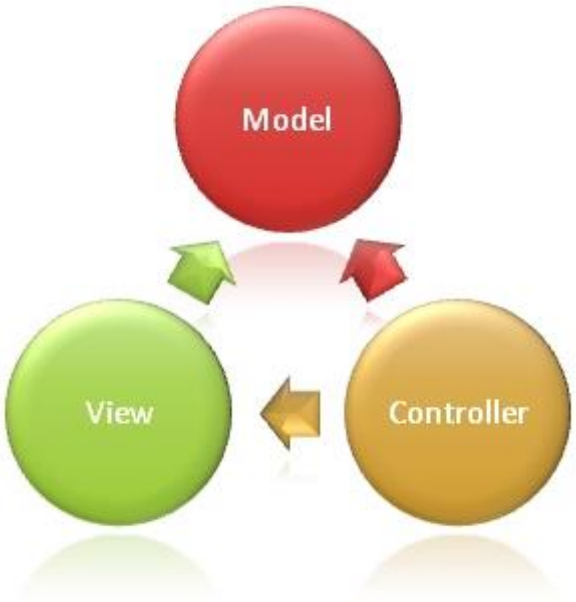
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

CZ.1.07/2.3.00/45.0035
Badatelsky orientovaná výuka ve školním a neformálním vzdělávání

Programování **webových** aplikací

Úvod do tvorby objektově orientovaného MVC frameworku v jazyce PHP

Jan Kubrický



PODPORA

Studijní text vznikl v rámci projektu číslo CZ.1.07/2.3.00/45.0035 „Badatelsky orientovaná výuka ve školním a neformálním vzdělávání“. Projekt byl spolufinancován Evropským sociálním fondem a státním rozpočtem České Republiky.

Tento studijní text slouží rovněž jako doplňkový výukový materiál k předmětům **Tvorba WWW stránek** a **Programování webových aplikací** v rámci výuky na Katedře technické a informační výchovy na PdF UP v Olomouci.

Součásti studijního textu jsou přiložené zdrojové kódy ve složce fw-mymvc.zip.

Co zde nenajdete

Studijní text neobsahuje výuku základů jazyka PHP a MySQL, ani detailní vysvětlení principu OOP. Některá pravidla OOP budou prezentována v rámci řešení jednotlivých etap vývoje frameworku, nicméně jejich podrobný výklad byste měli nastudovat v jiných publikacích či zdrojích. Zde bych mohl doporučit např. publikaci od W. J. Gilmora – Velká kniha PHP5 a MySQL.

Obsah

Úvod

Programovat webové aplikace tak, aby za Vámi byla vidět dobrá práce, tak, abyste se k této práci později rádi vraceli, a tak abyste svou práci pořád dokola z gruntu neměnili v závislosti na nově získaných zkušenostech, vyžaduje poměrně hluboké znalosti a dovednosti. V této souvislosti se jako pomocníci významně osvědčily tzv. **frameworky**, z nichž jmenujme především **ZEND Framework** nebo jeho zdatný český konkurent **Nette**. Tyto nástroje pomohou překlenout bariéru, která je stavěna nezkušenými programátory v nekonzistentním návrhu aplikací, v nichž sekáme chyby a prohřešky proti pravidlům programování na téměř každém řádku kódu.

S použitím frameworků a při dodržování zásad správného návrhu programu se přesouváme z amatérského vývoje aplikací k vývoji téměř profesionálnímu, s kterým je možné překonávat i ty nejnáročnější výzvy. **Český framework Nette a aplikace na něm postavené jsou toho jasným důkazem.**

Nicméně sám jsem poměrně dlouhou dobu přešlapoval před branami použití těchto skvělých nástrojů a nebyl sto je efektivně aplikovat. Důvody byly různé; nedostatek programátorských předpokladů ☹ (autor !== zázrak), nedostatek času k plnohodnotnému studiu a v neposlední řadě neschopnost opustit zažité konvence vlastního (špatného) vývoje aplikací. Až s vytvořením vlastního frameworku mohu částečně říci, že tvorba aplikací ve vzoru MVC vlastně není vůbec složitá a už nikdy více se jí nechci vzdát.

V tomto studijním textu bych Vám rád ukázal, jak lze začít s tvorbou MVC aplikací na bázi jednoduchého, základního Frameworku, který si sami vytvoříme. Představíme si hlavní základ návrhu, který sice nemusí být ve všech ohledech nejlepší ani nejideálnější, ale přiblíží nám strukturu MVC aplikace v dostatečném světle.

Budeme pro to potřebovat:

- Lokální webový a databázový server (Apache, MySQL) – doporučuji použití aplikace XAMPP.
- Vývojové prostřední některého oblíbeného IDE (např. Netbeans).
- Co se vstupních znalostí týče, je potřeba mít základní znalosti:
    - PHP a základní syntaxe při návrhu v OOP.
    - API MySQL v PHP.

Při psaní první verze tohoto studijního textu je využíval:

- XAMPP for Windows 5.6.3.
- Apache 2.4.10 32bit.
- PHP 5.6.3. Poslední stabil verze 5.6.11. Připravovaná verze new PHP 7.
- MySQL 5.6.25. Poslednívývojová verze 5.7.9.

1 Návrhové vzory v PHP

Dříve než se detailněji zaměříme na vývoj jednotlivých částí vlastního frameworku, povíme si něco o návrhových vzorech. Jejich použití je jednou ze základních stavebních částí pokročilého OOP. Představují řešení obvyklých situací, které řeší programátor pořád dokola a s jejich použitím lze docílit správný návrh aplikace, která se dá snadno rozšířit, předávat, upravovat atd.

Návrhových vzorů v PHP je celá řada, z nichž některé jsou využitelné častěji, jiné pouze při specifických požadavcích. Zájemce o vyčerpávající popis odkážu na publikaci Mariana Bohmera – Návrhové vzory v PHP.

My se dále blíže seznáme s třemi pro nás zásadními:

- Front Controller
- Registry
- MVC (Model – View – Controller)

1.1 Front Controller

Tento návrhový vzor bývá v jednodušších aplikacích implementován jedním souborem index.php. Tento soubor (označený tedy jako Front Controller) přebírá požadavky uživatelů a dále je deleguje zodpovědným řadičům (souborům Controller – viz MVC).

Jádro aplikace je tak tvořeno centrálně, obsahuje většinou další informace o konfiguraci atp. Pokud by všechny požadavky byly vždy delegovány přímo na samostatné soubory (např. users.php), museli bychom vždy zajistit připojení souboru obsahují nastavení atd., což by se časem ukázalo jako nepraktické.

1.2 Registry

Drtivá většina aplikací obsahuje objekty ve formě služeb, ke kterým musí mít přístup každá část aplikace. Jedná se o základní funkce, jako jsou např.:

- Zpracování adresy URL,
- Připojení k databázi,
- Emailové požadavky,
- Zpracování data z požadavků POST, SESSION, COOKIE, …

Každému objektu (ať už to bude Controller či Model naší aplikace) tak budeme předávat pouze jeden objekt vzoru Registr, který bude obsahovat vše potřebné. Vyhneme se tak komplikacím spojených s předáváním většího množství samostatně odkazovaných objektů.

Podívejme se nyní, jak jednoduše lze tento návrhový vzor vytvořit. Nebude demonstrovat přesný tvar návrhového vzoru, ale podobu jeho využití v námi vytvářeném frameworku:

```
class Registr {

    private $_services = array();


    /**
     * @param String $service
     * @return Object
     */
    public function getService($service)
    {
        if (!$this->_services[$service]) {
            $this->_services[$service] = new $service();
        }

        return $this->_services[$service];
    }
}


// použití
$registr = new Registr();
```

Vytvořili jsme třídu registru. Obsahuje jednu privátní vlastnost v datovém formátu pole (array), která v sobě bude ukládat již jednou použité objekty.

Třída registr dále obsahuje jedinou metodu getService(), která vrací objekt volané služby. Tedy uvažujme např. objekt třídy Db, která obsahuje databázové připojení.

Při volání ve tvaru:

```
$registr->getService(' Db');
```

Tedy žádám od registru objekt databázového připojení. Registr nejprve prohledá privátní pole _services a pokud tam nalezne tento objekt, okamžitě jej dá k dispozici. Pokud je ovšem objekt

databázové připojení volán z registru poprvé, musí se nejprve objekt databázového připojení vytvořit a uložit. Tento způsob chování je označován jako odložená inicializace a je použita z toho důvodu, aby registr nebyl zbytečně plněn službami, které pro dané požadavky uživatele nemají význam. Např. uživatel pouze prohlíží stránky a je mu tak zbytečná služba pro emailové požadavky.

