KAPITOLA 2

Plánování a vývoj základního frameworku

Nyní, když máme jasno v tom, co nás v této knize čeká a proč, můžeme začít s vývojem našeho sociálního webu. Abychom zajistili rychlý postup vývoje, investujeme v této kapitole nějakou část pečlivému naplánování a vývoji minimalistického frameworku, který se postará o řadu běžných úkonů. Bude se jednat o malý a jednoduchý framework, který neodvede naši pozornost od toho hlavního, a to vytvoření sociální sítě, a jehož smyslem je pomoci nám tohoto cíle dosáhnout.

V této kapitole se dozvíte:

- O běžných návrhových vzorech řešících časté problémy programátorů, včetně následujících vzorů:
  - MVC – architektura Model-View-Controller
  - Vzor Registry
  - Vzor Factory
  - Vzor Front Controller
- Jak efektivně strukturovat soubory v rámci vývojového frameworku.
- Jak vytvořit framework, včetně:
  - Autentizace uživatelů
  - Abstrakce přístupu k databázi
  - Správy šablon
- Jak vytvořit jednotný přístupový bod k webu.

Návrh frameworku

Ještě než se střemhlav vrhneme do programování, je důležité věnovat nějaký čas patřičnému naplánování a návrhu frameworku.

Návrhové vzory

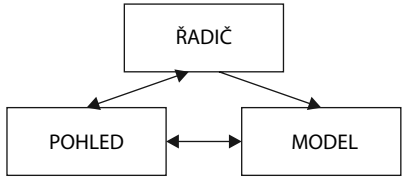
Návrhové vzory představují řešení běžných problémů programátorů a jejich správné použití může dopomoci ke korektnímu návrhu systému, na jehož základě se dá snadno stavět a ostatní s ním mohou jednoduše pracovat.

MVC (Model-View-Controller)

MVC je návrhový vzor oddělující uživatelské rozhraní od logiky aplikace. Uživatelské rozhraní (pohled) používá řadič pro přístup k logice a datům aplikace (model).

Zamysleme se, jak to bude vypadat v případě našeho webu Dino Space. Přidá-li uživatel jiného uživatele jako svého přítele, uvidí pohled **Přidat uživatele**. Když stiskne patřičné tlačítko zajišťující přidání přítele, zpracuje řadič tento požadavek uživatele a předá ho modelu. Ten aktualizuje seznam přátel uživatele a v případě potřeby odešle potvrzení. Pohled se aktualizuje na základě instrukcí řadiče a informuje uživatele o výsledku požadavku.

Na následujícím obrázku můžete vidět komponenty návrhového vzoru MVC:



Naše použití vzoru MVC nebude jeho exaktní implementací. Bude z něj však vycházet. Na téma MVC a jeho konkrétní implementaci na webech a ve frameworcích se vedou vášnivé diskuse, stejně jako nad otázkou, je-li pro webové aplikace vůbec vhodný.

Model

Modelem budou v našem frameworku třídy PHP, které mají na starost ukládání, správu a zpracování dat. Přístup k datům uloženým v databázi bude zajišťovat samostatná vrstva, kterou bude model používat. Modely jsou úzce spojené

s databází a reprezentují v ní uložená data vhodnějším způsobem, se kterým se snáze pracuje a ke kterému se přistupuje lépe než k vlastní databázi.

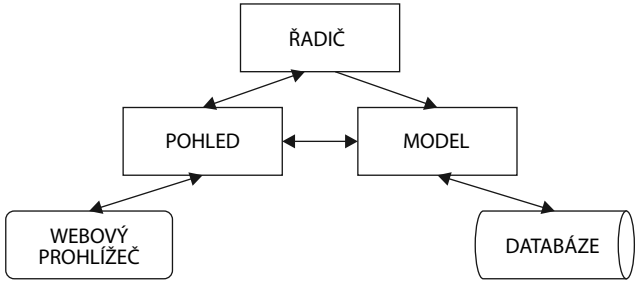
Pohled

Pohled bude v našem frameworku tvořit kombinace šablon (obsahujících kód HTML a zástupce pro dynamické vložení dat), obrázků, souborů CSS a JavaScriptu. Řadič zajistí dynamické spojení šablony s daty a zobrazení výstupu v prohlížeči uživatele.

