Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Komponenta pro přenos dat mezi e-shopem a účetním systémem

Plzeň 2019 Miroslav Soukup

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 27. června 2019

Miroslav Soukup

Poděkování

Rád bych touto cestou poděkoval Ing. Martinu Dostalovi, Ph.D. za odborné vedení mé práce, podnětné připomínky a korekci textu.

Abstract

The diploma thesis deals with the implementation of the PHP library, which ensures communication between e-shop or another web application and accounting system. The goal is to create library, which will be available via composer, and to create extension for Nette framework. The theoretical part deals with the survey of accounting systems and the description of the elements that make up the library. The practical part of this thesis contains the design of the library, its subsequent implementation and coverage of the implementation tests.

Abstrakt

Diplomová práce se zabívá realizací PHP knihovny, která zajistí komunikaci mezi e-shopem či jinou webovou aplikací a účetním systémem. Cílem je vytvořit tuto knihovnu, která bude dostupná přes composer a bude pro ni vytvořeno rozšíření do Nette frameworku. Teoretická část se zabývá průzkumem účetních systémů a popisem prvků, ze kterých se knihovna skládá. Praktická část této práce obsahuje návrh knihovny, její následnou implementaci a pokrytí implementace testy.

Obsah

1 Úvod **8**

2 Nette framework, composer a návrhové vzory **9**

2.1 Nette framework . . . 9

2.1.1 Bezpečnost Nette frameworku . . . 9

2.1.2 Architektura Nette frameworku . . . 11

2.1.3 Komponenty . . . 13

2.1.4 Latte . . . 13

2.1.5 NEON . . . 15

2.1.6 Operace s databází . . . 15

2.1.7 Tracy . . . 17

2.1.8 Nette Sandbox . . . 18

2.2 Composer . . . 20

2.3 Návrhové vzory . . . 21

2.3.1 Tvořivé návrhové vzory . . . 22

2.3.2 Strukturální návrhové vzory . . . 24

2.3.3 Návrhové vzory chování . . . 26

3 Analýza účetních systémů **28**

3.1 POHODA Stormware . . . 28

3.2 ABRA FlexiBee . . . 35

4 Návrh komponenty pro připojení na účetní systémy **38**

4.1 Funkční požadavky . . . 38

4.2 Mimofunkční požadavky . . . 38

4.3 Obecný návrh . . . 39

4.4 Připojení na účetní systémy . . . 40

4.5 Formát požadavků . . . 41

4.5.1 Systém POHODA . . . 42

4.5.2 Systém FlexiBee . . . 42

4.6 Návrh implementace unit testů . . . 42

5 Implementace a publikace komponenty **43**

5.1 Inicializace projektu . . . 43

5.2 Konektory na účetní systémy . . . 44

5.3 Parsování odpovědí . . . 47

5.4 Požadavky pro získávání dat z POHODA systému 48
5.5 Požadavky pro odesílání dat do POHODA systému 51
5.6 Požadavky pro získávání dat z FlexiBee systému 51
5.7 Požadavky pro odesílání dat do FlexiBee systému 53
5.8 Základní kolekce 54
5.9 Rozšíření pro Nette framework 54
5.10 Publikace komponenty 55

6 Testování **57**
6.1 Unit testy 57
6.1.1 Systém POHODA 58
6.1.2 Systém FlexiBee 58
6.2 Testování pomocí vzorové aplikace 59

7 Závěr **60**

Literatura a použité zdroje **61**

Seznam zkratek **63**

Seznam obrázků **64**

Seznam kódů **64**

Přílohy **66**

A Správce IIS **67**

B Struktura komponenty **68**

C Kolekce **69**

D Podoba balíčku v Packagist.org **70**

E Rozšíření pro Nette framework **71**

1 Úvod

Dnešní doba klade velký důraz na integraci dat mezi různými informačními systémy, které nemusí být vždy snadno propojitelné. Velké množství provozovatelů e-shopů či jiných webových aplikací využívá k udržování stavu účetnictví a skladových zásob software třetích stran, konkrétně účetní nebo ERP systémy.

Teoretická část se nachází ve druhé kapitole a je rozdělena na tři části. V první podkapitole je popsán Nette framework, který je v České republice poměrně rozšířený a v rámci práce je pro něj vytvořeno rozšíření, pro snadnou integraci komponenty do projektů vytvořených v Nette. Druhá podkapitola se zabývá stručným popisem composeru, pomocí něhož bude možné komponentu nainstalovat jedním příkazem. Třetí podkapitola pojednává o vybraných návrhových vzorech.

Praktická část se zabývá procesem vytvoření komponenty. V počátku třetí kapitoly jsou představeny účetní systémy obecně a je zde uveden důvod výběru následujích systémů. Konkrétně byly vybrány tyto - systém POHODA od společnosti STORMWARE s.r.o. a ABRA FlexiBee od ABRA Flexi s.r.o. Čtvrtá kapitole se zabývá se návrhem komponenty s ohledem na informace získané v analýze účetních systémů. V páté kapitole je popsána implementace komponenty a jaké problémy při implementaci nastaly. Komponenta je implementována v programovacím jazyce PHP. Šestá kapitola obsahuje popis průběhu ručního testování a popis implementace unit testů, které slouží k odhalení chyb.

Cílem této práce je vytvoření komponenty (knihovny) zajšťující napojení webové aplikace, která je napsána v PHP, na účetní systém. Komunikace mezi webovou aplikací a účetním systémem musí umožňovat získávání dat ze zmiňovaných systémů a také odesílání dat do nich. Ovládání komponenty by mělo být pro budoucího programátora, který ji bude využívat, co nejsnazší a intuitivní.

2 Nette framework, composer a návrhové vzory

V této kapitole bude představen Nette framework, jelikož v knihovně bude vytvořeno rozšíření pro snadnou integraci do toho frameworku. Dále budou představeny základy composeru a stručně popsáno několik návrhových vzorů.