K úplnosti vzoru registry nám chybí metoda setService($name), která by umožnila přímé uložení zvolené služby do registru. Vypadala by následovně:

```
/**
 * @param String $key
 * @param Object $service
 * @return Object
 */
public function setService($key, $service)
{
    $this->_services[$key] = $service;
}
```

1.3 MVC (Model – View – Controller)

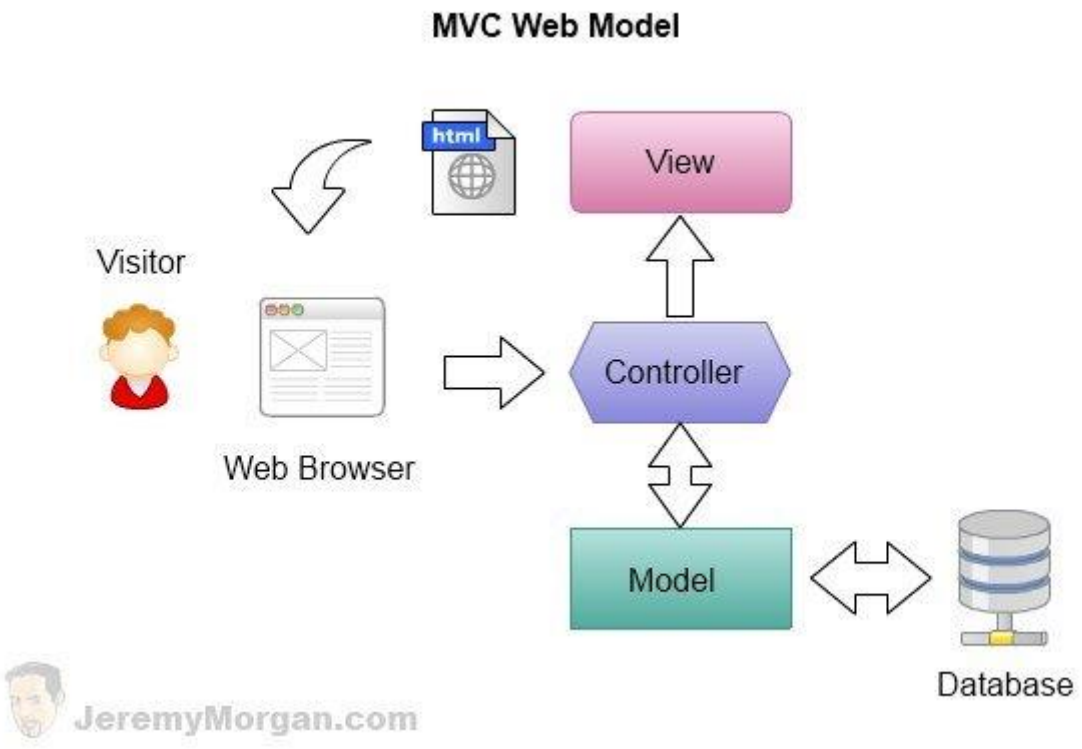
Nyní se dostáváme k tomu nejdůležitějšímu a to je pro nás návrhový vzor MVC. Jeho hlavním úkolem je oddělení logických celků aplikace a vytvoření struktury, která definuje tyto části:

Model – představují ty třídy PHP, které mají za úkol ukládání, správu a zpracování dat. Bude se tedy jednat převážně o třídy zpracovávající data z databáze MySQL.

View – jedná se především o soubory HTML šablon, obrázky, CSS a JavaScript. Šablony HTML nemohou být striktně HTML soubory, musí obsahovat i příkazy pro dynamické vložení dat. Tento problém částěčně eliminují tzv. šablonovací systémy jako např. SMARTY, nicméně tomu se věnovat nebudeme.

Controller – představují ty třídy PHP, které mají za úkol zpracovávat požadavky, komunikaci s modelem a vytváření pohledů (View).

Velmi názorně ukazuje celý princip MVC ilustrace od Jeremyho Morgana.



Obr.: MVC architektura (zdroj: www.jeremymorgan.com)

MVC aplikace je tedy postavena tak, že po požadavku uživatele projde proces jednoduchým sledem kroků. Objekt Controlleru vyhodnotí požadavek (např. filtrování obsahu stránky e-shopu). Zvolený Controller tak kontaktuje vybraný objekt modelu, který poskytne příslušná data z databáze. Controller data zpracuje a sestaví View pro výsledek odeslaný do prohlížeče uživatele. Důležité je, aby každá část MVC aplikace plnila své úkoly a zbytečně se nemíchala do práce jiných vrstev.

1.4 Důkladný Singleton

Na závěr této kapitoly si ještě ukážeme další z častěji využívaných návrhových vzorů, se kterým se můžete běžně setkat. Je to návrhový vzor **Singleton**. Využívá se pro ty případy, kdy se hodí mít k dispozici jedinečný, **globálně dostupný** objekt. Příkladem může být objekt obsahující např. konfiguraci aplikace. Obecný tvar třídy v návrhovém vzoru Singleton, včetně ošetření klonování objektu může vypadat následovně:

```
Class Singleton {
	private static $instance;

	private function __construct()
	{
	}

	public static function getInstance()
	{
		If (!isset(self::$instance)) {
			self::$instance = new self;
		}
		return self::$instance;
	}

	private function clone()
	{
	}
}

// použití
$singleton = Singleton::getInstance();
```

Nejprve si povšimneme, že bylo použito **statických členů třídy**. Jak vlastnost $instance, tak metoda getInstance() jsou statické a je k nim potřeba přistupovat trochu jinak. Uvnitř třídy používáme klíčové spojení **self::** místo **$this->**. Mimo třídu se pak používá celý název třídy a operátor ::.

Objekt Singletonu nejde vytvořit mimo třídu pomocí operátoru new, neboť konstruktor této třídy má obor **private** – nelze volat mimo třídu. Stejně tak mimo třídu nelze objekt Singletonu klonovat, protože má opět obor **private**.

Jediným přístupovým rozhranním Singletonu je tak statická veřejná metoda getInstance(), která má snadnou úlohu. Zkontroluje, zdali již byl vytvořen objekt singletonu a je uložen ve vlastnosti $instance. Pokud ne, vytvoří jej a uloží. Metoda tak vrací vždy jedinečný objekt dané třídy.

Přípravy Frameworku

2 Adresářová struktura

Použitá adresářová struktura vyvíjeného frameworku musí být jasná vzhledem k použité architektuře. To znamená jasné rozdělení controllerů, modelů a pohledů, včetně jejich fyzického oddělení od veřejně přístupné části, která zahrnuje soubor index, složku css a další, které byste nejspíše používali (složku pro uploadované soubory, obrázky, javascript atd.)

Základní struktura:

Hlavní adresář nazvaný **framework**:

- app
- web

Adresář **app** obsahuje:

- app
    - conf – pro konfigurační soubory
    - controllers – složka pro soubory controller
    - models – složka pro soubory modelů
    - services – složka pro třídy objektů služeb a objektu Registry
    - views – složka obsahující šablony členěné do složek

Adresář **web**:

- web
    - css – složka pro soubory kaskádových stylů
    - .htaccess – soubor konfigurace webového serveru
    - Index.php – soubor našeho front controlleru

3 Nastavení lokálního serveru

Pro testovací účely osobně nejraději využívám aplikaci **XAMPP**[1], která sdružuje webový server Apache, databázi MySQL, aplikaci PHPMyAdmin, FTP server FileZilla a některé další nástroje. Při jednoduchém použití jsme asi zvyklí vytvořit novou složku v adresáři webových dokumentů /htdocs/ označenou např. tedy framework a v ní automaticky vytvořit přímo soubor index.php. Adresa v prohlížeči pak vypadá klasicky – http://localhost/framework

Uvedené řešení je ale min. ve dvou ohledech velmi nepraktické:

1. V podstatě znemožňuje umístění veřejně přístupných dokumentů, které v našem frameworku reprezentuje složka **web** do podsložky a oddělit tak aplikační logiku (složka **app**) včetně konfigurace od zranitelné části adresářové struktury.
2. Tvar adresy localhost/framework definuje název serveru localhost a název adresáře framework, což ale neodpovídá reálnému provozu, kde název serveru je reprezentován jednotným názvem, např. www.programyx.cz.

Z těchto důvodů a možnosti se co nejvíce se přiblížit reálnému provozu, vytvoříme si na našem lokálním serveru virtuálního hostitele.

3.1 Virtual Host

Na webu naleznete spousty návodů jak na to, a to nejen v rámci balíku XAMPP, ale rovněž pro různé OS atd. Ukážu vám rychlý návod pro Windows:

- Nejprve si v nějakém editoru např. PS Pad otevřeme soubor pro nastavení virtuálních hostitelů: xampp – apache – conf – extra – http-vhosts.cnf
- **Soubor si zálohujte!**
- Odkomentujeme v něm řádek NameVirtualHost *:80
- A přidáme tento blok pod vzorový blok virtuálního hostitele:

```
<VirtualHost *:80>

    ServerAdmin webmaster@dummy-host.example.com

    DocumentRoot "C:/xampp/htdocs/"

    ServerName localhost

    ServerAlias www.localhost

    ##ErrorLog "logs/dummy-host.example.com-error.log"

    ##CustomLog "logs/dummy-host.example.com-access.log" common
```

[1] https://www.apachefriends.org/index.html

```
</VirtualHost>
```

Tato část konfigurace nám zajistí, že všechny požadavky adresy http://localhost budou stále směřovány do výchozího adresáře pro webové dokumenty /htdocs/

Pod uvedenou konfiguraci vložíme další část:

```
<VirtualHost *:80>

    ServerAdmin webmaster@dummy-host.example.com

    DocumentRoot "C:/xampp/htdocs/framework/web/"

    ServerName fw.local

    ServerAlias www.fw.local

    ##ErrorLog "logs/dummy-host.example.com-error.log"

    ##CustomLog "logs/dummy-host.example.com-access.log" common

</VirtualHost>
```

Samozřejmě Vám doporučuji vytvořit si rovněž adekvátní logovací soubory s patřičným názvem a uložit je do složky **logs** serveru apache. Budou se Vám v nich ukládat logy spojené pouze s provozem tohoto virtuálního hostitele.

