FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Knihovna pro tvorbu REST API v PHP
Student:	Martin Fabík
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

1. Analyzujte nejčastěji používané knihovny pro tvorbu REST API v jazyce PHP a porovnejte je (použitelnost, podpora, limitace, apod...).
2. Analyzujte nejčastěji požadované a nejčastěji chybějící funkcionality ve stávajících řešeních.
3. Na základě provedených analýz navrhněte vlastní knihovnu, která bude pokrývat nejčastější chybějící funkcionality.
4. Implementujte navrženou knihovnu v jazyce PHP, včetně vhodných testů (unit, integrační, apod...).
5. Navrhněte vhodnou referenční implementaci vytvořené knihovny, včetně vhodné ukázkové REST API služby.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 16. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Knihovna pro tvorbu REST API

Martin Fabík

Katedra softwarového inženýrství

Vedoucí práce: Ing. Jiří Chludil

4. června 2020

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. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. června 2020 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Fabík, Martin. *Knihovna pro tvorbu REST API.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Cílem práce je analýza stávajících možností pro tvorbu REST API služeb dostupných v různých PHP frameworcích a na základě vzniklé analýzy identifikovat nejčastěji chybějící funkcionality a navrhnout řešení požadovaných funkcionalit. Praktickým výstupem této práce bude nezávislá PHP knihovna, která nabídne nástroje a možnosti pro tvorbu REST API služeb. V písemné části se autor zabývá převážně analýzou stávajících řešení spolu s rozborem požadavků a návrhem konkrétní podoby výsledné knihovny.

Klíčová slova REST API, REST frameworky, REST API nástroje, návrh API, OpenAPI, PHP knihovna

Abstract

The goal of this thesis is to analyze the existing possibilities for creating REST API services in various PHP frameworks and identify the most frequently missing functionalities based on provided analysis and then propose solution for the required functionalities. The practical output of this work will be an independent PHP library that will offer various tools for creating REST API services. In written part, the author deals mainly with the analysis of existing solutions and analysis of requirements and proposes design of the resulting library.

Keywords REST API, REST frameworks, REST API tools, API design, OpenAPI, PHP library

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V současnosti, kdy se čím dál tím více subjektů na internetu snaží využívat výhod REST API služeb[1], je důležité, aby měli vývojáři k dispozici kvalitní nástroje pro vývoj právě REST API služeb. V práci proto cílím převážně na analýzu možností, které má v současnosti vývojář k dispozici a následně navrhuji knihovnu, která pokrývá důležité chybějící funkcionality.

V úvodu práce se zabývám podrobnější analýzou dostupných frameworků z různých odvětví, se kterými se vývojář může potkat. Během analýzy zkoumám dostupná řešení z pohledu tvorby REST API serveru, tedy především jejich zacházení s daty. Z tohoto pohledu jsem v průběhu analýzy při seznamování se s frameworky využil jak drobných implementací, tak revize kódu.

Práce je strukturovaná tak, aby čtenáři přehledně a srozumitelně poskytla výstupy jednotlivých částí tvorby softwaru. Během analýzy se tedy zabývám identifikací klíčových funkcionalit, které považuji za důležité a které ve většině řešení chybí, nebo různé frameworky poskytují implementaci na rozdílné úrovni. Jako ověření mých hypotéz byl proveden také průzkum mezi vývojáři. tento průzkum cílil na jejich potřeby a jejich stávající řešení. Výstupem práce je poté samostatná PHP knihovna jako nástroj umožňující efektivní tvorbu REST API služby stejně tak jako referenční použití této knihovny ve frameworku Nette.

Kapitola 1

Cíl práce

Cílem této práce je na základě analytické části práce navrhnout a realizovat PHP knihovnu pro tvorbu REST API služeb (serverové části).

Cílem analytické části práce je vytipovat a analyzovat zajímavé PHP frameworky a posoudit možnosti tvoby *REST API* v těchto frameworcích. Dále identifikovat klady a zápory zvolených řešení a vytipování a konkrétní specifikace užitečných funkcionalit. Na základě vyspecifikovaných funkcionalit posléze navrhnout samostatně použitelnou PHP knihovnu, která poslouží jako nástroj pro tvoru REST API služeb.

Cílem praktické části je realizace výše zmíněné knihovny v jazyce PHP, její otestování a také ukázková implementace a použití nově vzniknuvší knihovny.

Kapitola 2

REST

Před samotným zkoumáním dostupných řešení a tvorbou vlastních požadavků, je nutné nejdříve znát, co REST API obnáší, na jakých principech je postaveno a jakou používá architekturu. V tomto krátkém, avšak důležitém úvodu se budu zabývat převážně serverovou částí REST API, avšak je nutná také znalost z pohledu klienta.

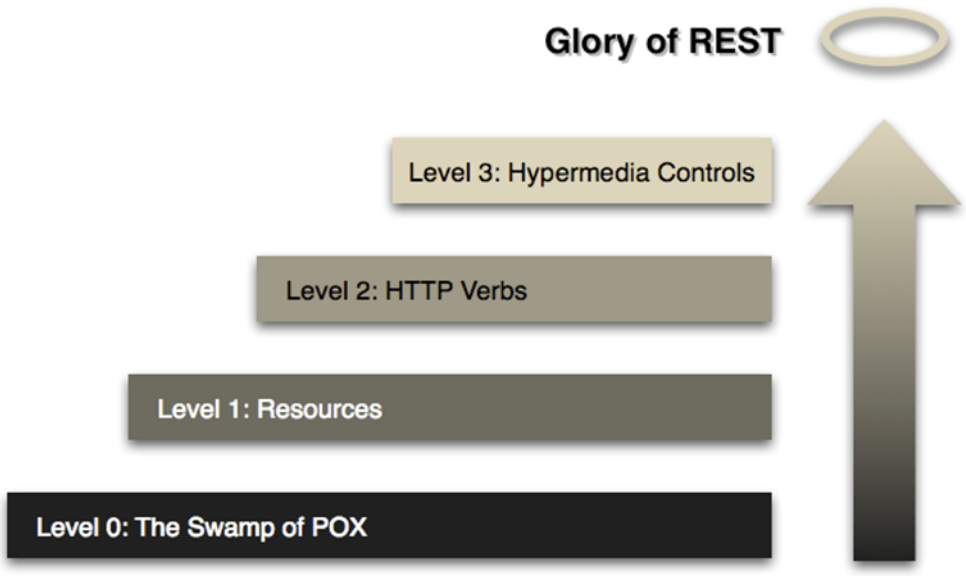
2.1 Základy RESTu

REST [2] (Representational State Transfer) je architektonický styl, který je narozdíl od tehdy dosupných technologií, jako například XML-RPC nebo SOAP, datově orientován. Primárním zaměřením REST API jsou rozhraní v distribuovaném prostředí, přičemž *„se zaměřuje na jednotný a snadný přístup ke zdrojům (resources). Zdroj může být prakticky cokoliv, koncept nemá žádná omezení. Může to být například nějaký konkrétní objekt v databázi, dokument, výsledek nějakého výpočtu nebo webová stránka“* [3].

Jelikož se tato práce zaobírá návrhem a implementací REST API služeb z pohledu RESTu se jedná o návrh a definování **zdrojů**.

2.2 Úrovně RESTu

[4] Pro přehlednost je výhodné si REST znázornit v úrovních jako na obrázku 2.1:



Obrázek 2.1: Čtyři úrovně pohledu na REST[4]

2.2.1 The Swamp of POX

Jedná se o úroveň, na které pohlížíme na samotný přenos mezi serverem a klientem. Nejčastěji se používá protokol HTTP, avšak samotný REST se neváže jen na tento konkrétní protokol. V této práci se však budu zabývat pouze přenosem pomocí HTTP protokolu, jelikož se jedná o práci pojednávající o tvorbě REST API v prostředí webu.

2.2.2 Resources

Při práci s REST API nejsou požadavky z pohledu klienta posílány na jeden centrální bod a dále rozlišovány podle obsahu, ale jsou klientem zasílány na jednotlivé, jednoznačně identifikované zdroje. Zdrojem může být cokoli, tedy různé dokumenty, obrázky, ale nejčastěji se jedná o strukturovaná data, například data z databáze.

Dále je klientům umožněno přistupovat k jednotlivým instancím jednoho zdroje (můžeme si také představit jako prvky kolekce). Například.:

- **/article** - manipulace se seznamem článků (kolekce)
- **/article/:id** - manipulujeme s jedním konkrétním prvkem kolekce, identifikovaným podle *:id*

Pokud pracujeme se strukturovanými daty, téměř vždy narazíme na problém, jak řešit relace. Ačkoli samotný REST nedefinuje způsob, jak relace řešit (klade pouze požadavek na to, aby byl každý zdroj jednoznačně identifikován), je nepsaným pravidlem uvádět relace v následujícím tvaru: `<zdroj>/:idZdroj/<relace>/:idRelace`. Pokud toto promítnu v konkrétních případech, pak:

- **/article/1/comment** - seznam všech komentářů pro článek 1 (kolekce)
- **/article/1/comment/2** - konkrétní komentář ke konkrétnímu článku

Pro tuto konkrétní práci již z této definice vyplývá důležitá a nutná vlastnost, a tou je práce s parametrickými URL.