Řadič

Řadiče bude tvořit množina tříd PHP, které se starají o zpracování požadavků uživatele, komunikují s modelem a vytváří pohledy. Technicky vzato je součástí řadiče i část kódu v JavaScriptu (obzvláště v kombinaci s AJAXem), protože pracuje na úrovni mezi pohledem a modelem. Jedná se tedy o rozšíření řadiče.

Protože používáme vzor MVC ve webovém prostředí, lze výše uvedenou architekturu znázornit detailněji, s ohledem na webový prohlížeč a databázi. Následující obrázek ukazuje, jak do architektury MVC zapadají prohlížeč a databáze (architektura MVC rozšířená o prohlížeč a databázi):



Front Controller

Vzor Front Controller tvoří jeden soubor, skrze který prochází všechny požadavky (v našem případě s využitím modulu `mod_rewrite` serveru Apache). V případě našeho frameworku se téměř jistě bude jednat o soubor `index.php`. Tento soubor zpracuje požadavky uživatelů a předá je patřičným řadičům.

Díky použití jednoho hlavního (front) řadiče může jádro obsahovat soubory, nastavení a další nezbytnosti, takže bez ohledu na požadavek uživatele víme, že jsou všechny tyto prostředky na svém místě.

Pokud bychom pro vyřízení požadavků uživatele používali samostatné soubory, například `friends.php` pro akce spojené s přáteli, museli bychom tyto standard-

ní funkce a nastavení buď zkopírovat, anebo je připojit ve specifickém souboru, který je obsahuje. To se může ukázat jako nešťastné rozhodnutí, bude-li třeba provést refactoring kódu anebo soubor odstranit či přejmenovat (bylo by nutné aktualizovat veškeré odkazy na něj).

Registry

Ve většině webových frameworků existuje několik základních objektů, případně objektů obsahujících základní funkce, ke kterým musí mít přístup všechny ostatní části aplikace. Vzor Registry umožňuje uložení všech těchto klíčových objektů do jediného centrálního objektu, odkud je možné k nim přistupovat.

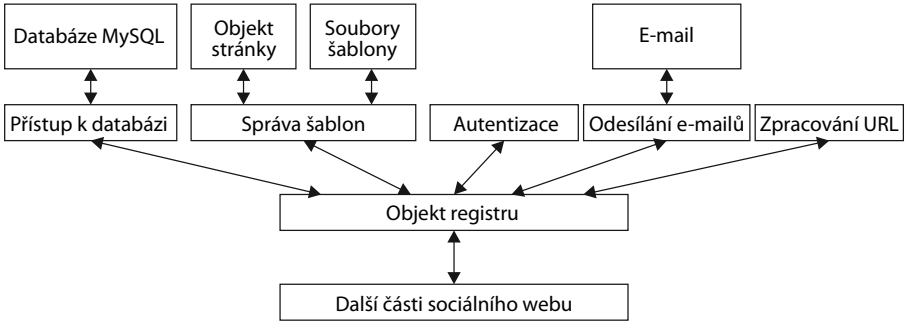
TIP

Vzor Registry také zjednodušuje vazby a závislosti, protože namísto několika globálních objektů (využívajících vzor Singleton, který se často považuje za nevhodný), které by se musely předat všem požadovaným modelům a řadičům, stačí předat jediný objekt registru obsahující všechny požadované objekty, stejně jako pole a proměnné s nastavením systému.

V rámci našeho sociálního webu existuje řada úkolů, které budeme často provádět, jako jsou například tyto:

- Ověření, jestli je uživatel přihlášený.
- Získání informací o přihlášeném uživateli.
- Dotazování databáze a provádění dalších funkcí spojených s databází.
- Odesílání potvrzení e-mailem, například když uživatel přidá jiného uživatele jako přítele.
- Odesílání dat pohledům, které je zobrazí v prohlížeči uživatele.
- Zpracování adresy URL, ke které uživatel přistupuje a na jejímž základě se rozhodne, jaká akce se má provést, který řadič použít a jaká metoda zavolat.