3.1.1 NASTAVENÍ SOUBORU HOSTS

Před úspěšným použitím virtuálního hostitele máme ještě dva úkoly. První z nich je zásadní. Musíme našemu počítači říci, že kromě názvu localhost, který je automaticky přesměrován na interní smyčku IP adresy 127.0.0.1, musí toto provádět i s názvem serveru našeho virtuálního hostitele, tj. *fw.local*. Tj. bude potřeba zasáhnout do konfigurace OS. V rámci Windows se tento soubor jmenuje čistě hosts a najdeme ho:

Windows – System32 – drivers – etc

Soubor si opět otevřeme např. v PSPadu a doplníme v něm tento řádek:

127.0.0.1 fw.local

Poznámka:

Soubor hosts bývá většinou chráněn před neoprávněnou změnou zápisu. Budete ho tedy muset otevřít jako správce. Rovněž většina antivirových programů znemožňuje přepis tohoto souboru, a proto jej budete muset v tento moment pozastavit. V rámci 64-bitové verze OS Windows bývají

složky **drivers** nebo **etc** skryty. Je tedy potřeba použít např. Free Commander nebo Total Commander pro zobrazení skrytých souborů či přímé otevření zadané cesty.

Po vytvoření virtuálního hostitele a úpravy souboru hosts restartujte server Apache. Nyní můžeme založit naši adresářovou strukturu a začít pracovat na jádře našeho frameworku.

3.2 Soubor .htaccess

Jak už asi víte, tento soubor je konfigurační soubor webového serveru. Přestože se jedná o jednoduchý textový dokument, může rozhodujícím způsobem ovlivňovat fungování stránek. Soubor .htaccess slouží k základním funkcím na straně serveru, jako je třeba přesměrování/podstrčení stránek, SEO url apod. Tím výrazným způsobem zvyšuje uživatelské pohodlí i správu webu.

Asi nejčastěji se v tomto souboru používá jeden z modulů serveru Apache, který je označen **mod_rewrite** a umožňuje vytvářet tzv. URL friendly. To znamená vyhnout se adresám ve tvaru index.php?page=users&id=3. Mnohem lépe jistě vypadá tvar adresy users/3/. K tomuto účelu bude potřeba využít mod_rewrite. Obsah našeho souboru .htaccess vypadá pro začátek následovně:

```
RewriteEngine On

#RewriteCond %{HTTP_HOST} ^adresa.cz$ [NC]

#RewriteRule ^(.*)$ http://www.adresa.cz/$1 [R=301,L]

RewriteCond %{REQUEST_FILENAME} !-d

RewriteCond %{REQUEST_FILENAME} !-f

RewriteRule . index.php [L]
```

1. První řádek aktivuje modul.
2. Druhý a třetí řádek je zatím okomentovaný, ale v reálném provozu bychom v něm doplnili přesný název našeho serveru a účel je vám asi zřejmý. Všechny požadavky uživatele začínající bez www se automaticky přesměrují na tvar začínající s www. Tím odstraníme nežádoucí duplicitu adres. **RewriteCond** značí podmínku, **RewriteRule** pak přepisovací pravidlo.
3. Řádky 4 a 5 určují dvě podmínky před posledním přepisovacím pravidlem. Pakliže URL adresa obsahuje odkaz na soubor nebo adresář, provede se výchozí akce – tedy např. stažení souboru nebo otevření adresáře. Pokud tomu tak není, všechny požadavky jsou řádkem 6 směrovány na soubor index.php.

Problematika souboru .htaccess je širší a více se ní můžete dočíst třena zde:

http://httpd.apache.org/docs/current/howto/htaccess.html

Tímto jsou nezbytné přípravy za námi a můžeme se pustit do samotného vývoje.

Vývoj Frameworku

Všechny zde uváděné postupy a návrhy jsou pouze zjednodušenou, nicméně funkční alternativou frameworku. Nebudeme se např. zabývat dynamickým načítáním obsahu celých stránek jako v případě použití redakčních systémů atp. Zaměříme se na základní princip práce. Další rozšíření či modifikace jsou již ve vaší režii.

4 Registr služeb

Náš první úkol bude spočívat ve vytvoření registru, jehož účel již známe. Bude nám představovat jeden objekt, který bude poskytovat objekty další – v našem případě služby frameworku. Vytvoříme se tedy soubor třídy (PHP Class) ve složce services a nazveme ho Registr. Jmenná konvence tříd velí, aby první písmeno názvu třídy bylo velké – stejně tak pro název souboru, který třídu obsahuje.

Zdrojový kód:

```
class Registr {

    private $_services = array();


    /**
     * @param String $service
     * @return Object
     */
    public function getService($service)
    {
        if (!$this->_services[$service]) {
            require_once FW_PATH.'app/services/'.ucfirst($service).'.php';
            $this->_services[$service] = new $service();
        }

        return $this->_services[$service];
    }
}
```

Zvykejte si používat komentáře, minimálně pro metody tříd. Základem je vyspání datových typů vstupních argumentů a návratové hodnoty.

Náš registr služeb je přímočarý. Obsahuje vlastnost **services**, která se postupně plní objekty volaných služeb. Z tohoto pole jsou poté metodou getService() objekty poskytovány ostatním částím aplikace.

Metoda getService() přejímá jako argument název služby. Např. db, což bude značit požadavek na objekt řídící přístup k databázi. Nejprve tedy v podmínce kontroluje, zdali je objekt už uložen, a pokud ne, načte nejprve pomocí příkazu require_once soubor s touto třídou, vytvoří objekt této třídy a uloží jej do pole services. Z tohoto pole jsou poté služby poskytovány.

Poznámka:

Všimněte si použití zatím nedefinované konstanty FW_PATH. Ta určuje cestu k hlavnímu adresáři – framework. Tuto konstantu si poté nadefinujeme v souboru index.php. Obecně používání příkazu require či include se v aplikacích moc nedoporučuje, protože je to min. nepraktické. Lze se toho úplně zbavit pomocí tzv. **autoloadingu tříd** a **jmenných prostorů**. Nicméně tomuto se věnovat nebudeme.

4.1 Magická metoda __get()

Určitě jste někdy v souvislosti s OOP slyšeli o magických metodách a min. jednu již použili. Magické metody mají tu schopnost, že jsou volány automaticky při spuštění asociované akce. Už víte tu nejznámější? Ano konstruktor. Ten je spuštěn vždy při vytvoření objektu. Jak vidíme z třídy Registr, jeho definice není povinná a stejně tak je tomu u jiných magických metod.

Podívejme se nyní, jak se bude z registru volat např. služba pro obsluhu databázového připojení:

```
$registr = new Registr;

$db = $registr->getService('Db');
```

Druhý řádek se bude v našem frameworku objevovat častěji a to v delší podobě díky použití $this. Magická metoda __get() nám to pomůže zjednodušit. Tato metoda je volány vždy, když použijete volání nedefinované vlastnosti objektu.

Vzhledem k tomu, že máme pouze jednu vlastnosti třídy nazvanou services, můžeme této skutečnosti využít. Metoda getService() se tedy změní na magickou metodu __get(). Tělo metody zůstává stejné:

```
public function __get($service)
{
    if (!$this->_services[$service]) {
        require_once FW_PATH.'app/services/'.ucfirst($service).'.php';
        $this->_services[$service] = new $service();
    }

    return $this->_services[$service];
}
```

Nicméně volání služby Db již bude vypadat jednodušeji. Srovnejte s předchozím:

```
$db = $registr->Db;
```

Více k magickým metodám v doporučené literatuře. Než se pustíme do další práce, zopakujme si ještě jedno z hlavních pravidel OOP:

PŘÍSTUP K DATŮM VŽDY V RÁMCI TŘÍDY ZAPOUZDŘÍME A POSKYTNEME METODY, POMOCÍ NICHŽ LZE TYTO DATA ZÍSKAT.

5 Konfigurace Frameworku

Zásadním prvkem každé aplikace je jeho nastavení. Jedná se většinou o citlivé údaje, které je potřeba chránit a v žádném případě nemohou být součásti veřejně přístupné složky **web**. Vytvoření konfiguračního souboru se řeší různě. Může to být jednoduchý PHP soubor obsahující konstanty či konfigurační proměnné, může to být soubor jiného formátu než PHP anebo to může být PHP třída. Právě toto poslední řešení použijeme.

V adresáři conf si vytvoříme nový soubor obsahující PHP třídu **Configurator**. Zdrojový kód:

```
class Configurator {


    const

        BASEURL = 'http://fw.local',

        DEFAULTCONTROLLER = 'Page';


    /** DB Connection Parametrs */

    const

        HOST = 'localhost',

        USER = 'root',

        PASSWORD = '',

        DB = 'test';


    /** @var Array URL Mask */

    public static $urlmask = array("controller", "action", "id");

}
```

Tato třída obsahuje základní konstanty, ke kterým poté můžeme přistupovat jako ke statickým členům tříd, tedy voláním názvu třídy::název konstanty.

Např. Configurator::DEFAULTCONTROLLER

V konfiguraci jsme si rovněž vytvořili statické pole **ulrmask** (maska url), která definuje jednotlivé součásti, ze kterých se námi používané URL budou skládat. Vysvětlíme si to na ukázce:

Už víme, že pomocí mod_rewrite se můžeme vyhnout složitým a ne příliš čitelným tvarům adresy URL. Naše adresy by měly vypadat nějak takto:

http://fw.local (v reálu něco ve tvaru http://www.programyx.cz)

- defaultní tvar – index.php předá řízení defaultnímu controlleru.

http://fw.local/user

- První část URL adresy za adresou serveru označíme jako controller. Index.php předá řízení souboru UserController.

http://fw.local/user/new

- Za názvem controlleru následuje název akce tohoto controlleru. Controller UserController tedy musí obsahovat akci (nějakou metodu) new – zřejmě pro vytvoření nového uživatele.

http://fw.local/user/delete/10

- Poslední maskou je ID. Akce odstranění uživatele musí znát jeho ID v databázi (jednoznačný identifikátor), aby mohla být provést akce detete úspěšně.

