Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Zend Framework a jeho nadstavba

Diplomová práce

Autor: Filip Šimerda

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové srpen 2016

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 27. srpna 2016 Filip Šimerda

Poděkování

Chtěl bych na tomto místě poděkovat doc. Ing. Filipu Malému, Ph.D. za ochotu a odborné rady při vedení mé diplomové práce. Rovněž chci poděkovat mé rodině za podporu během studia.

Anotace

Diplomová práce se zabývá problematikou PHP frameworku Zend a dlouhodobým vývojem aplikací a informačních systému postavených na tomto frameworku, z čehož se postupem času vytříbila oddělitelná nadstavba rozšiřující možnosti tohoto již dnes zastaralého frameworku.

Práce se nejprve věnuje přehledu hlavních komponent Zend Frameworku napsaném v jazyce PHP. Na tomto frameworku je dlouhodobě vyvíjen informační systém a proto byla nutnost nad tímto frameworkem vytvořit nadstavbu tzv. modelovou vrstvu, která nejen usnadňuje jeho použití, ale rozvíjí jeho možnosti. Popisem jednotlivých částí modelové vrstvy, které rozšiřují komponenty Zend Frameworku se věnuje praktická část práce. Důležitou částí je rovněž přehled návrhových vzorů aplikovaných jazykem PHP, které byly při vývoji modelové vrstvy respektovány. Obsahem práce je také přehled ostatních nejrozšířenějších PHP frameworků, které jsou v současnosti na trhu dostupné. Nedílnou součástí je shrnutí výsledků a zkušeností dosažených při vývoji aplikací postavených na Zend frameworku a poznatků při postupném vytváření nadstavby nad tímto frameworkem.

Annotation

Title: Zend Framework and extension

Diploma Thesis deals with problematic of PHP Zend Framework and long-term development of software applications and information systems built on this framework, which was gradually crystallized detachable extension expands the possibilities of this already obsolete framework.

The work firstly focuses to an interview of main components of the Zend Framework written in PHP script language. In this framework is developed long-term real estate information system and therefore it was a necessity to create superstructure of this framework called Model Layer, which not only facilitates its use, but is developing its possibilities. The practical part is devoted to the elements of the model layer expanding the components of the Zend Framework. The important part is also a list of design patterns applied to PHP, consistently used for developing model layer. This work contains an overview of the other most widely used PHP frameworks that are currently available on the market. An integral part is a summary of the results and the experience gained in the development of applications built on Zend Framework and knowledge in the gradual creation of this framework superstructure.

Obsah

1 Úvod

V dnešní době jsou frameworky nedílnou součástí vývoje webových aplikací. Použití frameworku významně zkracuje čas při vytváření aplikací, jelikož obsahují znovupoužitelné komponenty a standardizované rozhraní, ve kterém se aplikace tvoří. Frameworky rovněž dodržují vzor MVC, který odděluje prezentační vrstvu od logické vrstvy, což zajišťuje vývojářům a kodérům přehlednost a snadné oddělení práce při společném vývoji webové aplikace. Díky frameworkům je možné soustředit se přímo na vytváření samotné aplikace a části a vrstvy jako jsou například zabezpečení, přihlašování uživatelů či práce s databází nechat na samotném frameworku, případně jednoduše nakonfigurovat nebo upravit s pomocí dokumentace. Z důvodu velkého rozšíření scriptovacího jazyka PHP a jeho oblíbenosti mezi webovými vývojáři vzniklo i mnoho frameworků napsaných v tomto jazyce. Průkopníkem mezi PHP frameworky byla firma Zend Technologies se svým Zend Frameworkem vyvinutým roku 2006. Aktuálně jsou na trhu podle různých průzkumů nejpopulárnější frameworky Laravel a Symfony.

Ačkoliv jsou frameworky vyvíjeny rozsáhlou komunitou uživatelů případně přímo pod záštitou vývojářské společnosti, některé jejich komponenty či vrstvy nemusejí vývojářům vyhovovat. Ať už je to dané jiným způsobem myšlení a záměrem při vývoji, nedostatečnou propracovaností nebo zastaráváním při dlouhodobé neaktualizaci. Těmito problémy se vývojář začne zabývat ve chvíli kdy již delší dobu pracuje s frameworkem, či je zvyklý na nějaké principy vývoje a framework tyto principy nerespektuje. V takovém případě začne některé komponenty či přímo vrstvy frameworku upravovat, aby si pracovní prostředí přizpůsobil pro svoje účely a vývoj zjednodušil. Často se totiž vývoj některých modulů stává rutinní činností, při které je vhodné co nejvíce společné funkcionality, kterou lze zobecnit, vytěsňovat do nadřazených tříd. Či existuje komponenta, která je příliš komplexní nebo naopak neobsahuje určitou funkcionalitu, a je potřeba takovou komponentu přizpůsobit nejen pro momentální ale i opakované použití. Díky těmto úpravám se postupně nad samotným frameworkem začne vyvíjet vrstva, která může významně měnit a rozšiřovat jeho možnosti.

Ať už je nadstavbová vrstva šířena mezi komunitu uživatelů původního frameworku a některé její části mohou být zapracovány do oficiální distribuce nebo slouží jen pro

ulehčení vývoje v úzkém okruhu vývojářů, je vhodné řídit se obecnými zásadami vývoje aplikací. Mezi tyto zásady patří principy objektově orientovaného vývoje a při řešení programátorských záludností využít návrhové vzory. Ty umožňují standardním způsobem vyřešit spoustu problémů, které plynou nejen ze špatného návrhu a struktury aplikace ale rovněž samotný běh aplikace zefektivnit. Programový kód aplikací vyvinutých s využitím návrhových vzorů je pak rovněž mnohem lépe čitelnější a jejich použití významně urychluje vývoj aplikací nejen samotnému programátorovi ale i v rámci týmu.

2 Zend Framework

Jedním z prvních frameworků podporující jazyk PHP, který se objevil na trhu, byl Zend Framework 1. Publikován byl v roce 2006 a rychle se rozšířil mezi vývojáře. Zend Framework je open source objektově orientovaný framework napsaný v jazyce PHP 5 [1].

Zend Framework je komplexním nástrojem pro tvorbu webových aplikací, který poskytuje vývojářům plnou podporu jak pro vývoj jednoduchých webových prezentací, tak tvorbu rozsáhlých informačních systémů.

2.1 Komponenty Zend Frameworku

Framework nabízí celou řadu komponent pokrývajících nejrůznější oblasti webové aplikace a usnadňující jejich vývoj:

- Zend_DB – práce s databázemi
- Zend_Controller – podpora controllerů a definování akcí
- Zend_Auth – autentifikace uživatelů do aplikace
- Zend_Acl – řízení uživatelských práv
- Zend_View – podpora pohledů a helperů
- Zend_Layout – definování layoutu webové aplikace
- Zend_Form – vytváření formulářů definovaných jako objekty
- Zend_Cache – použití nejrůznějších forem kešování
- Zend_PDF – vytváření PDF souborů

2.1.1 Komponenta Zend_Controller

Jednou z nejdůležitějších komponent v Zend frameworku je komponenta Zend_Controller. Stará se o ovládání modelu a slouží jako prostředník mezi pohledem a modelem. Komponenta v sobě zaštiťuje sadu dalších komponent, které mají široký záběr a umožňují snadnou práci při vývoji.

2.1.1.1 Zend_Controller_Action

Komponenta Zend_Controller_Action je základem, ze kterého v Zend Frameworku vychází každý controller. Tedy musí být potomkem třídy Zend_Controller_Action a jeho jméno musí končit slovem „Controller“ např.: IndexController. Jednotlivé moduly mají většinou svoje vlastní controllery. Akce, které jsou na modulech prováděny jsou pak rozděleny do metod na controlleru a jejich jméno končí klíčovým slovem „Action“ např.: editAction(). Standardně se pak k těmto akcím dá přistupovat přes URL adresy, kde jako první část je název controlleru a druhá část je název akce např.: /user/new zavolá metodu newAction() na controlleru UserController.

Action controller poskytuje řadu užitečných metod. Jednou z nich je metoda init() sloužící k vykonání příkazů při vytváření instance action controlleru. K tomuto účelu se využívá místo metody __construct(), která je již zabrána samotnou třídou Zend_Controller_Action, v jejímž konstruktoru je metoda init() volána. Pro vykonání kódu, který se bude provádět před a po proběhnutí akce se používají metody preDispatch() a postDispatch(). K odchytávání akcí, které jsme v controlleru nenadefinovali lze využít magickou metodu __call(), která je prováděna vždy, pokud není nalezena odpovídající metoda.

2.1.1.2 Zend_Controller_Request

Pomocí komponenty Zend_Controller_Request pracujeme s požadavkem jako s objektem. Standardně je tímto objektem instance třídy Zend_Controller_Request_Http. Pracujeme tedy s objektem HTTP požadavku. Na objektu je možné pomocí metod getQuery(), getPost(), getCookie(), getEnv() a getServer() získat data ze superglobálních polí $_GET, $_POST, $_COOKIE, $_ENV, $_SERVER. Metoda getParam() slouží k získaní hodnoty jednoho parametru a getParams() pak celého pole hodnot všech parametrů asociovaných podle názvu.

Z objektu požadavku lze rovněž získat název akce, controlleru a modulu metodami getActionName(), getControllerName() a getModuleName(). Na objektu je také možné získat další hodnoty HTTP požadavku jako jsou například údaje o prohlížeči či volaná adresa. Dalšími metodami je umožněno získat informace o typu požadavku. Tedy zda se například jedná o požadavek typu GET nebo POST metodami isGet() a isPost(), či zda se jedná například o XmlHttpRequest čili ajaxový požadavek metodou isXmlHttpRequest().

2.1.1.3 Zend_Controller_Response

Komponenta Zend_Controller_Response má na starost obalení obsahu odpovědi do objektu a jeho následné odeslání. V objektu existují dvě sady metod, které se starají o úpravu obsahu a úpravu hlaviček odesílaných do prohlížeče. Pro úpravu obsahu jednotlivých segmentů stránky jsou k dispozici metody setPrepend(), setBody() a setAppend() aj. Hlavičky odesílané do prohlížeče lze upravovat pomocí metod setHeader(). HTTP status kód odesílané odpovědi pak lze nastavovat metodou setHTTPResponseCode(). K přesměrování na jinou adresu nastavením hlavičky Location se využívá metoda setRedirect(), kde jako druhý parametr můžeme uvést HTTP status kód. O odeslání odpovědi se standardně stará front controller automaticky ke konci běhu scriptu, ale je možné se zavoláním metody sendResponse() o odeslání ručně postarat.

2.1.1.4 Zend_Controller_Action_Helper

V případě, že chceme v action controlleru použít na více místech stejný kus kódu, vytěsníme jej do speciální metody. Může se však stát, že stejný kód chceme použít i v dalších action controllerech, v tom případě je možné si vytvořit třídu ze které budou všechny actiony controllery vycházet a do té společnou metodu vložit. To ale s narůstajícím množstvím metod může začít způsobovat problémy. Řešením je použít action helper. O načítání action helperů se stará automaticky komponenta Zend_Controller_Action_Broker.

Action helper implementuje abstraktní třídu Zend_Controller_Action_Helper_Abstract a lze v něm implementovat metody init(), preDispatch() a postDispatch(). Také jsou přístupné objekty Request a Response stejně jako v action controlleru. Pro jednodušší použití je možné definovat metodu direct(), díky níž je možné action helper volat přímo s parametrem, který se předá právě metodě direct().

2.1.2 View Renderer

Pro snazší používání Zend View je možné využívat ViewRenderer. Jedná se o action helper, který je přístupný defaultně v action controlleru. ViewRenderer slouží k inicializaci pohledu a automaticky připojuje šablonu, do které se budou vypisovat data, takže není nutné v action controlleru pohled nastavovat. Pokud je potřeba změnit script šablony, lze jej definovat po zavolání metody setRenderer() případně přímo přes metodu direct().

2.1.3 Error Handler

Výjimky zachycené v průběhu zpracování programu je možné zachytit error handlerem a zpracovat. Ke zpracování zachycených chyb potřebujeme ještě ErrorController, ve kterém bude implementována metoda errorAction(). V případě zachycení výjimky se pak zavolá error akce, kde se nastaví příslušné HTTP stavové kódy a do pohledu se předá seznam chyb, které je pak v šabloně možné vypsat.

2.1.4 Dispatcher

Úlohou dispatcheru je na základě parametru v request objektu získaných ze směrovače vytvořit instanci správného action controlleru. Poté dispatcher volá správnou metodu akce. Tato volání probíhají v cyklu, dokud nejsou obslouženy všechny požadavky. Dispatcher pracuje nativně bez nutných zásahů ze strany vývojáře. Část jeho metod je pak možno volat přes front controller.

2.1.5 Zend_Acl

Pomocí komponenty Zend_Acl je možné řídit, kam mohou uživatelé v systému přistupovat a co tam mohou dělat. Provádí se to pomocí tzv. přístupových seznamů (access control list – ACL). Tyto přístupové seznamy definují role, zdroje, privilegia a pravidla.

2.1.5.1 Zend_Acl_Role

Role se vytvářejí pomocí komponenty Zend_Acl_Role. Roli má nastavenou uživatel systému podle jeho uživatelského účtu. Každý uživatel má k uživatelskému účtu napárovanou roli. Pokud uživatel uživatelský účet nemá, může dostat výchozí roli např. host (guest). Role mohou od sebe dědit privilegia.

2.1.5.2 Zend_Acl_Resource

Zdroje se definují přes komponentu Zend_Acl_Resource. Zdroje můžeme chápat jako větší logické celky v informačním systému tzv. moduly. Zdroje mohou být například správa uživatelů, evidence záznamů atd.

2.1.5.3 Privilegium

Privilegia neboli oprávnění popisují, co mohou s daným zdrojem které role dělat. Takovým oprávněními mohou být například oprávnění vypsat, změnit smazat atd.

2.1.5.4 Pravidla

Pravidla určují, zda je možné provést se zdrojem nějaký úkon. Tedy, zda je to povoleno, či ne. Určuje se to voláním metod allow() nebo deny() na objektu přístupového seznamu.

Nyní již známe všechny prvky, pomocí nichž se tvoří přístupové seznamy a lze tedy pro uživatelské role definovat, co mohou na kterém místě v informačním systému dělat a co je jim odepřeno. Přístupový seznam se inicializuje vytvořením instance třídy Zend_Acl. Nejprve je nutné definovat uživatelské role a zdroje v systému. Role se definují vytvořením instance třídy Zend_Acl_Role, kde je jako parametr konstruktoru jméno role. Do přístupového seznamu se pak přidávají pomocí metody addRole(). Prvním parametrem je objekt role a druhý nepovinný parametr je role, ze které se případně dědí oprávnění. Zdroje se pak definují vytvořením instance třídy Zend_Acl_Resource, kde je jako parametr konstruktoru jméno zdroje. Do přístupového seznamu se přidávají metodou addResource(). Jak už bylo popsáno, metody allow() a deny() určují, zda smí nebo nesmí uživatelské role ve zdroji provádět nějakou činnost. Volají se přímo na objektu přístupového seznamu a mají tři parametry. Prvním je název uživatelské role, druhým je název zdroje a třetím je oprávnění, které uživatelská role může nebo nesmí na zdroji provádět. Takto se dá pokrýt kompletní množina oprávnění v systému a řídit tak přístup uživatelských rolí do všech částí systému. Ověřování přístupů a oprávnění je pak možné pomocí metody isAllowed() na objektu přístupového seznamu. Parametry jsou jméno uživatelské role a jméno zdroje. Takto lze ověřit, zda má uživatel do jednotlivých zdrojů vůbec přístup. Třetím parametrem je jméno oprávnění. Tím se upřesňuje, zda může uživatelská role na zdroji provádět nějakou konkrétní činnost. Metoda vzhledem k jejímu pojmenování vrací buď true v případě, že uživatelská role oprávnění má, nebo false v případě, že oprávnění nemá.

2.1.5.5 Členění práv podle controllerů a akcí

Jedním ze způsobů jak pojmout systém vytváření oprávnění a jeho strukturu v aplikaci je, že se jako zdroje berou jednotlivé controllery a jako oprávnění akce na těchto controllerech. Přístupové seznamy pak jen popisují, která uživatelská role má přístup do té které akce na controllerech, či zda má přístup vůbec do samotného controlleru. Tento systém členění práv se sám nabízí, ale má své limity ve chvíli, kdy uživatelským rolím

chceme v rámci jedné akce povolit různé množiny úkonů a viditelností. Pak je potřeba systém členění práv rozšířit.

2.1.5.6 Definice oprávnění pomocí konfiguračního souboru

Definování oprávnění uživatelů je možné provést různými způsoby. Jedním z nich je i konfigurační soubor. Jedná se o ini soubor, ve kterém jsou tři oddělené bloky s definicí rolí, zdrojů a oprávnění. Pomocí komponenty Zend_Config_Ini jsou pak data načtena a zpracována podle struktury konfiguračního souboru. Díky komponentě Zend_Config_Ini je práce s takovýmto konfiguračním souborem jednoduchá, jelikož se se souborem pracuje jako s objektem. Jednotlivé definice rolí, zdrojů a oprávnění je pak možné procházet soustavou foreach cyklů.

2.1.5.7 Cachování

Načítání práv po každém sestavení aplikace ať už z konfiguračního souboru či jiného datového zdroje by bylo neefektivní a při velkém zatížení systému bylo mohlo způsobovat problémy. Proto se pro načítání definice práv používá cache. Zend framework nabízí pro práci s cachí komponentu Zend_Cache, která cachování velmi usnadňuje. Komponenta poskytuje pro načítání a ukládání metody load() a save(). Načítání a ukládání definice práv se zpravidla provádí v bootstrapu aplikace a zde se taktéž obstará cachování. Nejprve se metodou load() s parametrem jména cachovaného objektu data načtou a ve chvíli, kdy není cache k dispozici se objekt nově inicializuje, naplní daty například z databáze či konfiguračního souboru a uloží do cache. Cache má nastavenou platnost a lze ji manuálně invalidovat například ve chvíli, kdy jsou práva nějakým způsobem změněna. Tímto je zaručeno, že se při každém sestavení aplikace načítají definice oprávnění z cache a jsou validní.

2.1.6 Zend_Auth

Zend framework nabízí pro autentizaci uživatelů v systému komponentu Zend_Auth. Pro ověření o jakého přihlášeného uživatele jde lze použít více prostředků tzv. adaptérů. Nejčastěji jím bude databázový adaptér. Ověření o jakého se jedná uživatele se provádí pomocí metody authenticate() na objektu Zend_Auth. Ta dostává jako parametr přímo adaptér. Adaptéru jsou pomocí metod setIdentity() a setCredential() předány uživatelské

jméno a heslo a pomocí nich se vyhledají data uživatele, která určují jeho identitu.

2.1.7 Databázová komponenta Zend_Db

Komponenta Zend_Db poskytuje plnou podporu pro práci s většinou relačních databázových systémů. Obsahuje řadu dalších subkomponent, které se specializují na připojení k databázi, vytváření dotazů a abstrakci databázových tabulek.