2.1 Nette framework

Nette je český framework pro vývoj webových aplikací, postavený nad PHP. Jeho autorem je vývojář David Grudl. Framework je v České republice velmi rozšířený a je aktivně vyvíjen již po dobu deseti let. Jako příklad webových aplikací, které jsou v něm naprogramovány, mohou být uvedeny například knihkupectví Knihy Dobrovský[1], filmová databáze ČSFD.cz[2] nebo bonusový program obchodního řetězce Globus[3] [11] [16].

V roce 2015 se Nette framework dostal na třetí místo oblíbenosti v anketě `PHP Framework Popularity at Work` magazínu SitePoint [17]. Komunita okolo Nette je považována za nejaktivnější v České republice. Existuje mnoho rozšíření a doplňků, které komunita dále rozvíjí. Díky čistě objektově orientovanému návrhu a předpřipravené struktuře Nette podporuje vyvíjení čístého kódu a jeho případné budoucí rozšíření [11].

2.1.1 Bezpečnost Nette frameworku

Framework nativně obsahuje základní bezpečnostní prvky pro ochranu webové aplikace [11]. Ochrana je zajištěna vůči těmto bezpečnostním problémům:

- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Session hijacking, session stealing, session fixation [14]

[1] `https://www.knihydobrovsky.cz`
[2] `https://www.csfd.cz`
[3] `https://www.globusbonus.cz`

XSS

Cross-Site Scripting je typ útoku, u kterého útočník vloží svůj vlastní (škodlivý) kód na požadovanou stránku. Využívá neošetřených výstupů v aplikaci, jež je cílem útoku. Proti tomuto typu útoku se lze bránit správným ošetřením všech výstupních řetězců [14].

Příkladem může být uložení kusu kódu do databáze skrze formulář, např. pro přidávání komentářů. Kód bude uložen a u neošetřené stránky se, při jejím obnovení, zmiňovaný kód provede. Následující text představuje text zadaný do formuláře.

```
Skvely clanek! <script>alert('XSS utok')</script>
```

Kód 2.1: Ukázka XSS

Tento kus kódu při každém obnovení stránky, kde má být komentář zobrazen, pouze vypíše řetězec `XSS utok`, ale podobným způsobem je možné například zcizit identitu uživatele [14].

Nette má zabudovanou technologii, kterou nazývá `Context-Aware Escaping`. Tato technologie zamezuje Cross-Site Scriptingu bez toho, aniž by programátor musel cokoliv ošetřovat. Výstupy jsou ošetřeny automaticky frameworkem.

CSRF

Cross-Site Request Forgery je útok, který má právě přihlášeného uživatele přimět k odeslání škodlivého požadavku do webové aplikace. Škodlivá akce se potom jeví, jako že ji provedl onen přihlášený uživatel. Tyto útoky jsou prováděny pomocí formulářů. Účelem útoku není získat data, protože útočník nemůže vidět odpověď aplikace, ale změnit její stav. Příkladem může být odstranění záznamu v databázi.

Proti těmto útokům se lze bránit přidáním ověřovacího tokenu do formuláře. Nette obsahuje speciální metodu `addProtection()`, která automaticky zajistí přídání a kontrolu zmiňovaného tokenu [14].

Session hijacking, session stealing, session fixation

Tento útok je zaměřen na odcizení session či podstrčení jinému uživateli své session ID a tím získání přístupu do aplikace, aniž by znal jeho heslo. Proti takovým útokům je možné se bránit správným nastavením serverů a PHP [14].

Nette konfiguruje PHP automaticky, jediným požadavkem je mít povolenou PHP funkci `ini_set()` [14].

2.1.2 Architektura Nette frameworku

Nette framework je založen na MVC architektuře. MVC představuje zktratku tří slov a to následujících: Model, View a Controller. V Nette jsou controllery nahrazeny presentery, ale plní totožnou funkci. Tato architektura byla vytvořena za účelem oddělení kódu obsluhy (presenter) od kódu logiky aplikace (model) a od kódu, který zobrazuje data (view). Tím je zajištěno zpřehlednění kódu aplikace a zároveň je možné testovat jednotlivé části odděleně [12].

Model

Model je část aplikace, která obsahuje funkční a datovou část aplikace. Obsahuje aplikační logiku a komunikuje s datovými uložišti, například s databází. Jeho součástí jsou v podstatě veškeré akce, které mohou být uživatelem provedeny, ať už se jedná o přihlášení či uložení uživatelského vstupu do databáze. S okolím komunikuje pomocí rozhraní a ostatní vrstvy nemají žádné povědomí o existenci modelu. Při zavolání funkce zmiňovaného rozhraní je možné měnit nebo zjišťovat informace modelu [12].

View

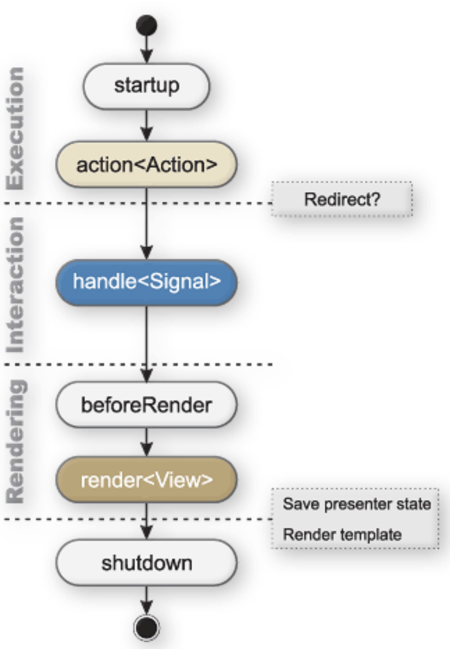
View je část aplikace, která má za úkol zobrazit výsledek požadavku, který byl odeslán uživatelem. Představuje funkci grafického rozhraní mezi uživatelem a aplikací. V Nette tuto funkci zastává šablonovací systém, který se nazývá `Latte` [12].

