Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Popis moderních PHP Frameworků

Bakalářská práce

Autor: Martin Gold
Studijní obor: AI-3

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Hradec Králové Duben 2020

Prohlášení:

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

V Hradci Králové dne 29.4.2020 Martin Gold

Poděkování:

Děkuji vedoucí bakalářské/diplomové práce Mgr. Daniele Ponce, Ph.D. za metodické vedení práce.

Anotace

Bakalářská práce se zaměřuje na popis moderních PHP frameworků (Nette, Symfony a Laravel). Porovnání je zaměřeno jak na frameworky jako na celky, tak i na jednotlivé komponenty. Frameworky jsou představeny na funkčně stejných aplikacích.

Annotation

Title: Modern PHP frameworks comparison

The graduation work is devoted to description of modern PHP frameworks (Nette, Symfony and Laravel). The description is focused on whole frameworks and its individual components. These frameworks are introduced on sample application with equivalent functionality.

Obsah

1 Úvod 1

2 Cíl práce 2

3 Kritéria hodnocení 3

4 Teoretická část 4

4.1 Jazyk PHP 4

4.1.1 Představení jazyka PHP 4

4.1.2 Stručné výhody a nevýhody PHP 4

4.2 Moderní PHP 4

4.2.1 Organizace PHP FIG 4

4.2.2 Balíčkovací systém Composer 5

4.2.3 Typovost PHP 6

4.3 Přechod od skriptování k webovým aplikacím 7

4.3.1 Starý přístup k vývoji 7

4.3.2 Front Controller 8

4.3.3 MVC (Model-View-Controller) 8

4.3.4 Dependency Injection 9

4.3.5 Dependency Injection Container 11

4.3.6 Šablony 11

4.4 Databázová vrstva 12

4.4.1 Active Record 13

4.4.2 Data Mapper 13

5 Praktická část 16

5.1 Instalace frameworků 17

5.2 Běhové prostředí aplikace 18

5.3 Instalace aplikace 18

5.4 Frontendové závislosti 19

5.4.1 Laravel – Mix 20

5.4.2 Symfony – Encore ... 20
5.4.3 Nette ... 21
5.5 Směrování ... 21
5.5.1 Symfony ... 21
5.5.2 Laravel ... 22
5.5.3 Nette ... 23
5.6 Šablony ... 25
5.6.1 Symfony – šablonovací jazyk Twig ... 25
5.6.2 Laravel – šablonovací jazyk Blade ... 26
5.6.3 Nette – šablonovací jazyk Latte ... 27
5.7 Databázová vrstva ... 29
5.7.1 Symfony – Doctrine ... 30
5.7.2 Laravel – Eloquent ... 33
5.7.3 Nette – Database Explorer ... 35
5.8 Konzole ... 37
5.8.1 Symfony ... 37
5.8.2 Laravel ... 39
5.8.3 Nette ... 40
5.9 Databázové migrace ... 41
5.9.1 Symfony ... 41
5.9.2 Laravel ... 42
5.9.3 Nette ... 42
5.10 Controllery ... 43
5.10.1 Symfony ... 43
5.10.2 Laravel ... 44
5.10.3 Nette ... 44
6 Závěr ... 47
7 Seznam použité literatury ... 49

Seznam obrázků

Obrázek 1 Use Case diagram 16
Obrázek 2 Struktura databáze 30

Seznam tabulek

Tabulka 1 Přehled instalačních balíčků 17
Tabulka 2 Porovnání směrování 25
Tabulka 3 Porovnání šablonovacích jazyků 29
Tabulka 4 Porovnání databázových vrstev 37
Tabulka 5 Porovnání konzolí 41
Tabulka 6 Porovnání migrací 42
Tabulka 7 Porovnání controllerů 46

1 Úvod

Dnes je dostupných mnoho webových frameworků s různými účely a přednostmi, proto je těžké zvolit správnou variantu. Existuje mnoho kritérií, dle kterých se dá framework posuzovat, např. rychlost vývoje, kvalita dokumentace, rozšiřitelnost nebo dostupnost zkušených vývojářů. Pro vývojáře, který nemá s PHP frameworky žádnou zkušenost, může být volba složitá a je zde možnost, že sáhne po ne úplně vhodné variantě.

O to je dnes výběr složitější, protože lze kombinovat různé komponenty. Jelikož jsou frameworky koncipovány jako sada komponent, tak často není problém části různě zaměňovat a využít nejlepší komponenty z každého. Proto se práce zabývá nejen popisem celků, ale i popisem jednotlivých částí. Například velice časté je nahrazení výchozí databázové vrstvy v Nette databázovou vrstvou Doctrine ze Symfony. Framework Laravel je postaven na komponentách ze Symfony a následně na Laravelu je postaven micro-framework Lumen, který používá jak komponenty ze Symfony, tak i z Laravelu.

2 Cíl práce

Cílem práce je popsat moderní PHP Frameworky (Nette, Symfony, Laravel) na úrovní jednotlivých komponent tak i celků. Frameworky budou porovnávány na funkčně stejné aplikaci.

3 Kritéria hodnocení

Frameworky jsou hodnoceny na základě vývoje vzorové aplikace MovieDB v každém z frameworků. Každá dílčí část frameworku použita při vývoji bude slovně ohodnocena a srovnána s ostatními. Hlavním kritériem je jednoduchost použití, přehlednost kódu a její možnosti. Pokud framework funkcionalitu neobsahuje nebo alespoň nedoporučuje postup v dokumentaci, tak hodnocen nebude.

MovieDB je aplikací obsahující velice běžnou funkcionalitu tak, aby bylo možné frameworky srovnat na funkčně stejné aplikaci bez využití knihoven třetích stran nebo implementace chybějících funkcionalit a zároveň porovnat komponenty, které jsou potřebné v každé webové aplikaci. Mezi hlavní úlohy aplikace patří vkládání filmů a hodnocení do databáze, evidence uživatelů a přehled všech uživatelů a filmů. Jedná se o velice běžnou funkcionalitu webových aplikací a je k tomu potřeba využít nejzákladnější komponenty frameworků.

V aplikaci není implementované ověřování účtů pomocí emailu a funkce zapomenutého hesla. Frameworky sice mají své způsoby pro odesílání e-mailů, ale tato funkcionalita není pro porovnání frameworků podstatná. Stejně jako logování, které není v aplikaci použito, se jedná o komponentu s velice specifickou funkcionalitou, která není s frameworky nijak svázána a bez větších obtíží ji lze nahradit kteroukoliv jinou knihovnou.

Dále se aplikace a práce nezabývá jednotkovými testy, jelikož doporučovaným způsobem pro Laravel i Symfony je použití testovacího frameworku PHPUnit a pouze Nette má svůj nástroj nette/tester, který je použitím velice podobný.

Laravel a Symfony obsahují velké množství funkcionalit, které nemusí být použity ani ve větších aplikacích jako například fronty, notifikace nebo již zmíněné logování. Oproti tomu Nette se soustředí na vývoj nejnutnější funkcionality a vývoj komponent které poskytují abstrakci nad HTTP cyklem aplikace.

4 Teoretická část

4.1 Jazyk PHP

4.1.1 Představení jazyka PHP

Jazyk PHP je interpretovaný strukturálně a objektově orientovaný jazyk, který se využívá povětšinou na serverech pro tvorbu webových stránek a aplikací.

4.1.2 Stručné výhody a nevýhody PHP

Mezi hlavní výhody PHP patří jednoduchá konfigurace a dostupnost hostingů. Pro spuštění základního dynamického webu stačí, byť v obyčejném textovém editoru, vytvořit PHP soubor a ten pomocí FTP nahrát na hosting. Tento přístup je vhodný maximálně pro doplnění statických stránek o dynamické prvky, ovšem lze velice rychle poznat sílu PHP a vyzkoušet si některý z frameworků, které poskytují základy pro tvorbu složitějších webových stránek.

4.2 Moderní PHP

4.2.1 Organizace PHP FIG

Organizace PHP Framework Interop Group slouží pro tvorbu standardů PSR (PHP Standard Recommendation). Mezi nejdůležitější standardy patří PSR-0 a PSR-4, které umožnily jednotný a pohodlný způsob pro načítání knihoven a vlastních tříd (tzv. autoloading). Nejrozšířenějšími standardy jsou PSR-1 a PSR-2. Ty určují doporučené formátování a styl kódu tak aby se v rámci projektů a knihoven dodržoval jednotný formát. Velice používaným standardem je PSR-19, který se dodnes nachází ve stavu návrhu. Doporučuje způsoby používání PHPDoc anotací. Ty mohou velice důkladně dokumentovat kód a uvádět typy proměnných, argumentů a návratových hodnot kde to z důvodu slabin jazyka PHP není možné. Díky různým dalším standardům bylo docíleno spolupráce frameworků. Jedním z nich je PSR-15 (standard doporučující společný interface pro middlewary). [1]

4.2.2 Balíčkovací systém Composer

Balíčkovací systém Composer vznikl z potřeby snadno instalovat balíčky a rozšíření, což bylo v době před vznikem Composer prováděno ručním stažením souboru s nemožností provádět snadno aktualizace balíčků nebo prostřednictvím balíčkovacího systém PEAR, který je již dnes zastaralý. Hlavní výhodou Composeru oproti předchůdci PEAR je možnost instalovat balíčky pro každý projekt zvlášť a možnost využívat PSR-4 autoloading. [2]