2.1.7.1 Zend_Db_Adapter

Připojení k databázi je umožněno pomocí komponenty Zend_Db_Adapter [2]. Zend_Db respektive Zend_Db_Adapter podporuje několik druhů databází využívající PHP extenze [3]:

- IBM DB2 a Informix Dynamic Server (IDS) – pdo_ibm, ibm_db2 PHP extenze
- MariaDB – pdo_mysql, mysqli PHP extenze
- MySQL – pdo_mysql, mysqli PHP extenze
- Microsoft SQL Server – pdo_dblib PHP extenze
- Oracle – pdo_oci, oci8 PHP extenze
- PostgreSQL – pdo_pgsql PHP extenze
- SQLite – pdo_sqlite PHP extenze
- Firebird (Interbase) – php_interbase PHP extenze

Inicializace adaptéru probíhá pomocí zavolání statické metody factory() na třídě Zend_Db, kde je jako parametr uveden název adaptéru a pole parametrů konfigurace připojení. Adaptér je rovněž možné inicializovat pomocí konfiguračního souboru. Samozřejmé je použití více adaptérů zároveň pro různá spojení s databázemi, což je umožněno díky zásuvnému modulu Zend_Application_Resource_MultiDb. [2]

2.1.7.2 Vytváření dotazů

Zend framework umožňuje základní operace nad databází jako načítání, změna, vložení a mazání, které je možné provádět pomocí metod volaných na databázovém adaptéru. [2]

2.1.7.3 Načítání

K načítání dat z databáze slouží sada metod, které vracejí data v odlišných formách pro různé účely použití [2]:

- fetchAll() - Základní metoda vracející dvourozměrné pole klíčované číselným indexem řádku a názvem sloupce. Způsob vrácení dat je možné ještě modifikovat přepínačem na adaptéru metodou setFetchMode().

  Přepínače:

  - FETCH_ASSOC – defaultně nastaveno
  - FETCH_NUM – indexace pomocí čísel řádků a sloupců
  - FETCH_BOTH – kombinace FETCH_ASSOC a FETCH_NUM
  - FETCH_OBJ – pole stdClass objektů
- fetchAssoc() - jako index je použita hodnota prvního sloupce
- fethcCol() - vrací jen hodnoty prvního sloupce indexované od nuly
- fetchPairs() - vrací pole, kdy první sloupec je určen pro indexaci a druhý jako hodnota
- fetchRow() - výsledkem je pouze jeden řádek, kdy jsou jako klíče použity názvy sloupců
- fetchOne() - vrací pouze hodnotu v prvním řádku a sloupci

2.1.7.4 Vkládání

Vložení údajů do databáze je obstaráno metodou insert(), kdy je nutné jako první parametr uvést jméno tabulky, kam se má záznam uložit a jako druhý pole reprezentující záznam klíčovaný jmény sloupců. Po úspěšném vložení záznamu metoda vrací primární klíč záznamu.

2.1.7.5 Změna

Měnit údaje v databázi lze pomocí metody update(), která akceptuje tři parametry. Prvním je název tabulky, druhým pole dat klíčovaným jmény sloupců a třetím specifikace

podmínky, které záznamy budou měněny.

2.1.7.6 Mazání

Operace mazání je přístupná pomocí metody delete(). Metoda akceptuje dva parametry. Prvním je název tabulky a druhým specifikace podmínky, které záznamy se smazají.

2.1.8 Zend_Db_Select

Komponenta Zend_Db_Select slouží k objektově orientovanému vytváření SELECT dotazů. Odstiňuje nás od rozdílnosti dotazů pro různé databázové systémy. Rovněž zajišťuje i správné pořadí jednotlivých sekvencí dotazů. Posloupnost volání jednotlivých metod při sestavování dotazu tedy není důležitá. Další funkcí komponenty je ošetření vstupních dat, která do jednotlivých metod při sestavování dotazu posíláme v rámci parametrů.

Dotazy do databáze je možné vytvářet pomocí sady metod intuitivně pojmenovaných podle jednotlivých sekvencí SQL dotazu SELECT:

- from() - Pomocí této metody definujeme, jaké mají být zdrojové tabulky a z jakých sloupců mají být načítána data.
- join() - Touto metodou specifikujeme propojení k dalším tabulkám podle pravidel definovaných jako druhý parametr metody, třetím parametrem jsou pak sloupce, z kterých se mají v propojené tabulce načítat data. Pro různé typy spojení existují další příbuzné metody:
    - joinLeft()
    - joinRight()
    - joinFull()
    - joinCross()
    - joinNatural()
- where() - Slouží k definici podmínek, podle kterých se mají načítat záznamy z tabulek. Syntaxe je stejná jako v SQL. Metoda umožňuje nahrazovat zástupné znaky

„?“ za ošetřená vstupní data vložená jako druhý parametr.

- group() - Specifikuje podle jakých sloupců se mají seskupovat data. V SQL se vypíše jako „GROUP BY“.
- having() - Umožňuje definovat podmínky, které se aplikují na skupinu řádků.
- limit() - Touto metodou redukujeme počet výsledných záznamů. Jako druhý parametr můžeme zadat počet řádků, které se mají přeskočit.
- order() - Umožňuje definovat sloupce seřazené podle priority, podle kterých se mají záznamy řadit.
- distinct() - Metoda vyřazuje z výsledku duplicitní záznamy.

2.1.9 Zend_Db_Table

Komponenta Zend_Db_Table implementuje návrhový vzor Table Data Gateway. Třída, která extenduje abstraktní třídu Zend_Db_Table_Abstract reprezentuje tabulku v databázi. Této třídě můžeme nastavit atribut $_name, který značí jméno tabulky a atribut $_primary, který značí primární klíč tabulky. Toto nastavení je nepovinné. V případě absence nastavení se bere jako název tabulky jméno třídy a primární klíč se vyčte přímo z popisu tabulky z databáze.

Při vytváření nové instance třídy, která je potomkem třídy Zend_Db_Table_Abstract, musíme jako parametr konstruktoru uvést objekt databázového adaptéru, tedy zdroj, z kterého se mají načítat dat. Na tomto objektu je pak možné volat sadu základních metod pro práci se záznamy tabulky:

- fetchAll(Zend_Db_Select $select = null) - načítání všech dat vyhovujících kritériu předanému v parametru, vrací objekt Zend_Db_Table_Rowset, tedy sadu záznamů (objektů Zend_Db_Table_Row)
- fetchRow(Zend_Db_Select $select) – načítání jednoho řádku, vrací objekt Zend_Db_Table_Row
- find(array $ids / $id) – vrací objekt Zend_Db_Table_Row podle id hodnot primárního klíče
- insert(array $data) – vloží nový záznam do tabulky

- update(array $data, $where) – upraví záznamy v tabulce vyhovující předané podmínce zapsané v SQL
- delete($where) – vymaže záznamy v tabulce vyhovující kritériu předaném v parametru

3 Ostatní frameworky

Framework je vyvíjen v současné době již přes deset let a během této doby přestal postupně dostačovat požadavkům, které na něj kladl rapidně se rozvíjející obor, jakým webové aplikace jsou. Během této doby se původně desktopové aplikace a informační systémy začaly přesouvat na web, a ty kladly vyšší nároky jednak na vývoj, ale i zabezpečení a výkon technologií, ve kterých měly být implementovány na webu. Současně s tím se rozvíjely databázové a frontendové technologie, které framework nebyl schopen obsáhnout. Vznikalo také mnoho dalších konkurenčních frameworků, které vždy lépe či hůře odrážely aktuální trend vývoje webových aplikací. Proto následovaly nové verze 2 a 3, jejichž úkolem bylo napravit tento deficit a reflektovat moderní směr vývoje PHP aplikací. Druhá verze však trpěla řadou neduhů jakými bylo například příliš složité routování. Třetí verze se naopak objevila příliš dlouho po uvedení druhé a v době, kdy byl již značně populární framework Symfony, jenž již několik let dominuje trhu. Nové verze tedy už nikdy nebyly tak oblíbené a rozšířené jakou byla verze 1, která se objevila jako jeden z prvních robustních PHP frameworků.

3.1 Symfony

Symfony je rychle se rozvíjející open-source framework aktuálně již ve verzi 2 konkrétně 2.6. Stojí za ním francouzská firma Sensio Labs přesněji je autorem Fabien Potencier. Rovněž je podporován a rozvíjen početnou komunitou vývojářů. Framework má velmi obsáhlou dokumentaci. Obsahuje řadu komponent, které se dají použít nezávisle na frameworku například Doctrine 2. Symfony podporuje moderní přístupy v programování jako je Dependency Injection nebo aspektově orientované programování (AOP). [4] Symfony charakterizuje především používání anotací, XML nebo YAML souborů pro konfiguraci aplikace a také logické části, na které se aplikace dělí tzv. bundly.

3.1.1 Bundly

Bundle je v Symfony logická část aplikace, kterou je možné ve většině případů oddělit a znovu použít. Existují univerzální bundly, které se dají stáhnout ze společného repositáře a použít ve vlastní aplikace. Odpadá tak zbytečný vývoj částí aplikace, které již někdo vytvořil a je možné se soustředit se na vývoj specifických částí.

3.1.2 Konfigurace aplikace

Symfony nabízí pro konfiguraci aplikace různé možnosti. Většinou je možné použít pro stejný účel celkem 4 způsoby konfigurace:

- Anotace
- YAML
- XML
- PHP script

3.1.3 Formuláře

Podpora tvorby formulářů je jednou z nejdůležitějších funkcí PHP frameworku. Symfony pro vytváření formulářů nabízí svou vlastní komponentu Symfony\Component\Form\Forms. [5] Ta podporuje kromě vlastního vytváření formulářů ještě: [6]

- zpracování požadavků – zpracování dat z formuláře včetně binárních jako jsou soubory
- ochrana proti CSRF – používá se pro každého uživatele vlastní token
- šablonování – pomocí šablonovacího systému Twig
- lokalizace do různých jazyků – šablonovací systém Twig podporuje pomocí rozšíření TranslationExtension lokalizaci do více jazyků
- validace dat – formulářová komponenta má zvláštní rozšíření ValidatorExtension sloužící pro ověření platných dat odeslaných z formuláře

3.1.3.1 Vytváření formulářů

Pro vytváření formulářů lze zvolit dvě cesty. Formulář je možné vytvořit přímo v controlleru nebo pro formulář vytvořit vlastní třídu FormType extendující třídu AbstractType, což je z hlediska přehlednosti lepší řešení. Tato třída vystavuje ven metodu buildForm() , kde se definují pole formuláře a getName(), která vrací unikátní identifikátor formuláře. Parametry této metody jsou formulářový builder a nastavení. Samotný objekt formuláře se pak vytváří pomocí univerzální metody createForm() v controlleru, která

akceptuje jako parametr vytvořený objekt třídy FormType. [6]

3.1.4 Routování

Pochopit princip routování u frameworků je často oříšek. Nejinak je tomu u Symfony. Pro definici vlastních URL adres ve webové aplikaci je možné v Symfony opět použít všechny 4 cesty, a to anotace, YAML, XML i samotné PHP. Zpracování požadavků probíhá v Symfony probíhá tak, že se nejprve požadavek zpracovává front controllerem, který požadavek předá jádru. Jádro se ptá routeru, jestli neobsahuje nějakou routovací kombinaci vyhovující požadavku. V kladném případě vrací router informaci, jaké kombinaci požadavek vyhovuje a na jaký controller a akci se má směrovat. Kernel pak vytváří objekt odpovědi obsahující informace předané z routeru. [7]

3.2 Laravel

Laravel je open-source framework vyvinutý Taylorem Otwellem. Je určený k vývoji webových aplikací a respektuje architekturu MVC. Framework je distribuován pod licencí MIT. Laravel se drží v popularitě několik let stabilně na předních pozicích, což je způsobeno jednak rychlou učící křivkou tak využitím nejrůznějších nástrojů jako je např. Composer. [8] Laravel využívá několik základních komponent z příbuzného frameworku Symfony 2 a dalších stran: [9]

- Doctrine – ORM framework vyvinutý vývojáři frameworku Symfony
- Monolog – knihovna inspirovaná Python knihovnou LogBook sloužící pro logování
- Predis – Redis key-value databáze přizpůsobená jazyku PHP

3.2.1 Service Locator

Framework Laravel je založen oproti jiným PHP frameworkům, které respektují návrhový vzor Dependency Injection, na návrhovém vzoru Service Locator. Service Locator je považován jako návrhový antivzor, jelikož používá k řešení problémů nesprávných přístupů. Se vzrůstající velikostí webové aplikace vzrůstá i silná provázanost mezi jednotlivými třídami a údržba kódu je neúměrně náročná. Výhodou přístupu je jednoduchost u malých aplikací typu webů či blogů, kde je naopak tento přístup jednodušší na používání. [10]

3.2.2 Práce s databází

Laravel podporuje jako řada frameworků různé nejpoužívanější databázové systémy:

- Mysql
- Postgres
- SQLite
- SQL Server

Framework práci s databází usnadňuje jednak nástrojem pro sestavování dotazů do databáze objektově, kdy se jednotlivé sekvence dotazu skládají za sebe tzv. fluent query builder a také nabízí ORM nástroj nazvaný Eloquent ORM. Samozřejmě je možné dotazy psát přímo jako SQL příkazy v případě potřeby. Jednoduchý SQL dotaz se zapíše jako statické volání příslušné metody, pojmenované intuitivně podle názvu příkazu (select, update, delete, drop atd.), přímo na třídě DB, jejímž parametrem je tělo dotazu dalšími parametry jsou pak bindované proměnné, které v těle dotazu nahrazují zástupné znaky. Zde je vidět evidentní rozdíl oproti jiným frameworkům, kde se využívá volání metod na instancích tříd. Laravel používá statické volání metod. Framework rovněž nabízí zajímavý nástroj pro odchytávání volaných dotazů do databáze, který lze využít například pro logování nebo testování dotazů. Používá se pro to metoda listen() volaná rovněž na třídě DB. Listener je nutné zaregistrovat v tzv. Service Provideru. [11]

3.2.3 Query builder

Podobně jako u jiných PHP frameworků nabízí i Laravel svůj vlastní nástroj pro sestavování SQL dotazů objektově a je univerzální v rámci všech podporovaných databázových systémů. Query builder používá systém bindování proměnných do SQL dotazů, čímž zabezpečuje sestavování SQL dotazů proti útokům typu SQL Injection. K dispozici je také nástroj pro tzv. pesimistické zamykání. Principem je, že dokud neskončí transakce, jsou ovlivněné záznamy zamknuty pro úpravu. Sestavování dotazů probíhá vždy tak, že se nejprve na třídě DB staticky zavolá metoda table(), která vytvoří instanci Query builderu. Jejím parametrem je jméno tabulky. Na této instanci je pak možné volat sadu metod, které nám nabízí Query builder pro definování dotazu do databáze: [12]

- get(), all() - Získaní všech záznamů z tabulky – Vrací pole či kolekci objektů jako

instance třídy StdClass. K jednotlivým sloupcům se pak přistupuje jako k atributům.

- first() - Vrací jenom první jeden záznam.
- where() - Podmínku lze upřesnit kombinací s voláním další metody where(), jejímiž parametry jsou jméno sloupce a hodnota, kterou má záznam, jenž chceme vrátit.
- value() - V případě, že nechceme vrátit celý řádek, ale pouze jeho fragment. Omezíme výsledek voláním metody value(). Jejími parametry jsou jména sloupců, která chceme získat ve výsledných záznamech.
- chunk() - Pokud máme rozsáhlou databázi a výsledek dotazu na získaní záznamů z určité tabulky by vrátil tisíce záznamů, je možné volat metodu chunk(), která zpracovává záznamy po dávkách. Prvním parametrem je velikost dávky, druhým parametrem je anonymní funkce zpracovávající jednu dávku záznamů.
- orderBy() - Řazení výsledných záznamů podle sloupce definovaného v prvním parametru metody v pořadí definovaném v druhém parametru.
- take() - Vrací jen určitý počet záznamů. V SQL je ekvivalentem LIMIT.

3.2.4 Eloquent ORM

ORM nástroj Eloquent ORM využívá návrhový vzor ActiveRecord. Databázové tabulky jsou reprezentovány jednotlivě jako samostatné tzv. Modely. Každý model reprezentující databázovou tabulku pak nabízí sadu metod, které kopírují standardní operace nad databázovými tabulkami. Každý Model vychází ze třídy Illuminate\Database\Eloquent\Model a je možné jej vygenerovovat pomocí rozhraní Artisan, které Laravel používá pro generování vývojového prostředí a ovládání frameworku přes příkazovou řádku. Jméno tabulky se odvozuje od jména třídy v množném čísle. V případě, že je jiné, vyplní se explicitně jako hodnota třídní proměnné $table. Primární klíč je možné definovat v třídní proměnné $primmaryKey. Eloquent ORM rovněž umožňuje ovládat automatické doplňování sloupců, jako je datum vytvoření nebo úpravy (v základním nastavení zapnuté) a definovat jejich formát. [13]

Úprava záznamů může být provedena různými způsoby. Jendou možností je načíst

záznam, upravit jeho atributy a uložit pomocí metody save(). Rovněž upravovat záznamy splňující podmínky definované metodou where() rovnou v rámci jedné operace. K tomu slouží metoda update(), jejímiž parametry jsou jména sloupců a jejich hodnoty.

3.3 Nette Framework

Nette framework je PHP framework, který vznikl v roce 2004. Autorem frameworku je David Grudl a přestože je v České republice velmi populární a má poměrně velkou komunitu, je stále jeho hlavním vývojářem a má největší podíl na jeho rozvoji. [14] Nette framework je založený na objektově orientovaném přístupu a je postaven na architektuře MVC (model-view-controller) respektive MVP (model-view-presenter). [15]

3.3.1 Architektura MVC versus MVP

Ve frameworku je využita modifikovaná varianta architektury MVC. Namísto controlleru je použit pojem presenter. Presenter tak jako controller komunikuje s pohledem i s modelem. Taktéž vybírá pohled a poskytuje mu data získaná z modelu. Rovněž je jeho úkolem reagovat na uživatelské podněty. Uživatelskými podněty mohou být: změna pohledu, změna stavu či změna modelu. Zásadním rozdílem v Nette, kterým se liší od ostatních framework, je tvorba odkazů. Odkazy se vytváří dynamicky a kliknutí na ně volá zvláštní metody, které jsu určeny pro volání zvenčí. Aby byly tyto metody odlišené od ostatních a nedaly se volat i jiné nechtěné metody, je v pro ně v Nette použit prefix “handle”. Tímto přístupem se blíží k frameworku ASP.NET, v němž se k odkazům vážou tzv. “handlery”. [15]

3.3.2 Formuláře

Pro vytváření, validaci a zpracování formulářů nabízí Nette framework vlastní komponentu, která je nazvaná Nette\Forms. Standardně jsou formuláře chráněny proti nejrůznějším typům útoků jako je XSS (Cross-site Scripting), CSRF (Cross-Site Request Forgery), UTF-8 attack atd. [16] Formulář se vytváří pomocí nové instance třídy Form, ke které je možné přiřadit pomocí speciálních metod formulářová pole. Tyto metody jsou intuitivně pojmenovány podle názvů jednotlivých formulářových polí. Zde je výčet některých metody pro přidání nejpoužívanějších formulářových polí: [17]

- addText() - přidá formulářový input typu text

- addTextArea – přidává speciální víceřádkové formulářové pole typu textarea
- addRadioList() - přidá formulářová pole typu radio, třetím parametrem je pole obsahující jednotlivé volby, samostatné pole lze přidat pomocí metody addRadio()
- addSelect() - přidá formulářový input typu select, třetím parametrem je pole, které obsahuje volby vypsané do tagů option
- addCheckboxList – přidá formulářová pole typu checkbox, třetím parametrem je pole s jednotlivými volbami, samostatné pole lze přidat pomocí metody addCheckbox()
- addUpload() - přidává pole pro přidání souboru
- addMultiUpload() - přidává pole pro přidání více souborů
- addPassword() - přidává speciální pole pro zadání hesla
- addSubmit() - přidává pole pro odeslání formuláře
- addHidden() - přidává skryté formulářové pole typu hidden

3.3.2.1 Validace formulářů

K formulářovým polím lze navázat pravidla, která musí být splněna pro úspěšné zpracování formuláře. Pravidla se přidávají metodou addRule(), kde prvním parametrem je kritérium, které se bude ověřovat a druhým parametrem je text chybové zprávy. [16] Kritéria se liší podle typu formulářového prvku a výčet nejpoužívanějších je následující: [17]

- Form::FILLED – kontrola, zda je zadána hodnota pole
- Form::EQUAL – kontrola, zda se hodnota pole rovná hodnotě v třetím parametru
- Form::IS_IN – ověřuje, zda je hodnota v jedné z hodnot výčtu
- Form::PATTERN – kontrola oproti regulárnímu výrazu
- Form::INTEGER – ověření, zda je hodnota pole číslo
- Form::RANGE – kontrola, zda je hodnota pole v rozmezí hodnot definovaných v pole zadaném v třetím parametru

- Form::EMAIL – ověření, zda je hodnota pole validní emailová adresa
- Form::URL – kontrola, zda je hodnota pole validní URL adresa

Nette nabízí ještě mnohem víc validačních pravidel, jejichž obsáhnutí je mimo rozsah této práce. Kromě validace na straně serveru je předností frameworku automatické vytvoření validace na straně klienta. [16] Není tedy nutné definovat validační pravidla v PHP a JS a duplikovat tím tak funkcionalitu, což má přínos zejména v přehlednosti a rychlejším změnám.