Presenter (Controller)

Presenter představuje prostředníka mezi modelem a view. Přijímá požadavky od uživatele. O jaký konkrétní požadavek se jedná určuje komponenta `Router`, která je popsána v kapitole 2.1.8. Na základě rozpoznaného požadavku, zavolá presenter odpovídající akci v modelu. V modelu je provedena požadovaná operace a jsou vrácena data presenteru. Presenter předá data dále do view, kde jsou zobrazena. Pro správnou funkčnost musí být presentery v Nette potomky třídy `Nette\Application\UI\Presenter` [12].

Životní cyklus presenteru

Presenter je třída, obsahující specifické metody. Objekt z ní je vytvořen v okamžiku, kdy `router` z požadavku rozpozná, o jaký konkrétní presenter se jedná. Po obsloužení požadavku je objekt zrušen. V presenteru, jak již bylo zmíněno, existují specifické metody, které je možné vidět na obrázku 2.1.



Obrázek 2.1: Nette - životní cyklus presenteru [12]

Metoda startup() je volána ihned po vytvoření nové instance presenteru. Může zde probíhat incializace proměnných nebo kontrola práv uživatele. Při přetížení této metody je vždy nutné zavolat metodu `startup()` od předka [12].

Metoda action<Action>() je metoda, která slouží pro provedení určité akce, například přihlášení uživatele či uložení dat do databáze. Je obdobou metody `render<View>()` s tím rozdílem, že po provedení akce většinou přesměruje na jinou stránku. Identifikátor `<Action>` představuje libovolný název metody, který bude použit pro konstrukci URL [12].

Metoda handle<Signal>() je metoda, zpracovávající tzv. signály. Signál je akce, která je provedena, aniž by se změnilo `view`. Metoda je určena především ke zpracování AJAXových[4] požadavků [12].

[4]AJAX - Asynchronous JavaScript and XML - zpracování požadavků bez nutnosti obnovení stránky

Metoda beforeRender() je volána před každou metodou `render<View>()` a slouží předávání parametrů, které jsou společné pro více šablon či k nastavení šablon [12].

Metoda render<View>() obvykle předává data do šablony. Šablona je vyhledána automaticky podle identifikátoru `<View>` a názvu presenteru. Například bude existovat presenter s názvem `DefaultPresenter` a v něm metoda `renderDetail()`. Na stejné úrovni jako je presenter bude v adresářové struktuře složka `templates` v ní podle názvu presenteru podsložka `default` a v ní šablona `detail.latte` [12].

Metoda shutdown() je volána při ukončení životního cyklu presenteru [12].

Životní cyklus presenteru je možné ukončit předčasně, než bude vykreslena šablona. To může být užitečné například v případě, kdy je potřeba pouze odeslat data bez vykreslení šablony. Toho se využívá u AJAXových požadavků, příkladem může být našeptávač u vyhledávače. Není potřeba překreslovat celou stránku, ale pouze měnit data v našeptávání na základě měnícího se vstupu od uživatele.

2.1.3 Komponenty

Jak již bylo řečeno, Nette podporuje best practices[5]. Komponenty představují efektivní způsob jak vytvořit znovupoužitelný kód. Komponenta je vykreslitelný objekt, například formulář pro přihlášení, menu či tabulkový výpis. Složena je ze samostatné třídy, šablony a v ideálním případě i z továrny, která bude komponentu vytvářet [7].

Komponenta je inicializována v presenteru a vykreslena v šabloně, která obsahuje klíčové slovo `control 'nazev dane komponenty'`. Mohou ji být předávány parametry a to buď přes zmiňovanou továrnu, settery či přímo při vykreslování v šabloně akce presenteru.

2.1.4 Latte

Latte je šablonovací systém vytvořený v rámci Nette frameworku. Může být však použit nezávisle na něm, například v čistém PHP, stačí jej nainstalovat přes composer do projektu a pak jej začít vyvužívat. Dokumentace k instalaci a požívání latte je uvedena na oficiálních stránkách[6] [6].

[5]Best practices - směrnice či idey reprezentující co nejefektivnější postupy

[6]https://latte.nette.org/cs/guide

Hlavními vlastnostmi latte jsou:

- **Rychlost** - latte je přeloženo na jednoduchý a optimalizovaný kód.
- **Bezpečnost** - latte automaticky provádí escapování vypisovaných dat a kontrolu odkazů, to souvisí s bezpečností popsanou v kapitole 2.1.1.
- **Intuitivnost** - psaní kódu v latte šablonách je velmi jednoduché a intiutivní [6].

Představuje vrstvu view popsanou v kapitole 2.1.2. Po zpracování požadavku presenterem, jsou šabloně předána data, která jsou v šabloně naformátována a vypsána jako HTML stránka.

Výpis předaných dat je realizován pomocí tzv. maker a filtrů, která odstraňují nutnost v šabloně psát značky `<?php ?>`. Ty jsou automaticky generovány při kompilaci Latte Enginem. Makra i filtry jsou zapisovány přímo do HTML kódu. Zápis maker je možné provést dvěma způsoby. Prvním způsobem je vložení makra do složených závorek a to následujícím způsobem `{$variable}`. Za předpokladu, že byla presenterem šabloně předána proměnná `variable`, bude vypsán její escapovaný[7] obsah [6].

Druhým způsobem využívání maker jsou tzv. `n:makra`. Jejich zápis probíhá následujícím způsobem:

```
<div n:foreach="$rows as $row">
    ...
</div>
```

Kód 2.2: Ukázka n:makra v Latte

Takto vytvořené makro projde pomocí cyklu foreach pole `$rows` a na základě toho, kolik je v něm záznamů, vytvoří stejný počet HTML elementů div. Maker existuje velké množství a opět jsou popsána na oficiálních stránkách[8].

Filtry slouží k formátování výstupů dat. V makrech byl uveden příklad výpisu proměnné `{$variable}`. Proměnná je automaticky escapována. To ale není vždy žádané, pokud je například pro ukládání článků používán WISIWYG[9] editor, speciální znaky escapované být nesmí. To lze při vypisování dat zařídít právě zmiňovanými filtry, v tomto případě konkrétně `{$variable}|noescape`. Filtry jsou vždy přidávány za znak `| - pipe` a lze jimi formátovat text, číselné hodnoty, časové údaje atd.