2.2.3 HTTP Verbs

Jelikož se zabýváme přenosem dat pomocí protokolu HTTP, představíme si v této kapitole, jaké metody nám HTTP protokol nabízí a které jsou RESTem používané. V tabulce 2.1 naleznete název a význam použitých metod.

GET	Získání dat
POST	Vytvoření dat
PUT	Úprava dat
DELETE	Odstranění dat
PATCH	Částečná úprava dat

Tabulka 2.1: HTTP metody pouzivane v souvislosti s REST

Kromě těchto definic se můžeme setkat ještě s vyjádřením, že metoda *GET* je *safe*. To znamená, že při volání této metody nedochází k žádné změně stavu nad volaným zdrojem. Kromě toho, že je metoda *safe* se dále používá označení *idempotent* a to pro metody *GET*, *PUT* a *DELETE*. To znamená, že opakované volání stejného zdroje se stejnými daty a stejnou metodou vede ke stejné změně stavu nehledě na počet volání.

V praxi je poté použití metody *PATCH* často opomíjené a používají se často pouze první čtyři zmíněné metody [3]. Částečným důvodem může být také to, že metoda *PATCH* byla do HTTP protokolu zakomponována jako

rozšíření dodatečně [5]. Z pohledu RESTu je pak nutné implementovat čtyři základní operace, označované jako *CRUD*. Jedná se o *Create*, *Retrieve*, *Update* a *Delete*. Jak vidíme, protokol HTTP je zcela dostačující pro pokrytí požadovaných metod.

Kromě různých metod, které lze použít při dotazování se na zdroj nám HTTP poskytuje také sadu stavových kódů pro odpověď. Jejich kompletní seznam společně s významem je k dispozici v oficiální dokumentaci.

2.2.4 Hypermedia Controls

Poslední úroveň je známa pod akronymem HATEOAS (*Hypertext as the Engine of Application State*). Popisuje princip, kdy každý zdroj kromě samostných dat poskytne také seznam operací, které lze využít v souvislosti s daným požadavkem. To přináší řadu výhod, například odpadá nutnost udržovat na straně klienta všechny adresy zdrojů, ale po prvotním dotazu na server klient získá seznam dostupných operací.

Ačkoli však tento přístup přináší své výhody, v současnosti není až na drobné výjimky využíván zejména díky chybějícím nástrojům jak efektivně přenášet tyto odkazy.

Kapitola 3

Analýza

Během analýzy se zaměřuji na vydefinování očekávané funkcionality a porovnání s dostupnými řešeními. Mezi dostupná řešení se snažím vybrat zajímavé frameworky, se kterými se může vývojář v různých odvětvích setkat.

3.1 Výběr dostupných řešení

V této kapitole porovnávám aktuálně dostupná řešení, která se snažím vybírat z různých odvětví, jelikož ne vždy má programátor možnost zvolit si framework, ve kterém bude své řešení implementovat.

3.1.1 Dostupné frameworky

3.1.1.1 Laravel[6]

Tento framework se řadí mezi ty populárnější frameworky a je velice oblíben pro svou jednoduchost a tedy je vhodný pro rychlé prototypování. Laravel sám o sobě obsahuje svůj vlastní ORM framework Eloquent. Vybral jsem jej zejména kvůli jeho oblíbenosti mezi vývojáři.

3.1.1.2 Symfony[7]

Dlouhodobě stabilní s poměrně rozšířený framework, který je postaven na sadě samostatně použitelných komponent. Tento koncept je u tohoto frameworku velice oceňován a mnoho vývojářů používá práve samostatných komponent frameworku. Byl vybrán pro svou rozšířenost mezi vývojáři.

3.1.1.3 Magento 2[8]

Jeden z největších e-commerce frameworků, které se na trhu nacházejí. Je velice robustní a poskytuje nepřeberné množství modulů, mnohdy postavených

na komponentách ostatních frameworků, jako je Zend nebo Symfony. Framework poskytuje mimo jiné také velmi propracovaný podpůrný systém pro tvorbu REST API a proto byl zahrnut do této práce.

3.1.1.4 Nette[9]

Mezi vývojáři v české republice velice populární framework postavený na MVC architektuře. Je jednoduchý, snaží se o používání komponent a je také velmi snadno použitelný. Do této práce jsem jej zahrnul pro svou oblíbenost na českém trhu. Pro účely této práce bylo hodnoceno nette s rozšířením Apitte, které jsem shledal asi nejlepším rozšířením pro REST API pro tento framework.

3.2 Zhodnocení dostupných řešení

V této kapitole se zabývám zhodnocením výše uvedených frameworků a knihoven podle definovaných kritérií.

3.2.1 Výběr hodnotících kritérií

Než bude možné zhodnotit vybraná řešení, je nutné si stanovit hodnotící kritéria. Hodnocení bude probíhat na škále od 1 do 5, jako ve škole. Tedy 1 znamená nejlepší a 5 nejhorší.

Dokumentace

Pro správné a efektivní používání frameworku/knihovny potřebuji kvalitní dokumentaci. Pokud zvolené řešení nemá kvalitně vypracovanou dokumentaci a já narazím na problém, může se mi používání značně zkomplikovat. Stejně tak pokud nemám informaci o všech možnostech daného řešení, nemusím plně využít jeho potenciál. Z tohoto důvodu ze při hodnocení dokumentace zaměřuji na:

- Obsáhlost – tedy zda obsahuje dostatek informací potřebných pro používání dokumentace
- Zda dokumentace obsahuje příklady použití u jednotlivých konceptů, případně v jaké míře
- Přehlednost – tedy jestli je dokumentace přehledně strukturovaná, zda má logické členění a zda se dostanu všude tam, kam potřebuji

Jednoduchost používání

Knihovnu/framework si vybírám proto, aby mi pomohla. Pokud budu nucen při návrhu REST API řešit věci, které přímo nesouvisejí s návrhem, pak je dané řešení hodnoceno negativně. Jednoduchost používání je zde posuzována z pohledu usnadnění návrhu REST API (jak mi framework pomáhá při jeho tvorbě).

Podpora různých formátů

Rozšiřitelnost z hlediska podpory různých aplikačních formátů, nejčastěji JSON nebo XML. Od vybraného řešení očekávám flexibilní podporu více formátů, jako i jejich jednoduchou rozšiřitelnost. Tato vlastnost je v REST API terminologii označována jako *Content Negotiation*[10].

Práce s daty

Toto kritérium hodnotí samotnou práci s daty. Tedy zda je k datům přistupováno strukturovaně, jsou-li data validovaná (řešení podporuje validaci data na úrovni vstupu) a jak složité datové struktury je schopno dané řešení pojmout.

Možnosti konfigurace

Zde se hodnotí možnost flexibilního používání. Zejména se pak řeší otázka, zda řešení vyžaduje určité jmenné konvence nebo je možné tyto konfigurovat (například zda je vyžadováno implementovat metodu read() a nebo je možné využívat i jinou metodu pro získání dat).

Celkové zhodnocení

Obecné hodnocení daného řešení. Shrnutí předchozích bodů s ohledem na další vlastnosti zvoleného řešení, které je dobré brát v úvahu. Také případné vyzdvihnutí pozitivních vlastností, které by bylo dobré mít ve svém finálním řešení.

3.2.2 Hodnocení řešení

3.2.2.1 Hodnocení Laravel

Dokumentace

Dokumentace frameworku Laravel je dostupná online a je velice pěkně zpracovaná. Je přehledně strukturovaná, s dostatkem názorných příkladů na použití popisovaných konceptů. Z hlediska obsáhlosti je na dostačující úrovni, avšak není zdaleka tak obsáhlá, jako u Symfony. Celkově je tedy mé hodnocení **2**.

Jednoduchost používání

Framework poskytuje nástroje a možnosti pro tvorbu API a určitě se snaží programátorovi pomoct s návrhem a používáním. Nastavení routování je jednoduché a navíc framework dokáže poskytnout instanci objektu pokud mu poskytneme data ve strukturované podobě. Hodnocení je tedy veskrze pozitivní: **2**.

Podpora různých formátů

Laravel v základu poskytuje podporu JSON formátu, který je v oblasti REST API nejpoužívanějším formátem (uvedení zdroje). Dále je možné rozšířit funkcionalitu frameworku o formát XML a stejně tak i další uvažované formáty dle libosti, díky konceptu middleware, který může kdokoli využít. Hodnocení je stejné jako v předchozím bodě, tedy **2**.

Práce s daty

Framework s daty pracuje poměrně uspokojivě – parsuje data do objektové reprezentace, i když tato není definovaná uživatelem (jedná se pouze o obecnou třídu). Následné používání vstupů je objektové, avšak pokud vyžaduji striktnější validace, nevyhnu se tomu, abych si nepsal vlastní přímo v obslužné metodě. Návratové hodnoty mohou být také objekty, což značně zjednoduší práci, protože uživatel potom není nucen ručně volat transformace. Hodnocení je tedy kladná **2**.

Možnosti konfigurace

Framework je konfigurován ve vyčleněném PHP souboru, kde se odkazuji na různé direktivy. Z hlediska přehlednosti bych jako vývojář ocenil mít samostatný konfigurační soubor, ve kterém by se veškerá konfigurace odehrávala. Není zde možnost určit si případné mapování parametrů (myšleno query string) nebo jejich validace. Hodnocení je za **3**.