3.3.3 Databáze

Pro práci s databází nabízí Nette komponenty v namespace Nette\Database. K připojení databáze se používá komponenta Nette\Database\Connection. Práci s tabulkami databáze abstrahuje komponenta Nette\Database\Table a obecné funkce poskytuje komponenta Nette\Database\Context. [18]

3.3.3.1 Připojení k databázi

Nette prostřednictvím komponent Nette\Database\Connection podporuje nejrůznější druhy databází. Plnou podporu nabízí pro 4 druhy databázových serverů: MySQL, PostgreSQL, Sqlite 3 a MS SQL. Připojení je možné vytvořit novou instancí třídy Connection, jejímiž parametry jsou standardně DSN, což je soubor parametrů určujících typ databáze, hostname databázového serveru, jméno databáze, případně port. Dalšími parametry jsou volitelně přihlašovací jméno, heslo a pole s nastavením dalších parametrů. [18]

3.3.3.2 Práce s tabulkami

Výběr dat z tabulky se provádí pomocí metody table() volané přímo na objektu vytvořeného připojení do databáze. Tato metoda vrací instanci třídy Selection, která reprezentuje výběr dat z tabulky. Z instance třídy Selection můžeme získat jednotlivé řádky, které jsou reprezentovány jako instance třídy ActiveRow. Komponenta Table poskytuje tabulky jednotlivě jako různé instance a dotazy do databáze jsou rozděleny pro každou tabulku zvlášť. [19]

3.3.4 Dependency Injection

Jedním z principů, na kterém je založen Nette framework, je Dependency Injection. Při Dependency Injection si třídy obecné služby nezískávají samy, ale jsou jim předávány při inicializaci objektů v konstruktoru. Jedním z příkladů takové služby může být připojení k databázi. Potřebujeme jej používat napříč aplikací a nebylo by dobré pro tento účel používat například globální proměnné. V tuto chvíli je vhodné dodat připojení k databázi zvenčí při vytváření objektu jako parametr konstruktoru. O vytváření objektů potřebných pro běh aplikace se stará systémový kontejner. Tam se vytváří obecné objekty například pro připojení databáze nebo zabezpečení. [20]

3.3.4.1 Dependency Injection kontejner

Pojem Dependency Injection kontejner se v Nette označuje továrna sloužící pro vytváření služeb. Vytváření objektů služeb v továrně se používá z důvodu možné změny konstruktoru, ve kterém pomocí parametrů předáváme parametry služeb. Ve chvíli, kdy potřebujeme v rámci Dependency Injection předávat další parametry, upraví se pouze zápis v DI kontejneru oproti přepisování inicializace objektů na všech místech aplikace. Dalším principem, který se využívá v souvislosti s Dependency Injection kontejnerem, je tzv. lazy loading. Ten se využívá například ve chvíli, kdy potřebujeme uchovávat jen jednu instanci databázového připojení. Ta se vytváří až ve chvíli, kdy objekt potřebujeme využít a po opětovném volání se již znovu nevytváří a použije dříve vytvořená instance databázového připojení. [20]

3.3.4.2 Nette\DI\Container

Implementace principů DI kontejneru je v Nette dostupná pod komponentou Nette\DI\Container. Abstrahuje vytváření služeb a objektů pomocí standardizovaně pojmenovaných metod. Služby se vytváří pomocí metody createService, kde na konci uvedeme název služby. Služby budou vytvářené pomocí lazy loadingu, tedy bude dostupná vždy jen jedna instance. Objekty dostupné napříč aplikací se vytvoří pomocí jednoduché metody create s názvem vytvářeného objektu. Zavolání této metody bude vždy vytvářet novou instanci. V Nette je možné vytvářet do kontejneru služby i dynamicky za běhu aplikace. K tomu se využívá metody addService, kde prvním parametrem je název služby a druhým tělo služby zapsané jako anonymní funkce. [20]

3.3.5 Autoloader

Nette framework používá automatické nahrávání souborů tzv. autoloading. Framework se načítá příkazem require 'Nette/loader.php'. Tento příkaz nahraje framework a v případě, že použijeme nějakou třídu frameworku, loader se postará o její načtení. Odbourá se tak nutnost používat příkazy pro načítání každé třídy zvlášť, kterou v aplikaci potřebujeme a zároveň se načítají jen ty třídy, se kterými se skutečně pracuje. Nette pro načítání tříd nabízí navíc ještě nástroj RobotLoader, prostřednictvím něhož můžeme ovlivnit, které adresáře se budou procházet. To můžeme provést metodou addDirectory() volanou na instanci RobotLoaderu, jejímž parametrem je cesta k danému adresáři. Indexované adresáře a třídy v nich se ukládají do cache, takže se při dalším použití použijí uložená data z cache. Cache se na vývojovém prostředí obnovuje automaticky při volání neexistující třídy. Na produkčním prostředí je potřeba cache obnovit po nahrání nové verze aplikace. [21]

4 Architektura MVC

Základním principem, kterým se řídí dnes většina frameworků pro webové aplikace je architektura MVC. Architektura MVC dělí aplikaci na 3 logické celky, a to Model, View a Controller. Toto oddělení má svůj účel v tom, aby bylo možno jednotlivé části snadno upravovat bez vlivu na ostatní části. Model do sebe zahrnuje jednak práci s daty a business logiku, View obstarává uživatelské rozhrání (GUI) a Controller se stará řízení aplikace. [22] Oddělení a komunikace těchto částí spolu se liší podle různých přístupů. Nejčastěji však Controller přímo ovládá Model a výsledek operace posílá do View, kde se zobrazuje uživateli. Uživatel také může přes View a Controller provádět změny na Modelu.

4.1 Model

Jak už bylo řečeno. Model v sobě obsahuje jak data tak i business logiku aplikace. Souhrnně můžeme Model označit jako doménovou logiku aplikace. Do modelu spadají třídy, validátory, moduly, mappery, databázová vrstva atd. Model je ovládán Controllerem, ale sám o sobě nic neprovádí. Od Controlleru dostane nějaký vstup a vrátí výstup, který Controller pošle do View. [23] V Zend Frameworku v případě objektově relačního mapování (ORM) do modelu můžeme zařadit:

- DAO – databázová vrstva, která na základě požadavků provádí CRUD operace na úrovni databáze
- mappery – převodníky, které obsahují převodní mechanismy z objektů do jiných datových struktur a zpět
- moduly – zapouzdřují v sobě standardizované ovládání a operace s entitami např. CRUD
- validátory – obsahují sadu pravidel nutných k akceptování entity například pro operaci ukládání
- třídy – třídy business logiky a také formy pro reálné objekty

Jak vidíme na přehledu. V Modelu v Zend Frameworku je zahrnuta velká část aplikace a je tedy nutné Model rozdělit ještě do podrobnějších logických částí, aby byl splněn požadavek na oddělitelnost jednotlivých celků aplikace a byla snadno a rychle

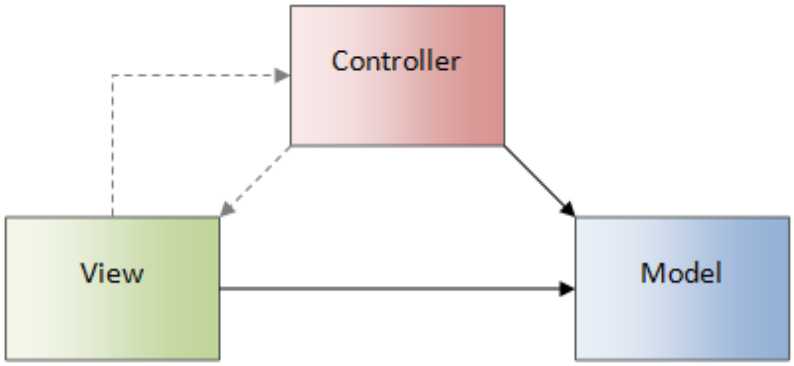
modifikovatelná.

4.2 View

View je vrstva aplikace, která má na starosti zobrazování výsledku zpracování požadavku a také jejím prostřednictvím uživatel může požadavky zadávat a na ně pak reaguje Controller. Často je ve View použit standardizovaný šablonovací systém, který je obdobou frameworku, ale zaměřuje se jen na View vrstvu. V češtině se používá pro View pojem pohled. [24]

4.3 Controller

V češtině používáme pro Controller název řadič. Controller je prostředníkem v komunikaci mezi. Modelem a View. Jeho prostřednictvím uživatel zadává požadavky a podle těchto požadavků Controller reaguje a provádí různé operace s Modelem. Výsledek těchto operací pak vrací zpět do View, kde se zobrazují uživateli. [23]



Obrázek 1 - Schéma spolupráce vrstev MVC [22]

5 Objektově relační mapování

Jedním z problémů při objektově orientovaném vývoji je odlišná reprezentace objektů v aplikaci a datovém úložišti, kterým je nejčastěji relační databáze. Problém částečně řeší použití objektové či dokumentové databáze, ale ty nejsou vhodné pro všechny účely. Řešením převodu objektů mezi aplikací a úložištěm je použití ORM (objektově relačního mapování).

Mezi hlavní výhody ORM frameworků dále patří zapouzdření práce se specifickými databázovými systémy a přenositelnost díky abstraktní databázové vrstvě.

Existují různé způsoby a návrhové vzory, jak ORM implementovat. Pro PHP existuje řada ORM frameworků. Nejznámějšími jsou [25]:

- Propel – Vývoj od roku 2005.
- Doctrine – Novější než Propel. Vyvíjen od roku 2007. Stojí za ním PHP framework Symfony. V současnosti je k dispozici verze 2.

Oba frameworky přistupují k problematice komplexně včetně dalších pomocných nástrojů, které úplně nesouvisí s ORM. Přílišné složitosti se snažila vyhnout verze Doctrine 2, která se zabývá čistě ORM vrstvou. [26] Jako nejdůležitější zástupce ORM frameworků si zaslouží Doctrine 2 představit.

5.1 Doctrine 2

Doctrine 2 je ORM framework určený pro aplikace napsané v jazyce PHP. Inspiruje se obdobným frameworkem Hibernate jazyka Java. Podporuje základní databázové systémy [26]:

- MySQL
- Oracle
- PostgreSQL
- Microsoft SQL Server
- SQLite

Na rozdíl od svého předchůdce je Doctrine 2 založen na návrhovém Data Mapper místo

Active Record. Doménový objekt se nyní nestará o CRUD (create, read, update, delete) operace. O tuto činnost se stará EntityManager. Definice datových typů sloupců a dalších údajů nutných pro ukládání do databáze se určuje pomocí anotací, které jsou uvedeny v komentářích v záhlaví tříd a jejich atributů. Doctrine 2 se skládá ze tří vrstev zajišťujících různé úlohy: [26]

- Common – Obsahuje nástroje k práci s kolekcemi a anotacemi. Definice rozhraní, třídy a knihovny.
- DBAL (Database Abstraction Layer) – Slouží k abstrakci od určitého databázového systému. Poskytuje univerzální dotazovací jazyk DQL (Doctrine Query Language)
- ORM (Object-Relational Mapping) – Stará se o objektově relační mapování.

V modelové vrstvě se vyhneme nasazení hotového řešení v podobě ORM frameworku, ale z části se budeme řídit a respektovat jeho paradigma. Základním principem, kterým se inspirujeme bude mapování objektů z entit do formátu akceptovatelného databázovou vrstvou a naopak.

5.2 Mapper

Pro účel převodu mezi surovými daty z datového úložiště a objekty používáme převodní mechanismus popsaný v tzv. mapperu. Mapper obsahuje metody sloužící k převodu dat vždy jedním směrem:

- mapDataToEntity
- mapEntityToData

Taktéž je nutné specifikovat, jak se bude záznam reprezentující mapovaný objekt v úložišti identifikovat. Nejčastěji takovým údajem bude sloupec primárního klíče tabulky, který se bude mapovat do objektu jako id. Jméno tabulky a primární klíč definujeme v DAO (Data Access Object).

Každá entita pak má svou převodní tabulku a je možné doplnit i další metody k převodu do různých typů úložišť jako jsou například:

- dokumentová databáze

- cache
- relační databáze

Použití mapperu je pak úkolem modulu, který si na mapperu při jednotlivých CRUD operacích volá příslušné metody:

- vytváření – Nejprve jsou data převedena z formuláře jako datový typ pole do příslušné entity a poté z entity do formátu určeného převodníkem: např. pole indexované podle sloupců databázové tabulky.
- načítání – Podle identifikačního klíče jsou data načtena z úložiště a dle převodníku převedena do entity, která je již použita libovolným způsobem například pro výpis, nebo úpravu.
- úprava – Upravená entita je přemapována do formátu příslušného úložišti a poté na základě identifikačního klíče uložena.
- mazání – Záznam se shodným identifikačním klíčem je z úložiště smazán.

6 Nadstavba Zend Frameworku

Během několikaleté práce se Zend Frameworkem se postupně vyvinula nadstavba, která tento framework obohacuje o pokročilé objektově relační mapování a zapouzdření logických jednotek aplikace do tzv. modulů.

6.1 Analýza

Primárním požadavkem pro udržitelnost vývoje v Zend Frameworku byla revize architektury modelu a práce s ním. Původní řešení, kdy mapper plnil roli jednak mapování, tak i ovládání databázové tabulky, bylo nevyhovující. Snahou bylo jednotlivé články rozdělit tak, aby každý plnil svou jednoznačnou úlohu a spolupráci mezi nimi řídil jeden centrální prvek, tedy modul.

6.1.1 Nadstavba komponent

Přestože Zend Framework pokrývá téměř veškeré aspekty vývoje webové aplikace, postupem času byla potřeba práci s některými komponentami zjednodušit, nebo naopak jejich funkcionalitu rozšířit. Díky tomu, že framework obsahuje velké spektrum oddělitelných komponent, bylo jejich rozšiřování snadné pomocí extenze bázových tříd frameworku a umístěním do stejné struktury jako má framework ve zdrojové složce lib. Tudíž bylo zajištěno oddělení rozvíjející se vrstvy nadstavby frameworku a frameworku samotného, který se dal stále aktualizovat na novější verze.

6.1.2 Modelová vrstva

Rozvoj komponent byla jednou z částí vývoje nadstavby frameworku. Druhou byla kompletní změna pojetí modelu tak jak je realizován v Zend Frameworku. Toto řešení bylo nedostačující a proti principům objektově orientovaného programování. Hlavním neduhem byla kombinace mapování s databázovou vrstvou, kdy se o ukládání, načítání dat a jejich mapování starala jedna třída, kde se navíc například opakoval kód pro mapování jednoho a více záznamů. [27] Prvotním cílem bylo tedy oddělit tyto jednoznačně oddělitelné funkcionality mapování a databázovou vrstvu do specializovaných tříd (Mapper a DAO) a maximum obecné funkcionality vytěsnit do abstraktních tříd či rozhraní. Samotné entity respektive jejich třídy pak měly být pouze jakýmisi

přepravkami na data a obsahovat pouze atributy a obslužné metody pro tyto atributy jako jsou gettery, settery, hasery a isery a metody určující chování těchto entit. Prostřednictvím mapperů se pak databázová data transformovala do entit a obráceně. Tím bylo provedeno základní rozdělení modelové vrstvy tak, aby každý její prvek vykonával co skutečně ze svého pojmenování a určení má.

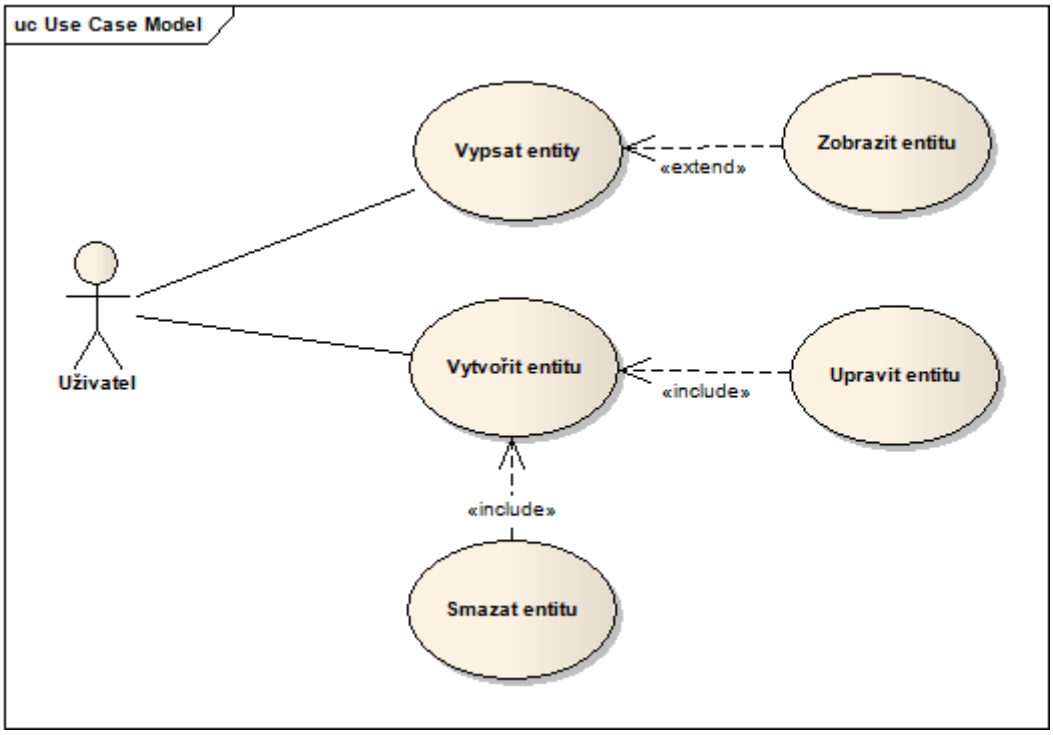
6.1.2.1 Zapouzdření pod modul

Další fází vývoje modelové vrstvy bylo tyto jednotlivé prvky provázat a zapouzdřit jejich spolupráci. Vznikl tzv. modul tedy správce, který se stará o práci s databázovou vrstvou prostřednictvím DAO objektu reprezentující připojení ke konkrétní databázové tabulce a prostřednictvím mapperu transformuje na entity a obráceně z entit prostřednictvím mapperu transformuje data na datovou strukturu používanou konkrétní databází a posílá do DAO, které se stará již jen o uložení. Každý objekt v aplikaci má pak svůj modul, který poskytuje univerzální rozhraní podporující CRUD operace. Pro načítání dalších podobjektů slouží tzv. loader. Implementován je na úrovni modulu, kde je pro každé načítání podobjektů speciální metoda, která na základě vazby vyhledává v jiném modulu objekty a ty pak doplňuje jako podobjekty. Volání loaderů je obsaženo v dotazovacím objektu query, který mimo jiné obsahuje také:

- filtery - pro filtrování záznamů na základě kriteria
- sortery – pro řazení záznamů na základě kriteria
- modifiery – pro omezení nebo modifikaci načítaných dat

6.1.3 Use Case Model

Základní požadavkem, kterou má umožňovat navrhovaná modelová vrstva, jsou tzv. CRUD operace. Tedy vytváření, editace, výpis a mazání záznamů. Každý modul, který se stará o správu záznamů, bude tuto sadu operací podporovat.