Projekt či knihovna obsahuje manifest (standardně soubor composer.json), který obsahuje všechny závislosti (knihovny). U každé knihovny je uvedena verze ve formátu semver. Semver (sémantické verzování) je formát pro zapisování verzí, který se skládá ze tří čísel oddělených tečkou (MAJOR.MINOR.PATCH). Pokud se navýší třetí číslice (PATCH), tak by měla být zachována stoprocentní zpětná kompatibilita. Toto číslo je povýšeno ve verzích, které zvyšuje výkon nebo opravují chyby. Pokud se mění druhé číslo, tak většinou bývá přidána nová funkcionalita. Změna prvního čísla (MAJOR) značí kompletní přepsání nebo větší sadu změn s mnoha zpětně nekompatibilními změnami, kdy není zachováno veřejně poskytované API. Nejčastěji je v projektu závislost na knihovně definována ve formátu např. ^2.1, což znamená verze větší než nebo rovny verzi 2.1.0 a zároveň verzi menší než 3.0.0. V praxi to znamená, že veškeré výhody vyšších verzí (bezpečnostní záplaty, vyšší výkon) jsou použity se zachováním zpětné kompatibility. [3]

Composer vytváří i soubor nazvaný composer.lock, který zamyká verzi knihoven na konkrétní verzi (pokud je uveden rozsah např.). Pokud tento soubor neexistuje, tak se nainstaluje nejnovější možná verze z daného rozsahu. Tento soubor slouží k tomu, aby Composer dokázal nainstalovat identické verze knihoven na v různých prostředích. Tato vlastnost se může zdát zbytečná, jelikož díky sémantickému verzování by všechny verze se stejným MAJOR číslem měli být kompatibilní, ovšem v praxi k různým chybám a nekompatibilitám dochází, proto vznikl tento soubor.

Může se stát, že celá aplikace projde jednotkovými, akceptačními či ručními testy, přesto na jiném stroji aplikace nefunguje (díky neúmyslné chybě v MINOR nebo PATCH verzi knihovny).

Composer umožňuje instalovat i vývojové závislosti, které nejsou nezbytně nutné k běhu aplikace, ale hodí se při vývoji. Například testovací frameworky, různé lintovací knihovny nebo různé nástroje.

Composer umožňuje využití různých zdrojů balíčků. Ve výchozím nastavení jsou stahovány z veřejného zdroje Packagist. Existuje i projekt Satis, který umožňuje hostovat repozitář na vlastním hardwaru. Jako zdroj balíčků můžou sloužit i git repozitáře. Tyto zdroje se nastavují v composer.json v sekci repositories, kde každý repozitář je samostatným objektem s parametry type a url adresou.

4.2.3 Typovost PHP

PHP začíná v poslední době získávat podporu silné typovosti (Vynucení typu proměnných předávaných funkcím, metodám a atributům objektů). Právě díky typovosti získalo na robustnosti a lze psát aplikace s předvídatelnějším chováním. Verze 5.4 přinesla možnost typovat argumenty funkcí a metod. Ovšem typy byly omezeny pouze na objekty a nebylo možné specifikovat skalární typ (int, float, string, bool). Toto omezení bylo odstraněno ve verzi 7.0, která zároveň přidala podporu pro návratové typy. Verze 7.1 přidala podporu nullable type hintů (možnost vrátit daný typ nebo null). [4]

Příklad kódu s typy:

```
public function getScore(?int $points): Score         {
    return new Score($points);
}
```

Framework Nette je jedním z prvních frameworků, který je plně typovaný. Je vidět, že moderní knihovny a frameworky začínají striktně používat typovost.

Ovšem PHP se ještě nedostalo do stavu, kdy je možné zcela jistě deklarovat typy jen za použití vlastností jazyka a prozatím se musí využívat PHPDoc anotací. Například chybí podpora pro generika, která by umožnila typovat kolekce a pole. Toho teď lze dosáhnout jen nápovědou v PHPDocu.

```
/**
 * @return \App\Collection|\App\Entity\Score[]
 */
public function getScores(): array
{
    return $this->scores;
}
```

Tento nedostatek by se mohl vyřešit podporou generik a předchozí ukázka by mohla vypadat například takto

```
use App\Entity\Score;

public function getScores(): Score[]
{
    return $this->scores;
}
```

nebo takto:

```
use App\Entity\Score;

public function getScores(): Array<Score>
{
    return $this->scores;
}
```

4.3 Přechod od skriptování k webovým aplikacím

4.3.1 Starý přístup k vývoji

Jazyk PHP byl původně vyvíjen jako skriptovací jazyk, kde URL adresy napřímo směřovali na konkrétní skripty. Například požadavek na adresu example.com/about.php webový server obslouží, tak že interpretuje přímo soubor about.php a výsledek vrátí. Takovýto skript většinou zajišťuje vše potřebně od autorizaci přes načítání dat z databáze až po vykreslování šablony. Tento přístup je

velice jednoduchý na pochopení, jelikož lze na jednom místě zjistit, jak je výsledná stránka sestavena. Pokud je takových skriptů více, tak nastává problém s udržitelností a přehledností kódu. Tyto skripty řeší vše od obsahu společného menu, konfiguraci připojení k databázi nebo autorizaci. Pokud by bylo třeba přidat novou položku do společného menu, tak je nutno všechny skripty upravit. Při nízkém počtu skriptů to není problém, ale při aplikaci s několika desítkami až stovkami pohledů to není udržitelné. Možným řešením je funkcionalitu vyčlenit do knihovny společných funkcionalit, ovšem tyto funkce je třeba volat a zajistit, že společné kroky jsou řádně spouštěny v každém skriptu. Další nevýhodou je kombinování aplikační logiky s šablonami.

4.3.2 Front Controller

Front Controller je návrhovým vzorem, jehož hlavním principem je obsluhování všech požadavků jedním vstupním skriptem, který na základě parametrů spouští různé akce (parametrem většinou bývá URL adresa) a umožňuje vykonat společnou funkcionalitu na jednom místě. Například konfigurace celé aplikace se načítá v jednom místě, tudíž není třeba měnit více skriptů při změně. [5] Toho bývá docíleno pomocí směřování všech dotazů na vstupní skript (zpravidla index.php), ve kterém je inicializována aplikace a pomocí směrování je vykonána akce.

4.3.3 MVC (Model-View-Controller)

MVC je architektonickým vzorem, který rozděluje aplikaci na tři hlavní logické komponenty a tím odděluje jednotlivé zodpovědnosti.

Model obsahuje veškerá data a business logiku aplikace (Načítání dat z databáze, úprava a transformace dat, perzistence).

View je reprezentací dat v modelu. V případě webových aplikací představuje HTML šablonu, do které jsou vykreslována data a určuje uživatelské rozhraní aplikace.

Controller je vrstva která obsluhuje uživatelské požadavky tak že získává na základě parametrů data z modelu a vykresluje šablonu.

Například uživatel zažádá o přehled položek v nákupním košíku, Controller přijme požadavek, získá data z modelu, předá je šabloně a vrátí vykreslenou šablonu. [6]

4.3.4 Dependency Injection

Dependency Injection je technika, která umožňuje vytvářet třídy, které jsou na sobě více nezávislé pomocí předávání všech závislostí namísto jejich vytváření. Existuje více způsobů, jakým lze závislosti předat. Například uváděním potřebných závislostí v konstruktoru nebo pomocí setter metod. Pomocí předávání konstruktorem lze docílit objektu, který je vždy validní a lze na jeho chování spoléhat. Pokud je použito předávání settery, tak ne vždy je zaručeno, že objekt bude v konzistentním stavu. Když třída nedefinuje žádné settery a její vnitřní stav lze měnit pouze při inicializaci objektu (v konstruktoru), tak je tzv. immutable třídou.

Pokud by bylo potřeba využívat více variant konstruktorů, tak v jazyce PHP nastává problém. Jazyk samotný neumožňuje přetěžování metod a konstruktorů, ovšem to se dá obejít pomocí definování privátního konstruktoru a několika statických metod s potřebnými argumenty. Oproti definici veřejného konstruktoru s volitelnými argumenty lze definovat potřebné permutace varianty objektu a lépe popsat chování jednotlivých variant. Následující ukázka obsahuje třídu pro rozesílání slevových akcí.

```php
<?php declare(strict_type=1);

use Mailer;
use Model\User;

final class DiscountNewsletter
{

    /**
     * @var array<\Model\User>
     */
    private $users;

    /**
     * @var \Mailer
     */
    private $mailer;

    /**
     * @param array<Users>$users
     */
    public function __construct(array $users, Mailer $mailer)
    {
        $this->users = $users;
        $this->mailer = $mailer;
    }

    public function send()
    {
        foreach ($this->users as $user) {
            $this->mailer->send($user, 'Slevová akce',
                'Nyní je všechno zboží o 15% levnější.'
            );
        }
    }
}
```

Třída využívá Dependency Injection (popisuje všechny závislosti na ostatních třídách v přímo v konstruktoru) a tudíž je na konzumentovi třídy rozhodnout jakou implementaci rozhraní Mailer třídě předá a třída se nemusí starat o to, jak slevovou akci odešle. Například by bylo možné předat třídu, která realizuje odesílání pomocí emailů, push notifikací na mobilních zařízeních, notifikací ve webové aplikaci nebo například SMS zpráv. Tímto je docíleno vysoké znovupoužitelnosti.

Pokud by třída obsahovala následující konstruktor, tak není možné bez zásahu do kódu změnit chování třídy a funkcionalita by byla napevno daná. Tuto třídu by nemělo zajímat, jak bude zpráva o slevě odeslána, ale pouze co bude jejím obsahem.