Ve většině případů si vystačíme s tímto základním rozdělením URL adresy. Pakliže by došlo k potřebě doplnit ji o další parametry, můžeme ji klasicky doplňovat např. takto:

http://fw.local/user/delete/10/?backup=yes

přičemž argument backup získáme jednoduše pomocí některé filtrační funkce – např. **filter_input().**

Masku URL použijeme poté ve službě, která bude mít na starost zpracování URL adresy. Třída Configuratoru vám nabízí prostor pro další potřebné nastavení, které s vývojem frameworku či konkrétní aplikace bude potřebovat.

6 Služby

6.1 Request

Máme k dispozici vše, co budeme potřebovat pro sestavení základních služeb. Tou hlavní je pro nás služba označená Request, která bude zpracovávat 2 hlavní požadavky a to požadavky GET a POST. Požadavky definované v URL se většinou umisťují do zvláštního objektu, kterému e říká Router, nicméně pro jednodušší účely si vystačíme s naším objektem Request. Ten by samozřejmě mohl obsahovat i zpracování požadavků COOKIE a SESSION, ale tuto implementaci si již musíte doplnit sami.

Založíme PHP třídu **Request** v adresáři services. Třída bude obsahovat dvě privátní vlastnosti a konstruktor, který se postará o naplnění těchto vlastností:

```
private $url = array();

private $post = array();


public function __construct()

{

	$this->setUrl();

	$this->post = filter_input_array(INPUT_POST);

}
```

Vlastnosti jsou pochopitelně stejně jako jejich superglobální předchůdci datového formátu pole. Konstruktor nejprve volá metodu **setUrl**(), ke které později. Vlastnost **post** je naplněna okamžitě díky jedné z filtračních funkcí. Tyto funkce bystě měli používat a nepřistupovat napřímo k poli $_POST. Více k těmto filtračním funkcím třeba na W3Schools.com[2].

Dále se podíváme se metodu setUrl(), která musí rozložit URL a uspořádat ji podle zvolené masky v konfiguraci:

[2] http://www.w3schools.com/php/php_ref_filter.asp

```
public function setUrl()

{

        $url = filter_input(INPUT_SERVER, 'REQUEST_URI');

        $urlbits = explode("/", $url);

        $urlcleanbits = array_slice($urlbits, 2);


        foreach($urlcleanbits as $key => $bit) {

            $this->url[Configurator::$urlmask[$key]] = $bit;

        }

}
```

Nejprve si „sejmeme“ pomocí filtrační funkce část URL adresy, která leží za názvem serveru. Máme-li tedy adresu ve tvaru http://fw.local/user/new, získáme v proměnné url řetězec **/user/new**. Musíme tak ocenit název serveru virtuálního hostitele. Pokud bychom totiž pracovali klasicky, získali z http://localhost/framework/user/new řetězec **/framework/user/new**, který by bylo dále potřeba zbavit první části.

Pomocí funkce explode() získáme kousky url v poli, které si uložíme do proměnné urlbits. Ještě je potřeba zbavit se prvního prázdného prvku a máme k dispozici všechny použitelné části v poli urlcleanbits.

Vlastnost objekt **Request** url poté naplníme dvojicí klíč => hodnota s klíčem odpovídajícím masce url na dané pozici. Příklad:

http://fw.local/user/delete/10

```
print_r($this->url)

       controller => user,

       action => delete,

       id => 10
```

Nyní můžeme vytvořit geter, který bude vracet hodnot URL podle volané masky:

```
    /**

     * @param String (controller, action, id) $mask

     * @return mixed

     */

    public function getUrl($mask)

    {
```

```
        if ($this->url[$mask]) {
            return $this->url[$mask];
        }
        else {
            return false;
        }
    }
```

V argumentu metody $mask tak budeme předávat buď řetězec controller, action nebo id a získáme jeho odpovídající hodnotu z URL adresy nebo informaci o tom, že hodnota neexistuje.

V METODÁCH TŘÍD NEZAPOUZDŘUJEME JEN DATA, ALE TAKÉ ALGORITMY, KTERÉ OŠETŘUJÍ JEJICH STAVY.

Poslední metodou je metoda getPost(), která zřejmě nebude potřebovat komentář:

```
    /**
     * @param String $index
     * @return mixed
     */
    public function getPost($index = false)
    {
        if (!$index) {
            foreach($this->post as $key => $val) {
                $this->post[$key] = filter_var($val, FILTER_SANITIZE_STRING);
            }

            return $this->post;
        }
        elseif (isset($this->post[$index])) {
            return filter_var($this->post[$index], FILTER_SANITIZE_STRING);
        }
        else {
            return false;
        }
    }
```

6.2 Db

Služba správy databázového připojení nebude mít na starosti jen připojení jako takové, ale bude pro náš framework vykonávat základní manipulace nad databázovými tabulkami MySQL.

V poslední době je jako rozhranní pro připojení k databázi používáno **PDO** – což je vlastně abstraktní vrstva, která sjednocuje odlišnosti použití různých databázových systémů (např. MSSQL, MySQL, Oracle či PostgreSQL). Rozhodně vám doporučuji se s ním do budoucna seznámit.

Nicméně v této studijní opoře se budeme držet rozhranní používaného přímo pro MySQL a pokusíme se vytvořit příslušnou třídu, tak, aby nám do budoucna umožnila pohodlné změny. A především, aby většinu SQL dotazů sestavovala centrálně.

Opět si tedy založíme novou PHP třídu **Db** v adresáři services.

```
class Db {


    private $connection;

    private $queryCounter = 0;

    private $last;


    /**
     * Create New Connection With DB Server
     * @param String $host
     * @param String $user
     * @param String $password
     * @param String $database
     * @return Int
     */
    public function connection( $host, $user, $password, $database )
    {
        if (!$this->connection) {
            $this->connection = new \mysqli( $host, $user, $password, $database );
            mysqli_set_charset($this->connection, "utf8");


            if (mysqli_connect_errno()) {
```

```
                trigger_error('Chyba při připojování. '.$this->connection->error,
E_USER_ERROR);
            }
        }
    }

    /**
     * Execute SQL
     * @param SQL $queryStr
     * @return queryCounter++
     */
    public function executeQuery($queryStr)
    {
       if (!$result = $this->connection->query( $queryStr )) {
           echo $queryStr;
            trigger_error('Chyba: '.$this->connection->error, E_USER_ERROR);
       }
       else {
             $this->last = $result;
             return $this->queryCounter++;
        }
    }

    /**
     * Return Rows number
     * @return Int
     */
    public function getNumRows()
    {
        return $this->last->num_rows;
    }
```

```
/**
 * Return Results From Query
 * @return Int
 */
public function getRows()
{
    return $this->last->fetch_array(MYSQLI_ASSOC);
}

/**
 * Remove data From Table
 * @param String $table
 * @param String $condition
 * @param Int $limit
 */
public function deleteRecords($table, $condition, $limit)
{
   $limits = ( $limit == '' ) ? '' : ' LIMIT ' . $limit;
   $delete = "DELETE FROM {$table} WHERE {$condition} {$limits}";
   $this->executeQuery( $delete );
}

/**
 * Update data in Table
 * @param String $table
 * @param Array $changes
 * @param String $condition
 * @return boolean
 */
public function updateRecords($table, $changes, $condition)
{
   $update = "UPDATE " . $table . " SET ";
```

```
        foreach( $changes as $field => $value ) {
               $update .= "`" . $field . "`='{$value}',";
        }

        // remove last ','
        $update = substr($update, 0, -1);
        if ( $condition != '' ) {
              $update .= " WHERE " . $condition;
        }
        $this->executeQuery( $update );
        return true;
    }

    /**
     * Insert Data to Table
     * @param String $table
     * @param tArray $data
     * @return boolean
     */
    public function insertRecords($table, $data)
    {
        // setup some variables for fields and values
        $fields  = "";
        $values = "";

         foreach ($data as $f => $v) {
                  $fields .= "`$f`,";
                  $values .= ( is_numeric( $v ) && ( intval( $v ) == $v ) ) ? $v."," : "'$v',";
         }

        // remove last ','
        $setfields = substr($fields, 0, -1);
```

```
   $setvalues = substr($values, 0, -1);


    $insert = "INSERT INTO $table ({$setfields}) VALUES({$setvalues})";
    $this->executeQuery( $insert );
    return true;
}


/**
 * @return Int
 */
public function getLastInsertID()
{
   return $this->connection->insert_id;
}


/**
 * Return Count Queries
 * @return Int
 */
public function getQueryCount()
{
    return $this->queryCounter;
}


/**
 * @param String $data
 * @return Escaped data
 */
public function sanitizeData($data)
{
   return $this->connection->real_escape_string($data);
}
```

```
    /**
     * Close Connection
     */
    public function __destruct()
    {
        $this->connection->close();
    }
}
```

Všimněte si použití rozhranní **mysqli**, které nahradilo starší **mysql**, jenž byste již neměli používat vůbec a které bude z PHP brzy úplně odstraněno!