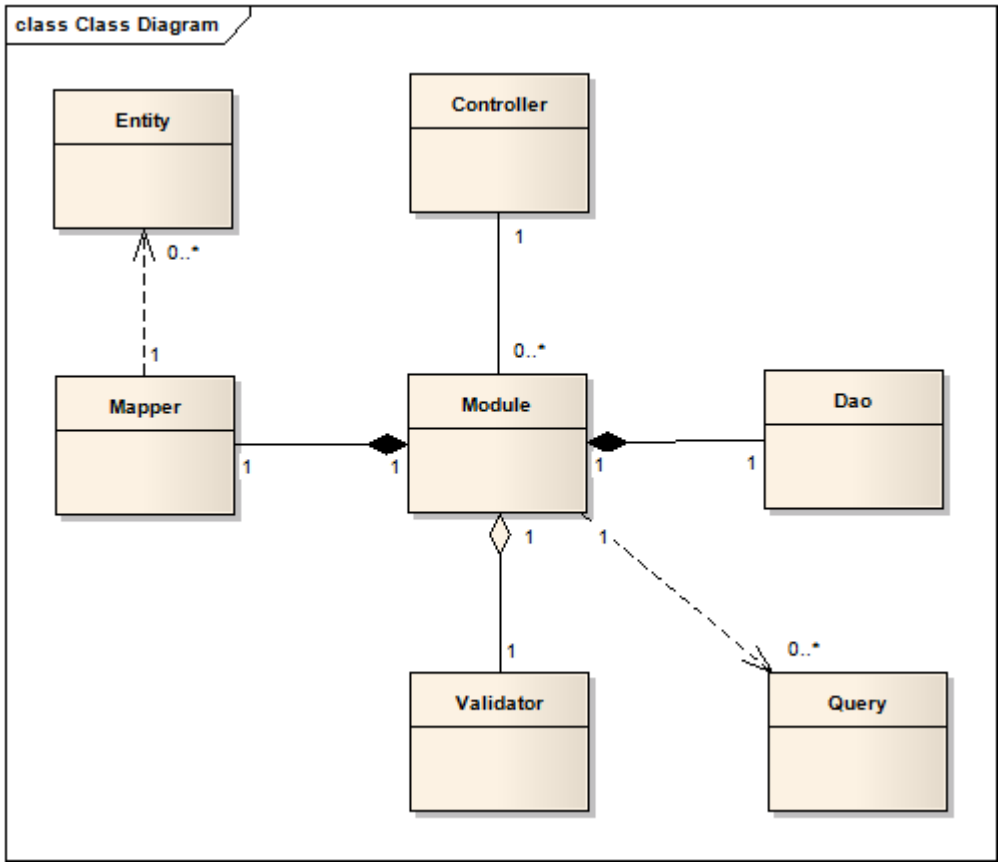
Obrázek 2 - Use Case Model [Zdroj: autor práce]

Případy užití:

- uživatel vytvoří entitu
- uživatel vypíše entity
- uživatel zobrazí entitu
- uživatel upraví entitu
- uživatel smaže entitu

6.1.4 Diagram tříd

Základní kostrou modelové vrstvy je sada tříd, kdy každá má svoji speciální úlohu a mají mezi sebou určité vazby. Vazby a jejich násobnost nejsou pevně dané a zobrazují strukturu platnou pro obecný modul. V praxi může mít modul například více různých datových zdrojů jako je relační nebo dokumentová databáze a tím i připojených více tříd, které reprezentují datový zdroj.



Obrázek 3 - Diagram tříd [Zdroj: autor práce]

6.1.5 Controller

Všechny požadavky, které uživatel vytváří do aplikace, jsou odbavovány speciální třídou, která se nazývá controller. Každý modul systému, který je spravován uživatelem, má vlastní controller. controller umožňuje vystavit akce, kterými je pak ovládán samotný modul. Prostřednictvím controlleru je tedy možné zveřejnit a uživatelem ovládat CRUD operace modulu a další potřebnou funkcionalitu. Controller je tedy úzce svázán s modulem a jméno modulu by mělo korespondovat se jménem controlleru. Názvy akcí jsou pak názvy samotných operací na modulu.

6.1.6 Modul

Modul je ústřední jednotkou celé modelové vrstvy. Každý modul má k dispozici :

- Mapper – slouží k transformaci objektů do dat a z dat do objektů
- DAO – reprezentuje databázovou vrstvu a přístup k záznamům v DB
- Query – třída sloužící pro definování dotazů napříč aplikační a databázovou

vrstvou

- Validator – umožňuje kontrolovat správnost dat na úrovni entit
- Entita – třída reprezentující samotný záznam, slouží jen jako forma na data

Modelová vrstva využívá objektově relační mapování. Tedy že všechny operace by měly fungovat na úrovni entit a transformovat se přes aplikační vrstvu na úroveň databázových dotazů. Při manipulaci s daty se rovněž upravují data na úrovni objektů a při ukládání se objekty pomocí mapperu mapují na pole klíčované názvy sloupců tabulky, které lze snadno uložit do databáze.

6.1.7 Mapper

Pro mapování objektů na data a z dat na objekty slouží speciální třída mapper. Ta obsahuje metody pro účel transformace z databáze do objektů a objektů do databáze. Každý modul má k dispozici jednu třídu mapper. Při načítání dat z DB se data transformují na objekty. Při úprávě nebo vytváření záznamů tedy ukládání se pak objekty transformují do dat.

6.1.8 Entity

Entity reprezentuje reálný objekt, který je abstrahován systémem. V modelové vrstvě se entita plní daty načítanými prostřednictvím DAO a mapovanými mapperem přímo do entity. Celý tento proces ovládá modul. V celé aplikaci se pak pracuje přímo s entitami prostřednictvím modulu a modul tak abstrahuje načítání a ukládání entit do datového zdroje.

6.1.9 Query

Při načítání nebo mazání entit je potřeba vyspecifikovat, jaký soubor dat z databáze se bude načítat případně mazat. K tomu jsou určeny filtry. query však slouží i pro rozšíření načítaných dat o data z dalších modulů. O to se starají loadery.

6.1.10 Validator

Před uložením entity je nejprve nutné ověřit, zda obsahuje data, která jsou validní pro uložení. Entita může být v nekonzistentním stavu nebo obsahovat data, jejichž uložení do databáze by mohlo způsobit chybu. O tuto kontrolu by se měl starat validátor. Validátor by

měl otestovat celou entitu a znemožnit uložení v případě, že je entita v nevalidním stavu.

6.1.11 DAO

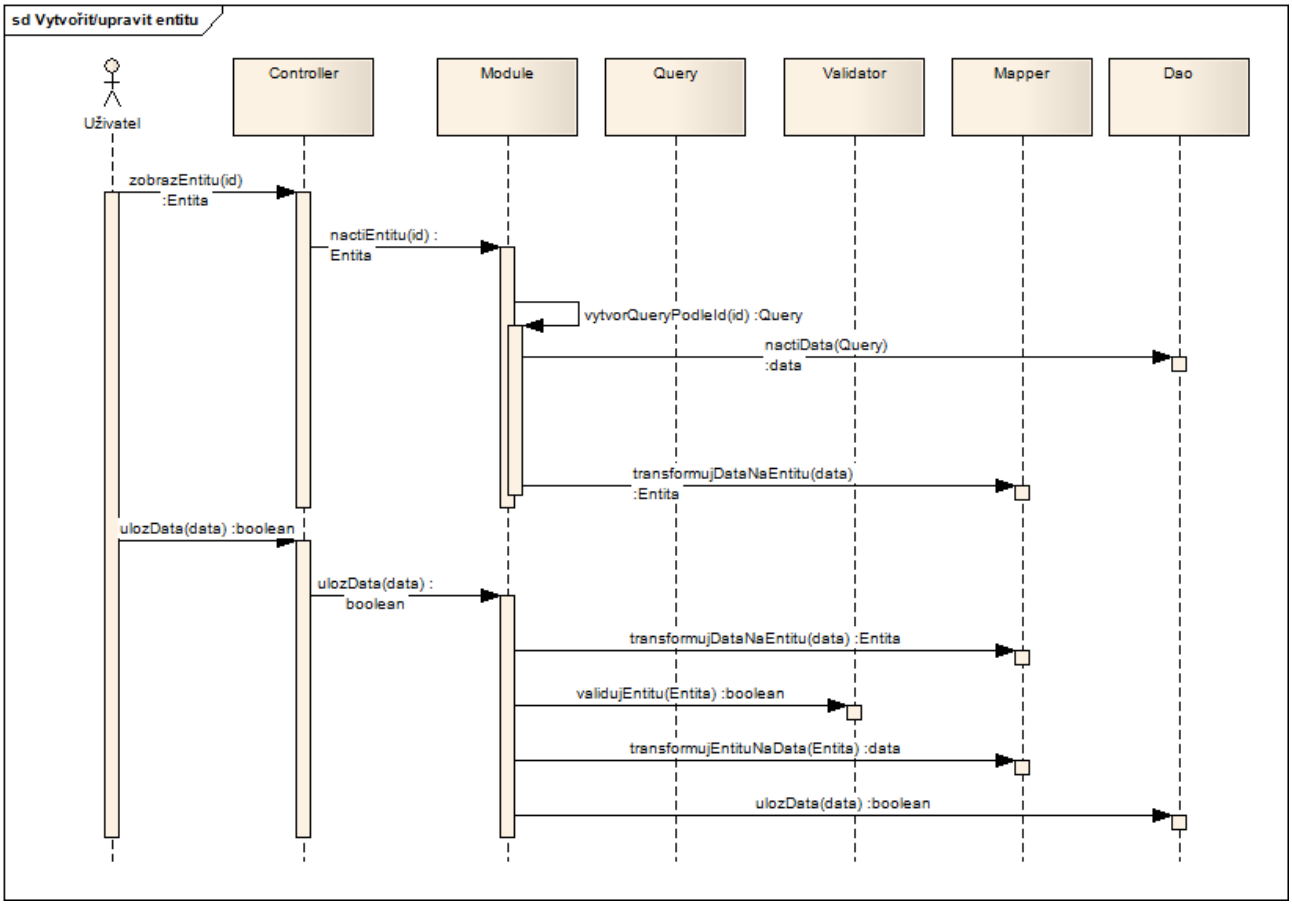
Třída reprezentující zdroj dat a práci s databázovou vrstvou je DAO (Data Access Object). Každý modul má přiřazen jednu třídu DAO pomocí ní komunikuje s databází. Tak jako modul i controller i DAO podporuje stejnou sadu operací CRUD. DAO ovšem již nepracuje se samotnými entitami, nýbrž s daty, do kterých jsou entity transformovány prostřednictvím mapperu.

6.1.12 Sekvenční diagram

Sekvenční diagramy vyobrazují, jak se při jednotlivých CRUD operacích chovají jednotlivé prvky modulové vrstvy. Tyto prvky mají každý svou speciální úlohu a vytvářejí mezi sebou interakci.

6.1.12.1 Vytvoření a úprava entity

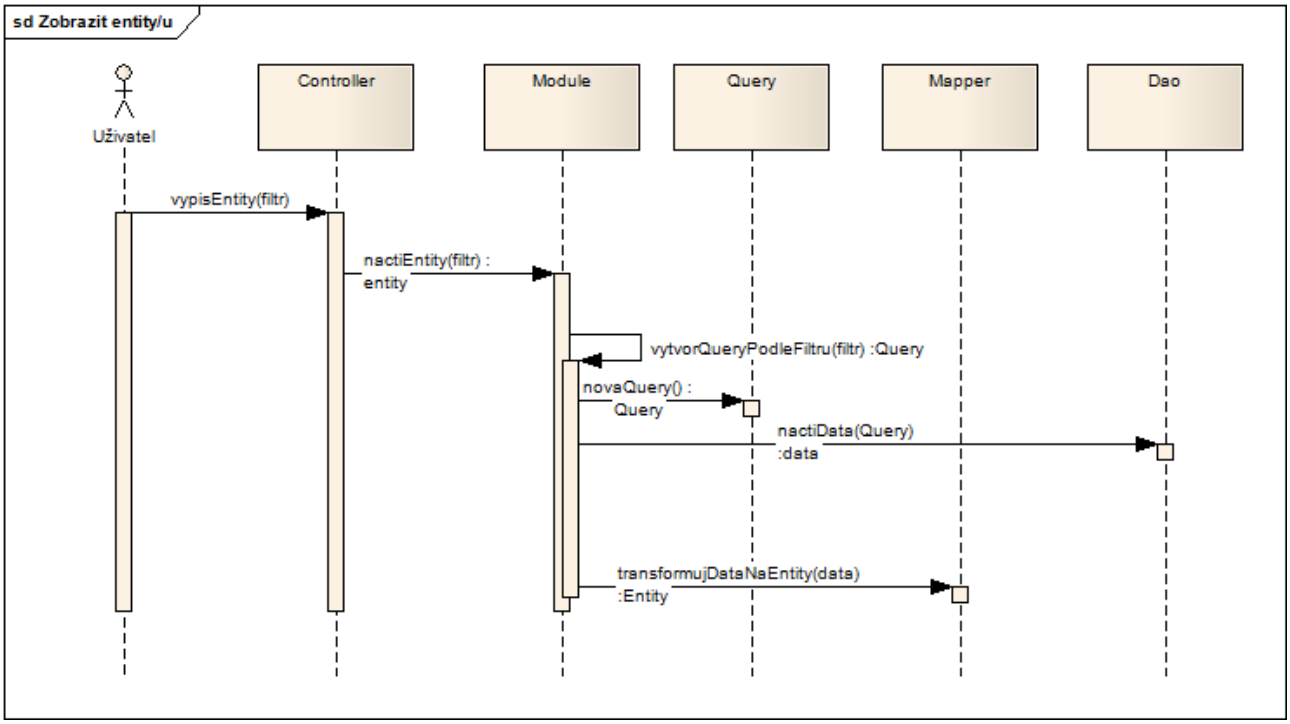
Vytváření a úprava entity probíhá velmi podobně. Jediným rozdílem je, že při editaci se nejprve upravovaná entita načítá pomocí modulu na základě ID záznamu z datového zdroje. Ukládání probíhá tak, že se data z formuláře transformují prostřednictvím modulu do entity. Entita je nejprve validována pomocí validátoru a v případě, že obsahuje správná data a je v konzistentním stavu, je modulem transformována prostřednictvím mapperu do struktury dat, které akceptuje datový zdroj. Modul pak transformovaná data ukládá prostřednictvím DAO do datového zdroje. Výsledek celého procesu je v případě úspěchu nebo chyby vrácen zpět přes všechny vrstvy zpět uživateli.



Obrázek 4 - Vytvoření a úprava entity [Zdroj: autor práce]

6.1.12.2 Zobrazení entity

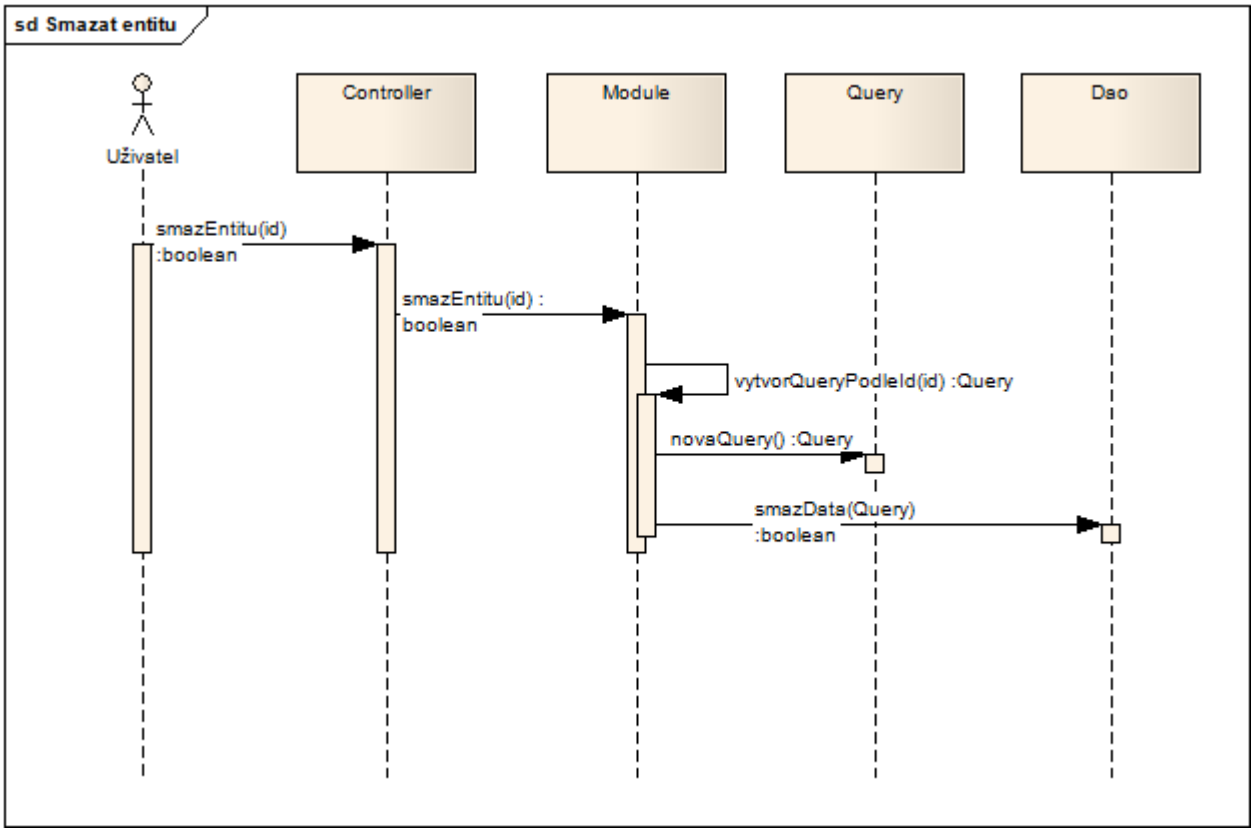
Výpis i zobrazení jedné entity je spojen do jednoho procesu, jelikož se jedná o velmi podobné operace. Při načítání entity se modulu prostřednictvím query sdělí, jaké má ID, případně jiná kriteria, podle kterých lze soubor načítaných entit omezit. Tato kriteria se nastavují jako filtry. V případě, že nespecifikujeme query, jsou načteny všechny záznamy, které jsou uloženy v datovém zdroji. Query a její filtry se v DAO transformují na databázový dotaz a jeho výsledek je modulem poslán do mapperu, kde je namapován na entitu případně entity. Namapované entity jsou pak vráceny jako výsledek uživateli.



Obrázek 5 - Zobrazení entity [Zdroj: autor práce]

6.1.12.3 Smazání entity

Mazání entity respektive jejího záznamu v datovém zdroji probíhá rovněž prostřednictvím modulu. Každý záznam má svoje ID v datovém zdroji podle něhož jej lze identifikovat. Při komunikaci mezi modulem a DAO se však využívá univerzální query. Proto i kriterium na ID se jako filtr transformuje do query a poté posílá do DAO, kde je z query opět převedeno na databázový dotaz, podle něhož je záznam smazán z datového zdroje. Informace o výsledku operace mazání je pak uživateli přes všechny vrstvy vrácena zpět.