Bylo by třeba definovat novou třídu pro každý typ notifikace a vznikalo by velké množství duplicitního kódu, který je těžší udržovat a hrozí vyšší riziko vzniku chyb.

```
/**
 * @param array<Users>$users
 */
public function __construct(array $users)
{
    $this->users = $users;
    $this->mailer = new WebNotificationMailer();
}
```

4.3.5 Dependency Injection Container

Dependency Injection Container je objekt, který ví, jak vytvořit a nakonfigurovat jiné objekty. Třídy, které používají jiné objekty by neměli vědět, jak jsou jiné objekty vytvářeny a ani jak jsou konfigurovány. Pokud nějaká třída požaduje například připojení k databázi, tak veřejně deklaruje závislost na této třídě pomocí konstruktoru a DI kontejner buď pomocí autowiringu nebo ze své konfigurace získá třídu reprezentující připojení k databázi a předá ji.

Autowiring je vlastnost DI kontejnerů, pomocí které lze velice zjednodušit konfiguraci kontejneru tím, že není třeba uvádět způsob vytvoření každé závislosti. Toho je docíleno pomocí přečtení typové signatury konstruktoru každé třídy a získání potřebných služeb podle typu. Tímto může vznikat velice složitý strom závislostí, který musí knihovny implementující DIC musí vyřešit.

4.3.6 Šablony

PHP vznikl jako skriptovací jazyk pro webové aplikace a původně byl určen pro vkládání dynamických částí do statických webových stránek, což se hodí pro jednoduché statické stránky s různými doplňkovými funkcionalitami. Tuto funkcionalitu podporuje dodnes, ovšem s použitím MVC návrhu je vhodné oddělit funkcionalitu pro získávání dat, jejich zpracování a zobrazování. Ve větších vývojových týmech je toto žádoucí, protože šablony odstíní front-end kodéra od aplikační logiky a backend vývojáře od způsobu vykreslování HTML. V následující

ukázce je zobrazena velice jednoduchá šablona v čistém HTML, která vypisuje seznam uživatelů.

```
<html>
 <head>
    <title><?php echo $title; ?></title>
 </head>
<body>
    <ul>
    <?php foreach ($users as $user) {?>
        <li><?php echo $user->getName(); ?></li>
    <?php } ?>
    </ul>
</body>
</html>
```

Pro šablonování se používá více různých způsobů, od čistého PHP v samostatných souborech po speciálně vytvořených jazycích, které se kompilují do čistého PHP. Tyto šablonovací systémy umožňují kratší a přehlednější zápis. Následující ukázka zobrazuje stejnou šablonu zapsanou v šablonovacím jazyce Twig.

```
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
<ul>
    {% for user in users %}
        <li>{{ user.name }}</li>
    {% endfor %}
</ul>
</body>
</html>
```

4.4 Databázová vrstva

Většina webových aplikací potřebuje nějakým způsobem uchovávat data a k těm je potřeba přistupovat a ukládat je. Pro tyto účely lze použít PDO (PHP Data Objects), což je abstraktní vrstva poskytovaná samotným PHP pro přístup k databázi. Pro každý druh databáze musí být nainstalovaný PDO driver který ví, jak s danou databází komunikovat. Neobsahuje databázovou abstrakci (stejné SQL dotazy nebo emulace nepodporovaných funkcí) ale pouze jednotné API k vykonávání příkazů. [7]

```
$stmt = $pdo->query('SELECT name FROM users');
while ($row = $stmt->fetch())
{
    echo $row['name'] . "\n";
}
```

Tento kód by vypsal seznam uživatelů z databáze. Nevýhodou tohoto přístupu je repetitivní psaní SQL dotazů pro každou funkcionalitu. Pomocí ORM (Object Relation Mapping) lze s daty v databázi pracovat jako s nativními objekty, které mají své atributy a vazby na další objekty. To umožňuje objekty obohatit o business logiku a různé validace a tím zajistit zapouzdřenější chování.
Existují dva hlavní přístupy k ORM.

4.4.1 Active Record

Active Record je objekt, který obaluje řádek z databáze, přidává metody pro manipulaci s řádkem a definuje vlastní metody s business logikou. Tento objekt obsahuje jak data, tak i metody. Obsahuje přesně ty stejné atributy jako sloupce tabulky, kterou reprezentuje. Active Record je výhodný v aplikacích bez složité business logiky, kdy objekty jsou velice podobné databázové struktuře. Pokud aplikace obsahuje mnoho business pravidel a komplexní logiku a je třeba využít složitých vztahů nebo dědičnosti, tak je vhodnější zvolit Data Mapper. [8] Ukázka vzoru Active Record pomocí knihovny Eloquent ORM

```
class Post extends Model {

    public function comments()
    {
        return $this->hasMany('App\Comment');
    }

}

$post = Post::find(1);
$post->title = 'Článek';
$post::save();
```

4.4.2 Data Mapper

Doménový model aplikace a relační databáze používají různé mechanismy pro strukturování dat. Například relační databáze nepodporují kolekce ani dědičnost. Tyto vlastnosti jsou výhodné budování business logiky aplikace a je výhodné je

používat. Data Mapper je vrstva, která umožňuje odstínit tyto rozdíly a oddělit reprezentaci dat v aplikaci od reprezentace dat v databázi. Modely se skládají z entit představující jednotlivé objekty, ze kterých se systém skládá a tyto entity jsou namapovány na relační databázi. Výhodou je, že entity nejsou nijak spojeny s databází a ani neví, že jsou ukládány a načítány z databáze (případně jakéhokoliv jiného zdroje).

Při práci s těmito entitami je důležité sledovat, jak se mění, vytvářejí a mažou. Pokud by se veškeré změny projevovaly v databázi ihned, tak by se provádělo obrovské množství operací nad databází, což by vedlo k nízkému výkonu. Unit of Work je objekt, který sleduje veškeré změny. Jakmile je potřeba promítnout změny do databáze, tak Unit of Work rozhodne, co vše je potřeba vykonat. Se samotným Unit of Work programátor nikdy nepracuje, pouze může zažádat o promítnutí změn.

Identity Map je strategie pro načítání instancí entit z databáze pouze jednou pomocí mapy, kde klíč je unikátní identifikátor entity a hodnotou samotná entita. Před dotazem na entitu nějakého typu s nějakým identifikátorem se Data Mapper nejdříve podívá do Identity Mapy příslušného typu a zjistí, zda již entitu nedrží v paměti. Tím se předejde problém s nekonzistentním stavem entity a jejich zbytečnému načítání vícekrát.

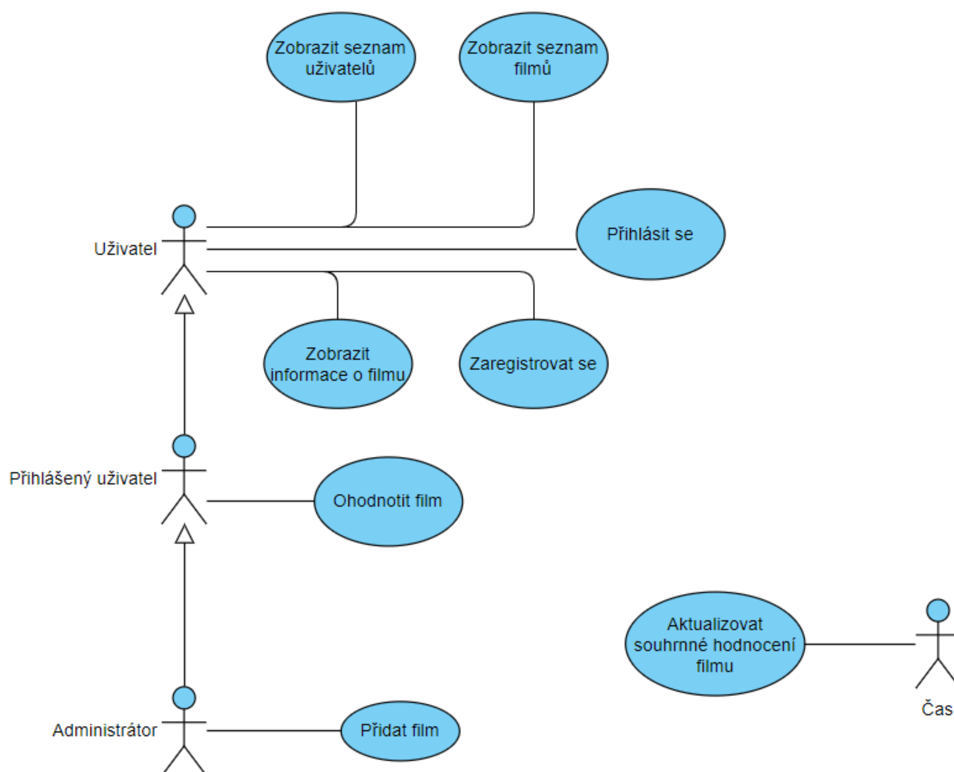
V systému, kde jsou tři jednoduché entity User, Post a Comment, kdy jeden uživatel má více příspěvků, jeden příspěvek má více komentářů a každý komentář má jednoho autora (Uživatele) by docházelo při načtení jakéhokoliv uživatele k načtení téměř celé databáze do paměti kvůli vazbám. Proto se používá tzv. Lazy Loading díky kterému entity nemusí obsahovat kompletní data, ale vědí, jak data získat, když jsou potřeba. [8]

```php
<?php declare(strict_types=1);

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Post
{

    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;
    /**
     * @ORM\Column(type="string")
     */
    protected $content;
    /**
     * @ORM\ManyToOne(targetEntity="user")
     */
    protected $author;

    public function __construct(string $content, User $author)
    {
        $this->content = $content;
        $this->author = $author;
    }

    public function getContent(): string
    {
        return $this->content;
    }

}
```

Ukázka entity namapované na databázi pomocí knihovny Doctrine 2.

5 Praktická část

Používání ukázkové aplikace (nazvané MovieDB) představuje následující USE CASE diagram.



