FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Rozšiřitelná architektura pro framework Laravel.
Student:	Bc. Tomáš Novotný
Vedoucí:	Ing. Lukáš Janeček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je vytvořit co nejlepší objektový návrh pro vývoj webových aplikací, kde by byly jasně definované zodpovědnosti, při dodržení principů čistého kódu (SOLID, GRASP, ...). Výstupem bude metodika pro tvorbu architektury modulárních webových aplikací, která umožní snadnou udržovatelnost a rozšiřitelnost aplikace a zlepší její přehlednost.

Tato metodika bude demonstrována při návrhu sady modulů pro PHP framework Laravel, které přidají funkcionalitu ecommerce aplikace a zároveň umožní plnou rozšiřitelnost aplikace.

Pokyny:
1) Analyzujte stávající webové frameworky a existující moduly pro Laravel a popište jejich nevýhody.
2) Navrhněte architekturu aplikace tak, aby umožňovala snadnou konfigurovatelnost, rozšiřitelnost a udržovatelnost.
3) Navrhněte sadu modulů pro podporu ecommerce ve frameworku Laravel podle navržené metodiky.
4) Implementujte řešení a vytvořte jeho dokumentaci.
5) Řešení otestujte a srovnejte jeho architekturu z pohledu rozšiřitelnosti a přizpůsobitelnosti.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 24. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Rozšiřitelná architektura pro framework Laravel

Bc. Tomáš Novotný

Katedra softwarového inženýrství
Vedoucí práce: Ing. Lukáš Janeček

7. května 2018

Poděkování

Děkuji všem, kteří se mnou měli trpělivost při tvorbě této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 7. května 2018 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Tomáš Novotný. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Novotný, Tomáš. *Rozšiřitelná architektura pro framework Laravel.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je vytvořit co nejlepší objektový návrh pro vývoj webových aplikací, kde by byly jasně definované zodpovědnosti, při dodržení principů čistého kódu (SOLID, GRASP, ...). Výstupem bude metodika pro tvorbu architektury modulárních webových aplikací, která umožní snadnou udržovatelnost a rozšiřitelnost aplikace a zlepší její přehlednost.

Klíčová slova Laravel, PHP, webová aplikace, modulární architektura

Abstract

The aim of the thesis is to design and implement a set of modules for PHP framework Laravel, which add functionality of the ecommerce application. Architecture design must allow full scalability of the applications that will use these modules.

Keywords Laravel, PHP, web application, modular architecture

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Při vývoji webové aplikace je velmi snadné dostat se do stavu, kdy je zdrojový kód nepřehledný a je náročné přidávat nové funkcionality či opravovat stávající kód – a to mohl projekt při spuštění dodržovat všechny doporučované postupy a principy čistého kódu.

Jeden z nejčastějších případů je aplikace, která je plánována jako jednoduchý krátkodobý projekt. Hlavním cílem je vyvinout aplikaci co nejrychleji a co s nejmenšími náklady. Aplikace tak neobsahuje žádnou složitou logiku ani nepracuje s velkým množstvím dat. Postupem času jsou opravovány nalezené chyby a přidávány drobné úpravy. Tyto úpravy jednotlivě nutně nemusejí porušovat pravidla pro psaní kvalitního kódu. Pokud neustále přicházejí nové požadavky, může tento vývoj trvat delší dobu, než bylo předpokládáno. Aplikace se dostane do stavu, kdy začíná být nepřehledná a jakákoli další úprava je čím dál tím více časově náročná. Je tak nutné provést celkový refaktoring, nebo v krajním případě dojde ke přepsání celé aplikace. Pokud ovšem není změněn způsob vývoje aplikace, dostane se časem do stejně špatného stavu jako byla před refaktorizací.

Nemusí se jednat jenom o původně malé projekty, ale i prvoplánově rozsáhlý projekt. Tento projekt mohl projít předimplementační analýzou a obsahuje všechny zjištěné požadavky. Objevují se ovšem nové uživatelské požadavky, které vyžadují rozšíření stávajícího řešení, či přepracovaní části aplikace. Následné opakované úpravy vedou ke stejnému závěru, kterým je nepřehledný kód.

Cena razatní změny (např. změna datového modelu) roste úměrně s dobou běhu projektu. Vždy je výhodnější provést náročné rozhodnutí o návrhu aplikace nazačátku vývoje, než v pozdějších fázích. Správný návrh a nastavení pravidel, a jejich dodržování již od začátku vývoje, je tak nejdůležitější předpoklad úspešného dlouhodobého vývoje. Zejména, když 40–80 % doby životnosti aplikace je stráveno údržbou [1].

Ve správném objektovém návrhu je nutné předpokládat, že po implementaci přijdou nové požadavky. Samotná předimplementační analýza často ne-

odhalí všechny potřebné funkcionality, či jsou zjišteny jiné požadavky až po skutečném spuštění aplikace v produkčním prostředí. To ovšem neznamená, že je nutné již od začátku projektu vyvíjet aplikaci s podporou všech možností, které by mohly být vyžadovány. To by odporovalo principu KISS[1]. Naopak je nutná dobře navržená architektura, která umožňuje pohodlné zapracování nových požadavků. Také je nutné vyvíjet systém tak, aby splňoval principy SOLID [2], hlavně princip otevřenosti a uzavřenosti. Jejich dodržení napomáhá snadné implementaci změn při zachování vysoké kvality kódu.

Definice cílů

Cílem práce je vytvořit co nejlepší objektový návrh pro vývoj webových aplikací, kde by byly jasně definované zodpovědnosti, při dodržení principů čistého kódu (SOLID [2], GRASP [3], ...). Výstupem bude metodika pro tvorbu architektury modulárních webových aplikací, která umožní snadnou udržovatelnost a rozšiřitelnost aplikace a zlepší její přehlednost.

Tato metodika bude demonstrována při návrhu sady modulů pro PHP framework Laravel, které přidají funkcionalitu ecommerce aplikace a zároveň umožní plnou rozšiřitelnost aplikace.

- Analyzujte stávající webové frameworky a existující moduly pro Laravel a popište jejich nevýhody.
- Navrhněte architekturu aplikace tak, aby umožňovala snadnou konfigurovatelnost, rozšiřitelnost a udržovatelnost.
- Navrhněte sadu modulů pro podporu ecommerce ve frameworku Laravel podle navržené metodiky.
- Implementujte řešení a vytvořte jeho dokumentaci.
- Řešení otestujte a srovnejte jeho architekturu z pohledu rozšiřitelnosti a přizpůsobitelnosti.

[1] *Keep It Simple, Stupid!* – Většina systémů pracuje nejlépe, pokud je udržována spíše jednodušší nežli komplexnější.

Kapitola 1

Představení frameworku Laravel

Cílem této kapitoly je představit, proč je vhodné při vytváření vlastní webové aplikace využít některý z webových frameworků a vysvětlit, proč byl pro implementační část této práce vybrán framework Laravel. Dále poukázat na výhody použití frameworku a dalších dostupných balíčků při vývoji vlastní aplikace a proč může být výhodné zveřejnit vlastní projekt pod svobodnou licencí na některé z webových služeb pro správu projektů.

1.1 Otevřený software

Při vývoji vlastní aplikace budou prakticky vždy využívány knihovny, moduly či celé frameworky vytvořené jiným vývojářem či společností. Tyto moduly bývají veřejně dostupné jako tvz. *open source*[2] na webových službách jako GitHub[3], Packagist[4] a dalších.

Využití těchto projektů ve vlastní aplikaci přináší mnoho výhod a dovolí vyvíjet aplikaci rychleji. Není totiž nutné implementovat a spravovat již jednou (i vícekrát) vytvořený kód. Ten je navíc ucelený a řádně otestovaný.

Vývoj webové aplikace se tak v dnešní době neobejde bez využití nějakého dostupného frameworku [4]. Ten pomáhá urychlit vývoj převzetím odpovědnosti typických problémů dané oblasti (u webových aplikací např. routování, připojení k databázi, apod.). Je tak možné se soustředit pouze na vlastní zadání nad rámec poskytované funkcionality.

Stejně tak zveřejnění vlastní aplikace pod svobodnou licencí pomáhá vývoji lepšího kódu [5]. Veřejný kód nás bude nutit psát lepší a přehledný kód, který obsahuje jak automatické testy, tak dostatečnou dokumentaci. U projektů s velkou komunitou jsou mnohem rychleji zjišťovány případné chyby, než v případě interního testování. Aktivní komunita také navrhuje případná

[2] Svobodný software, který zaručuje uživatelům svobodu jej spouštět, kopírovat, distribuovat, studovat, měnit a zlepšovat.

[3] `https://github.com/`

[4] `https://packagist.org/`

zlepšení, a to dobrovolně a zdarma. To nemůže poskytnout žádná společnost při interním vývoji.

1.2 Framework Laravel

Framework lze popsat jako sadu nástrojů, knihoven, konvencí a osvedčených postupů, které vytváří abstrakci nad rutinními úkoly do obecných modulů, které mohou být lehce znovu využity [6].

Jedním z nejoblíbenějších moderních frameworků je Laravel [7] [8] [9], ten je použit v implementační části této práce. Oproti jiným frameworkům nabízí mnoho nadstandardních funkcí (např. autentizace, autorizace, a další). Díky těmto funkcím je tak možné ihned po instalaci aplikaci používat.

1.2.1 Základní informace

Laravel je PHP[5] framework pro webové aplikace. Autorem frameworku Laravel je sofwarový inženýr Taylor Otwell z amerického Arkansasu [10]. Od roku 2011, kdy byla publikována první demoverze, do konce roku 2014 byl vývoj frameworku Laravel pro Otwella jeho vedlejší činností. Od ledna 2015 se již věnuje dalšímu vývoji frameworku na plný úvazek [11].

Laravel se snaží především o čistý a výstižný kód s přehlednou vnitřní strukturou. To umožňuje i začínajícím programátorům, kteří nikdy předtím žádný framework nepoužívali, pustit se ihned do vývoje vlastní webové aplikace. Zároveň však umožnuje vytvořit vlastní složité návrhové vzory zkušenějšími uživateli pro rozsáhlé robustní projekty. Tuto snahu popisuje i oficiální text, kterým se framework Laravel prezentuje [12].

Laravel je webový aplikační framework s výstižnou a elegantní syntaxí. Věříme, že vývoj aplikací může být zábavná a tvůrčí činnost, která vývojáře opravdu baví. Laravel se snaží obtížnosti u vývoje ulehčit zjednodušením běžných funkčností většiny webových projektů, jako je např. autentizace, routování, práce se session nebo kešování.

Cílem frameworku Laravel je, aby byl proces vývoje aplikací příjemný pro vývojáře, aniž by to negativně ovlivnilo jejich funkčnost. Šťastní vývojáři tvoří ten nejlepší kód. Pro tento účel jsme se snažili zkombinovat to nejlepší, co jsme se naučili u ostatních frameworků, a to bez ohledu na programovací jazyk, jako např. Ruby on Rails, ASP.NET MVC a Sinatra.

Laravel je dostupný, ale výkonný framework, poskytuje nástroje pro velké robustní aplikace. Vynikající IoC[6] container, přehledný migrační systém a pevně intergrovaná podpora pro jednotkové testování jsou nástroje, které potřebujete pro vývoj jakékoliv aplikace, kterou vytváříte.

[5] PHP: Hypertext Preprocessor

[6] *Inversion of Control* – Obrácené řízení je návrhový vzor, který umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami.

1.2.2 Historie

Robustní framework, který je dobře otestovaný a připravený pro vývoj aplikací pro produkční nasazení, vyžaduje značnou dobu pro svůj vývoj. Oproti svým konkurentům je Laravel poměrně mladý framework. Jeho vývoj postupoval odlišněji než u ostatních zavedených frameworků a v poněkud rychlejším tempu, než je obvyklé (viz tab. 1.1).

Tabulka 1.1: Historie frameworku Laravel

verze	datum vydání
Laravel 1.0	20. června 2011
Laravel 2.0	24. listopadu 2011
Laravel 3.0	22. února 2012
Laravel 4.0	28. května 2013
Laravel 4.1	12. prosince 2013
Laravel 4.2	1. června 2014
Laravel 5.0	4. února 2015
Laravel 5.1	9. června 2015
Laravel 5.2	21. prosince 2015
Laravel 5.3	23. srpna 2016
Laravel 5.4	24. ledna 2017
Laravel 5.5	30. srpna 2017
Laravel 5.6	7. února 2018

V srpnu v roce 2009 byla vydána nová verze PHP 5.3, která mimo jiné přinesla podporu pro jmenné prostory (*namespaces*) či možnost volání anonymních funkcí (*closures*). Tyto nové funkce umožnily vývojářům psát více a lépe objektově orientované PHP (OOP[7]) aplikace. Ačkoliv nová verze PHP poskytovala mnoho výhod, ne všechny zavedené frameworky se na ně zaměřily. Místo toho se soustředily na podporu starších verzí PHP. V této době se na seznamu frameworků nejvíce vyskytovaly Symfony, Zend, Slim micro framework, Kohana, Lithium a CodeIgniter. Z nich byl pravděpodobně nejvíce známý PHP framework CodeIgniter. Vývojáři ho upřednostňovali pro jeho obsáhlou dokumentaci a jednoduchost. Kterýkoliv PHP programátor s ním mohl velmi rychle začít vyvíjet aplikaci. Framework měl také okolo sebe velkou komunitu vývojářů a velmi dobrou podporu od svých tvůrců.

V roce 2011 ovšem ve frameworku CogeIgniter sházely funkčnosti, které Taylor Otwell považoval za základní při budování webové aplikace, jako například zabudovaná autentizace, či routování pomocí annonymních funkcí (*closure routing*). A tak byla 9. června 2011 vydána první beta verze frameworku Laravel 1. Podle svého autora byl tento projekt vytvořen pouze proto, aby vyřešil rostoucí obtížnost v používání frameworku CogeIgniter [13].

[7] *Object-oriented programming* – Objektově orientované programování

Laravel 5.1, vydaný v červnu 2015, byl první verzí frameworku Laravel s dlouhodobnou podporou LTS[8] s naplánovanou opravou chyb po dva roky, a opravou bezpečnostních chyb po tři roky. Také bylo rozhodnuto vydávat dlouhodobě podporované verze Laravelu každé dva roky. Aktuální nejnovější verze s podporou LTS je Laravel 5.5.

Aktuální stabilní vydání frameworku Laravel je 5.6.12[9].

1.2.3 Dokumentace

Oficiální webové stránky dokumentace pro framework Laravel se nacházejí na adrese `https://laravel.com/docs`.

Laravel má take vlastní vzdělávací videoportál Laracast [14]. Nachazí se zde video návody k frameworku Laravel a k programovaní v jazyce PHP obecně. Některé návody jsou uveřejněny zdarma, ostatní jsou pak k dispozici za měsíční poplatek. Autorem webu a také hlavní autor většiny příspěvků je lektor Jeffrey Way.

1.2.4 Licence

Laravel je *open source* PHP framework pro webové aplikace distibuovaný pod licencí MIT, která umožňuje volně využívat software, s jedinou podmínkou – zahrnutí textu licence do všech kopií softwaru [15].

1.2.5 Styly pro psaní kódu

Laravel dodržuje standardy PSR–0 [16], PSR–1 [17] od verze Laravel 4 a standard PSR–2 [18] od verze Laravel 5.1.

[8] *Long-term support* – dlouhodobá podpora softwaru

[9] Informace je aktuální ke dni 25. dubna 2018.

Kapitola 2

Analýza existujících řešení

Cílem této kapitoly je analyzovat stávající webové frameworky a existující řešení pro framework Laravel. U jednotlivých analyzovaných řešení jsou popsány jejich nevýhody, kterých se bude implementační část snažit vyvarovat.

2.1 Frameworky

2.1.1 Symfony

Stabilita, konzistence a modularita jsou typické důvody, proč bývá framework Symfony populární mezi vývojáři. Komponenty tohoto frameworku jsou využívány v mnoha dalších projektech – včetně frameworku Laravel.

Navzdory kvalitní dokumentaci má horší křivku učení než framework Laravel. Problémem je vyšší požadavek na režii při konfiguraci (např. manuální konfigurace závislostí *dependency injection*[10]). To dělá Laravel preferovanější volbou.

2.1.2 Phalcon

Framework nabírající na popularitě, hlavně díky své nepřekonatelné rychlosti [19]. Nejedná se ovšem o klasický PHP MVC[11] framework instalovaný pomocí běžně používaného manažera závislostí Composer[12]. Místo toho se jedná o kompilované PHP rozšíření založené na jazyce C. Bohužel velká rychlost je jediná silná stránka ve které framework Falcon vyniká nad ostatními frameworky. Implementace samotné aplikace by byla zbytečné obtížná.

[10] Technika pro vkládání závislostí mezi jednotlivými komponentami programu tak, aby jedna komponenta mohla používat druhou, aniž by na ni měla v době sestavování programu referenci.

[11] *Model–view–controller* – třívrstvá architektura

[12] Nástroj pro správu závislostí v jazyku PHP – `http://getcomposer.org`

2.1.3 Laravel

Implementační část je postavena na frameworku Laravel, který byl představen v kapitole 1.2. Tento framework sám o sobě poskytuje všechny potřebné funkcionality k okamžitému nasazení. Proto je preferovaný pro tvorbu malých projektů, které plně využívají jednoduchého a rychlého vývoje.

Aplikace využívající framework Laravel nejsou typicky používány pro dlouhodobé projekty. Je tak nutné navrhnout vlastní architekturu, která bude podporovat udržitelnou modulární strukturu.

2.2 Existující řešení

2.2.1 Laravel Modules

Tento balíček pro framework Laravel poskytuje funkcionalitu pro správu modulů. Umožňuje jejich automatické načtení do aplikace[13] jako by se jednalo o klasickou závislost definovanou manažerem závislostí Composer. Obdobná funkcionalita, vhodná pro správu modulů, bude implementovaná v aplikaci.

2.2.2 Shopsys Framework

Škálovatelná e-commerce aplikace od firmy Shopsys, která je postavená na frameworku Symfony. Navdory svému jménu se nyní jedná spíše o e-commerce aplikaci než o framework (projekt je nazačátku svého vývoje). Veškeré zdrojové kódy jsou veřejně dostupné na portálu GitHub[14]. Na počátku vývoje nové verze této aplikace bylo zamýšleno ji předělat na plně modulární aplikaci. Nakonec byl preferován vývoj v jediném repozitáři, hlavně z důvodů jednodušší správy repozitářů a možnosti rychlejšího vývoje [20]. Aplikace se sice skládá z jednotlivých komponent, ale nejedná se o samostatné moduly – ve vlastní aplikaci tak není možné využívat pouze konkrétní části. Systém sice umožňuje značnou konfiguraci a poskytuje mnoho různých nastavení, ale v případě potřeby změny nad rámec konfigurace je jediná možnost zkopírování (tvz. *fork*[15]) projektu a následné provedení vlastní změny.