[7]Escapování znaků - speciální znaky jsou vypsány jako řetězec, například text `<script>alert(1);</script>` by bez escapování provedl JavaScriptový kód

[8]https://latte.nette.org/cs/macros

[9]What you see is what you get - představuje způsob editace dokumentů, která umožňuje zobrazení stejné podoby dokumentu při editaci i výsledné prezentaci

Filtry mají možnost přijímat parametry, což je vhodné například při formátování čísla, kde má být specifikován počet desetiných míst, či separátor tisíců. Zmiňovaný filtr má tuto podobu `{$variable}|number:2:',':' '`, kde parametry jsou odděleny dvojtečkou, první parametr udává počet desetinných míst, druhý znak pro oddělení desetinné části a třetí je separátor tisíců. Kompletní seznam filtrů je uveden na oficiálních stránkách[10]. Latte také umožňuje definici vlastních filtrů [6].

2.1.5 NEON

NEON jsou konfigurační soubory, sloužící k nastavení parametrů aplikace či jako registr a konfigurátor dependency injection[11] (podrobněji popsáno v kapitole 2.1.8) [13]. Soubory mají příponu `.neon` Po instalaci Nette je vytvořen vzorový konfigurační soubor a jeho formát je následující:

```
php:
    date.timezone: Europe/Prague

session:
    expiration: 14 days
```

Kód 2.3: Ukázkový záznam v NEON souboru

Parametry ve zmíněném kódu jsou definovány defaultně Nette frameworkem. Kompletní seznam defaultně definovaných parametrů je uveden v dokumentaci[12].

2.1.6 Operace s databází

Nette framework implicitně poskytuje rozhraní pro komunikaci s databází. Typy podporovaných databází jsou uvedeny v tabulce 2.1. Pro inicializaci připojení k databázi je nutné vytvořit konfiguraci v příslušném NEON souboru. Možná konfigurace vypadá následovně:

```
database:
  dsn: 'mysql:host=host.docker.internal;dbname=example'
  user: root
  password: root
```

Kód 2.4: Záznam v NEON souboru pro databázi

Klíčové slovo `database` (stejně jako `dsn, root i password`) je jedno z defaultně definovaných frameworkem. Přidáním zmiňovaného vstupu

[10] `https://latte.nette.org/cs/filters`

[11] Dependency injection - Technika pro vkládání závislostí

[12] `https://doc.nette.org/cs/3.0/configuring`

do konfiguračního souboru je možné pomocí dependency injection předávat instance tříd `Nette\Database\Connection`, která představuje vrstvu `Databse Core` a `Nette\Database\Context` představující `Databse Explorer` [8].

Database Core

Nette Database Core je základní vrstva umožňující přístup k databázi. Její pomocí lze jednoduše spravovat transakce a prováďet ručně psané dotazy. Nabízí možnost do dotazů přidávat parametry, které jsou frameworkem automaticky ošetřeny proti SQL injection[13] [9].

Database Explorer

Nette Database Explorer představuje jednoduchý způsob získávání dat z databáze. Nabízí stejné funkce jako `Database Core`, ale disponuje také možností získávat data bez nutnosti psaní SQL[14] dotazů. Explorer obsahuje metodu `table()`, které je jako parametr předán název tabulky. Metoda sama získá veškerá data z databáze. Nad metodou `table()` lze vyvolávat další metody, které simulují například omezení dotazu podmínkou, vyběr konkrétního sloupce či změnu řazení.

Databázový server	DSN jméno	Podpora v Core	Podpora v Explorer
MySQL (>= 5.1)	mysql	ANO	ANO
PostgreSQL (>= 9.0)	pgsql	ANO	ANO
Sqlite 3 (>= 3.8)	sqlite	ANO	ANO
Oracle	oci	ANO	NE
MS SQL (PDO_SQLSRV)	sqlsrv	ANO	ANO
MS SQL (PDO_DBLIB)	mssql	ANO	NE
ODBC	odbc	ANO	NE

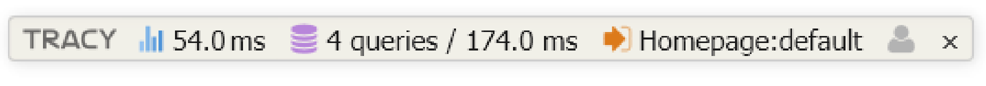
Tabulka 2.1: Podpora databází v Nette [8]

[13] SQL injection - technika napadení databázové vrstvy přes neošetřený uživatelský vstup

[14] Structured Query Language - strukturovaný dotazovací jazyk

2.1.7 Tracy

Tracy je knihovna, která slouží pro slouží pro ladění aplikace. Proto se jí také někdy říká "laděnka". Mezi její vlastnosti patří rychlé odhalení chyb, logování chyb, vypisování proměnných a měření času (např. databázových dotazů, trvání požadavku) a měření paměťových nároků [15]. Komponenta se defaultně zobrazuje v pravém dolním rohu prohlížeče, její podobu je možné vidět na obrázku 2.2.



Obrázek 2.2: Nette - Tracy [15]

Pro inicializaci Tracy je nutné ji v souboru Bootstrap.php[15] zapnout. Dá se nastavit i tak, aby byla zapnuta pouze při přístupu z určité IP adresy. To je vhodné, pokud chceme mít Tracy zapnutou při vývoji aplikace a například na testovacím serveru, ale na produkčním už musí být vypnuta.

