# Vývojový framework Laravel a možnosti pokročilého kešování

Bc. Dominik Pfeffer

Diplomová práce
2019



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Bc. Dominik Pfeffer** |
| Osobní číslo: | **A17245** |
| Studijní program: | **N3902 Inženýrská informatika** |
| Studijní obor: | **Počítačové a komunikační systémy** |
| Forma studia: | **prezenční** |
| Téma práce: | **Vývojový framework Laravel a možnosti pokročilého kešování** |
| Téma anglicky: | **The Laravel Development Framework and Advanced Caching Methods** |

Zásady pro vypracování:

1. **V rámci teoretické části popište vlastnosti a architekturu vývojového frameworku Laravel a jeho základní komponenty.**
2. **Stručně charakterizujte návhové vzory využívané v oblasti tvorby webových aplikací.**
3. **Prozkoumejte možnosti a popište správné metody kešování ve vývojovém frameworku Laravel.**
4. **Implementujte webovou aplikaci, která bude integrovat nejpoužívanější druhy kešování, využijte nástrojů FW Laravel pro generování testovacích databázových dat.**
5. **Testovacím provozem otestujte výkon aplikace s různým nastavením kešováním, metody testů popište a jejich výsledky vyhodnoťte.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:


1. **STAUFFER, Matt. Laravel: up and running: a framework for building modern PHP apps. Sebastopol, CA: O'Reilly Media, 2016. ISBN 9781491936085.**
2. **Design patterns in PHP and Laravel. New York, NY: Springer Science+Business Media, 2016. ISBN 9781484224502.**
3. **SOLIMAN, Ahmed. Getting Started with Memcached. Birmingham: Packt Publishing, 2013. ISBN 978-1-78216-322-0.**
4. **DAS, Vinoo. Learning Redis. Birmingham: Packt Publishing, 2015. ISBN 978-1783980123.**
5. **BEAN, Martin. Laravel 5 Essentials. Birmingham: Packt Publishing, 2015. ISBN 9781785283017.**
6. **Laravel [online]. USA: Taylor Otwell, 2018 [cit. 2018-11-27]. Dostupné z: https://laravel.com/**
7. **DAYVSON DA SILVA, Maxwell. Redis Essentials. Birmingham: Packt Publishing, 2015. ISBN 978-1784392451.**
8. **MALATESTA, Francesco. Learning Laravel's Eloquent. Birmingham: Packt Publishing, 2015. ISBN 978-1784391584.**


Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **30. listopadu 2018**

Termín odevzdání diplomové práce: **17. května 2019**

Ve Zlíně dne 10. prosince 2018


doc. Mgr. Milan Adámek, Ph.D.
*děkan*

Ing. Miroslav Matýsek, Ph.D.
*ředitel ústavu*

**Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 21. 5. 2019 Dominik Pfeffer, v.r.

## ABSTRAKT

Tato diplomová práce se zabývá návrhem a implementací webové aplikace pro testování nejpoužívanějších cachovacích systémů. Aplikace je psána pomocí jazyka PHP ve vývojovém frameworku Laravel. Práce zahrnuje cachovací metody, zhodnocení cachovacích systémů a popis frameworku Laravel a je rozdělena na teoretickou a praktickou část. V teoretické části jsou popsány pojmy typu framework, návrhový vzor, cache aj. Velká část práce je věnována samotnému frameworku Laravel a jeho vlastnostem. Dále obsahuje základní popis cachování. V praktické části je provedena integrace nejpoužívanějších cachovacích systémů a následné zhodnocení.



Klíčová slova: Laravel framework, Laravel, MVC, Cache, Web cache, PHP, Redis, Memcached



## ABSTRACT

This thesis deals with the desing patterns and implementation of web application for testing cache's systems. The application is written by PHP in the Laravel development framework. The thesis includes cache systems, assesment of the caching systems and description of Laravel framework. The thesis is divided into theoretical and practical part. In the theoretical part are described the terms of framework, design pattern cache etc. A great part of thesis is devoted to Laravel's framework and its features. Next part is devoted to caching itself. In the practical part is described integration of the most used caching systems and application is made.



Keywords: Laravel framework, Laravel, MVC, Cache, Web cache, PHP, Redis, Memcached

Tímto bych chtěl poděkovat mému vedoucímu diplomové práce Ing. Radkovi Valovi, Ph.D., za odborné vedení, konzultace a rady.

*„When you see a good move, look for better one.“*

*Emanuel Lasker*

Prohlašuji, že odevzdaná verze bakalářské/diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

# OBSAH

# ÚVOD

Tématem diplomové práce je prozkoumat možnosti pokročilého cachování ve vývojovém frameworku Laravel a implementovat je v testovací aplikaci. Diplomová práce neslouží jako učebnice cachování, může však pomoci při výběru cachovacích metod a navést k jejich správné implementaci.

Téma práce bylo vybráno kvůli zvýšeným nárokům na rychlost webových aplikací a malé informovanosti o správných metodách cachování mezi vývojáři webových aplikací. Neexistuje mnoho návodů či prací, které by rozebíraly správné metody cachování, popisovaly cachovací systémy a srovnávaly jejich výkonnost.

V teoretické části budou objasněny pojmy jako návrhový vzor či framework. Ve druhé kapitole bude popsán framework Laravel, jeho historie, základní vlastnosti, jako jsou například migrace, seeding, Eloquent model a mnohé další. V dalším bodě diplomové práce budou zmíněny základní informace o cachování pomocí Laravelu. Třetí kapitola popisuje cache, webovou cache a rozdělení cachí. Závěr teoretické části se věnuje NoSQL databázím, především databázím typu klíč-hodnota a nejpoužívanějším cachovacím systémům navržených na jejich principu, jako jsou Redis a Memcached.

Praktická část obsahuje popis technologie použité pro tvorbu testovací aplikace. V další kapitole budou zmíněny funkční požadavky na tvorbu testovací aplikace a testování cachovacích systémů. Následně přibližuje možnosti zprovoznění cachovacích systémů na operačním systému Windows. Dále budou popsány tři základní metody cachování ve webové aplikaci. Pomocí ukázky kódu budou rozebrány důležité části aplikace. Testování a porovnání cachovacích systémů je pro větší přehlednost popsáno na závěr ve vlastní kapitole.

# I. TEORETICKÁ ČÁST

# 1 OBECNÉ NÁVRHOVÉ VZORY

Návrhový vzor je postup, dle kterého se program vytváří. Jedná se o ověřené programovací postupy pro řešení pravidelně se vyskytujících situací, podle nichž vývojáři přistupují k problémům. To znamená, že návrhové vzory umožňují napsat čitelnější kód, v němž se budou orientovat i vývojáři, jež tento kód nepsali. [15]

Návrhový vzor se obecně skládá ze čtyř základních prvků:

**Název vzoru** je záhlavím, jež je používáno k popisu návrhového vzoru, jeho řešení a důsledků. Popisuje se zpravidla jedním nebo dvěma slovy.

**Problém** obecně popisuje podmínky, kdy se má daný vzor použít. Slouží k vysvětlení problému a jeho kontextu. Může popisovat specifické návrhové problémy, jako jsou například možnosti reprezentace algoritmů jako objektů. Dále může problém obsahovat podmínky, které musí být splněny, než bude možné zavést daný návrhový vzor.

**Řešení** popisuje prvky, z nichž se návrh skládá, avšak nepopisuje konkrétní problém. Vzor slouží jako šablona, která se používá v různých situacích. Poskytuje abstraktní popis návrhového vzoru. [23]

**Důsledky** jsou výsledky a kompromisy použití vzoru. Softwarové výsledky se zabývají prostorovými a časovými kompromisy. Mohou řešit jazykové a implementační problémy.

Návrhové vzory se dělí:

**Creational patterns** (vytvářející) – řeší problémy související s vytvářením objektů v systému. Popisují postupy výběru třídy nového objektu a zajišťují správný počet těchto objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu. Mezi tyto vzory patří například Singleton, Abstract Factory, Factory Method. [24]

**Structural patterns** (strukturální) – představují skupinu návrhových vzorů, které se zaměřují na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Hlavním cílem je zpřehlednit systém za pomocí strukturalizace kódu. Mezi tyto vzory patří například Facade, Decorator, Adapter. [24]

**Behavioral patterns** (chování) – jsou zpravidla založeny na třídách nebo objektech, zajímají se o chování systému. U tříd využívají především principu dědičnosti. Dále řeší spo-

lupráci mezi objekty a skupinami objektů, která zajišťuje dosažení požadovaného výsledku. Mezi tyto vzory patří například Observer, Interpreter, Command. [24]

# 2 LARAVEL

V následující kapitole bude popsána adresářová struktura, historie, základní vlastnosti frameworku Laravel, definice frameworku.

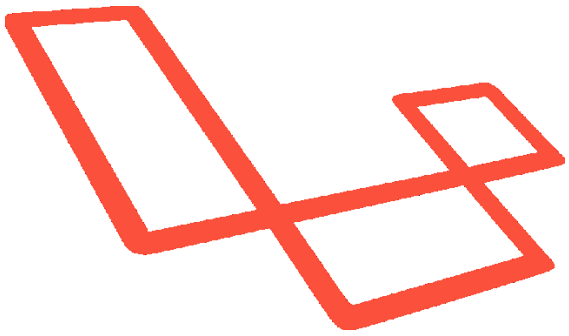
## 2.1 Framework

Framework je sada předpřipravených softwarových bloků, knihoven, nástrojů, konvencí a osvědčených postupů, které vytváří abstrakci nad rutinními úkoly do obecných modelů, které mohou být lehce použity znovu. Jsou určeny pro vykonávání základní funkcionality aplikace. Při použití frameworku se programátor může více soustředit na celkový projekt namísto psaní neustále se opakujícího kódu. Jedná se o části, které se vyskytují pravidelně a téměř v každém větším projektu, např. přihlášení a odhlášení uživatelů, tvorba a zpracování formulářů. [1]

K nejznámějším PHP frameworkům patří Symfony, Zend, CodeIgniter, Laravel. V České republice je nejvíce používán Nette. [1]

## 2.2 Historie

Laravel je PHP Framework pro webové aplikace. Framework Laravel vytvořil sir Taylor Otwell jako náhradu pro starší PHP Framework CodeIgniter. Oficiální verze frameworku byla zveřejněna 20. června 2011. Do konce roku 2014 byl vývoj tohoto frameworku Otwellovou vedlejší činností. Od ledna 2015 je práce na Laravelu jeho hlavní činností. [24]



Obrázek 1: Logo frameworku Laravel [19]

Tabulka 1: Historie frameworku Laravel [26]

| Verze | Datum vydání |
|---|---|
| **V1** | 20. června 2011 |
| **V2** | 24. listopadu 2011 |
| **V3** | 22. února 2012 |
| **V4** | 28. května 2013 |
| **5.0** | 4. února 2015 |
| **5.1** | 9. června 2015 |
| **5.2** | 21. prosince 2015 |
| **5.3** | 23. srpna 2016 |
| **5.4** | 24. ledna 2017 |
| **5.5** | 30. srpna 2017 |
| **5.6** | 7. února 2018 |
| **5.7** | 6. září 2018 |
| **5.8** | 26. února 2019 |

## 2.3 Adresářová struktura

Výchozí adresářová struktura nabízí dobrý začátek pro malé i velké aplikace. Další organizaci adresářů si uživatel může přizpůsobit svým potřebám.

- *app* – tento adresář obsahuje hlavní kód aplikace.
- *bootstrap* – soubory obsažené v tomto adresáři slouží k nastartování frameworku a konfiguraci autoloadingu. Uvnitř adresáře se nachází i složka *cache*, která obsahuje frameworkem generované soubory za účelem optimalizace výkonu.
- *config* – obsahuje všechny konfigurační soubory aplikace.
- *database* – tato složka obsahuje databázové migrace a seedy.
- *public* – tento adresář obsahuje soubor *index.php*, který je vstupním bodem při všech požadavcích aplikace. Dále se zde nachází *obrázky*, *javascript* nebo *css soubory*.
- *resources* – zde se nachází všechny pohledy (views) a nezkompilované soubory, jako například *SASS*, *LESS* nebo *Javascript*. Dále se zde nachází multijazyčné soubory pro snadný překlad.

- *routes* – tato složka obsahuje všechny definované cesty (routy) aplikace. Nachází se zde tři soubory *web.php*, *api.php* a *console.php*.
- *storage* – tento adresář obsahuje zkompilované *blade šablony*, *sessions*, *cache* a jiné soubory vygenerované frameworkem, dále se zde nachází složky *app*, *framework* a logs.
- *tests* – tento adresář obsahuje všechny integrační testy (*feature*) a jednotkové testy (*unit*).
- *vendor* – tento adresář obsahuje všechny *composer balíčky*.