Celkové zhodnocení

Framework jako takový se mi líbí, je minimalistický a výkonný. Poskytuje také většinu standardních nástrojů pro vývoj webu. V oblasti REST API jsou zde nedostatky především ohledně konfigurace, nicméně celkově se dá framework považovat za jeden z těch lepších pro vývoj REST API. Na frameworku se mi líbilo mapování vstupu přímo na entity, avšak za dodržení určitých pravidel a pouze pro jednoduché struktury. Celkově tedy hodnotím za **3**.

3.2.2.2 Hodnocení Magento 2

Dokumentace

Framework poskytuje obsáhlou dokumentaci online. Do dokumentace přispívá komunita. Ačkoli se jedná o poměrně obsáhlou a na příklady bohatou dokumentaci, mnohé funkcionality, které nejsou často využívané, uživatel dohledá až při samotném používání. Toto může činit nesnáze při používání a proto hodnotím za **3**.

Jednoduchost používání

Zde je největší slabina tohoto frameworku, jelikož jej mnozí hodnotí jako náročný na používání. Toto je pravda zejména v počátcích vývoje, jelikož samotný framework pracuje s poměrně odlišnými koncepty, než ostatní dostupné PHP frameworky a to i v oblasti REST API. Nicméně celkovou práci s frameworkem hodnotím neutrálně, tedy za **3**.

Podpora různých formátů

Zde je framework naprosto vyhovující. V základu poskytuje podporu pro formáty *JSON* a *XML*, nicméně je možné dodat podporu vlastního formátu, framework v tomto ohledu nepřináší žádné omezení. Celkově se mi podpora formátů u tohoto frameworku líbila a hodnotím velmi kladně, tedy **1**.

Práce s daty

V této oblasti je framework Magento 2 také velmi nápomocen, jelikož veškeré vstupní data se snaží namapovat na objekty. Jako vstupní objekt do REST API může uživatel použít jak model entity v databázi, tak vlastní model nezávislý na databázi. Z oblužné metody pro REST API je možné vrátit také objekt, který je dále převeden do reprezentace určené pro klienta, není tedy nutné vlastnoručně převádět data do asociativního pole. Oproti ostatním frameworkům není nutná ani žádná úprava vráceného objektu, jelikož data jsou z objektu extrahována pomocí getterů. Hodnocení je stejné jako v předchozím bodě, tedy **1**.

Možnosti konfigurace

Samotný framework je konfigurován pomocí *xml* souborů. Při konfiguraci REST API tomu není jinak. Je možné konfigurovat která konkrétní metoda bude oblushovat daný endpoint. Navíc je možné použít interface jako obslužnou třídu a *dependency injection* kontejner se postará o doplnění správné instance. Na rozdíl od ostatních frameworků je zde vidět jasné oddělení konfigurace od kódu. Hodnocení je kladné, tedy **1**.

Celkové zhodnocení

Tento framework poskytuje řadu možností jak zadefinovat REST API endpoint a jak jej používat. Jako vývojář oceňuji možnosti mapování vstupních a výstupních dat z/do objektů a to i složitějších struktur. Rovněž oceňuji to, že není nutné modfikovat objekty do/ze kterých má být mapování provedeno. Framework totiž používá všem dobře známé getry a setry a dané objekty tak neobsahují nic navíc. Celková známka, kterou tomuto frameworku uděluji je tedy **2**.

3.2.2.3 Hodnocení Nette

Dokumentace

Dokumentace Nette samotného je na tom o něco hůře, než dokumentace knihovny, kterou jsem zvolil pro tvorbu REST API. V porovnání s ostatními frameworky mi u Nette frameworku vadí nedostatečná dokumentace jednak kódu (chybějící *PHPDoc*) a také chybějící příklady pro běžné úkony. U použité knihovny pro REST API byla dokumentace na webu dostatečná, avšak chybějící *PHPDoc*, který by objasnil účel některých metod/tříd stále chybí. Celkově tedy musím udělit **4**.

Jednoduchost používání

Samotné rozběhnutí projektu a následné používání je u tohoto frameworku poměrně jednoduché. Používání REST API služeb (za použití knihovny *Apitte*) je poměrně jednoduché a uživatel se ve struktuře snadno orientuje. Zde uděluji hodnocení shodné s frameworkem Laravel - tedy **2**.

Podpora různých formátů

Pro Nette existuje spousta knihoven pro tvorbu REST API, kdy mnohé knihovny neposkytují jiné formáty než *JSON*. Mnou použitá knihovna *Apitte* však podporuje jak *JSON*, tak *XML* a *CSV*, ačkoli to vyžaduje jistá specifika. Je také možné doplnit vlastní formát poměrně jednoduše zaregistrováním dalšího rozšíření. Celkově tedy hodnotím známkou **1**.

Práce s daty

Rozšíření Apitte poskytuje nástroj, jak využít mapování do objektové reprezentace, avšak toto mapování musí specifikovat samostatně vývojář. Další nevýhodou v tomto ohledu je nutnost dědění z abstraktní entity pro každou vlastní entitu, která má být renderovatelná pro klienta. Toto omezení činí práci s entitami náročnější a pokud bychom chtěli použít základní možnosti práce s daty, pak se jedná o přístup k obsahu odpovědi přímo přes obecný getter, proto hodnotím známkou **2**.

Možnosti konfigurace

Samotná Apitte knihovna pro Nette se konfiguruje pomocí anotací a ty poskytují pestrou paletu možností. Pomocí anotací je možné nakonfigurovat mapování vstupu do entity, validace a případné další vlastnosti. Je možné používat také rozšířené funkce jako vlastní validátor pomocí *SymfonyValidator* avšak pro tento je zapotřebí instalace dalších balíčků. Konfigurace je za mě zcela dostačující a tedy hodnotím známkou **1**

Celkové zhodnocení

U Nette frameworku a rozšíření Apitte je velice pozitivní přístup k datům. Oproti jiným knihovnám a předchozím verzím nette je vidět posun v kvalitě provedení. Velmi kladně hodnotím možnost více formátů stejně jako možnosti konfigurace. Celkově framework hodnotím známkou **2**

3.2.2.4 Hodnocení Symfony

Dokumentace

Dokumentace samotného Symfony frameworku je obsáhlá, avšak postrádám zde rozsáhlejší ukázky kódu. Při orientaci v dokumentaci mne překvapila nemožnost zobrazit kompletní strukturu dokumentace. Navíc při orientaci v dokumentaci se menu mění, a tedy výsledný efekt je matoucí. Samotná obsáhlost a množství informací v dokumentaci obsažané jsou dostatečné, nicméně díky nekonzistentní orientaci v dokumentaci hodnotím negativně - tedy **4**.

Jednoduchost používání

Náročnost používání bych hodnotil jako náročnější a to zejména z důvodu vysoké modularity systému. Podobně jako u Magento 2 frameworku, je i zde problém s počátečním používáním frameworku, nicméně po čase je použití přehlednější. Z tohoto důvodu hodnotím náročnost použití za **3**.

Podpora různých formátů

Framework s rozšířením podporuje spoustu formátů, včetně standardního *JSON* a *XML* formátu. Stejně tak nechybí ani možnost doplnit formát vlastní, tedy zde je framework nanejvýše vstřícný a hdnocení tedy uděluji velmi kladné – *1*.

Práce s daty

Symfony společně s rozšířením *FOSRestBundle*[11] a rozšířením *SensioFrameworkExtraBundle*[12] poskytuje velice užitečnou možnost mapování do objektové reprezentace (entity). Je také možné dodat své vlastní mapování.

Podobným mechanismem je poté obejtková reprezentace zpracovávána do výstupu, který je opět možné vracet z obslužné metody. Zde hodnotí o něco lépe, než předchozí řešení, tedy **1**.

Možnosti konfigurace

Na tomto frameworku se mi líbí možnost zvolit si způsob konfigurace. Na výběr je *yaml*, *xml* a také *php*. U cest pro REST API a obslužných metod je možné využívat také anotací, které určují transformace na nejnižší úrovni. Z tohoto pohledu je framework naprosto dostačující. Hodnocení je tedy nejlepší, tedy *1*.

Celkové zhodnocení

Celkově se mi na frameworku líbilo směřování směrem k modularizaci a znovupoužitelnosti komponent, jako samostatné jednotky. Díky tomuto konceptu, je možné vypínat/zapínat jednotlivé balíčky dle potřeby a dynamicky si obohatit své REST API o další funkcionality. Ovšem díky složitějšímu použití a pro mě ne příliš dobře strukturované dokumentaci musím hodnotit celkově za **2**.

3.2.3 Závěr hodnocení

V tabulce 3.1 můžeme nalézt přehled hodnocení:

	Laravel	**Magento 2**	**Nette**	**Symfony**
Dokumentace	2	3	4	4
Jednoduchost používání	2	3	2	3
Podpora různých formátů	2	1	1	1
Práce s daty	2	1	2	1
Možnosti konfigurace	3	1	1	1
Celkové zhodnocení	3	2	2	2

Tabulka 3.1: Přehled hodnocení jednotlivých frameworků

Z přehledu hodnocení vidíme, že kromě frameworku *Laravel* je hodnocení napříč kategoriemi zhruba stejné, u zbývajících frameworků byla největší slabinou dokumentace a dále jednoduchost používání. Zejména u jednoduchosti používání je komplikací také to, že každý framework používá trochu odlišné koncepty a díky tomu vývojáři musí věnovat více času a úsilí pro samotné používání různých frameworků.