Obrázek 1 Use Case diagram
Zdroj: vlastní tvorba

Aplikace umožňuje registraci, přihlášení pro uživatele, zobrazování a hodnocení filmů pro přihlášené uživatele a přidávání nových filmů pro administrátory. Speciálním případem užití je aktualizace souhrnného hodnocení filmu, která je vykonávána periodicky. Na této akci je demonstrováno vykonávání úloh v aplikacích, které se nezakládají na běžném HTTP request-response cyklu spouštěné uživatelem.

5.1 Instalace frameworků

Všechny frameworky se instalují pomocí Composeru příkazem

```
composer create-project --prefer-dist [název balíčku] [název projektu]
```

Framework	Balíček	Popis
Nette	nette/web-project	Základní varianta Nette frameworku obsahující všechny komponenty. Doporučená varianta.
	nette/sandbox	Podobné jako web-project ovšem obsahuje implementovanou registraci a přihlášení uživatelů.
Symfony	symfony/skeleton	Základní instalace Symfony, která neobsahuje téměř nic a je třeba doinstalovat potřebné komponenty. Hodí se například pro vývoj konzolových aplikací nebo aplikací bez databáze.
	symfony/web-skeleton	Balíček obsahující základní komponenty pro tvorbu webových aplikací (obsahuje například podporu HTTP, databázové vrstvy Doctrine 2, Formulářovou abstrakci a další)
Laravel	laravel/laravel	Jediná výchozí instalace Laravelu

Tabulka 1 Přehled instalačních balíčků
Zdroj: vlastní tvorba

Aplikace je cíleně vytvořena pouze za pomocí balíčků poskytnutých v základní instalace frameworku a balíčcích které jsou zmíněné v oficiální dokumentaci.

Symfony používá k instalaci balíčkovací systém Symfony Flex který je abstrakci nad Composerem. Zjednodušuje instalaci komponent tím, že pomocí tzv. recipes

nakonfiguruje komponentu pro instantní použití. Například vytvoří defaultní konfigurační soubory, vytvoří potřebné složky nebo přidá runtime soubory do souboru .gitignore.

5.2 Běhové prostředí aplikace

Aplikace běží v Docker kontejnerech. Tím je zajištěno stabilního běhového prostředí nezávisle na operačním systému a snadné instalace. Celé prostředí je popsáno v souboru docker-compose.yml.

```
version: '3'
services:
    php:
        build:
            context: ./docker/php
        ports:
            -  80:80
        volumes:
            - .:/var/www/html
            - "./docker/log:/var/log/apache2"
            - ./docker/php/php.ini:/usr/local/etc/php/php.ini
        links:
            - db:db
    db:
        image: mysql:5.6
        ports:
            - 3306:3306
        environment:
            MYSQL_ROOT_PASSWORD: root
            MYSQL_DATABASE: moviedb-symfony
```

Celá aplikace se skládá ze dvou propojených kontejnerů. Prvním je webový server Apache s PHP a druhým je databáze MySQL.

5.3 Instalace aplikace

Pro spuštění aplikace je třeba obstarat zdrojové kódy např. naklonováním z veřejného git repozitáře, spuštěním Docker kontejnerů, instalací závislostí aplikace a vytvořením schématu databáze.

Příklad instalace MovieDB aplikace v Symfony:

```
git clone https://github.com/martingold/moviedb-symfony.git \
&& cd moviedb-symfony \
&& docker-compose up-d \
&& docker-compose exec php composer install
&& docker-compose exec php d:s:u -f
```

Tento postup je stejný pro všechny varianty aplikace až na poslední krok vytvoření schématu databáze. Pro každou variantu aplikace je přesný postup instalace uveden v souboru readme.md.

5.4 Frontendové závislosti

Většina webových aplikací potřebuje ke svému správnému fungování nějaké CSS a Javascriptové knihovny. Aplikace MovieDB ke svému fungování potřebuje CSS framework Tailwind. Ten je možné v aplikaci používat pomocí vložení přímo samotného frameworku v hlavičce z lokálního souboru nebo z CDN (Content distribution network) následujícím způsobem:

```
<link href="/tailwind.min.css" rel="stylesheet">
```

Toto řešení má nevýhodu, že nelze nijak framework upravovat a pokud by se používalo více knihoven, tak by po načtení stránky probíhalo příliš mnoho požadavků a webová aplikace by se tím zpomalila. Pomocí balíčkovacího systému jako je Webpack lze veškeré závislosti zabalit do jednoho či více souborů.

PHP frameworky s frontend závislostmi přímo nesouvisí, jelikož server vrací HTML odpověď nebo data v různých formátech a tím práce frameworku končí, ovšem Symfony a Laravel obsahují své nástroje pro balíčkování Javascriptu a CSS založené na Webpacku. Tím umožňují rapidně zrychlit vývoj a pokrývají běžné případy použití. Jedná se o velice elegantní způsob zpracování frontendových závislostí a snižuje vstupní bariéru díky odstranění nutnosti konfigurovat Webpack.

5.4.1 Laravel – Mix

Laravel přišel jako první s tímto konceptem a nazval jej Laravel Mix. Jeho konfigurace vypadá následovně:

```
const mix = require('laravel-mix');
const tailwindcss = require('tailwindcss');

mix.js('resources/js/app.js', 'public/js')

mix.sass('resources/sass/app.scss', 'public/css')
    .options({
        processCssUrls: false,
        postCss: [ tailwindcss('./tailwind.config.js') ],
    })
```

Oproti Symfony Encore neposkytuje výrazné výhody.

5.4.2 Symfony – Encore

Symfony Flex vzniknul jako reakce na Laravel Mix a nepřináší zásadnější funkce. Jeho konfigurace vypadá následovně.

```
var Encore = require('@symfony/webpack-encore');

if (!Encore.isRuntimeEnvironmentConfigured()) {
    Encore.configureRuntimeEnvironment(process.env.NODE_ENV || 'dev');
}

Encore
    .setOutputPath('public/build/')
    .setPublicPath('/build')
    .addEntry('app', './assets/js/app.js')
    .splitEntryChunks()
    .enableSingleRuntimeChunk()
    .cleanupOutputBeforeBuild()
    .enableBuildNotifications()
    .enableSourceMaps(!Encore.isProduction())
    .enableVersioning(Encore.isProduction())

    .configureBabelPresetEnv((config) => {
        config.useBuiltIns = 'usage';
        config.corejs = 3;
    })
    .enablePostCssLoader();

module.exports = Encore.getWebpackConfig();
```

5.4.3 Nette

Nette žádný doporučený způsob nebo nástroj pro balíčkování frontend závislostí nemá ani nedoporučuje, proto byl v aplikaci použit způsob vložení Tailwind frameworku v hlavičce z CDN.

5.5 Směrování

5.5.1 Symfony

V Symfony je potřeba směrování doinstalovat pomocí Symfony Flex, pokud nebyl použit web-skeleton pomocí příkazu `composer require annotations`.

Symfony nabízí více možnosti pro definování cest pomocí YAML, XML, PHP nebo anotací. V aplikaci se používají anotace, protože se jedná o výchozí způsob v dokumentaci.

Ostatní zápisy mají ekvivalentní funkcionalitu, pouze se používá oddělený soubor s jinou syntaxí. Záleží na preferenci a domluvě týmu, který ze způsobů je vhodnější. Definice cesty v anotacích je přehlednější a je přímo u akce controlleru, ovšem cesty definované ve zvláštním souboru nabízí lepší celkový přehled o všech cestách v aplikaci. Symfony má možnost příkaz pro vypsání všechny cest pomocí příkazu a tím získat celkový přehled.

```
docker-compose exec php bin/console router:debug
```

Například cesta pro ohodnocení filmu definovaná pomocí anotací vypadá následovně:

```
/**
 * @Route(path="/film/{slug}/ohodnotit", name="movie_comment")
 */
public function comment(Movie $movie, Request $request)
{
      ...
}
```

Parametr name není povinný, ovšem je velice praktické mít všechny cesty pojmenované, protože beze jména nelze generovat odkazy. Toto jméno musí být v rámci celé aplikace unikátní. U cesty lze ještě dále uvést např. HTTP metody, které má cesta podporovat pomocí parametru `methods={"GET","POST"...}`. Když není parametr uveden tak cesta reaguje na všechny metody.

Velice zajímavý je způsob, jakým je předán parametr $movie do akce Controlleru. Framework se pomocí reflexe podívá na definici metody pro akci, zjistí si které všechny parametry jsou potřebné a jakého jsou typu. Pokud jsou známého typu tak se nijak neřeší, ovšem pokud argument není jasný, tak se jej pokusí pomocí tzv. ParamConverter převést a vyřešit. Například pro entity je použit DoctrineConverter, který načte a předá entitu dle typu argumentu v definici akce a názvu proměnné v definici cesty. (V tomto případě by pro url „/film/pulp-fiction/ohodnotit“ našel entitu typu App\Entity\Movie kde atribut slug je roven „pulp-fiction“). Jedná se o velice pohodlný způsob práce s parametry akcí.

Symfony má ze všech tří frameworků nejvariabilnější směrování a nabízí nejvíce možností konfigurace. Ovšem v aplikaci se používá pouze jeden způsob (pomocí anotací) a je velice praktické vidět definici cest přímo u akce.

5.5.2 Laravel

Cesty pro web jsou ve výchozí instalaci definovány v souboru routes/web.php. Ty jsou definovány pomocí voláním metod třídy Illuminate\Support\Facades\Route. I v Laravelu je třeba mít cesty pojmenované, aby se daly generovat odkazy. Definice cesty pro hodnocení filmu vypadá následovně:

```php
<?php

use Illuminate\Support\Facades\Route;

...
Route::get('film/{slug}/ohodnotit', 'MovieController@rate')
    ->name('movie_comment');
Route::post('film/{slug}/ohodnotit', 'MovieController@handleRate')
    ->name('movie_comment');
...
```

Cesta je zde definovaná dvakrát, jednou pro požadavek typu GET při vykreslení formuláře hodnocení a podruhé typu POST při zpracování hodnot z formuláře. Pokaždé je ovšem odkazována jiná akce.

