si můžete prohlédnout v souboru `login.php` v přiloženém archivu `cestovka.zip`.

Zatímco programátor musí pro přihlášení uživatele provést veškeré nezbytné výše zmíněné kroky, pro ROAD framework se přihlášení omezuje na pouhé zavolání metody `login()` třídy `TSecurity`. Metoda očekává jako první parametr informace o přihlašovaném uživateli. Tento parametr může nabývat jakékoli hodnoty libovolného datového typu a je jen na programátorovi, jaký údaj/údaje v něm

předá. Důležité je, že po zbytek sezení, budou tyto informace přístupné přes vlastnost `$userInfo` a programátor tak může kdykoliv tyto informace přečíst a použít.

Druhý parametr, který metoda `login()` očekává, je úroveň oprávnění přihlašovaného uživatele. Úroveň může být reprezentována buď číselnou hodnotou, přičemž čím vyšší hodnota, tím vyšší oprávnění, nebo pseudo-adresářovou strukturou, o které se více dozvíte v sekci *Autentizace a autorizace* kapitoly *4.5.1. Třída TRoad* samotné diplomové práce. V našem případě si vystačíme s číselnou úrovní oprávnění.

výpis kódu č. 65 - **Přihlášení uživatele**

```
$uzivatel = $dataset->fetchObject();
$this->road->security->login($uzivatel,(int)$uzivatel->typ);
```

Podívejte se na úryvek kódu výše a představte si, že objekt `$dataset` obsahuje záznam z tabulky *uzivatele* odpovídající právě přihlašovanému uživateli. Uložíme data ze záznamu do standardního objektu a předáme tento objekt metodě `login()`, která ho uloží pro pozdější použití. Jako úroveň oprávnění použijeme pole *typ*. Přetypování do datového typu *integer* je zde velmi důležité, neboť řetězcové hodnoty považuje metoda `login()` za vyjádření pseudo-adresářové struktury.

A.8.2 Vyhodnocení oprávnění

Vyhodnocení oprávnění probíhá v ROAD pomocí metod `isAllowed()` a `requireRight()`. Obě metody očekávají jako svůj jediný parametr požadovanou úroveň oprávnění. Metody se však liší svým chováním v případě, že uživatel nedosahuje požadovaného oprávnění. Metoda `isAllowed()` v tomto případě jen vrátí logickou hodnotu **`false`**, ale metoda `requireRight()` rovnou ukončí generování stránky a informuje uživatele o nedostatečném oprávnění.

výpis kódu č. 66 - **Vyhodnocení oprávnění**

```
$road->security->requireRight(1);

if($road->security->isAllowed(99)) {
  ...
}
```

Jelikož nechceme, aby nám kdokoliv upravoval výlety, vložíme na začátek souboru `upravit-vylet.php` kontrolu, zda přihlášený uživatel dosahuje alespoň úrovně 1:

výpis kódu č. 67 - **Omezení přístupu k formuláři pro úpravu výletu**

```
<?php
  require_once('road/run.php');
  $page->title = 'Úprava výletu';
  $road->security->requireRight(1);
  ...
?>
```

Když se nyní pokusíte dostat na formulář pro úpravu výletu, jediné co byste měli vidět je:

obrazovka č. 18 - **Omezení přístupu k formuláři pro úpravu výletu**

Pro provedení této akce nemáte dostatečná oprávnění.

```
http://server/upravit-vylet.php?vylet-keys=4
```

Co se týče správy služeb, budeme trochu přísnější a povolíme jí pouze uživateli typu administrátor, tedy uživateli, u nějž má pole *typ* hodnotu alespoň *99*.

výpis kódu č. 68 - **Omezení přístupu ke správě služeb**

```
<?php
  require_once('road/run.php');
  $page->title = 'Služby';
  $road->security->requireRight(99);
  ...
?>
```

Jiný problém budeme řešit u seznamu výletů, který by měl být sice veřejně přístupný, ale neměl by nabízet nepřihlášeným návštěvníkům odkaz směrující na nepřístupný formulář pro úpravu výletu. K tomuto účelu použijeme rozhodování pomocí metody `isAllowed()`:

výpis kódu č. 69 - **Skrytí odkazu na úpravu výletu pro nepřihlášené návštěvníky**

```
if($road->security->isAllowed(1)) {
  $seznam->nazev->onGetHref = '> return "/upravit-vylet.php?vylet-keys=".$data->getRowId();';
}
```

Jelikož v našem konkrétním případě nepotřebujeme ve skutečnosti hlídat úroveň oprávnění, ale pouze to, zda je přihlášen nějaký uživatel, můžeme místo metody `isAllowed()` použít metodu `isLoggedIn()`, která právě tuto informaci poskytuje.