Další zajímavostí je, že ačkoli u ORM frameworků, kde je nejrozšířenějším frameworkem Docrtine 2, kterou lze použít napříč různými projekty, mnohdy dokonce se stejnými entitami (existuje jistá přenostielnost mezi projekty), pro

REST API podobný framework není k dispozici. Jednou z možných příčit může být uzší provázanost REST API na jádro frameworku a business logiku, ovšem v kombinaci s přenositelným ORM frameworkem vidím potenciál v samostatně použitelné knihovně.

3.3 Požadovaná funkcionalita

Cílem této podkapitoly je hrubě nastínit požadavky na funkcionality a finální použití, které se pokusím dosáhnout ve vlastním řešení.

3.3.1 Oddělení REST API a web kontrolerů

Pokud porovnám *Nette* framework s *Magento 2* frameworkem, tak si můžu všimnout zásadního rozdílu ve způsobu zpracování požadavků. Tím rozdílem je přístup ke zpracování požadavků. Zatím v Nette frameworku se setkáme s přístupem, kdy REST API kontroler je rozšířením klasického web kontroleru (v mnohých případech, netýká se ovšem Appite rozšíření), v Magento 2 frameworku je tomu přesně naopak. Tedy REST API služby využívají jiný mechanismus zpracování a jsou odděleny od klasických kontrolerů.

Ve vlastním řešení bude upřednostněn přístup s odděleným REST API a webovým kontrolerem. Tím bude do jisté míry zajištěna nezávislost na daném frameworku a celkově to přispěje k znovupoužitelnosti komponent vytvořených pro vlastní knihovnu.

3.3.2 Podpora více formátů

Všechny porovnávané frameworky nějakým způsobem nabízejí podporu více formátů, které může uživatel využít. Jedná se o velmi užitečnou funkcionalitu, kterou bych chtěl zachovat i ve finálním řešení. Nejlépe propracovanou stukturu vidím u frameworku *Symfony*, kdy je možné poměrně snadno doplnit další formát využitelný pro *REST API*. Podpora formátů by pak měla být zachována jak na vstupu, tak na výstupu.

Vlastnosti, kdy server vrátí klientovi odpověď v požadovaném formátu se říká *Content negotiation*[10] , konkrétně v této knihovně budeme využívat tzv. *Server-driven content negotiation*[10]. Rozlišení, jaký formát bude použit se bude dále řídit pomocí *HTTP hlaviček Accept* a *Content-Type*.

3.3.3 Mapování do objektové reprezentace

Velice užitečným nástrojem při práci s REST API je jistě mapování vstupu do objektové reprezentace. To samé platí pro výstup. Aby byla práce s vstupem nebo výstupem efektivní. Obslužná metoda by měla na vstupu dostat mapovaný vstup, a na výstupu vrátit objekt, který je určen pro převedení na

výstup. Dalším požadavkem je možnost zpracovávat složité datové struktury a tedy nutnost zajistit mapování nejen skalárních hodnot, ale také složitějších objektových struktur.

Mapování v tomto případě bude prováděno pomocí getterů a setterů, jelikož to dále umožňuje vývojáři rozšíření funkcionality, například pokročilejší transformace přímo v obejktu na základě hodnot. Tento přístup používá framework *Magento 2* a hodnotím jej velmi kladně, proto byl zvolen pro výsledné řešení.

Díky tomuto získá vývojář přesnější kontrolu nad daty a výsledná implementace v obslužné metodě se může zabývat čistě logikou specifickou pro aplikaci.

3.3.4 Validace vstupu

Validace dat jsou velmi důležité snad v každém API rozhraní (nejenom REST). Cílem validace je zajistit, aby daná příchozí data byla platná a nevznikaly problémy například s datovou konzistencí. Vstup bude ve výsledném řešení typově validován ještě před mapováním do obejktové reprezentace. Další pokročilejší validace (například platnost relací na jiné entity), bude možné zajistit díky mapování pomocí setterů, jak bylo zmíněno v podkapitole 3.3.3.

3.3.5 Autentizace uživatele

Pokud chceme poskytovat rozsáhlejší REST API, které umožňuje také úpravu entit, avšak nechceme aby tato úprava byla proveditelná kýmkoli, je nutné zavést jistý způsob autentizace. K tomuto účelu bude knihovna samotná psána jako mezi vrstva – tzv. middleware – aby bylo možné zařadit zpracování knihovny až po definované autentizační mechanismy. Tento mechanismus bude velice podobný tomu, který používá knihovna *Apitte* pro *Nette*. Samotná autentizace nemůže být součástí výsledné knihovny, jelikož by to přineslo přílišné provázání s cílovým řešením.

3.4 Dotazníkové šetření

Za účelem ověření důležitosti různých vlastností výsledné knihovny, jsem vypracoval dotazník zaměřující se na zkušenosti a očekávání vývojářů při tvorbě *REST API* služeb. Sběr dat probíhal v období od 10.3.2020 do 15.4.2020. Během této doby na dotazník odpovědělo celkem 35 responsentů.

Dotazníkové šetření bylo zvoleno aby bylo možné porovnat vyspecifikované požedavky s reálnou potřebou uživatelů, jelikož v této práci vycházím převážně z potřeb, které vnímám na různých pozicích v posledních letech. Cílem dotazníku je tedy získání širšího pohledu na věc a výsledky dotazníku poslouží

jako ukazatel, na kterou z vydefinovaných funkcionalit se ve výsledném řešení zaměřit.

Dotazník byl rozeslán na počátku března 2020 třemi hlavními informačními kanály, a to sice:

- Pomocí firemní platformy Slack (ve společnosti, kde jsem v době rozeslání dotazníku působil)
- Přes Facebook skupiny, mezi spolužáky na FIT ČVUT
- Pomocí emailových kontaktů na starší spolupracovníky a spolužáky ze SPŠEI Ostrava

3.4.1 Členění dotazníku

Dotazník je rozčleněn do čtyř sekcí a to podle druhu informací, který se v dané sekci snažím získat. Jedná se o následující sekce s uvedeným významem:

Úvod

Cílí především na základní údaje o profilu programátora, tedy s jakými nástroji zatím pracoval, zda se aktivně věnuje vývoji v PHP a další otázky směřované na jeho aktuální stav. Na zákadě odpovědí zde uvedených se dále určí, zda je nutné aby zodpovídal otázky v sekci druhé, nebo přejde rovnou do sekce třetí.

Tvorba REST API služeb v jazyce PHP

Snahou této sekce je získat informace o tom, jak je konkrétní vývojář spokojen s postupem tvorby REST API služeb v jazyce PHP, jaký framework považuje za nejlepší a zda byly dané nástroje vybírány s ohledem na budoucí tvorbu REST API služeb.

Očekávání od nástrojů

Jaká jsou hlavní očekávání od nástrojů pro tvorbu REST API, bez ohledu na framework, tedy co vývojáři považují za důležité, aby bylo ve výsledné implementaci zakomponováno. Z pohledu této práce je tato sekce dotazníku klíčová a bude jí proto věnována největší pozornost.

Závěr

Obsahuje základní demografické údaje o vývojáři a jeho zařazení v pracovním procesu.

3.4.2 Výsledky dotazníku

Z celkového dotazníku bych nyní zdůraznil především sekci číslo 3, jelikož v ní se pojednává o očekávání, která má vývojář od zvoleného řešení a jakou hrají roli. V tabulce 3.2 je vidět přehled možností a jakou váhu jim přikládají responsenti:

Vlastnost	**Hodnocení**
Možnost definovat validace nad daty	4,14
Rozsáhlost dokumentace	3,86
Možnost si flexibilně zadefinovat obslužné metody	3,57
Podpora striktního typování	3,51
Vyšší míra abstrakce - návrh na úrovni interface	3,43
Snadná instalace řešení	3,43
Podpora různých datových formátů	3,34
Co vše si můžu konfigurovat bez zásahu do kódu	2,94

Tabulka 3.2: Průměrné zhodnocení požadovaných vlastností dle responsendů

Dále v této sekci byla možnost přidat vlastní požadované funkcionality. Jeden respondent zdůraznil potřebu podpory ORM entit, jeden oceňuje možnost generování serveru i klienta z definice REST API (například ze specifikace OpenAPI). Dále pak jeden resnpodent vyjádřil očekávání podpory nástrojů pro integrační testování.

Komplentí výsledky dotazníku je možné nalézt na přiloženém médiu jako přílohu.

Kapitola 4

Návrh

Tato kapitola se zabývá podrobným návrhem knihovny, včetně věcí s tím souvijecích, jako například použití verzovacích nástrojů, zapracovávání dalších verzí a celkově procesů týkajích se výsledné knihovny.

4.1 Platforma

Výsledná knihovna bude psána pro jazyk PHP ve verzi 7.2 kompatibilní. Vzhledem k povaze knihovny není nutné vybírat databázi.

4.2 Verzování

Jelikož plánuji knihovnu dále upravovat, případně zveřejnit, je vhodné zvolit vyhovující verzovací strategii již na začátku projektu. Za tímto účelem byl zvolen nástroj Git, který poskytuje velice efektivní správu verzí a umožnuje zapojení více autorů.

[13]V současné době existuje více strategií, jak verzovat kód, já se pokusím nejčastější z nich popsat a vybrat takové, které bude tomuto projektu nejvíce vyhovovat.