V definici cest lze použít metodu pro vytvoření cesty obsluhující více typů HTTP pomocí metody Route::match(), která jako první argument přijímá pole podporovaných metod (např. `['get', 'post']`). Ještě existuje metoda Route::any() která akceptuje všechny HTTP metody.

Namísto řetězce pro definici akce controlleru lze použít anonymní funkci, která vrací požadovaný výsledek, což se může hodit např. pro vracení statických JSON odpovědí.

Pokud cesta pouze vrací statickou šablonu, pak lze využít metodu Router::view(), která přijímá jako první argument definici cesty a název šablony jako druhý argument.

Laravel v základní instalaci nepodporuje konvertování argumentů na objekty a je třeba si argumenty v akcích zpracovávat ručně.

Směrování v Laravelu je velice přehledné a svůj účel plní velice dobře. Oproti Symfony nenabízí více variant zápisu.

5.5.3 Nette

Směrování je doporučeno definovat ve službě RouterFactory, která je registrována v konfiguraci v app/config/common.local pomocí:

```
services:
	router: App\Router\RouterFactory::createRouter
```

Tato metoda vrací objekt typu `Nette\Application\Routers\RouteList`. Celá třída RouterFactory v aplikaci MovieDB vypadá následovně:

```php
<?php declare(strict_types=1);

namespace App\Router;

use Nette;
use Nette\Application\Routers\RouteList;


final class RouterFactory
{
	use Nette\StaticClass;

	public static function createRouter(): RouteList
	{
		$router = new RouteList();
		$router->addRoute(
			'film/<slug>/ohodnotit',
			'Movie:comment'
		);
		...

		return $router;
	}
}
```

Pokud by nebylo třeba definovat tvar URL adres, lze použít obecnou cestu:

```php
$router->addRoute('<presenter>/<action>', 'Homepage:default');
```

Akce hodnocení filmu by při použití této obecné cesty měla URL adresu movie/comment/pulp-fiction.

Nette umí převádět parametry na úrovni směrování, ovšem ne tak elegantně jako Symfony. Na rozdíl od Symfony a Laravelu neumožňuje odlišovat různé HTTP metody. Velice se hodí možnost definovat obecnou cestu, pokud pro akce není důležitý tvar URL adresy. Oproti Symfony nenabízí více variant zápisu.

	Přehlednost	Možnosti
Symfony	Velice přehledné a praktické při použití anotací v controllerech.	Nabízí nejpokročilejší možnosti především díky vice způsobům zápisu a převádění parametrů
Laravel	Zápis je velice čitelný a praktický. Po získání zkušeností lze definice různě kombinovat a zapisovat je velice přehledně.	Podporuje veškeré potřebné vlastnosti a je velice praktické možnost vrátit vykreslenou šablonu přímo z definice cesty
Nette	Při definování složitějších cest se může zápis stát nepřehledným.	Velice praktickou vlastností je definice obecné cesty, která je vhodná pro akce, které nemusí mít specifický tvar URL adresy.

Tabulka 2 Porovnání směrování
Zdroj: vlastní tvorba

5.6 Šablony

Každý framework používá svůj šablonovací systém a všechny jsou si dost podobné. Všechny podporují automatické ošetření vypisovaných proměnných (zabránění útoku typu XSS), znovupoužitelné bloky, kompilují se do čistého PHP a liší se pouze syntaxí a různými drobnými vlastnostmi, které ulehčují použití a čitelnost.

5.6.1 Symfony – šablonovací jazyk Twig

Syntaxe se zakládá na tzv. mustache šablonách, kde dvojitá složená závorka znamená výpis a složená závorka následovaná procentem značí řídící tagy jako foreach nebo if které řídí logiku šablony. Ve Twigu není možné používat čisté PHP a všechny použitá data se musí předat do šablony pomocí parametrů. K atributům a metodám objektů se přistupuje pomocí tečkové notace.

Šablona pro výpis uživatelů ve Twigu:

```
{% extends 'base.html.twig' %}

{% block title %}Přehled uživatelů{% endblock %}

{% block body %}
<h1 class="text-2xl mt-4 font-semibold">
    Uživatelé podle počtu hodnocení
</h1>
<div class="bg-white mt-4 border rounded bg-white">
    {% for user in users %}
    <div
        class="{{cycle(['bg-gray-100', ''], loop.index0)}} p-2 flex">
        <span class="flex-grow">
            {{ user.email }}
        </span>
        <span>
            {{ user.ratings.count }} hodnocení
        </span>
    </div>
    {% endfor %}
</div>
{% endblock %}
```

Největší předností Twigu je možnost použít tzv. sandboxový režim. Ten umožňuje bezpečně vkládat šablony, které mohou pocházet z cizích zdrojů. Tato funkcionalita se velice hodí, pokud by bylo třeba mít možnost nechat uživatele upravovat vzhled a strukturu stránky (Například úprava struktury přehledu produktu v e-shopu).

Twig umožňuje používání filtrů pro snazší výpis proměnných. Například výpis zaokrouhleného čísla lze zapsat takto `{{ $number|round }}`. Přehled všech filtrů lze najít v dokumentaci.

Twig je velice schopný šablonovací systém se všemi důležitými funkcemi. Velikou výhodou může být sandboxovaný režim. Drobným nezvykem je zápis foreach cyklu oproti běžnému PHP, který má prohozené argumenty a používá klíčové slovo in namísto as (`for (user in users)` oproti `foreach ($users as $user)`).

5.6.2 Laravel – šablonovací jazyk Blade

Řídící tagy se zapisují pomocí zavináčové notace. Přístup k proměnným a metodám je stejný jako v PHP. V Blade je možné používat čisté PHP, což dává vývojářům

volnější ruku. Blade filtry nepodporuje, ovšem je možné použít některou z knihoven, které tuto funkcionalitu přidávají.

```
@extends('layouts.master')

@section('title', 'Přehled uživatelů')

@section('content')

    <h1 class="text-2xl mt-4 font-semibold">
        Uživatelé podle počtu hodnocení
    </h1>

    <div class="bg-white mt-4 border rounded bg-white">
        @foreach ($users as $user)
            <div class="p-2 flex
               {{$loop->iteration % 2 === 0 ? ' bg-gray-100' : ''}}
               ">
                <span class="flex-grow">
                    {{ $user->email }}
                </span>
                <span>
                    {{ $user->relations->count() }} hodnocení
                </span>
            </div>
        @endforeach
    </div>
@stop
```

Blade nemá zásadní nedostatky, kromě neexistence užitečných filtrů, které zkracují a zpřehledňují zápis.

5.6.3 Nette – šablonovací jazyk Latte

Syntaxe se velice neliší, ovšem má velkou výhodu v tom, že neodlišuje značky pro výpis a pro řídící struktury. Pro začínající vývojáře to může být drobná výhoda. Latte podporuje jak filtry, tak tzv. n:makra která velice zpřehledňují zápis a platí pro ně, že zápis:

```
{foreach $users as $user}
    <div>{$user->name}<div>
{/foreach}
```

je ekvivalentní se zápisem:

```
<div n:foreach="$users as $user">
    {$user->name}
<div>
```

Ukázka výpisu uživatelů pomocí Latte:

```
{block title}Přehled uživatelů dle hodnocení{/block}

{block content}

<h1 class="text-2xl mt-4 font-semibold">
    Uživatelé podle počtu hodnocení
</h1>
<div class="bg-white mt-4 border rounded bg-white">
    <div
        n:foreach="$users as $userRow"
        class="p-2 flex{if $iterator->even} bg-gray-100{/if}
    ">
        <span class="flex-grow">
            {$userRow->name}
        </span>
        <span>
            {$userRow->count} hodnocení
        </span>
    </div>
</div>
```

Za povšimnutí stojí nepovinná koncová značka `{/block}`. Pro použití čistého PHP lze použít tag {php …}, který vyhodnotí výraz ovšem nevypíše jej.

Latte je ze šablonovacích systému nejpokročilejší a nabízí nejvíce možností, jak zpřehlednit a zkrátit zápis především pomocí n:maker.

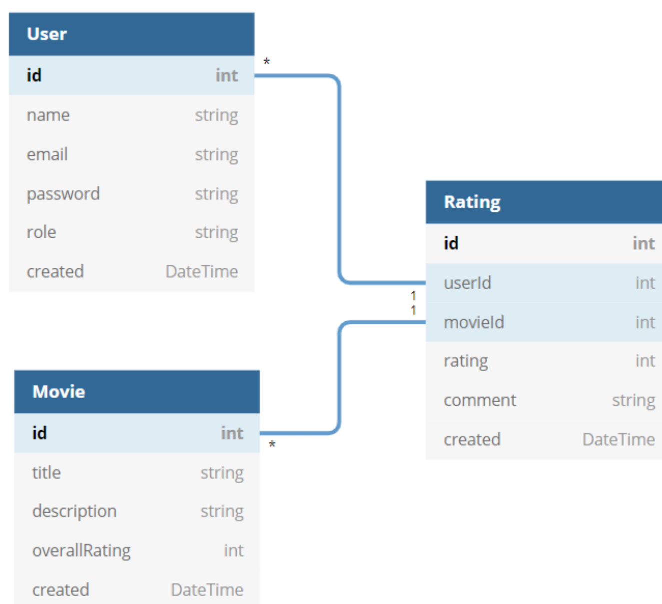
	Přehlednost	Možnosti
Symfony	Zápis je velice čitelný, ovšem rozdílná syntaxe pro výpis proměnných a řídící tagy může být matoucí. Při přechodu mezi čistým PHP a Twigem je matoucí opačné pořadí ve foreach cyklu.	Twig nabízí sandboxový režim který rozšiřuje uplatnění toho šablonovacího jazyka. Poskytuje vše potřebné.
Laravel	Oproti ostatním šablonovacím jazykům se v přehlednosti nijak neliší a oproti ostatním nepoužívá řídící tagy obalené z obou stran.	Nabízí nejmenší možnosti. Neposkytuje podporu filtrů, sandboxového režimu a ani maker.
Nette	Zápis je nepřehlednější, jelikož se nejvíce podobá čistému PHP a n:makra velice zpřehledňují zápis.	Funkcionalitou je Latte někde mezi, poskytuje vše potřebné včetně filtrů, ovšem chybí sandboxový režim.