**Důležité soubory nacházející se v kořenovém adresáři**

- *.env* – lokální nastavení aplikace.
- *artisan* – PHP skript realizující konzoli. Umožňuje spouštět artisan příkazy v příkazovém řádku, které se používají při vývoji aplikace.
- *composer.json a composer.lock* – obsahuje informace o verzích PHP knihoven.
- *package.json* – v tomto souboru se nachází informace o verzích knihoven pro npm.
- *Webpack.mix.js* – nastavení webpack. [14]

## 2.4 Základní vlastnosti

Laravel si obecně zakládá na jednoduchém a přehledném kódu a je optimalizovaný pro reálné webové aplikace.

### 2.4.1 MVC architektura

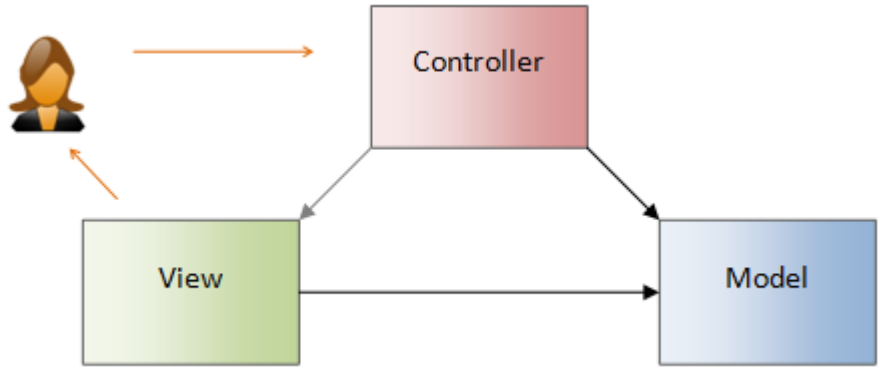
Tento typ architektury využívá mnoho webových frameworků, mezi které se řadí Symfony, .NET či Laravel. Model–View–Controller je softwarový architektonický vzor, jehož základní myšlenkou je oddělení logiky od výstupu. Hlavní uplatnění spočívá nejčastěji u aplikací klient/server (webových aplikací). Klade si za cíl vyřešit problémy tzv. „špagetového kódu“, kdy máme v jedné třídě logické operace a zároveň renderování (vykreslení) výstupu. To znamená, že soubor obsahuje databázové dotazy, logiku a různé HTML tagy. Autorem této architektury je Trygve Reenskaug, jenž ji navrhnul již v sedmdesátých letech 20. století. [3]

Celá aplikace je tedy rozdělena do tří částí, které se nazývají: model, view, controller.

**Model** – obsahuje aplikační logiku a data, do níž patří databázové dotazy, výpočty, validace apod. Funkce modelu je založena na přijetí a zpracování parametrů. O existenci view nebo controlleru neví. [3]

**View** – tato vrstva se stará o zobrazení výstupu uživateli. Obsahuje minimální množství logiky nutné pro vykreslení. Vrstva view neví, odkud data pochází, stará se pouze o jejich výpis. Webové frameworky obvykle využívají pro větší pohodlí uživatelů šablonovací systémy. [3]

**Controller** – je řídící vrstva, která se chová pro model a view jako prostředník. Zpracovává veškeré požadavky od uživatele, komunikuje s modelem i pohledem a vzájemně je propojuje. [3]




Obrázek 2: Schéma MVC [4]

Na obrázku výše můžeme vidět princip fungování MVC modelu.

Uživatel vykoná akci v uživatelském rozhraní. Tato akce je zachycena pomocí controlleru. Controller pomocí modelu získá všechna data, která bude potřebovat. Tato data controller předá správnému view. Následně je view zobrazen uživateli. [4]

### 2.4.2 Migrations

Migrace si lze představit jako verzovací systém pro databázi, který umožňuje týmu spolupracovníků upravovat a sdílet databázovou strukturu. Díky migracím odpadá nutnost manuálního vytváření databázových tabulek. U větších aplikací je tento proces velmi zdlouhavý a náchylný na chyby. Migrace tento proces sjednocují.

Funguje to tak, že se vytvoří soubor migrací, ve kterém se definuje požadovaná struktura databázových tabulek. Následně migraci spustíme a Laravel se za nás postará o zbytek. Migrace po vygenerování můžeme najít ve složce *app/database/migrations* a jsou typicky spárované se *Schema builderem*, což je třída obsahující metody pro definici a úpravu struktur databázových tabulek. [18]

### 2.4.3 Seeding

Laravel svým uživatelům nabízí jednoduchý způsob nahrávání (seeding) testovacích dat do databáze za použití speciálních tříd. Pomocí těchto seedovacích tříd definujeme Laravelu, jaký typ a kolik údajů má do konkrétní tabulky vložit. Tyto údaje můžeme definovat ručně anebo náhodně generovat pomocí *Factories,* které jsou popsané níže. Všechny seedovací třídy jsou uloženy v adresáři *database/seeds*. [14]

### 2.4.4 Factories

Při testování může vývojář potřebovat vložit data do databáze. Místo manuálního specifikování hodnot pro každý sloupec tabulky, umožňuje Laravel definovat základní nastavení atributů pro každý *Eloquent model* za použití tzv. továrniček (Factories). Ke generování konkrétních záznamů se používá knihovna *Faker generator*. Tato knihovna generuje data tak, aby jejich tvar byl co nejvíce podobný realitě. Můžeme generovat záznamy nejrůznějšího druhu, jako například e-mailovou adresu, jména, příjmení uživatelů, telefonní čísla. Factories jsou zpravidla ukládány v adresáři *database/factories*. [19]

### 2.4.5 Eloquent ORM

Eloquent ORM (Object–relational mapping) přináší jednoduchou implementaci návrhového vzoru *ActiveRecord* pro práci s databází. Každá databázová tabulka má příslušný *Model* sloužící k jejich interakci. Model umožňuje vytvářet databázové dotazy, upravovat nebo vkládat nové záznamy bez nutnosti psaní SQL dotazů. [21]

### 2.4.6 Šablonovací systém Blade

Webové frameworky používají nejrůznější nástroje pro práci se šablonami. Tyto nástroje si kladou za cíl oddělit HTML kód od programovacího jazyka.

Blade je jednoduchý šablonovací engine poskytovaný frameworkem Laravel. Na rozdíl od většiny šablonovacích systémů Blade nezakazuje používání PHP kódu v pohledu, tudíž neslouží pouze pro vkládání proměnných do HTML kódu. Tento šablonovací systém používá speciální syntaxi a kompiluje všechny pohledy (views) do čistého PHP kódu. Tyto pohledy jsou ukládány do mezipaměti pro příští použití z důvodu rychlejšího načítání. Šablony se ukládají do mezipaměti při každé nové úpravě. Soubory šablonovacího systému Blade používají příponu *.blade.php* a jsou uloženy v adresáři *resources/views*. [18]

Mezi nesporné výhody tohoto šablonovacího systému patří dědičnost a rozdělení pohledů do sekcí. Každá sekce poté zobrazuje určitou část aplikace. Jako první se definuje hlavní šablona tzv. layout, která se využívá pro všechny ostatní šablony. Zde se definují části aplikace, které jsou pro všechny stránky stejné, například hlavička, patička a menu. Do této hlavní šablony se poté načítá obsah jednotlivých stránek. Velkou výhodou tohoto způsobu je jeho jednoduchá údržba.

Ukázka hlavní šablony tzv. layoutu:

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

### 2.4.7 Laravel cachování

Laravel nabízí cachování na úrovni webové aplikace, které je hlavním tématem práce a bude podrobně popsáno v praktické části. Zde jsou popsány jen základní informace.

Laravel poskytuje sjednocené API pro nejrůznější druhy cachování. Nastavení cachování se nachází ve složce *config* v souboru *cache.php*. V tomto souboru je možné specifikovat,

který cache ovladač bude v aplikaci použit. Laravel podporuje nejrůznější možnosti cachování, jako je Memcached a Redis. [7]

Ve výchozím nastavení je Laravel nastaven pro cachování do souboru za použití ovladače *cache file*. Pro rozsáhlejší webové aplikace se doporučuje používat *Memcached* nebo *Redis* ovladače. [7]

Máme na výběr z šesti různých úložišť pro cachování:

- database,
- apc,
- array,
- file,
- memcached,
- redis.

**Database** – data jsou ukládána do databázové tabulky, kde je definovaný klíč, hodnota a datum expirace dané cache.

**Apc** (Alternative PHP cache) – jedná se o open source cachovací plugin pro PHP. Apc byl vyvinut s cílem poskytnout zdarma knihovnu pro cachování a optimalizaci PHP kódu. Tato knihovna je již zastaralá, a proto se používá jen zřídka. Byla nahrazena modernější verzí, která se nazývá APCu.

**Array** – tato metoda cachování je určena převážně pro testování, a to z toho důvodu, že obsah se ukládá do paměti a není přístupný mimo běžící proces PHP aplikace. [8]

**File cache** (souborová cache) – cachování do souboru dovolují všechny moderní PHP frameworky. Zpravidla je tato možnost implementována jako uložení serializovaného objektu na serveru ve formě souboru. Využívá se především u malých stránek. Tato možnost cachování je defaultně nastavena ve frameworku Laravel. [8]

**Memcached** – je open-source key-value paměťový systém. Memcached běží jako služba oddělená od aplikace. Tato databáze slouží především jako cache paměť zaměřená na vysoký výkon. Složitost získání dat z klíče je O(1), to znamená, že je konstantně rychlá, bez ohledu na to, co do ní uložíte. [10]

**Redis** – je NoSQL (Not only SQL) open source key-value databáze, která je především používaná jako cache paměť nebo pro sdílení dat mezi různými procesy. Podporuje standardní datové typy, jako jsou string, hash, list, sorted set a v závislosti na daném typu nabí-

zí některé pokročilejší funkce (inkrementace, uložení nebo aktualizace hodnot v asociativním poli najednou, průnik, sjednocení a rozdíl množin hodnot). [9]

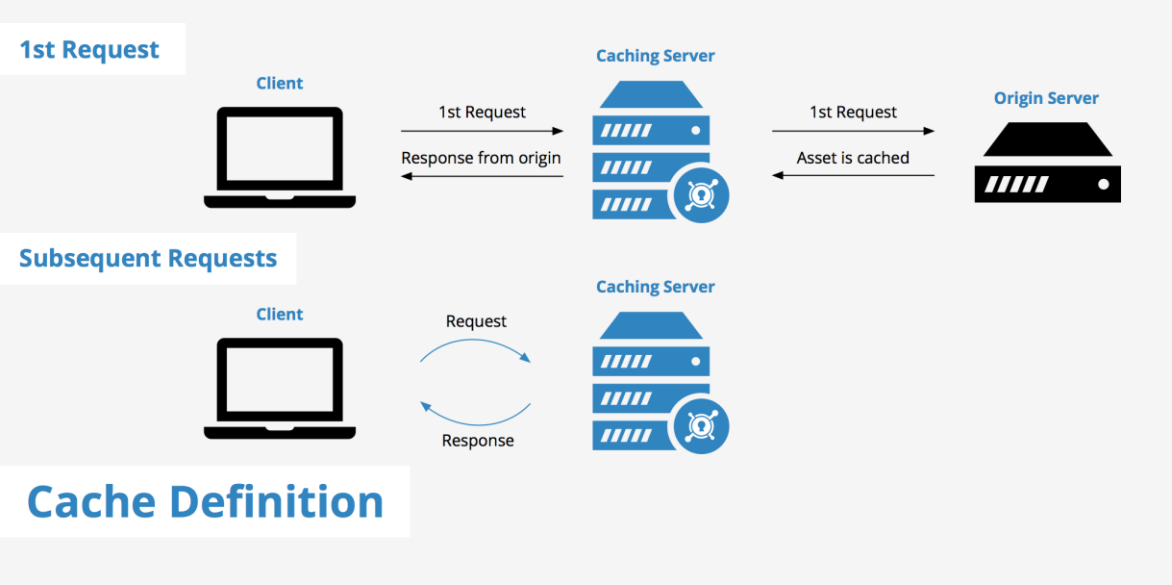
# 3 CACHE

V následujících kapitolách je popsána cache, webová cache, její rozdělení a důležité informace pro samostatné pochopení cachování.

## 3.1 Základní definice

Obecně cache (keš) je označení pro rychlou mezipaměť, do které se duplikují často používaná a těžce dostupná data, z důvodu rychlejšího přístupu k nim. [6]

Cachování je jednou ze základních technik optimalizace výkonu aplikace. Základní myšlenka cachování je umístění informace na rychlejší médium, a tím docílení znatelného zlepšení výkonu při přistupování k těmto datům. [6]




Obrázek 3: Schéma cachování [5]

Existuje více druhů cachovacích mechanismů, které existují ve výpočetní technice. Zde budou uvedeny nejpoužívanější.