Shopsys Framework dodržuje principy DDD[16] pro oddělení servisních tříd od datových modelů. Nicméně vlastní servisní třídy, které vykonávají aplikační logiku, nemají žádnou danou strukturu ani jasnou metodiku pro jejich vytváření a používání.

Aby nevzikal duplicitní kód jsou servisní třídy postupně refaktorovány na menší samostatné prvky tak, aby společnou logiku mohlo využívat více servisních tříd. Důsledkem je, že buď vzniká rozsáhlý strom závislostí jednotlivých

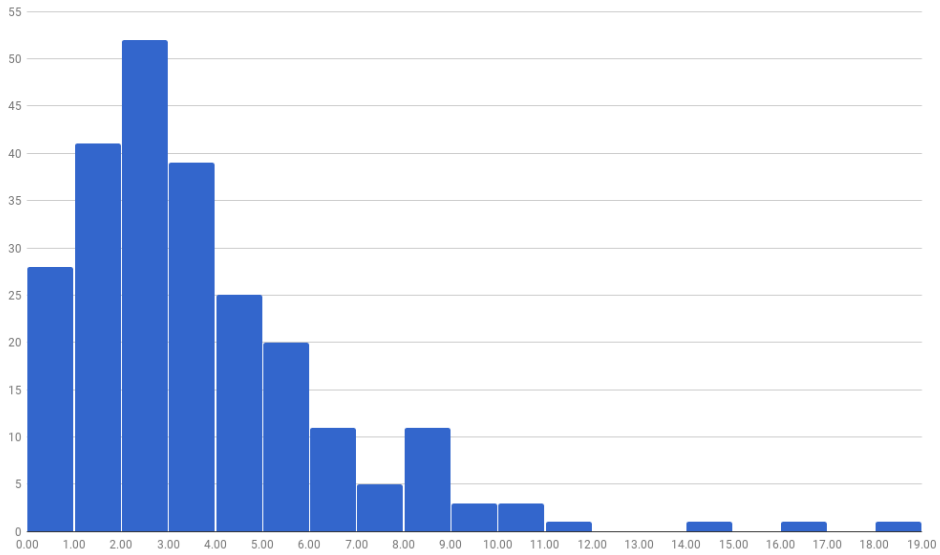
[13] bez nutnosti registrovat *ServiceProvider* pro každý modul zvlášť

[14] `https://github.com/shopsys/shopsys`

[15] `https://guides.github.com/activities/forking/`

[16] *Domain Driven Design* – Metoda modelování komplexního objektově orientovaného software

servisních tříd nebo mají hlavní servisní třídy velký počet přímých závislostí. Obrázek 2.1 zobrazuje, že ačkoliv většina servisních tříd má přiměřený počet závislostí (méně než čtyři), existují takové, které mají až 20 přímých závislostí (zejména se jedná o třídy implementující návrhový vzor *Facade*). Pro přehlednost aplikace by bylo vhodnější snížit počet přímých závislostí a vytvořit určitý řád, jak budou servisní třídy implementovány a jak budou vykonávány.



Obrázek 2.1: Histogram počtu přímých závislostí servisních tříd ve frameworku Shopsys [21]

2.2.3 Apiato

Škálovatelná webová aplikace využívající framework Laravel 5.6. Veškeré zdrojové kódy jsou veřejně dostupné na portálu GitHub[17]. Aplikace dodržuje samozvaný softwarový architektonický vzor Porto navržený tak, aby pomáhal vývojářům organizovat jejich kód udržitelným způsobem. Hlavním cílem tohoto návrhu je organizovat aplikační logiku, aby byla lehce škálovatelná a dlouhodobě udržitelná [22]. Jedná se o alternativu ke standardnímu vzoru MVC, zejména pro velké a dlouhodobé projekty.

Projekt je rozdělen na hlavní část (*Ship*), obsahující nezbytné části pro běh aplikace, a na jednotlivé moduly (*Containers*), které obsahují další rozšíření aplikace. Stále se však jedná o jediný repozitář obsahují veškerý kód. Není tedy možné využívat moduly jako samostatnou závislost ve vlastním projektu.

[17] `https://github.com/apiato/apiato`

Oproti frameworku Shopsys, popsanému v předcházející kapitole 2.2.2, je aplikační logika rozdělena do servisních tříd, které dodržují jednotnou strukturu (podle architektonického vzoru Porto). Ovšem samotná implementace aplikační logiky odporuje myšlence čistého a přehledného kódu. Jedná se hlavně o použití nepřehledné syntaxe (magické metody[18] či tzv. *Facades*), které zhoršují přehlednost kódu a možnost navigace v IDE[19]. Tyto praktiky jsou vyčítány i samotnému frameworku Laravel, který ovšem jejich použití nevyžaduje.

[18] `http://php.net/manual/en/language.oop5.magic.php`
[19] *Integrated Development Environment* – vývojové prostředí

Kapitola 3

Modulární architektura

V předcházející kapitole byly analyzovány existující projekty využívající modulární architekturu. Ani jedno řešení ovšem ve stávajícím stavu neumožňuje využití jednotlivých komponent samostatně. Od implementační části této práce je vyžadováno, aby byl kód strukturován modulárně a jednotlivé moduly byly samostatně dostupné. Cílem této kapitoly je analyzovat nejběžnější způsoby vývoje modulárních aplikací a jejich správu ve verzovacích repozitářích.

Modulární aplikace se skládají z více menších ucelených částí. Jednotlivé části jsou díky kratšímu kódu přehlednější a díky menší vzájemné provázanosti lépe spravovatelné. Existuje několik různých způsobů nastavení verzovacích repozitářů, které podporují modulární architekturu aplikace. Kapitola 3.2 popisuje možnost vytvoření vlastního repozitáře pro každý jednotlivý modul aplikace. Kapitola 3.3 popisuje možnost, jak zachovat výhody jediného repozitáře společně s modulární strukturou kódu.

3.1 Motivace pro modulární architekturu

Pokud již od začátku vývoje není projekt plánován jako skupina jednotlivých modulů, je vytvořen pouze jeden repozitář. Tento repozitář obsahuje veškeré zdrojové kódy, spravuje všechny závislosti a určuje verzi aplikace. Všichni vývojáři pracují pouze s jedním repozitářem. Závislosti v každé nově vydané verzi budou jednoznačné, nemůže vzniknout nekonzistence, kdy by jedna část aplikace vyžadovala jinou verzi stejné knihovny, než část jiná. Práce s takovýmto projektem je jasná a nenáročná na režii.

Jakmile začne projekt růst a začne přibývat více kódu, aplikace může začít být nepřehledná. Jednotlivé části mohou být nepředpokládaně závislé. Změna jedné části tak může vést k nutnosti úpravy části zcela jiné, což dává prostor k tvorbě chyb a vzniku duplicitního kódu. Navíc jsme nuceni vydávat novou verzi celé aplikace, ať se jedná o sebemenší změnu. Zprvu rychlý vývoj se tak bude nutně zpomalovat.

Prvním řešením je rozdělit kód na jednotlivé části do samostatných repozitářů, kde každý modul či projekt bude mít jasnou zodpovědnost, vlastní sadu závislostí a vlastní historii změn.

3.2 Více repozitářů

Jeden ze způsobů vývoje modulární aplikace je postupné oddělování ucelených prvků z hlavního projektu do samostatných modulů. Moduly je možné aktualizovat nezávisle na sobě, obsahují vlastní sadu závislostí a vlastní testy. Kód v repozitáři je odpovědný pouze za jednu část celé aplikace a je tak přehlednější pro vývojáře. Hlavní projekt bude záviset na konkrétních verzích těchto modulů.

Rozdělení funkcionality do více menších modulů přináší mnoho výhod. Ovšem s rostoucím počtem repozitářů, které je potřeba spravovat, vzniká větší počet nevýhod, které postupně mohou tyto výhody zastínit.

Výhody

- Lze řídit přístup do jednotlivých modulů zvlášť. Je tak možné aby vývojáři pracovali na jiných modulech nezávisle na sobě.
- Správa závislostí mezi moduly je jednodušší.
- Menší ucelenější zdrojový kód repozitáře ulehčuje údržbu a umožňuje rychleji pochopit kód novým vývojářem.
- Menší repozitáře napomáhají méně častému přepisování kódu při práci více vývojářů (tzv. *merge conflict*).
- Repozitáře umožňují samostatné verzování – vydavání nových verzí se tak týká pouze relevantních částí.
- Manipulace s menšími repozitáři je méně časově náročné, než u jednoho velkého repozitáře (např. když pomocné skripty často provádějí klonování repozitáře).

Nevýhody

- Pro větší počet (desítky až stovky) nezávislých repozitářů je velmi těžké udržet jednotnou strukturu a standardy pro kód a názvosloví.
- Změna napříč více moduly (např. razatní změna struktury, přechod na novou verzi PHP, apod.) je značně náročná na režii – je potřeba přidat změny do mnoha různých repozitářů, opravit závislosti a vydat nové verze a moduly vzájemně otestovat.

- Je možné se dostat do nekonzistentního stavu, kdy jeden modul vyžaduje změnu v závislém modulu, která ještě nebyla implementována.
- Obtížné integrační testovaní.
- Riziko, že nikdo nebude znát jak funguje systém jako celek. Budou existovat pouze skupiny vývojářů se znalostí části aplikace.

Proces vývoje

Vývoj funkcionality napříč více repozitáři není optimální co se týče procesu vývoje a testování. Běžný proces může vypadat následovně:

1. Přidání nové změny do modulu (či více modulů).
2. Nahrání změny na vzdálené uložiště (GitHub, GitLab, apod.) a vydání nové verze pro každý jednotlivý repozitář.
3. Aktualizování závislostí hlavního projektu a otestování výsledku.
4. Při zjištění chyby opakování celého procesu.

Aktualizace modulů

Pokud jsou moduly veřejné (*open source*) projekty, je možné využívat výhod, které poskytuje komunita vytvořená kolem projektu (viz kap. 1.1). Aktivní vývoj je v tomto případě ovšem obtížnější, protože modul je využíván i v jiných projektech a případné chyby se tak dotýkají více lidí.

Řešením problému obtížného procesu vývoje by bylo upravování všech (či většiny) modulů najednou v jednom vývojovém prostředí. Zároveň je vyžadováno zachovat výhodu jednotlivých modulů (tj. sadu závislostí, dokumentaci, historii změn, apod.). To vše umožňuje tzv. monolitický repozitář, který je popsaný v následující kapitole.

3.3 Monolitický repozitář

Dalším ze způsobů vývoje modulární aplikace je využití monolitického repozitáře. Monolitický repozitář (tzv. *monorepo*) je jediný repozitář, který může obsahovat více či méně nezávislých balíčků. To znamená, že dokonce i když je vyvíjena funkce překlenující více balíků, provedou se změny v jediném úložišti.

Vývoj v jednom repozitáři minimalizuje režii, která je nutná při vývoji a správě mnoha jednotlivých repozitářů. Jeden monolitický repozitář nutně neznamená jeden monolitický kód. Aplikace je tvořena z mnoha jednotlivých modulů, ale jejich údržba a vývoj probíhá v jednom okně jednoho vývojového prostředí. Pouze finální produkt je množství dílčích repozitářů.

Tento způsob využívá spousta velkých firem (např. Facebook [23] či Google [24]) k udržování svých jednotlivých projektů a modulů. *Monorepo* firmy Google obsahuje naprostou většinu jejich projektů a pracuje s ním 95 % vývojářů. Obsahuje přibližně dvě miliardy řádků kódu a má velikost 86 TB [24].

Monolitický repozitář je také využívám frameworky jako například Symfony, či Laravel. V repozitáři frameworku Laravel[20] je vidět, že všechny komponenty ve složce `src/Illuminate/` jsou samostatné moduly. Aby bylo možné používat pouze specifické části *monorepa* jsou komponenty dostupné samostatně ve vlastních repozitářích.

3.3.1 Lokální repozitáře

Lokální repozitář je způsob jak pomocí manažera závislostí Composer spravovat závislosti modulu v rámci monolitického repozitáře.

Aby mohl jeden projekt obsahovat více komponent, chovající se jako samostatné repozitáře, je potřeba nejdříve vytvořit potřebnou strukturu. Jedním ze způsobů jak dát kódu strukturu více repozitářu je rozdělit logiku do podsložek (např. `/src/Modules/Name`). Mohla by vzniknout struktura podobné té na obrázku 3.1.

```
framework
├── src
│   ├── App ..... kód aplikace
│   └── Modules ..... složka obsahující jednotlivé moduly
│       ├── Filesystem ..... modul pro práci se soubory
│       └── Support ..... modul s podpůrnými funkcemi
└── composer.json ..... nastavení závislostí projektu
```

Obrázek 3.1: Struktura modulů jako podsložky repozitáře

Na této struktuře jsou vidět ukázkové „moduly“ `Filesystem` a `Support`. Toto řešení je velmi jednoduché na implementaci, ale je zatíženo nedostatkem,

[20] `https://github.com/laravel/framework`

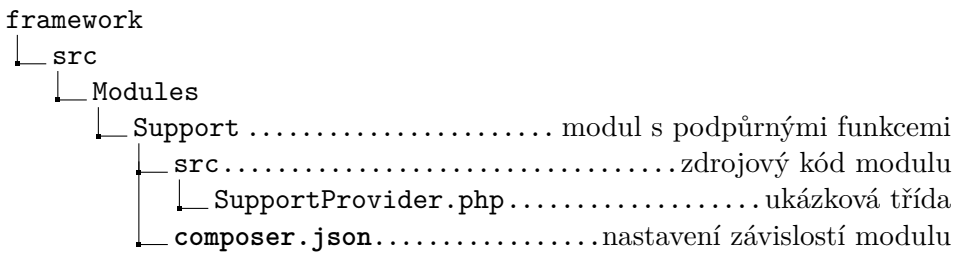
že jednotlivé moduly jsou stále plně vázané na konkrétní aplikaci. Není tak možné modul použít nezávisle na jiném projektu. Navíc může vznikat tzv. cyklická závislost, kdy aplikace závisí na obsahu modulu, který naopak závisí na samotné aplikaci.

Použití manažera zavislostí

Při tvorbě webové aplikace v jazyku PHP lze využít možnosti manažera závislostí Composer. Ten umožňuje zadat závislost na modulu, který je umístěn pouze lokálně v hlavním repozitáři a nemusí tak existovat žádny samostatný repozitář ve službě GitHub (či privátně na službě GitLab).

Rozšíření struktury modulů

Vytvoření takového lokálního repozitáře z již připravené struktury projektu je velmi jednoduché – rozšířením struktury modulu o vlastní konfigurační soubor `composer.json`, který spravuje závislosti modulu (viz obr. 3.2).



Obrázek 3.2: Rozšíření modulu o seznam závislostí

Seznam závislostí modulu

Soubor `src/Modules/Support/composer.json` bude mít následující obsah:

```
{
  "name": "modules/support",
  "require": {
    "php": "7.1.*"
  },
  "autoload": {
    "psr-4": {
      "Modules\\Support\\": "src"
    }
  }
}
```

Zaregistrování modulů

Nově vytvořený modul přidáme v hlavním projektu do seznamu závislostí úpravou souboru `src/composer.json` o následující kód:

```
{
  "name": "modules/framework",
  "require": {
    "modules/support": "@dev"
  },
  "repositories": [
    {
      "type": "path",
      "url": "src/Modules/Support"
    }
  ]
}
```

Aktualizace projektu

Nyní stačí klasický příkaz `$ composer update` pro aktualizaci závislosti ze všech zaregistrovaných modulů. Není potřeba předcházející *commit*[21] ani nové verze pro daný modul pro aktualizaci změn jako v případě procesu vývoje při práci s více repozitáři uvedeném v kapitole 3.2.

Nahrazování závislostí na jednotlivých modulech

Lokální repozitáře sice rozdělují kód do ucelených prvků a umožňují spravovat závislosti jednotlivých modulů, ale stále se jedná o nedílnou součást jednoho hlavního repozitáře. Pro vytvoření skutečně nezávislého modulu je nutné jít ještě o krok dále, vytvořením repozitáře (privátního či veřejného) pro každý modul.

Jakým způsobem je toho možné dosáhnout je popsáno v následujících kapitolách. Nyní předpokládejme, že tyto repozitáře již existují. Je nutné naznačit, že monolitický repozitář obsahuje části, které implementují stejný kód jako vlastní repozitáře modulů (tudíž je vlastně nahrazuje). Toho je docíleno rozšířením souboru `src/composer.json` o následující kód:

```
"replace": {
  "modules/filesystem": "self.version",
  "modules/support": "self.version"
}
```

Pokud je nyní vyžadována závislost `modules/framework` a zároveň modulu `modules/support` je nainstalován pouze projekt `modules/framework`.

[21] Zapsaní změn v souborech do vzdáleného uložiště.

Vytvoření repozitářů z podsložek

Existují dvě hlavní možnosti jak docílit vytvoření repozitářů pro jednotlivé moduly. První možnost je pomocí tzv. *submodulů* popsaných v kapitole 3.3.3. Druhá možnost je manipulace s Git historií hlavního repozitáře a její překopírování do mnoha dílčích repozitářů, jak je popsáno v kapitole 3.3.2.

3.3.2 Splitsh

První způsob rozdělení monolitického repozitáře do samostatných modulů je manipulace s Git historií.

Veškerý kód je v jednom repozitáři a je možné normálně pracovat s kódem jako by se jednalo o jednolitý projekt. Následně je spuštěn příkaz, který repozitář rozdělí pomocí Gitu na jednotlivé dílčí repozitáře. To lze udělat pomocí příkazu `$ git subtree`[22] (či `$ git subsplit`, což je jen obálka nad `subtree`) nebo pomocí programu Splitsh[23].

Program Splitsh je násobně rychlejší než `git subtree` (*cachuje* již jednou rozdělenou historii), ale neumí pracovat s historií, která byla pomocí `git subtree` přidána.

Verzování modulů

V Gitu nelze mít pro jeden projekt dvě stejné verze (dva stejně pojmenované tagy). To znamená, že buď budou mít všechny moduly stejnou verzi nebo budeme vydávat verze složitějším způsobem (např. pomocí vytvořeného skriptu, či manuálně pro každý dílčí repozitář).

Způsob, kdy mají všechny moduly shodnou verzi, používají například již zmíněné frameworky Symfony a Laravel. To ale znamená, že se vydávají verze modulů, ve kterých se vůbec nic nezměnilo. To odporuje systému sémantického verzování [25]. Pro určitý druh projektů tento systém vydávání nových verzí není až tak problematický, jelikož je přednostně využíván jako celek (např. frameworky), a tak jeho uživatel dává přednost jednotnému verzování s jistotou konzistentního stavu.

Výhody

- Mnohem snadnější správa závislostí v rámci modulů.
- Jednodušší vývoj a snadnější implementace funkcí, které pokrývají více než jeden modul.
- Snadné integrační testování všech modulů.