System info

Execution time	1 761.7 ms
CPU usage user + system	1 % + 7 %
Peak of allocated memory	1.75 MB
Included files	147
OPcache	100% cached
Classes + interfaces + traits	121 + 26 + 3
Your IP	172.18.0.1
Server IP	172.18.0.2
HTTP method / response code	GET / 200
PHP	7.3.4-1+0~20190412071213.37+jessie~1.gbpabc171
Tracy	2.5.8
Server	Apache/2.4.10 (Debian)

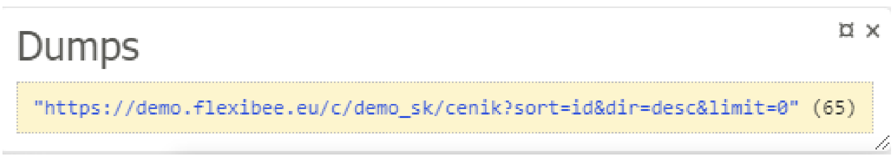
Composer Packages (20 + 1 dev) ►

Obrázek 2.3: Nette - Tracy zobrazení informací

Vypisování proměnných (obrázek 2.4) i měření časů a náročnosti (obrázek 2.3) je možné vidět přímo v Tracy. System info udává již

[15] Bootstrap.php - soubor, který se stará o inicializaci Nette frameworku

zmiňované informace a paměťové náročnosti, trvání požadavku a o jakou metodu požadavku se jednalo, verze PHP atd. Sekce `Dumps` nám umožňuje zobrazit data, která jsou potřebná pouze pro vývojové účely.



Obrázek 2.4: Nette - Tracy výpis

Zapnutá Tracy velmi přehledně vizalizuje vyvolané výjimky a syntaktický chyby zanesené do aplikace. Je zde popsáno jaká výjimka byla zachycena a její přesná zpráva. Detaily, jako je zdrojový soubor atd. je možně interaktivně procházet pomocí odkazů a je možné si přímo v chybě procházet kód, na jakém místě byla výjimka zachycena, i kde byla vyvolána.

2.1.8 Nette Sandbox

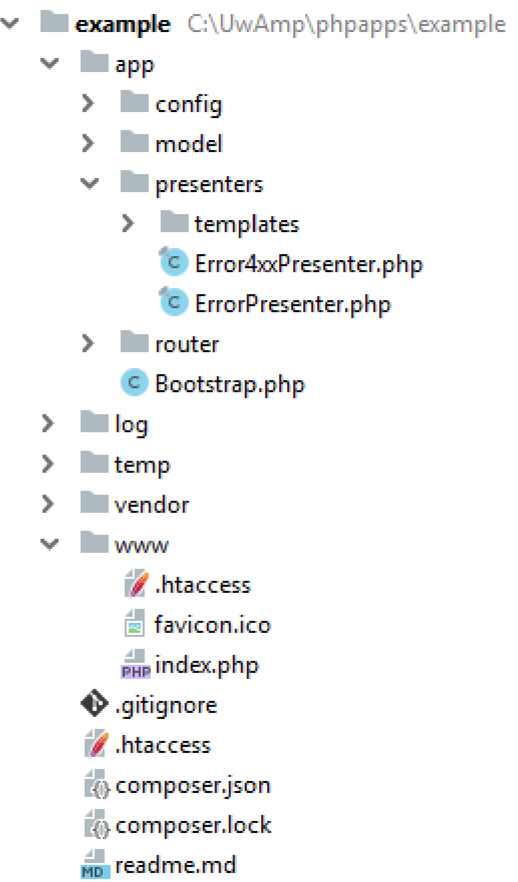
Pro začátek práce s Nette frameworkem je třeba mít takzvaný sandbox. Sandbox je jinak řečeno kostra pro webovou aplikaci, předpřipravená právě tímto frameworkem. Je možné jej stáhnout přímo ze stránek Nette[16], kde je k aktuálně[17] dispozici verze 3.0, vyžadující PHP 7.1, nebo pomocí composeru: `composer create-project nette/web-project`. Tento příkaz vytvoří kostru aplikace prozatím[17] ve verzi 2.4, která vyžaduje PHP 5.6. Composer bude detailněji popsán v následující kapitole. Autor uvádí, že každá verze je aktivně udržována po dobu jednoho roku a kritické a bezpečnostní chyby jsou opravovány po dobu dvou let [10].

Na obrázku 2.5 je zobrazena možná podoba struktury projektu (sandboxu). Sandbox obsahuje několik podadresářů. Jsou to app, log, temp, vendor a www. Složka www obsahuje soubory, které jsou veřejně přístupné, takže například CSS styly, JavaScript či obrázky. Vendor je složka obsahující závislosti projektu, nainstalované přes composer. Složka temp obsahuje dočasné soubory, mohou být generovány frameworkem, například cache, nebo samotnou webovou aplikací. Log je složka uchovávající chybové stavy, které v aplikaci nastanou, či programátorem nadefinované logy v projektu.

[16] `https://nette.org/cs/download`
[17] K datu 15.6.2019

Složka app je dále dělena na několik podsložek. První z nich je `model`, ve které jsou uchovávány třídy, které obstarávají logickou stránku aplikace. `Presenter` je složka obsahující presentery a šablony (views), v Nette konkrétně jsou to soubory s příponou `.latte`. Tyto tři prvky představují MVC architekturu, která byla popsána v kapitole 2.1.2.

Config je složka, ve které jsou ukládány konfigurační soubory s příponou `.neon`. Tyto soubory slouží ke konfiguraci aplikace, registraci rozšíření a tříd pro dependency injection kontejner. Nette si tento kontejner vytváří automaticky na základě tříd registrovaných v těchto souborech. V kontejneru se při startu aplikace vytvoří instance registovaných tříd. Sestavování závislostí ovšem funguje opačně, než by se mohlo zdát. Aplikace se při spuštění podívá do všech tříd v projektu, jaké mezi sebou mají závislosti. Poté zkontroluje zda jsou tyto závislosti registoványzy v konfiguračním souboru a jestli zde nemá nějaké další parametry a až poté instanci vytvoří.