**Serverová cache** – jedná se o server, který slouží k ukládání webových zdrojů do mezipaměti. Tento typ cachovacího mechanismu se používá u proxy serverů. Tyto servery mohou být umístěny v mnoha geografických oblastech. [5]

**Prohlížečová cache** – webové prohlížeče ukládají soubory do své lokální cache paměti, takže k nim lze přistupovat rychleji, protože je není potřeba stahovat ze serveru. Využití cache paměti prohlížeče je důležitou optimalizační metodou, která je i snadno implementovatelná. [5]

**Paměťová cache** – tato cache je mírně odlišná od dvou výše uvedených typů mezipamětí. Tento mechanismus ukládání dat v počítači slouží k urychlení přístupu k datům v rámci aplikace. Určité části dat se ukládají do statické paměti RAM (SRAM), protože se k takto uloženým souborům přistupuje rychleji, než kdyby byly uložené na pevném disku. Příkladem může být program pro nahrávání hudby, kdy jsou načteny určité zvukové soubory do SRAM. [5]

**Disková cache** – tato cache je velmi podobná paměťové cachi, protože také slouží k urychlení přístupu k datům v rámci aplikace. Rozdíl spočívá v tom, že místo použití SRAM používá cache konvenční paměť RAM. Disková cache ukládá data, která byla nedávno přečtena, a sousední bloky dat, ke kterým se bude pravděpodobně brzy přistupovat. [5]

## 3.2 Web cache

Webové cachování je dočasné ukládání objektů (například HTML dokumentů) pro pozdější obnovení. [6]

Webová cache je zařízení nebo softwarová aplikace použitá k dočasnému ukládání kopií informací dodaných ze serveru. Využívání webové cache má dvě hlavní výhody:

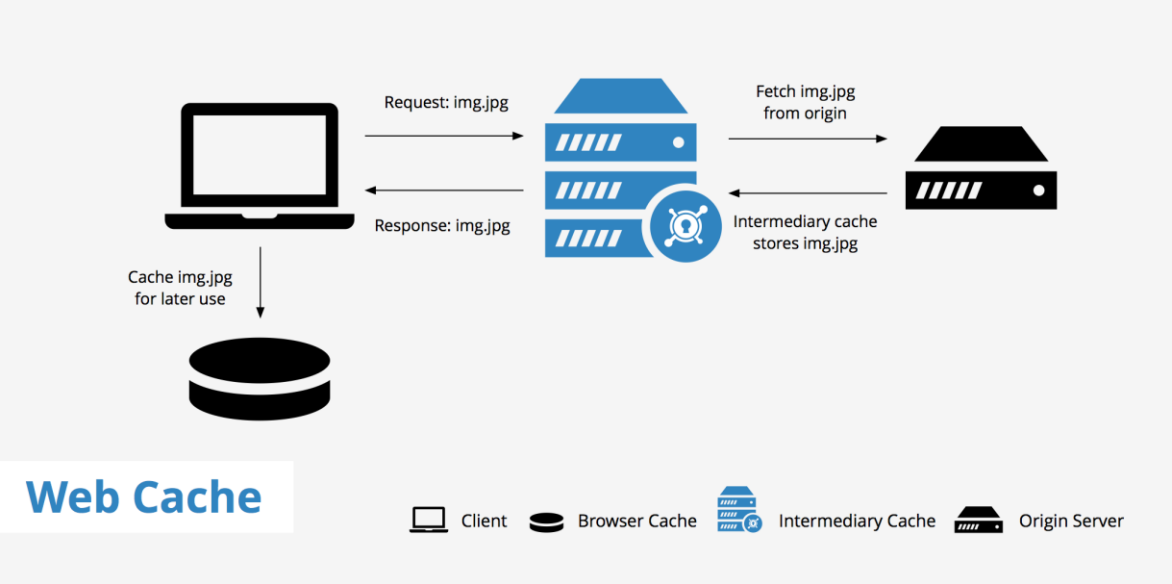
- **Snižuje latency** – webová cache snižuje latency tím, že poskytuje metodu ke zkrácení fyzické vzdálenosti mezi klientem a zdrojem dotazu. V případě CDN jsou Multiple edge servery rozšířeny po celém světě a poskytují uživatelům přístupový bod, který se nachází v jejich blízkosti.
- **Snížení zátěže původního serveru** – vzhledem k tomu, že požadované prostředky jsou dodávány z mezipaměti namísto původního serveru, dochází ke snížení zatížení na tomto serveru. Snížená zátěž na původním serveru vede k lepší připravenosti na dopravní špičky, což snižuje šanci havárie serveru.

### 3.2.1 Jak webcache pracuje

CDN (content delivery network) je dobrý příklad webové cache. CDN stojí mezi původním serverem a klientem, jenž dotazuje nějakou informaci, a ukládá kopie zdrojových dat z původního serveru. S povoleným CDN se provede následující proces:

1. Klient se dotazuje na nějakou konkrétní stránku, tato stránka obsahuje data *a*, *b* a *c*, která musí být načtena.

2. Dotaz se dostane na CDN, kde se zkontroluje, zda už tato data byla uložena. Existují dvě možnosti:
    a. Data jsou uložena v mezipaměti a CDN server vrátí dotazem požadovaná data klientovi.
    b. Data ještě nebyla uložena do mezipaměti a proces pokračuje bodem 3.
3. Dotaz pokračuje přes CDN server a jde přímo do zdrojového serveru.
4. Data *a*, *b* a *c* jsou načtena ze zdrojového serveru.
5. Na cestě zpátky ke klientovi jsou data uložena do paměti CDN serveru.




Obrázek 4: Schéma webové cache [22]

### 3.2.2 Druhy webových cachí

Zde je popsáno dělení webových cachí a jejich základní popis.

**Prohlížečová cache**

Prohlížečová cache funguje na úrovni internetového prohlížeče. Všechny prohlížeče mají cachovací mechanismus k lokálnímu ukládání webových dat, aby k nim mohlo být rychleji přistupováno. Tyto data se ukládají na pevném disku uživatelova počítače. Prohlížečová cache pracuje na základě přísně jednoduchých pravidel. Zpravidla jednou za sesssion (jednou za spuštění prohlížeče) se u každého uloženého objektu ujistí, že je objekt čerstvý. Tyto frekvence kontrolování aktuálnosti dat se dají přenastavit. [22]

### Proxy cache

Proxy cache server je server, který ukládá často požadované soubory na pevný disk a při opakujících se požadavcích na tyto soubory je poskytuje přímo z disku. Proxy obsluhuje velké množství uživatelů.

Z toho vyplývá, že tyto servery fungují na stejném principu jako prohlížečové cache, avšak v mnohem větším měřítku. [6]

### Cache na branách

Nazývají se také *rezervní proxy cache* nebo *náhradní cache,* jedná se o cachovací systémy mezi klientem a serverem, například *CDN* nebo reverzní proxy jako *Varnish* . Cache na bránách jsou nasazovány samotnými webmastery s cílem mít dostupnější, spolehlivější a funkčnější web. [22]

# 4 NOSQL DATABÁZE

V následujících podkapitolách bude definován pojem NoSQL databáze. Z databází, které do této kategorie patří, bude rozebrána především databáze typu klíč-hodnota, protože do této kategorie patří cachovací systémy použité pro samostatné testování cachování. Dále jsou rozebrány cachovací systémy Redis a Memcached. Tyto systémy jsou nejpoužívanější v rámci cachování u webových aplikací.

## 4.1 Popis NoSQL

Not only SQL neboli NoSQL, jak je tato databáze populárně nazývána, navrhnul Carlo Strozzi v roce 1998 a byla znovu představena v roce 2009 Ericem Evansem. [27]

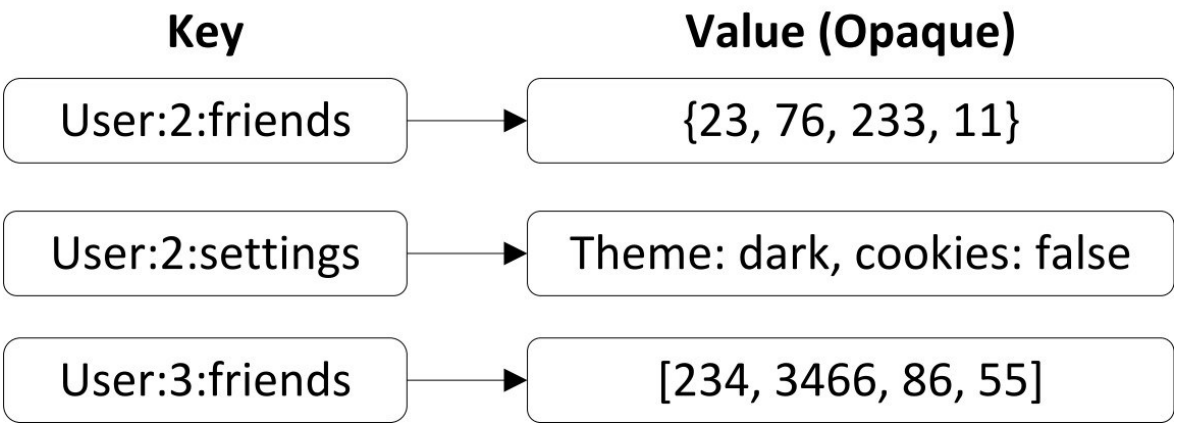
Základní charakteristikou NoSQL databáze je absence pevného schématu (označují se jako tzv. schemaless). U relačních databází se nejdříve definuje schéma databáze, do níž se následně data ukládají. To znamená, že schéma databáze vypovídá o tom, jaké jsou v ní obsaženy tabulky, sloupce a vztahy mezi jednotlivými tabulkami. V NoSQL databázích nebývá schéma předem definováno. Podle datového modelu se rozlišují čtyři hlavní typy NoSQL databází – klíč-hodnota (*key-value store*), sloupcově orientované (*column-oriented*), dokumentově orientované (*document-oriented*) a grafové (*graph*). Databáze typu klíč-hodnota a dokumentově orientované dovolují vkládat jakákoliv data propojená s klíčem. Ve sloupcově orientovaných databázích je možné vytvářet sloupce, do kterých lze ukládat data a přitom počet a názvy sloupců nemusejí být předem definované. Obdobným způsobem se v grafových databázích přidávají nové hrany a definují se jejich vlastnosti a vlastnosti uzlů. [13]

**Databáze typu klíč-hodnota**

Do systému jsou ukládány dvojice skládající se z klíče a hodnoty, přičemž klíč musí být unikátní. Tyto databáze se podobají hash tabulkám, klíč je zároveň indexem, proto je vyhledávání záznamů v těchto databázích velmi rychlé. Preferuje se zde především rychlost a možnost obsluhy více požadavků paralelně na úkor konzistence. Hodnota může být strukturovaná i nestrukturovaná. V systému není možné vytvářet sekundární indexy. Velké množství databází klíč-hodnota zajišťuje perzistenci dat. [13]

U pokročilých databázových systémů mohou být hodnoty reprezentovány několika typy. Tento typ databáze je zpravidla oblíbený díky vysokému výkonu a jednoduchosti. Zpravi-

dla poskytují jen tři základní operace – čtení, zápis a smazání, vše podle klíče. Spousta systémů umožňuje vytvářet buckets (skupiny nebo jmenné prostory) hodnot, aby se hodnoty mohly částečně třídit. Mezi představitele databází tohoto typu patří *Amazon*, *Redis*, *DynamoDb*.




Obrázek 5: Příklad databáze klíč-hodnota [13]

## 4.2 Redis (Remote DIctionary Server)


Obrázek 6: Logo databázového systému Redis [11]

Jedná se o jednoho z nejvýznamnějších zástupců NoSQL databází, patří do kategorie databází *klíč-hodnota*. Nejdříve se jednalo o soukromý projekt vývojáře Salvatora Sanfilippa a nepředpokládalo se, že se později stane opensourcovým projektem. Časem se Salvator Sanfilipp rozhodl vydat volně přístupnou verzi. Její popularita rychle rostla a pod tlakem velkého počtu uživatelů začal autor do databáze přidávat další funkcionality. Díky rostoucí

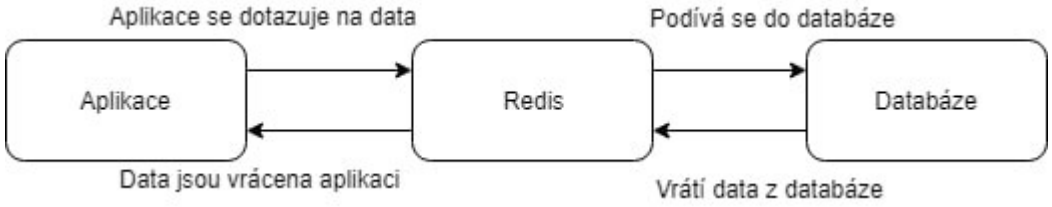
popularitě projektu se autor připojil k firmě VMWare, která měla o tento projekt velký zájem a chtěla pokračovat v jeho rozvoji. [11]

Redis je databáze typu *klíč-hodnota*, jejíž hlavní předností je vysoký výkon. Vývoj Redisu je v dnešní době orientován na cloudové technologie. Redis je psaný v programovacím jazyce C a je oficiálně podporován na UNIX platformách, ale existuje i experimentální verze pro Windows. [9]

Patří do skupiny hybridně perzistentních databází, která primárně funguje jako *in-memory* databáze, jenž má veškerá data v operační paměti, ale umožňuje ukládání dat na disk. Tím se významně snižuje čas čtení / zápis. [20]

Pro ukládání dat Redis používá speciální strukturu, která se nazývá *Simple Dynamic String* (SDS). Tato struktura se skládá ze tří částí:

- **buff** – pole znaků se samotnými daty,
- **len –** číslo typu long, uchovávající délku pole buff,
- **free –** počet dalších bytů volných k použití. [17]




Obrázek 7: Schéma cachování za pomocí Redisu [Zdroj vlastní]

Na obrázku č. 7 lze vidět, že existují dva scénáře při přístupu k datům.