[22] `http://git-memo.readthedocs.io/en/latest/subtree.html`
[23] `https://github.com/splitsh/lite`

Nevýhody

- Všechny zahrnuté balíčky jsou zveřejňovány společně (tj. žádný dílčí modul monolitického repozitáře nemůže mít jinou verzi než ostatní).
- Neexistuje jednoduchý způsob, jak přímo upravovat dílčí repozitáře. Všechny změny musí být provedeny v hlavním projektu. Tudíž jsou repozitáře modulů pouze pro čtení.

Příklad použití

Je předpokládáno, že existuje struktura aplikace, používané jako *monorepo*, s jednotlivými komponentami ve složce `/src/Modules/` (jako na obrázku 3.2). Provedené změny nad celým repozitářem se odešlou na vzdálené uložistě (`git push`). Následně se provede rozdělení monolitického repozitáře do již předpřipravených dílčích repozitářů. Pro modul `Support` (a pro každý další dílčí repozitář) je potřeba provést následující příkazy:

```
$ git remote add support git@gitlab.com/support.git
$ SHA=‘/bin/splitsh-lite --prefix=src/Modules/Support/‘
$ git push support "$SHA:master"
```

Obsahem repozitáře `modules/support` pak budou pouze následující soubory (viz obr. 3.3) a historií změn tohoto repozitáře budou pouze změny, které se týkaly těchto souborů.

```
support
├── src
│   └── SupportProvider.php
└── composer.json
```

Obrázek 3.3: Struktura modulu `Support`

3.3.3 Submoduly

Druhý způsob rozdělení monolitického repozitáře do samostatných modulů je použití submodulů.

Submoduly jsou nativní technikou Gitu, která umožňuje udržovat repozitář Git jako podadresář jiného uložiště Git[24]. Skutečnost, že submoduly jsou stále samostatnými repozitáři, může být výhodou (umožňuje nezávislé verzování) nebo nevýhodou (řízení více repozitářů) v závislosti na přístupu, kterého se chce dosáhnout.

Git projekt, který virtuálně obsahuje další podprojekty (submoduly) se oficiálně nazývá *superprojekt*. Superprojekt umožňuje spravovat více repozitářů z jednoho vývojového prostředí jiným způsobem, než metodou Splitsh s lokálními repozitáři (viz kap. 3.3.2).

Submoduly nejsou totiž nic jiného, než ukazatele na jiný plnohodnotný Git repozitář. Při této konfiguraci se superprojekt chová jako kontejner na tyto dílčí repozitáře. Má uložené jak jejich lokální umístění, tak umístění na vzdáleném uložisti.

Vzhledem k tomu, že je v submodulu plnohodnotný Git repozitář, je možné pracovat s tagy verzí samostatně, a vydávat tak verze nezávisle na ostatních modulech.

Výhody

- Mnohem snadnější správa závislostí v rámci modulů.
- Všechny submoduly jsou samostatné repozitáře, tj. mohou být nezávisle verzovány a spravovány.

Nevýhody

- Je nutné spravovat moduly v několika repozitářích.
- Nutné zveřejňovat změny v jednotlivých repozitářích (moderní IDE umožňují jednotnou práci s mnoha repozitáři, není tak nutné neustále přepínat mezi repozitáři a složkami).

Příklad použití

Přidání nového dílčího repozitáře do složky `src/Modules/Support/` je provedeno příkazem:

```
$ git submodule add support git@gitlab.com/support.git
  src/Modules/Support/
```

Superprojekt si pak drží informaci o submodulech v souboru `.gitmodules`.

[24] `https://git-scm.com/docs/git-submodule`

3.4 Shrnutí

Použití více repozitářů umožňuje jejich větší kontrolu (např. nezávislé verzování, apod.), ale spravování většímu počtu repozitářů je velmi náročné na režii. Toto řešení je vhodné pro aplikace obsahující malé množství modulů. Není ovšem dobře škálovatelné, což není vhodné, jelikož nemusí být zřejmé kolik repozitářů bude nakonec spravováno.

Lepší škálovatelnost má použití monolitického repozitáře. Přidávání nových modulů nepřináší vyšší nároky na režii, jelikož je stále spravován pouze jeden repozitář. Moduly mohou být rozděleny do vlastních repozitářů. Tyto repozitáře ovšem mají vždy stejnou verzi a jsou dostupné pouze ke čtení – nejdou upravovat samostatně. Veškeré změny musí být provedeny v hlavním monolitickém repozitáři.

Použití submodulů je kombinací předchozích dvou způsobů. Kód je dostupný v jednom repozitáři – to umožňuje jednodušší vývoj více modulů najednou – ale jedná se o více samostatných repozitářů. Jednotlivé moduly tak lze nezávisle verzovat a upravovat. Ovšem při jejich větším počtu se zvyšují nároky na správu těchto repozitářů.

Kapitola 4

Návrh struktury

Cílem této kapitoly je navrhnout architekturu aplikace tak, aby umožňovala snadnou rozšiřitelnost a udržovatelnost. Toho je dosaženo oddělením jednotlivých vrstev aplikace a rozdělení kódu do komponent zodpovědných pouze za část prováděné logiky. Návrh komponent je bez závislosti na použitém programovacím jazyce či konkrétní implementaci. Návrh zodpovědnosti jednotlivých komponent vychází z architektonického vzoru Porto (viz kap. 2.2.3).

4.1 Struktura aplikace

Cílem práce je navrhnout způsob jak organizovat a psát kód tak, aby umožňoval snadnou udržitelnost pro dlouhodobé projekty. Struktura by měla oddělovat funkcionalitu do jednotlivých oddělených podsložek (modulů), které spolu budou komunikovat pouze přes sdílená rozhraní a budou tak splňovat principy čistého kódu (SOLID [2], GRASP [3], DRY, SoC, ...).

Požadavky

- Znovupoužitelnost aplikační logiky napříč více projekty.
- Jednoduchá správa a testovatelnost.
- Jasně definované názvosloví a struktura aplikace (zajišťuje přehlednost systému a snižuje potřebnou režii a komunikaci mezi vývojáři).
- Možnost napojení dalšího uživatelského rozhraní (API, CLI, apod.).
- Rozdělení zodpovědnosti, aby úprava jedné části neovlivňovala jinou.
- Rozdělení logiky do odpovědných částí (modulů).
- Vysoká škálovatelnost, pro snadné rozšiřování aplikace.

- Nepřímá závislost na použitém frameworku (jednoduchý přechod na nové verze).
- Malé přehledné třídy.
- Čistý a srozumitelný kód pro vývojáře (bez magických metod, apod.).

4.2 Komponenty

V této kapitole jsou popsány komponenty použité v návrhu architektury aplikace. Jedná se o popis zodpovědnosti jednotlivých komponent, bez závislosti na použitém programovacím jazyce či konkrétní implementaci. Tento návrh vychází z architektonického vzoru Porto (viz kap. 2.2.3). Návrh a implementace objektového návrhu je uvedena v kapitole 6.

Návrh vychází z vícevrstvé architektury MVC a z metody DDD. Návrh obsahuje komponenty, které používá většina PHP frameworků (např. routování, *requests*, *controllers*, *entities*, *views*, apod.), a vlastní třídy, ve kterých je prováděna aplikační logika. Aplikační (servisní) vrstva je rozdělena na hierarchicky uspořádáné třídy `Action` a `Task`. Tímto způsobem je podporováno vytváření malých přehledných tříd s menším počtem přímých závislostí než by tomu bylo u klasických služeb (*services*) definovaných DDD [26].

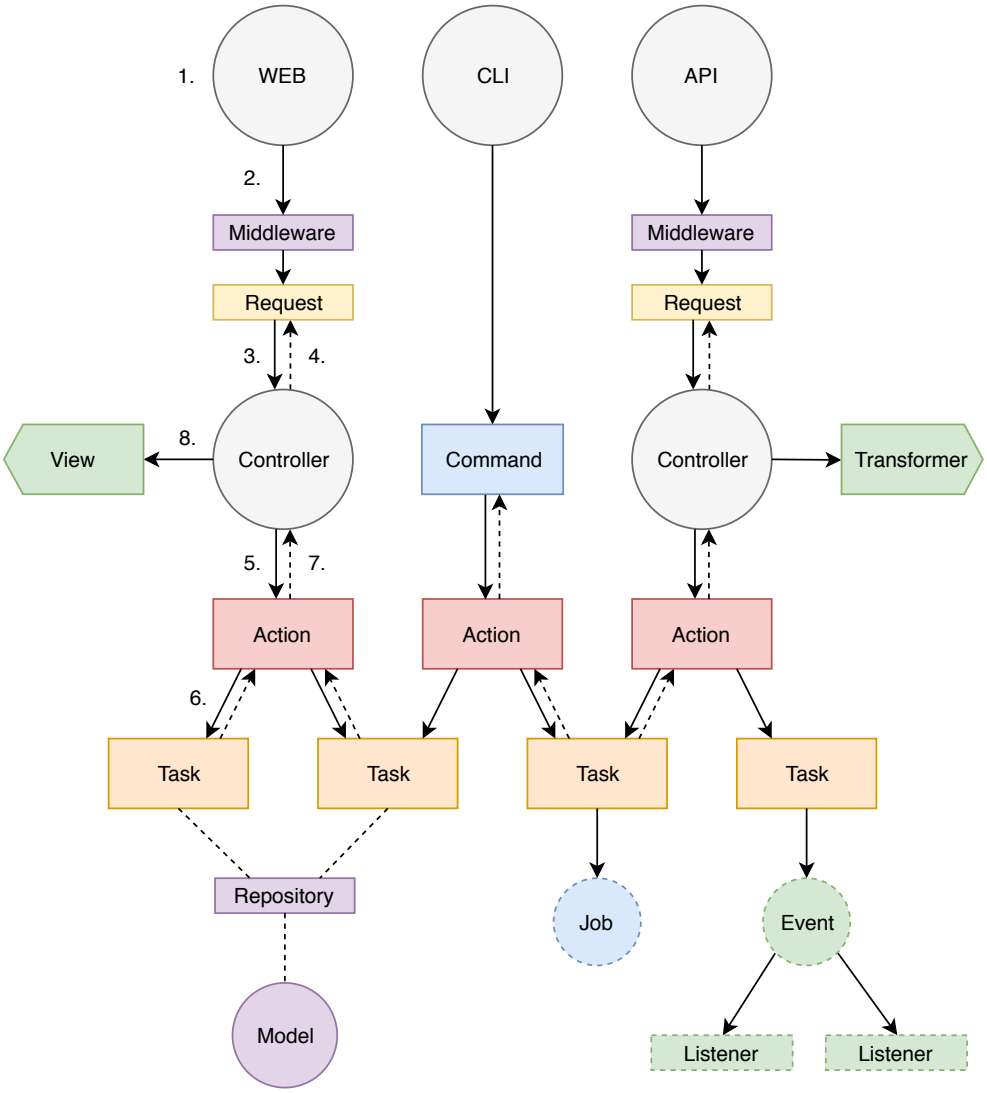
Aplikace je rozdělena do několika skupin komponent, přičemž každá skupina se stará o konkrétní část ze životního cyklu požadavku. Komponenty jsou implementovány jako samostatné třídy.

Životní cyklus požadavku

Životní cyklus požadavku je znázorněn na obrázku 4.1.

1. Uživatel odesílá HTTP požadavek na přístupové body (URL) definované v routování.
2. Aplikuje se množina tříd `Middleware`, které obstarávají potřebnou logiku pro každý požadavek (autentizace, zabezpečení, kontrola a nastavování hlaviček HTTP požadavku, apod.).
3. Je zavolána příslušná metoda v třídě `Controller` (definovaná v routování).
4. Třída `Request`, určená třídou `Controller`, provádí autorizaci požadavku a případnou validaci příchozích dat.
5. Třída `Controller` předává přijatá data třídám `Action`.
6. Třída `Action` provádí aplikační logiku (případně nechá třídy `Task` vykonat část logiky, která je společná pro více tříd `Action`).

7. Třída `Action` vrací zpracovaná data zpět do třídy `Controller`.
8. Třída `Controller` připraví formát odpovědi (v závislosti na rozhraní pomocí `View` či `Transformer`) a odesílá ji zpět uživateli.



Obrázek 4.1: Diagram interakcí hlavních součástí aplikace.

4.2.1 Hlavní komponenty

V této kapitole jsou popsány hlavní komponenty, které jsou nutné pro průchod aplikací (viz obr. 4.1).

4.2.1.1 Routes

Routování má na starost přijímání HTTP požadavků a jejich mapování na příslušné metody tříd `Controller`. Přístupové body (*endpoints*) definují URL, na které se mapují příchozí požadavky.

- Dostupné URL se definují pro každé UI zvlášť (tj. pro webové rozhraní, API[25], či CLI[26]).
- Routování má za úkol vybrat příslušný `Controller` a přeposlat na něj příchozí požadavek.

4.2.1.2 Controllers

`Controller` je zodpovědný za validaci příchozího požadavku, jeho obsloužení a vytvoření odpovědi v příslušném formátu. Validace probíhá v jiné třídě (`Request`), ale za její provedení je zodpovědný `Controller`.

- Měl by být inicializován pouze v routování.
- Každé UI (webové, API, CLI, ...) má vlastní sadu tříd `Controller`.
- Přijímá uživatelský vstup (data).
- Neměl by vědět nic o aplikační logice, ani ji přímo vykonávat.
- Přeposílá přijatá data do tříd `Action`.
- Typicky pracuje s jednou třídou `Action` (obsluhující daný uživatelský požadavek), ale může jich využívat více.
- Neměl by přímo interagovat s třídami `Tasks`, pouze s třídami `Action`.
- Vytváří formát odpovědi (obvykle pomocí dat vrácených ze třídy `Action`) a odesílá ji zpět uživateli.

[25] *Application Programming Interface*
[26] *Command Line Interface* – příkazový řádek

4.2.1.3 Requests

Třída `Request` se stará o příchozí data od uživatale (parametry v URL, či obsah formuláře). Je využívána ve třídách `Controller`, kde slouží k validaci a autorizaci dat.

- Třída `Request` může provádět validaci příchozích dat.
- Třída `Request` může provádět autorizaci požadavku.

4.2.1.4 Actions

Třída `Action` představuje konkrétní *Use Case*, tj. nějakou akci, kterou uživatel může v aplikaci vykonávat (např. registrace, přihlášení, přidání produktu do košíku, apod.).

Tato třída provádí aplikační logiku. Případně nechá část logiky, která je společná pro více tříd `Action`, vykonat třídu `Task`. Přijímá data, provede příslušnou logiku, a navrací výsledná data zpět. Nemělo by ji zajímat, kde se přijímaná data vzala, ani jak budou následně reprezentovány.

Podle názvů těchto tříd by mělo být jasné jaké možnosti (*Use Cases*) daná část aplikace (modul) umožňuje.

- Třída `Action` by měla přijímat data (např. ze třídy `Controller`).
- Jsou inicializovány hlavně ve třídách `Controller`, ale mohou být využity i v dalších třídách (`EventListener`, `Command`, apod.).
- Každá třída `Action` by měla být zodpovědná právě za jeden *Use Case* aplikace.
- Může využít třídu `Task` na provedení části aplikační logiky.
- Může využít více tříd `Task` (i z jiných modulů).
- Nesmí interagovat s jinou třídou `Action`. Pokud více tříd `Action` provádí obdobnou akci, je nutné její provedení přesunout do samostatné třídy `Task`.
- Může vracet data (v libovolném formátu).
- Nesmí vracet formátovanou odpověď uživateli – o transformaci dat odpovědi se stará třída `Controller`.
- Musí implementovat jednotnou strukturu (splňovat rozhraní).

4.2.1.5 Tasks

Zodpovídá za provedení logiky společné pro více tříd `Action`. Každá třída `Task` je tak odpovědná za malou část aplikační logiky.

Použití třídy `Task` není vyžadováno. Typicky je aplikační logika prováděna ve třídách `Action`, a až v případě potřeby (např. vytvořením další třídy `Action`) je refaktorizací vytvořena třída `Task`, která provádí společnou část aplikační logiky dvou a více tříd `Action`.

- Třída `Task` by měla být inicializována pouze ve třídě `Action`.
- Každá třída `Task` by měla mít pouze jednu zodpovědnost (tvz. *job*).
- Třída `Task` nesmí využívat další třídy `Task`.
- Třída `Task` nesmí využívat třídy `Action`.
- Musí implementovat jednotnou strukturu (splňovat rozhraní).

4.2.1.6 Entities

Entity poskytují abstrakci nad databázovou vrstvou. Mají za úkol pouze reprezentovat objekty uložené v databázi.

- Entity mohou mít vztahy s dalšími entitami.
- Entita nesmí obsahovat žádnou aplikační logiku, pouze reprezentaci svých dat a vztahů s ostatními entitami.
- Entita je odpovědná za validace svých dat.

4.2.1.7 Repositories

Třída `Repository` zajišťuje komunikaci aplikační vrstvy s datovou vrstvou. Umožňuje tak aplikační vrstvě přístup k datovému objektu, bez znalosti jakým způsobem k přístupu dat dochází.

- Zodpovídá za persistenci dat do databáze.
- Může obsahovat *cachování* entit (či jinou optimalizaci).
- Může být vázaná na konkrétní ORM[27] či typ databáze, ale musí vždy navracet stejný typ entity (splňující společné rozhraní).

[27] *Object-relational mapping*

4.2.1.8 Views

Zajištují vytvoření HTML[28] kódu, který je navrácen uživateli, pomocí dat přijatých od třídy `Controller`. Jejich cílem je oddělení prezenční vrstvy od aplikační vrstvy.

- Mohou být použity pouze ve třídách `Controller` ve webovém uživatelském rozhraní. (viz obr. 4.1)
- Měly by být implementovány jako oddělené soubory (HTML, soubory šablonovacího systému, apod.) pro různé odpovědi.

4.2.1.9 Transformers

Třídy zastupují obdobnou činnost jako `Views` pro webové uživatelské rozhraní, ale pro API rozhraní. Formátují příchozí data od třídy `Controller` do datového formátu JSON (případně XML).

- Všechny odpovědi API rozhraní by měli být serializovány pomocí třídy `Transformer`.
- Každá entita použitá v API rozhraní by měl mít svoji třídu `Transformer`.
- Struktura JSON odpovědi by měla být ve standartním formátu (např. `json:api`[29]).

4.2.2 Další komponenty

V této kapitole jsou popsány komponenty, které nejsou nezbytně nutné pro průchod aplikací. Mohou být vázané na použitý programovací jazyk či použitý framework.

4.2.2.1 Middlewares

Obstarávají potřebnou logiku pro každý HTTP požadavek – zabezpečení před útoky, nastavení *session*, autentizace, zabezpečení, kontrola a nastavování hlaviček HTTP požadavku, apod.

4.2.2.2 Service Providers

Mají za úkol inicializovat modul, tj. načtení konfigurace, zaregistrovat konkrétní třídy na rozhraní pro *dependency injection*, načtení `views`, apod.