4.2.1 Centralized workflow

Jedná se o způsob práce s nástrojem GIT, kdy je na projektu udržován jeden centrální bod, ke kterému se vztahují všechny změny prováděné uživateli. Výhodou tohoto workflow je jeho jednoduchost, kdy je nutné udržovat pouze jednu hlavní větev – `master`.

Nevýhodou tohoto workflow je naopak jeho nízká flexibilata, obtížná správa příspěvků mimo tým/původní vývojáře a také nemožnost udržení více funkčních

verzí zároveň. Z tohoto pohledu se toto workflow jeví z dlouhodobého hledika jako naprosto nevhodné, avšak v počáteční fázi vývoje, zeména jedná-li se projekt udržován jedním vývojářem, jako je tato práce, svou jednoduchostí zcela dostačuje.

4.2.2 Feature branching workflow

Jedná se v podstatě o rozšíření centralizovaného workflow. Hlavním rozdílem je to, že veškerý nově vyvýjený kód je udržován v samostatné větvi, tzv. *feature branch*. Jakmile je nová funkcionalita vyvinuta, je nejprve otestována a teprve poté zahrnuta do hlavní větve, tzv. *master branch*.

Výhodou je tedy čistá větev `master`, která by neměla obsahovat chybný kód a měla by být vždy provozu schopná. Další výhodou je eliminace konfliktů při vývoji v týmu.

Z pohledu této závěrečné práce je však i toto workflow z dlouhodbého hlediska nevýhodné, a to sice z důvodu obtížné podpory více funkčních verzí zároveň. Nevyhovuje ani v počáteční fázi vývoje, jelikož se nejedná o skupinovou práci.

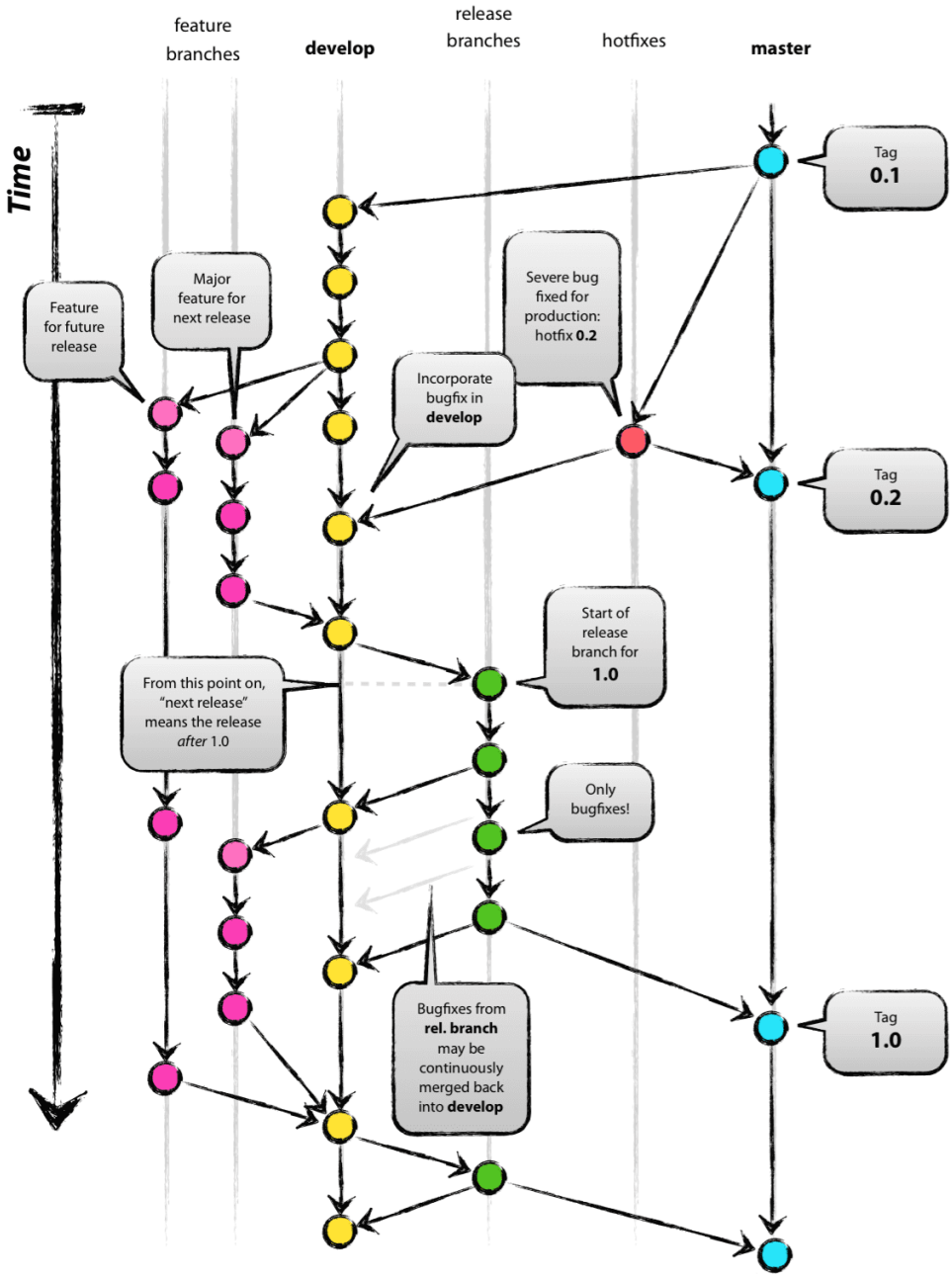
4.2.3 Gitflow workflow

Jedná se o poměrně propracované workflow poskytující podporu pro rychlé opravy v kódu u již zveřejněných verzí, tzv. *hotfix*. Lze na něj pohlížet jako na rozšíření *Feature branching workflow*, avšak větev, do které vývojář zapracovává své změny není `master`, ale jedná se o `develop`, který nemusí obsahovat vždy spustitelný kód, ale do kterého se integrují všechny vyvinuté funkcionality před zveřejněním nové verze.

Toto workflow se dá dále jednoduše rozšířit o možnost podpory více verzí současně a to využitím `release` verzí ne jen jako integračních a testovacích větví před začleněním do `master` vetve, ale využíváním těchto větví i po začlenení do `master` větve jako zázemí pro případné opravy ve chvíli, kdy je již vydána novější verze.

Pro tento projekt se jedná o ideální workflow ve chvíli, kdy bude dokončena první verze a knihovna bude zpřístupněna pro veřejnost a další přispěvatele. Díky `feature` větvím je možné kontrolovat, které funkcionality budou začleněny do nové verze, stejně tak kontrolovat případné bugfixy.

Kompletní schéma tohoto workflow lze nalézt na obrázku 4.1:



Obrázek 4.1: Schéma Gitflow

4.2.4 Forking workflow

Jedná se o zcela odlišný přístup k páci s nástrojem GIT než v předchozích případech. Není zde totiž udržován řídný centrální bod, kde by docházelo

k synchronizaci kódu s ostatními vývojáři. Každý vývojář zapojený do projektu pracuje jednak na své lokální kopii projektu, ale také vystavuje veřejnou podobu repozitáře, který mohou ostatní vývojáři využít.

Toto workflow je ovšem proti záměru této práce a to poskytnout vývojářům knihvnu. Ta totiž musí být někde přístupná a z pohledu této práce by nedávalo smysl aby byla distribuována takto decentralizovaně. Proto nebude tohoto workflow nijak využito.

4.2.5 Zvolené řešení

Jelikož z počátku vývoje (období tvorby této práce) budu na projektu pracovat pouze já a jelikož se jedná o nový projekt, nemá smysl udržovat složité workflow a tedy bude využito *Centralized workflow*. V okamžiku případného zeveřejnění zdrojových kódů pro veřejnost a umožnění ostatním vývojářům rozšíření knihovny o další funkcionality, bude toto jednoduché workflow nahrazeno sofistikovaným *Gitflow* tak, jak bylo popsáno v kapitole 4.2.3.

4.3 Composer

Jedná se o nástroj pro správu závislostí. V komunitě PHP vývojářů je hojně využíván a umí pracovat také s nástrojem GIT, který byl popsán v kapitole 4.2. Díky *Composeru*, je možné efektivně spravovat závislosti na externích knihovnách, řídit jejich verze, případně publikovat vlastní knihovnu pro širší komunitu. Jelikož je hojně využíván, budu jej používat i ve výsledné knihovně právě pro správu verzí knihovny.

4.4 OOP Design a principy

Vytvořená knihovna bude psána objektově orientovaným přístupem. To s sebou přináší také zodpovědnost za dodržování správného návrhu architektury. V této podkapitole se budu věnovat důležitým OOP principům, které je třeba dodržet za účelem dosažení pokud možno všech vyspecifikovaných požadavků. Základní principy při návrhu v OOP jsou tzv. *SOLID*[14] principy, které stručně popíši v této kapitole.

4.4.1 Princip jedné odpovědnosti

Jedná se o velice jednoduchý princip, který říká, že každá třída by měla mít pouze jednu zodpovědnost (jeden důvod ke změně). Použití tohoto principu vede na mnoho menších tříd, které pak lze samostatně udržovat a to vede k větší udržitelnosti systému. Rovněž přispívá k možné modularitě systému, jelikož nabízí více možností k rozdělení návrhu na jednotlivé oblasti.