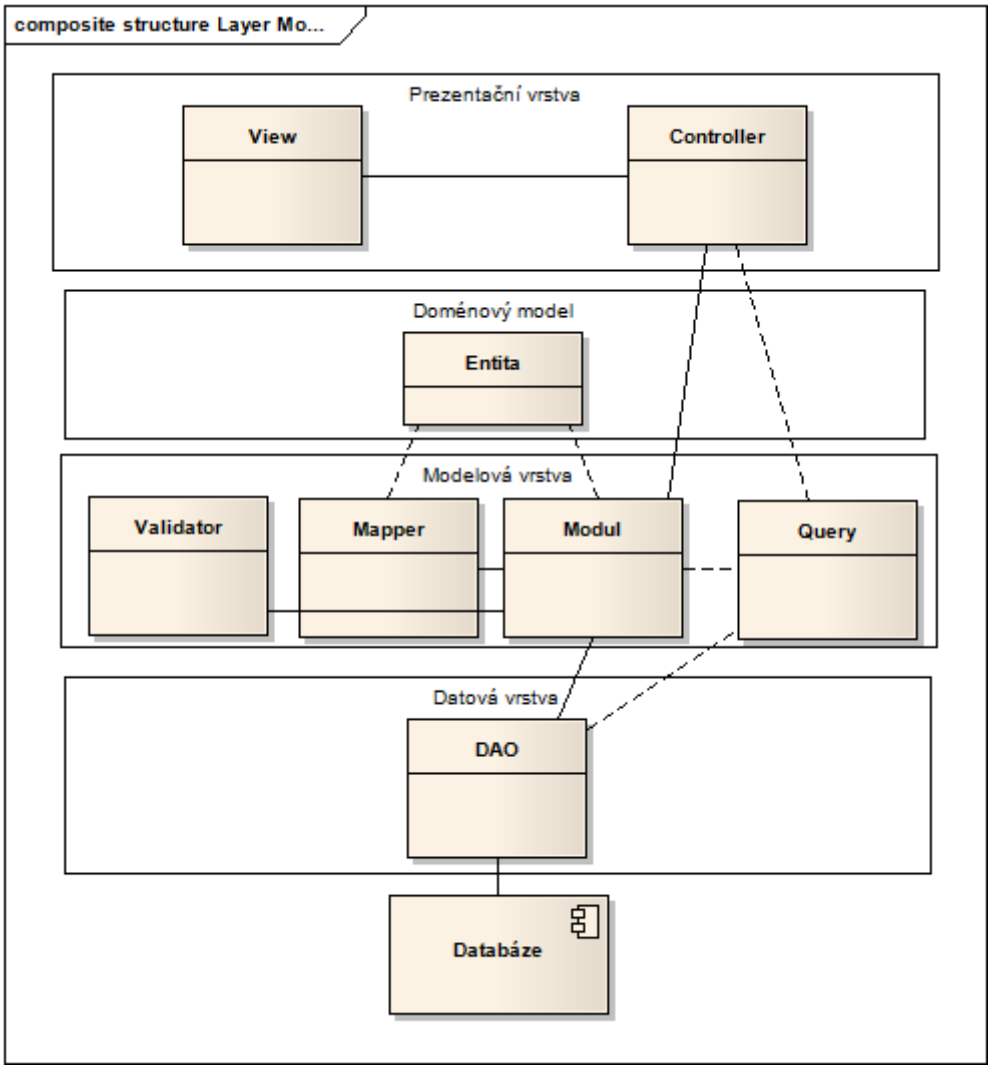


Obrázek 6 - Smazání entity [Zdroj: autor práce]

6.2 Návrh

Architektura je rozdělena do čtyř vrstev:

- prezentační vrstva – interakce s uživatelem a ovládání pohledu a modelové vrstvy
- doménový model – třídy modelující realitu, které jsou navzájem provázané
- modelová vrstva – servisní vrstva obsahující rovněž business logiku
- datová vrstva – napojení na datový zdroj a práce s ním



Obrázek 7 - Architektura tříd [Zdroj: autor práce]

6.2.1 Modul

Modul je základní jednotka modelové vrstvy, která obaluje a pracuje s jednotlivými články modelové vrstvy. Prostřednictvím modulu můžeme pracovat a spravovat entity, které má modul na starosti. Každý prvek modelové vrstvy má za úkol jinou specifickou

činnost. Výsledek své práce předává dál a nestará se, co se děje potom. Modul se naopak stará, aby práci těchto prvků modelové vrstvy sestavoval a řídil. Vždy platí, že pokud chceme použít nějakou funkcionalitu například na úrovni databázové vrstvy, musíme k této funkcionalitě přistupovat prostřednictvím modulu. Takže modul poskytuje rozhraní, které umožňuje pracovat s jednotlivými prvky vrstvy namísto přímého přístupu. Stačí tedy vytvořit instanci konkrétního modulu a pracovat pouze s touto instancí. Díky tomu je jednak kód přehledný a víme, které funkce poskytuje modul ven. Dalším pozitivem je, že se dá kód snadno refaktorovat udržovat. Při refactoringu je pouze nutné zachovat poskytované rozhraní modulu.

6.2.1.1 Načítání dat

Pro načítání dat poskytuje modul metody findAll() a findOne(). Jejich společným parametrem je objekt query s informacemi o tom, které objekty a jak se mají načítat. Obě metody fungují stejně, jediný rozdíl je v tom, že metoda findAll() vrací pole objektů a metoda findOne() vrací pouze jediný objekt respektive první objekt výsledku vyhledávání. Načítání dat probíhá v několika krocích:

- předání objektu query databázové vrstvě a vrácení výsledku z DAO
- transformace dat z databázové vrstvy prostřednictvím mapperu do entit
- vrácení pole entit případně entity jako výsledek

Kromě těchto základních operací se při načítání dat dějí ještě další akce:

- rozšíření načítaných dat o další podobjekty, které se načítají na základě zapnutých loaderů
- nastavení viditelností načítaným objektům pro aktuálního uživatele

6.2.1.2 Ukládání dat

Pro ukládání dat slouží v modulu obecná metoda save(), jejímiž parametry jsou ukládáná entita a příznak, jestli se má ukládání provádět transakčně.

6.2.1.3 Transakční ukládání

Při transakčním zpracování se nejprve vytváří transakce zavoláním metody

beginTransaction() na připojeném databázovém adaptéru. Jméno transakce je určeno konstantou TRANSACTION_NAME přímo na modulu a modul, který chce podporovat transakční ukládání musí mít jméno transakce touto konstantou specifikované. Dále se provede operace ukládání dat a zavolá metoda commit() na databázovém adaptéru. Vzhledem k tomu, že je ukládání ošetřeno odchytáváním výjimek vyvolaných v DAO, tak se v případě odchycené výjimky volá metoda rollback(), která všechny operace provedené v DAO během celé transakce anuluje a vrací zpět úložiště do původního stavu. Dvě transakce rovněž nemohou jít přes sebe, takže v jednu chvíli může ukládat v rámci jednoho modulu jen jeden uživatel. To je důležité zejména kvůli zachování integrity a konzistence dat.

6.2.2 DAO

Data access object ve zkratce DAO je abstraktní rozhraní pro přístup k databázi nebo jiným datovým zdrojům. Účelem je mít v aplikaci standardizovanou vrstvu, přes kterou je možno přistupovat k datům nezávisle na použitém datovém zdroji. Díky tomu můžeme datový zdroj kdykoliv vyměnit za jiný a nemusíme provádět rozsáhlé změny v aplikaci. V modelové vrstvě je DAO reprezentováno třídou App_Model_Dao, která implementuje sadu metod pro přístup k datům a práci s nimi.

6.2.2.1 Načítání dat

Pro načítání dat v DAO slouží metoda findAll(), která akceptuje jako parametr objekt query. Metoda vytváří objekt select typu App_Db_Select, který jako zdroj dat bere databázovou tabulku definovanou v třídní proměnné tableName. Na základě nastavení objektu query se sestavuje objekt select pomocí metody applyQuery(). Metoda applyQuery() musí být implementována v konkrétním DAO příslušného modulu a jsou v ní nastavovány veškeré filtry do objektu select na základě nastavení objektu query. Další metoda důležitá pro sestavení objektu select je metoda getColumnsByQuery(). Tato metoda musí být rovněž implementována v konkrétním DAO modulu a určuje, které sloupce na základě nastavení objektu query budou načítány po provedení select dotazu. Můžeme tak ovlivnit, zda budou načteny všechny sloupce tabulky nebo jen některé, případně budou načteny doplněné sloupce například jako hodnoty uložených procedur či vnořeného dotazu. Dalším důležitým úkonem, který provádí metoda findAll, je aplikace stránkování do dotazu. Na základě nastavení stránkování query objektu tedy čísla stránky a

počtu záznamů na stránku je stránkování převedeno do dotazu select a jsou vráceny jen záznamy určené pro konkrétní stránku výpisu. Stránkování využijeme například při dávkovém zpracování nebo na výpisu záznamů.

6.2.3 Query

K předávání dotazů na získání objektů se v modelové vrstvě užívá speciální tzv. query objekt. Aby byla aplikace nezávislá na použití databázového systému, uchovává v sobě query objekt tzv. filtry, sortery, loadery a modifiery. Query objekt se přes modul dostane až do data access object (DAO) a tam jsou zpracovány hodnoty, které jsou v něm nastaveny. V DAO je na základě podmínek sestaven z query objektu dotaz pro konkrétní databázový systém.

6.2.3.1 Filter

Každý filter slouží pro omezení výsledného výběru vrácených objektů na základě nějaké podmínky. Filter se definuje přímo v třídě query jako konstanta. Uvozuje se klíčovým slovem „FILTER“ a pak následuje konkrétní jméno filteru. Jako hodnota konstanty se volí jednoznačný identifikátor, který se pak využije jako klíč v parametru name v tagu input ve formuláři. Přidávání filteru do query objektu se provádí pomocí metody addFilter(), jejímiž parametry jsou konstanta filteru a hodnota, kterou mají vyhledávané položky splňovat. Metoda přidává do pole filters, což je třídní proměnná třídy query, hodnotu filteru oklíčovanou podle jména filteru. Do filteru můžeme přidávat libovolné hodnoty - ať už řetezce, čísla nebo pole hodnot. Zpracování filteru by mělo být na všechny možné hodnoty připraveno. Na přítomnost filteru v query objektu se ptáme metodou hasFilter(), ta ověřuje, zdali je v poli filters hodnota s požadovaným klíčem. Konečně hodnoty filteru získáváme z query pomocí metody getFilterValues(). Parametrem metody je jméno konstanty filteru. V DAO pak podle nastavených filterů z query sestavujeme výsledný dotaz pomocí objektu select. Většinou při přítomnosti filteru rozšiřujeme podmínku výběru pomocí metody where(), ale může nastat situace, kdy například potřebujeme omezit výběr pomocí vnitřního spojení. To už je ale záležitost, za kterou je odpovědné DAO. Query ví pouze, že má nastavený filter, jeho zpracování a vyhodnocení je již kompetence jiných článků modelové vrstvy.

6.2.3.2 Persistování

Pro účely uchovávání nastavených hodnot v query objektu jako je například nastavení filteru, je možné uložit aktuální stav nastavení do query. Pro ukládání nastavení se používá session. Každý druh dat v query má určen svůj jmenný prostor, do kterého se ukládají oklíčované hodnoty nastavení. Data jsou navíc ještě uložena ve jmenném prostoru, který je určen jménem třídy samotného query objektu. Každé nastavení query je tak jasně identifikováno v session a je možné tak uchovat nastavení například filterů ve výpisu napříč celým systémem. V rámci query je možné uchovávat jen některé hodnoty persistentní. Děje se tak pomocí metody setPersistentValues() volané na query objektu, jejímž parametrem jsou uchovávané hodnoty například filteru oklíčované podle hodnot konstant identifikujících daný filter. Načítání uložených hodnot query ze session provádíme pomocí metody loadPersistentValues() volané taktéž přímo na objektu query.

6.2.3.3 Přípustné hodnoty filtrů

Query objekt poskytuje mimo nastavených hodnot filtrů rovněž i přípustné hodnoty filtrů jako pole objektů. Na konkrétní třídě Query lze implementovat metody, které při zavolání vrací seznam objektů, které jsou akceptovatelné pro výběr hodnoty filtru. Načítání objektů je vyřešeno pomocí lazy loadingu, tudíž se provádí jen jeden dotaz do databáze a mapování při více zavolání té samé metody. Tento přístup nám umožňuje odstranit načítání přípustných hodnot filtru na úrovni controlleru a tím podstatně zvýšit přehlednost kódu. Dále předáváme odpovědnost za načítání objektů pro výběr ve filtru vrstvě, která má mít za tento úkon správně zodpovědnost. Skutečným benefitem je pak předávání pouze objektu query mezi jednotlivými view. Není nutné předávat několik polí objektů pro vypsání do filtrů mezi controllerem a view. Stačí jeden query objekt, který obsahuje jak přípustné hodnoty filtrů, tak rovněž i hodnoty nastaveného filtru.

6.2.3.4 Loader

Loader slouží k rozšíření načítaných dat z úložiště o další data v případě potřeby. Tento přístup nám dovoluje ovládat načítání dat podobjektů například při zobrazení na detailu či editaci záznamu. Naopak při výpisu záznamů data z podobjektů často nepotřebujeme a načítání všech návazností by bylo velmi neefektivní. Při výchozím stavu, kdy jsou loadery vypnuté, se vytvářejí prázdné objekty pouze s identifikátorem. Pokud se loader zapne, na

základě id podobjektů se načtou kompletní data podobjektů a nastaví se místo původních fragmentů podobjektů. Pro každý loader musí být definována metoda na modulu, která se volá při zapnutém loaderu. Uvozuje se klíčovým slovem load a doplní názvem načítaného objektu. Tyto metody mají za úkol efektivně načítat kompletní data objektů a obohacovat jimi cílené entity. Efektivně znamená v tomto případě, že se nejprve získají id podobjektů z obohacovaných entit. Pole těchto id se nastaví jako filter do query objektu a na základě takto sestavené query se natáhnou data podobjektů z modulu podobjektu metodou findAll(). Podobjekty se poté rozmístí namísto původních fragmentů podobjektů zpět do entit a metoda vrací jako výsledek entity obohacené o kompletní podobjekty. Nezáleží tedy, jestli chceme načítat podobjekt či podobjekty pouze do jedné entity nebo do více. Jeden dotaz nám načte všechny pdoobjekty, což šetří systémové prostředky. Loader se stejně jako filter definuje jako konstanta na třídě Query. Název konstanty se uvozuje klíčovým slovem LOAD a pokračuje názvem příslušného načítaného podobjektu. Hodnota konstanty se volí podobně jako řetězec obsahující klíčové slovo load a jméno načítaného podobjektu. Přidání loaderu do query objektu se děje pomocí metody addLoader(), jejímž parametrem je konstanta loaderu. Metoda přidává do pole loaders prvek klíčovaný podle hodnoty konstanty loaderu a jako hodnota se nastaví true. Přítomnost loaderu v query objektu se ověřuje pomocí metody hasLoader(), jejímž parametrem je opět konstanta loaderu. Takto lze snadno prostřednictvím query objektu předávat informaci o tom, zda je loader zapnutý či nikoliv. Vzhledem k tomu, že je query objekt dostupný jak na úrovní modulu i DAO, může se vždy daná vrstva zeptat na přítomnost loaderu a provést činnost, za kterou má v danou chvíli zodpovědnost. Tedy DAO donačítá případně další sloupce s id asociovaných záznamů a modul zase volá příslušné metody loaderů. Většinou je však loader otázka spíše na úrovni modulu.

6.2.3.5 Modifier

Občas se dostaneme do situace, že chceme mít data načtená v trochu odlišné formě, než obvykle bývají. Nebo chceme oproti loaderu omezit rozsah načítaných dat například pro účel vypsání seznamu objektů do filtru, kdy vypisujeme pouze id záznamu a jeho název. V takovém případě nám poslouží tzv. modifier. Na query objektu máme k dispozici metody pro přidání (addModifier(), setModifiers()), získaní (getModifiers()) a ověření přítomnosti modifieru na query (hasModifier(), hasModifiers()). Stejně jako filtry musí

mít i modifier definovanou vlastní konstantu na třídě Query a vlastní unikátní název jako hodnotu konstanty. Na úrovni DAO pak implementujeme, co se má v případě přítomnosti modifieru v query objektu dít. Například je možné omezit výběr sloupců v dotazu select, nebo místo jednoho sloupce podvrhnout jiný, případně zavolat nějakou proceduru tvořící hodnotu sloupce s jinými parametry.

6.2.4 Databázové migrace

Pro verzování databáze se užívají databázové migrace. Každá migrace má svoji vlastní třídu, která extenduje abstraktní třídu AbstractChange a musí implementovat metody up() a down(). Metoda up() se provede po spuštění migrace a jsou zde postupně zapsány dotazy pro úpravu úložiště do výsledného stavu. Metoda down() v sobě naopak obsahuje dotazy, které úložiště vrací do původního stavu před spuštěním migrace. Ve většině případů, kdy se přidávají v rámci jedné migrace tabulky, sloupce nebo nové řádky je možné udělat inverzní operaci, která tabulky, sloupce nebo řádky vymaže. V případě, kdy se data upravují nebo přegenerovávají, inverzní operace postrádá smysl a je v podstatě nemožná. Jediným řešením je v takovém případě vytáhnout původní data ještě před spuštěním migrace a vytvořit zálohu nebo přizpůsobit dotazy v metodě down(), aby vraceli úložiště do původního stavu podle dat ze zálohy.

7 Základní pravidla návrhu softwaru

Nejen při návrhu modelové vrstvy ale obecně při vývoji jakékoliv aplikace je dobré se řídit základními pravidly návrhu softwaru. Umožní nám to, že systém bude možno v budoucnu rozšířit a flexibilně reagovat na změny. Díky tomu nebude budoucí zapracování změn obnášet přepsání velké části kódu, nýbrž na to bude aplikace připravená. [28] Základní pravidla návrhu softwaru vychází z obecných charakteristik objektově orientovaného přístupu.