Tabulka 3 Porovnání šablonovacích jazyků
Zdroj: vlastní tvorba

5.7 Databázová vrstva

Frameworky se velice liší v použitých databázových vrstvách. Symfony používá návrhový vzor Data Mapper ve své knihovně Doctrine2. Eloquent v Laravelu používá vzor Active Record a Nette Database Explorer je Query Builder, který umožňuje přistupovat k relacím. Všechny frameworky myslí na bezpečnost a automaticky ošetřují parametry dotazu, tudíž dokud jsou dotazy skládány doporučovaným způsobem, tak nemůže nastat útok typu SQL Injection. V hodnocení není použito přímé srovnání, protože každý framework používá zcela odlišný přístup.

Celá databázová struktura aplikace je popsána tímto diagramem:



Obrázek 2 Struktura databáze
Zdroj: vlastní tvorba

5.7.1 Symfony – Doctrine

Symfony aplikace používá tři entity v namespace App\Entity a to Movie, Rating a User. Tyto třídy jsou obyčejné PHP třídy, které nemusí dědit od žádného předka nebo implementovat žádný interface a z hlediska OOP nemají povědomí o tom, že by byly načítány a ukládány do databáze. Tak jako u směrování, Symfony podporuje více možných zápisů mapování entity (Anotace, XML, Yaml a PHP kód). V aplikaci se používá výhradně mapování pomocí anotací.

Každá třída má uvedenou anotaci entity, které říká, že se jedná o entitu spravovanou Doctrine.

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface
{
...
}
```

Zároveň je u této anotace uveden parametr repositoryClass určující, které třída se má používat pro přístup k entitám. Například třída UserRepository má jednu metodu navíc oproti defaultní implementaci repozitáře, ze které všechny repozitáře v aplikaci dědí:

```
/**
 * @return array<\App\Entity\User>
 */
public function getByRatingsCount(): array
{
        return $this->getEntityManager()->getRepository(User::class)
            ->createQueryBuilder('u')
            ->select(['u', 'r'])
            ->leftJoin('u.ratings', 'r')
            ->orderBy('COUNT(r)', 'DESC')
            ->getQuery()
            ->getResult();
}
```

Tato metoda vrací pole uživatelů s alespoň jedním hodnocením, seřazených podle počty udělených hodnocení. Výhodou repozitářů je možnost znovupoužití dotazů na různých místech aplikace.

Největší výhodou entit je, že jsou zcela zapouzdřeny a můžou si vynutit určité chování. Pokud je uveden konstruktor s parametry, tak v aplikaci nelze vytvořit instanci uživatele (bez použití reflexe nebo jiných vlastností PHP které k tomu nejsou určeny), který by nabýval nekonzistentního stavu.

```
public function __construct(Movie $movie, User $user)
{
    $this->movie = $movie;
    $this->user = $user;
    $this->created = new DateTime();
}
```

Hodnocení filmu není možné vytvořit bez vazby na uživatele a film a zároveň je při vytvoření zaznamenán čas. V komplexnějších aplikacích se velice hodí mít možnost takto definovat různé chování. Stejné principy lze uplatnit u setterů.

```
public function setOverallRating(?float $overallRating): void
  {
      if ($overallRating === null) {
          $this->overallRating = null;
          return;
      }

      if ($overallRating >= 0.0 && $overallRating <= 100.0) {
          $this->overallRating = (string) $overallRating;
          return;
      }

      throw new Exception(sprintf(
          'Rating must be in (0, 100) range \'%s\' provided.',
          $overallRating
      ));
  }
```

Při nastavování nekorektního průměrného hodnocení filmu aplikace vyhodí výjimku informující o nesprávném použití metody. Takto nemůže v aplikaci nastat nekonzistentní stav průměrného hodnocení. Pokud data podléhají nějakému chování, tak lze definovat metodu pro operaci s těmito daty. V aplikaci je použita metoda pro přepočet procent na počet hvězd, která je na jednom místě a pokud by se někdy v budoucnu měla měnit logika výpočtu, tak ji lze snadno na jednom místě upravit bez vedlejších efektů.

Atributy jsou mapovány na databázové sloupce pomocí @Column anotace. Zde je povinný pouze parametr type, který říká, jakého je sloupec typu. Z následující ukázky je patrné, že je zde typ uveden dvakrát (v anotaci a nativní typehint), ovšem ne vždy datový typ v PHP odpovídá přesně databázovému typu.

```
/**
 * @ORM\Column(type="text", nullable=false)
 */
private string $description = '';
```

Za zmínku stojí parametr nullable, který určuje, zda mapovaný sloupec může nabývat hodnoty null. Parametr name udává název sloupce v databázi, ovšem ne vždy je ho nutné uvádět, protože Doctrine si dokáže název odvodit pomocí výchozí konvence převodu názvu atributu z camelCase do snake_case („overallRating“ na „overal_rating“).

Vztahy se určují podobně anotací u atributů, ovšem používá se anotace závislá na použitém vztahu mezi entitami. U vztahu mezi Filmem a Hodnocením existuje obousměrný One to Many vztah, který je definovaný následovně v entitě Movie

```
/**
 * @var Collection<\App\Entity\Rating>
 * @ORM\OneToMany(targetEntity="Rating", mappedBy="movie")
 */
private Collection $ratings;
```

a takto v entitě Rating

```
/**
 * @ORM\ManyToOne(targetEntity="Movie", inversedBy="ratings")
 */
private Movie $movie;
```

Doctrine je z hlediska architektury nejčistší, jelikož odděluje zodpovědnost do více tříd (separation of concerns) a nemíchá persistenci s daty. Ovšem kód může být celkem dlouhý.

5.7.2 Laravel – Eloquent

Eloquent staví na vzoru Active Record a v aplikaci jsou použity tři třídy User, Rating a Movie v namespace \App. Tyto modely musí dědit ze třídy Illuminate\Database\Eloquent\Model. Tato třída obstarává vše od připojení k databázi, přes perzistenci až po vytváření dotazů.

Dotazování má velice elegantní syntaxi. Například získání všech filmů seřazených dle hodnocení lze provést pomocí vykonáním následujícího:

```
$movies = Movie::query()->orderBy('overall_rating')->get();
```

Díky užití magické metody __callStatic() v rodičovské třídě modelu lze zápis zkrátit (Magické metody jsou v PHP metody, které mají předurčené chování a volají se v různých případech. Konkrétně __callStatic se volá, pokud se nad objektem volá statická metoda, která neexistuje, tudíž umožňuje definovat dynamické chování)

```
$movies = Movie->orderBy('overall_rating')->get();
```

Toto chování je vhodné pro čitelnost, ovšem statická metoda orderBy ve třídě Movie ani rodičovské třídě Model není definována, tudíž tento přístup není až tak transparentní a nefunguje napovídání metod v IDE.

V modelové třídě nejsou definovány jednotlivé atributy, ale jsou uloženy v poli $attributes, ke kterým se přistupuje běžně jako k public atributům, ovšem je při tom využíváno magických metod __get a __set (magické metody volané při přístupu a nastavování hodnot neexistujících atributů třídy). To má nevýhodu, že na první pohled do třídy modelu není jasné, co vše obsahuje a IDE opět neumí napovídat. To lze spravit používáním PHPDoc anotací @property u třídy.

```
/**
 * @property string $title
 * @property string $description
 * ...
 */
class Movie extends Model
{
    ...
}
```

Vztahy mezi modely jsou definovány pomocí metod. Metoda vztahu mezi hodnocením a filmem vypadá metoda takto v modelu Movie:

```
public function ratings()
{
    return $this->hasMany(Rating::class);
}
```

a následovně v modelu Rating:

```
public function movie()
{
    return $this->belongsTo('App\Movie');
}
```

V modelu lze upravit chování získávání atributu (funkcionalita jako getter v OOP), tak že se definuje metoda, která využívá magických metod a její návratová hodnota se vrací při přístupu atributu modelu.

```
public function getTitleAttribute(string $value): string
{
    return strtuopper($value);
}
...