- Aplikace se dotazuje na konkrétní data a zkontroluje, jestli jsou uložená v databázovém systému Redis, pokud ano, okamžitě je vrátí a aplikace s nimi může pracovat.
- Aplikace se dotazuje na konkrétní data, tato data se však nenachází v databázovém systému Redis, proto Redis předá požadavek dál a podívá se do databáze aplikace. Data jsou nejprve uložena do Redisu a následně vrácena aplikaci, aby při opakovaném dotazování mohla být tato data načtena přímo z Redisu.

## 4.3 Memcached

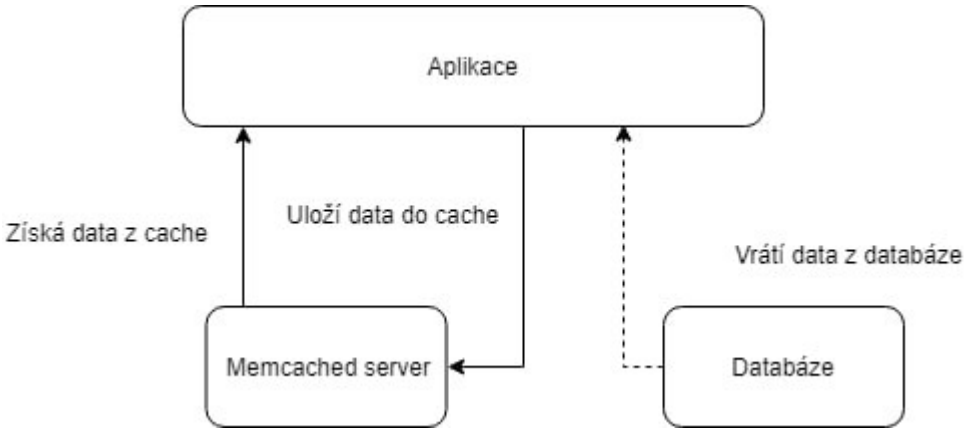
Obrázek 8: Logo databázového systému Memcached [10]

První verze Memcached vznikla v roce 2003 pro LiveJournal, jejím autorem byl Brad FitzPatrick.

Memcached je popisován jako výkonný distribuovaný paměťový cachovací systém s otevřeným zdrojovým kódem a je využíván ke zrychlení dynamických webových aplikací, kvůli snížení potřeb načítání databáze. Jedná se o *in-memory* úložiště, které uchovává dvojice *klíč-hodnota* nebo malá data například řetězce a objekty, které uchovávají výsledky vykreslených stránek, databázových dotazů a API volání. [12]

Memcached funguje tak, že přiděluje nevyužité části paměti systému aplikacím, jež je potřebují.

Na obrázku č. 9 můžeme vidět zpracování dotazu při cachování za pomocí *Memcached*. V této chvíli data zatím nejsou uložena v *Memcached*. Při prvním dotazu jsou data z databáze uložena do operační paměti *Memcached serveru*. Pokud nastane další volání stejného databázového dotazu, data se nebudou získávat z databáze, ale přímo z Memcached serveru. [12]

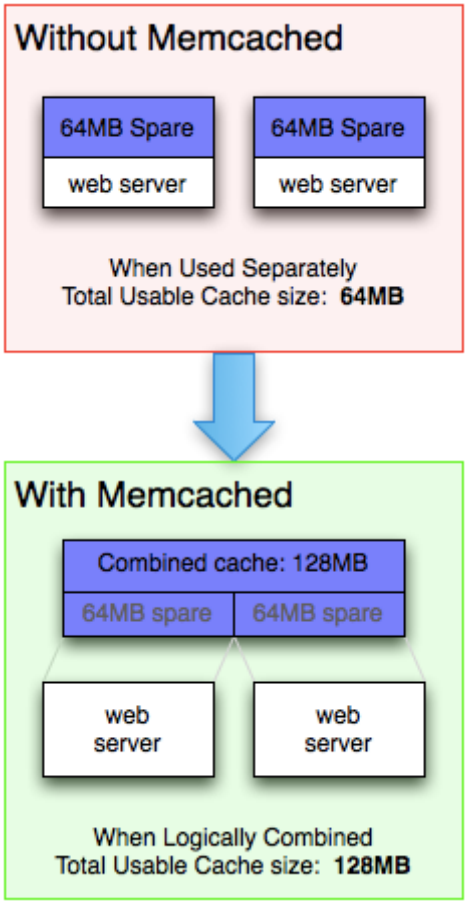



Obrázek 9: Blokové schéma cachování dat zapomocí Memcached [Zdroj vlastní]

Server se nestará o to, jak data vypadají. Data jsou na serveru uložená jako klíč, expirační čas, volitelně může být příznak a poté samotná data. U Memcached nejsou servery navzájem propojeny, což znamená, že neprobíhá žádná synchronizace, zasílání zpráv ani replikace dat. Přidáním dalších serverů se zvyšuje dostupná paměť. Složitost získání dat z klíče je O(1), to znamená, že je konstantně rychlá, bez ohledu na to, co je do ní uloženo. [12]

Implementace Memcached je částečně klientská a částečně serverová. Klient například vybírá, který server bude číst nebo zapisovat data. Server se stará o uložení a načítání dat, řídí, kdy se má paměť vymazat, nebo znovu použít. [16]

Při nedostatku operační paměti je využíván LRU algoritmus pro odložení starých dat na disk. Příkazy mají konstantní asymptotickou složitost a jsou implementovány tak, aby byly provedeny co nejrychleji. [12]




Obrázek 10: Scénáře využití paměti serveru [10]

Ve vrchní části obrázku č. 10 lze vidět, že každý uzel paměti je kompletně samostatný. Ve spodní části obrázku vidíme, že každý server může využít paměť toho druhého.

Memcached se skládá z:

- klientského serveru, kterému je dán list dostupných Memcached serverů,
- klientsky založeného hashovacího algoritmu, který vybere server na základě *klíče*,
- serverového softwaru, který ukládá hodnoty s jejich klíči do vnitřní hashovací tabulky,
- LRU, který určuje, kdy zapomenout stará data (pokud je nedostatek paměti), nebo opětovné použití paměti

Aktuální verze byla vydána v dubnu 2019. [12]

# II. PRAKTICKÁ ČÁST

# 5 VÝBĚR TECHNOLOGIE

V teoretické části byly popsány vlastnosti, technologie vývojového frameworku Laravel spolu s různými druhy cachování. V této části budou popsány metody cachování a postupy nasazení cachovacích systému na framework Laravel.

Jako hlavní nástroj pro programování této testovací aplikace byl zvolen vývojový Framework Laravel. Framework byl zvolen pro svůj vysoký výkon, přehlednou a podrobnou anglickou dokumentaci a variabilní možnosti cachování. Budou využity cachovací systémy a mechanismy, jako jsou Redis, Memcached, database, file cache. Apc ani array cache zde popisovány nebudou z toho důvodu, že Apc je již zastaralé a array cache se využívá zpravidla jen pro testovací účely a není pro reálné nasazení vhodná.

Celá aplikace je psána ve vývojovém prostředí Visual Studio Code, toto IDE (Integrated Development Environment) je zdarma, obsahuje mnoho rozšíření a je použitelné i pro komerční účely.

Technologie:

- MySQL 5.7.25,
- PHP 7.2,
- Laravel Homestead 8.4.0,
- Virtual box 6.0.8,
- Vagrant 2.24,
- Laravel 5.5,
- Memcached 1.5.6,
- Redis 4.0.9.

# 6 APLIKAČNÍ POŽADAVKY

Následující podkapitoly popisují požadavky, cíle a základní funkčnost testovací aplikace.

## 6.1 Funkční požadavky

Aplikace bude realizovat nejpoužívanější možnosti cachování v rámci webové technologie. Obrázky, data a texty v rámci aplikace jsou vygenerované pomocí Laravelu, za pomocí třídy *faker,* která byla popsána v teoretické části.

Aplikace je rozdělena na dvě části. Jedna část obsahuje data načítaná z databáze a druhá z cache paměti. Obě části obsahují stejnou strukturu popsanou níže:

**Články** – kategorie *Posts* vykresluje seznam článků řazených dle abecedy, rozdělených do kategorií. Pomocí těchto kategorií se dají články filtrovat. Detail článku je zobrazen po kliknutí na název článku, tento název slouží jako odkaz.

**Uživatelé** – sekce *Users* vypisuje seznam uživatelů a jejich základní data. Data jsou vypsána do tabulky a zobrazena koncovému uživateli.

**Obrázky** – kategorie *Images* slouží k vykreslení obrázků, jejich názvů a krátkého textového popisku.

## 6.2 Možnosti cachování

Nastavení a realizace cachování jsou popsány v dalších kapitolách. Webová aplikace integruje možnosti cachování za pomoci:

- cachování do souboru,
- cachování do databáze,
- cachování pomocí služby Memcached,
- cachování pomocí služby Redis.

## 6.3 Cíle aplikace

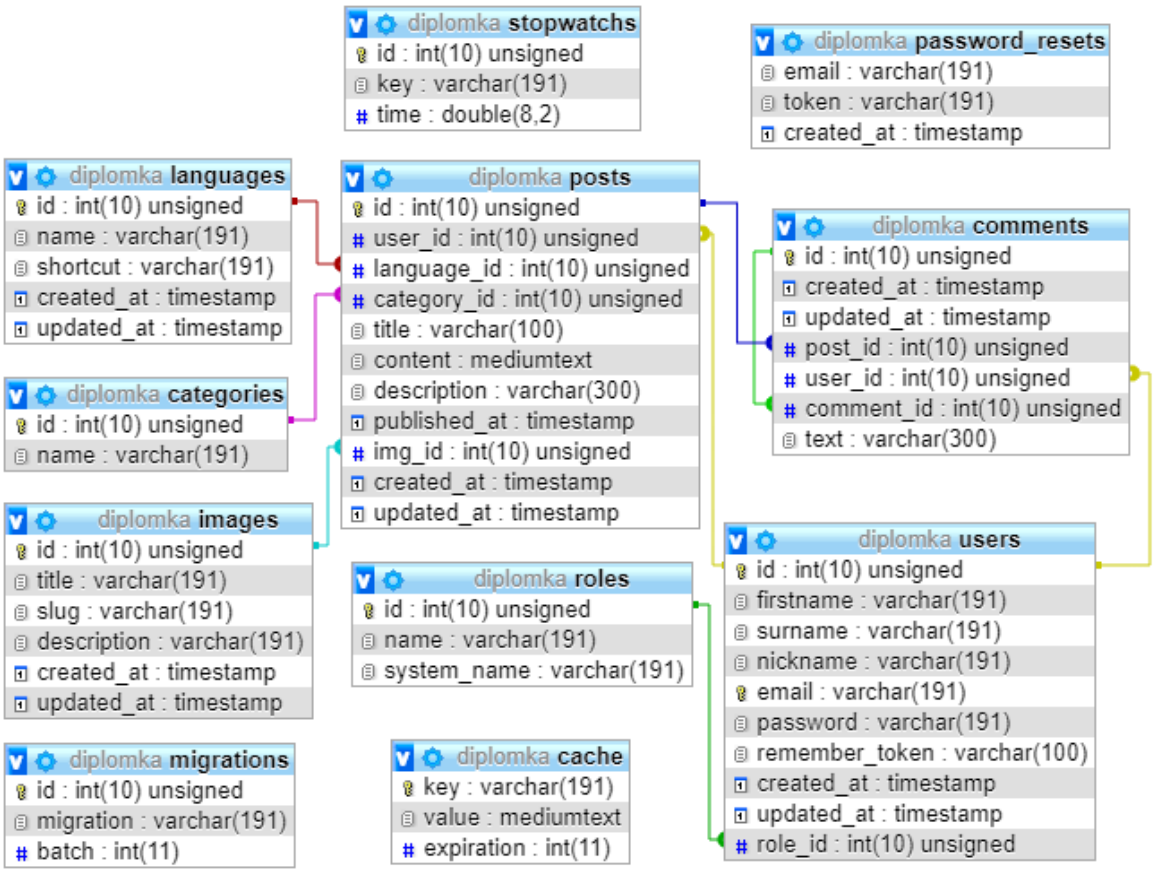
Aplikace slouží ke zjištění, která z popsaných cachovacích metod je optimální pro použití a nasazení do reálného webového provozu.