Tyto funkce se dočkají abstrakce do svých vlastních objektů, centrálně uložených v registru. Ostatní kód tvořící web může k objektům i daným funkcím přistupovat přímo z registru. Architekturu registru ilustruje následující obrázek:



Objekt Factory v registru

Dalším návrhovým vzorem, který využijeme, je vzor Factory. Abychom nemuseli vytvářet všechny objekty uložené v registru a vkládat je do něj, jednoduše sdělíme registru názvy objektů, které se mají vytvořit. Registr zajistí připojení patřičných tříd a vytvoření jejich nových instancí. Vytvořené objekty registr následně uloží do svého interního pole objektů. Návrhový vzor se jmenuje Factory (továrna), protože jeho objekt (v našem případě registr) vytváří jiné objekty.

Poznámka k návrhovému vzoru Singleton

Za zmínku bezesporu stojí také návrhový vzor Singleton. Tento vzor v podstatě obnáší vytvoření statického objektu, který v rámci celé aplikace vždy existuje nejvýše v jedné instanci. Statická povaha vzoru Singleton má za následek, že je možné ho volat kdekoli v kódu.

Použití návrhového vzoru Singleton k tomuto účelu by nebylo nejvhodnější, protože by ostatní objekty musely znát detaily objektu Singletonu. Jak jsme si řekli už dříve, měl by se objekt registru předávat přímo objektům, konkrétně jejich konstruktorům, čímž se eliminuje nutnost globálně přístupného objektu.

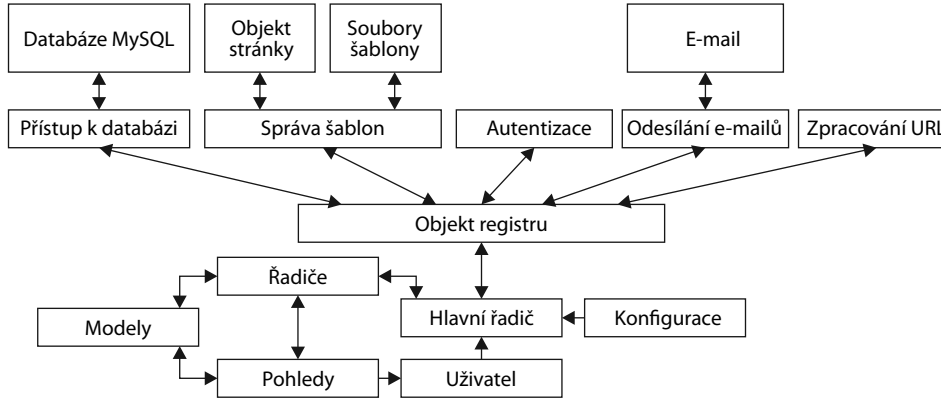
Přestože by se hodilo realizovat registr formou Singletonu, protože vždy chceme pouze jednu instanci tohoto objektu, není třeba se tím v PHP 5 zabývat, protože se zde objekty standardně předávají referencí. To znamená, že se metodě předá reference na objekt namísto kopírování objektu (jak tomu bylo v PHP 4). Další instance registru by se vytvořila pouze explicitním naklonováním nebo vytvořením nového objektu registru.

UPOZORNĚNÍ

Velmi se to podobá ukazatelům v jazyce C, kde ukazatel jednoduše ukazuje na místo v paměti obsazené objektem nebo proměnnou. Když se objekt nebo proměnná aktualizuje, přistupuje se k ní pomocí ukazatele. Nemusí se tedy aktualizovat její kopie či klony.

Registr a MVC

Kombinací vzoru MVC se vzory Registry a Front Controller jsme vytvořili framework, kde všechny požadavky prochází jedním centrálním bodem, který zajistí vytvoření registru a nezbytných řadičů. Řadiče vytvoří odpovídající modely a v některých případech předají, před vytvořením šablon a vygenerováním pohledů, řízení dalším řadičům. Následující diagram ukazuje všechny tyto komponenty a jejich vzájemné propojení:



Struktura adresářů

Další velmi důležitou součástí procesu plánování systému je použitá adresářová struktura, která pomůže zajistit správnou organizaci našich souborů. Když pak budeme nějaký soubor chtít najít nebo upravit, víme přesně, kam se podívat.