$title = $movie->title; // Zavolá výše definovanou metodu
```

Eloquent je velice jednoduchý na používání, ovšem samotné modelové třídy mají až příliš zodpovědnosti a používáním magických metod občas může docházet k nesnadno laditelným problémům. Používání je jinak velice intuitivní a snadné.

5.7.3 Nette – Database Explorer

Database Explorer nepoužívá žádné modelové třídy, namísto toho vrací instance třídy Selection a jednotlivé řádky jsou instance třídy ActiveRow.

Pro dotazování nad daty se používá třída Nette\Database\Context, které je query builderem.

Příklad dotazu v repozitáři pro dotahování filmů seřazených podle hodnocení:

```
/**
 * @return array<mixed>
 */
public function bestMovies(): array
{
    return $this->database->table('movie')
        ->order('overallRating DESC')
        ->limit(100)
        ->fetchAll();
}
```

Lze dotahovat i vztahy mezi tabulkami pomocí metod ref() a related(). Například pro načtení hodnocení komentářů k filmu je v šabloně realizováno následovně:

```
{foreach $movie->related('rating') as $rating}
    ...
{/foreach}
```

Pokud sloupce dodržují jmennou konvenci, tak není třeba uvádět názvy sloupců které vztah určují. Database Explorer není databázová vrstva, spíše se jedná o knihovnu usnadňující skládání dotazů. Dotahování vztahů mezi tabulkami je příjemná vlastnost. Pro tuto úlohu se spíše hodí databázové vrstvy z Laravelu a Symfony

	Přehlednost	Možnosti
Symfony	Využití nativních PHP tříd je velice přehledné a srozumitelné. Ovšem pro začátečníky může být celý koncept mapování databáze na třídy matoucí.	Oddělení modelové vrstvy od návrhu databáze je rozhodně nejpokročilejší řešení a nabízí nejvyšší míru abstrakce od databáze. Ve velice rozsáhlých aplikací je to velkou výhodou.
Laravel	Skládání dotazů je velice praktické a přehledné. Samotný zápis entit není až tak přehledný a například u relací není možné	Nabízí vše, co je třeba pro tvorbu aplikace a rozhodně vlastnostmi nezaostává, ovšem při návrhu, který přesně neodpovídá struktuře tabulek můžou nastat problémy.

	bez znalosti a hlubšího zkoumání snadno zjistit chyby.	
Nette	Skládání dotazů je jednoduché a přehledné. Některé dotazy lze zapsat různými způsoby a dokumentace je velice stručná.	Nenabízí takové možnosti a použití je značně omezené.

Tabulka 4 Porovnání databázových vrstev
Zdroj: vlastní tvorba

5.8 Konzole

Moderní aplikace ne vždy vykonávají akce pomocí HTTP rozhraní a například různé periodické úlohy nebo systémové úlohy je dobré definovat pomocí CLI (příkazové řádky). V aplikaci MovieDB je jedna úloha pro přepočítávání průměrného hodnocení filmu, která není náročná na systémové prostředky a pokud by se hodnocení počítalo za běhu aplikace, tak by nenastalo velké zpomalení. Slouží spíše za účelem ukázky konzolového rozhraní aplikace.

5.8.1 Symfony

V základní instalaci symfony/skeleton je třeba konzoli doinstalovat pomocí příkazu composer require symfony/console. Konzole umí vypsat všechny možné příkazy se stručným popisem. V aplikaci by se mohl spustit například takto:

```
docker-compose exec php bin/console
```

Každý balíček a aplikace samotná může definovat své příkazy a existují příkazy od vypsání všech dostupných cest. přes generování controlleru, entit či šablon, až po automatické vytváření migrací. Příkaz pro přepočet hodnocení vypadá následovně:

```php
<?php declare(strict_types=1);

namespace App\Command;

use App\Repository\MovieRepository;
...

class OverallRatingCommand extends Command
{

    protected static $defaultName = 'movie:overall-rating';

    private MovieRepository $movieRepository;
    private OverallRatingCalculator $overallRatingCalculator;

    public function __construct(
        MovieRepository $movieRepository,
        OverallRatingCalculator $overallRatingCalculator)
    {
        parent::__construct();
        $this->movieRepository = $movieRepository;
        $this->overallRatingCalculator = $overallRatingCalculator;
    }

    protected function configure()
    {
        $this->setDescription(
            'Update average rating for Movie entity'
        );
    }

    protected function execute(
        InputInterface $input,
        OutputInterface $output)
    {
        $movies =  $this->movieRepository->findAll();
        foreach ($movies as $movie) {
            $movie->setOverallRating(
                $this
                    ->overallRatingCalculator
                    ->calculateOverallRating($movie)
            );
            $this->movieRepository->update($movie);
        }
        return 0;
    }
}
```

Symfony console je komponenta, která velice usnadňuje vytváření CLI příkazů a oproti Laravelu není zásadně jiná.

5.8.2 Laravel

Konzole Laravelu staví na Symfony konzoli, tudíž funkcionalita je velice podobná. Popis a konfigurace příkazu se nedefinuje v metodě configure, ale využívá se protected atributů třídy. Spouštěcí soubor je přímo v root složce aplikace a jmenuje se artisan.

```
<?php

namespace App\Console\Commands;

use App\Movie;

class CalculateOverallRating extends Command
{

    /**
     * @var string
     */
    protected $signature = 'movie:overall-rating';

    /**
     * @var string
     */
    protected $description = 'Calculate movie overall rating';

    /**
     * @var \App\OverallRatingCalculator
     */
    private OverallRatingCalculator $overallRatingCalculator;

    public function __construct(
        OverallRatingCalculator $overallRatingCalculator)
    {
        parent::__construct();
        $this->overallRatingCalculator = $overallRatingCalculator;
    }

    public function handle()
    {
        $movies = Movie::with('ratings')->get();
        foreach ($movies as $movie) {
            $rating = $this
               ->overallRatingCalculator
               ->calculateOverallRating($movie);
            $movie->overall_rating = $rating;
            $movie->save();
        }
        return 0;
    }
}
```

Jelikož konzole Laravelu používá ke konzoli ze Symfony, tak až na drobné rozdíly poskytuje stejnou funkcionalitu.

5.8.3 Nette

Nette samotné nepodporuje žádnou konzoli ani nedoporučuje způsob pro přístup k aplikaci pomocí CLI. V aplikaci MovieDB je přepočet hodnocení řešen pomocí PHP skriptu, který si vytvoří Dependency Injection kontejner, který poskytne instanci služby OverallRatingCalculator. Skript zavolá metodu updateOverallRating() této služby.

```
#!/usr/bin/env php
<?php

declare(strict_types=1);

use App\Service\Movie\OverallRatingCalculator;

require __DIR__ . '/../vendor/autoload.php';

$container = App\Bootstrap::boot()
        ->createContainer();

$overallRatingCalulator = $container
        ->getByType(OverallRatingCalculator::class);

$overallRatingCalulator->updateOverallRating();
```

	Přehlednost	Možnosti
Symfony	Definice příkazu pomocí třídy je velice přehledné a snadno pochopitelné. Rodičovská Command třída má pěkně pojmenované metody a lze příkazy vytvářet i bez dlouhého zkoumání dokumentace.	Díky příkazům je možné vykonávat akce i bez http prostředí (Symfony lze použít i jako čistě konzolový framework) a u rozsáhlejších aplikací je to velice přínosné
Laravel	Díky tomu že Laravel využívá konzoli ze Symfony s drobnými úpravami, tak jsou možnosti a přehlednost téměř totožné.	

Nette	Jelikož Nette nenabízí žádné možnosti konzole a vše je třeba ručně implementovat, tak není srovnáváno s ostatními frameworky.

Tabulka 5 Porovnání konzolí
Zdroj: vlastní tvorba

5.9 Databázové migrace

Pro správnou funkčnost aplikace je třeba držet schéma databáze synchronizované s kódem. Toho je docíleno pomocí migračních skriptů, které se vykonávají pokaždé při instalaci nové verze aplikace. Fungují tak, že se v databázi v oddělené tabulce evidují již vykonané migrace a při spouštění příkazu pro migraci se spustí jen ty skripty, které nejsou ve zmíněné tabulce evidovány. Migrace jsou předvedeny na přidání telefonního čísla k uživateli.

5.9.1 Symfony

Doctrine podporuje automatické generování migrací pomocí porovnání definic entit a aktuálního schématu databáze. Přidání nové telefonního čísla pro entitu Uživatele by probíhalo následovně:

1. Spuštění příkazu ./bin/console doctrine:schema-tool:update –dump-sql pro kontrolu, zda je Doctrine správně rozpozná změny v databázi
2. Spuštění příkazu ./bin/console migrations:diff pro vygenerování migrace
3. Spuštění příkazu ./bin/console migrations:migrate pro provedení změn

Tento postup nevyžaduje od programátora přemýšlet nad databází a většina změn se provádějí v definici entit. Samozřejmě pokud by bylo třeba měnit samotná data, tak je třeba si vygenerovat prázdnou migraci pomocí migrations:create a vytvořit potřebné SQL příkazy.

Symfony umožnuje velice pohodlné vytváření migrací, což je oproti Laravelu obrovská výhoda. Ovšem pokud je třeba vytvářet migraci, která není vygenerovaná, tak je Laravel přehlednější.

5.9.2 Laravel

V Laravelu je třeba vytvářet migrace ručně pomocí příkazu

```
./artisan make:migration user_add_phone
```

Tím se vytvoří prázdná migrace s názvem user_add_phone a je třeba v metodě up() a down() vykonat změny:

```
    public function up()
    {
        Schema::table('user', function (Blueprint $table) {
            $table->string('phone');
        });
    }

    public function down()
    {
        Schema::table('user', function (Blueprint $table) {
            $table->dropColumn('phone');
        });
    }
```

Zápis migrací je velice intuitivní a praktický. Jelikož Eloquent kódu odvodit potřebné sloupce, tak ani nelze migrace generovat. Symfony v tomto případě má velikou výhodu.

5.9.3 Nette

Nette samotné žádný svůj nástroj nebo doporučený postup pro verzování databáze nemá.

	Přehlednost	Možnosti
Symfony	Migrace využívají čistého SQL oproti Laravelu můžou být méně přehledné.	Velice praktickou a užitečnou vlastností je generování migrací automaticky z entity.
Laravel	Migrace se zapisují velice lehce a snadno. Výsledná migrace je čitelná a jasná.	Kromě funkcionality generování migrací umí úplně to stejné s hezčím zápisem
Nette	Nette nástroj pro databázové migrace nepodporuje, proto není hodnocen.	

Tabulka 6 Porovnání migrací
Zdroj: vlastní tvorba

5.10 Controllery

5.10.1 Symfony

Controllerem může být jakákoliv funkce nebo metoda třídy. Je doporučováno používat třídy s metodami reprezentující jednotlivé akce. Tato třída nemusí dědit od společného předka, ovšem dokumentace Symfony doporučuje dědit od třídy Symfony\Bundle\FrameworkBundle\Controller\AbstractController, která zjednodušuje přesměrování nebo vykreslování šablon. Příklad controlleru pro akce spojené s uživatelem:

```
namespace App\Controller;

use App\Repository\UserRepository;
...

final class UserController extends AbstractController
{

    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    /**
     * @Route(path="/uzivatele", name="user_list")
     */
    public function list(): Response
    {
        return $this->render('User/list.html.twig', [
            'users' => $this->userRepository->getByRatingsCount()
        ]);
    }
}
```

Controllery oproti ostatním frameworkům nemají zásadnější rozdíl a používají se podobně.

5.10.2 Laravel

V Laravelu vycházejí Controllery z rodičovské třídy Illuminate\Routing\Controller. Jsou to třídy s obyčejnými metodami, které jsou odkazovány v definicích cest.

Ukázka controlleru s akcí pro výpis uživatelů:

```
<?php declare(strict_types=1);

namespace App\Http\Controllers;

use App\User;

class UserController extends Controller
{

    public function list()
    {
        return view('user/list', [
            'users' => User::all(),
        ]);
    }

}
```

Controllery oproti ostatním frameworkům nemají zásadnější rozdíl a používají se podobně.

5.10.3 Nette

Na rozdíl od předchozích dvou frameworků používá MVP architekturu. Z praktického hlediska zde není až takový rozdíl. Presenter může definovat action a render metody. Action metody jsou v životním cyklu presenteru volány dříve než render metody a můžou změnit šablonu.

Ukázka controlleru a akce pro výpis uživatelů:

```
<?php declare(strict_types=1);

namespace App\Presenters;

use App\Repository\UserRepository;

class UserPresenter extends BasePresenter
{

    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        parent::__construct();
        $this->userRepository = $userRepository;
    }

    public function renderList(): void
    {
        $this->template->users = $this->userRepository
            ->getUserRatingCount();
    }

}
```

Controllery oproti ostatním frameworkům nemají zásadnější rozdíl a používají se podobně.

	Přehlednost	Možnosti
Symfony	Díky možnosti definovat routy v anotacích přímo u akcí je Laravel nejpřehlednější a ve velkých aplikacích je jednoduché spravovat cesty přímo u akcí.	Controller nabízí vše nutné. Tedy přijmout požadavek a vrátit odpověď.
Laravel	Contoller je stejně přehledný jako u Symfony. Laravel nemá doporučený způsob, jak držet logiku aplikace mimo controller, tudíž časem můžou narůstat a stát se méně přehledné.	Funkcionalita a možnosti jsou velice podobné ostatním frameworkům.
Nette	Díky jiné architektuře se používá jinak, ovšem po krátkém zorientování je vše jasné a přehledné.	Funkcionalita je velice podobná a není neumí nic navíc, co by ostatní frameworky nedokázaly.

Tabulka 7 Porovnání controllerů

Zdroj: vlastní tvorba

6 Závěr

Na aplikaci byly ukázány rozdíly a srovnány všechny frameworky. V každé kapitole praktické části byly v tabulce porovnány jednotlivé komponenty použité v aplikaci. V závěru práce jsou srovnány frameworky jako celky a jsou uvedeny jejich největší přednosti.

Použitý způsob porovnání frameworků není platný pro všechny druhy aplikací. Například pro vývoj konzolové aplikace se Nette nehodí a pokud by frameworky byly porovnány právě na konzolové aplikaci, tak by práce mohla vést k jinému závěru porovnání. Stejně tak by porovnání mohlo velice výrazně ovlivnit porovnání nejen samotných komponent frameworků, ale i komunitních knihoven a integrace různých nástrojů, které jsou pro každý framework důležité a ovlivňují jeho použitelnost.

Nejsilnější stránkou Symfony je důraz na přehlednost, jednoznačnost a architekturu aplikace. Podporuje velice pokročilé vlastnosti a funkce, které nejsou v rozsahu této práce. Mezi jeho největší přednosti patří rozšiřitelnost a bohatý ekosystém, ve kterém existuje již vyvinutá funkcionalita pro téměř vše. Symfony se snaží být neutrálním frameworkem, který dokáže poskytnout solidní základnu pro tvorbu různorodých aplikací.

Laravel dbá na jednoduché a pochopitelné používání tak, aby se dalo vyvíjet rychle a bez dlouhého učení. Disponuje velice kvalitní dokumentací s mnoha ukázkami a příklady. Je velice vhodným kandidátem pro začínající vývojáře a prototypování. Laravel si jde svou vlastní cestou a mnohdy spoléhá na konvence, což umožnuje zkrátit zápisy a vyvíjet rychle, ovšem pokud něco nefunguje, tak jak má, tak je obtížné zjistit proč.

Nette se soustředí pouze na vývoj samotného jádra frameworku a nechává vývojáři volnou ruku, tím že neposkytuje tolik funkcionality a komponent jako jiné

frameworky. Pro veškerou důležitou funkcionalitu existuje rozšíření a v tomto ohledu velmi pomáhá komunita. Je vhodnější spíše pro zkušenější programátory, kteří dokáží vybrat vhodnou knihovnu, případně si ji sami implementovat. To se může zdát jako nevýhoda, ovšem pro někoho to může být naopak výhoda, že se mu framework neplete do cesty a neurčuje způsob, jakým co dělat. V tomto ohledu velice pomáhá komunita nadšených vývojářů. Jeho největší výhodou je šablonovací systém Latte.

Frameworky jsou dnes při vývoji aplikací standardem a již není výhodné ani udržitelné budovat aplikaci čistě v PHP. Přispívají i bezpečnosti, jelikož pomáhají předcházet velice častým útokům jako například XSS, CSRF nebo SQL Injection. Každý z nich se vyvíjí svým tempem a navzájem se ovlivňují. Je zde velice velká šance, že pokud jeden framework přijde s nějakou opravdu převratnou vlastností, tak ostatní tuto vlastnost začnou podporovat svým způsobem. Nejedná se o bezduché plagiátorství, nýbrž o zdravou konkurenci, která žene celý ekosystém okolo PHP dopředu. Například Doctrine 2 nemá přímou konkurenci a v posledních letech nepřinesl její vývoj žádnou novou zásadní vlastnost.

7 Seznam použité literatury

[1] PHP FRAMEWORK INTEROP GROUP. *PHP Standards Recommendations - PHP-FIG* [online]. [vid. 2019-11-05]. Dostupné z: https://www.php-fig.org/psr/

[2] ADERMANN, Nils a Jeff BOGGIANO. Composer. *Composer documentation* [online]. [vid. 2019-01-31]. Dostupné z: https://getcomposer.org/doc/

[3] PRESTON-WERNER, Tom. Semantic Versioning 2.0.0. *Semantic Versioning* [online]. 7 2013 [vid. 2019-01-31]. Dostupné z: https://semver.org/

[4] DOUG, Bierer. *PHP Programming Cookbook*. 2016. vyd. B.m.: Packt Publishing, nedatováno. ISBN 978-1-78588-344-6.

[5] ZANDSTRA, matt. *PHP objects, patterns, and practice. Fifth edition.* 5. vyd. California: Apress, nedatováno. ISBN 978-1-4842-1995-9.

[6] JUNADE, Ali. *Mastering PHP Design Patterns*. 2016. vyd. B.m.: Packt Publishing, 2020. ISBN 1-78588-713-0.

[7] *PHP: Introduction - Manual* [online]. [vid. 2020-02-02]. Dostupné z: https://www.php.net/manual/en/intro.pdo.php

[8] FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, Addison-Wesley. ISBN 978-0-321-12742-6.

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

Jméno a příjmení:	**Martin Gold**
Osobní číslo:	**I1600529**
Adresa:	Lidická 391/4, Horka nad Moravou, 78335 Horka nad Moravou, Česká republika
Téma práce:	PHP frameworky
Téma práce anglicky:	PHP frameworks
Vedoucí práce:	Mgr. Daniela Ponce, Ph.D. Katedra informačních technologií

Zásady pro vypracování:

Cílem práce je popsat moderní PHP Frameworky (Nette, Symfony, Laravel) na úrovní jednotlivých komponent tak i celků. Frameworky budou porovnávány na funkčně stejné aplikaci.

1. Úvod
2. Jazyk PHP
3. Stučná historie PHP
4. Moderní PHP
5. Rozdíly mezi frameworky
6. Zaměnitelnost jednotlivých komponent mezi frameworky

Seznam doporučené literatury:

LOCKHART, Josh. Modern PHP: new features and good practices. Sebastopol, CA: O'Reilly Media, 2015. ISBN 1491905018.

STAUFFER, Matt. Laravel: up and running: a framework for building modern PHP apps. Sebastopol, CA: O'Reilly Media, 2016. ISBN 1491936088.

FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003. ISBN 0321127420.

BERGMANN, Sebastian. Real-world solutions for developing high-quality PHP frameworks and applications. Indianapolis, IN: Wiley Publishing, [2011]. ISBN 978-0470872499.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum:

© IS/STAG, Portál – Podklad kvalifikační práce , goldma1, 23. dubna 2020 18:12