Postupně si sami projděte jednotlivé vlastnosti a metody. Základem je metoda **connection(),** která naváže spojení. Pro potřeby provedení dotazů SELECT máme k dispozici metodu **executeQuery()**, která načtené výsledky uloží do vlastnosti **last**. Z ní pak můžeme pomocí metod **getRows**() získat jednotlivé řádky nebo jejich počet pomocí metody **getNumRows**().

Pro potřeby provedení příkazů INSERT, UPDATE nebo DELETE máme vytvořeny zvláštní metody, které přebírají název tabulky, data a podmínky. Tyto příkazy SQL tak nebudeme nikde v našich objektech zapisovat (vyjma SELECT), práci za nás provede objekt služby **Db**, což opět výrazně zpřehlednění kód naší aplikace.

Povšimněte si rovněž v závěru použití magické metody **__destruct()**, která je automaticky volána při likvidaci objektu z paměti.

VŽDY SE SNAŽÍME PROGRAMOVAT TAK, ABYCHOM DOSÁHLI ZNOVUPOUŽITELNOSTI KÓDU A NIKOLI JEHO DUPLICITY.

7 Soubor Index

Na řadu přichází výchozí bod, tj. front controller aplikace, soubor index.php. Jak už jsme zmiňovali výše, nebudeme ho řešit jako třídu, ale jako jednoduchý PHP soubor, který centralizuje inicializaci a nastavení frameworku, a řízení poté předává požadovanému controlleru. Výchozí zdrojový kód:

```
<?php

// Front Controller


// Konstanta cesty k frameworku

define('FW_PATH', dirname(__DIR__).'/');


// Zpřístupnění Konfigurace

require_once FW_PATH.'app/conf/Configurator.php';


// Inicializace registru

require_once FW_PATH.'/app/services/Registr.php';

$registr = new Registr();


// Načtení controlleru

$controller = ($registr->Request->getUrl('controller') ? :
Configurator::DEFAULTCONTROLLER);

$contollerClass = ucfirst($controller).'Controller';


// Vytvoříme objekt Controlleru

if (file_exists(FW_PATH.'app/controllers/'.$contollerClass.'.php')) {

    require_once FW_PATH.'app/controllers/'.$contollerClass.'.php';

    new $contollerClass($registr);

}

else {

    require_once '404.php';

}
```

Pro potřeby načítání souborů jsme si vytvořili konstantu **FW_PATH**, která využívá magickou konstantu PHP __DIR__. Tato konstanta vypisuje cestu aktuálního souboru, tj. souboru index.php. Protože ale soubor index.php leží ve veřejně přístupné složce **web** a my budeme potřebovat cesty k souborům, které leží ve složce **app** (složky web a app leží na stejné úrovni), musíme získat cestu k jejich nadřazenému adresáři framework. Pro tento účel jsme použili ještě funkci dirname().

Dále přeskočíme na sekci načtení controlleru, tj. předání řízení podle požadavků uživatele.

Do proměnné $controller si načteme hodnotu controlleru z URL. Pokud v adrese hodnota controlleru chybí, načteme hodnotu defaultního controlleru z konfigurace.

Dále je potřeba vytvořit si název třídy controlleru podle stanovené konvence. Tyto třídy budeme vždy označovat jako **EntitaController**. Např. tedy jako UserController nebo PageController.

Potom tedy URL adresa ve tvaru:

http://fw.local/user

znamená v našem případě vytvoření objektu z třídy UserController. Ověříme existenci stejnojmenného souboru v příslušné složce pro controllery a v případě úspěchu vytvoříme nový objekt:

```
new $contollerClass($registr);
```

Parametr konstruktoru našich controllerů bude obsahovat jeden argument a to objekt **registru** služeb. Uložení registru si centralizujeme v abstraktní třídě **BaseController**, ze které budou všechny další controllery dědit.

8 Bázové třídy

Náš framework bude obsahovat dvě bázové (základní neboli rodičovské) třídy, ze kterých budou odvozeny v prvém případě controllery a v druhém případě modely. Tyto bázové třídy tak budou poskytovat data a funkčnost, která bude centralizovaná v jejich vlastnostech a metodách, a umožní nám tak, vyhnout se duplicitě kódu v třídách, které budou z těchto bázových tříd dědit.

8.1 BaseController

Třída BaseController je bázovou třídou pro controllery. Umístíme ji do složky controllers. Podívejme se nejprve na její zdrojový kód:

```
abstract class BaseController {


    protected $registr;

    protected $title;


    /**
     * @param Registr $registr
     */
    public function __construct(Registr $registr)
    {
        $this->registr = $registr;
    }


    /**
     * @param String $title
     */
    public function setTitle($title)
    {
        $this->title = $title;
    }
```

```
    public function getTitle()
    {
        return $this->title;
    }

    /**
     * Metoda pro přesměrování URL
     * @param String $param
     * @return Header new Location
     */
    public function redirect($param = '', $statusCode = 303)
    {
        header('Location: ' . Configurator::BASEURL.'/'.$param, true, $statusCode);
        exit();
    }
}
```

Třída poskytuje dvě vlastnosti a vyjma konstruktoru tři metody. První vlastnost je určena pro uložení registru služeb, který je uložen v konstruktoru. Druhá vlastnost je řetězcová, a přístup k ní je řízen metodami setTitle() a getTitle(). Jak název napovídá jedná se o titulek webové stránky. Poslední metoda označená redirect() je metoda pro jednoduché přesměrování URL, např. po akcích POST.

K uvedenému zdrojovému kódu bychom měli uvést pár vysvětlení.

8.1.1 ABSTRAKTNÍ TŘÍDY

Třída je označena jako abstraktní (klíčové slovo **abstract** na začátku její definice). Abstraktní třídy jsou novinkou PHP5. Abstraktní třídy je vhodné vytvářet tehdy, když nechceme umožnit z této třídy vytvoření objektu.

8.1.2 OBOR PROTECTED

Určitě jste si všimli použití oboru protected u vlastností bázové třídy. Protected používáme tehdy, pokud tyto vlastnosti nebo metody mohou využívat i potomky této třídy, což u **private** nelze! To znamená konkrétně controllery, které budou z třídy BaseController dědit. Obor protected lze používat i pro metody.

8.1.3 *Určení typu parametru*

V hlavičce magické metody konstruktor vidíme před samotným parametrem určení jeho typu. Před proměnnou je uvedeno, že $registr musí být instancí třídy (objektem) Registr. Konstruktor poté akceptuje jen ty objekty, vytvořené z třídy Registr. Kromě tříd je možné specifikovat i rozhranní[3], nebo pole. **Skalární typy jako např. číslo nebo řetězec omezit nelze.**

8.2 BaseModel

Třída BaseModel je bázovou třídou pro modely. Umístíme ji do složky models. Podívejme se nejprve opět na její zdrojový kód:

```
abstract class BaseModel {


    protected $db;


    /**
     * Naváže připojení s databází
     * @param Db $db
     */
    public function __construct(Db $db)
    {
        $this->db = $db;
        $this->db->connection(Configurator::HOST, Configurator::USER,
Configurator::PASSWORD, Configurator::DB);
    }
}
```

Konstruktor bázové třídy BaseModel přejímá vstupní parametr objektu Db. Nejprve si odkaz na tento objekt uloží do své vlastnosti a poté provede připojení k databázi. K tomu nutně potřebuje připojovací údaje, které získá z konstant třídy Configurator.

Možná se nyní leknete, zdali každá třída modelu, která bude dědit z třídy BaseModel si vytvoří své vlastní vlákno připojení k db serveru, což by bylo jistě nežádoucí. Toho se nemusíme obávat. Jak určitě víte, od verze PHP5 jsou objekty předávány odkazem a nikoli hodnotou. To znamená, stále odkazujeme na původní objekt Db, který byl vytvořen v registru. Navíc pokud si prohlédnete metodu

[3] Rozhranní je v OOP kolekce neimplementovaných definic metod a konstant. Slouží jak jistý náčrt třídy. Více k rozhranním na: http://php.net/manual/en/language.oop5.interfaces.php

connection třídy Db, ta připojení naváže pouze tehdy, pokud ještě nebylo uloženo do vlastnosti connection.

Aplikace

9 Default Controller

Tak v této chvíli je jádro aplikace (náš jednoduchý framework) připraven k použití a my se můžeme pustit do práce na nějaké aplikaci. Vytvoříme si aplikaci jednoduché správy uživatelů. Pro začátek si zpracujeme defaultní controller, jehož úkolem je v podstatě jen nastavit odpovídající titulek stránky a zobrazit šablonu úvodní stránky.

Ve složce controllers si proto vytvoříme novou tříd u controlleru PageControler – její název jsme si určili jako defaultní v konfiguraci. Zde je její zdrojový kód:

```
require_once FW_PATH.'app/controllers/BaseController.php';


class PageController extends BaseController {


    public function __construct(Registr $registr)

    {

        parent::__construct($registr);

        $this->actionDefault();

    }


    public function actionDefault()

    {

        $this->setTitle('Úvodní stránka aplikace správy uživatelů');

        $this->renderDefault();

    }


    public function renderDefault()

    {
```

```
        require_once FW_PATH.'app/views/Page/default.php';
    }
}
```

První controller jistě nevypadá složitě. Nicméně provedeme si jeho detailní rozbor, aby nám pro další práci nevznikaly nejasnosti.

9.1 Dědění třídy

Třída PageController dědí z třídy BaseController. Z toho důvodu jsme nejprve zpřístupnili bázovou třídu příkazem require. Třídě PageController se tak říká dceřiná třída, podtřída nebo odvozená třída.

Dědění se dociluje tak, že za názvem dceřiné třídy připojíme klíčové slovo **extends** a za ním název bázové třídy.