[28] *HyperText Markup Language* – Značkovací jazyk používaný pro tvorbu webových stránek.

[29] `http://jsonapi.org/`

4.2.2.3 Event–Listeners

Třídy `Events` jsou jednoduchá DTO[30], která jsou vytvořena při provedení nějaké akce v aplikaci. Na tuto akci lze synchronně reagovat.

Třída **Listener** odchytává třídy `Events`, které jsou registrované v aplikaci. Následně provádí nějakou činnost (např. odeslání notifikace, logování, apod.). Může vykonávat aplikační logiku s využitím tříd `Action`.

4.2.2.4 Commands

Implementují obdobnou zodpovědnost jako třídy `Controller`, ale pro CLI. Mohou vykonávat aplikační logiku s využitím tříd `Action`. S jejich pomocí lze ovládat aplikaci.

4.2.2.5 Exceptions

Výjimky, které aplikace inicializuje při specifických chybách při provádění aplikační logiky.

4.2.2.6 Jobs

Třídy `Job` zastávají obdobnou činnost jako třídy `Task`. Aplikační logika třídy `Job` může být provedena asynchronně. Jsou tak vhodné na provádění časově náročných úkolů.

[30] *Data Transfer Object* – Objekt sloužící pouze k přepravě dat mezi vrstvami aplikace.

4.3 Datová vrstva

Požadavky navržených komponent entit (viz kap. 4.2.1.6) a repozitářů (viz kap. 6.6.5) určují, že entity nezodpovídají za persistenci svých dat do databáze, a že tuto zodpovědnost mají třídy repozitářů. Tyto požadavky musí být splněny pro všechny implementované způsoby persistence dat.

Aplikace by měla umět pracovat na různých databázových systémech. Závislost na jedné platformě neumožňuje jednoduchý přechod na jiný systém bez radikálního zásahu do kódu. Frameworky (včetně Laravelu) poskytují seznam podporovaných napojení, které lze snadno nastavit (Laravel 5.6 podporuje MySQL, PostgreSQL, SQLite a SQL Server). Tento stav je nutné zachovat i v implementační části této práce.

Laravel obsahuje vlastní ORM Eloquent (viz kap. 4.3.2), jehož použití je obvyklou součástí všech aplikací využívající tento framework. Eloquent je chválen za jednoduché a přehledné použití, a zároveň je mu vyčítáno použití mapování *Active Record* a nemožnosti optimalizace SQL dotazů. Proto je vhodné vytvoření takového systému, který by podporoval více ORM napojení, a umožňoval jednoduchou možnost jejich výměny. Pokud totiž bude postupováno podle běžných způsobů, výsledná aplikace bude na Eloquent plně závislá a nebude možné ho jednoduše odstranit. Toho je docíleno použitím návrhového vzoru repozitářů.

4.3.1 Objektově relační zobrazení

Objektově relační zobrazení neboli ORM je způsob jak mapovat řádek (resp. řádky) v databázových tabulkách relační databáze jako instanci třídy v objektově orientovaném jazyce. ORM zajišťuje automatickou konverzi mezi relační databází a objektově orientovaným programovacím jazykem. Díky tomu je vývojář do jisté míry odstíněn od nutnosti pracovat přímo s SQL dotazy konkrétní databáze. Použití ORM usnadňuje především CRUD[31] operace, tj. čtení, zápis, úpravu a mazání dat.

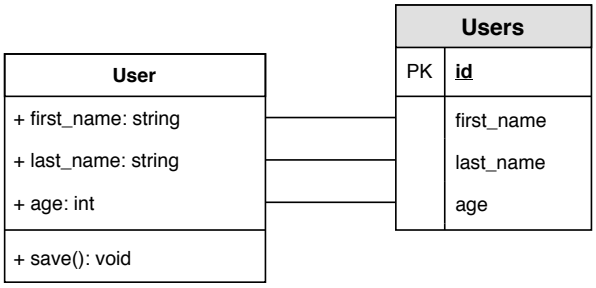
Hlavním cílem ORM je synchronizace mezi používanými objekty v aplikaci a jejich reprezentací v databázovém systému tak, aby byla zajištěna persistence dat.

Existuje několik návrhových vzorů pro ORM, nejznámnější jsou *Active Record* a *Data Mapper*.

[31] *Create*, *Read*, *Update*, *Delete*

4.3.1.1 Active Record

Active Record mapuje třídu přímo na tabulku databáze. Jedna instance třídy tak přesně odpovídá jednomu řádku příslušné tabulky – sloupec tabulky odpovídá stejně pojmenovanému veřejně dostupnému atributu třídy (jak je vidět na obrázku 4.2). Jelikož je entita úzce spjata s databázovou tabulkou, je tento způsob vhodný používat, když doménová logika není příliš složitá. Jakékoliv změny datového modelu způsobují nutnost úpravy celé aplikace.

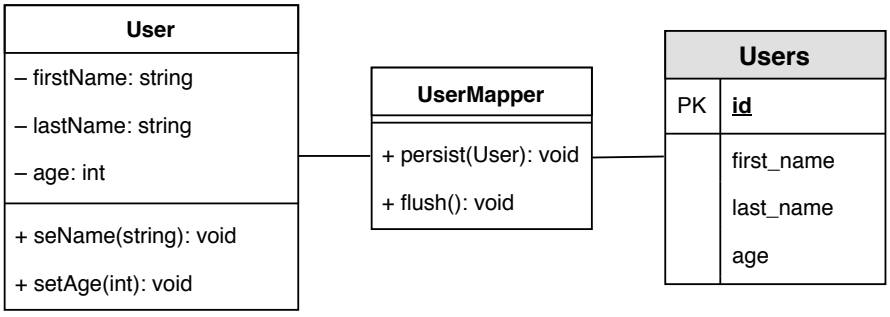


Obrázek 4.2: Třída `User` mapovaná pomocí *Active Record*.

4.3.1.2 DataMapper

Narozdíl od *Active Record*, který CRUD operace provádí velmi jednoduše, je v případě *Data Mapper* potřeba jednotlivé operace implementovat.

Jak je vidět na diagramu na obrázku 4.3, tento způsob přidává navíc třídu `UserMapper`. Ta přebírá zodpovědnost za mapování atributů na řádek příslušné databázové tabulky. Třída `User` tak nemá znalost jak se propisují její data do databáze.



Obrázek 4.3: Třída `User` mapovaná pomocí *Data Mapper*.

4.3.2 Eloquent

Velmi často se jako největší výhoda frameworku Laravel uvadí jeho jednoduchá a přehledná syntaxe, která umožňuje rychlý a snadný vývoj. K tomu přispívá i jeho ORM Eloquent využívající *Active Record*. To sice na jednu stranu umožňuje kratší a na první pohled uživatelsky čitelnější syntaxi, ovšem vytváří kód, který je velmi špatně optimalizovatelný a upravovatelný. Většina dostupných metod je dostupná pomocí statických či magických metod. U těch mají i nejmodernější IDE problém zjistit, kde všude v aplikaci k volání metody dochází, a těžko provádí refaktorizaci.

Třída entity rozšiřuje třídu `Eloquent`, která obsahuje logiku pro čtyři různé zodpovědnosti:

- *Data model* – Instance obsahuje pravidla a logiku pro správu stavu.
- *Row data gateway* – Instance může ukládat své data do databáze.
- *Table data gateway* – Umožňuje hledání, mazání i vytváření nových záznamů.
- *Factory* – Umožňuje vytváření nových instancí.

To samo o sobě nezpůsobuje problémy, ale umožňuje vývojáři použít některé funkcionality v části aplikace, kde by neměl. Výsledná aplikace nemusí být až tak velká, aby se začaly projevovat problémy s náročným načítáním většího počtu entit (včetně entit, které jsou s nimi ve vztahu, tzv. `N+1` problém). Tento problém pak nelze vyřešit optimalizací, jelikož neexistuje třída zodpovědná za komunikaci s databází.

To ovšem neznamená, že Eloquent (a obecně *Active Record*) nelze využít odpovědně. Je možné vytvoření jiné třídy zodpovědné za komunikaci s databází – např. pomocí návrhového vzoru repozitářů (viz kap. 6.6.5). Metody modelu Eloquent, které komunikují s databází, budou použity pouze v této třídě.

Příklad použití

```
$user = new User();
$user->first_name = 'Steve';
$user->last_name = 'Smith';
$user->age = 20;

$user->save();

$user = User::where('first_name', 'Steve')->find();
$user->delete();
```

Nevýhody

Jelikož každá entita rozšiřuje databázový model, může tak sama na sobě provádět příslušně operace pro ukládání dat. To umožňuje velmi rychlý vývoj, ale u velkých projektů může způsobovat problémy:

- Při změně databázového modelu (např. přejmenování sloupce) je potřeba provést změnu v celém kódu.
- Přímé mapování třídy na řádek nemusí být bezpečné. Je možné zadat nesprávně hodnoty (v modelu není provedena validace).
- Optimalizace (zrychlení) dotazů nad modelem není možná.
- Neobsahuje *Identity Map* – systém jak zamezit opakovanému propisování stejné nezměněné entity do databáze.

4.3.3 Doctrine

Nejznámější implementace *Data Mapper* ORM je Doctrine2[32]. Tento systém využívá například framework Symfony.

Příklad použití

Předpokládejme, že existuje stejná tabulka `users`, jako v přechozím případě, se sloupci `first_name`, `last_name` a `age` (viz obr. 4.3).

Samotná manipulace s atributy entity je prováděna pomocí veřejných *getter* a *setter* metod. Třída `EntityManager`, zodpovědná za persistenci dat, obsahuje metodu `persist()`, která vytvoří příslušný SQL příkaz. Metoda `flush()` provede příkazy v databázi jako jednu transakci.

```
$user = new User();
$user->setFirstName('Steve');
$user->setLastName('Smith');

$entityManager->persist($user);
$entityManager->flush();
```

Doctrine2 má proti ORM Eloquent oddělenou zodpovědnost persistence dat do samostatné třídy. Tento systém tak více odpovídá požadavkům komponent uvedených v kapitole 4.2.1.6.

[32] `https://github.com/doctrine/doctrine2`

Kapitola 5

Implementace servisní vrstvy

Při návrhu webových aplikací se běžně používá klasická třívrstvá MVC[33] architektura. Návrh uvedený v kapitole 4.2 se oproti této architektuře liší hlavně přesnou specifikací, jak má být implementována aplikační (servisní) vrstva.

Dle návrhu je aplikační logika prováděna především ve specializovaných servisních třídách `Action` a `Task`. Cílem této kapitoly je popsat, jakým způsobem je možné tuto část návrhu aplikace implementovat tak, aby splňovala všechny zadané požadavky (viz kap. 4.2.1.4 a 4.2.1.5).

Existuje několik možných způsobů implementace. Jedna z možností je nechat servisní třídy, aby se inicializovaly a prováděly samostatně (viz kap. 5.1). Další možností je využít jediné specializované třídy, která bude zodpovídat za provádění všech servisích tříd (viz kap. 5.2).

5.1 Nezávislé servisní třídy

Jedna z možností implementace aplikační vrstvy je nechat servisní třídy, aby se inicializovaly a prováděly samostatně a tak zcela nezávisle na ostatních třídách.

Tento způsob je velmi jednoduchý a přehledný, ale nepřináší nic, co by napomáhalo jednoduché rozšiřitelnosti a dlouhodobé udržitelnosti aplikace. Nikde není vyžadováno, aby tyto třídy implementovaly společné rozhraní (*interface*). Dodržení jednotné struktury servisních tříd je tak zcela na vývojáři.

```
public function addItem()
{
  // ... set needed variables
  $action = new AddItemToCartAction();
  $action->handle($cart, $product, $amount);
}
```

[33] *Model–view–controller*

Servisní třídy je možné vkládat do aplikace pomocí *dependency injection*. Tento způsob umožňuje opatřit každou servisní třídu vlastním rozhraním. Při nevyhovující implementaci servisní třídy v modulu třetí strany je možné vytvoření vlastní implementace, která splňuje dané rozhraní, a která bude využita místo původní konktrétní třídy[34]. Vytváření rozhraní pro každou servisní třídu ovšem může být velmi náročné na implementaci.

```
public function addItem(AddItemToCartActionInterface $action)
{
  // ... set needed variables
  $action->handle($cart, $product, $amount);
}
```

Velkou nevýhodou nezávislých servisních tříd je absence jediné třídy zodpovědné za jejich provádění. Není tak možné implementovat logiku, která by se aplikovala pro každou servisní třídu (např. logování, automaticky provedený *rollback* selhané akce, apod.).

Výhody

- Možnost využití rozhraní (*interface*) v závislostech namísto konkrétních tříd.
- Dostupná nápověda datových typů parametrů (*type hinting*) v IDE.
- Dostupná nápověda návratových hodnot (*return type hinting*) v IDE.

Nevýhody

- Není vyžadována jednotná struktura – je možné využít všech veřejných metod třídy.
- Každá třída pracuje zcela nezávisle.

[34] Ve frameworku Laravel je možné nastavit *dependency injection* aby změnil závislosti pro jednotlivé třídy způsobem: `$app->when($container)->needs($abstract)->bind($concrete)`.

5.2 Dispatcher

Třída `Dispatcher` je zodpovědná za provádění servisních tříd. Použitím této třídy je odstraněna hlavní nevýhoda předchozího řešení nezávislých servisních tříd, kdy se každá servisní třída prováděla samostatně. Jednotný přístup ke každé prováděné servisní třídě umožňuje aplikovat společnou logiku. Může se jednat například o obalení blokem `try/catch` pro odchytávání chyb, jejich logování, a následné provedení metody `rollback()` v prováděné třídě.

Kdyby neexistovala třída zodpovědná za provádění servisních tříd, vznikl by podobný vzor jako je například ORM Eloquent využívající mapování *Active Record*, kdy entita zastupuje několik různých zodpovědností. Toho se tato implementace snaží vyvarovat.

5.2.1 Inicializace servisní třídy

Existuje několik možností jakým `Dispatcher` může přijímat informaci o tom, jaká servisní akce se má provádět a jaké používá parametry.

5.2.1.1 Formátovaný řetězec

Servisní třída je specifikována řetezcem v předem domluveném formátu (např. název modulu a název třídy). Parametry jsou zaslany zvlášť jako jednotlivé položky pole.

```
$dispatcher->call(
  'Cart@AddItemToCartAction', [$cart, $product, $amount]
);
```

Tento způsob je implementován v ukázkové aplikaci architektonického návrhu Porto (viz kap. 2.2.3). Po testovací implementaci bylo zjištěno, že tento způsob je velmi nepřehledný a nevhodný pro dlouhodobou udržitelnost, jelikož nikde není uvedená celá cesta k dané třídě, a je tak prakticky nemožné provádět jakoukoliv refaktorizaci.

Výhody

- Inicializace servisní třídy až ve třídě `Dispatcher`.

Nevýhody

- Nemožnost jakékoliv refaktorizace.
- Moduly obsahující větší množství servisních tříd se stávají nepřehledné.
- Není dostupná nápověda datových typů parametrů (*type hinting*) v IDE.
- Není dostupná nápověda návratové hodnoty (*return type hinting*) v IDE.

5.2.1.2 Celý název třídy

Jedná se o jednoduché vylepšení předchozího řešení, které využívá formátovaného textu pro identifikaci servisní třídy. Namísto řetězce s formátovaným identifikátorem třídy je použito úplného názvu třídy (*namespace*).

```
use Modules\Cart\Actions\AddItemToCartAction;

$dispatcher->call(
  AddItemToCartAction::class, [$cart, $product, $amount]
);
```

Název třídy je tak refaktorovatelný, a moderní IDE umožňují rychlou navigaci na implementaci třídy. To napomáhá přehlednosti aplikace při vývoji. Zbytek nevýhod zůstává stejných jako v předchozím případě.

Výhody

- Možnost omezené refaktorace (např. názvu a umístění třídy).
- Inicializace servisní třídy až ve třídě `Dispatcher`.
- Větší přehlednost aplikace.

Nevýhody

- Není dostupná nápověda datových typů parametrů (*type hinting*) v IDE.
- Není dostupná nápověda návratové hodnoty (*return type hinting*) v IDE.

5.2.1.3 Inicializová třída

Třída `Dispatcher` přijímá inicializovanou instanci servisní třídy. Ta má již vložené požadované parametry a není potřeba je předávat třídě `Dispatcher`.

```
use Modules\Cart\Actions\AddItemToCartAction;

$dispatcher->call(
  new AddItemToCartAction($cart, $product, $amount)
);
```

Tento způsob umožňuje vynucení použití jednotného rozhraní (*interface*) implementující metodu `handle()`. To u přechozích způsobů (viz kap. 5.2.1.1 a 5.2.1.2) nebylo možné, kvůli různým parametrům této metody. Také umožňuje nápovědu datových typů parametrů viditelnou již při vývoji, nikoliv až za běhu aplikace (*runtime*).

Výhody

- Možnost plné refaktorace.
- Vynucuje použití jednotného rozhraní (*interface*).
- Velká přehlednost aplikace.
- Dostupná nápověda datových typů parametrů (*type hinting*) v IDE.

Nevýhody

- Obtížnější testování (kvůli inicializovaným třídám se obtížně nastavuje tzv. *mockování*).
- Není dostupná nápověda návratové hodnoty (*return type hinting*) v IDE.

5.2.2 Přístup ke třídě v aplikaci

Existuje několik způsobů jakým je možné přistupovat k třídě `Dispatcher` a využívat ji.

5.2.2.1 Statický přístup (*Facade*)

Použití návrhového vzoru *Facade* (či statické metody) je velmi vhodné pro pohodlné použití kdekoliv v aplikaci.

```
\Dispatcher::call(
  new AddItemToCartAction($cart, $product, $amount)
);
```

5.2.2.2 Kompozice (*Composition*)

Způsob, splňující principy SOLID, je vkládání třídy `Dispatcher` definové pomocí rozhraním využitím DI[35]. Je možné vkládat třídu v každé metodě, kde je využívána.

```
public function addItem(DispatcherInterface dispatcher)
{
  // ... set needed variables

  $dispatcher->call(
    new AddItemToCartAction($cart, $product, $amount)
  );
}
```

[35] *Dependency Injection*

Další možností je vytvoření privátní proměnné, ke které lze následně přistupovat ve všech metodách třídy.

```
private $dispatcher;

public function __construct(DispatcherInterface dispatcher)
{
  $this->dispatcher = $dispatcher;
}

public function addItem()
{
  // ... set needed variables

  $this->dispatcher->call(
    new AddItemToCartAction($cart, $product, $amount)
  );
}
```

Tento způsob ovšem umožní přístup ke všem metodám dané třídy, což nemusí být kvůli požadavkům návrhu vhodné (např. třídy `Action` by neměly mít možnost využívat ostatní třídy `Action`, ale tuto možnost by v tomto případě měly).