A.8.3 Vyžádání přihlášení

Přijde mi trochu hrubé vůči uživateli, zobrazit mu, pokud není přihlášen, pouhou hlášku o tom, že není oprávněn k dané akci. Mnohem lepší by bylo, kdyby byl uživateli automaticky otevřen přihlašovací formulář, kterým by mohl prokázat, že k akci oprávněný je. Pro realizaci této funkce využijeme událost `onDenied` třídy `TSecurity`. Jelikož potřebujeme, aby vyžádání přihlášení fungovalo celoplošně, musíme obsluhu této události umístit do zaváděcího souboru `road/run.php` a to ještě před inicializaci jádra, tedy volání `$road->init()`.

výpis kódu č. 70 - **Vyžádání přihlášení**

```
function pageDenied($sender, $operation)  {
  if(!$sender->isLoggedIn())  {
    // upozorníme uživatele na nutnost přihlášení
    $sender->road->queueMessage('Pro přístup k této stránce se musíte
přihlásit',MSG_ERROR);
    // a přesměrujeme ho na přihlašovací formulář
    $sender->road->redirect('/login.php?returnTo='.urlencode(relativeUri(':')));
  }
}
$road->security->onDenied = 'pageDenied';
$road->init();
```

Parametr `returnTo`, který předáváme přihlašovacímu formuláři, určuje, na jakou stránku má být uživatel po úspěšném přihlášení přesměrován. Tato funkce je implementovaná přímo v souboru `login.php` a není součástí frameworku.

obrazovka č. 19 - **Vyžádání přihlášení**

```
http://server/upravit-vylet.php?vylet-keys=4
```

Uživatelské účty v naší databázi:

login	admin	lucie	karel
úroveň	99	1	1

Všichni uživatelé mají implicitně prázdné heslo.

A.9 Vlastní šablona

Většina dnešních uživatelů webu je zvyklá, že stránky mají vesměs stálou strukturu. V horní části stránky očekávají logo, pod ním nebo po stranách stránky navigační menu s odkazy do nejdůležitějších sekcí webového serveru a v patičce stránky různé další informace o firmě či stránkách samotných. My si proto ukážeme, jak doplnit tyto obvyklé prvky do našeho katalogu bez toho, abychom je museli vytvářet na každé stránce. K úpravě struktury stránky nám v ROAD slouží mechanizmus šablon. Každý programátor si díky němu může vytvořit odlišný vzhled a strukturu aplikace bez toho, aby zasahoval do souborů frameworku.

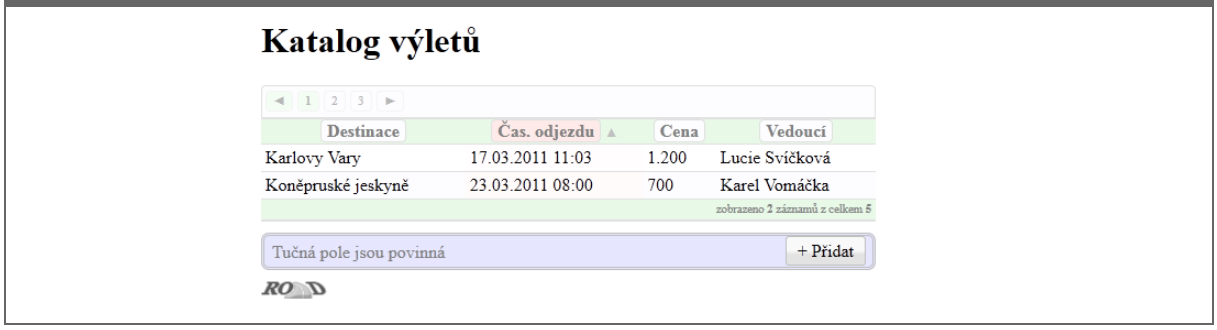
A.9.1 Vzorová šablona

Tvorba šablony je vlastně tvorba vlastního render-setu. Náš render-set si prozatím vystačí s přizpůsobením vzhledu stránky jako celku a proto budeme přepisovat jen render komponenty `TPage`. Šablony, neboli uživatelské render-sety, jsou umísťovány v adresáři `road/template/`**`renderset`**, kde **`renderset`**

zastupuje název render-setu, ke kterému šablonu tvoříme. Ve standardní instalaci ROAD frameworku je obsažena jedna vzorová šablona pro render-set *full*. Jediný soubor této šablony – `template.php.example` – je tedy umístěn v adresáři `road/template/full`.