Obrázek 2.5: Nette - struktura sandboxu

Ve složce router je obsažena komponenta zvaná Router. Tato komponenta je předřazená před presentery a slouží k směrování požadavků na konkrétní presenter na základě uživatelem zadané URL. Defaultní nastavení routeru funguje tak, že požadavky směruje podle názvu presenteru a podle akcí (metod) v něm definovaných. Například URL ve tvaru `localhost/product/detail/1` reprezentuje požadavek na presenter pojmenovaný ProductPresenter, jeho akci `actionDetail($id)`, kde číslo jedna udává hodnotu parametru `$id` této akce. V akci se typicky volají metody z modelů, kde se zpracuje požadovaná logika a po provedení těchto operací jsou předána data do šablony, která zobrazí výsledek.

2.2 Composer

Composer je nástroj pro správu závislostí PHP projektů. Umožňuje deklarovat knihovny, na kterých je projekt závislý a dokáže je pomocí jednoduchých příkazů instalovat a aktualizovat.

Composer byl vyvinut dvěma lidmi. Prvním z nich je Nils Adermann a druhým je Jordi Boggiano. První verze composeru byla publikována v roce 2013 [5].

Pro používání composeru je nutné ho nejdříve nainstalovat na počítač. V závisloti na operačním systému instalace probíhá různě. Na operačním systému Windows je třeba stáhnout spustitelný .exe soubor a dle instrukcí jej nainstalovat. Pro použití v composeru v příkazové řádce v operačním systému Windows je ještě nutné přidat nainstalovaný composer do systémových cest operačního systému. Instalace na operačních systémech Linux a macOS je o něco složitější a je detailně popsána na oficiálních stránkách getcomposer.org.

Závislosti projektu jsou definovány v souboru `composer.json`. Tyto závislosti je možné přidávat buďto ručním zápisem, nebo pomocí příkazu `composer require balíček "verze"`, kde balíček udává název knihovny, na které chceme vytvořit závislost a verze udává konkrétní verzi knihovny, kterou chceme v projektu použít. Verze knihovny není povinná a pokud není udána, bude přidána závislost na nejnovější verzi. Konkrétním příkladem použití tohoto příkladu může být přidání knihovny vytvořené v rámci této práce příkazem: `composer require soukupm/acc-sync "0.1.7-alpha"`. Tento příkaz vytvoří v souboru `composer.json` následující záznam.

```
{
    "require": {
        "soukupm/acc-sync": "0.2.0"
    }
}
```

Kód 2.5: Záznam v souboru composer.json

Instalace knihoven, které jsou již v projektu nadefinované, probíhá pomocí příkazu `composer install`, který vezme závislosti primárně ze souboru `composer.lock`, nebo ze souboru `composer.json`, pokud soubor s příponou lock neexistuje. Soubor `composer.lock` je vytvořen při instalaci příkazem `composer install` a je aktualizován pokaždé při přidání nového balíčku nebo při aktualizaci knihoven, která je popsána dále.

Pro aktualizaci knihoven se používá příkaz `composer update`. Tento příkaz aktualizuje všechny knihovny uvedené v souboru `composer.json`. Knihovny se dají aktualizovat i individuálně a to uvedením názvu knihovny za klíčovým slovem update.

Knihovny se ukládají do složky `vendor`, která je automaticky vytvořena na stejné úrovni jako soubor `composer.json`.

Composer využívá repozitáře, aby věděl, odkud má knihovny stahovat. Centrálním repozitářem je Packagist. Funguje v podstatě jako registr odkazů na repozitáře verzovacích nástrojů jako je Git nebo Svn. Je také možné si vytvářet vlastní composer repozitáře.

2.3 Návrhové vzory

Návrhové vzory obecně poskytují řešení, často se opakujících problémů při návrhu aplikace [1].

Neposkytují však konkrétní postup toho, jak daný problém implementovat v kódu, ale pouze popis, či šablonu, jak je možné se s problémem vypořádat v různých situacích. Mohou urychlit vývoj poskytnutím testovaných, ověřených vývojových paradigmat. Používání návrhových vzorů napomáhá prevenci proti malým chybám, které by mohli způsobit velké problémy v budoucnu a zlepšují čitelnost kódu [19].

Návrhových vzorů existuje velké množství a v PHP je lze dělit do tří skupin a to tvořivé, strukturální a vzory chování. V této kapitole budou popsány základní návrhové vzory, z nichž některé byli použity při realizaci této práce.

2.3.1 Tvořivé návrhové vzory

V této kapitole budou popsány návrhové vzory, které slouží k vytváření objektů, bez použití klíčového slova `new`.

Vybrány byly tyto čtyři:

- Singleton - Jedináček
- Factory method - Tovární metoda
- Builder - Stavitel
- Abstract factory - Abstraktní továrna

Singleton

Singleton je jeden z nejpoužívanějších a zároveň nejjednodušších návrhových vzorů. Zajišťuje, že od dané třídy bude vytvořena pouze jedna instance. Zároveň by tato instance měla být globálně dostupná [1].

Při užití tohoto návrhového vzoru je potřeba dodržet následující:

- Vytvoření centrálního bodu pro přístup k instanci daného objektu.
- Centrální přístupový bod musí vždy vracet stejnou instanci objektu, nezávisle na počtu volání.
- Zabránění vytvoření další instance tohoto objektu [1].

Příkladem může být například třída, která bude zajišťovat připojení k databázi, protože připojení k databázi by při běhu aplikace mělo zůstat neměnné.

Factory method

Facotry method má programátora odstínit od neustáleho využívání klíčového slovíčka `new` při vytváření nových instancí konkrétních objektů, protože takto vytvářené objekty roztrousí závislosti po celé aplikaci. Poté bude při změně třídy (úpravy parametrů konstruktoru), ze které se instance vytváří, nutné vyhledat všude v kódu, kde je vytvoření realizováno a vše řádně upravit [1].

Tento návrhový vzor definuje rozhraní pro vytváření objektů a vytváření instancí tříd přenechává svým potomkům. Potomci poté rozhodují o tom, jaká implementace se má použít [1].

Při užití tohoto návrhového vzoru je možné postupovat například takto:

- Vytvořit abstraktní třídu, která bude obsahovat definici nezbytné struktury.
- V této třídě implementovat kód, který bude pro všechny potomky stejný.
- Implementovat potomky abstraktní třídy, kteří se budou starat o vytváření různých instancí, které potřebujeme [1].