# 7 NÁVRH DATABÁZE

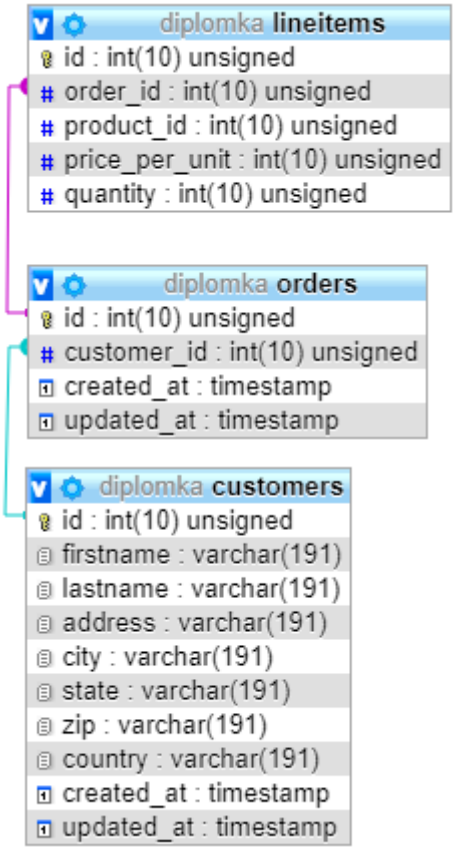
Databáze byla navržena v prostředí *phpMyAdmin* a zkonstruována tak, aby splnila veškeré funkční požadavky a mohla sloužit při testování cachování.

Schéma databázových tabulek bylo vybráno pro testování jak jednoduchých, tak i složitějších databázových dotazů. Složitost databázových dotazů se dále projeví při testování cachování.

Tato struktura databázových tabulek slouží pro simulování načítání článků, jejich členění, zobrazování komentářů. Každý článek má autora, který zastřešuje nějakou roli v systému, např. admin či editor. Tabulka *stopwatchs* slouží k ukládání jednotlivých časů cachování pro další srovnání. Všechny tabulky byly vytvořeny pomocí databázových migrací, které byly popsány v teoretické části.




Obrázek 11: Návrh databáze [Zdroj vlastní]




Obrázek 12: Návrh databáze [Zdroj vlastní]

Na obrázku výše je zobrazena dodatečná databázová tabulka použitá pouze pro testování času komplikovaných databázových dotazů s využitím agregačních funkcí.

# 8 CACHOVÁNÍ LARAVEL

Zde budou popsány základní funkce a metody sloužící ke cachování ve vývojovém frameworku Laravel.

Parametry využívané u cachovacích metod:

- key – speciální identifikátor, podle kterého se uložená data hledají,
- value – data, které chceme uložit,
- seconds – čas, po který mají data zůstat uložená v paměti ve vteřinách.

## 8.1 Metody pro vložení dat do cache paměti

**Základní metoda pro vkládání dat do cache paměti.**

Do cache paměti můžeme přidávat data za pomocí metody *put()*. Tato metoda přijímá tři povinné argumenty.

```
Cache::put('key', 'value', $seconds);
```

**Metoda pro vložení dat, pokud daný klíč neexistuje**

Metoda *add()* vloží data do cache paměti, pouze pokud už v paměti neexistují data se zadaným klíčem. Tato metoda přijímá stejné tři parametry jako metoda předchozí.

```
Cache::add('key', 'value', $seconds);
```

**Metoda pro dlouhodobé ukládání dat**

Metoda *forever()* se používá, pokud chceme uložit data do cache paměti permanentně. Data uložená tímto způsobem neexpirují, to znamená, že pro jejich odebrání je musíme odebrat ručně za pomocí metody *forget()*.

```
Cache::forever('key', 'value');
```

## 8.2 Metody pro získávání dat z cache paměti

Metoda *get()* slouží pro načtení dat z cache paměti. Pokud se žádná položka se zadaným klíčem v mezipaměti nenachází, bude vrácena hodnota *null*.

```
$value = Cache::get('key');
```

Pokud klíč neexistuje a chceme vrátit nějakou defaultní hodnotu, můžeme jako druhý parametr zavolat funkci s návratovou hodnotou.

```
$value = Cache::get('key', function () {
        return DB::table(...)->get();
    })
```

## 8.3 Kontrola existence klíče

Občas je potřeba zkontrolovat, jestli klíč existuje dříve, než aktualizujeme či vrátíme data. K tomu slouží metoda *has()*.

```
if (Cache::has('key')){
        Cache::get('key');
    } else {
        Cache::put('key', $values, 10);
    }
```

Na příkladu výše kontrolujeme, jestli klíč existuje, a pokud ano, vrátíme data, pokud ne, data se zadaným klíčem vložíme do cache paměti.

## 8.4 Vrácení nebo uloží dat do cache paměti

Další metoda pro ukládání dat do cache paměti je *remember()*. Tato metoda je speciální, nejprve zkontroluje, jestli už zadaný klíč existuje, pokud ano, vrátí požadovaná data, a pokud ne, uloží data, která se nachází ve funkci pod zadaným klíčem. Metoda *remember()* je hlavní metodou, která se používá při vývoji dále popisované testovací aplikace.

```
$value = Cache::remember('users', $seconds, function () {
        return DB::table('users')->get();
    });
```

Pro získání či uložení dat do cache paměti natrvalo můžeme použít metodu *rememberForever()*, tato metoda se používá pro ukládání položek, které se měnit nebudou vůbec anebo jen zřídka.

```
$value = Cache::rememberForever('users', function () {
        return DB::table('users')->get();
    });
```

## 8.5 Mazání paměti cache

Cache paměť může být vymazána za pomocí metody *forget()*. Tato metoda vymaže data, jež se nacházela v mezipaměti pod zadaným klíčem.

```
Cache::forget('key');
```

Pro získání dat a jejich následné vymazání se použije metoda *pull()*.

```
$value = Cache::pull('key');
```

Nastanou situace, kdy je třeba vymazat celou cache pamět. V této situaci se používá metoda *flush()*.

```
Cache::flush();
```

# 9 REALIZACE

Zde jsou popsány důležité části vytvořené aplikace. Podkapitoly se věnují správné integraci cachovacích systémů do webové aplikace. Dále je popsána instalace prostředí Laravel Homestead. Laravel Homestead slouží jako lokální prostředí celé aplikace, které integruje služby Redis a Memcached.

## 9.1 Instalace prostředí Laravel Homestead

Laravel Homestead je oficiální předpřipravený *Vagrant* balíček, jenž poskytuje vývojové prostředí bez nutnosti instalování PHP, webového serveru a dalších podpůrných softwarů. Toto prostředí bylo zvoleno kvůli integraci služeb Redis a Memcached, které jsou důležité pro autorem testovanou aplikaci.

Před instalací samotného Homesteadu je potřeba nainstalovat níže uvedený software:

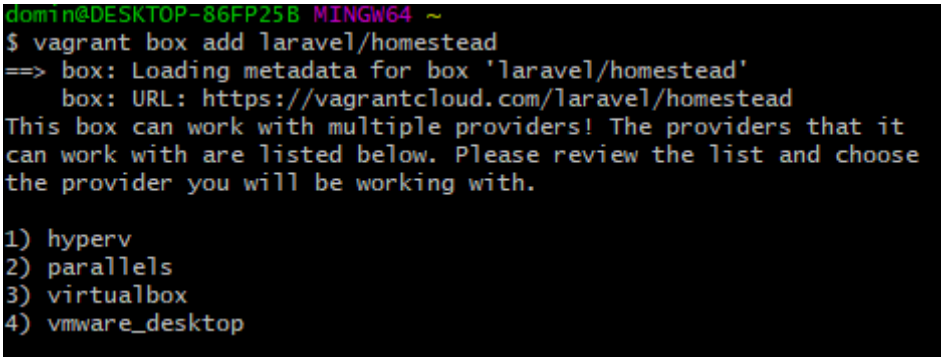
- VirtualBox,
- Vagrant,
- Composer,
- Git.

Veškerý výše uvedený software je k dispozici zdarma a je možné jej používat i v komerčním prostředí. Dostupný je zdarma na oficiálních webových stránkách distributora. Programy lze nainstalovat klasickým způsobem pomocí *.exe* souboru. Nejsou potřeba žádná speciální nastavení instalovaného software.

Instalace prostředí Homestead probíhá přes konzoli *gitu*. Ta se otevře pomocí kliknutí pravým tlačítkem na složku, do které se bude Homestead instalovat a výběrem možnosti *Git Bash Here*. Do konzole *gitu* je zadán příkaz:

```
vagrant box add laravel/homested
```

Tímto příkazem se ve Vagrantu vygeneruje balíček *laravel/homestead*. Následně byl vybrán číselnou volbou VirtualBox. Tato akce chvíli trvá, stažení balíčku závisí na rychlosti internetového připojení.




Obrázek 13: Instalace Laravelu Homestead [Zdroj vlastní]

Následuje samotná instalace prostředí Homestead pomocí příkazu:

```
git clone https://github.com/laravel/homestead.git ~/Homestead
```

Tento příkaz naklonuje *git* repozitář ze zadané adresy do složky s názvem Homestead. Dalším krokem je spuštění batch souboru, který je uložen ve složce Homestead. Tento soubor se jmenuje *init.bat*. Po spuštění tohoto souboru se vygeneruje důležitý konfigurační soubor s názvem *Homestead.yaml*. Soubor *Homestead.yaml* se vytvoří ve složce *C:\Users\uzivatelske_jmeno\Homestead* a slouží pro mapování složek, aby se změny provedené v projektu projevily i v prostředí Homestead.

```
ip: "192.168.10.10"
memory: 2048
cpus: 2
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
    - ~/.ssh/id_rsa

folders:
    - map: C:\www\Laravel
      to: /home/vagrant/code

sites:
    - map: diplomka.test
      to: /home/vagrant/code/diplomka/public
    - map: phpmyadmin.test
      to: /home/vagrant/code/phpmyadmin
```

```
databases:
    - homestead
```

Zde se mapuje adresář, který se bude používat ve Vagrantu. Jedná se o adresář, ve kterém se nachází zdrojový kód celé aplikace.

```
folders:
    - map: C:\www\Laravel
      to: /home/vagrant/code
```

Konfigurace uvedená níže říká, že pokud bude v prohlížeči zadána adresa *diplomka.test*, pak Vagrant poskytne zdroje z adresáře */home/vagrant/code/diplomka/public*. U mapování adresy *phpmyadmin.test* to funguje obdobně, je ale nutné ho nejprve stáhnout a přidat do výše namapované složky (Laravel).

```
sites:
    - map: diplomka.test
      to: /home/vagrant/code/diplomka/public
    - map: phpmyadmin.test
      to: /home/vagrant/code/phpmyadmin
```

Aby bylo možné přistupovat k aplikaci z webového prohlížeče, je potřeba namapovat IP adresu aplikace. Adresu *diplomka.test* je potřeba přidat do souboru *hosts*, který se nachází na adrese *C:Windows\Systems32\drivers\etc\hosts*. Soubor *hosts* je chráněný proti zápisu a je nutné ho otevřít v poznámkovém bloku s administrátorskými právy. Na konec tohoto souboru byl přidán následující řádek:

```
192.168.10.10 diplomka.test
```

Dalším krokem je generování SSH klíče. Tento klíč slouží pro autentifikaci uživatele. Do konzole Gitu je zadán příkaz:

```
ssh-keygen –t rsa –C “emailová adresa“
```

Tento příkaz vygeneroval dvojici SSH klíčů, jeden privátní a druhý veřejný. Po vygenerování popsaných SSH klíčů je instalace a nastavení Homestead prostředí kompletní.

## 9.2 Konfigurace základních cachovacích metod

Jak bylo zmíněno, tato práce se zabývá čtyřmi základními metodami cache – Database, File, Redis, Memcached a jejich nutná konfigurace je popsána v bodech níže.