5.2.2.3 Dědičnost (*Inheritance*)

Další možností je vytvoření vlastní metody (např. v nadřazené abstraktní třídě) tak, aby přijímala pouze rozhraní daného typu servisní třídy. Implementace je možná např. pomocí tzv. *traits* pro jednoduchou znovupoužitelnost ve více třídách.

```
public function addItem()
{
  // ... set needed variables

  $this->dispatchAction(
    new AddItemToCartAction($cart, $product, $amount)
  );
}

public function reduceStock()
{
  // ... set needed variables

  $this->dispatchTask(
    new ReduceStockForItemTask($product, $amount)
  );
}
```

5.2.3 Implementace

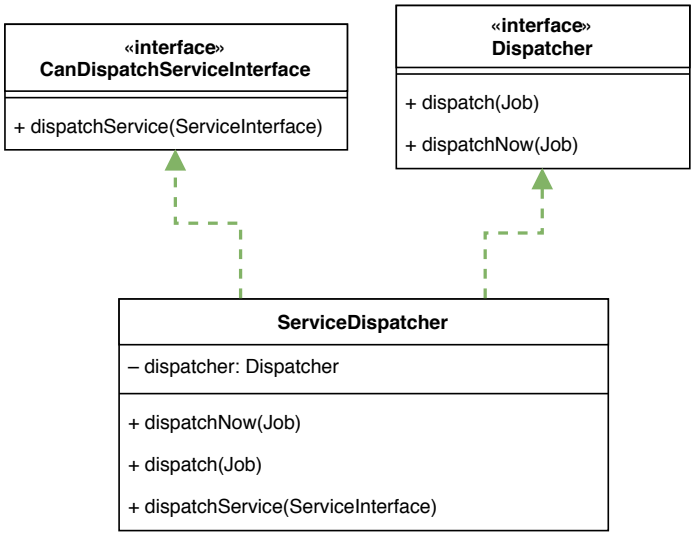
Finální implementace bude přijímat servisní třídy jako inicializované instance (viz kap. 5.2.1.3) a bude využívat dědičnost (*inheritance*) pro přístup ke specifickým metodám třídy `Dispatcher` (viz kap. 5.2.2.3).

Toto řešení přináší více režie při ukládání parametrů v servisních třídách, kvůli nutnosti použití privátních proměnných při inicializaci. Také zde není možné využít DI pro vkládání závislostí, jako v ostatních případech inicializace servisních tříd. Tyto nevýhody ovšem vyvažuje možnost plné refaktorace a dostupná nápověda datových typů parametrů.

Tyto třídy jsou odbavovány obdobným způsobem jako třídy *Job*[36] využívané ve frameworku Laravel.

5.2.3.1 Objektový návrh

Vlastní třída `ServiceDispatcher` pro odbavování servisních tříd je implementována jako dekorátor (viz kap. 5.2.3.2) existující třídy splňující rozhraní `Dispatcher`[37], použité ve frameworku Laravel pro provádění asynchronních úkolů (*Jobs*). Zároveň splňuje rozhraní `CanDispatchServiceInterface`, které implementuje metodu pro odbavení servisních tříd (viz obr. 5.1). Tímto způsobem je využita funkcionalita, která je již frameworkem využívána a je tak známá pro vývojáře, a je pouze rozšířena o nově požadovanou funkcionalitu



Obrázek 5.1: Objektový návrh třídy `ServiceDispatcher`

[36] `https://laravel.com/docs/master/queues`
[37] `\Illuminate\Contracts\Bus\Dispatcher`

5.2.3.2 Dekorátor

Dekorátor se vytváří za účelem změny instancí tříd bez nutnosti vytvoření nových odvozených tříd, jelikož pouze dynamicky připojuje další funkčnosti k objektu. V případě implementace třídy `ServiceDispatcher` se jedná o metody, vyžadované původním rozhraním, `dispatch()` a `dispatchNow()`. Ty kontrolují, zda přijatý objekt je servisní třída, kterou poté odbaví specializovanou metodou (viz obr. 5.2). Pro zbytek metod vyžadovaných původním rozhraním řešení neimplementuje žádnou vlastní funkcionalitu a pouze odkazuje na původní třídu `Dispatcher`.

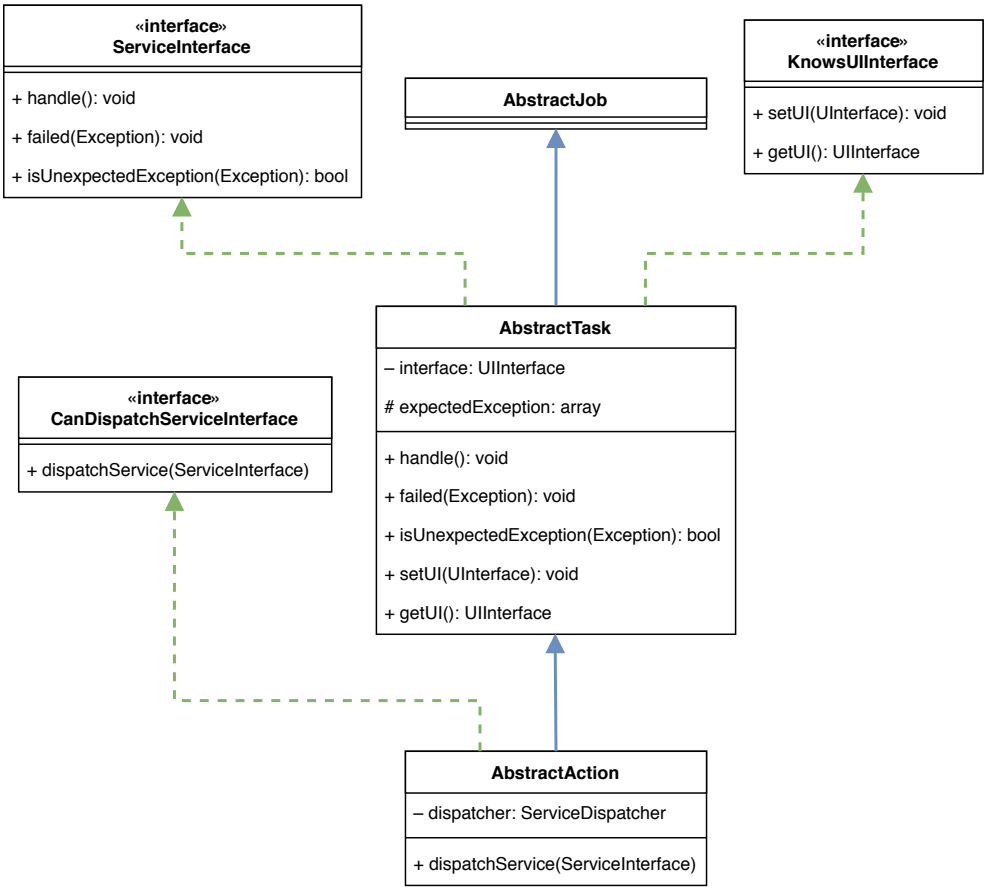
```
namespace Modules\Bus;

use Exception;
use Illuminate\Bus\Dispatcher;
use Illuminate\Contracts\Bus\Dispatcher as DispatcherInterface;
use Modules\Bus\Contracts\CanDispatchServiceInterface;
use Modules\Bus\Contracts\ServiceInterface;

class ServiceDispatcher implements
  DispatcherInterface, CanDispatchServiceInterface
{
  private $dispatcher;

  public function __construct()
  {
    $this->dispatcher = new Dispatcher();
  }

  public function dispatch($job)
  {
    if ($job instanceof ServiceInterface) {
      return $this->dispatchService($job);
    }

    return $this->dispatcher->dispatch($job);
  }

  public function dispatchService(ServiceInterface $service)
  {
    try {
      // service is handled like clasic Job instance
      return $this->dispatcher->dispatch($service);
    }
    catch (Exception $exception) {
      // service fails with unexpected exception, run rollback
      if ($service->isUnexpectedException($exception)) {
        $service->failed($exception);
      }
      // pass exception back to the application
      throw $exception;
    }
  }
```

Obrázek 5.2: Ukázková implementace třídy `ServiceDispatcher`

5.3 Servisní třídy

Třída `ServiceDispatcher` je navržena a implementována tak, že servisní třídy `Action` a `Task` jsou jen specializované třídy *Job* (viz obr. 5.1) a bylo by možné je odbavovat i původní implementací třídy `Dispatcher`.

Dle požadavků v návrhu (viz kap. 4.2.1.4) se třída `Action` oproti třídě `Task` liší pouze tím, že může sama využívat další třídy `Task` pro provádění částí logiky. To znamená, že by mohla být implementována jako její podtřída, která navíc implementuje požadované rozšíření – možnost využití tříd `Task`. Tento objektový návrh je zobrazen na obrázku 5.3.

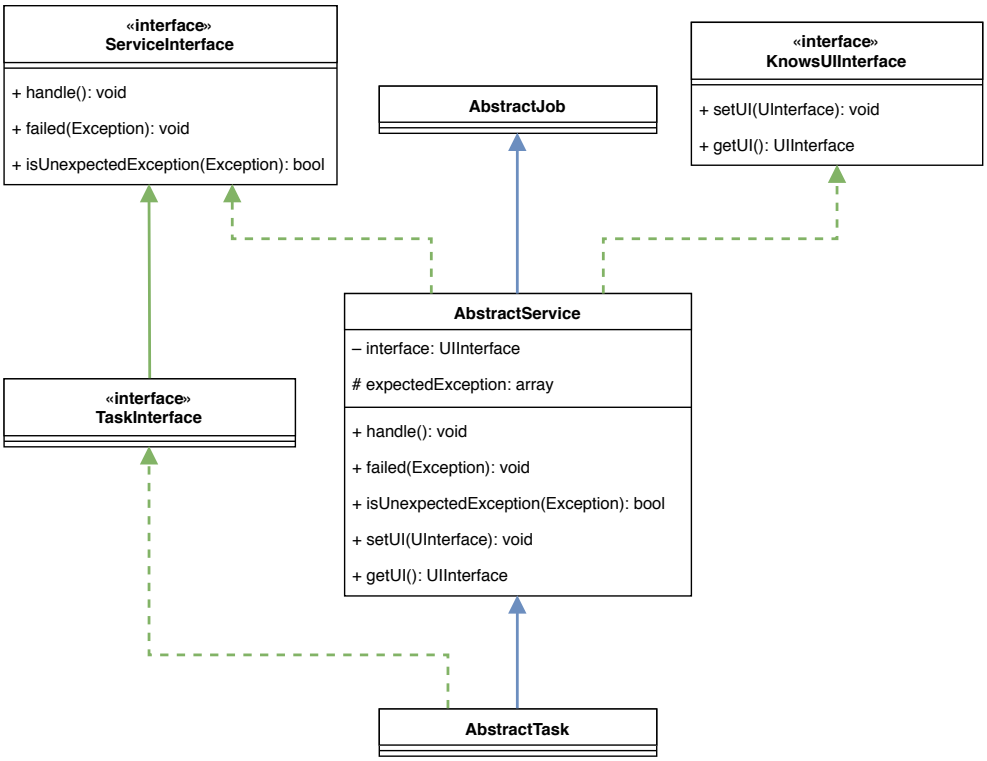


Obrázek 5.3: Objektový návrh abstraktní třídy `Action` jako podtřídy třídy `Task`

Tento návrh ovšem odporuje Liskovově principu zaměnitelnosti [2]. Ten říká, že pokud nějaký kód používá bázovou třídu, musí fungovat i v případě, kdy místo bázové třídy použijeme jakoukoliv z jejích podtříd. Kódu používajícímu bázovou třídu tedy musí být jedno, zda dostane bázovou třídu nebo její podtřídu. Všechny podtřídy musí být z pohledu uživatele bázové třídy vzá-

jemně zaměnitelné. Uživatel bázové třídy také nesmí být nucen měnit svoje chování podle typu obdržené podtřídy.

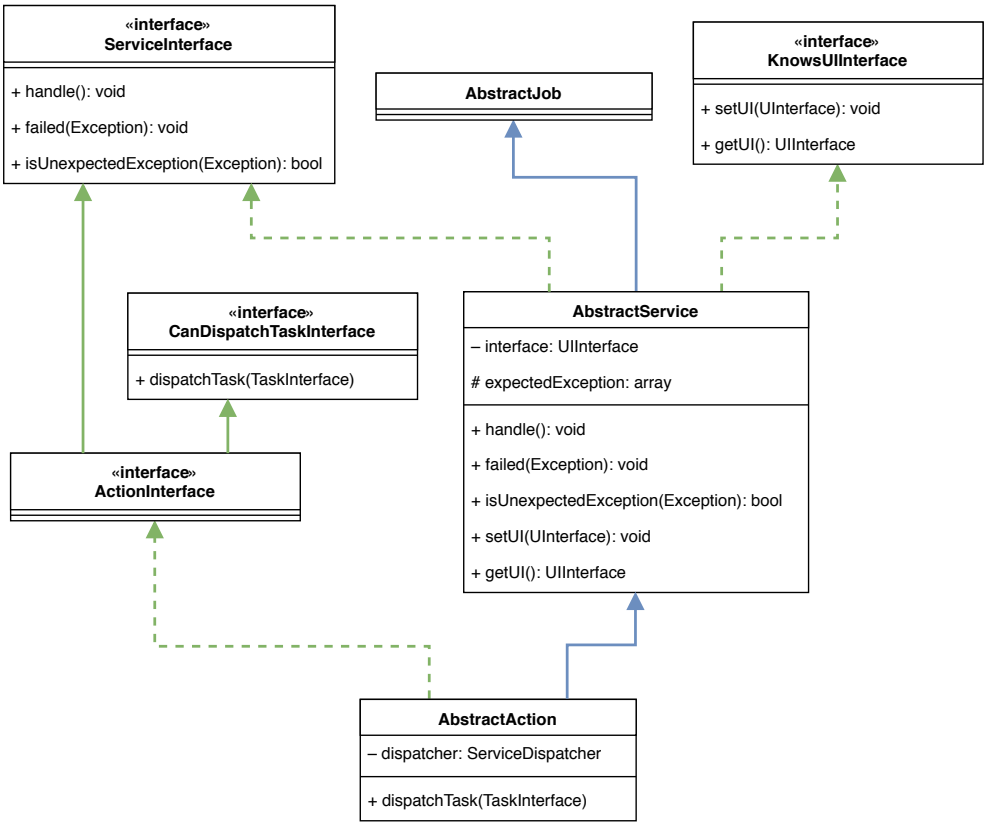
To znamená, že kdekoliv kde je možné použít třídu `Task`, by mělo být možné použít její podtřídu `Action`. Tedy možnost využití tříd `Action` v jiných instancích třídy `Action`, což je ale dle zadaných požadavků znemožněno (viz kap. 4.2.1.4).



Obrázek 5.4: Objektový návrh abstraktní třídy `Task`

Je tedy nutné vytvořit rozhraní `ServiceInterface` společné pro všechny servisní třídy a specializované rozhraní `TaskInterface` (viz obr. 5.4) a další rozhraní `ActionInterface` (viz obr. 5.5) pro různé typy servisních tříd. Třída `ServiceDispatcher` bude moct zpracovávat všechny servisní třídy se společným rozhraním `ServiceInterface` a třídy `Action` pouze třídy splňující rozhraní `TaskInterface` (např. pomocí metody v abstraktní třídě, jak bylo uvedeno v kapitole 5.2.2.3).

Jak je uvedeno v návrhu architektury, třídy `Controller` (a další třídy) by měly používat pouze třídy `Action` na provádění svých *Use Case* a přímé použití tříd `Task` je nedoporučováno, nicméně je možné. Tento návrh zaručuje, že nevznikají zbytečné třídy, které by jenom přeposílaly přijaté parametry do třídy `Task` implementující veškerou potřebnou logiku (např. CRUD operace).



Obrázek 5.5: Objektový návrh abstraktní třídy `Action`

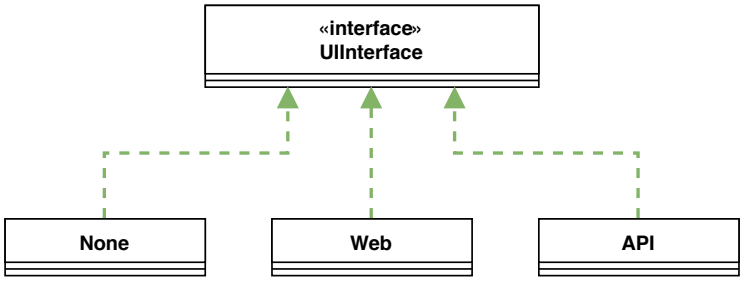
5.4 Uživatelské rozhraní

Webová aplikace může zobrazovat data v různých formátech, dle toho, co je od ní požadováno. Nejběžnější způsob přístupu k webové aplikaci je přes webový prohlížeč, kdy jsou data odpovědi ve formátu HTML. Někdy může webová služba sloužit jako API, poskytující práci s daty ve formátu JSON, XML, či jiném. Také může poskytovat více možností zároveň – webové rozhraní pro běžný přístup, veřejné API pro napojení externích aplikací, či interní API pro administrativní účely.

Tyto různé rozhraní provádějí obdobnou aplikační logiku, proto je vhodné oddělení aplikační (servisní) části aplikace tak, aby byla nezávislá na použitém uživatelském rozhraní, a výsledná data pouze transformovat do požadovaného formátu.

Nicméně, někdy může být vhodné, aby servisní třídy měly přehled, ve kterém rozhraní jsou prováděny. Tato znalost může vést k minimální změně prováděné akce (neměla by mít vliv na výsledek), např. detailnější logování, optimalizace SQL operací, apod.

Stav UI je implementován pomocí návrhového vzoru *State*. Aplikace tedy není omezena jen na několik předdefinovaných rozhraní, ale umožňuje vytvoření libovolného množství nových uživatelských rozhraní. Servisní třídy nutně nevyžadují specifikované UI, tudíž existuje i prázdný stav uživatelského rozhraní `None`.



Obrázek 5.6: Objektový návrh tříd `UI`

Použití různých UI je další důvod proč je využito dědičnosti (*inheritance*) pro přístup ke třídě `ServiceDispatcher`. Tímto způsobem je možné v metodě abstraktní třídy nastavit servisní třídě aktuální UI.

```
abstract public function getUI(): UIInterface;

// ... AbstractController method
public function dispatchService(ServiceInterface $service)
{
  // set UI to service instance
  if ($service instanceof KnowsUIInterface) {
    $service->setUI($this->getUI());
  }

  // pass service instance to dispatcher
  return $this->dispatcher->dispatch($service, 'handle');
}


// ... AbstractAction method
public function dispatchTask(TaskInterface $task)
{
  // set UI to task instance
  if ($task instanceof KnowsUIInterface) {
    $task->setUI($this->getUI());
  }

  // pass task instance to dispatcher
  return $this->dispatcher->dispatch($task, 'handle');
}
```

Obrázek 5.7: Způsob vkládání aktuálního UI do servisních tříd

Kapitola 6

Implementace modulů

Tato kapitola popisuje implementaci sady modulů pro podporu e-commerce aplikace ve frameworku Laravel podle navržené metodiky. Aplikační logika je implementována pomocí servisních tříd (popsány v předcházející kapitole) a dodržuje rozdělení zodpovědností jednotlivých komponent podle návrhu uvedeného v kapitole 4. Řešení je v následující kapitole otestováno vzhledem k rozšiřitelnosti a přizpůsobitelnosti. Všechny zdrojové kódy jsou přiloženy v příloze této práce či veřejně dostupné v hlavním monolitickém repozitáři na portálu GitHub na `https://github.com/novott20/MI-DIP_framework`.