Vzhledem k použití vzorů MVC a Registry se nabízí zjevný způsob organizace souborů rozdělením na modely, pohledy, řadiče anebo soubory spojené s registrem. Začneme tedy u těchto adresářů:

- Controllers
- Models
- Registry
- Views

Do adresáře `Views` uložíme soubory šablon, obrázky, soubory CSS a soubory s kódem v JavaScriptu. Pokud budou moci uživatelé přepínat mezi různými grafickými návrhy, je žádoucí, aby byly všechny soubory konkrétního návrhu uložené v jednom podadresáři. V konkrétním pohledu navíc můžeme použít kromě kódu v JavaScriptu také knihovny JavaScriptu, které je také vhodné oddělit. Podtrženo sečteno to celé může vypadat nějak takto:

- Controllers
- Models
- Registry
- Views
  - MainView
  - CSS
  - Images
  - JavaScript
  - Templates

Soubory nahrané na server se nejspíše budou dělit do dvou kategorií – soubory, které jsme na server nahráli my jako administrátoři (prostředky), a soubory, které nahráli uživatelé (uploady). Soubory nahrané uživateli mohou využívat různé části sociální sítě, a proto je vhodné je dále rozdělit:

- Controllers
- Models
- Registry
- Resources
  - Images
    - Small
    - Large
    - Original
  - Files
- Uploads
  - ProfilePics
    - Small
    - Large
  - Photos
    - Small
    - Large
  - Files

- Views
  - MainView
  - CSS
  - Images
  - JavaScript
  - Templates

Vytvoření frameworku

Doporučené postupy při vytváření frameworku naší sociální sítě už známe, je tedy načase začít s jeho tvorbou.

Registr

Začneme s registrem, který představuje velmi důležitou část našeho frameworku. Registr tvoří samotný objekt registru a objekty, které jsou v registru uložené.

Objekt registru

Samotný objekt registru je poměrně jednoduchý. Obsahuje dvě pole, jedno pro uložení nastavení a dat a druhé pro uložení objektů centrálně přístupných z registru.

```
<?php
/**
 * Sociální síť v PHP
 * @author Michael Peacock
 * Třída Registry
 */

class Registry {

/**
 * Pole objektů
 */
private $objects;

/**
 * Pole nastavení
 */
private $settings;

public function __construct() {
}
```

Pro každé z těchto dvou polí jsou zapotřebí dvě metody – jedna pro ukládání dat resp. objektů do odpovídajícího pole a druhá pro jejich získávání. Vzhledem k tomu, že pro ukládání objektů použijeme vzor Factory, bude se tento kód lišit od toho pro ukládání nastavení.

```
/**
 * Vytvoří nový objekt a uloží ho do registru
 * @param String $object prefix objektu
 * @param String $key klíč, pod kterým bude objekt přístupný
 * @return void
 */
public function createAndStoreObject( $object, $key )
{
  require_once( $object . '.class.php' );
```

Jak jsme si řekli už dříve, většina objektů vyžaduje přístup k objektu registru, včetně objektů uložených v registru. Abychom tuto podmínku splnili, předáme objekt registru jako parametr konstruktoru objektů. Objektu tak předáme *referenci* na danou instanci registru (viz výše uvedené poznámky ke vzoru Singleton).

```
  this->objects[ $key ] = new $object( $this );
}
```

Když ukládáme nastavení, stačí jednoduše data vzít a uložit přímo do pole.

```
/**
 * Uloží nastavení
 * @param String $setting data
 * @param String $key klíč v poli nastavení
 * @return void
 */
public function storeSetting( $setting, $key )
{
  $this->settings[ $key ] = $setting;
}
```

Načítání dat i objektů z registru probíhá stejným způsobem, což dokazují metody `getSetting` a `getObject`, které tvoří stejný kód a liší se pouze prací s odpovídajícím polem.

```
/**
 * Získá nastavení z registru
 * @param String $key klíč v poli nastavení
 * @return String nastavení
 */
public function getSetting( $key )
{
```

```
    return $this->settings[ $key ];
}

/**
 * Získá objekt z registru
 * @param String $key klíč v poli objektů
 * @return Object
 */
public function getObject( $key )
{
    return $this->objects[ $key ];
}
}

?>
```