### 9.2.1 Database

Pro použití databáze jako cache je potřeba vytvořit databázovou tabulku, která bude ukládat data. Na příkladu níže je ukázka použité tabulkové struktury.

```
Schema::create('cache', function ($table) {
            $table->string('key')->unique();
            $table->text('value');
            $table->integer('expiration');
        });
```

Tuto tabulku lze vytvořit ručně pomocí migrací anebo za pomocí příkazu:

```
php artisan cache:table
```

Tento příkaz automaticky vytvoří migraci *cache_table.* Tabulka je v databázi vytvořena po zadání příkazu:

```
php artisan migrate
```

V souboru *.env* je dále změněn cache driver na *database*.

```
CACHE_DRIVER=database
```

### 9.2.2 File

Cachování do souboru je nastaveno jako defaultní, to znamená, že není potřeba instalovat žádné další balíčky či vytvářet databázové tabulky. Nastavení cachování, jak už bylo popsáno, je v souboru *.env*.

```
CACHE_DRIVER=file
```

### 9.2.3 Memcached

#### Instalace

Memcached není linuxová distribuce a neexistuje PHP rozšíření pro Memcached, ale pouze pro zastaralou verzi Memcache, proto bylo zvoleno lokální prostředí Laravel Homestead, které tuto službu integruje. Instalace tohoto prostředí byla popsána v kapitole 9.1.

#### Nastavení Laravelu

V souboru *.env* je nastaven cache driver na *memcached*. Pro spolupráci Laravelu a Memcached již není nutné přidávat žádné další nastavení.

```
CACHE_DRIVER=memcached
```

### 9.2.4 Redis

Redis není linuxová distribuce, proto byl zvoleno lokální prostředí Laravel Homestead, které tuto službu integruje. Instalace tohoto prostředí byla popsána v kapitole 9.3.

#### Nastavení Laravelu

Aby mohl Laravel spolupracovat s Redisem, je nutné nejdříve nainstalovat *predis/predis* balíček pomocí composeru.

```
composer require predis/predis
```

Natavení Redisu se nachází ve složce *config* v souboru *database.php*. Na konci tohoto souboru se nachází pole s označením *redis*, které obsahuje nastavení Redis serveru. Toto nastavení lze níže vidět v defaultním stavu a není nutné ho nijak dále upravovat.

```
'redis' => [

        'client' => 'predis',

        'default' => [
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'password' => env('REDIS_PASSWORD', null),
            'port' => env('REDIS_PORT', 6379),
            'database' => 0,
        ],

    ]
```

Po instalaci *predis* balíčku je do souboru *composer.json* automaticky přidána nová závislost, která je níže zvýrazněna žlutou barvou.

```
"require": {
        "php": ">=7.0.0",
        "fideloper/proxy": "~3.3",
        "laravel/framework": "5.5.*",
        "laravel/tinker": "~1.0",
```

```
        "predis/predis": "^1.1",
        "twbs/bootstrap": "4.3.1"
    },
```

V souboru *.env* je nastaven cache driver na *redis*.

```
CACHE_DRIVER=redis
```

### Testování

Pomocí funkce *print_r* si lze vypsat základní informace. Pokud je *Redis* správně nastaven, po přístupu na url *redistest* budou pomocí funkce *print_r* vypsány základní informace.

```
Route::get('/redistest', function(){
    print_r(app()->make('redis'));
});
```

Kódem popsaným níže je otestovano uložení a načtení hodnoty z cache paměti. Hodnota *testValue* je uložena pod klíčem *key1*, následně je pomocí tohoto klíče uložená hodnota načtena a vypsána.

```
Route::get('/redistest', function(){
    $redis = app()->make('redis');
    $redis->set("key1", "testValue");
    return $redis->get("key1");
});
```

Funkci *Redisu* je možné otestovat i v *git* konzoli. Pro připojení k Redis serveru je zadán příkaz *redis*-cli. Pomocí *keys* * jsou vypsány všechny uložené klíče, následně pomocí metody *GET* a názvu příslušného klíče je vypsána uložená hodnota, v tomto případě *testValue*.

```
127.0.0.1:6379> keys *
1) "key1"
127.0.0.1:6379> Get key1
"testValue"
127.0.0.1:6379>
```

Obrázek 14: Vypsání uložených hodnot v konzoli [Zdroj vlastní]

## 9.3 Popis aplikace

Aplikace se skládá ze dvou částí. První část obsahuje načítání článků, obrázků, uživatelů bez cachovacího algoritmu, kde jsou všechna data brána přímo z databáze, a tudíž je jejich načtení pomalejší. Druhá část aplikace má stejnou strukturu s tím rozdílem, že zahrnuje cachovací algoritmus pro ukládání dat.

V následujících podkapitolách je popsáno, jak funguje vykreslení uživatelů s pomocí cachování a bez něj.

***UserControlleru***

Zde bude popsán příklad funkčnosti UserControlleru.

Metoda *index* zobrazená níže slouží pro základní vykreslení uživatelů. Do proměnné *timeStop* je uložena instance třídy *WatchStop*, která slouží jako dodatečný nástroj po stopování času databázového dotazu.

V proměnné *users* jsou uloženi všichni uživatelé a jejich role. Tito uživatelé jsou seřazeni podle křestního jména. Poté je ukončeno stopování času databázového dotazu. Hodnoty uložené v proměnné *users* jsou předány do šablony pro vykreslení.

```
public function index(){

        $timeStop = new WatchStop('UserController');
        $users = User::with('role')->orderBy('firstname')->where('role_id',
'=', '1')->get();
        $timeStop->endTime();
        return view('users.index', compact('users'));
    }
```

Pro metodu *index* existuje ekvivalent, tím je metoda *indexCache*. Tato metoda vrací do šablony stejný výstup jako metoda předchozí. Pro cachování je zde využita *Repository* vrstva, která je volána jako *singleton* a je přístupná ve všech třídách.

```
  public function indexCache(){
        $timeStop = new WatchStop('UserControllerCache');
        $users = userRepository()->all('firstname');
        $timeStop->endTime();
        return view('users.index', compact('users'));
    }
```

Metoda *storeUser* slouží jen jako ukázka uložení nového uživatele do databáze. Pokud dojde k uložení uživatele do databáze, spustí se *event,* který obsahuje novou instanci třídy *UserTableChanged*. Laravel smaže data uložená v cache paměti a při dalším přístupu k těmto datům se uloží data aktuální.

```
public function storeUser(){

        $user = new User();

        $user->firstname = 'Dominik';
        $user->surname = 'Pfeffer';
        $user->nickname = 'Najky';
        $user->email = 'dominik@email.cz';
        $user->password = bcrypt('dominik');
        $user->role_id = 1;

        $user->save();

        event(new UserTableChanged($user));

    }
```

### UserRepository

*UserRepository* obsahuje dvě metody. Metodu pro získání všech uživatelů a metodu pro získání jednoho konkrétního uživatele podle jeho *ID*.

Metoda *all* slouží pro získání všech uživatelů a zároveň jejich rolí. Metoda *remember* se snaží získat data z cache paměti pod klíčem *users*, a pokud tato data neexistují, načte je z databáze a následně je do cache paměti uloží pod zadaným klíčem. Pro zjednodušení je zde načteno pouze 300 uživatelů a čas expirace cachování je nastaven na 10 minut. Tento čas by se měl odvíjet podle toho, jak často se budou data měnit v databázi. U neměnných dat nebo dat, jejichž aktuálnost není nezbytná, lze nastavit vysoký čas expirace.

```
public function all($orderBy){
        $key = "all.{$orderBy}";
        $cacheKey = $this->getCacheKey($key);
        $users = Cache::remember('users', Carbon::now()-
>addMinutes('10'),function() use($orderBy){
            return User::with('role')->orderBy($orderBy)->where('role_id',
'=', '1')->take(300)->get();
        });
        return $users;
    }
```

Metoda *get* funguje na stejném principu jako metoda *all* s tím rozdílem, že načítá a ukládá do cache paměti jenom jednoho uživatele.

```
public function get($id){
        $key = "get.{$id}";
        $cacheKey = $this->getCacheKey($key);

        $users = Cache::remember($cacheKey, Carbon::now()-
>addMinutes('10'),function() use($id){
            return User::with('role')->where('id', $id)->first();
        });
        return $users;
    }
```

### ClearUserCache

Třída *ClearUserCache* je uložena ve složce *Listeners* a slouží jako tzv. *posluchač*, který se používá k obsluhování událostí. Metoda *handle* reaguje na vytvoření nového uživatele, jako parametr dostává instanci třídy *UserTableChanged*. Pomocí této instance je možné přistupovat k základním atributům vytvořeného uživatele.

Metoda *handle* zde pomocí funkce *forget* vymaže data uložená pod klíčem *users*. Pokud by byl odkomentován vrchní řádek, dojde k vymazání všech dat uložených v cache paměti.

```
/**
     * Handle the event.
     *
     * @param  UserTableChanged  $event
     * @return void
     */
    public function handle(UserTableChanged $event)
    {
        //Cache::flush();
        Cache::forget('users');
    }
```

### Routování

Routování je v Laravelu intuitivní, nachází se v souboru *web.php*. Obecně se dělí na metody *get* a *post*. Metoda *get* slouží k získávání dat a *post* k jejich ukládání. První parametr metod níže je URL adresa, druhý parametr obsahuje Kontroler a jeho metodu.

```
Route::get('/', 'PageController@index');
Route::get('/user', 'UserController@index');
Route::get('/usercache', 'UserController@indexcache');
Route::post('/storeuser', 'UserController@storeuser');
```

### WatchStop

Třída *WatchStop* slouží pro trackování času databázových dotazů v rámci aplikace. Obsahuje dvě metody.

Metoda *endTime* odečítá konečný čas, kdy byli načtení uživatelé s časem před načtením uživatelů. Poté výsledný čas zformátuje na 10 desetinných míst a zavolá metodu pro uložení do databázové tabulky.

```
  public function endTime(){
        $endTime = microtime(true);
        $difference = $endTime - $this->time
        $this->time = number_format($difference, 10);
        $this->saveToDatabase();
    }
```

Metoda níže obsahuje logiku pro uložení hodnot do databázové tabulky *stopwatchs*. Do sloupce *key* se vkládá textový řetězec, který obsahuje název Kontroleru a zda byla metoda cachována či nikoliv. Do sloupce *time* je vložen čas databázového nebo nacachovaného dotazu.

```
private function saveToDatabase(){
        DB::table('stopwatchs')->insert([
            'key' => $this->key,
            'time' => $this->time,
        ]);
    }
```

### Singleton

Poprvé při zavolání metody *singleton* je vytvořená nová instance třídy a je vrácena při každém dalším volání. Zde je ukázka vytvoření *singletonu* pro třídu UserRepository.

```
public function register()
    {
        $this->app->singleton(UserRepository::class, function () {
            return new UserRepository();
        });
    }
```

### Helpers

V *helpers* jsou definovány globální funkce, které je možné volat z jakékoliv třídy. Pro každou *Repository* třídu je vytvořen samostatný *helper*. Zde je uveden příklad pro UserRepository.

```
if (!function_exists('userRepository')) {
    /**
     * Vrací instanci na repository
     * @return \App\Repositories\UserRepository
    */
    function userRepository(): \App\Repositories\UserRepository
    {
        static $repository = null;
        if (null === $repository) {
            $repository = resolve(\App\Repositories\UserRepository::class);
        }
        return $repository;
    }
}
```

# 10 SPRÁVNÉ POSTUPY CACHOVANÍ

Cachovat by se měla pouze data, která se nemění anebo jejich neaktuálnost nezpůsobí žádné větší problémy. Cachovaná data, jako jsou kategorie produktů, zobrazované články, profily uživatelů, nezpůsobí při neaktuálnosti žádnou větší újmu, avšak ceny produktů či podobná vysoce aktuální data by se zpravidla cachovat neměla.

## 10.1 Přístupy ke cachování

Volba správného přístupu ke cachování záleží na programátorovi a jeho zhodnocení. Obecně by se nemělo používat nic komplexnějšího, než je třeba. Cachování dat, která se neopakují na více místech, je možné provést přímo v kontroleru. U dat, která se opakují na více místech a není potřeba u nich cachování nijak dál nastavovat či upravovat za běhu aplikace, je dobré použít návrhový vzor Repository. U komplexnějších aplikací, kde je třeba různého nastavení cachování, je možné použít návrhový vzor typu Dekorátor.

**Repository pattern**

Návrhový vzor repository je použitý v autorem popisované testovací aplikaci. Repository je samostatná, znovupoužitelná třída, která obsahuje veřejně přístupné funkce. V tomto případě funkce slouží pro cachování databázových dotazů.

**Decorator patter**

Návrhový vzor dekorátor se používá pro změnu funkčnosti nějakého objektu bez nutnosti změny tohoto objektu či použití dědičnosti. Jako dekorátor je označována třída, která obdrží nějakou instanci a tuto instanci obohatí o novou funkcionalitu.

Dekorátor se při cachování používá na místech, kdy je potřeba v určitých situacích cachování vypnout nebo zapnout. Níže bude uveden jednoduchý příklad použití návrhového vzoru dekorátor.