6.1 Použité technologie

Aplikace vyžaduje PHP verze 7.1 (s plnou optimalizací pro nejnovější verzi PHP 7.2). Všechny závislosti jsou uvedeny v souboru `composer.json`, který je součástí jak hlavního monolitického repozitáře, tak jednotlivých modulů.

Framework Laravel

Moduly jsou vytvořeny speciálně pro použití s frameworkem Laravel 5.6, který je popsaný v kapitole 1.2. Implementace přímo využívá funkce tohoto frameworku a použití s jiným frameworkem nikdy nebude optimální.

Twitter Bootstrap

Jelikož vytváření propracovaných *frontend* šablon není cílem práce, použité šablony slouží pouze k demonstraci vypsaných dat. Pro stylování webového rozhraní je využito CSS frameworku Twitter Bootstrap 4.

Laravel Doctrine

Jak bylo uvedeno v kapitole o datové vrstvě 4.3, použití ORM Eloquent (nativní systém pro framework Laravel) není optimální pro správné rozdělení

zodpovědností pro persistenci dat. Je tak využito modulu Laravel Doctrine, který implementuje ORM Doctrine2 pro použití s frameworkem Laravel.

Laravel Passport

Oficiální modul pro framework Laravel, který obstarává autentizaci a autorizaci uživatelů v API rozhraní pomocí OAuth2 či JSON Web Token.

Spatie Fractal

Modul, který obstarává implementaci tříd `Transformer` (viz kap. 4.2.1.9), tj. transformaci entit, či jiných dat, do JSON formátu strukturovaného podle doporučovaných standardů. Díky tomu je výsledné API velmi přehledné a jednoduše použitelné.

Form Builder

Pro ulehčení práce při tvorbě formulářů je využito modulu Form Builder. Ten je inspirován obdobnou komponentou z frameworku Symfony. Tento modul generuje samotný HTML kód formuláře namísto vývojáře.

PHPUnit

Knihovna PHPUnit je standard pro testování PHP aplikací. Byl vytvořen pro tvorbu jednotkových testů, ale lze jej použít i pro tvorbu integračních či funkčních testů [27]. Integrační testy, oproti jednotkovým, mohou testovat více částí (tříd) aplikace současně [28]. Integrační testování je použito pro testování servisních tříd.

Mockery

Služba umožňující snadnější nastavení tzv. *mockování* tříd při automatickém testování pomocí služby PHPUnit.

6.2 Instalace

Hlavní monolitický repozitář, obsahující všechny moduly, lze naistalovat pomocí příkazu:

```
$ composer require novott20/MI-DIP_framework
```

či přidáním následující závislosti do souboru `composer.json`:

```
"require": {
  "novott20/MI-DIP_framework": "~1.0"
}
```

a následně provedením příkazu `$ composer update` pro aktualizování závislostí projektu.

6.2.1 Registrace

Registrace modulů je provedena přidáním cesty ke třídě `ServiceProvider` do pole `providers` v hlavním konfiguračním souboru `config/app.php`.

```
'providers' => [
  // ...framework service providers
  Modules\Loader\Providers\ServiceProvider::class,
]
```

Další možnosti nastavení registrace modulů jsou popsány v kapitole 6.5.1.

6.2.2 Konfigurace

Publikování konfiguračních souborů, lokalizovaných překladů a šablon pro všechny moduly, do příslušných složek aplikace, lze provést pomocí příkazů:

```
$ php artisan vendor:publish --tag="config"
$ php artisan vendor:publish --tag="translations"
$ php artisan vendor:publish --tag="views"
```

6.2.3 Nastavení databáze

Vytvoření potřebných tabulek v databázi pro běh modulů a vytvoření ukázkových dat lze provést pomocí následujících příkazů:

```
$ php artisan migrate
$ php artisan module:seed
```

6.3 Použité standardy a principy

Kód v implementační části této práce dodržuje standardy PSR–0 [16], PSR–1 [17], PSR–2 [18] včetně budoucího standardu PSR–12 [29]. Výsledný zdrojový kód dodržuje principy čistého kódu (SOLID [2], GRASP [3], DRY, SoC, ...).

6.3.1 Datové typy

Ve verzi PHP 7.0 byla přidána podpora skalárních typů[38] a možnost definovat návratové datové typy funkcí. Nastavení striktní kontroly dodržení datových typů je nutné v kódu přímo deklarovat:

```
declare(strict_types = 1);
```

Díku této deklaraci lze psát korektnější a sebedokumentující kód. Umožňuje také větší kontrolu nad kódem a usnadňuje jeho čtení.

6.3.2 Jmenné konvence

Pro jednotnou strukturu a celkovou přehlednost aplikace jsou definovány konvence pro názvy tříd (nad rámec zmíněných standardů PSR):

- Třídy závislé na konkrétní technologii musí obsahovat její název ve svém jméně, např.:
  - `MysqlDbPersister`, `MongoDbPersister`
  - `EloquentUserRepository`, `DoctrineUserRepository`
- Rozhraní (*interface*) musí mít příponu `Interface`.
- Abstrakní třídy musí mít předponu `Abstract`.
- Třídy musí příponu podle svého typu, tj.:
  - Třídy *Controller* musí mít příponu `Controller`.
  - Třídy *Forms* musí mít příponu `Form`.
  - Třídy *Request* musí mít příponu `Request`.
  - Třídy *Actions* musí mít příponu `Action`.
  - Třídy *Task* musí mít příponu `Task`.
  - Třídy *Provider* musí mít příponu `Provider`.
  - Třídy *Exception* musí mít příponu `Exception`.
  - Třídy *Seeder* musí mít příponu `Seeder`.
  - Třídy *Repository* musí mít příponu `Repository`.
  - Třídy *Factory* musí mít příponu `Factory`.

[38] *string*, *integer*, *float* a *boolean*

6.3.3 Kvalita kódu

Delší vývoj aplikace povede ke vzniku nekonzistencí v kódu – postupné zanesení různých chyb, zakomentovaný kód, překlepy, apod. Při práci na projektu je tak výhodné mít nějaký automatický systém pro kontrolu formátu a kvality kódu. Pro vícečlenné týmy je toto přímo nezbytností.

Monolitický repozitář obsahující moduly je díky CI[39] navázán na automatické testování včetně analýzy kódu. Výsledky této analýzy jsou dostupné na `https://scrutinizer-ci.com/g/novott20/MI-DIP_framework`.

6.4 Struktura repozitáře

Každý *open source* projekt by měl obsahovat určitá *metadata*[40], aby byl přehledný pro ostatní vývojáře. Tyto soubory pomáhají k pochopení k čemu modul slouží, jak ho je možné použít, zda je dobře a přehledně napsaný, pod jakou licencí je možné software používat a zda je projekt řádně otestovaný.

Nejjednoduší způsob, jak do repozitáře vložit všechna potřebná a běžně využívaná *metadata*, je využít předpřipraveného projektu od *The PHP League*[41]. Obsah tohoto projektu lze jednoduše zkopírovat do vlastního repozitáře a pomocí interaktivního příkazu `$ php prefill.php` naplnit správnými osobními údaji.

Takto je vytvořena struktura, jak hlavního monolitického repozitáře, tak jednotlivých modulů.

6.5 Moduly

V kapitole 3 byla provedena analýza možných řešení nastavení verzovacích repozitářů podporující modulární aplikaci. V implementační části této práce bude využito kombinace zmíněných postupů. Pro hlavní část aplikace, která je velmi provázána – běžně se používají všechny části v každém projektu (tak jako v případě frameworků) – bude použito metody monolitického repozitáře (viz kap. 3.3) se strukturou lokálních repozitářů pro jednoduchou práci se závislostmi (viz kap. 3.3.1).

Tento způsob je velmi jednoduchý na implementaci – není zde potřeba znalosti nestandardně používaných příkazů Gitu, ani jiných programů. Nevýhodu jednotného verzování zde převáží výhody rychlejší a snažší implementace. Navíc z tohoto řešení lze jednak vyjít pro implementaci zveřejňování repozitářů pomocí metody Splitsh (viz kap. 3.3.2), tak i případně rozdělit dosavadní vývoj do submodulů (viz kap. 3.3.3).

[39] *Continuous Integration* – Průběžná integrace
[40] Data, která poskytují informaci o jiných datech.
[41] `https://github.com/thephpleague/skeleton`

Moduly, které jsou používané pouze v nějakém případě (např. importy či exporty pro daný typ projektu, implementace jedné konkrétní platební brány, apod.), budou vytvořeny jako samostatné projekty ve vlastních repozitářích. Oddělení od hlavního repozitáře je především kvůli vlastnosti použití metody Splitsh, která vyžaduje aby všechny moduly měly stejnou verzi. Tyto repozitáře se ale nemění příliš často, a proto by bylo zbytečně neustále vyvíjet nové verze. Navíc, při použití správného vývojového prostředí, se i tyto moduly dají vyvíjet v rámci *monorepa* (i bez použití Git submodulů).

V následujících kapitolách jsou popsány moduly, do kterých je aplikace rozdělena, a jakou funkcionalitu dané moduly implementují.

6.5.1 Loader

Tento modul je inspirován balíčkem Laravel Modules (viz kap. 2.2.1) – kvůli jeho nedostatečné konfiguraci je obdobný modul implementován vlastní. Slouží k automatickému zaregistrování všech ostatních modulů do aplikace. Je tak možné v hlavním konfiguračním souboru (`config/app.php`) namísto uvedení třídy `ServiceProvider`[42] pro každý modul zvlášť, uvést pouze třídu tohoto modulu.

```
'providers' => [
  // ...framework service providers
  Modules\App\Providers\ServiceProvider::class,
  Modules\Support\Providers\ServiceProvider::class,
  Modules\Localization\Providers\ServiceProvider::class,

  // ...more module service providers
]
```

Obrázek 6.1: Původní způsob registrace více modulů

```
'providers' => [
  // ...framework service providers
  Modules\Loader\Providers\ServiceProvider::class,
]
```

Obrázek 6.2: Nový způsob registrace modulů

Seznam aktualně načítáných modulů lze zobrazit pomocí příkazu:

```
$ php artisan module:list
```

[42] Způsob jakým se ve frameworku Laravel registrují balíčky do aplikace.

Načítané moduly jsou definované konfiguračním souborem `module.json` umístěném v kořenové složce modulu, např.:

```
{
  "name": "Catalog",
  "namespace": "Modules\\Catalog\\",
  "priority": 1,
  "autoload": true
}
```

Nastavení, jaké složky má modul `Loader` prohledávat při načítání modulů, lze nastavit v konfiguračním souboru, či pomocí proměnné prostředí:

```
MODULES_PATH="/path/to/modules,/path/to/another/modules/"
```

Je také možné některé načítané moduly explicitně zakázat:

```
MODULES_BLACKLIST="app,support"
```

6.5.2 Support

Podpůrný modul obsahující různé třídy potřebné pro všechny ostatní moduly. Obsahuje abstraktní třídy implementující základní funkcionalitu jednotlivých komponent a jejich rozhraní (*interface*).

Mimo jiné také obsahuje třídy `ViewRenderer` pro zpracování šablon do výsledného HTML kódu (viz kap. 6.6.3), třídu `FormsBuilder` pro jednotné vytváření formulářů (viz kap. 6.6.4) a abtraktní třídy `Controllers` (viz kap. 6.6.2).

6.5.3 Bus

Tento modul implementuje abstraktní třídy a rozhraní (*interface*), popsané v kapitole 5, potřebné pro podporu servisních tříd dle návrhu v kapitole 4.2.

Rozšiřuje službu pro odbavování akcí[43], integrovanou ve frameworku Laravel, o možnost odbavovat servisní třídy. Kromě nové funkcionality je tak možné využít původní systém pomocí metody `dispatch()`:

```
use Illuminate\Foundation\Bus\DispatchesJobs

class TestController
{
  use DispatchesJobs;

  public function action()
  {
    $this->dispatch(new TestAction());
  }
}
```

[43] Služba je dostupná v DI kontejneru aplikace jako implementace splňující rozhraní `Illuminate\Contracts\Bus\Dispatcher`

6.5.4 App

Tento modul implementuje hlavní třídy jádra aplikace (`Http Kernel`, `Console Kernel` a `Exception Handler`). Není tak nutné implementovat stejné nastavení pro každý projekt (např. registrovaní tříd `Middleware` pro zabezpečí aplikace, transformaci odchycených chyb do JSON formátu, apod.).

Pro integraci tohoto modulu do aplikace je nutné upravit určité třídy aplikace (viz obr. 6.3). Tímto způsobem je stále možné provádět konfiguraci zvlášť pro každý projekt, a přitom zachovat společnou logiku implementovanou v modulu.

```php
namespace App\Exceptions;

use Modules\App\Engine\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
  //
}
```

```php
namespace App\Http;

use Modules\App\Engine\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel
{
    //
}
```

```php
namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Modules\App\Engine\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
  protected $commands = [
    // commands provided by your application
  ];

  protected function schedule(Schedule $schedule)
  {
      parent::schedule($schedule);
      // $schedule->command('inspire')->hourly();
  }
}
```

Obrázek 6.3: Integrace modulu `App` do aplikace

6.5.5 Authentication

Modul, který rozšiřuje již existující způsob autentizace, který je implentovaný ve frameworku Laravel. Obsahuje možnost autentizace jak pro webové rozhraní tak pomocí API. Autentizace API rozhraní je implementovaná pomocí knihovny Laravel Passport (viz kap. 6.1).

6.5.6 Authorization

Modul, který rozšiřuje již existující způsob autorizace, který je implentovaný ve frameworku Laravel. Rozšiřuje ho o uživatelské role a oprávnění, které lze dynamicky přirazovat jednotlivým uživatelům. Je tak možné nastavit mnoho různých rolí s různý stupněm oprávnění.

6.5.7 User

Modul sdružující moduly pro autentizaci a autorizaci s entitou uživatele. Bezchybně funguje s existujícím způsobem autentizace implementovaných ve frameworku Laravel.

Jelikož použitá entita pro autentizaci je závislá na aktuálním nastavení modulů, je nutné využít konfigurační soubor `auth.php` v tomto modulu. Je tak nutné smazat konfigurační soubor `config/auth.php`.

6.5.8 Localization

Podpůrný modul implementující lokalizaci aplikace. Poskytuje možnost nastavení povolených jazykových lokalizací a jejich přepínání – ať ručně či automaticky – v rámci aplikace.

6.5.9 Order

Modul obsahující jednoduchý systém e-commerce aplikace. Implementuje nákupní košík a následné vytvoření objednávky, která je připravená na integraci různých způsobů platby.

6.5.10 Catalog

Modul poskytující základní způsob vytváření produktů přispůsobených pro použití v nákupním procesu implementovaném v modulu `Order`.

6.6 Komponenty

V této kapitole je popsán objektový návrh a implementace některých komponent definovaných v kapitole 4.2. Všechny komponenty jsou implementovány tak, aby byla možná jejich výměna za jinou konkrétní implementaci (použitím rozhraní, či jinou konfigurací). Všechny komponenty jsou implementované způsobem, jaký je typický pro framework Laravel. Ve všech metodách je použita specifikace datových typů[44].

6.6.1 Servisní třídy

Modul `Bus` (viz kap. 6.5.3) implementuje potřebné rozhraní a abstrakní třídy pro implementaci servisních tříd `Action` a `Task`. Konkrétním servisním třídám pak stačí rozšiřovat třídu `AbstractAction` (resp. `AbstractTask`). Na obrázku 6.4 je zobrazena ukázka jak může vypadat implementace servisní třídy `AddItemToCartAction`.

Pro větší přehlednost je u každé implementované třídy rozsáhlá dokumentace (pomocí standardu PHPDoc[45]). Kromě dokumentace každé metody je zdokumentováno jaké další servisní třídy daná třída využívá (pokud se jedná o třídu `Action`), jaké výjimky (*Exceptions*) je možné očekávat, a jaké události (*Events*) mohou vznikat během běhu třídy[46].

Proměnná `$expectedExceptions` zařizuje, aby `ServiceDispatcher`, který odbavuje servisní třídy (viz kap. 5.2), mohl rozpoznat, které výjimky nejsou očekáváným chováním třídy[47], a které výjimky jsou skutečné chyby vzniklé v průběhu vykonávání servisní třídy. Při zjištěné chybě je proveden *rollback* akce zavoláním metody `failed()` v selhané servisní třídě.

[44] Podpora skalárních datových typů byla představena ve verzi PHP 7.0

[45] `https://docs.phpdoc.org/`

[46] Anotace `@event` neexistuje jako standard, jelikož vytváření událostí nemá přesně danou specifikaci a je specifické pro každé prostředí, resp. použitý framework.

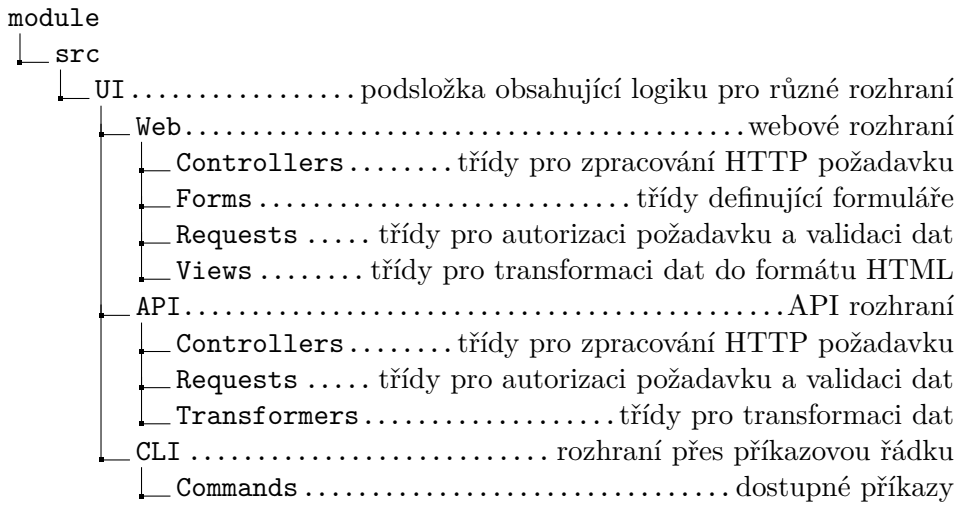
[47] Jedná se o běžnou praxi v programování – tímto systémem není nutné navracet chybové hlášky pomocí návratových hodnot.

```php
/**
 * @package Modules\Order

 * @author  Tomas Novotny <novott20@fit.cvut.cz>
 *
 * @uses    \Services\GetProductByIdTask
 * @uses    \Services\GetCartTask
 * @uses    \Services\ReduceReservedStockTask
 * @uses    \Services\AddItemToCartTask
 *
 * @event   \Events\ItemAddToCart;
 * @event   \Events\ReservedStockChanged;
 *
 * @throws  \Exceptions\ResourceNotFoundException
 * @throws  \Exceptions\NotEnoughStockException
 */
class AddItemToCartAction extends AbstractAction
{
  protected $expectedExceptions = [
    ResourceNotFoundException::class,
    NotEnoughStockException::class,
  ];

  private $productId;

  private $amount;

  public function __construct(int $productId, int $amount)
  {
    $this->productId = $productId;
    $this->amount    = $amount;
  }

  public function handle()
  {
    $product = $this->dispatchTask(
      new GetProductByIdTask($this->productId)
    );

    $cart = $this->dispatchTask(new GetCartTask());

    $this->dispatchTask(
      new ReduceReservedStockTask($product)
    );

    $this->dispatchTask(
      new AddItemToCartTask($cart, $product, $this->amount)
    );
  }

  public function failed(Throwable $exception)
  {
    if($exception instanceof NotAddToCartException) {
      $this->dispatchTask(
        new CalculateReservedStockTask($this->productId)
      );
    }
  }
}
```

Obrázek 6.4: Příklad implementace třídy `Action`

6.6.2 Controllers

V kapitole 5.4 ja popsána implementace UI pomocí vzoru *State*. Každý typ UI má ve struktuře modulu svou vlastní podsložku, ve které implementuje, mimo jiné, i své vlastní třídy `Controller`.