Objekty v registru

Na samotném objektu registru není nic složitého. Jeho smyslem je uložení dat a objektů. Jsou to v něm uložené objekty, které jsou tím složitým. Objekty uložené v registru budou zajišťovat následující funkce:

- Přístup k databázi
- Autentizace uživatelů
- Správa šablon
- Odesílání e-mailů
- Zpracování adres URL

Databáze

Naše třída zprostředkovávající přístup k databázi (`registry/mysqldb.class.php`) musí zajistit základní úroveň abstrakce přístupu k databázi. Díky ní je možné zjednodušit základní úkony jako je vkládání záznamů do databáze, aktualizace existujících záznamů a, je-li to zapotřebí, také vytváření a editace tabulek databáze.

Třída musí umět:

- Připojit se alespoň k jedné databázi.
- Spravovat spojení s více databázemi, je-li s nimi současně navázáno spojení.
- Provádět dotazy.
- Vracet výsledky provedených dotazů.
- Vracet informace o provedených dotazech, jako je například identifikátor záznamu naposledy přidaného do databáze.

- Ukládat výsledky dotazů do mezipaměti (hlavním cílem je integrace výsledků dotazů do pohledu tím, že se uloží do mezipaměti a následně asociují s určitou částí šablony).

Mnohé z metod této třídy budou jednoduše volat po stávající funkci databáze MySQL s několika doplňky navíc a dokonalejší správou chyb.

Spojení s databází a správa spojení

Aby bylo možné připojit se k více databázím, je třeba udržovat záznamy o jednotlivých spojeních. Toho lze docílit uložením každého navázaného spojení do pole a udržováním informace o tom, která z položek pole představuje aktivní spojení. Provede-li se nějaký dotaz, provede se s použitím právě aktivního spojení.

```
<?php
/**
 * Třída pro přístup k databázi: základní abstrakce
 *
 * @author Michael Peacock
 * @version 1.0
 */
class Mysqldb {

  /**
   * Umožňuje více spojení s databází
   * každé spojení se uloží jako prvek pole, aktivní spojení
   * identifikuje samostatná proměnná (viz níže)
   */
  private $connections = array();

  /**
   * Specifikuje spojení, které se má použít
   * voláním setActiveConnection($id) je možné aktivní spojení změnit
   */
  private $activeConnection = 0;

  /**
   * Provedené dotazy, jejichž výsledky se uložily do mezipaměti pro
   * pozdější použití, primárně pro potřeby šablonového systému
   */
  private $queryCache = array();

  /**
   * Připravená data uložená do mezipaměti pro pozdější
   * použití, primárně pro potřeby šablonového systému
   */
  private $dataCache = array();
```

```
  /**
   * Počet provedených dotazů
   */
  private $queryCounter = 0;

  /**
   * Výsledek posledního provedeného dotazu
   */
  private $last;

  /**
   * Objekt registru
   */
  private $registry;

  /**
   * Konstruktor databázového objektu
   */
  public function __construct( Registry $registry )
  {
    $this->registry = $registry;
  }
```

Připojení k databázi vyžaduje adresu jejího hostitele, uživatelské jméno, heslo a samozřejmě také název databáze, ke které se chceme připojit. Výsledné spojení se uloží do pole spojení a vrátí se identifikátor spojení (klíč v poli spojení).

```
/**
 * Vytvoří nové spojení s databází
 * @param String adresa hostitele
 * @param String uživatelské jméno
 * @param String heslo
 * @param String požadovaná databáze
 * @return int the id of the new connection
 */
public function newConnection( $host, $user, $password, $database )
{
  $this->connections[] = new mysqli( $host, $user, $password, $database );
  $connection_id = count( $this->connections )-1;
  if( mysqli_connect_errno() )
  {
    trigger_error('Chyba při pokusu o připojení k databázi. '.
      $this->connections[$connection_id]->error, E_USER_ERROR);
  }

  return $connection_id;
}
```