Příkladem může být v přeneseném významu třeba továrna na dopravní prostředky. Abstraktní třídou bude továrna, která bude mít nějaké společné části a její potomci potom továrny na konkrétní typy dopravních prostředků. Tím odstíníme programátora od dalšího vytváření závislostí.

Builder

Návrhový vzor Builder odděluje konstrukci komplexních objektů od jejich reprezentace tak, že stejný konstrukční proces může vytvořit jiné reprezentace [19].

Následujícími kroky vytvoříme strukturu podle návrhového vzoru Builder:

- Vytvoření třídy, která bude reprezentovat výslednou instanci (produkt) [1]. Pod tím by bylo možné si předtavit obyčejnou třídu, která bude reprezentovat HTML stránku. Bude mít pro zjednodušní jeden atribut, nadpis.
- Definice rozhraní stavitele, který vytvoří objekt předchozího bodu (builder)[1]. Například abstraktní třída, která bude vytvářet instanci objektu z předchozího bodu a abstraktní metody, které naplní atributy této instance.
- Vytvoření alespoň dvou konkrétních implementací předchozího bodu [1].
- Vytvoření třídy, která pomocí stavitele z předchozího bodu řídí sestrojení objektu (direktor). Třída, které předáme závislost na předchozí třídu a ta z ní vytvoří výsledný produkt [1].

Abstract factory

Abstraktní továrna je návrhový vzor, který zapouzdřuje skupinu individuálních továren pro vytváření příbuzných nebo závislých objektů, bez specifikace konktrétních tříd těchto objektů. Po konkrétní

implementaci abstraktní továrny, se poté již používá jen obecné rozhraní této továrny k vytváření dalších objektů [1].

Při aplikování tohoto návrhového vzoru je třeba provést tyto kroky:

- Definice abstraktní třídy či rozhraní, které bude deklarovat metody pro vytváření každého z požadovaných objektů.
- Implementace abstraktních tříd, reprezentující jednotlivé objekty ze skupiny objektů.
- Implementace libovolného množství potomků těchto tříd.
- Implementace továrny, která bude vytvářet objekty z výše definovaných tříd.
- Pomocí Dependency Injection (předávání závislostí) nebo Factory Method předat továrnu do místa, kde ji potřebujeme. Takto bude v budoucnu snadné továrnu přípdaně vyměnit [1].

2.3.2 Strukturální návrhové vzory

Strukturální návrhové vzory slouží k pospojování více jednoduchých objektů do větších struktur [1].

V této kapitole budou zjednodušeně popsány tyto návrhové vzory:

- Adapter - Adaptér
- Decorator - Dekorátor
- Facade - Fasáda

Adapter - Adaptér

Při vývoji aplikace je možné se setkat s objektem, který není kompatibilní s námi požadovaným rozhraním, například při spolupráci dvou rozdílných aplikací se stejným zaměřením. Návrhový vzor adaptér přizpůsobí rozhraní nekompatibilní třídy našemu požadovanému rozhraní a umožní tak spolupráci tříd, které by kvůli rozdílným rozhraním běžně spolupracovat nemohly [1].

Při implementaci adaptéru je potřeba provést následující kroky:

- Lokalizace rozdílů mezi požadovaným a nabízeným rozhraním.
- Implementace třídy, která bude poskytovat požadované rozhraní.

- Vytvoření způsobu předání objektu do adaptéru, který má být přizpůsobený, například pomocí dependency injection.
- Implementace metod vyžadovaných rozhraním a delegování požadavků na odpovídající metody původního objektu.
- Přihlížení k signalizaci chyb.
- V aplikaci využívat objekt adaptéru a pomocí něj obalit původní objekt [1].

Decorator - Dekorátor

V aplikaci je možné se setkat s požadavkem, že je potřeba rozšířit požadovanou třídu o novou funkčnost za běhu aplikace. Při dědičnosti tohoto nelze docílit, protože je tato funčknost přidána při kompilaci a je spjatá s konkrétní třídou. Návrhový vzor dekorátor rozšiřuje objekt o novou funkčnost, nebo dokáže měnit jeho metody za běhu aplikace [1].

Pro vytvoření dekorátoru je tředa dodržet následující postup:

- Vytvoření základní třídy, která má stejný typ jako objekt, který chceme dekorovat. Vytvoření potomka této třídy a implementace metod rozhraní.
- Ve vytvořené základní třídě provést definici konstruktoru, která bude přijímat jako parametr objekt, který je dekorovaný a tento objekt uložit ve třídě dekorátoru.
- Implementace všech metod základní třídy tak, že se bude volání delegovat na objekt, který má být dekorován.
- Implementace konkrétních dekorátorů, kteří jsou potomky třídy z předchozího bodu. Dosáhnout je toho možné tak, že se metody v tomto potomkovi přetíží a nadefinuje se v nich nová požadovaná funkčnost.
- Kombinací více dekorátorů lze dosáhnout jejich vzájemným dekorováním [1].

Facade - Fasáda

Fasáda je návrhový vzor, který poskytuje zjednodušené rozhraní pro sérii rozhraní. Úkolem fasády je odstínění od složitých objektových kompozic za jednoduchým rozhraním. Pod jednoduchým rozhraním se rozumí,

že ostatní progamátoři aplikace nemusí podrobně znát všechny třídy, které fasáda zastřešuje, ale pouze tuto fasádu.

Následujícími kroky docílíme vytvoření návrhového vzoru fasáda:

- Identifikace tříd systému, které mají být skryty za fasádou.
- Definice operací (metod), které má fasáda umožňovat.
- Implementace fasády a vytvoření metod z předchozího bodu. V těchto metodách bude fasáda schopna přistupovat ke schovaným třídám.
- Konkrétní implementace vytvořených metod, které budou poskytovat požadovanou funkčnost a umožní zjednodušený přístup ke schovaným komponentám [1].