4.4.2 Princip otevřenosti a uzavřenosti

Tento princip nám říká, že třídy by měly být navrženy tak, aby bylo možné je rozšířit ideálně bez zásahu do existujícího kódu. Díky tomuto se sníží riziko, že při přepisování jedné části aplikace poškodíme části jiné. Klíčem pro správnou aplikaci tohoto principu je hojné využití abstrakce a polymorfismu.

4.4.3 Liskovové princip zaměnitelnosti

Velice jednoduchý, ale velice jednoduše porušitelný princip, který nám říká, že pokud máme nějakou bázovou třídu, kterou nahradím jejím potomkem, pak bude implementace fungovat korektně i s tímto potomkem. Díky tomuto principu je jasné, že každá podtřída musí být schopna poskytnout stejné vlastnosti jako třída bázová. Právě díky této vlastnosti je možné efektivně a jednoduše rozšiřovat výsledný systém/knihovnu.

4.4.4 Princip oddělení rozhraní

V zásadě říká, že více různých rozhraní je lepších, než jedno velké univerzální. Pokud máme jedno společné rozhraní, ovšem jeho funkcionality spolu příliš nesouvisí, pak se rozrůstá okruh tříd, které na daném rozhraní závisí a v případě úpravy je poté náročné kontrolovat změny, které by mohly ovlivnit všechny třídy, které by mohly rozhraní používat. Právě díky tomu je lepší rozhraní rozdělovat. Je však třeba dávat pozor na vhodnou granularitu, jelikož by striktní dodržování tohoto principu mohlo vyústit v příliš mnoho drobných rozhraní.

4.4.5 Princip obrácení závislostí

Jedná se o princip vyžadující aby závislosti byly vždy na abstraktním a ne na konkrétním. Tedy podle totoho principu je nutné, aby konkrétnější implementace závisela na abstraktnější, ne naopak. Díky tomuto principu v kombinaci s *Liskovové principu zaměnitelnosti* je možné jednoduše měnit implementace dané funkcionality bez většího zásahu do kódu.

4.5 PHP-FIG PSR

PHP-FIG[15] nabízí PHP standardy – tzv. PSR – které, pokud chceme mít knihovnu kterou bude využívat více lidí, je dobré dodržovat. V této kapitole se budu věnovat důležitým standardům, které využiji při implementaci a budu je aktivně využívat.

Díky těmto standardům a jejich dodržování je možné jednotlivé komponenty přenášet mezi frameworky. A protože cílem této práce je knihovna

přenositelná mezi frameworky, pak je budu dodržovat. V následujících kapitolách popíši standardy, které budu nejčastěji používat.

4.5.1 PSR-1

Jedná se o základní standard, který je troufám si říct tím základním, co by měl vývojář dodržovat. Jedná se o popis vzhledu PHP souboru, který by byo dobré dodržovat kvůli čitelnosti. V mnoha firmách je tento standard vyžadován a je podporován také v mnohých IDE. Nedodržování tohoto standardu vede k nečitelnému, nebo těžce čitelnému kódu, který je obtížné udržovat.

4.5.2 PSR-4

Díky tomuto standardu je možné využívat knihovnu i mimo definované prostředí. Standard popisuje jak má vypadat adresářová struktura, aby bylo možné třídy a rozhraní načítat dynamicky. Tento standard je užíván ve spojistosti s nástrojem *Composer*, který byl popsán v kapitole 4.3.

4.5.3 PSR-7

Tento standard je pro tuto práci snad nejdůležitějším. Popisuje totiž práci s HTTP požadavky, jejich obsahem a obecně popisuje, jak má dané rozšíření zpracovávat požadavky a jaké nástroje má k dispozici. Právě díky tomuto standardu bude možné využívat výslednou knihovnu v různých frameworcích (které poskytují implementaci *PSR-7* standardu).

4.5.4 PSR-11

Standard poskytující rozhraní pro kontejner závislostí. Díky tomuto standardu nemusím v knihovně specifikovat a popisovat vkládání závislostí a vyhledávání služeb, ale můžu si pohodlně vyžádat kontejner, který mi dané služby poskytne i se závislostmi.

4.6 Implementační návrh jádra knihovny

Díky principům popsaných v kapitole 4.4 a díky znalosti požadavků z kapitoly 3.3 můžeme nyní navrhnout knihovnu tak, aby bylo možné knihovnu dále používat v různých frameworcích.

Během návrhu je nutné brát v úvahu případnou možnost implementační změny daných služeb. Díky tomu by se měl návrh pohybovat v maximální možné míře na úrovni rozhraní. Konkrétní implementace bude dodána posléze kontejnerem cílové aplikaci.

Jednotlivé dílčí implementace knihovny pro použití v různých frameworcích se díky tomu může lišit a dodávat své vlastní specifické úpravy pro danou službu.

4.6.1 Jádro knihovny

Aby bylo možné používat knihovnu efektivně v různých prostředích, byla zvolena jak implementace jako koncové rozhraní, tak jako middleware. To znamená, že programátor používající tuto knihovnu se může rozhodnout, jak ji začlení do svého kódu.

Samotné jádro je nutné rozdělit na dílčí služby, které zajišťují podporu pro různé funkční celky. Toto členění bylo zvoleno zejména kvůli udržitelnosti vývoje do budoucna. Služby, na které se jádro dělí jsou:

RestCore

Tato služba by měla být vstupním bodem do knihovny. Poskytuje dvě mětody, kdy jedna slouží jako middleware - tedy přijímá požadavek a odpověď, potencionálně předszpracovanou jinou službou, a vrací odpověď obohacenou o vlastní data. Požadavek i odpověď jsou vyžadovány dle doporučení PSR-7, tedy jakýkoli framework dodržující nebo poskytující PSR-7 implementaci HTTP požadavku a odpovědi bude moci využít tuto knihovnu.

Router

Tato služba slouží k vyhledání vhodného koncového bodu (rzv. endpoint) pro příchozí požadavek a pokud nalzne vhodný endpoint, pak vrátí jeho meta informace. O tom, jaká je použitelná HTTP metoda a jaký má být vzor pro URL (regex URL patter) rozhoduje samodný endpoint.

Vzhledem k odlišnosti přístupu ke konfiguračním souborům v různých frameworcích a různým možnostem parsování jak souborů, tak kódu, je navrženo řešení kdy knihovna vyžaduje objekt, který v sobě drží informace o dostupných cestách. Tento objekt ve výsledné knihovně bude vycházet z rozhraní *HolderInterface*, které poskytuje jednoduchou metodu pro manipulaci s endpointy. Pro pohodlnou manipulaci s tímto objektem je navrženo, aby implementoval rozhraní *Iterator*.

Samotný způsob načítání dostupných cest, ať už z konfigurací, nebo z kódu (anotací), je ponechán na dílčích podpůrných implementacích pro konkrétní řešení.

Dispatcher

Tato služba by měla být volána ve chvíli, kdy je známé, který endpoint bude obsluhován. Úkolem *Dispatcher* služby je zajistit správné zjištění vstupních parametrů pro požadovanou cestu, správné namapování zjištěných parametrů a v neposlední řadě také zpracování výstupu z obluhované metody.

Dispatcher spoléhá na přítomnost požadované metody v PSR-11 kontejneru, tedy kontejneru, který má zajistit správné poskytnutí služeb dle daného typu. Díky použití PSR-11 kontejneru je možné vyžadovat i složitější objekty, které případně vyžadují další závislosti.

Ve chvíli, kdy z nějakého důvodu není možné obsloužit požadavek, což může být způsobeno například chybějícím parametrem, nebo poskytnutí parametru jiného než očekávaného typu, je vytvořena výjimka a tato je dále zachycena a zpracována službou *ExceptionMapper*.

ExceptionMapper

Tato služba má za úkol mapovat výjimku na vstupu do odpovídající struktury na výstupu. Za tímto účelem je vytvořen pomocný objekt, viz kapitola 4.6.6. Služba je zde zřízena především za účelem budoucí rozšiřitelnosti dle přání programátora, který bude knihovnu využívat, protože poskytuje možnost mapovat různé typy výjimek do různých sturktur.

Využití může být například, pokud v aplikaci používám výjimky *NoSuchEntityException* a *DuplicateEntryException*, můžu každou z nich namapovat do jiné stuktury, kterou posledéze knihovna vyrenderuje do výsledného formátu tak, aby bylo možné ji zobrazit klientovi.

4.6.2 Formáty

Jak bylo deklarováno v kapitole 3.3.2, ne nutné zajistit podporu více formátů pro danou knihovnu a ne se jen spoléhat na předem definované vstupní/výstupní formáty. Za tímto účelem je navrženo rozhraní *FormatInterface*, které obsahuje metodu, která definuje metodu, díky které je možné zjistit, zda daný formát (na základě mime-type) je možné využít.

Zároveň poskytuje gettery pro tzv. *Reader* a *Writer*, tedy objekty, které se dále starají o konverzi dat ze vstupu na interní reprezentaci a obráceně. Formát bude možné nadefinovat pouze jako vstupní, případně výstupní na základě dostupných *Reader*ů a *Writer*ů.

Způsob udržování seznamu dostupných formátů bude realizován podobně, jako seznam dostupných endpointů a to s ohledem na vyšší míru konfigurovatelnosti. Díky tomuto způsobu je možné provést napojení na výsledné řešení.