Abychom vzorovou šablonu aktivovali, odstraníme z názvu souboru dodatečnou příponu a přejmenujeme tak soubor na `template.php` V tuto chvíli je již soubor ROAD frameworkem považován za platnou šablonu a při generování stránky se ji pokusí načíst.

obrazovka č. 20 - **Vzorová šablona**



`http://server/seznam-vyletu.php`

Když prozkoumáme vzorovou šablonu, zjistíme, že je v ní vytvořen potomek renderu `TXhtmlPage`, který je vzápětí zaregistrován voláním metody `registerRenderer()`. Díky tomu použije ROAD při renderování komponenty `TPage` namísto implicitního renderu právě tento přizpůsobený. Třída přizpůsobeného renderu přepisuje tři metody:

- `renderHeader()`
  - je volána, když je potřeba vykreslit záhlaví stránky
  - v kódu této metody se očekává vygenerování záhlaví včetně navigační lišty a dalších obecných prvků stránek
- `renderFooter()`
  - je volána, když je potřeba vykreslit zápatí stránky
  - v kódu této metody bude většinou umístěno generování copyrightu, oznámení o validitě stránky, či jiná případná upozornění
- `beforeRender()`
  - je volána těsně před tím, než začne renderování stránky
  - v kódu této metody je vhodné ke stránce připojit všechny potřebné stylové soubory a případné soubory se skripty pomocí metod `metaAddStyle()` a `metaAddScript()`

A.9.2 Přizpůsobení šablony potřebám aplikace

Pro potřeby naší aplikace rozšíříme záhlaví o lištu odkazů do hlavních sekcí aplikace a zápatí o informaci o čase potřebném k vygenerování stránky. Abychom aplikaci alespoň trošku přikrášlili, připojíme ke stránce jednoduchý CSS stylový soubor.

Záhlaví

Generování XHTML kódu probíhá, v rámci renderů ROAD frameworku, pomocí metod třídy `TOutput` jako `rTagOpen()`, `rTagClose()`, `rTagText()`. Metody většinou vyžadují jako své parametry název tagu, případně obsah a také seznam atributů ve formě asociativního pole. Výhody používání těchto metody oproti vypisování XHTML kódu pomocí příkazu **`echo`**, nebo přímo mezi značkami `?>` a `<?php`, jsou popsány v kapitole *4.8.1* samotné diplomové práce.

Do záhlaví přidáme v podobě tlačítek odkazy do hlavních sekcí aplikace. Z jakékoliv stránky aplikace se tak uživatel bude moci dostat na hlavní stránku, na seznam všech výletů, do přihlašovacího formuláře a v případě, že dosahuje požadovaného oprávnění, i do správy seznamu služeb.

výpis kódu č. 71 - **Záhlaví stránky**

```
/* Here you can draw your divs, headers etc. */
$this->output->rTagText('h1',CFG_APPLICATION_NAME);

$this->output->rTagOpen('div',array('class'=>'menu'));
$this->output->rLinkText('/index.php','Hlavní stránka',array('class'=>'rButton'));
$this->output->rLinkText('/seznam-vyletu.php','Všechny výlety',array('class'=>'rButton'));
if($this->road->security->isAllowed(99)) {
  $this->output->rLinkText('/sluzby.php','Seznam služeb',array('class'=>'rButton'));
}
$this->output->rLinkText('/login.php','Přihlášení',array('class'=>'rButton'));
$this->output->rTagClose('div');
```

Zápatí

Ještě mnohem důležitější, než pohodlí při psaní webové aplikace, je, aby byla tato aplikace snadno použitelná pro uživatele. Pohodlí, které uživatel při práci s touto aplikací má, je určitě negativně ovlivňováno dobou její odezvy. Přidáme si proto do zápatí každé stránky informaci o tom, jak dlouho serveru trvalo, než tuto stránku vygeneroval. Díky tomu budeme mít již při vývoji alespoň rámcový přehled o tom, jakou bude mít která stránka odezvu.

výpis kódu č. 72 - **Zápatí stránky**

```
/* Here you can draw your copyrights, cross-links and validity proofs. */
$this->output->rTagOpen('div',array('class'=>'footer'));
// spread the ROAD across the universe
$this->output->rRunningOnTheRoad();
$this->output->rCode(
  'vygenerováno za <strong>'
  .number_format($this->road->getExecutionTime(),3,',','').'</strong> s'
);
$this->output->rTagClose('div');
```