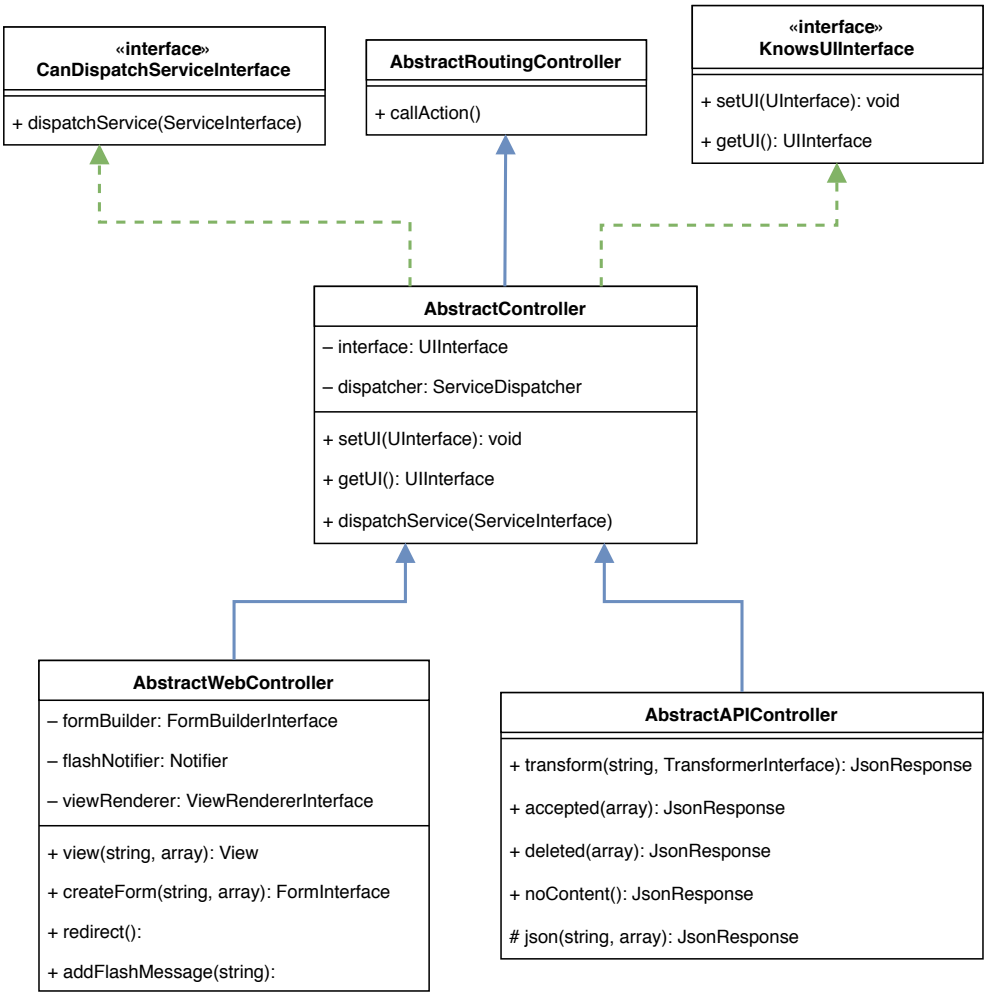
Obrázek 6.5: Struktura různých UI aplikace.

Tyto třídy mohou provádět různou logiku pro různé UI, nebo pro všechny stejnou a lišit se pouze ve formátu a struktuře odpovědi. Rozdělení do různých UI s vlastnímy třídami `Controller` je přehlednější – je jasné co jaké UI poskytuje za možnosti (ze seznamu existujících tříd) a nevzniká např. naprosto stejná podmínka v každé akci (metodě navázané na URL):

```
if ($request->isAjax()) {
  return $this->json($data);
}
else {
  return $this->view('path-to-template', $data);
}
```

Modul `Support` (viz kap. 6.5.2) implementuje abstraktní třídy pro aktuálně implementované UI, které obsahují pomocné metody pro typické funkcionality, které dané UI provádějí.

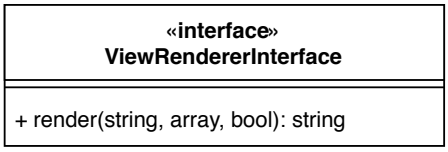
Třída `AbstractWebController` obsahuje metody pro vykreslování formulářů a celých šablon. Třída `AbstractApiController` zase obsahuje pomocné metody pro časté formáty odpovědí (viz obr. 6.6).



Obrázek 6.6: Objektový návrh třídy `Controller`

6.6.3 View Renderer

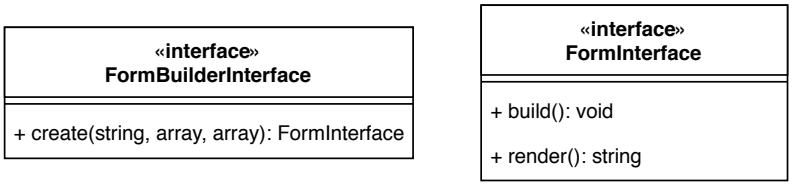
Moduly generují šablony přes třídu splňující rozhraní `ViewRendererInterface`. Je tak možné vytvořit vlastní implementaci, která např. využívá jiný šablonovací systém, bez nutnosti úpravy zbytku aplikace.



Obrázek 6.7: Objektový návrh třídy `ViewRenderer`

6.6.4 Forms Builder

Podpůrný modul `Support` definuje jednoduché rozhraní pro vytváření formulářů `FormBuilderInterface` a `FormInterface`. Není tak přímo napojená žádná konkrétní služba, a je možné využít vlastní řešení bez nutnosti úpravy zbytku aplikace. V základu je implementované řešení pomocí externího modulu Form Builder, který je uveden v seznamu použitých technologiích v kapitole 6.1.



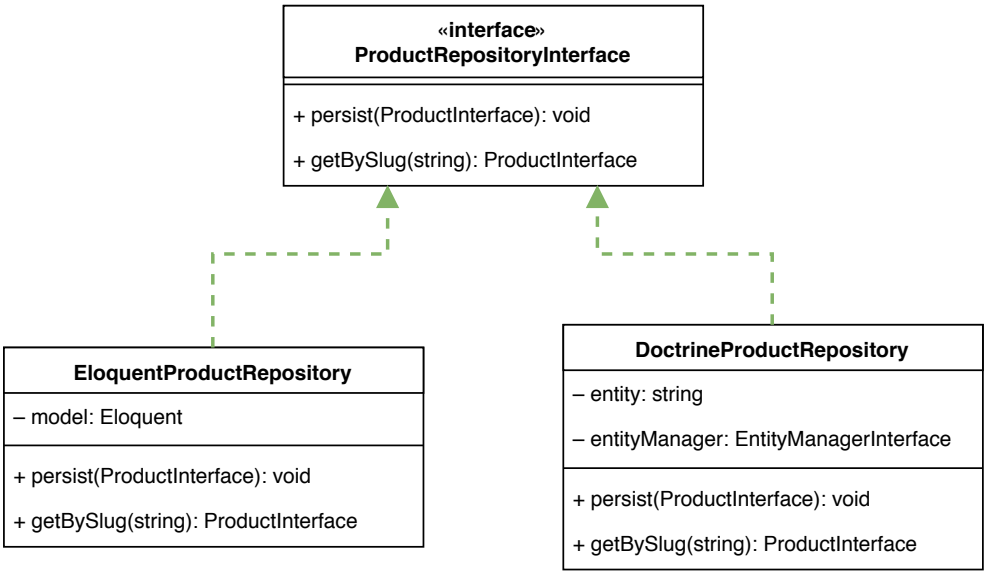
Obrázek 6.8: Objektový návrh třídy `FormBuilder`

6.6.5 Repozitáře

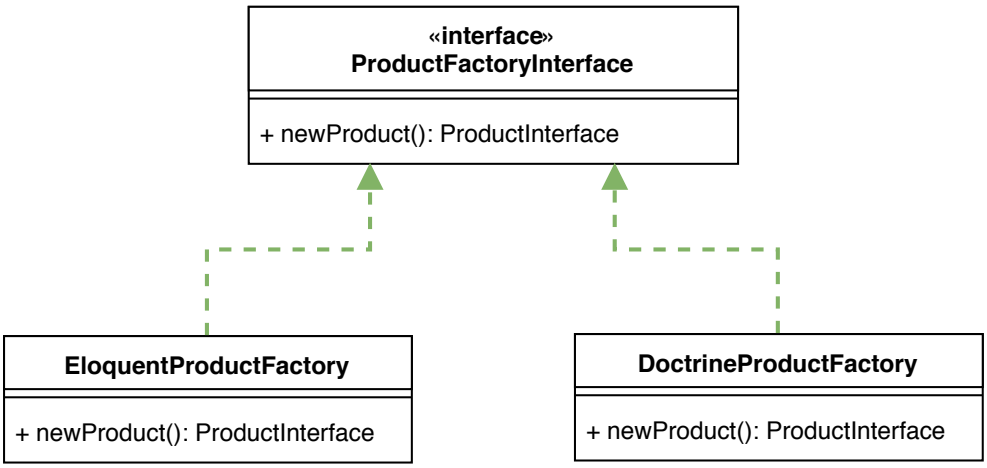
Dle popisu v kapitole 4.3 o datové vrstvě by implementace měla poskytovat možnost napojení vlastního datového modelu či ORM systému pro persistenci dat. Implementační část této práce podporuje ORM Doctrine2. Tento systém ovšem nemusí vždy vyhovovat – ať už z důvodu preference jiného ORM systému, či již existujícího datového návrhu.

Všechny repozitáře (*Repositories*), fabriky (*Factories*) i datové modely (*Entities*) implementují vlastní rozhraní (viz obr. 6.9 a 6.10). Napříč moduly tak existuje jednotné nastavení pro různé typy databázové vrsty. Implementace v aktuálním stádiu podporuje Eloquent a Doctrine2.

Konfigurací DI lze nastavit jaký systém bude v aplikaci využíván. Stejnou konfigurací lze nastavit použití jiného, vlastního řešení.



Obrázek 6.9: Objektový návrh třídy `ProductRepository`



Obrázek 6.10: Objektový návrh třídy `ProductFactory`

Kapitola 7

Testování implementace

7.1 Automatické testování

Monolitický repozitář, obsahující všechny moduly, je díky službě Travis CI[48] navázán na automatické testování. Servisní třídy `Action` a `Task` jsou testovány pomocí *white box*[49] integračních testů. Výsledky tohoto testování jsou veřejně dostupné na `https://travis-ci.org/novott20/MI-DIP framework`.

7.2 Testovaní rozšiřitelnosti

V kapitole 6.5 bylo popsáno do jakým modulů je implementovaná aplikace rozdělena. Tyto moduly lze využívat buď jako celý framework, který obsahuje všechny moduly, nebo jako jednotlivé moduly – díky využití monolitického repozitáře a nástroje Spliths (viz kap. 3). Cílem je vyvinout modulární aplikaci tak, aby si každý projekt mohl navolit jaké moduly bude používat. Jelikož tyto moduly implementují celkovou funkcionalitu webové aplikace (routování, zpracování HTTP požadavků, formátování odpovědi, ...) je nutné, aby moduly byly plně rozšiřovatelné – jen tak bude možné je využívat na větší počet projektů.

Na konkrétních projektech, které tyto moduly využívají, budou vždy vznikat nové požadavky, které nejsou implementované v základním řešení (tj. v modulech). Tam by ani implementované být neměly, protože se může jednat o specifický požadavek jednoho projektu, který na ostatních projektech není vyžadován.

Většina jiných modulů, balíčků, či knihoven rozšiřitelnost řeší pomocí konfigurace. Konfigurací ovšem vzniká pouze omezený počet možností řešení, které aplikace podporuje.

[48] `https://travis-ci.org/`

[49] Známe vnitřní strukturu, je možné lépe otestovat všechny průchody zdrojovým kódem.

Framework Laravel používá rozhraní (*interface*) pro určení, které konkrétní třídy budou použity. Tudíž lze použít vlastní implementaci pouhým předefinováním DI. Tento způsob je použit i v implementační části této práce. Lze tak určité části aplikace změnit, aniž by bylo nutné upravovat zbytek aplikace (princip otevřenosti a uzavřenosti [2]).

7.2.1 Testovací aplikace

Pro testovaní rozšiřitelnosti je vytvořena nová instalace frameworku Laravel:

```
$ git clone https://github.com/laravel/laravel.git
```

Do této aplikace jsou naistalovány implementované moduly poskytující základní funcionalitu e-commerce aplikace podle instalačního postupu popsaného v kapitole 6.2.

Následně je na typických případech demonstrováno, jak lze aplikaci rozšiřovat o nové požadavky.

7.2.2 Požadavky

Typicky požadované změny byly zjištěny v průběhu dvouletého vývoje obdobné aplikace, kdy jednotlivé projekty vycházely ze stejného jádra (řady modulů) obsahující společnou logiku.

Tabulka 7.1: Seznam požadavků na změnu aplikace

číslo	popis požadavku
1	Jiná šablona
2	Změna lokalizovaných překladů
3	Změna textu lokalizované URL
4	Změna celého požadavku
5	Propsání dodatečných dat do šablona
6	Jiná databázová vrstva
7	Rozšíření datového modelu
8	Jiný šablonovací systém
9	Změna formátu API
10	Různé oprávnění administrátorů
11	Optimalizace rychlosti
12	Optimalizace SQL dotazů
13	Změna aplikační logiky

7.2.2.1 Jiná šablona

Moduly implementují aplikační logiku a přeposílají výsledná data do šablon. Logika sice může být společná pro více projektů, ale vzhled aplikace bude vždy odlišný. Musí tedy existovat způsob jak využívat pro zobrazování dat vlastní šablony.

Existují dva způsoby jak splnit tento požadavek. První z nich je přepsání existujících šablon:

1. Publikování šablon modulů do lokální složky pomocí příkazu:

   ```
   $ php artisan vendor:publish --tag="views"
   ```

2. Úprava publikované šablony ve složce projektu. Šablony se publikují do složky `resources/views/modules/{module}/{UI}`, kde `{module}` je název modulu a `{UI}` je název uživatelského rozhraní (viz kap. 5.4).

Druhý způsob využití vlastní šablony pro zobrazení dat je změna konfigurace:

1. Publikování konfigurace modulů pomocí příkazu:

   ```
   $ php artisan vendor:publish --tag="config"
   ```

2. Vytvoření vlastní šablony (např. `custom-catalog.blade.php`).

3. Změna části konfigurace, která mapuje jaké šablony budou použity (např. `config/module-catalog.php` pro modul `Catalog`).

   ```
   'views' => [
     // ...more views
     'product-catalog' => 'custom-catalog',
   ],
   ```

7.2.2.2 Změna lokalizovaných překladů

Moduly obsahují lokalizované překlady (texty v šablonách, chybové hlášení či jiných informativních zpráv, ...).

1. Publikování překladů modulů do lokální složky pomocí příkazu:

   ```
   $ php artisan vendor:publish --tag="translations"
   ```

2. Úprava publikovaného souboru obsahující překlady ve složce projektu. Překlady se publikují do složky `resources/lang/modules/{module}`, kde `{module}` je název modulu.

3. Změna překladů v publikovaných souborech.

7.2.2.3 Změna textu lokalizované URL

Všechny URL definované v modulech jsou lokalizované. URL mají svůj kód, na který lze odkazovat bez nutnosti znát výsledný formát URL.

Změny tak docílíme stejným způsobem jako v případě změny překladů (viz kap. 7.2.2.2).

7.2.2.4 Změna celého požadavku

Pokud je vyžadováno příliš mnoho změn u jednoho požadavku (akce vázané na URL), je výhodnější vytvoření vlastní akce třídy `Controller`, která daný požadavek zpracuje.