Ve třídě *Posts* je metoda *all*. Tato metoda vrací všechny články uložené v databázi.

```
public function all(){
            return  $this->posts->all();
    }
```

Třída dekorátor bude pojmenována např. *CacheablePosts* obsahuje konstruktor, který přijímá jako parametr instancí třídy, již dekoruje. Například články, jak je vidět na příkladu níže.

```
protected $posts;

    public function __construct($posts)
    {
        $this->posts = $posts;
    }
```

Následně obsahuje metodu, která napodobuje funkci, již bude měnit. V této metodě se odkazujeme na funkci, kterou dekorujeme. Zde metodu dekorujeme tím, že k ní přidáme cachovací vrstvu.

```
public function all(){
        return Cache::remember('posts.all', 60*60, function(){
            return  $this->posts->all();
        });
    }
```

Pokud bude potřeba cachovat články vytvoří se nová instance třídy *CacheablePosts*.

```
new CacheablePosts(new Posts);
```

### Cachování přímo v Controlleru

Nejedná se o žádný návrhový vzor. Cachování v kontroleru je nejjednodušší možný způsob cachování. Tato možnost se používá v jednoduchých aplikacích, kde není potřeba komplexního cachování a data jsou cachována pouze na jednom místě. Při cachování stejných dat na více místech musí programátor kopírovat či psát rutinní cachovací kód stále dokola.

```
$posts = Cache::remember('posts', 60*60, function (){
            return Post::all();

        });

        return $posts;
```

# 11 TESTOVÁNÍ

V této kapitole proběhne testování a porovnání časů jednotlivých cachovacích metod. Zvolené metody pro cachování jsou file, database, redis, memcached. Testování databázových dotazů probíhalo pomocí autorem vytvořené třídy WatchStop.

Před samotným testováním byly provedeny změny nastavení databáze a systému Memcached. Databáze nedovoluje spouštět pakety nad určitou velikost, proto byly pomocí konzole provedeny tyto změny:

```
mysql --max_allowed_packet=100M
```

Do jednoho z kontrolerů byl přidán řádek pro zvýšení množství paměti při zpracování skriptů v PHP.

```
ini_set('memory_limit', '2048M');
```

Protože memcached nedovoluje v základním nastavení ukládat data větší než 1MB, byl v bash konzoli přidán příkaz pro zvýšení paměti pro ukládání dat.

```
memcached -I=32M
```

Databáze při testování obsahuje:

- článků: 80 000,
- uživatelů: 100 000,
- komentářů: 10 000,
- kategorií: 9,
- obrázků: 50 000,
- rolí: 3,
- jazyků: 4.

### Posts

Pro cachování jednoduchého dotazu bylo zvoleno načítání 10 000 článků i s uživatelskými údaji.

```
$posts = Post::with('user')->take(10000)->get();
```

Tabulka 2: Srovnání jednotlivých časů – dotaz 1.

| POČET MĚŘENÍ | BEZ CACHE | FILE | DATABASE | REDIS | MEMCACHED |
|---|---|---|---|---|---|
| - | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] |
| NACACHOVÁNÍ | - | 2986,895 | 3152,143 | 1133,116 | 1162,639 |
| 1. | 1127,219 | 167,647 | 102,456 | 197,728 | 180,304 |
| 2. | 1035,061 | 130,467 | 154,083 | 107,345 | 139,532 |
| 3. | 930,308 | 141,920 | 148,324 | 100,046 | 124,437 |
| 4. | 981,941 | 127,817 | 159,746 | 99,217 | 108,525 |
| PRŮMĚRNÝ ČAS | 1018,632 | 141,962 | 141,152 | 126,084 | 138,200 |

Dotaz sloužící pro získání produktů v objednávce pro každého uživatele v určitém časovém rozmezí.

```
        $dayStart = date('2019-05-01 00:00:00');
        $dayEnd = date('2019-05-15 12:25:45');
        $lineitems = DB::table('lineitems')
            ->join('orders', 'orders.id', '=', 'lineitems.order_id')
            ->where('orders.created_at', '>=', $dayStart)
            ->where('orders.created_at', '<', $dayEnd)
            ->join('customers', 'customers.id', '=', 'orders.customer_id')
            ->where('customers.country', '=', 'US')
            ->select('product_id', 'price_per_unit', 'quantity', 'or-
ders.created_at', 'customers.firstname', 'customers.lastname',
DB::raw('SUM(price_per_unit) as total_price'))
            ->groupBy('lineitems.id')
            ->get();
```

Tabulka 3: Srovnání jednotlivých časů – dotaz 2.

| POČET MĚŘENÍ | BEZ CACHE | FILE | DATABASE | REDIS | MEMCACHED |
|---|---|---|---|---|---|
| - | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] |
| NACACHOVÁNÍ | - | 377,164 | 790,004 | 424,623 | 388,266 |
| 1. | 340,402 | 72,093 | 105,769 | 94,568 | 71,019 |
| 2. | 318,748 | 95,208 | 126,051 | 91,401 | 73,744 |
| 3. | 269,114 | 80,108 | 86,671 | 80,026 | 58,763 |
| 4. | 256,664 | 99,416 | 104,471 | 80,945 | 62,989 |
| PRŮMĚRNÝ ČAS | 296,232 | 86,706 | 105,741 | 86,735 | 66,628 |

Načtení 1500 článků s informacemi o uživateli a jazyku, v jakém jsou psány.

```
Post::with('user')->with('language')->take(1500)->get();
```

Tabulka 4: Srovnání jednotlivých časů – dotaz 3.

| POČET MĚŘENÍ | BEZ CACHE | FILE | DATABASE | REDIS | MEMCACHED |
|---|---|---|---|---|---|
| - | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] |
| NACACHOVÁNÍ | - | 1161,102 | 2038,444 | 1016,777 | 1066,974 |
| 1. | 1027,076 | 147,714 | 145,166 | 145,250 | 154,839 |
| 2. | 916,520 | 130,798 | 197,765 | 135,292 | 140,110 |
| 3. | 997,075 | 177,057 | 223,201 | 151,488 | 120,557 |
| 4. | 971,675 | 166,792 | 146,859 | 142,606 | 105,812 |
| PRŮMĚRNÝ ČAS | 296,232 | 155,590 | 178,247 | 143,659 | 130,329 |

**Users**

Načítání uživatelů i s uživatelskými rolemi, řazení uživatelů podle křestního jména, kde je role větší než 1.

```
$users = User::with('role')->orderBy('firstname')->where('role_id', '>',
'1')->take(600)->get();
```

Tabulka 5: Srovnání jednotlivých časů – dotaz 4.

| POČET MĚŘENÍ | BEZ CACHE | FILE | DATABASE | REDIS | MEMCACHED |
|---|---|---|---|---|---|
| - | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] | Čas[ms] |
| NACACHOVÁNÍ | - | 580,819 | 623,463 | 566,244 | 491,973 |
| 1. | 545,922 | 83,349 | 78,567 | 117,249 | 70,135 |
| 2. | 468,510 | 58,036 | 131,244 | 74,532 | 47,686 |
| 3. | 462,519 | 48,733 | 128,322 | 69,071 | 78,489 |
| 4. | 456,286 | 47,499 | 84,199 | 77,521 | 49,594 |
| PRŮMĚRNÝ ČAS | 483,309 | 59,404 | 105,583 | 84,593 | 61,476 |

## 11.1 Shrnutí

Memcached i Redis běžely na jednom stroji spolu s testovací aplikací v prostředí Laravel Homestead, a i když je Homestead oficiální lokální prostředí uzpůsobené pro testování aplikací, mohl zapříčinit občasné pomalejší reakce těchto služeb. Pokud by tyto služby běžely zvlášť na samostatném serveru, výsledky by byly znatelně lepší. Aplikace a veškeré systémy běží na autorově notebooku.

V minulé podkapitole byly znázorněny tabulky času cachování při použití různých cachovacích ovladačů. Z tabulek vyplývá, že nejvíce času pro nacachování potřebuje databáze a zároveň je při načítání dat nejpomalejší. Časy souborové cache byly v některých případech téměř srovnatelné s časy Redis, velkou zásluhou na tom má rychlý SSD disk. Na klasickém pevném disku by časy byly znatelně pomalejší.

Memcached je v některých případech efektivnější, ale ze zkušenosti programátorů je většinou lepší použít Redis. Obě tyto služby jsou uložiště typu hodnota-klíč a patří mezi NoSQL databáze, které ukládají data v RAM paměti.

Memcached doporučuji použít při cachování relativně malých a statických dat, protože základní nastavení umožňuje cachovat pouze soubory velikosti 1MB. Memcached podporuje z datových typů pouze řetězce a ty jsou ideální pro cachování pouze dat, která jsou určena ke čtení. Memcached je *volatile store*, to znamená, že po restartu serveru nebo počítače mohou být uložená data ztracena. Memcached velmi dobře pracuje s webovými stránkami s vysokou návštěvností, jelikož dokáže číst velké množství informací najednou s malou odezvou.

Redis dává vývojářům mnohem více možností, hlavně co se týká objektů, které dokáže cachovat. Dále se Redis lépe hodí pro práci s komplexními daty, protože umožňuje cachovat mnohem více datových typů než Memcached. Redis je *non-volatile store* a synchronizuje cachovaná data na disk, to má za následek, že po restartu serveru nebo počítače zůstanou data stále přístupná.

## 11.2 Další možnosti zrychlení aplikace

Laravel nabízí i další možnosti zrychlení výkonu aplikace. Mezi tyto možnosti patří cachování konfigurace aplikace, cachování route, optimalizace mapování apod.

**Cachování konfigurace**

Po nacachování konfigurace aplikace nemají provedené změny na běh aplikace žádný vliv. Toto cachování poskytuje znatelné zvýšení výkonnosti aplikace. Příkaz pro nacachování konfigurace aplikace:

```
php artisan config:cache
```

Příkaz pro vymazání konfigurace z cache paměti:

```
php artisan config:clear
```

### Cachování route

Cachování route (cest) je další základní možností optimalizace výkonu. Toto platí především u webových stránek s mnoha routami. Funguje to tak, že jsou routy ukládány do jednoduchého pole, ze kterého jsou čteny rychleji. Příkaz pro cachování route je:

```
php artisan route:cache
```

Po každé změně route či konfiguračního souboru je nutné spustit příkaz pro vymazání dat z mezipaměti a poté je znovu nacachovat, jinak dojde k načtení starých dat. Příkaz pro mazání cachovaných route:

```
php artisan route:clear
```

# ZÁVĚR

Cílem diplomové práce bylo prozkoumat možnosti cachování ve vývojovém frameworku Laravel a otestovat je za použití testovací aplikace.

Práce se skládá ze dvou částí. V první části byly popsány základní pojmy a vlastnosti frameworku Laravel. Byly charakterizovány základní návrhové vzory, které se používají u webových aplikací. Konkrétněji byla popsána MVC architektura, jelikož je nedílnou součástí frameworku Laravel. Samostatná kapitola se věnuje přímo frameworku Laravel a popisuje jeho historii, vlastnosti a základní možnosti cachování. Další z kapitol obsahuje základní pojmy v oblasti cachování a jeho funkcí. Závěr teoretické části se zabývá NoSQL databází, konkrétně databází typu klíč-hodnota. Podkapitoly se věnují nejpoužívanějším cachovacím systémům využívajícím datovou strukturu typu klíč-hodnota, konkrétně systému Redis a Memcached.

V praktické části byly popsány technologie, které byly použity pro tvorbu testovací aplikace. Dále byly představeny použité nástroje a služby. Popsány byly základní cachovací metody ve frameworku Laravel. Nedílnou součástí je i popis funkčních požadavků na testovací aplikaci. Byly nastíněny správné cachovací postupy. Před samotnou tvorbou aplikace je popsána instalace lokálního prostředí, které bylo důležité pro běh všech systémů sloužících pro cachování. Pomocí ukázky kódu byly popsány nejdůležitější části aplikace, jako je například tvorba Repository pro cachování, nastavení cachovacích ovladačů, třída pro stopování času cachování jednotlivých systémů a způsobů či zachytávání databázových změn. Testování rychlosti cachovacích systémů proběhlo v samostatné kapitole. Dále byly popsány i další možnosti cachování, jako je cachování konfigurace a routování.