Stylový soubor

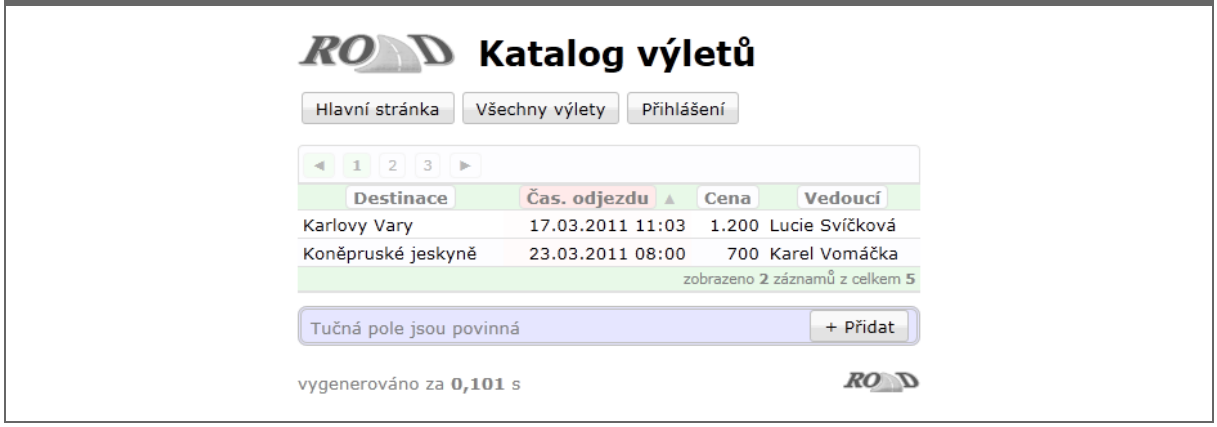
Aby vypadala naše aplikace k světu, využijeme metodu `beforeRender()` a ke stránce připojíme CSS stylový soubor, ovlivňující především vzhled záhlaví a zápatí.

výpis kódu č. 73 - **Připojení stylového souboru**

```
public function beforeRender(TComponent $component, $flags = null) {
  parent::beforeRender($component,$flags);

  /* This is a proper place to link all needed style and script files. */
  $this->metaAddStyle('/'.relativePath(dirname(__FILE__)).'/style.css');
  //$this->metaAddScript('/'.relativePath(dirname(__FILE__)).'/script.js');
}
```

obrazovka č. 21 - **Stránka se záhlavím, zápatím a připojeným stylovým souborem**




```
http://server/seznam-vyletu.php
```

A.10 Vylepšení titulní stránky

Návštěvníky katalogu určitě potěší informace o aktuálním čase, která se na naší titulní stránce neustále aktualizuje. Co by je ale určitě potěšilo více, je nabídka výletů. Přeci jen kvůli těmto výletům do katalogu zavítali. Samozřejmě bychom jim mohli nabídnout strohý seznam všech výletů tak, jak se zobrazuje na stránce `seznam-vyletu.php`, ale myslím si, že zobrazit jen několik zajímavých výletů včetně popisu, seznamu zahrnutých služeb a hlavně poutavých obrázků, bude o poznání lepší. Určitě použijeme komponentu `TControlPager`, abychom se nemuseli starat o stránkování a definování každého výletu zvlášť. Další postup je možné realizovat dvěma způsoby:

- sestavit výlet z více komponent (`TStaticText`, `TPicture`...)
- vytvořit vlastní komponentu a její render(y)

Druhý způsob je vhodnější hned z několika důvodů:

- kratší odezva aplikace (v paměti se generuje méně objektů)
- řešení je snadno znovupoužitelné (stačí vytvořit instanci komponenty)

- na komponentu může být navázáno více renderů (pro různá koncová zařízení)

V souboru `index.php` tedy nejprve nadefinujeme jednoduchou komponentu `TVylet`, která bude mít vlastnost `$sluzby` obsahující pole názvů všech služeb, které tento výlet zahrnuje. V kódu metody `load()`, která je volaná v okamžiku, kdy jsou komponentě přiřazena data voláním metody `setDataset()`, naplníme vlastnost `$sluzby` tím, že provedeme dotaz do databáze a vrácené hodnoty naformátujeme do jednorozměrného pole.

výpis kódu č. 74 - **Vlastní komponenta**

```
class TVylet extends TComponent {
  public $sluzby;
  public function load() {
    $this->sluzby = $this->getDb()->query(
      'SELECT nazev FROM sluzby
       WHERE id IN
         ( SELECT sluzba FROM sluzby_vyletu
           WHERE vylet = '.$this->dataset->fields['id']->value->getAsSqlQuoted().'
         )'
    )->fetchValueList(0);
  }
}
```