4.6.3 Reflexe

Jelikož v knihovně budeme často pracovat s reflexí[16], a to zejména při mapování obejktů a jejich následné dekompozici zpět do asociativního pole, je zde navrženo využít zastřešující služby nazvané *ReflectionHolder*. Tato služba bude mít na starosti správné zjišťování vstupních parametrů pro settery a výstupních parametrů pro gettery.

Pro práci s reflexí je vhodné použít nějakou z již existujících knihovnen. Pro realizaci této knihovny bylo zvoleno použití knihovny *laminas-code*[17] (dříve známá také jako *zend-code*). Knihovna bylo zvolena pro svou jednoduchost použití.

4.6.4 Mapování vstupních dat

Úkolem knihovny je mimo jiné zajištění mapování vstupních dat pro oblužnou metodu, kdy pro zajištění správného mapování a rozšiřitelnosti je využito podobného principu jako u formátů – tedy že existuje jedna služba držící všechny dostupné mapovací strategie. Mapovací strategie budou realizovány pomocí tzv.: *Mapperů* a je pro to vyhrazeno rozhraní *MapperInterface*.

V základním stavu poskytuje knihovna mapování pro všechny primitivní datové typy, objektové mapování a také mapování pro objektově relační framework Doctrine 2, resp. jeho entity.

Mapování do objektů je prováděno pomocí setterů, což pro účely knihovny jsou všechny metody začínající klíčovou předponou `set`. Seznam těchto metod, včetně jejich typů bude získán za pomocí reflexe z *ReflectionHolder* kontejneru. Struktura mapování by měla být taková, aby podporovala i vnořené datové struktury.

Díky mapování pomocí setterů bude možné zavést složitější validace spojené s business logikou přímo v dané entitě. Díky provázání různých validačních frameworků a technik na dané cílové řešení, bylo zvoleno základní typové validování v knihovně, ale pokročilejší validační pravidla související s business logikou aplikace nechť jsou prováděny v příslušných setterech.

U jednotlivých mapperů musí být možné určit, kterého typu vstupu se mapper týká. Oblasti, kterých se může mapper týkat jsou:

- URI požadavku
- Query string požadavku
- Samotné tělo požadavku

4.6.5 Mapování výstupních dat

Podobně jako u vstupu, kdy je možno mapovat vstupní data do objektové reprezentace, knihovna by měla poskytovat také způsob jak mapovat objekty na výstupu do formátu požadovaného klientem.

Služba, která toto bude zajišťovat se bude jmenovat *ServiceOutputProcessor* a bude provádět mapování objektů do asociativního pole na základě getterů, tedy metody začínající na klíčovou předponu `get` a poskytující hodnoty pro daný atribut. Zde je opět hojně využit *ReflectionHolder* k získávání příslušných informací.

Mapování výstupních dat, stejně jako mapování vstupních dat, je navrženo tak, aby bylo možné zpracovávat i složitější datové struktury.

4.6.6 Zpracování požadavku

Zpracování požadavku provede, jak bylo popsáno v kapitole 4.6.1, služba *Dispatcher*. Tato služba jistí z příchozího požadavku v jakých formátech komunikovat s klientem a v jakém formátu komunikuje klient samotný (je tedy možné použít různé formáty pro vstup a pro výstup).

V následujícím kroku by si měl *Dispatcher* zjistit dostupné a požadované hodnoty na vstupu. Pro zjištění požadovaných hodnot servisní metody je nutné použít reflexi[16], kterou budeme často využívat a proto je pro ni zřízena speciální služba, tzv. *ReflectionHolder*. Více o reflexi v kapitole 4.6.3

Po zjištění dostupného formátu je provedena konverze vstupních hodnot do požadovaných parametrů servisní metody. Díky tomuto se uživatel používající tuto knihovnu nemusí starat o konverze, ani to, kde vzít dané hodnoty. Jendoduše uvede požadovaný typ a ten následně dostane.

Následně je oblužná metoda požadavku zavolána a její výstup je poté mapován jako odpověď klientovi. Odpověď může být různých typů, přičemž existují různá pravidla pro mapování různých typů.

MinimalisticResponse

Pokud je vrácena tato odpověď, pak je jako výstup zpracován ne celý objekt, ale jeho data, ktrá vložil před vytvořením tohoto objektu uživatel. Výhodou použití tohoto objektu pro výstup je možnost definovat si návratový kód, který bude navrácen klientovi.

ResponseInterface

Pokud je z oblužné metody navrácen objekt implementující *ResponseInterface*, pak je z *Dispatcheru* vrácen přímo tento objekt, jelikož je zde předpoklad, že uživatel ví co dělá a odpověď náležitě zpracoval již dříve.

Ostatní typy

Jakýkoli jiný typ je mapován pomocí tzv. *ServiceOutputProcessor*u, který se stará o převedení případné objektové reprezentace do asociativního pole, které je dále konvertováno do příslušného formátu, jak bylo popsáno v kapitole 4.6.5 a 4.6.2.

4.6.7 Příklady budoucího užití

Tato kapitola je určená pro bližší ilustraci výsledného použití knihovny, společně s bližším popisem tak, aby čtenář získal představu o výsledném produktu.

Základní užití knihovny

Základní užití knihovny by mělo být velice jednoduché. Stačí správně uvádět typy, které očekávám na vstupu a které poskytuji na výstupu dané oblužné metody. Následně stačí zaregistrovat URI, které povede k dané oblužné metodě a to je vše. Pro koncového programátora by to mělo znamenat minimum konfigurace.

```php
<?php

namespace App\Api\Controller;

class SomeService {
    /**
     * @param string $string
     *
     * @return string
     */
    public function get(string $string): string {
        return $string;
    }
}
```

Mapování s ORM objekty

Následující příklad ilustruje jednoduchost použití ve spojení s ORM frameworkem Doctrine 2. V příkladu lze vidět, že vyžadovaný typ na vstupu je `Library`.

Pokud je `Library` entita spravovaná pomocí Doctrine 2 a já mám definovanou cestu ke zdroji jako `api/v1/library/:lib/booksAvailable`, kde místo parametru `:lib` dosadím konkrétní ID knihovny, pak se mapování postará o načtení dané entity a já ji mám jako programátor k dispozici v oblužné metodě.

V případě, že daná entita neexistuje, je vyhozena výjimka, kterou můžu pohodlně namapovat díky *ExceptionMapper* službě, která je popsána v kapitole 4.6.1.

```php
<?php

namespace App\Api\Controller;

use App\Entity\Library;
use App\Entity\Book;

class SomeService {
    /**
     * @param Library $lib
     *
     * @return Book[]
     */
    public function get(Library $lib): array {
        return $lib->getBooksAvailable();
    }
}
```

Jak si také můžete všimnout, návratový typ uvedený v PHP je `array`, ale reálně se jedná o pole objektů typu `Book`. Knihovna si tuto informaci správně zpracuje z anotace a provede příslušné mapování odpovídající danému typu.

Kapitola 5

Realizace

V této kapitole se budu věnovat výstupům praktické části této práce a postupům, které vedly k jejich dokončení.

5.1 Postup realizace

V této kapitole pojednávám o postupu relalizace dané knihovny v kontextu celé práce. Pro správnou implementaci bylo zapotřebí nejprve přenést návrh do definovaných rozhraní v jazyce PHP. Rovněž bylo zapotřebí vytvoření balíčku pro danou knihovnu.

Úvodní fáze

Po zapracování rozhraní v jazyce PHP začala implementace hlavních komponent, které se starají o životní cyklus daného požadavku. Pro snadnější a rychlejší testování bylo již v prvotní fázi implementace rozhodnuto o paralelní implementaci referenčního použití ve frameworku Nette (více o referenční implementaci v kapitole 7).

Paralelní implementace umožňila vyzkoušet si aktuální rozpracovanou knihovnu, ale vyžadovalo to také striktní dodržování rozhraní, které knihovna obsahuje a měla by obsahovat z důvodu rozšiřitelnosti a přenositelnosti.

Během úvodní fáze bylo rozhodnuto o implementaci tří formátů, které může uživatel knihovny použít tak jak jsou. Jedná se o formát *JSON*, *XML* a *CSV*.

Hlavní fáze vývoje

Během hlavní fáze vývoje byly vyvinuty veškeré komponenty, do funkčního stavu a rovněž jsem začal psát testy pro klíčové komponenty knihovny (více o

testování v kapitole 6). Během vývoje knihovny jsem se nepotýkal se závažnějšími problémy a knihovna vznikla bez nutnosti výrazně modifikovat návrh.

Během této fáze však došlo k situaci, kdy práce s reflexí byla často se opakující a bylo tedy nutné ji více uhladit. Kvůli této skutečnosti jsem se rozhodl k další fázi.

Refaktoring

Po dokončení intenzivního vývoje bylo nutné uhladit práci tak, aby zbytečně neobsahovala opakujícíc se kód a byla co nejvíce minimální (a díky tomu udržitelná do budoucna). Kvůli této skutečnosti následoval refaktoring kódu, tedy přeskupení určitých částí kódu do logických celků.

Tuto fázi považuji za velmi důležitou, ač se nejedná o refaktoring v pravém slova smyslu. Je důležitá právě díky skutečnosti, že dělá kód přehlednější a mnohem více udržitelný.