7.1 Zapouzdření

Jednou z důležitých zásad objektově orientovaného programování je zapouzdření. Třída by měla poskytovat přístup k atributům jen pomocí metod, aby byl přístup k těmto údajům skrytý. Zvenku pak není vidět, jakou cestou jsou údaje získávány, či co se s nimi děje. Známe jenom metodu, pomocí níž údaje z objektu získáme. [28] Mimo atributů se skrývají rovněž metody před okolním světem. Jsou to metody, které využívá třída pro své interní účely. Ven tedy třída poskytuje jen rozhraní určené pro komunikaci s okolním světem. [29] V PHP a jiných jazycích jako je třeba JAVA se pro nastavení práv pro přístup k metodám a atributům tzv. viditelnosti používají klíčová slova:

- private – privátní atributy a metody přístupné jen uvnitř samotné třídy
- protected – rozšiřuje příznak private a to tak, že odvozené samotná třída, ale i odvozené třídy mohou přistupovat k těmto atributům a metodám
- public – plný přístup z okolního světa, slouží jako rozhraní pro komunikaci s okolním světem, public by měly být pouze metody

7.2 Rozšiřitelnost

Už samotný objektově orientovaný přístup dává příslib k tomu, že je aplikace rozšiřitelná. V návrhu rozhraní je však nutné na budoucí možnost rozšíření aplikace dbát. Zpočátku není možné dohlédnout všechny budoucí scénáře vývoje a vyvarovat se všem budoucím problémům při rozšiřování systému. Je však vhodné se nad každým zásadnějším krokem při vývoji zamyslet a položit si otázku, zda nám cesta, kterou se ubíráme, v budoucnu nezpůsobí překážku ve vývoji. Obecně je vhodné a ze svých

zkušeností mohu doporučit, že je výhodné se při vývoji systému modelující reálný svět držet co nejvěrnější reprezentace objektů reálného světa. Nesnažit se tedy transformovat objekty reálného světa složitě do podmínek našeho systému, nýbrž v jeho modelu zachovat všechny jeho vlastnosti a chování, tak jak ve skutečnosti jsou. Samozřejmě s ohledem na to, které vlastnosti a chování v systému využijeme. Každá složitá transformace je totiž závislá na myšlení konkrétního vývojáře a je značně subjektivní. Rozklíčovat poté myšlenkové pochody a záměr vývojáře, který vytvářel kód, v němž zrovna pracujete, je pak obtížné. Naopak ve chvíli, kdy je svět modelován, tak jak je, neměl by mít s pochopením takového kódu nikdo problém. Transformovat objekty reálného světa do objektů v aplikaci co nejvěrněji je v konečném důsledku mnohem rychlejší, protože na všechny otázky ohledně vazeb objektů, vlastností a chování máme vždy odpověď v realitě.

7.3 Znovupoužitelnost

Další důležitou zásadou pří vývoji softwaru je znovupoužitelnost. Naším cílem by vždy mělo být, abychom se vyhýbali opakujícím se sekvencím kódu. Snahou je takové části kódu vytěsnit do speciálních metod. Tyto metody pak voláme na místech, kde se vyskytoval duplicitní kód. Dobrou zásadou je rovněž mít metody logicky rozdělené tak, aby je bylo možné v budoucnu opět použít. Nesnažíme se tedy pouze přesouvat duplicitní kód do zvláštních metod, ale rovněž jej rozdělit tak, aby v různých metodách spolu byly logicky související části kódu a metoda tak byla v budoucnu znovupoužitelná. Jak již bylo zmíněno, někdy neodhadneme budoucí možnosti znovupoužitelnosti. V tom případě je ale na místě v danou chvíli, kdy rozšiřujeme aplikaci o novou funkcionalitu, prozkoumat kód a zkontrolovat, zda již neobsahuje nějaké možnosti pro vytěsnění kódu do zvláštní metody a sloučení podobných funkcionalit.

7.4 Abstrakce

Téma abstrakce již bylo nastíněno, ale je vhodné se jím zabírat podrobněji. Jedná se o jeden ze základních principů objektově orientovaného programování. Abstrakce je činnost, při které převádíme objekty složité reality do objektů modelu. Snahou je se nezabývat nepotřebnými informacemi, které nebudeme v aplikaci používat, nýbrž těmi důležitými. Rozlišit, co je podstatné a co není, je jedním z důležitých úkonů při návrhu. Dalším úkonem při abstrakci je hledat souvislosti a vazby mezi objekty. Je vhodné u

souvisejících objektů najít společné vlastnosti a chování a tyto pak vytěsnit do nadřazené třídy, která bude jejich rodičem v případě, že to má logický smysl. Je nutné všímat si vazeb mezi jednotlivými objekty a tyto vazby reflektovat rovněž i v modelu. [30] Důkladná a správná abstrakce je tedy jedno z nejdůležitějších kritérií při návrhu softwaru.

7.5 Rozdělení do tříd

Z hlediska přehlednosti a udržitelnosti je vhodné jednotlivé části kódu segmentovat do tříd. Pokud programujeme například bránu, která odesílá sms a emaily, není vhodné mít všechnu logiku odesílání prostřednictvím obou médií v jedné třídě brána. Naopak je výhodné pro každou formu odesílání mít zvláštní třídu, která obsahuje stejně pojmenovanou metodu pro odeslání zprávy. Třídy, které odesílají zprávy, mají společnou metodu pro odeslání. Pro takové třídy není vhodné vytvářet abstraktní třídu, nýbrž rozhraní, které budou obě třídy implementovat. [28]

7.6 Vkládání závislostí – Dependency injection

Samotná brána si zatím musí uchovávat informace o obou třídách pro odeslání sms i emailu. To je ovšem špatný přístup. Lepší řešení je, pokud bráně v konstruktoru předáme konkrétní instanci služby pro odeslání sms nebo emailu. Brána ví, že má na tomto objektu zavolat metodu odeslat a o další se nestará. Toto předávání objektů nazýváme vkládání závislostí neboli Dependency Injection. [28]

8 Návrhové vzory

Návrhové vzory usnadňují řešení stále se opakujících rutinních problémů, na které programátoři při vývoji narážejí. Standardizují implementaci podobných úloh a přinášejí pravidla, díky nímž jsou aplikace méně náchylné na chyby. V průběhu vývoje programovacích jazyků vzniklo nespočet různých návrhových vzorů i jejich variant, které jsou vhodné pro konkrétní programovací jazyky, či jsou univerzální napříč všemi jazyky. Každý návrhový vzor byl vytvořen jako nejlepší technika pro řešení konkrétního problému při vývoji aplikace a obsahuje popis, za jakých podmínek a na který problém je možno daný návrhový vzor použít. Dnes se setkáváme zejména s návrhovými vzory určené pro objektově orientované programování, ale toto určení není pravidlem. Existují i návrhové vzory pro strukturální programování atd., ale nás budou zajímat ty určené pro objektově orientované programování. Při vývoji modelové vrstvy byla využita řada návrhových vzorů, které je vhodné představit v konkrétní implementaci pro jazyk PHP. [31]

8.1 Návrhový vzor Singleton - jedináček

Návrhový vzor Singleton – jedináček je jeden z nejpoužívanějších návrhových vzorů a často o jeho použití ani vývojář netuší, protože k němu dospěje přirozeně sám. Singleton v principu slouží k zabezpečení existence pouze jedné instance dané třídy. Děje se tak prostřednictvím dvou přístupů:

- lazy loading – vytváření instance až v době potřeby
- eager loading – vytváření instance pomocí metody getInstance()

Tyto techniky se používají z důvodu toho, že v PHP nelze jinak zabezpečit, aby existovala pouze jedna instance dané třídy. Singleton se používá zejména z důvodu ušetření paměti, kdy by vytváření více instancí zároveň bylo zbytečné. Pro to, abychom zajistili pouze jednu instanci dané třídy potřebujeme vytvořit místo, ze kterého se daná jedna jediná instance třídy bude získávat. Dále musíme zajistit, aby nešla jiným způsobem instance vytvořit. Musíme tedy znemožnit vytvářet jiné instance dané třídy. Místo klasického vytváření instancí pomocí operátoru new zvolíme k vytváření a vrácení instance statickou metodu getInstance() přímo v samotné třídě, kde chceme zabezpečit

pouze jednu jedinou instanci. Tato metoda v sobě obsahuje ověření, zda již neexistuje nějaká instance třídy. V případě, že neexistuje, vytváří novou instanci, ukládá si ji do třídní proměnné například $instance a vrací ji jako výsledek. Naopak v případě, že existuje, novou instanci nevytváří a vrací jako výsledek původní již uloženou instanci. Atribut třídy $instance musí být rovněž jako statická metoda getInstance() s příznakem static. Metodou getInstance() jsme tedy zabezpečili, že při jejím volání dostaneme vždy tu samou instanci dané třídy. Stále je však možné vytvářet instance mimo tuto metodu standardně pomocí operátoru new. Tomu musím zamezit pomocí viditelnosti protected, kterou přiřadíme konstruktoru. Vytváření nové instance pomocí operátoru new nám tedy bude znemožněno a skončí chybovým hlášením, že voláme metodu __construct() bez potřebného oprávnění. Stejně jako jsme znemožnili vytváření instancí pomocí operátoru new, potřebujeme zabezpečit, aby nebylo možno objekt klonovat. To zabezpečíme podobně jako u konstruktoru pomocí viditelnosti. Metodě __clone() nastavíme viditelnost private. Viditelnost protected je u konstruktoru nastavena schválně, aby mohly poděděné třídy přepisovat konstruktor. U metody __clone() takový požadavek není. [28]

8.1.1 Využití

Využití návrhové vzoru Singleton je pestré. Jednak je možné ho využít například pro uchovávání konfigurace nebo databázového připojení. V modelové vrstvě se používá také pro uchovávání instancí jednotlivých modulů, jelikož je cílem mít vždy pouze jednu existující instanci daného modulu. To samé pravidlo se aplikuje rovněž u jednotlivých článků modelové vrstvy. Přístup pomocí jedné instance se v modelové vrstvě tedy aplikuje na tyto články modelové vrstvy:

- Modul
- DAO
- Query
- Mapper
- Validátor

V rámci komponenty System, která je vedle modelové vrstvy se uchovávají rovněž instance jednotlivě. To se týká nejen přístupu k samotné komponentě System kdekoliv je

využita, ale i samotných prvků komponenty System:

- přihlášený uživatel
- připojení k databázi
- práva uživatelů
- nastavení jazyka a státu
- překlady
- konfigurace aplikace

Využití návrhového vzoru Singleton je tedy velmi rozsáhlé. Skýtá však celou řadu problémů.

8.1.2 Problémy použití návrhového vzoru Singleton

Jeden z problémů, který byl již částečně nastíněn je nutnost zamezit vytváření instance pomocí standardních způsobů vytváření instancí tedy pomocí metod __construct a __clone(). To se dá provést buď nastavením omezené viditelnosti protected nebo private, či klíčovým slovem final a vyhazováním výjimek znemožňující použití těchto metod. To je sice funkční řešení, ale problémem je neustále opakující sekvence stejného kódu a rovněž i zodpovědnost třídy řešit, zda má pouze jednu instanci nebo ne, což by správně nemělo být její náplní. Další problém se může objevit při testování, kdy potřebujeme podstrčit místo původní instance například databázového připojení mock simulovaného databázového připojení. To se nám ale nepovede, jelikož takovou instanci nelze vytvořit. Jediným řešením, které se v takovém případě nabízí je odstranit metody znemožňující vytvořit instanci standardním způsobem. Tím ale narážíme na podstatu návrhového vzoru a tím je zabezpečení existence pouze jedné instance. [32] V této situaci stojíme před otázkou, zda opravdu potřebujeme metody, které nám znemožňují vytvářet instance standardním způsobem a navíc je nutné je psát všude, kde chceme zabezpečit pouze jednu instanci. Akceptovatelným kompromisem je tedy tyto restriktivní metody vynechat a spolehnout se na zkušenost a intuici programátorů, kteří dokáží odhadnout, kde je a není nutné vytvářet více instancí té samé třídy. Mezi další úskalí, na které můžeme při použití návrhového vzoru Singleton narazit je situace, kdy potřebujeme nějakým způsobem

parametrizovat vytvářené instance. Například, kdy potřebujeme mít k dispozici bránu na odesílání sms prostřednictvím více operátorů. Jedním z řešení je mírně modifikovat implementaci návrhového vzoru a vytvořit si instanci sms brány pro každého operátora zvlášť. Zajistíme to tak, že metodu getInstance() uzpůsobíme, aby přijímala parametr s identifikátorem operátora prostřednictvím unikátní konstanty. Rovněž atribut $instance uzpůsobíme, aby mohl uchovávat více instancí tím, že jej změníme na pole a hodnoty v poli budeme uchovávat oklíčované právě pomocí speciálních identifikátorů. Tímto způsobem bude možné vždy získat konkrétní instanci, kterou potřebujeme, a zároveň máme zajištěno, že bude k dispozici vždy ta samá instance. [28]

8.2 Návrhový vzor Factory – továrna

Problémem vytváření nových instancí standardně pomocí operátoru new je, že máme na několika místech v aplikaci explicitně určeno, které konkrétní třídě se bude vytvářet instance. V případě, že vytvoříme nějakou třídu, která dědí od původní třídy, jejichž instance jsou vytvářeny na více místech aplikace, nebo v případě změny argumentů v konstruktoru nám vyvstane nutnost změnit na všech místech výskytu vytváření nových instancí kód. Tomu se vyhneme, pokud využijeme návrhový vzor Factory. Ten se, jak už podle jména napovídá, stará o vytváření nových instancí. Factory je speciální třída, která má za úkol vytvářet nové instance pomocí speciální metody uvozené nejčastěji klíčovým slovem create. Jejím parametrem je nejčastěji nějaký typ, podle kterého se vytvoří instance konkrétního potomka bázové třídy. Návrhový vzor Factory tedy využijeme tehdy, pokud chceme zajistit:

- vytváření nových instancí na jednom místě
- rozhodnutí o vytvoření instance konkrétní odvozené třídy od bázové třídy na základě typu
- provedení určitých operací po vytvoření instance podobně jako v konstruktoru

K tomu, aby byla vytvořena instance pomocí Factory není nutné vytvářet instanci samotné factory. Stačí uvést metodu create() jako statickou.

8.2.1 Kombinace návrhových vzorů Singleton a Factory

Pokud z nějakého důvodu potřebujeme pomocí Factory vytvářet a udržovat pouze jednu instanci, je možné zkombinovat návrhové vzory Factory a Singleton tím, že v metodě create() budeme na základě typu vytvářet instance pomocí statické metody getInstance(). Takto zabezpečíme, že továrna vyrábí vždy pouze jednu instanci dané třídy. [28]

8.3 Návrhový vzor Abstract Factory

Návrhový vzor Abstract Factory – abstraktní továrna slouží pro vytváření skupin objektů, které spolu určitým způsobem souvisí. Hodí se zejména v situaci, kdy máme definovaný nějaký objekt se strukturou, kterou chceme odlišně reprezentovat.

8.3.1 Využití

Abstract Factory by mohl být aplikován pro částečné generování modelové vrstvy. Máme objekt, pro který chceme vytvářet nový modul. Tento objekt má atributy, které mají určitý datový typ. Vytváření všech článků modelové vrstvy pro tento objekt je mechanická činnost, kterou můžeme zautomatizovat a využijeme k tomu právě návrhový vzor Abstract Factory. Pro generování se přímo nabízejí tyto články modelové vrstvy:

- Entity – generování atributů, getterů a setterů
- Mapper – generování převodníků pro všechny atributy z pole do entity a zpět
- Migrace – generování migrace pro vytvoření tabulky, jejíž sloupce jsou atributy objektu

8.3.2 Implementace

Nejprve si vytvoříme abstraktní třídu, kterou pojmenujeme například Object. Tato třída bude mít atributy, pro které rovněž vytvoříme abstraktní třídu Attribute. Třída Object má přiřazeny jednotlivé atributy v poli $attributes a pro přiřazení atributu slouží metoda addAttribute(), která jako parametr vyžaduje explicitně instance třídy odvozené od abstraktní třídy Attribute. Každý atribut má nějaký datový typ, pro který taktéž vytvoříme abstraktní třídu DataType. Přiřazení datového typu k attributu je možné pomocí metody setDataType(), která vyžaduje instanci třídy odvozené od abstraktní třídy DataType. Ve

všech třídách vytvoříme abstraktní metodu write(), která v jednotlivých implementacích generátorů bude vypisovat konkrétní část generovaného objektu. Dále potřebujeme vytvořit abstraktní třídu ObjectFactory, ve které budou abstraktní metody createObject(), createAttribute() a createDataType(). Nyní přišel čas na konkrétní implementace generátorů jednotlivých prvků modelové vrstvy. Nejdříve vytvoříme generátor pro třídu nazvanou Entity, která reprezentuje samotný objekt. Tato třída bude nazvána EntityGenerator a bude odvozena z abstraktní třídy Object. Tato třída bude implementovat metodu write(), ve které budou vypisovány jednotlivé atributy a rovněž jejich gettery a settery. Dále budou vytvořeny třídy EntityAttribute, EntityGetter a EntitySetter s odpovídajícím obsahem v metodě write(). To jak bude sestaven kód třídy Entity pak bude záviset na tom, co obsahují metody write() v jednotlivých třídách. Analogicky dotvoříme generátor pro Mapper i migraci a při generování kódu podstrčíme vždy konkrétní Factory daného generátoru. Sestavení struktury entity tedy můžeme mít pouze na jednom místě a vypsání kódu bude vždy záviset na tom, jaký generátor resp. Factory generátoru zrovna používáme.

8.3.3 Kritéria

Pro správné použití návrhové vzoru Abstract Factory se potřebujeme řídit těmito pravidly: [28]

- pro popisovanou skupinu objektů je potřeba vytvořit abstraktní třídu továrny, která obsahuje abstraktní metody pro vytvoření každého objektu ze skupiny
- vytvořit pro každý objekt ze skupiny abstraktní třídu, která je bude reprezentovat
- vytvořit konkrétní implementaci jednotlivých objektů jako potomky abstraktních tříd
- vytvořit konkrétní továrnu jako potomka abstraktní továrny obsahující metody pro vytvoření konkrétních objektů
- používat jen metody továrny při vytváření jednotlivých objektů

Při respektování těchto kritérií je zabezpečeno, že je nutné pro generování celé skupiny objektů odlišným způsobem zaměnit vždy jen konkrétní továrnu. Tato technika nám rovněž umožňuje vytvářet nové továrny, pokud se v budoucnu objeví nějaká nová

forma, do které by se struktura objektů mohla transformovat. Další výhodou je i snadná správa a přehlednost uvedeného řešení. [28]

8.4 Návrhový vzor Decorator

V některých případech, kdy potřebujeme rozšířit funkce nějaké entity v průběhu běhu aplikace, řešíme otázku, jak toho docílit. Intuitivně nás napadne, že bychom mohli využít dědění a vytvořit potomka dané entity, ale to vyžaduje všechny vlastnosti dané entity přemapovat do dědící třídy, což se nejeví jako úplně nejlepší řešení. V takovém případě je vhodné využít návrhového vzoru Decorator.

8.4.1 Implementace

Při implementaci návrhového vzoru Decorator potřebujeme vytvořit abstraktní třídu Decoratoru, která bude vycházet ze stejného rozhraní jako objekt, který budeme obohacovat o novou funkcionalitu. Abstraktní třída Decoratoru bude mít atribut, do kterého budeme přiřazovat objekt obohacovaný o novou funkcionalitu. Tento objekt se bude ukládat při vytváření nové instance Decoratoru v konstruktoru. Dalším úkonem je vytvoření třídy konkrétního dekorátoru vycházející z abstraktní třídy Decorator a přepsání metod, které chceme ovlivnit respektive vytvoření nových metod s rozšiřující funkcionalitou, které budou provázány s metodami a rozšiřovaného objektu. Použití je pak následující: [28]

- vytvoříme instanci rozšiřovaného objektu
- vytvoříme instanci konkrétního Decoratoru a v konstruktoru mu předáme rozšiřovaný objekt
- modifikované metody voláme na úrovni objektů třídy Decorator
- nemodifikované metody voláme stále na rozšiřovaném objektu (tomu se vyhneme, pokud vytvoříme mechanismus delegování metod mezi dekorátory a rozšiřovaným objektem)

8.4.2 Souhrn

Návrhový vzor Decorator má výhodu zejména v tom, že není nutné modifikovat

původní třídu stejně jako u dědění. Další výhodou je oproti použití dědičnosti absence nutnosti přemapování všech vlastností děděného objektu do potomka a stejně jako u dědičnosti možnost použití více různých variant rozšíření původního objektu a uchování těchto variant v různých třídách. Rozšíření můžeme kombinovat mezi sebou, ale v tom případě je nutnost vytvořit mechanismus, který bude volané neexistující metody delegovat na předchůdce zkombinovaných dekorátorů. Volaná metoda neexistující na žádném dekorátoru ani na původním rozšiřovaném objektu nám ale i tak způsobí chybové hlášení. Řešením je v takovém případě vytvořit speciální metodu ověřující existenci metody na rozšiřovaném objektu či na některém z dekorátorů. [28] To však vytváří nutnost tuto metodu volat před voláním metody přítomné na jednom z dekorátorů a ověřovat tak její existenci, což je značně krkolomné a oproti dědičnosti a přemapování atributů nám nepřináší takovou evoluci, jak se nám zpočátku jevila.