Je-li třeba přepnout na jiné spojení, například za účelem získání dat z externího zdroje nebo autentizace u jiného systému, musíme databázovému objektu říct, aby použil jiné spojení. K tomu slouží metoda `setActiveConnection`.

```
/**
 * Změní aktivní spojení pro následující dotazy
 * @param int identifikátor nového spojení
 * @return void
 */
public function setActiveConnection( int $new )
{
  $this->activeConnection = $new;
}
```

Provádění dotazů

Poté, co se provede dotaz, bude často žádoucí znát jeho výsledek. Z tohoto důvodu se výsledek posledního dotazu ukládá do vlastnosti `last` třídy, ke které pak mohou přistupovat ostatní metody třídy.

```
 /**
  * Provede dotaz
  * @param String dotaz
  * @return void
  */
 public function executeQuery( $queryStr )
 {
  if( !$result = $this->connections[$this->activeConnection]
    ->query( $queryStr ) )
  {
    trigger_error('Chyba při provádění dotazu: ' . $queryStr .' - '.
      $this->connections[$this->activeConnection]->error, E_USER_ERROR);
  }
  else
  {
    $this->last = $result;
  }
}
```

Záznamy, které vrátil dotaz, získáme voláním metody `fetch_array` objektu výsledku dotazu uloženého ve vlastnosti `last`.

```
/**
 * Získá záznamy vrácené posledním provedeným dotazem
 * @return array
 */
public function getRows()
```

```
{
  return $this->last->fetch_array(MYSQLI_ASSOC);
}
```

Zjednodušení běžných dotazů

Běžné dotazy jako je `INSERT`, `UPDATE` a `DELETE` se často opakují. Snadno se však dají abstrahovat a přidat do naší třídy pro práci s databází. Úplně pokaždé je sice nebude možné použít, ve většině případů by nám to však mělo zjednodušit život. V rámci této třídy můžeme abstrahovat i dotazy výběru dat. Ty jsou však podstatně komplikovanější a mnohem častěji budou obsahovat komplexní logiku jako jsou například poddotazy, spojení a aliasy. Tuto logiku by bylo nutné zahrnout do kódu třídy.

Ke smazání záznamu z databáze stačí pouze název tabulky, podmínka a omezení. V některých případech nemusí být omezení (klauzule `LIMIT`) zapotřebí, nastaví se tedy pouze, je-li odpovídající parametr metody neprázdný.

```
/**
 * Odstraní záznamy z databáze
 * @param String název tabulky, ze které se mají záznamy odstranit
 * @param String podmínka, kterou musí odstraňované záznamy splnit
 * @param int počet odstraňovaných záznamů
 * @return void
 */
public function deleteRecords( $table, $condition, $limit )
{
  $limit = ( $limit == '' ) ? '' : ' LIMIT ' . $limit;
  $delete = "DELETE FROM {$table} WHERE {$condition} {$limit}";
  $this->executeQuery( $delete );
}
```

Aktualizace a vkládání záznamů jsou operace, které osobně považuji za nejvíce krkolomné. Snadno je však lze abstrahovat pomocí názvu tabulky, asociativního pole názvů a hodnot sloupců a v případě aktualizace podmínky.

```
/**
 * Aktualizuje záznamy v databázi
 * @param String název tabulky
 * @param array asociativní pole změn
 * @param String podmínka
 * @return bool
 */
public function updateRecords( $table, $changes, $condition )
{
  $update = "UPDATE " . $table . " SET ";
```

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

    // odstranění nadbytečného znaku "," na konci
    $update = substr($update, 0, -1);
    if( $condition != '' )
    {
      $update .= "WHERE " . $condition;
    }
    $this->executeQuery( $update );

    return true;
  }

  /**
   * Vloží záznamy do databáze
   * @param String název tabulky
   * @param array asociativní pole vkládaných dat
   * @return bool
   */
  public function insertRecords( $table, $data )
  {
    // inicializace proměnných pro názvy a hodnoty sloupců
    $fields  = "";
    $values = "";

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

    // odstranění nadbytečného znaku "," na konci
    $fields = substr($fields, 0, -1);
    // odstranění nadbytečného znaku "," na konci
    $values = substr($values, 0, -1);

    $insert = "INSERT INTO $table ({$fields}) VALUES({$values})";
    //echo $insert;
    $this->executeQuery( $insert );
    return true;
  }
```