Oprava chyb

Na závěr byly doplněny vzniklé testovací scénáře o ještě více případů a byly odhaleny chyby v různých částech knihovny. Zejména se jednalo o chyby v renderování a konverze obsahu z a do formátu *XML*. V závěrečné fázi byly tyto chyby zapracovány a bylo docíleno funkčního stavu knihovny.

5.2 Instalační příručka

Knihovna je psaná pro verzi PHP 7.2 a vyšší. Instalace knihovny vyžaduje použití nástroje *Composer*[18]. Jelikož se v době zveřejnění této práce nejedná o veřejně přístupnou knihovnu, je nutné si stáhnout přílohu této práce a postupovat dle návodu v dokumentaci nástroje *Composer* – konkrétně se jedná o instalaci pomocí artefaktu[19].

Pro různé frameworky je poté nutné instalovat a nebo vytvořit mezivrstvu mezi cílovou aplikací a knihovnou. Více informací naleznete v kapitole 7 o referenční implementaci.

5.3 Programátorská dokumentace

Dokumentaci pro vývojáře, včetně ukázkového použití lze nalézt v příloze ve složce `/lib/core/doc` ve formátu Markdown[20]. V přiložené dokumentaci lze nalézt detailnější ukázky použití, včetně detailnějšího popisu vnitřní architektury. Pokud by to bylo nutné, je možné vygenerovat také dokumentaci z PHPDoc[21] anotací.

Aby bylo možné generovat dokumentaci pomocí nástroje *phpDocumentor*[22], bylo nutné dodržovat zásadu, že veškerý kód je zdokumentován. To také umožňujě dalším vývojářům, kteří by se chtěli do projektu zapojit ve snadnější orientaci v kódu.

5.4 Zhodnocení výsledné knihovny

Výsledkem realizace je praktická, minimalistická knihovna, který je velice snadno přenositelná mezi frameworky a splňuje má očekávání. Je možné ji využít pro mapování složitých struktur, stejně jak jako i jednodušších aplikací.

Knihovna jako taková poskytuje také možnost navrhnout API na základě interfacu – tedy je možné využít vyšší míru abstrakce jak u entit, tak u servisních metod, což je velice užitečná vlastnost právě z důvodu přenositelnosti a udržitelnosti.

Kapitola 6

Testování

Nedílnou součástí vývoje bylo a je také testování. Testování se zaměřovalo na klíčové části aplikace, především práci s formáty a mapováním.

6.1 Jednotkové testování

Pro jednotkového testování bylo využito testovacího frameworku PHPUnit[23]. Tento framework poskytuje stabilní zázemí pro tvorbu jednotkových testů a je velmi jednoduchý na použití. jednotkové test je možné nalézt na přiloženém médiu ve složce `/lib/core/test`.

Spuštění testů vyžaduje instalaci knihovny jako takové, nikoli jako součást většího řešení. Instalace knihovny s *PHPUnit* se provádí opět pomocí nástroje *Composer*.

Kapitola 7

Referenční implementace

Za účelem demonstrace využití knihovny a také za účelem názorné ukázky práce s knihovnou, jsem se rozhodl vytvořit referenční implementaci mezivrstvy mezi knihovnou a cílovým frameworkem.

K tomuto účelu bylo nutné určit vhodnou platformu pro takovou implementaci a následně vývoj mezivrstvy probíhal společně s hlavní knihovnou.

7.1 Výběr vhodné platformy

Pro implementaci mezivrstvy jsem vybíral z frameworků, které jsem analyzoval v prvotní části této práce. Nakonec byl vybrán Nette framework jako vhodná platforma pro demonstraci možností knihovny.

Důvodů pro Nette framework bylo více, především však dlouho trvající potřeba mít kvalitní nástroj pro mapování objektů v rámci REST API do aplikace. Jelikož Nette framework provozuji na větším množství projektů a již vícekrát jsem byl požádán o tvorbu REST API v Nette, tento framework se stal přirozenou volbou.

7.2 Realizace

Realizace mezivrstvy poskytuje možnost využít konfiguraci jak formátu Neon[24], což je základní formát se kterým Nette framework pracuje. Kromě možnosti nastavovat endpointy pro REST API pomocí definovaných anotací.

Závěr

Výsledná knihovna je velikým přínosem pro mé stávající i budoucí projekty. Rovněž současně se zveřejněním této práce očekávám zveřejnění rovněž zdrojových kódů na platformě GitHub, kde bude možné tuto knihovnu využívat.

Výsledná knihovna umožňuje mnohem vyšší modularitu co se týče REST API než základní možnosti jednotlivých frameworků a díky tomu je možné snadněji přecházet mezi frameworky. To přispěje k vyšší udržitelnosti projektů do budoucna.

Ve vývoji knihovny hodlám pokračovat i nadále, jelikož je zde stále spousta prostoru pro zlepšení, například provedení testů výkonosti a následných optimalizací, zejména v oblasti mapování.

Dle mého názoru byly splněny všechny části zadání, avšak ukázková aplikace mohla být většího rozsahu, což by umožnilo provést také testování pomocí nástroje Postman na rozsáhlejší aplikaci. Toto rozšíření ukázky plánuji společně s vývojem mezivrstvy pro Symfony framework.

Bibliografie

1. Google Trends - REST API [online] [cit. 2020-05-19]. Dostupné z: https://trends.google.com/trends/explore?cat=32&date=all&q=REST%20API.
2. FIELDING, Roy T. *Architectural styles and the design of network-based software architectures* [online]. Irvine: University of California, Irvine, 2000 [cit. 2020-05-19]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.
3. JANDA, Radim. *Technologie pro tvorbu RESTful API* [online]. Fakulta informačních technologií ČVUT, 2018 [cit. 2020-05-19]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/76228/F8-DP-2018-Janda-Radim-thesis.pdf.
4. HANÁK, Drahomír. Stopařův průvodce REST API [online] [cit. 2020-05-19]. Dostupné z: https://www.itnetwork.cz/programovani/nezarazene/stoparuv-pruvodce-rest-api/.
5. PATCH Method for HTTP [online]. 2010 [cit. 2020-05-19]. Dostupné z: https://tools.ietf.org/html/rfc5789.
6. Laravel - dokumentace [online] [cit. 2020-05-19]. Dostupné z: https://laravel.com/.
7. Symfony - dokumentace [online] [cit. 2020-05-19]. Dostupné z: https://symfony.com/.
8. Magento - dokumentace [online] [cit. 2020-05-19]. Dostupné z: https://devdocs.magento.com/.
9. Nette - dokumentace [online] [cit. 2020-05-19]. Dostupné z: https://nette.org/cs/.
10. Content Negotiation [online] [cit. 2020-05-19]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation.

11. FOSRestBundle - documentation [online] [cit. 2020-05-19]. Dostupné z: https://symfony.com/doc/master/bundles/FOSRestBundle/index.html.
12. SensioFrameworkExtraBundle - documentation [online] [cit. 2020-05-19]. Dostupné z: https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html.
13. Comparing Workflows - Atlassian tutorial [online] [cit. 2020-05-19]. Dostupné z: https://www.atlassian.com/git/tutorials/comparing-workflows.
14. JONÁŠ, Martin. Návrhové principy: SOLID [online]. 2012 [cit. 2020-05-19]. Dostupné z: https://www.zdrojak.cz/clanky/navrhove-principy-solid/.
15. PHP-FIG [online] [cit. 2020-05-19]. Dostupné z: https://www.php-fig.org/.
16. PHP Reflection - documentation [online] [cit. 2020-05-19]. Dostupné z: https://www.php.net/manual/en/book.reflection.php.
17. Laminas Code - documentation [online] [cit. 2020-05-19]. Dostupné z: https://docs.laminas.dev/laminas-code/.
18. Composer - A Dependency Manager for PHP [online] [cit. 2020-05-19]. Dostupné z: https://getcomposer.org/.
19. Composer - Documentation - Artifact installation [online] [cit. 2020-05-19]. Dostupné z: https://getcomposer.org/doc/05-repositories.md#artifact.
20. Markdown - Specification [online] [cit. 2020-05-19]. Dostupné z: https://daringfireball.net/projects/markdown/.
21. PHP Doc - documentation [online] [cit. 2020-05-19]. Dostupné z: https://docs.phpdoc.org/latest/.
22. phpDocumentor - documentation [online] [cit. 2020-05-19]. Dostupné z: https://www.phpdoc.org/.
23. PHPUnit - documentation [online] [cit. 2020-05-19]. Dostupné z: https://phpunit.de/.
24. Neon - dokumentace [online] [cit. 2020-05-19]. Dostupné z: https://doc.nette.org/cs/3.0/neon.

Příloha **A**

Seznam použitých zkratek

JSON JavaScript Object Notation

XML Extensible markup language

REST Representational State Transfer

API Application Programming Interface

CSV Comma-separated values

ORM Object-relational mapping

PHP-FIG PHP Framework Interop Group

PSR PHP Standard Recommendation

HTTP Hypertext Transfer Protocol

OOP Object-oriented programming

IDE Integrated development environment

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `lib` ..... zdrojové kódy implementace
  - `core` ..... hlavní implementace knihovny včetně unit testů
  - `nette` ..... referenční implementace knihovny pro framework Nette
- `text` ..... text práce
  - `bp.pdf` ..... text práce ve formátu PDF
  - `source` ..... text práce ve formátu LaTeX s příslušnými soubory
- `research` ..... dotazník prováděný v rámci práce