9.1.1 KONSTRUKTOR

Pro konstruktory mohou v případě dědění nastat různé případy:

1. Má-li rodičovská třída konstruktor a dceřiná třída ne, vykoná se automaticky konstruktor bázové třídy.
2. Má-li dceřiná třída konstruktor, vykoná se automaticky bez ohledu na to, zdali rodičovská třída konstruktor má či nikoli.
3. Má-li dceřiná třída konstruktor a rodičovská taktéž, a chceme-li vykonat oba dva, konstruktor dceřiné třídy musí obsahovat příkaz volání konstruktoru rodičovské třídy. Takto:

   ```
   parent::__construct(parametry rodičovského konstruktoru)
   ```

Poslední bod je přesně náš případ. Chceme vykonat rodičovský konstruktor, který uloží parametr registru, a zároveň chceme vykonat dceřiný konstruktor, který volá svou interní metodu. Parametr konstruktoru bázové třídy BaseController přebírá objekt třídy Registr. Ten si uloží do své chráněné (protected) vlastnosti. Jelikož má tato vlastnost obor protected, mají k ní přístup i dceřiné třídy, stejně jako mají přístup k metodám setTitle(), getTitle() a redirect() – ty mají totiž obor public, čili veřejný.

9.2 Životní cyklus controlleru

Přichází na řadu proces, který označujeme jako životní cyklus controlleru. Třída controlleru musí totiž definovat události, které vedou až k sestavení šablony posléze odeslané uživateli. Tato cesta (cyklus)

může mít před sebou vícero úkolů. Např. volání třídy modelu pro uložení či získání dat, validaci, odeslání emailu, přesměrování, ajaxové požadavky atd.

Např. PHP framework Nette pro to poskytuje předem připravené typy metod. Ty jsou volány automaticky v daném pořadí, což vývojáři umožňuje soustředit se pouze na jejich obsah a nikoli na jejich volání. V našem případě musíme řešit obojí, protože algoritmus automatického volání metod nemáme definován. Nicméně obejdeme se bez něj. Nejprve si určíme způsob pojmenování jednotlivých metod životního cyklu. A to nebude nic složitého, protože budou pouze dvě:

1. Metody ACTION – které budou vždy první a budou obsahovat akce, jako uložení dat, přihlášení uživatele, přesměrování na jinou URL atd.
2. Metody RENDER – které jak název napovídá, vykreslují šablony a rovněž načítají data do těchto šablon.

Pro každý Controller bude zřejmě existovat nějaká výchozí akce. Tuto akci si označíme default a přizpůsobíme jí i action a render metodu.

Jak vidíme na příkladu PageController, v konstruktoru je volána nejprve metoda actionDefault(). Tato metoda pouze nastaví titulek a volá metoda renderDefault(). Jejím úkolem je zatím jen vložení příslušné šablony.

10 View

Šablony aplikace budou umístěny ve složce **/views/**. V ní si vytvoříme složku označenou **/inc**. V této složce budou ležet dva základní soubory:

- head.php – obsahující HTML hlavičku a začátek těla stránky včetně menu stránek.
- bootom.php – obsahující úplný závěr zdrojového kódu HTML, včetně přiložených skriptů.

Tyto dva soubory si můžete prohlédnout v přiložených souborech a jejich zdrojových kódech; zde je uvádět nebudu.

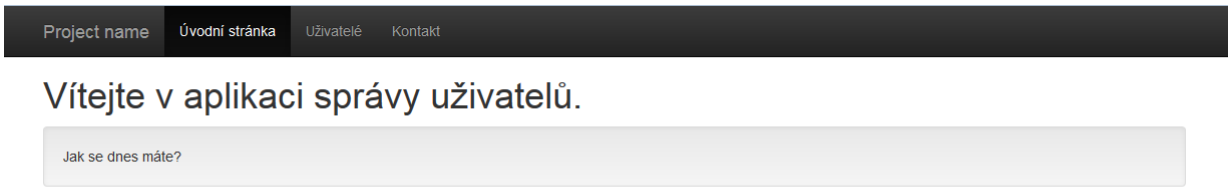
Oba soubory budeme rovněž přikládat k dílčím šablonám jednotlivých akcí konkrétních Controllerů. Ve složce views si proto vždy vytvoříme další složku, která ponese název daného Controlleru – např. tedy složku **Page**. V ní již budou umístěny šablony, jejichž název bude zase shodný s názvem akcí Controlleru. Složka Page tak bude obsahovat zatím pouze jednu šablonu, šablonu default.php. Její zdrojový kód:

```
<?php
    require_once FW_PATH.'app/views/inc/head.php';
?>

<div class="container">
    <h1>
        Vítejte v aplikaci správy uživatelů.
    </h1>

    <div class="well">
        Jak se dnes máte?
    </div>
</div>

<?php
    require_once FW_PATH.'app/views/inc/bottom.php';
```

Všimněte si přiložených souborů head.php a bootom.php ze složky **inc**.

11 Model

Od této chvíle uvidíme na při zadání adresy našeho serveru uvítací obrazovku:



Rovněž máme vytvořeno menu, jehož jednotlivé položky jsou uloženy v databázi. Přichází tak na řadu práce s třídami typu model, které pracují s daty. Ukážeme si, jak jednoduše takové menu vytvořit a jak užitečné pro tento účel budou obě bázové třídy našeho frameworku.

11.1 Menu

Menu webové stránky nebo aplikace je jistě jeho nepostradatelnou součástí a máme více možností, jak jej implementovat. Položky menu mohou být uloženy v databázi, v poli konfiguračního souboru atd. Nejčastěji jsou uloženy v databázi pro jejich jednoduchou správu. Stejný případ platí i pro nás. V MySQL v databázi test si vytvoříme tabulku menu a vložíme do ní data. Můžete použít tento SQL kód:

```
CREATE TABLE IF NOT EXISTS `menu` (
`id` int(11) NOT NULL,
  `controller` varchar(100) COLLATE utf8_czech_ci NOT NULL,
  `value` varchar(100) COLLATE utf8_czech_ci NOT NULL
) ENGINE=MyISAM AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci;
--
-- Vypisuji data pro tabulku `menu`
--
INSERT INTO `menu` (`id`, `controller`, `value`) VALUES
(1, 'page', 'Úvodní stránka'),
(2, 'user', 'Uživatelé'),
(3, 'contact', 'Kontakt');
--
-- Klíče pro tabulku `menu`
```

```
--

ALTER TABLE `menu`

 ADD PRIMARY KEY (`id`);

--

-- AUTO_INCREMENT pro tabulku `menu`

--

ALTER TABLE `menu`

MODIFY `id` int(11) NOT NULL AUTO_INCREMENT,AUTO_INCREMENT=4;
```

Naše menu tak obsahuje tyto položky:

- Úvodní stránka – název položky, page – název controlleru;
- Uživatelé, user;
- Kontakt, contact;

Začneme tedy vytvořením třídy modelu.

11.1.1 MENUMODEL

Ve složce models vytvoříme novou třídu MenuModel a vložíme do ní následující zdrojový kód:

```
require_once FW_PATH.'app/models/BaseModel.php';


class MenuModel extends BaseModel {

    /**
     * @param String $controller
     * @return Array
     */
    public function getMenuItems($controller)
    {
        $sql = "SELECT * FROM menu";
        $this->db->executeQuery($sql);


        while($row = $this->db->getRows()) {
            $items[$row['id']]['id'] = $row['id'];
```

```
            $items[$row['id']]['controller'] = $row['controller'];
            $items[$row['id']]['value'] = $row['value'];

            if ( $row['controller'] == $controller) {
                $items[$row['id']]['active'] = true;
            }
        }
        return (array) $items;
    }
}
```

Třída MenuModel bude potřebovat databázové připojení. Proto ji vytvoříme jako třídu dceřinou bázové třídě BaseModel. Všimněte si, že třída MenuModel nemá vlastní konstruktor, tedy bude vykonán pouze konstruktor rodičovské třídy. Nicméně té jako parametr musí být předána služba Db, což zajistíme při vytvoření objektu.

Třída MenuModel obsahuje jedinou metodu nazvanou getMenuItems(), která má parametr controller. Ten mu předáme buď přímo z URL nebo defaultní z Configuratoru. Použijeme jej pro označení aktivní položky menu.

Položky menu vrací metoda ve dvouúrovňovém poli **items**. Všimněte si prosím typování návratové hodnoty. Pokud potřebujeme vždy získat hodnotu items jako pole, použijeme tvar (array), čímž jej vždy získáme. Výpis tohoto pole při defaultním controlleru vypadá následovně:

```
Array (
      [1] => Array ( [id] => 1 [controller] => page [value] => Úvodní stránka
[active] => 1 )
      [2] => Array ( [id] => 2 [controller] => users [value] => Uživatelé )
      [3] => Array ( [id] => 3 [controller] => contact [value] => Kontakt )
)
```

11.1.2 Řízení menu v controlleru

Menu je **komponenta** webu. Jako taková bude součástí každé naší stránky a bylo by velmi nepraktické a především duplicitní umisťovat ji do všech našich controllerů. My ale máme k dispozici třídu BaseController, kde si na jednom místě tuto komponentu vytvoříme a máme jistotu, že bude k dispozici s každou třídou, která z třídy BaseController dědí.