8.5 Návrhový vzor Prototype

Mezi další hojně používané návrhové vzory patří návrhový vzor Prototype. Ten na rozdíl od jiných návrhových vzorů, které se starají o vytváření nových objektů šetří množství tříd, které jsou nutné pro zajištění vytváření všech různých variant objektů. Principem tohoto návrhového vzoru je vytváření kopií objektů podle vzoru, kterému se říká prototyp objektu. K tomu, abychom zajistili správné použití návrhového vzoru Prototype, potřebujeme vytvořit továrnu, která bude vytvářet kopie objektů na základě prototypů objektů, jež jsme jí předem dodali. [28]

8.5.1 Implementace

Vezmeme si hypotetický příklad, kdy potřebujeme přiřadit uživatelům různé provizní úrovně. Tyto provizní úrovně jsou reprezentovány třídou ProvisionLevel, která má atribut identifikátor provizní úrovně a dále koeficient, podle něhož se určuje uživateli podíl z celkové částky tzv. provize. Tyto provizní úrovně a koeficienty máme uloženy v databázi a tak je načítáme až za běhu aplikace, proto nemůžeme pro každou provizní úroveň vytvořit zvláštní třídu a musíme vytvářet instance dynamicky. Po načtení provizních úrovní z databáze si vytvoříme jednotlivé instance provizních úrovní s jejich identifikátory a koeficienty. V tuto chvíli máme seznam všech provizních úrovní, ale potřebujeme vytvořit mechanismus, jak tyto provizní úrovně rozdistribuovat mezi

uživatele. Nyní přichází na řadu návrhový vzor prototype. Jak již bylo zmíněno, nejprve je potřeba vytvořit továrnu, které budeme předávat prototypy objektů, podle jejichž vzoru se budou vytvářet jednotlivé kopie objektů. Tato továrna se bude jmenovat ProvisionLevelFactory a bude obsahovat metody:

- addProvisionLevelPrototype() - přiřazení prototypu objektu do továrny podle id provizní úrovně
- createProvisionLevelFromPrototype() - vytvoření objektu provizní úrovně podle id provizní úrovně na základě prototypu

Metoda addProvisionLevelPrototype() bude mít jako první parametr identifikátor provizní úrovně, což je pořadové číslo provizní úrovně v rámci hierarchie provizních úrovní. Druhým parametrem pak bude samotný prototyp objektu provizní úrovně, podle jehož vzoru se budou vytvářet další objekty přiřazené konkrétním uživatelům. Tato metoda bude přidávat do pole $prototypes jednotlivé protypy oklíčované podle svých identifikátorů. Druhou metodou v pořadí je metoda createProvisionLevelFromPrototype(), která se naopak stará o vytváření kopií jednotlivých prototypů objektů provizních úrovní. Jejím parametrem je pouze identifikátor prototypu objektu provizní úrovně. Metoda nejdříve ověřuje, zda vůbec prototyp objektu provizní úrovně s daným identifikátorem v poli $prototypes existuje a v případě že ne, vyhazuje speciální výjimku, kterou je nutno při vytváření kopií objektů tímto způsobem odchytávat. V případě, že takový prototyp objektu provizní úrovně v pole $prototypes již je, nastává kopírování prototypu objektu pomocí operátoru clone a zkopírovaný objekt je vrácen jako výsledek metody.

8.5.2 Problém mělké a hluboké kopie

Při klonování objektů nás může překvapit jedna skutečnost, a to že standardní klonování objektů provádí pouze tzv. mělkou kopii objektu. Mělká kopie znamená, že se vytvoří nový objekt (klon) totožný s tím původním klonovaným, ale veškeré vazby na podobjekty asociované s původním objektem zůstanou tak jak jsou, tedy nevytvoří se kopie podobjektů. Toto chování nás nemusí trápit, pokud klonujeme objekty s jednoduchou strukturou pouze s atributy, které mají primitivní datové typy. Ovšem v

situaci, kdy máme klonovat objekty, které mají vazby na další podobjekty a tyto podobjekty mají vazby na další podpodobjekty atd..., musíme explicitně nadefinovat, jaké všechny podobjekty a do které úrovně se mají klonovat. K tomu slouží tzv. hluboká kopie. Definici mechanismu klonování hlubokou kopií v PHP můžeme zajistit dvěma způsoby:

- přepsáním metody __clone() a definicí mechanismu hluboké kopie
- vytvořením speciální metody pro klonování pomocí hluboké kopie například deepCopy()

Druhý způsob má tu výhodu, že můžeme klonovat objekty dvěma způsoby a stále zachovat možnost mělké kopie. Může se nám totiž stát, že za nějakých okolností nebudeme chtít vytvářet hlubokou kopii například kvůli zvýšené paměťové náročnosti bude se nám hodit pouze mělká kopie.

8.5.3 Nevýhody

Nevýhodou návrhového vzoru prototype je právě nutnost definovat mechanismus vytváření hluboké kopie v situaci, kdy má objekt složitější strukturu podobjektů s více úrovněmi. Tomu bychom se však nevyhnuli ani v případě vytváření nových instancí například pomocí návrhového vzoru Factory. Za této situace by bylo nutno strukturu objektů rovněž nadefinovat a vytvářet znovu, což je analogicky totéž jako vytváření hlubokých kopií.

8.6 Návrhový vzor Adapter

Jedním z dalších návrhových vzorů, který je používán i v modelové vrstvě, je návrhový vzor Adapter. Tento návrhový vzor je používán zejména tam, kde potřebujeme sjednotit více různých rozhraní do jednoho univerzálního, prostřednictvím něhož pak jednotně komunikujeme se všemi subjekty, pro které je tento adaptér k dispozici. Ideální příklad je přístup k různým databázovým systémům. Pro přístup k těmto databázovým systémům existují různá rozhraní, které nabízí podobnou sadu metod pro připojení k databázi, provedení dotazu atd..., ale tyto metody se jmenují různě a rovněž akceptují jinou sadu parametrů. Pokud bychom v naší aplikaci chtěli vyměnit databázový systém za jiný, museli bychom přepsat všechna volání metod konkrétního databázového systému, což by nás stálo neúměrné úsilí. Naší snahou je mít aplikaci či framework nezávislý na

implementaci rozhraní konkrétního databázového systému, tak aby jej šlo v případě potřeby jednoduše vyměnit. Řešením je v takovém případě univerzální adaptér, který nabízí společnou sadu metod pro práci s databázovým systémem. Aplikace či framework pracuje přímo s tímto adaptérem a je odstíněna od rozhraní konkrétního databázového systému.

8.6.1 Implementace

Pro popis implementace použijme již představený příklad týkající se odlišnosti rozhraní databázových systémů. Pro každé takové rozhraní je potřeba vytvořit speciální adaptér, který bude tvořit prostředníka v komunikaci mezi aplikací a daným databázovým systémem. Tento adaptér bude mít přiřazenou instanci databázového systému. Adaptér musí implementovat společné rozhraní všech adaptérů databázových systémů třídy DbInterfaceAdapter. Tedy musí poskytovat metody:

- connect() - metoda pro připojení k databáze
- close() - metoda pro uzavření spojení s databází
- query() - metoda pro vykonání dotazu nad databází
- createDb() - vytvoření databáze
- dropDb() - odstranění databáze
- getDbName() - vrácení jména databáze

Každý databázový systém, který je podporován jazykem PHP tyto metody nabízí. Problém je v tom, že se jmenují různě. Proto adaptér transformuje volání univerzálních metod společného rozhraní do volání určeného konkrétnímu databázovému systému a předá mu parametry upravené do podoby akceptovatelné metodami daného rozhraní.

8.7 Návrhový vzor Bridge

Podobně jako návrhový vzor Adapter funguje i návrhový vzor Bridge s tím rozdílem, že odlišné implementace vytváříme my sami. Účelem návrhového vzoru Bridge je tedy oddělit rozhraní třídy od její implementace. [28] Nejnázornějším příkladem, na kterém lze ukázat použití návrhového vzoru Bridge, je ukládání a načítání objektů z různých typů úložišť. Máme k dispozici relační databázi a souborový systém a chceme vytvořit pro oba

typy úložišť možnost ukládání a načítání objektů tak, aby byl výsledek procesu ať už načítání nebo ukládání stejný a nezávislý na konkrétním typu úložiště.

8.7.1 Implementace

Pro zjednodušení budeme popisovat mechanismus pro ukládání objektu konkrétní třídy například User, abychom se nepohybovali v obecné rovině. Této třídě přísluší modul, který implementuje mimo jiné dvě metody:

- save() - uložení objektu do úložiště
- findAll() - načtení všech objektů z úložiště

Obě metody volají stejně pojmenované metody konkrétního typu úložiště, na které má modul přístup pomocí metody getDao(). DAO symbolizuje konkrétní typ úložiště a přístup k němu. Každý modul má své vlastní DAO a je možné podle potřeby různé formy úložiště pro konkrétní moduly střídat. Důležitou skutečností je, že každé DAO implementuje obecné rozhraní, což mu nařizuje implementovat právě metody save() a findAll(). Tímto máme zaručeno, že ať už v modulu budeme mít jakékoliv DAO, máme jistotu, že pomocí něj můžeme ukládat objekty a rovněž je i načítat. Nyní je potřeba vytvořit konkrétní třídy pro jednotlivá úložiště. První třídou bude DAO pro relační databázi. Metodu save() můžeme jednoduše implementovat tak, že provádí přímo konkrétní databázový dotaz INSERT. Objekt uživatele, tedy instance třídy User, přemapujeme pomocí mapperu z objektu do pole oklíčovaného názvy sloupců v tabulce. Poté stačí už jen data uložit tak, že si vytáhneme klíče pole jako jména sloupců a hodnoty pole jako nový řádek v tabulce. Výsledek metody save() pak bude id záznamu pomocí dotazu do DB na id posledního vloženého záznamu. Metoda findAll() bude fungovat obdobně, ale obráceně. Nejprve provedeme dotaz do databáze na vrácení všech záznamů pomocí dotazu SELECT. Výsledek dotazu je pole dat reprezentující jednotlivé řádky, která jsou oklíčována názvy sloupců. Toto pole proiterujeme a každý prvek pole transformujeme ze struktury pole do objektu pomocí mapperu. Výsledkem této iterace je pole objektů oklíčované svými id, které vracíme rovněž jako výsledek metody findAll(). Tímto máme implementovanou jednoduchou práci s úložištěm typu relační databáze a další na řadě nás čeká souborový systém, konkrétně jednoduchý textový soubor. Objekty, které chceme ukládat, rovněž

nejsou vhodnou datovou strukturou pro ukládání do textového souboru, takže je stejně jako u úložiště typu relační databáze budeme transformovat do pole. Pro ušetření práce použijeme stejné mapování jako u relační databáze. Metodu save() budeme implementovat takto:

1. načtení textového souboru (pro zápis a čtení) reprezentující úložiště objektů
2. přemapování ukládaného objektu do pole oklíčovaného názvy polí
3. vytvoření řádku z pole s políčky oddělenými pomocí oddělovačů
4. vložení řádku do souboru
5. spočítání celkového počtu řádků v souboru
6. vrácení celkového počtu řádků jako hodnotu id vloženého záznamu

Metodu findAll() pak rovněž jako u úložiště typu relační databáze budeme implementovat obdobně:

1. načtení textového souboru (pro čtení) reprezentující úložiště objektů
2. proiterování řádků z textového souboru
3. rozparsování každého řádků podle oddělovačů do pole oklíčovaného názvy políček
4. přemapování rozparsovaného řádku do objektu
5. uložení objektu do výsledného pole oklíčovaného pořadím řádku od 1
6. vrácení výsledného pole objektů zpět jako výsledek metody findAll()

Nyní máme k dispozici obě totožně použitelné verze úložišť a můžeme mezi sebou střídat podle potřeby samozřejmě za předpokladu, že data mezi úložišti synchronizujeme. Objekt a za situace, kdy úložiště předáváme modulu zvenčí, tak ani modul neví, jaké úložiště je v konkrétní chvíli používáno. Už je na naší svobodě, zda implementujeme přístup k dalším typům úložišť jako je například webová služba apod. V modelové vrstvě je například implementován přístup ke couchové databázi.

8.8 Návrhový vzor Flyweight

Účelem návrhového vzoru Flyweight je snížit množství instancí tříd a ušetřit tak

množství paměti potřebné pro udržení velkého množství velmi podobných objektů. Často jsou takové objekty velmi jednoduché a v případě, že mají velmi specifický účel jakým je například volání určité metody, jejíž výsledek závisí na hodnotách konstruktoru, je vhodnější místo držení velkého množství instancí v paměti vytvořit pouze jednu instanci třídy a volání metody uzpůsobit tak, aby její výsledek nezávisel na hodnotách předaných v konstruktoru, nýbrž na parametrech předaných metodě. To jde však proti pravidlům objektově orientovaného programování, jelikož porušuje jedno z nejdůležitějších pravidel, a to konkrétně zásadu o zapouzdření objektů. [28]

8.8.1 Implementace

Nasazení návrhového vzoru Flyweight je jednoduché. Tam kde jsme doposud vytvářeli nové instance velmi podobných objektů použijeme návrhový vzor Singleton a vytváření objektů opatříme kontrolou existence pouze jedné instance. Tím máme zajištěno, že budeme uchovávat vždy jednu instanci dané třídy, na které chceme aplikovat návrhový vzor Flyweight. Parametry konstruktoru, které určovaly stav objektu a výsledek metod, jež chceme transformovat, odstraníme. Odstraněné parametry pak předáváme vždy konkrétní metodě, která na původních parametrech konstruktoru určujících stav objektu závisela. Tím jsme aplikovali návrhový vzor Flyweight a díky této technice máme místo velkého množství velmi podobných objektů uchován v paměti pouze jeden, jehož metody a jejich výstup vždy závisí na předaných parametrech namísto na atributech objektu.

8.9 Návrhový vzor Observer – pozorovatel

V praxi se nám často stává, že potřebujeme sledovat stav nějakého objektu a na základě jeho stavu provádět nějakou činnost. K tomuto účelu slouží právě návrhový vzor Observer tedy pozorovatel. Jeho činnost by klidně mohl provádět samotný objekt, jehož stav chceme sledovat, ale to má jednak úskalí v tom, že znepřehledňujeme třídu daného objektu kódem, za který správně nemá odpovědnost a jednak je problém v údržbě a orientaci v takovém kódu, jelikož po delší době zapomeneme, co který objekt za určitých podmínek dělá. Naše aplikace se tak stává nepřehledná a chování objektů nepředvídatelné. Naopak použití návrhového vzoru Observer v sobě nese oddělení kontroly objektu do speciální třídy.

8.9.1 Implementace

Nejprve je potřeba vytvořit rozhraní pro pozorovatele pojmenované Observer. Toto rozhraní bude vyžadovat od ostatních tříd, které ho budou implementovat, existenci metody check(), jež se bude volat ve chvíli, kdy bude změněn stav pozorovaného objektu. Dále je potřeba vytvořit rozhraní pro pozorované objekty. Toto rozhraní pojmenujeme Observable a budou ho implementovat všechny třídy, jejichž objekty budou pozorovány. Každá třída, která implementuje rozhraní Observable pak obsahuje metodu inform(), která se bude volat při změně stavu, jenž má být monitorován. Metoda inform() má jednoduchou funkci: [28]

- projít všechny pozorovatele
- na každém pozorovateli zavolat metodu check(), která zkontroluje stav pozorovaného objektu

8.10 Návrhový vzor Mediator - prostředník

Často se při objektově orientovaném programování dostaneme do situace, kdy jsou naše objekty příliš svázané díky tomu, že mají pevnou vazbu. V této chvíli nastává zásadní problém se znovupoužitelností tříd, jež reprezentují takové objekty. Hodilo by se nám nějaké komunikační rozhraní, přes které by si objekty mohly univerzálně předávat zprávy. V takovém případě přichází na pomoc návrhový vzor Mediator tedy prostředník. [28]

8.10.1 Implementace

Mějme komunikující objekty například uživatelů, které si předávají zprávy. Každý uživatel může předat zprávu každému uživateli. Vzhledem k tomu, že aplikujeme návrhový vzor Mediator, vložíme do této komunikace zvláštní objekt, který se postará o rozeslání zpráv všem příjemcům. Úkolem odesílatele tedy bude pouze zavolat metodu pro odeslání zprávy a o vše už se postará prostředník. Pro úspěšnou implementaci návrhového vzoru Mediator je potřebné vytvořit: [28]

- rozhraní prostředníka
- konkrétní implementaci prostředníka

- abstraktní třídu, ze které budou vycházet všichni příjemci a odesílatelé zpráv
- konkrétní implementaci příjemců a odesílatelů zpráv

Rozhraní prostředníka bude vyžadovat od všech implementujících tříd implementaci metod send() a register(). Metoda send() se v konkrétní implementaci prostředníka bude starat o odesílání zpráv příjemcům a metoda register() bude přidávat do seznamu příjemců u prostředníka komunikace dalšího příjemce. Důležité je, že metoda register() bude vyžadovat jako parametr instanci třídy, která vychází z abstraktní třídy příjemců a odesílatelů. Tím máme zajištěno, že jim prostředník bude schopen zaslat zprávy. Abstraktní třída příjemců a odesílatelů bude obsahovat metodu send(), která prostřednictvím komunikačního prostředníka, kterého jsme předali při vytvoření instance, zavolá rovněž metodu send(). Mimo metody send(), abstraktní třída vyžaduje, aby odvozené třídy implementovaly metodu processMessage(). Tato metoda slouží ke zpracování zprávy a volá ji prostředník na všech příjemcích zprávy. Každý z příjemců může mít tuto metodu implementovánu jinak a je tak možné na odeslání a zpracování zprávy zareagovat u každého příjemce jiným chováním. Například jeden příjemce může výsledek zprávy uložit do databáze, jiný zase může výsledek vypsat obrazovku.

8.10.2 Nevýhody

Na rozdíl od návrhového vzoru Observer, který je návrhovému vzoru Mediator podobný tím, že reaguje na události, není možné u návrhového vzoru Mediator měnit množinu příjemců zpráv. Abychom toho docílili museli bychom implementaci modifikovat jednak tím, že bychom přidali metody pro odebírání příjemců zpráv a nebo bychom upravili metodu send(), abychom pomocí ní mohli definovat množinu příjemců zpráv.

8.11 Návrhový vzor Command - příkaz

V praxi se často stává, že potřebujeme dynamicky měnit souslednost nějakých procesů během spuštěné aplikace. Tyto procesy jsou jednoduše oddělitelné a navazují na sebe v určitém pořadí. Problémem je, že před spuštěním aplikace neznáme v jakém pořadí mají být prováděny a navíc jsou pro každý objekt volány procesy jiné. K takovému účelu nám poslouží návrhový vzor Command.