Samotná komponenta nám ale neříká nic o tom, jak bude vypadat na obrazovce prohlížeče. K zobrazení komponenty je potřeba vytvořit i její render. V rámci našeho nového renderu nás v našem příkladě bude zajímat pouze metoda `renderContent()`. V ní musíme vygenerovat XHTML kód komponenty `TVylet`. Generovat budeme stejně jako v minulé kapitole pomocí metod třídy `TOutput`. Pro přístup k údajům o výletu budeme používat metody třídy `TDataSet`, jejíž instance bude uložena ve vlastnosti `$dataset` komponenty `TVylet`.

výpis kódu č. 75 - **Renderování vlastní komponenty**

```
  class TXhtmlVylet extends TXhtmlRenderer {
    function renderContent(TComponent $component,$flags = null) {
      $this->output->rTagOpen('div',$this->getAttributes($component));
      $this->output->rTagText('h3',$component->dataset->getAsDisplay('nazev'));

      if($component->dataset->hasValue('obrazek')) {
        $this->output->rImg(  // fotografie
          $component->dataset->getAsSql('obrazek'),
          $component->dataset->getAsDisplay('nazev')
        );
      }
      if($component->dataset->hasValue('souradnice_ss')
        && $component->dataset->hasValue('souradnice_vd')
      ) { // odkaz na Google Maps
        $this->output->rLinkText('http://maps.google.com/maps?q='
          .$component->dataset->getAsSql('souradnice_ss')
          .','.$component->dataset->getAsSql('souradnice_vd'),
          'na mapě',array('class'=>'naMape')
        );
      }
      $this->output->rTagText(  // seznam služeb
        'p',
        implode(', ',$component->sluzby),
        array('class'=>'sluzby'));
      $this->output->rTagText(  // popis
        'p',
        $component->dataset->getAsDisplay('popis'),
```

```
        array('class'=>'popis')
      );
      $this->output->rTagText(  // cena
        'div',
        $component->dataset->getAsDisplay('cena').' Kč',
        array('class'=>'cena')
      );
      $this->output->rTagText(  // vedouci
        'div',
        $component->dataset->getAsDisplay('jmeno'),
        array('class'=>'vedouci')
      );
      $this->output->rTagClose('div');
    }
  }
  // registrace nového renderu
  $this->output->registerRenderer(new TXhtmlVylet($this),'TVylet');
```

Render můžeme umístit přímo do souboru `index.php`, kde ale musíme celou jeho definici i registraci obalit podmínkou, která kontroluje aktuálně používaný render-set:

výpis kódu č. 76 - **Kontrola aktuálně používaného render-setu**

```
if ($output->type && (OUTPUT_FULL | OUTPUT_MOBILE)) {
  ...
}
```

Druhou možností je umístit render do souboru vlastní šablony. Komponentu pak můžeme bez obav použít kdekoliv v aplikaci, protože příslušný render bude načten zároveň se šablonou. Zároveň díky tomu můžeme zahrnout do mobilní šablony[23] render, který bude generovat odlišný výstup.

A.11 Shrnutí

Účelem této přílohy bylo ukázat, jak rychle a snadno vytvořit jednoduchou webovou aplikaci s pokročilými funkcemi. Celá aplikace byla navržena tak, aby se na malých kusech kódu ukázala co největší část možností, které ROAD framework nabízí. Kdybychom si chtěli ukázat a popsat všechny funkce a jejich varianty, tato příloha by svým rozsahem několikanásobně překročila rozsah samotné diplomové práce. Proto doufám, že jsem při návrhu a vývoji aplikace *Katalog výletů* využil ty nejvýznamnější a nejužitečnější možnosti frameworku.

Snažil jsem se přílohu psát tak, aby byl vývoj naší ukázkové aplikace co nejsnadnější. Pokud jste se ale někde ztratili a nevíte jak dál, nebo aplikaci nechcete vyvíjet od píky, doporučuji se podívat do zdrojového kódu hotové aplikace, který je přiložen v archivu `cestovka.zip`. Pokud se vám aplikaci nechce vyvíjet vůbec, stačí do webového prohlížeče zadat adresu:

http://road.goce.cz/cestovka/

a přihlásit se prázdným heslem pod jedním z těchto uživatelských jmen:

admin, *lucie*, *karel*

23 Zdrojový kód šablony pro mobilní zobrazení je obsažen v souboru `road/template/mobile/template.php`