Pojďme tedy upravit zdrojový kód třídy BaseController:

1. Před definici třídy dáme k dispozici obsah souboru MenuModel.php:

```
require_once FW_PATH.'app/models/MenuModel.php';
```

2. Vytvoříme si chráněnou vlastnost $menuItems, do které budeme ukládat načtené položky z modelu:

```
protected $menuItems = array();
```

3. Vytvoříme novou privátní metodu createComponentMenu(), která se postará o naplnění položek:

```
 private function createComponentMenu()

 {

     $controller = ($this->registr->Request->getUrl('controller') ? :
strtolower(Configurator::DEFAULTCONTROLLER));

     $menuModel = new MenuModel($this->registr->Db);

     $this->menuItems = $menuModel->getMenuItems($controller);

 }
```

4. Doplníme šablonu head.php o výpis položek menu:

```
        <ul class="nav navbar-nav">

        <?php

          foreach ($this->menuItems as $key => $item):

              $class = ($item['active'] ? ' class="active"' : "");

              $url = ($item['controller'] ==
strtolower(Configurator::DEFAULTCONTROLLER) ? Configurator::BASEURL :
Configurator::BASEURL."/".$item['controller']);

        ?>

              <li<?=$class;?>>

                  <a href="<?=$url;?>"><?=$item['value'];?></a>

              </li>

        <?php

          endforeach;

        ?>

        </ul>
```

11.1.3 KRATŠÍ ZPŮSOB ZÁPISU PŘÍKAZU ECHO

Jistě jste si všimli, že používám zápis ve tvaru:

```
<?=$url;?>
```

Jedná se o zkrácený zápis příkazu echo, což oceníme především v souborech šablon, protože příkaz echo jako takový by jistě neměl být součástí našich tříd. Výše uvedený zápis je tedy kratší alternativou k zápisu:

```
<?php echo $url;?>
```

12 Správa uživatelů

Na závěr se dostáváme k uvažované aplikaci. Pro její potřeby máme téměř vše připraveno. Při správě uživatelů uvažujeme dopředu tyto základní akce:

- Default – výpis uživatelů v tabulce včetně odkazů pro akci na vytvoření nového uživatele, editace a odstranění stávajících uživatelů.
- Add – vytvoření nového uživatele.
- Edit – úprava profilu uživatele.
- Delete – odstranění uživatele.

Vytvořme si tedy v databázi tabulku **users**. Bude obsahovat id, křestní jméno, příjmení a datum vytvoření uživatele. Zde máme k dispozici SQL kód:

```
CREATE TABLE IF NOT EXISTS `users` (
`id` int(10) unsigned NOT NULL,
  `firstname` varchar(30) COLLATE utf8_czech_ci NOT NULL,
  `lastname` varchar(40) COLLATE utf8_czech_ci NOT NULL,
  `created` datetime NOT NULL
) ENGINE=MyISAM AUTO_INCREMENT=5 DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci;
ALTER TABLE `users`
 ADD PRIMARY KEY (`id`);
ALTER TABLE `users`
MODIFY `id` int(10) unsigned NOT NULL AUTO_INCREMENT,AUTO_INCREMENT=1;
```

12.1 UserModel

Nejprve se zaměříme na vytvoření třídy modelu. Ta nám jak je jejím úkolem bude poskytovat přístup k údajům, včetně jejich modifikace v databázi. Třída UserModel bude opět dceřinou třídou bázové třídy a bude obsahovat několik základních metod, které si postupně představíme. Ve složce models vytvoříme novou třídu UserModel:

```
require_once FW_PATH.'app/models/BaseModel.php';
class UserModel extends BaseModel {

    /**
     * @return Array
     */
    public function getUsers()
    {
        $sql = "SELECT * "
                . "FROM users";

        $this->db->executeQuery($sql);
        $data = array();

        while($row = $this->db->getRows()) {
            $data[] = $row;
        }
        return $data;
    }

    /**
     * @param Int $id
     * @return Array
     */
    public function getUserById($id)
    {
```

```
    $sql = "SELECT * "
            . "FROM users "
            . "WHERE id = {$id}";

    $this->db->executeQuery($sql);
    return (array) $this->db->getRows();
}

/**
 * @param Array $senddata
 */
public function save($senddata)
{
    $data['firstname'] = $senddata['firstname'];
    $data['lastname'] = $senddata['lastname'];
    $data['created'] = date('Y-m-d H:i:s');

    $this->db->insertRecords('users', $data);
}

/**
 * @param Int $id
 */
public function delete($id)
{
    $cond = "id = ".$id;
    $this->db->deleteRecords('users', $cond, 1);
}

/**
 * @param Int $id
 * @param Array $senddata
```

```
     */
    public function update($id, $senddata)
    {
        $data['firstname'] = $senddata['firstname'];
        $data['lastname'] = $senddata['lastname'];
        $cond = "id = ".$id;
        $this->db->updateRecords('users', $data, $cond);
    }
}
```

První metoda je označena **getUsers**() a jak název napovídá, tato vrací v poli data všech uživatelů. Výsledkem je tedy dvouúrovňové pole, podobně jako v případě položek menu.

Druhá metoda vrací údaje jen o jednom uživateli, který je jednoznačně identifikován pomocí id. Výsledkem je tedy buď prázdné, nebo jednoúrovňové asociativní pole.

Na řadu přichází metoda **save**(), která jako parametr přebírá data odeslaná při vytvoření nového uživatele. Tato data jsou uložena do lokální proměnné a následně uložena do databáze.

Metoda **delete**() přebírá parametr id záznamu, který má být odstraněn. V lokální proměnné $cond je sestavena podmínka odstranění a následně volána metoda třídy Db – deleteRecords(). Tato metoda má 3 parametry, z nichž třetí určuje max. počet odstraněných záznamů. Pokud nechceme limit omezit, předáváme tento třetí parametr hodnotou 0.

Poslední metoda je metoda **update**(), pomocí které je upraven zvolený záznam. Je pro to potřeba jak id zvoleného záznamu, tak upravená data.

Třída modelu je tímto hotova.

12.2 UserController

Třída UserController bude sice mnohem obsáhlejší, než jednoduchý PageController, který má na starosti pouze jednu defaultní akci, ale bude fungovat na nám již známém principu, takže bychom s ní neměli mít problém. Tentokrát si její zdrojový kód pro lepší přehled výkladově rozdělíme. Ve složce controllers si založte novou třídu UserController. Začátek kódu je jasný:

```
require_once FW_PATH.'app/controllers/BaseController.php';
require_once FW_PATH.'app/models/UserModel.php';

class UserController extends BaseController {

    private $model;

    public function __construct(Registr $registr)
    {
        parent::__construct($registr);
        $this->model = new UserModel($registr->Db);
        $this->manageAction();
    }
```

Protože objekt modelu bude potřeba téměř pro všechny akce controlleru, vytvořili jsme si privátní vlastnost $model, do které nový objekt z třídy UserModel v konstruktoru uložíme.

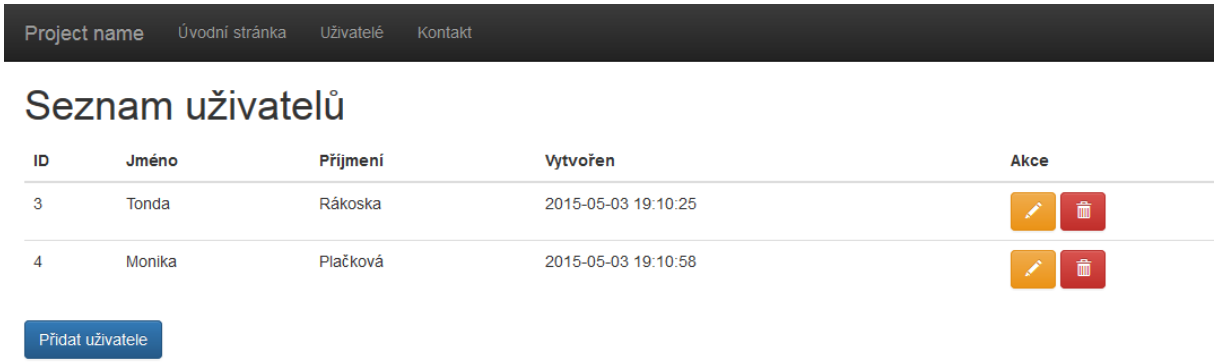
Dále si vytvoříme pomocnou metodu nazvanou **manageAction**(), která bude podle parametru akce v URL volat příslušné metody. Metoda manageAction() volána přímo v konstruktoru vypadá takto:

```
  private function manageAction()
    {
        $action = $this->registr->Request->getUrl('action');
        $id = $this->registr->Request->getUrl('id');

        if ($action == 'add') {
            $this->actionAdd();
        }
        elseif ($action == 'edit') {
            $this->actionEdit($id);
        }
        elseif ($action == 'delete') {
            $this->actionDelete($id);
```

```
        }
        else {
            $this->renderDefault();
        }
    }
```

Nejprve jsme si načetli z URL hodnoty parametrů **action** a **id**. Poté již pomocí jednoduchého větvení podmínky **if-else** voláme konkrétní **action metody** a v případě potřeby jim předáváme $id. Výchozí akce default žádnou action metodu nepotřebuje a proto rovnou voláme metodu typu render.

12.2.1 DEFAULTNÍ AKCE



Metoda **renderDefault**():

```
    public function renderDefault()
    {
        $this->setTitle('Správa uživatelů');
        $users = $this->model->getUsers();
        require_once FW_PATH.'app/views/User/default.php';
    }
```

V obsahu metody vidíme jednotlivé kroky:

- Nastavili jsme titulek stránky.
- Do lokální proměnné $users jsme prostřednictvím modelu načetli již vytvořené uživatele v dvourozměrném poli.
- Vkládáme šablonu default.php.