Na základě testování cachování bylo shledáno, že pro cachování větších datových struktur je nejlepší použít službu Redis. Pro ukládání větších datových struktur nad 1MB je nutné u Memcached dodatečné nastavení, avšak při cachování malých souborů byly časy téměř srovnatelné. Ukládání dat do souborové cache paměti běžící na serveru s SSD diskem bude téměř srovnatelné s externími cachovacími systémy, avšak načítání z klasického pevného disku by bylo značně pomalejší. Tento způsob doporučuji, pokud je třeba rychle nasadit cachování na jednom serveru a nejsou k dispozici další paměti RAM. Cachování pomocí databáze bylo nejpomalejší. Nehodí se pro cachování velkých objemů dat, protože MySQL server dokáže spustit databázové dotazy jen omezené velikosti a s rostoucí velikostí cachovaného souboru se snižuje rychlost jeho získání. Obecně je cachování použité pro elimina-

ci databázových dotazů, a proto tento způsob nedoporučuji. Autorem doporučovaný způsob cachování je pomocí systémů Redis, avšak zdůrazňuje, že volba je na každém programátorovi po zhodnocení požadavků samotné aplikace.

# SEZNAM POUŽITÉ LITERATURY

[1] What is Framework. Code project [online]. United States, 2003 [cit. 2019-03-03]. Dostupné z: https://www.codeproject.com/Articles/5381/What-Is-A-Framework

[2] What is Framework. Code project [online]. United States, 2003 [cit. 2019-03-03]. Dostupné z: https://www.codeproject.com/Articles/5381/What-Is-A-Framework

[3] MVC architektura. IT network [online]. Česká republika, 2014 [cit. 2019-04-15]. Dostupné z: https://www.itnetwork.cz/navrh/mvc-architektura-navrhovy-vzor/

[4] Prezentační vzory z rodiny MVC. Zdroják.cz [online]. Česká republika, 2009 [cit. 2019-04-15]. Dostupné z: https://www.zdrojak.cz/clanky/prezentacni-vzory-zrodiny-mvc/

[5] Cache Definition and Explanation. Keycdn [online]. United States, 2018 [cit. 2019-04-21]. Dostupné z: https://www.keycdn.com/support/cache-definition-explanation

[6] Kešovací návod. Jak psát web [online]. Česká republika, 2004 [cit. 2019-04-21]. Dostupné z: https://www.jakpsatweb.cz/clanky/caching-tutorial-czech-translation.html

[7] Cache. Laravel [online]. United States, 2019 [cit. 2019-04-21]. Dostupné z: https://laravel.com/docs/5.8/cache

[8] Laravel 5: Cache – General. Jeff's Refference [online]. United States, 2017 [cit. 2019-04-21]. Dostupné z: http://laravel.at.jeffsbox.eu/laravel-5-cache-general

[9] Introduction to Redis. Redis [online]. United States, 2019 [cit. 2019-04-21]. Dostupné z: https://redis.io/topics/introduction

[10] About Memcached. Memcached [online]. United States, 2019 [cit. 2019-04-21]. Dostupné z: https://memcached.org/about

[11] VMware: the new Redis home. Antirez weblog [online]. United States, 2010 [cit. 2019-04-21]. Dostupné z: http://oldblog.antirez.com/post/vmware-the-new-redis-home.html

[12] Overview. Github [online]. United States, 2019 [cit. 2019-04-21]. Dostupné z: https://github.com/memcached/memcached/wiki/Overview

[13] NoSQL Databases: a Survey and Decision Guidance. Medium [online]. United States, 2016 [cit. 2019-04-23]. Dostupné z: https://medium.baqend.com/nosql-databases-a-survey-and-decision-guidance-ea7823a822d

[14] STAUFFER, Matt. Laravel: up and running: a framework for building modern PHP apps. Sebastopol, CA: O'Reilly Media, 2016. ISBN 9781491936085.

[15] Design patterns in PHP and Laravel. New York, NY: Springer Science+Business Media, 2016. ISBN 9781484224502.

[16] SOLIMAN, Ahmed. Getting Started with Memcached. Birmingham: Packt Publishing, 2013. ISBN 978-1-78216-322-0.

[17] DAS, Vinoo. Learning Redis. Birmingham: Packt Publishing, 2015. ISBN 978-1783980123.

[18] BEAN, Martin. Laravel Essentials. Birmingham: Packt Publishing, 2015. ISBN 9781785283017.

[19] Laravel [online]. USA:Taylor Otwell, 2018 [cit.2019-04-23]. Dostupné z: https://laravel.com/

[20] DAYVSON DA SILVA, Maxwell. Redis Essentials. Birmingham: Packt Publishing, 2015. ISBN 978-1784392451.

[21] MALATESTA, Francesco. Learning Laravel's Eloquent. Birmingham: Packt Publishing, 2015. ISBN 978-1784391584.

[22] Web Cache - a Simple Guide. Keycdn [online]. USA, 2018 [cit. 2019-04-25]. Dostupné z: https://www.keycdn.com/support/web-cache

[23] BÖHMER, Marian. Návrhové vzory v PHP: [23 vzorových postupů pro rychlejší vývoj]. Brno: ComputerPress, 2012. ISBN 978-80-251-3338-5.

[24] Seriál návrhových vzorů – 1. díl. Programujte.com [online]. Czech republic, 2012 [cit. 2017-05-04]. Dostupné z: http://programujte.com/clanek/2012032900-serial-navrhovych-vzoru-1-dil/

[25] History of Laravel. W3schools [online]. USA, 2019 [cit. 2019-04-25]. Dostupné z: https://www.w3schools.in/laravel-tutorial/history/

[26] Laravel release process. W3schools [online]. USA, 2019 [cit. 2019-04-25]. Dostupné z: https://laravel-news.com/laravel-release-process [2

[27] Historie a trendy ve vývoji databází. Webová integrace [online]. Česká republika, 2012 [cit. 2019-05-21]. Dostupné z: http://www.web-integration.info/cs/blog/historie-a-trendy-ve-vyvoji-databazi/

# SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

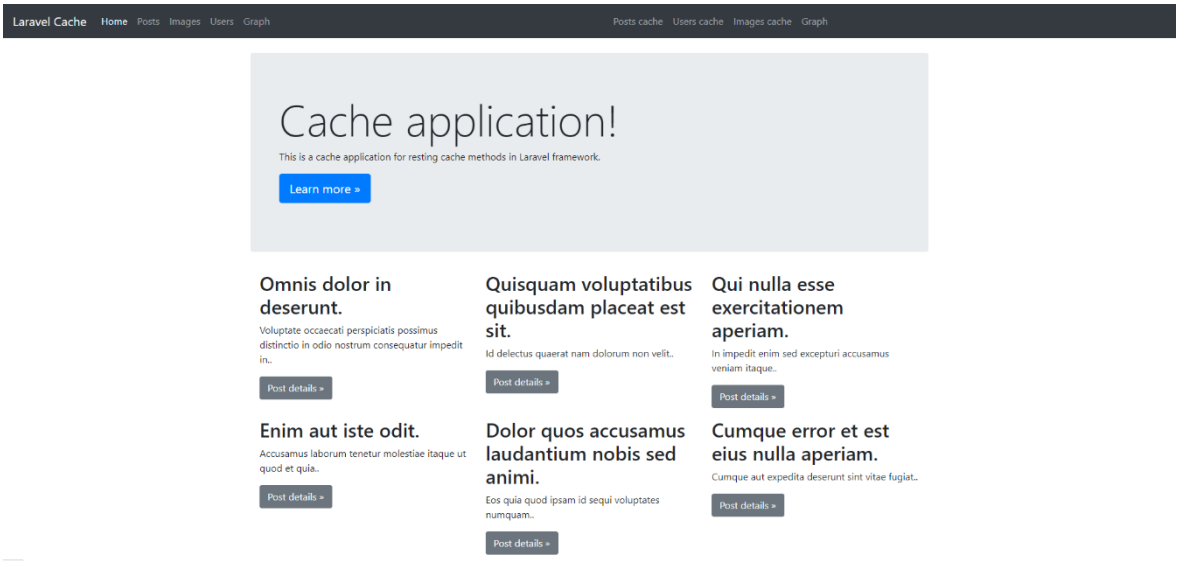
| | |
|---|---|
| PHP | Hypertextový preprocesor. |
| MVC | Model-view-controller. |
| URL | Uniform Resource Locator. |
| ID | Identification Number. |
| API | Application Programming Interface. |
| IDE | Inregrated Development Environment. |
| NoSQL | Not only SQL. |
| HTML | Hyper Text Markup Language. |
| RAM | Random-Access-Memory. |
| SRAM | Static-Random-Access-Memory. |
| CDN | Content Delivery Network. |
| SSD | Solid-state drive |
| SDS | Simple dynamic string |

# SEZNAM OBRÁZKŮ

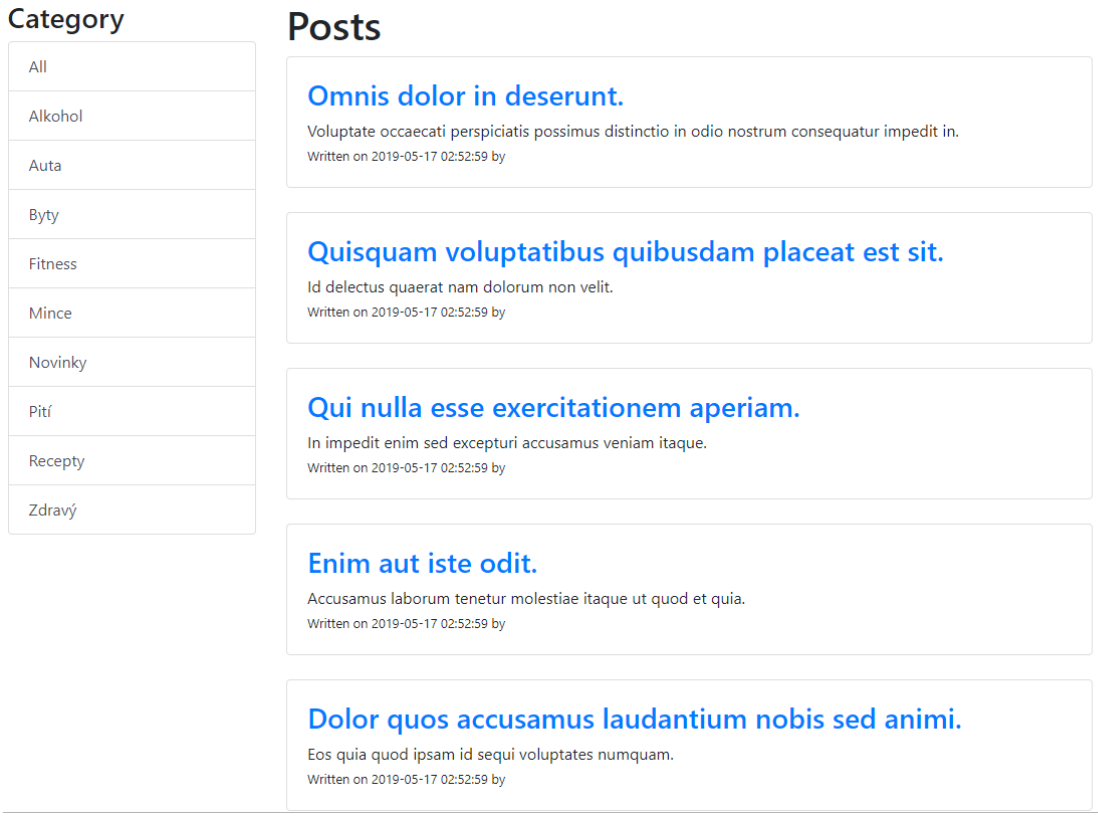
# SEZNAM TABULEK

# SEZNAM PŘÍLOH

# PŘÍLOHA P I: OBRÁZEK ÚVODNÍ STRANY TESTOVACÍ APLIKACE



# PŘÍLOHA P II: OBRÁZEK VÝPISU ČLÁNKŮ



# PŘÍLOHA P III: OBRÁZEK VÝPISU UŽIVATELŮ APLIKACE

| # | Firstname | Lastname | Nickname | Email | Role |
|---|---|---|---|---|---|
| **1** | Aaliyah | Schuster | auer.gilbert | wkassulke@example.org | Admin |
| **2** | Aaliyah | McLaughlin | buckridge.marta | lydia.dach@example.org | Editor |
| **3** | Aaliyah | Greenfelder | alessia68 | catharine30@example.org | Editor |
| **4** | Aaliyah | Frami | tatum.bergstrom | johnson.eloise@example.com | Editor |
| **5** | Aaliyah | Shields | jan10 | zieme.juana@example.net | Admin |
| **6** | Aaliyah | Schowalter | ruby26 | flossie.schneider@example.com | Editor |
| **7** | Aaliyah | West | fidel86 | johnathan.wisozk@example.net | Admin |
| **8** | Aaliyah | Stokes | reynolds.ova | elwin.flatley@example.com | Editor |
| **9** | Aaliyah | Mills | jamarcus62 | zulauf.garnett@example.com | Admin |
| **10** | Aaliyah | Roberts | ckoelpin | elta50@example.org | Admin |

# PŘÍLOHA P IV: OBRÁZEK VÝPISU OBRÁZKŮ APLIKACE