1. Vytvoření vlastní třídy `Controller`:

```
use Modules\Support\Parents\Controllers\
    AbstractWebController;

class CustomController extends AbstractWebController
{
  public function action()
  {
    // .. apply custom logic

    return $this->view('custom-path');
  }
}
```

2. Přetížení URL v routování projektu (soubor `routes/web.php`):

```
Route::get('/test', [
  'uses' => 'TestController@action'
]);
```

7.2.2.5 Propsání dodatečných dat do šablon

Typický požadek projektů je zobrazení dodatečných dat v šablonách. Toho je možné docílit dvěma způsoby. Jedním z nich je celkové přetížení URL a vytvoření vlastního odbavení požadavku (viz kap. 7.2.2.4). Druhá možnost je využití globální události, která umožňuje přidat (či měnit) data (s využitím použití jiné šablony 7.2.2.1), aniž by bylo nutné zasahovat do zbytku aplikace:

1. Vytvoření nové třídy `Event-Listener` (viz kap. 4.2.2.3):

```
namespace App\Event\Listeners;

use Modules\Support\Events\ViewCreated;
use Modules\Support\Parents\AbstractListener;

class Listener extends AbstractListener
{
  //
}
```

2. Zaregistrování vytvořené třídy, aby naslouchala události `ViewCreated`:

```
// EventServiceProvider

protected $listen = [
  Modules\Support\Events\ViewCreated::class => [
    App\Event\Listeners\Listener::class,
  ],
];
```

3. Přidání potřebných dat pomocí metod třídy:

```
namespace App\Event\Listeners;

use Modules\Support\Events\ViewCreated;
use Modules\Support\Parents\AbstractListener;

class Listener extends AbstractListener
{
  public function handle(ViewCreated $event)
  {
    if ($event->isView('catalog')) {
      $event->addData('custom', 'data');

      $event->editData('custom', 10);

      $event->removeData('products');
    }
  }
}
```

7.2.2.6 Jiná databázová vrstva

V kapitole 4.3 byly popsány výhody oddělení datových modelů (*Entities*) od vrstvy starající se o persistenci dat (*Repositories*). Všechny třídy spojené s datovým návrhem (repozitáře, fabriky a datové modely) implementují vlastní rozhraní (viz kap. 6.6.5). Napříč moduly tak existuje jednotné nastavení pro různé typy databázové vrsty. Implementace podporuje ORM systémy Eloquent a Doctrine2

Pokud již existuje podporovaný systém, lze ho hromadně nastavit pro všechny moduly:

1. Přidání proměnné prostředí (např. v souboru `.env`):

```
BINDINGS_DRIVER="eloquent"
```

Pokud požadovaný ORM systém není podporovaný nebo je potřeba změnit pouze některé konkrétní třídy, lze úpravu provést přenastavením DI:

1. Přidání proměnné prostředí (např. v souboru `.env`):

```
BINDINGS_DRIVER="custom"
```

2. Publikování konfigurace modulů pomocí příkazu:

```
$ php artisan vendor:publish --tag="config"
```

3. Nastavení, jaké konkrétní třídy budou využívány pro dané rozhraní:

```
'bindings' => [
  'custom' => [
    ProductInterface::class =>
      CustomEloquentProduct::class,

    ProductRepositoryInterface::class =>
      CustomEloquentProductRepository::class,

     // ...more DI bindings
  ],
],
```

7.2.2.7 Rozšíření datového modelu

Rozšíření entity o další funkcionalitu, která není implementovaná v modulech (např. parametrizace, kategorizace, další textové pole, apod.). Zbytek entity zachovává původní implementaci.

1. Vytvoření nové entity rozšířením původní:

```
namespace App\Entities;

use Modules\Catalog\Model\Entities\Doctrine\Product;

class CustomProduct extends Product
{
  private $perex;

  public function getPerex(): string
  {
    return $this->perex;
  }
}
```

2. Publikování konfigurace modulů pomocí příkazu:

```
$ php artisan vendor:publish --tag="config"
```

3. Nastavení, aby pro dané rozhraní byla využívána tato konkrétní třída:

```
'bindings' => [
  'doctrine' => [
    ProductInterface::class =>
      App\Entities\CustomProduct::class,
  ],
],
```

7.2.2.8 Jiný šablonovací systém

Projekt chce využívat jiný šablonovácí systém namísto použitého šablonovacího systému Blade[50]. Může se jednat například o šablonovací systém Twig[51] používaný ve frameworku Symfony. Může být také vyžadováno použití čistého HTML kódu.

Systém generuje šablony pomocí třídy splňující specifické rozhraní (viz kap. 6.6.3). Je tak možné vytvořit vlastní implementaci a nastavit jeho použití v konfiguraci DI.

1. Vytvoření třídy splňující rozhraní `ViewRendererInterface`:

```
use Modules\Support\Contracts\ViewRendererInterface;

class TwigRendered implements ViewRendererInterface
{
  public function render(
    string $view,
    array  $data = [],
    bool   $cached = true,
    array  $additionalKeyData = []
  ): string {

    // ... process twig template
  }
}
```

2. Konfigurace DI ve třídě `AppServiceProvider` (či jiném *ServiceProvider*):

```
public function register()
{
  $this->app->bind(
    ViewRendererInterface::class, TwigRendered::class
  );
}
```

7.2.2.9 Změna formátu API

Je vyžadována změna formátu dat přístupných přes API rozhraní. Všechny třídy `Transformer` splňují vlastní rozhraní, tudíž lze pomocí DI konfigurovat, která konkrétní třída se použije pro transformaci dat.

[50] `https://laravel.com/docs/master/blade`
[51] `https://twig.symfony.com/`

1. Vytvoření vlastního transformátoru:

```
use Modules\Support\Parents\AbstractTransformer;

class CustomUserTransformer extends AbstractTransformer
    implements UserTransfomerInterface
{
  public function transform(UserInterface $user)
  {
    return ['email' => $user->getEmail()];
  }
}
```

2. Konfigurace DI ve třídě `AppServiceProvider` (či jiném *ServiceProvider*):

```
public function register()
{
  $this->app->bind(
    UserTransfomerInterface::class,
    CustomUserTransformer::class
  );
}
```

7.2.2.10 Různé oprávnění administrátorů

V kapitole 6.5.6 bylo uvedeno, že modul `Authorization` implementuje role a oprávnění jako data uložená v databázi. Oprávnění jednotlivých uživatelských rolí tedy není přímo určené ve zdrojovém kódu aplikace. Je tak možné vytvářet libovolný počet uživatelských rolí s různým oprávněním.

7.2.2.11 Optimalizace rychlosti

Dobře použitelná webová aplikace musí být především dostatečně rychlá, aby uživatele neodrazovala od svého použití.

Měření rychlosti může probíhat na různých úrovních aplikace. Doba zpracování HTTP požadavku je sice ve výsledku nejdůležitější metrikou, ale pro identifikaci problému je nedostatečná. Měření databázové vrstvy (doby běhu SQL dotazů) může odhalit špatně napsané SQL dotazy[52], ale neodhalí zpomalení aplikace v důsledku sekvence několika pomalejších dotazů do databáze.

Pomocí navrženého způsobu zpracování aplikační logiky – pomocí servisních tříd a třídy `ServiceDispatcher` (viz kap. 5) – je možné měřit dobu běhu jednotlivých servisních tříd. Je tak možné odhalit, které akce aplikace jsou pomalé v závislosti na aktuálním stavu aplikace (čase, autentizovaném uživateli, zpracovávaných datech, apod.).

Měření doby běhu servisních tříd lze implementovat použitím návrhového vzoru *Decorator* na třídu `ServiceDispatcher`:

[52] Velmi často se jedná o dotaz na větší počet tabulek spojených pomocí klauzule *JOIN* způsobené špatně generovaným SQL dotazem systémem ORM

1. Vytvoření nové třídy `ServiceDispatcher`:

```
use Illuminate\Contracts\Bus\Dispatcher

class ProfilingServiceDispatcher implements Dispatcher
{
  private $dispatcher;

  public function __construct(ServiceDispatcher $dispatcher)
  {
    $this->dispatcher = $dispatcher;
    $this->logget     = new Logger();
  }

  public function dispatchService(ServiceInterface $service)
  {
    $startTime = microtime(true);

    try {
      $result = $this->dispatcher->dispatchService($service)
          ;

      $endTime = microtime(true);
      $time    = $endTime - $startTime;

      // ... add parameters, auth user, etc.
      $this->logger->info([
        'duration' => $time,
        'class'    => get_class($service),
      ]);

      return $result;
    }
    catch (Throwable $exception) {
      $endTime = microtime(true);
      $time    = $endTime - $startTime;

      $this->logger->error([
        'duration' => $time,
        'class'    => get_class($service),
      ]);

      throw $exception;
    }
  }
}
```

2. Konfigurace DI ve třídě `AppServiceProvider` (či jiném *ServiceProvider*):

```
public function register()
{
  $this->app->bind(
    Dispatcher::class, ProfilingServiceDispatcher::class
  );
}
```

7.2.2.12 Optimalizace SQL dotazů

Pokud je zjištěno, že některý SQL dotaz trvá příliš dlouho (např. pomocí měření servisních tříd uvedeného v kapitole 7.2.2.11), je možné provést optimalizaci v příslušné třídě `Repository` (přepsaní metody, které vytváří daný SQL dotaz). Jedná se tedy o rozšíření datového modelu, který je popsán v kapitole 7.2.2.7.

7.2.2.13 Změna aplikační logiky

Servisní třídy nesplňují žádné rozhraní (*interface*) definující konkrétní třídu, pouze společné rozhraní `ServiceInterface` (viz kap. 5.3). Nelze tedy nastavit, která konkrétní třída bude prováděna, pouze pomocí konfigurace DI. Tento způsob je žádanou funkcionalitou – je tak zajištěno, že nevznikne chyba v aplikaci z důvodu úpravy servisní třídy, pro použití v jednom *Use Case*, která je ovšem využívána i na jiném místě v aplikaci. Je tak nutná změna celého požadavku (viz kap. 7.2.2.4).

Samotná změna aplikační logiky probíhá vytvořením nové servisní třídy, která využívá již existující servisní třídy `Task`, či implementuje zcela novou funkcionalitu.

Závěr

Cílem práce bylo vytvořit co nejlepší objektový návrh pro vývoj webových aplikací, kde by byly jasně definované zodpovědnosti, při dodržení principů čistého kódu. Implementovaná aplikace dodržuje modulární architekturu a je navržena tak, aby jednotlivé moduly byly plně rozšiřitelné.

Byla provedena analýza existujících řešení s cílem vyvarování se chyb v existujících implementacích. Jako inspirace pro vlastní návrh architektury bylo využito architektonického návrhového vzoru Porto (viz kap. 2.2.3). Samotná implementace tohoto návrhu byla vytvořena samostatně.

V kapitole 3 byly probrány možné způsoby vývoje modulárních aplikací a jejich správy ve verzovacích repozitářích. Z nich byl vybrán způsob monolitického repozitáře s vnitřní strukturou umožňující rozdělení na samostatné moduly. Samotného rozdělení na více repozitářů je dosaženo využitím nástroje Splitsh (viz kap. 3.3.2). Aplikace je navržena tak, aby podporovala modulární vývoj, ale zároveň zachovala co nejlepší možnou přehlednost a umožnila pohodlný vývoj a snadnou udržitelnost celé aplikace.

Podle navržené metodiky byla implementována sada modulů, které implementují plně rozšiřitelnou e-commerce aplikaci pro PHP framework Laravel. Aplikační logika je implementována pomocí servisních tříd dodržující jednotnou hierarchickou strukturu (viz kap. 5). Tento návrh napomáhá vytvářet malé přehledné třídy s jasnou zodpovědností. Závislosti tříd jsou implementovány pomocí rozhraní. Implementace tudíž není závisla na konkrétních technologiích či třídách. Aplikaci tedy lze jednoduše rozšiřovat bez nutnosti úpravy zdrojového kódu modulů (viz kap. 6.6).

Výsledná implementace byla otestována podle typických požadavků projektů, které využívají řadu modulů jako společné jádro aplikace (viz kap. 7). Na těchto požadavcích bylo ukázáno, že výsledná aplikace je plně rozšiřitelná.

Literatura

[1] Glass, R. L.: *Facts and fallacies of software engineering*. Boston, MA: Addison-Wesley, 2003, ISBN 978-0321117427.

[2] Martin, R. C.: *Čistý kód*. Brno: Computer Press, vyd. 1 vydání, 2009, ISBN ISBN-978–80–251–2285–3.

[3] Larman, C.: *Applying UML and patterns*. Upper Saddle River, N.J.: Prentice Hall PTR, 2005, třetí vydání, ISBN 978-013-1489-066.

[4] Laine, J.: *Should You Use a PHP Framework? Five Pros and Cons [online]*. 2017, [cit. 2016-05-04]. Dostupné z: `https://code.tutsplus.com/tutorials/should-you-use-a-php-framework-five-pros-and-cons--cms-28905`

[5] Balter, B.: *6 motivations for consuming or publishing open source software [online]*. 2015, [cit. 2018-04-08]. Dostupné z: `https://opensource.com/life/15/12/why-open-source`

[6] Croft, J.: *Frameworks for Designers [online]*. 2007, [cit. 2018-03-06]. Dostupné z: `http://alistapart.com/article/frameworksfordesigners`

[7] Skvorc, B.: *The Best PHP Framework for 2015: SitePoint Survey Results*. Duben 2015, [cit. 2018-03-06]. Dostupné z: `http://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/`

[8] *The State of PHP MVC Frameworks in 2017*. 2016, [cit. 2018-04-09]. Dostupné z: `https://www.zenofcoding.com/2017/02/27/the-state-of-php-mvc-frameworks-in-2017-laravel-symfony-codeigniter-cakephp-zend/`

[9] *PHP MVC Frameworks preview of 2018*. 2017, [cit. 2018-04-09]. Dostupné z: `https://www.zenofcoding.com/2017/12/31/php-mvc-frameworks-preview-of-2018-phalcon-3-symfony-4-laravel-5-x-and-others/`

[10] Otwell, T.: *Twitter účet Taylora Otwella [online]*. [cit. 2018-03-06]. Dostupné z: https://twitter.com/taylorotwell

[11] Otwell, T.: *On laravel's Future: Part 2 [online]*. [cit. 2018-03-06]. Dostupné z: https://laravel-news.com/2014/11/laravels-future-part-2/

[12] Otwell, T.: *Laravel Philosophy [online]*. [cit. 2018-03-06]. Dostupné z: https//laravel.com/docs/4.2/introduction

[13] Surguy, M.: *History of Laravel PHP framework, Eloquence emerging [online]*. [cit. 2018-03-06]. Dostupné z: http://maxoffsky.com/code-blog/history-of-laravel-php-framework-eloquence-emerging/

[14] Way, J.: *Webový portál Laracast [online]*. [cit. 2018-03-06]. Dostupné z: https://laracasts.com

[15] *The MIT License (MIT) [online]*. [cit. 2018-03-06]. Dostupné z: https://opensource.org/licenses/MIT

[16] *PSR-0: Autoloading Standard [online]*. [cit. 2018-03-06]. Dostupné z: http://www.php-fig.org/psr/psr-0/

[17] *PSR-1: Basic Coding Standard [online]*. [cit. 2018-03-06]. Dostupné z: http://www.php-fig.org/psr/psr-1/

[18] *PSR-2: Coding Style Guide [online]*. [cit. 2018-03-06]. Dostupné z: http://www.php-fig.org/psr/psr-2/

[19] Phalcon Team: *Benchmarking Phalcon [online]*. 2017, [cit. 2016-05-06]. Dostupné z: https://blog.phalconphp.com/post/benchmarking-phalcon

[20] Vítek, R.: *How to maintain multiple Git repositories with ease [online]*. 2017, [cit. 2018-04-09]. Dostupné z: https://blog.shopsys.com/how-to-maintain-multiple-git-repositories-with-ease-61a5e17152e0

[21] Shopsys: *Shopsys Framework Package [online]*. Shopsys, 2018, [cit. 2016-05-05]. Dostupné z: https://github.com/shopsys/framework/

[22] Zalt, M.: *Porto (Software Architectural Pattern) [online]*. 2017, [cit. 2018-04-08]. Dostupné z: https://github.com/Mahmoudz/Porto

[23] Goode, D.: *Scaling Mercurial at Facebook [online]*. 2014, [cit. 2018-04-07]. Dostupné z: https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/

[24] Potvin, R.: *Why Google Stores Billions of Lines of Code in a Single Repository [online]*. [cit. 2018-03-06]. Dostupné z: `https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext`

[25] Preston-Werner, T.: *Semantic Versioning 2.0.0 [online]*. 2017, [cit. 2018-04-30]. Dostupné z: `https://semver.org/`

[26] Evans, E.: *Domain-driven design*. Boston: Addison-Wesley, c2004, ISBN 978-0321125217.

[27] *Testing [online]*. [cit. 2016-04-18]. Dostupné z: `http://silex.sensiolabs.org/doc/testing.html`

[28] Way, J.: *Laravel Testing Decoded [online]*. Leanpub, 2013.

[29] *PSR-12: Extended Coding Style Guide [online]*. [cit. 2018-04-23]. Dostupné z: `https://github.com/php-fig/fig-standards/blob/master/proposed/extended-coding-style-guide.md`

Příloha A

Seznam použitých zkratek

ACL (Access Control List). Seznam oprávnění na objekt.

API (Application Programming Interface)

CI (Continuous Integration). Označení pro nástroje, které mají za úkol ověřovat kvalitu software při vývoji. Integrace změn má být podle definice průběžná.

CLI (Command-line Interface). Uživatelské rozhraní ovládané přes příkazovou řádku.

CRUD (Create, Read, Update, Delete) dokumentů.

DI (Dependency Injection) je technika předávání závislostí mezi službami. V momentu inicializace objektu dochází k předávání závislostí pomocí veřejného rozhraní třídy (například pomocí konstruktoru).

DIC (Dependency Injection Container). Označení pro kontejnér služeb, který se využívá v návrhovém vzoru Dependency injection.

DRY (Don't repeat yourself)

DTO (Data Transfer Object)

HTML (HyperText Markup Language). Značkovací jazyk využívaný při vývoji webových stránek.

HTTP (Hypertext Transfer Protocol). Internetový protokol pro přenos dokumentů ve formátu (X)HTML.

IoC (Inversion of Control). Obrácené řízení je návrhový vzor, který umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami.

IDE (Integrated Development Environment). Software pro vývoj aplikací integrující veškeré potřebné funkce pro pohodlnou práci s určitým jazykem.

JSON (JavaScript Object Notation)

KISS (Keep It Simple, Stupid!)

MVC (Model-view-controller) Architektura rozdělující aplikaci do tří vrstev. Na vrstvu datovou, pohledovou (často zastoupena šablonovacím jazykem) a logickou.

OOP (Object-oriented Programming). Technika vývoje software, která využívá koncepci objektů. Nejčastějšími stavebními prvky OOP jsou rozhraní, třídy a jejich metody.

ORM (Object-relational mapping), neboli objektově relační mapování je vrstva, která zajišťuje mapování dat z relační databáze na objekty programovacího jazyka a naopak.

PDO (PHP Data Objects). Rozšíření, které umožňuje pracovat jednotně s různými databázemi.

PHP (PHP: Hypertext Preprocessor). Skriptovací jazyk primárně využívaný pro programování dynamických webových stránek.

REST (Representational state transfer). Způsob jak jednoduše vytvořit, číst, editovat nebo smazat informace ze serveru pomocí jednoduchých HTTP volání.

SoC (Separation of concerns)

SQL (Structured Query Language). Standardizovaný dotazovací jazyk využívaný v relačních databázích.

UI (User interface)

URL (Uniform Resource Locator). Řetězec pevně daného tvaru, který popisuje přesnou lokaci zdroje na internetu.

XML (eXtensible Markup Language). Standardizovaný značkovací jazyk, který vznikl jako zjednodušená varianta staršího jazyka SGML.

Příloha **B**

Obsah přiloženého CD

- `implementation` ..... zdrojové kódy implementace
- `thesis`
  - `images` ..... obrázky
  - `biblography.bib` ..... literární zdroje
  - `DP_Novotny_Tomas.pdf` ..... výsledná forma práce ve formátu PDF
  - `DP_Novotny_Tomas.tex` ..... zdrojová forma práce ve formátu LaTeX