Výpis uživatelů v šabloně default.php: uvedeme si pouze tělo tabulky obsahující cyklus procházející pole uživatelů v lokální proměnné metody renderDefault() $users:

```
        <tbody>
            <?php
                foreach ($users as $user):
            ?>
            <tr>
                <td><?=$user['id'];?></td>
                <td><?=$user['firstname'];?></td>
                <td><?=$user['lastname'];?></td>
                <td><?=$user['created'];?></td>
                <td>
<a href="<?=Configurator::BASEURL;?>/user/edit/<?=$user['id'];?>" class="…">
    <span class="glyphicon glyphicon-pencil"></span>
</a>
<a href="<?=Configurator::BASEURL;?>/user/delete/<?=$user['id'];?>" class="…">
    <span class="glyphicon glyphicon-trash"></span>
</a>
                </td>
            </tr>
            <?php
                endforeach;
            ?>
        </tbody>
```

12.2.2 VYTVOŘENÍ NOVÉHO UŽIVATELE

Po kliknutí na tlačítko Přidat uživatele, resp. obsahuje-li URL akci **add**, volá metoda manageAction() metodu **actionAdd**(). Tato metoda musí odlišit dva stavy:

1. Uživatel aplikace klikl na stránce přehledu uživatelů na tlačítko Přidat uživatele a je potřeba vykreslit příslušný formulář. O to se postará metoda renderAdd().
2. Uživatel na stránce vytvoření uživatele zadal údaje a klikl na tlačítko **Přidat**. Metoda actionAdd() tedy musí zpracovat data z formuláře a předat je modelu k uložení. Následně je aplikace přesměrována na výchozí stránku seznamu uživatelů.

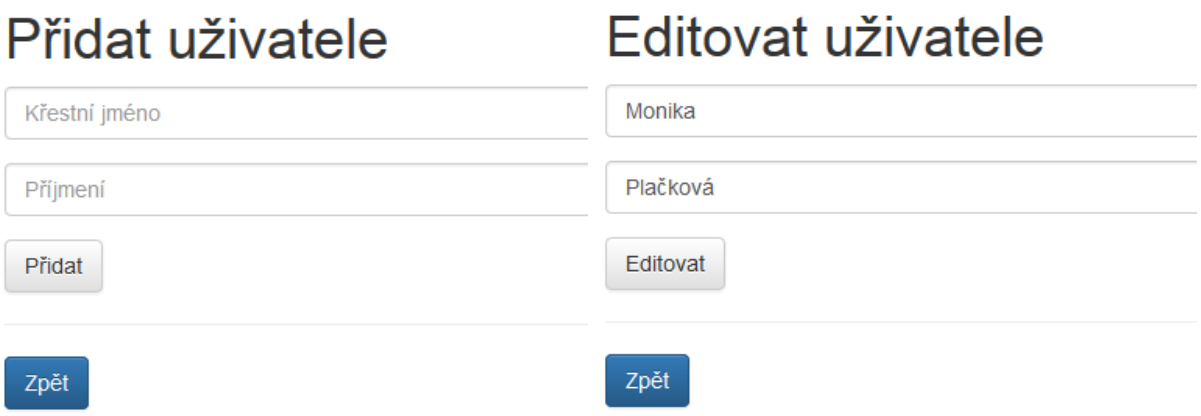
Metoda renderAdd() je velmi jednoduchá a nepotřebuje mého komentáře. Ukažme si ale jednu důležitou věc, kterou je lépe dodržovat:

Metody **přidání uživatele** a **editace uživatele** pracují nad shodným formulářem. Bylo by velmi nepraktické vytvářet v obou šablonách tento formulář zvláště, což by vedlo k duplicitě a zbytečným chybám. Proto jsme si vytvořili šablonu userform.php, kterou následně vkládáme do šablon **add.php** a **edit.php**. Zdrojový kód formuláře vypadá následovně:

```
<form method="post" action="">

  <div class="form-group">

      <input type="text" name="firstname" class="form-control" value="<?=$userData['firstname'];?>" placeholder="…">

  </div>

  <div class="form-group">

      <input type="text" name="lastname" required class="form-control" value="<?=$userData['lastname'];?>" placeholder="Příjmení">

  </div>

  <button type="submit" name="useraction" class="btn btn-default" value="1"><?=$actionRender;?></button>

</form>
```

Všimněte si především atributů **value**. Jejich hodnota je dána proměnnou $**userData**, která je při vytváření nového uživatele prázdná, kdežto při editaci obsahuje pole údajů daného uživatele, získaného z modelu na základě uživatelova id získaného z URL.

Popisek tlačítka **button** je zase dán hodnotou proměnné $**actionRender**, kterou si nastavujeme v metodách renderAdd() a renderEdit(). Máme tak jeden formulář pro obě akce.



Zdrojový kód metody actionAdd() musí, jak již víme odlišit zmíněné dva stavy. Pro tento účel si vystačíme s podmínkou, která bude monitorovat odeslání formuláře metodou POST. Tuto signalizaci zajistí služba Request:

```
private function actionAdd()
{
    if ($this->registr->Request->getPost('useraction')) {
        $this->model->save($this->registr->Request->getPost());
        $this->redirect('user');
    }
    $this->renderAdd();
}
```

V případě, že byl odeslán formulář (pole POST obsahuje parametr **useraction**, který je asociován s tlačítkem submit formuláře), volá model metodu save(), které předáváme všechna odeslaná data POST. Následně přesměrujeme aplikaci na defaultní výpis uživatelů. Volání metody renderAdd() tak nastane pouze tehdy, nebyl-li formulář odeslán.

12.2.3 EDITACE UŽIVATELE

Její průběh jsme již nastínili. V rámci editace musí URL obsahovat **id** editovaného uživatele. Toto id obsahuje výpis všech uživatelů, které zobrazujeme defaultně. Vytvářeli ve výpisu pro tlačítko editace tento odkaz:

```
<a href="<?=Configurator::BASEURL;?>/user/edit/<?=$user['id'];?>" class="…">
```

Metoda **actionEdit**():

```
/**
 * @param Int $id
 */
private function actionEdit($id)
{
    if (is_numeric($id) && $this->registr->Request->getPost('useraction')) {
        $this->model->update($id, $this->registr->Request->getPost());
        $this->redirect('user');
    }
    else {
        $this->renderEdit($id);
    }
```

```
}
```

V podmínce nyní provádíme dvojí kontrolu. Jednak odeslání formuláře, ale rovněž kontrolu předaného id, zdali je číselné, tedy zdali někdo nepřepisoval ručně URL. Id potřebujeme i pro metodu modelu **update**(), která potřebuje znát id záznamu, kterému budou upravována data.

```
/**
 * @param Int $id
 */
public function renderEdit($id)
{
   $this->setTitle('Editovat uživatele');
   $userData = $this->model->getUserById($id);
   $actionRender = "Editovat";
   require_once FW_PATH.'app/views/User/edit.php';
}
```

V renderovací metodě musíme získat data editovaného uživatele a naplnit pole userData. K tomuto účelu již máme v modelu vytvořenou metodu getUserById(), které předáváme id z URL.

12.2.4 ODSTRANĚNÍ UŽIVATELE

A jsme ve finále. Odstranění uživatele je hračkou. Při výpisu uživatelů jsme si podobně jako pro editaci vytvořili odkaz:

```
<a href="<?=Configurator::BASEURL;?>/user/delete/<?=$user['id'];?>" class="…">
```

Pro odstranění již nepotřebujeme renderovací metodu. Leda pokud byste chtěli zobrazovat stránku pro potvrzení odstranění, ale to už je ve Vaší kompetenci. Zde si vystačíme pouze z metodou actionDelete(), která provede odstranění a přesměruje zpět na výpis uživatelů:

```
/**
 * @param Int $id
 */
private function actionDelete($id)
{
    if (is_numeric($id)) {
        $this->model->delete($id);
    }
```

```
        $this->redirect('user');
    }
```

13 Shrnutí

A jsme u konce. Vytčené cíle se nám podařilo naplnit, je dále jen na Vás, jak se s dalším vývojem poperete. Framework nabízí téměř neomezenou paletu možností dalšího rozšíření. Od základního rozšíření front controlleru, přes další služby, užití šablonovacího systému, databázového rozhraní PDO, až třeba po automatizaci životního cyklu controlleru. Nebojte se využívání knihoven třetích stran, ať už se jedná o komuniktu vývojářů PHP, nebo dílčí části významných frameworků jako jsou Nette nebo Zend. Přeji hodně zdaru.

Použité zdroje

BÖHMER, Marian. *Návrhové vzory v PHP*. Brno: Computer Press, 2012. ISBN 978-80-251-3338-5.

GILMORE, W. Jason. *Velká kniha PHP5 a MySQL.* Brno: Zoner Press, 2011. ISBN 978-80-7413-163-9.

PEACOCK, Michael. *Programujeme vlastní sociální síť v PHP5*. Brno: Computer Press, 2012. ISBN 978-80-251-3626-3.

VRÁNA, Jakub. *1001 tipů a triků pro PHP.* Brno: Computer Press, 2010. ISBN 978-80-251-2940-1.

PHP Development site. [online]. Dostupné z: http://www.php.net.

W3Schools.com. [online]. Dostupné z: http://www.w3schools.com.

MySQL Dev. [online]. Dostupné z: http://dev.mysql.com/doc/.

Nette Framework. [online]. Dostupné z: http://nette.org/cs/