8.11.1 Implementace

Jako názorný příklad nám může posloužit ukládání uživatele v aplikaci. Na uživatele v naší aplikaci jsou závislé další systémy, které máme provázané v rámci jedné společnosti. Systémy komunikující s naší aplikací závislé na evidovaných uživatelích můžou být například následující:

- účetní systém
- systém spravující emailové účty
- systém pro evidenci docházky
- systém evidující odvedenou práci

Rovněž máme několik typů uživatelů, které chceme mít synchronizované v rámci některých těchto systémů. Jsou to například:

- administrátor
- ředitel
- zaměstnanec

Při ukládání uživatelů potřebujeme rozlišit, v rámci kterých těchto systémů se bude jaká skupina uživatelů jednostranně synchronizovat. Administrátora potřebujeme evidovat pouze v systému spravující emailové účty. Ředitele potřebujeme mít v účetním systému a v systému pro správu emailových účtů. Management je potřeba evidovat v účetním systému, v systému pro správu emailových účtů a v systémů evidující odvedenou práci a konečně zaměstnance chceme mít propsané do všech těchto systémů. Rovněž chceme zajistit, aby každá skupina uživatelů byla snadno zahrnutelná do dalších systémů. Zde nám velmi výhodně poslouží návrhový vzor Command. Pro vyřešení představeného problému je potřeba vytvořit:

- jednotné rozhraní pro jednotlivé procesy
- třídy reprezentující jednotlivé procesy a implementující společné rozhraní
- třídu vykonávající předem definovaný seznam procesů na objektu

Rozhraní pro jednotlivé procesy je jednoduché, jelikož po třídách, které ho

implementují, požaduje pouze metodu execute(). Tato metoda je univerzální a musejí ji implementovat všechny procesy. V našem příkladu se tedy bude jednat o synchronizaci uživatele do jednotlivých systémů. Metoda tedy pro každý konkrétní systém použije jeho komunikační rozhraní ať už JSON či SOAP a předá mu data o ukládaném uživateli. Poté zbývá vytvořit třídu, která se bude starat o vykonání předem definovaných procesů. Můžeme ji nazvat Executor. Tato třída bude obsahovat dvě metody. První se bude jmenovat addProcessQueue() a jejím úkolem bude přiřazovat množiny procesů do vykonávajícího objektu. Parametry metody bude identifikátor množiny procesů a seznam procesů ve formě pole objektů jednotlivých procesů. Druhá metoda se bude jmenovat executeProcessQueue() a bude se starat o vykonání definované množiny procesů na určeném objektu. Jejími parametry budou identifikátor množiny procesů a objekt, na kterém se množina procesů má vykonat.

8.12 Návrhový vzor Iterator

V jazyku PHP velmi často pracujeme se seznamy. Ať už jsou naprogramovány jako objekty nebo pole, musíme vždy zjišťovat, jakým způsobem s takovým seznamem máme pracovat. A právě k standardizaci práce s datovými strukturami slouží návrhový vzor Iterator. Jazyk PHP navíc v knihovně Standard PHP Library nabízí vlastní rozhraní Iterator implementující tento návrhový vzor a tím nám umožňuje oproti námi definovanému rozhraní výhodu použití cyklu foreach pro instance tříd implementující toto rozhraní. [28]

8.12.1 Implementace

V případě, že se rozhodneme implementovat přímo rozhraní Iterator knihovny SPL, je potřeba vytvořit v třídě reprezentující datovou strukturu a implementující toto rozhraní sadu metod:

- current() - vrací objekt vyskytující se na jednom řádku seznamu
- key() - vrací aktuální pozici v seznamu
- next() - skok na další pozici v seznamu
- rewind() - resetování pozice na začátek seznamu
- valid() - kontrola, zda aktuální pozice v seznamu je platná

Instance tříd, jež implementují toto rozhraní, pak můžeme využívat standardně v cyklu foreach stejně jako pole. Pomocí návrhového vzoru Iterator a standardního rozhraní Iterator z knihovny SPL tedy můžeme zapouzdřit práci se seznamy v různých datových strukturách ať už ve formě objektů nebo polí. Pro práci se seznamy ve formě polí navíc nabízí knihovna SPL vlastní třídu s názvem ArrayIterator, která rovněž implementuje rozhraní Iterator, a zbavuje nás nutnosti vytvářet vlastní třídu pro procházení pole. Použití je jednoduché, stačí při vytváření instance třídy ArrayIterator do konstruktoru vložit pole, které chceme mít zapouzdřené pro procházení v seznamu. Ještě jednou rozšiřující možností knihovny SPL pro práci se seznamy je použití tzv. externího iterátoru. Díky němu lze procházet seznamy paralelně. [28]

8.13 Návrhový vzor State

V objektově orientovaném programování pracujeme s objekty, které se dostávají do určitých stavů na základě jejich nastavených atributů. Na první pohled je řešení problému změny chování na základě nastavených atributů celkem jednoduché. Problém lze řešit soustavou podmínek ověřující aktuální stav a podle toho změnu výstupu. Ovšem se vzrůstajícím počtem stavů je jejich správa a ověřování ve všech metodách velmi pracné a nepřehledné. V takovém případě by se hodila nějaká technika poskytující jasný postup k řešení daného problému. A právě správu stavů a změnu chování objektu na základě aktuálního stavu popisuje návrhový vzor State. [28]

8.13.1 Implementace

Vhodným příkladem k vysvětlení a implementaci návrhového vzoru State je uživatel v aplikaci. Uživatel může mít několik různých stavů:

- host – nový neregistrovaný ani nepřihlášený uživatel
- registrovaný – uživatel již registrovaný, ale zatím nepřihlášený
- přihlášený – registrovaný a zároveň přihlášený uživatel
- odhlášený – registrovaný uživatel, který se nějakou dobu po přihlášení odhlásil

Na základě těchto stavů může uživatel v aplikaci provádět určité operace:

- host – může se registrovat

- registrovaný – může se přihlásit
- přihlášený – může se odhlásit
- odhlášený – může se přihlásit

Naším úkolem je nyní vytvořit společné rozhraní pro jednotlivé třídy stavů uživatele. Toto rozhraní nazveme UserStateInterface a bude vyžadovat od svých implementujících tříd reprezentujících jednotlivé stavy implementaci metod:

- register() - registrace uživatele
- logIn() - přihlášení uživatele
- logOut() - odhlášení uživatele

Uživateli vytvoříme rovněž rozhraní, které bude vyžadovat implementaci metody setState() umožňující nastavit každému uživateli konkrétní stav. Toto rozhraní nazveme jednoduše UserInterface. Dále je potřeba vytvořit abstraktní třídu, ze které budou vycházet jednotlivé třídy všech stavů. V této třídě specifikujeme, že při vytváření nové instance stavu se do parametru konstruktoru bude předávat instance uživatele. Tím máme zajištěno, že stav bude moci zjišťovat aktuální nastavení uživatele a bude možné rozhodnout, zda lze volat metody příslušející danému stavu. Po vytvoření abstraktní třídy následuje implementace konkrétních tříd jednotlivých stavů uživatele, které jsou následující:

- GuestUserState – stav uživatele, který je host – lze volat pouze metodu register(), ostatní metody způsobí vyvolání výjimky
- RegisteredUserState – stav registrovaného uživatele – lze volat pouze metodu logIn()
- LoggedInUserState – stav přihlášeného uživatele – lze volat pouze metodu logOut()
- LoggedOutUserState – stav odhlášeného uživatele – lze volat pouze metodu logIn()

Vzhledem k tomu, že má objekt stavu k dispozici instanci uživatele, může uživateli nastavovat odpovídající stav příslušející volané metodě. Ve chvíli, kdy jsou vytvořeny všechny třídy stavů, je ještě potřeba vytvořit jednotné místo, kde budou všechny dostupné. Takovému účelu se hodí již zmíněný návrhový vzor Factory. Vytvoříme tedy třídu továrny

seskupující všechny stavy uživatele. Tato továrna bude rovněž obsahovat statickou metodu createState() pro vytvoření instance konkrétního stavu. Každý stav bude mít v továrně svůj identifikátor pomocí konstanty. Na základě identifikátoru bude možné vytvořit instanci stavu a to tím způsobem, že se bude metodou createState() předávat rovněž instance uživatele a již jednou vytvořené stavy budou uloženy oklíčované podle svých identifikátorů ve statickém poli $states přímo v továrně. Tím jsme zkombinovali návrhový vzor Factory s návrhovým vzorem Singleton, který díky uchovávání pouze jedné instance konkrétní třídy šetří paměťový prostor. V tento okamžik již nic nebrání implementovat metody změny stavů i do vlastní třídy User. Tato třída bude rovněž obsahovat metody jako jednotlivé třídy stavů. Tedy register(), logIn() a logOut(). Tyto metody budou jednoduše delegovat volání na konkrétní asociovaný objekt stavu. To znamená například, že metoda register() bude volat tu samou metodu na objektu stavu. Poslední úkonem k dovršení implementace návrhového vzoru State je specifikace výchozího stavu tím, že při vytváření nové instance vytvoříme pomocí Factory instanci třídy GuestUserState, které v konstruktoru předáme sama sebe. Tuto instanci pak přiřadíme jako výchozí stav instanci uživatele. Nyní lze měnit libovolně stav uživatele a jednotlivé stavy si každý pohlídají, zda lze přepnout stav do jiného. Rovněž můžeme na uživateli implementovat další metody navázané na konkrétní stav s jistotou, že je uživatel v daném stavu oprávněně. Implementace návrhového vzoru umožňuje dodatečnou změnu sady povolených stavů jednoduchým přidáním další třídy stavu a definováním metod pro přechod mezi stavy.

9 Shrnutí a závěr

PHP frameworky se postupem času staly nejen pro vývoj webových aplikací ale i obyčejných webových prezentací pro vývojáře důležitým nástrojem, který umožňuje se soustředit přímo na implementaci konkrétního projektu a vyhnout se řešení obecných úkolů jako je přístup k datům, zabezpečení nebo kešování. Webové technologie procházejí rapidním vývojem a kladou tak stále větší nároky i na frameworky, které by měly rovněž tento vývoj reflektovat. V průběhu několika let se popularita a podpora mezi frameworky velmi měnila a spolehnout se při vývoji na jeden framework je velmi složité rozhodnutí, které ovlivní směr programátora, vývojového týmu i společnosti vyvíjející webové aplikace na dlouhá léta dopředu.

Tato práce ukazuje, že je možné v PHP frameworku dlouhodobě vyvíjet stabilní projekt, přestože je na trhu již přes deset let. Současně je však nutné framework nejen aktualizovat, ale i rozvíjet jeho funkcionalitu a upravovat jeho architekturu. Díky tomu, že je Zend Framework komponentový framework, tkví přidávání funkcionalit zejména v úpravě samotných komponent respektive jejich rozšiřování či vyvíjení komponent nových. Největšími zásahy jsou pak úpravy architektury frameworku nebo jeho vrstev. Každá úprava musí vycházet ze zkušeností nasbíraných dlouhodobou prací a detailním pochopení nejzákladnějších principů, na kterých framework funguje. Osvědčeným postupem je jednotlivé části rozšiřující vrstvy frameworku spojovat do logických celků a rovněž členit do vrstev, ze kterých pak vycházejí samotné implementace modulů aplikace. Tímto řešením lze v aplikaci udržovat více historických stádií vývoje architektury frameworku i aplikace. Není tedy nutné při každém rozšíření či úpravě architektury frameworku refaktorovat zbytek aplikace, ale u nových modulů již jen vycházet z aktuální verze. Důležitou zásadou při vývoji, která se projevuje zejména na tak důležitých částech jako je jádro či architektura aplikace, je pak dodržení správných principů návrhu softwaru a používání návrhových vzorů. Díky inspiraci osvědčenými postupy řešícími elegantně programátorské nesnáze se vyhneme častým problémům příliš složitých a nečitelných řešení, kterým nerozumí často ani sám autor. Použití správných principů objektově orientovaného přístupu je pak základním kamenem, na kterém se dnes dlouhodobě udržitelný software může stavět.

Z hlediska rozšiřitelnosti a možností úprav se Zend Framework velmi osvědčil. Přesto jeho možnosti v dnešní době již nedostačují aktuálním trendům, které nabízí novější frameworky jakým je zejména Symfony. Ať už je to načítání bundlů pomocí balíčkovacího nástroje Composer, pokročilé šablonovací nástroje nebo velmi rozvinutá možnost nastavení frameworku a bundlů pomocí konfiguračních souborů. Zend Framework však stále nabízí pro konzervativní aplikace jako jsou informační systémy spolehlivou a stabilní technologii.

10 Seznam použité literatury a zdrojů

[1] **Soubor z internetu:** Zend Framework - Overview. (on-line). Zend Technologies Ltd. (citace únor, 15., 2015). Přístup z Internetu: URL: https://framework.zend.com/manual/1.12/en/introduction.overview.html

[2] Marian Böhmer. Zend Framework, programujeme webové aplikace. 1. vyd. Brno: Computer Press, a.s., 2010, 416 s. ISBN 978-80-251-2965-4.

[3] **Soubor z internetu:** Zend Framework – Zend_Db_Adapter. (on-line). Zend Technologies Ltd. (citace březen, 10., 2015). Přístup z Internetu: URL: http://framework.zend.com/manual/1.12/en/zend.db.adapter.html

[4] **Soubor z internetu:** Symfony 2, těší mě!. (on-line). Jiří Koutný (citace březen, 22., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/symfony2-tesi-me/

[5] **Soubor z internetu:** Symfony - Forms. (on-line). Fabien Potencier (citace březen, 23., 2015). Přístup z Internetu: URL: http://symfony.com/doc/current/book/forms.html

[6] **Soubor z internetu:** Symfony – The Form Component. (on-line). Fabien Potencier (citace březen, 25., 2015). Přístup z Internetu: URL: http://symfony.com/doc/current/components/form/introduction.html

[7] **Soubor z internetu:** Symfony – Routing. (on-line). Fabien Potencier (citace březen, 27., 2015). Přístup z Internetu: URL: http://symfony.com/doc/current/book/routing.html

[8] **Soubor z internetu:** Laravel. (on-line). Wikipedia (citace duben, 15., 2015). Přístup z Internetu: URL: http://en.wikipedia.org/wiki/Laravel

[9] **Soubor z internetu:** A beginner's Guide to Laravel 4. (on-line). Laravel Wiki (citace duben, 16., 2015). Přístup z Internetu: URL: http://wiki.laravel.io/A_Beginner%27s_Guide_to_Laravel_4

[10] **Soubor z internetu:** Laravel 4 a statické peklo? Kdepak.... (on-line). Devel.cz (citace duben, 18., 2015). Přístup z Internetu: URL: http://devel.cz/otazka/laravel-4-a-staticke-peklo-kdepak

[11] **Soubor z internetu:** Database: Getting Started (on-line). Taylor Otwell (citace duben, 20., 2015). Přístup z Internetu: URL: http://laravel.com/docs/5.1/database

[12] **Soubor z internetu:** Queries (on-line). Taylor Otwell (citace duben, 22., 2015). Přístup z Internetu: URL: http://laravel.com/docs/5.1/queries

[13] **Soubor z internetu:** Eloquent (on-line). Taylor Otwell (citace duben, 23., 2015). Přístup z Internetu: URL: http://laravel.com/docs/5.1/eloquent

[14] **Soubor z internetu:** David Grudl: Nette Framework čeká zlomový rok (on-line). Martin Hassman (citace duben, 25., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/david-grudl-nette-ceka-zlomovy-rok/

[15] **Soubor z internetu:** Nette Framework: MVC & MVP (on-line). David Grudl

(citace duben, 28., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/nette-framework-mvc-mvp/

[16] **Soubor z internetu:** Nette Framework: Neprůstřelné formuláře (on-line). David Grudl (citace duben, 30., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/nette-framework-neprustrelne-formulare/

[17] **Soubor z internetu:** Forms (on-line). Nette Foundation (citace květen, 15., 2015). Přístup z Internetu: URL: http://doc.nette.org/en/2.3/forms

[18] **Soubor z internetu:** Database (on-line). Nette Foundation (citace květen, 17., 2015). Přístup z Internetu: URL: http://doc.nette.org/en/2.3/database

[19] **Soubor z internetu:** Database Table (on-line). Nette Foundation (citace květen, 17., 2015). Přístup z Internetu: URL: http://doc.nette.org/en/2.3/database-table

[20] **Soubor z internetu:** Dependency injection (on-line). Nette Foundation (citace květen, 20., 2015). Přístup z Internetu: URL: http://doc.nette.org/en/2.3/dependency-injection

[21] **Soubor z internetu:** Auto loading (on-line). Nette Foundation (citace květen, 20., 2015). Přístup z Internetu: URL: http://doc.nette.org/en/2.3/autoloading

[22] **Soubor z internetu:** Úvod do architektury MVC (on-line). Bernard Borek (citace červen, 25., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

[23] **Soubor z internetu:** MVC architektura (on-line). David čápka (citace červen, 26., 2015). Přístup z Internetu: URL: http://www.itnetwork.cz/mvc-architektura-navrhovy-vzor

[24] **Soubor z internetu:** Presenters (on-line). Nette Foundation (citace červen, 28., 2015). Přístup z Internetu: URL: http://doc.nette.org/cs/2.3/presenters

[25] **Soubor z internetu:** ORM frameworky pro PHP5: obecný úvod (on-line). František Tröster (citace červen, 30., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/orm-frameworky-pro-php5-obecny-uvod/

[26] **Soubor z internetu:** Doctrine 2: úvod do systému (on-line). Jan Tichý (citace červenec, 10., 2015). Přístup z Internetu: URL: http://www.zdrojak.cz/clanky/doctrine-2-uvod-do-systemu/

[27] **Soubor z internetu:** Zend Framework – Create Model. (on-line). Zend Technologies Ltd. (citace červen, 12., 2015). Přístup z Internetu: URL: https://framework.zend.com/manual/1.10/en/learning.quickstart.create-model.html

[28] Marian Böhmer. Návrhové vzory. 1. vyd. Brno: Computer Press, a.s., 2012, 320 s. ISBN 978-80-251-3338-5.

[29] **Soubor z internetu:** Zapouzdření v PHP (on-line). David čápka (citace červen, 28., 2015). Přístup z Internetu: URL: http://www.itnetwork.cz/php/oop/tutorial-php-zapouzdreni-objektove-orientovane-programovani/

[30] **Soubor z internetu:** Objektově orientovaný přístup (on-line). Vysoká škola ekonomická v Praze (citace červenec, 10., 2015). Přístup z Internetu: URL:

http://java.vse.cz/pdf/ZaklSWIOOP.pdf

[31] **Soubor z internetu:** Návrhový vzor (on-line). Vojtěch Hordějčuk (citace červenec, 20., 2015). Přístup z Internetu: URL: http://voho.cz/wiki/navrhovy-vzor/

[32] **Soubor z internetu:** Je singleton zlo? (on-line). David Grudl (citace červenec, 20., 2015). Přístup z Internetu: URL: http://phpfashion.com/je-singleton-zlo

Zadání k závěrečné práci

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2015/2016

Studijní program: Aplikovaná informatika
Forma: Kombinovaná
Obor/komb.: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Šimerda Filip	Žďár nad Orlicí 79, Žďár nad Orlicí	I1201765

TÉMA ČESKY:

Zend Framework a jeho nadstavba

TÉMA ANGLICKY:

Zend Framework and extension

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je popsat Zend Framework a jeho komponenty. Kromě samotného frameworku Zend rovněž představit některé hlavní frameworky, které jsou v současnosti na trhu. V praktické části je pak úkolem popsat nadstavbu Zend Frameworku, která byla implementována s využitím návrhových vzorů a správných principů OOP. Zkušenosti nasbírané během vývoje nadstavby frameworku pak shrnout.

Osnova práce:
Úvod
Zend Framework
Ostatní frameworky
Popis komponent Zend Frameworku
Nadstavba Zend Frameworku
Pravidla tvorby softwaru
Návrhové vzory
Diskuze
Závěr

SEZNAM DOPORUČENÉ LITERATURY:

Podpis studenta: ..... **Datum:** .....

Podpis vedoucího práce: ..... **Datum:** .....