2.3.3 Návrhové vzory chování

Návrhové vzory chování jsou takové vzory, jež se zabývají chováním a interakcí objektů. Nepopisují pouze učastnící se objekty a třídy, ale i komunikaci mezi nimi za běhu aplikace.

Stručně budou popsány tyto návrhové vzory:

- Observer - Pozorovatel
- Mediator - Prostředník

Observer - Pozorovatel

V aplikaci, která je robustnější a obsahuje velké množství tříd je potřeba, aby některé tyto třídy mezi sebou komunikovali. Tím ale může vzniknout pevná vazba mezi třídami, které spolu mají komunikovat. Návrhový vzor pozorovatel se stará o to, že jsou závisloti mezi třídami předávány nepřímo. Zároveň může mít pozorovaný objekt několik pozorovatelů [1].

Přesná definice vypadá takto: návrhový vzor pozorovatel definuje závislost 1:N mezi subjektem (pozorovaným) a libovolným počtem observerů. Při změně stavu subjektu jsou o této skutečnosti automaticky informováni všichni pozorovatelé [1].

Implementace návrhového vzoru observer vyžaduje následující kroky:

- Definice rozhraní pozorovatele (Observer), které bude mít metodu, pomocí níž budou informováni pozorovatelé a změně stavu pozorovaného objektu.

- Definice rozhraní pozorovaných objektů (Observable), které bude obsahovat metody pro přidání a odstranění pozorovatelů a metodu pro upozornění na změnu stavu pozorovaného objektu.
- V objektu, který má být pozorován implementovat rozhraní Observable.
- V metodách pozorovaného objektu, které mění jeho stav, umístit volání metody pro upozornění na změnu tohoto stavu.
- Implementace konkrétních pozorovatelů [1].

Mediator - Prostředník

Prostředník odstraňuje vazby mezi objekty, jež spolu komunikují a tím zajišťuje možnost komunikace těchto objektů, které nejsou v přímé integraci. Vazby mezi těmito objekty se potom mění z M:N na 1:N [1].

Pro implementaci prostředníka je nutné dodržet tyto kroky:

- Defince rozhraní prostředníka [1].
- Definice rozhraní objektů, které budou ke komunikaci využívat objekt z předchozího bodu. Rozhraní bude obsahovat metodu, kterou bude volat konkrétní prostředník [1].
- Implemntace konkrétního prostředníka. Tento prostředník bude obsahovat logiku, která bude zprostředkovávat komunikaci mezi objekty.
- Implementace konkrétního objektu, který bude kominkovat pomocí prostředníka.

3 Analýza účetních systémů

Ekonomické, účetní a ERP (Enterprise Resource Planning) systémy slouží ke správě a řízení podniku a evidenci například účetnictví, skladových zásob, objednávek atd. Takovýchto systémů existuje na trhu obrovské množství. Jako příklad mohou být uvedeny ABRA FlexiBee, POHODA Stormware, Money S3 (S4, S5) nebo Helios. S těmito systémy se ve většině případů pracuje pomocí desktopového softwaru, který má vzdálené datové uložiště.

K realizaci této práce byly zvoleny dva systémy a to ABRA FlexiBee a POHODA Stormware. Důvodem pro volbu těchto systémů bylo to, že mají volně dostupné prostředí pro testování. V případě systému POHODA je to jejich program, který je přístupný v plné verzi pro omezený počet záznamů. V případě FlexiBee je to trial verze jejich programu na tři měsíce, či online neomezené demo.

3.1 POHODA Stormware

POHODA je účetní a ekonomický systém, který nabízí velké množství funkcí a práci s různými agendami, jako je účetnictví, faktury, skladové zásoby, objednávky, mzdy atd. Pro zákazníky nabízí několik plánů. Plány jsou rozděleny podle velikosti zákazníka, který chce tento systém využívat a také podle počtu nabízených funkcí. Od těchto parametrů se poté odvíjí cena, která je přibližně od 2000 Kč až do řádově 100 000 Kč. Produkt se kupuje jednorázově.

Existuje také možnost program využít zdarma v plné verzi, tato verze se jmenuje POHODA Start a je dostupná na oficiálních stránkách. To je ovšem ale omezeno na jednu účetní jednotku (firmu), na určité množství počtu záznamů v databázi a na počet importovaných a exportovaných dávek. Počty záznamů se liší pro různé agendy. Maximální počet záznamů je uváděno 500 pro účetní deník, pokud bude provedena registrace, jinak je to 200 záznamů, dále se tyto čísla snižují. Po vyčerpání těchto hodnot je možné databázi smazat a znovu vytvořit.

Jako hlavní výhody a funkce jsou udávány tyto:

- Praktický design, jednoduché ovládání a vytváření tiskových sestav
- Adresář, kontakty a úkoly
- Daňová evidence, účetnictví a DPH

- Fakturace a objednávky
- Hotovostní prodej
- Elektronická evidence tržeb (EET)
- Sklady
- Majetek
- Mzdy a personalistika
- Kniha jízd a cestovní příkazy
- Internetové obchody a homebanking
- Import a export dat
- Volitelné parametry
- Bezpečnost dat
- Hosting [20]

Program POHODA má zabudovaný HTTP server pro online komunikaci s externími službami, nazýván je `POHODA mServer`. Komunikace probíhá na bázi XML požadavků a odpovědí. Pro umožnění této komunikace je nutné server zpřístupnit v rámci sítě. Server je vhodné umístit v rámci interní sítě a umožnit k němu přístup pomocí VPN, pokud toto řešení není možné, existuje druhá varianta. Druhou variantou je vytvoření HTTP konektoru, který bude umístěný na webserveru, který má přístup do interní sítě, tento konektor poté funguje jako jednoduchý proxy server mezi mServerem a klientem [21].

Program POHODA je možné nainstalovat pouze na operačním systému Windows. Pro správnou funkčnost POHODA mServeru je potřeba mít nainstalovanou systémovou funkci zvanou `Internetová informační služba (IIS)`. Toto rozšíření je možné stáhnout na oficiálních stránkách společnosti Mircrosoft [21].

Po instalaci tohoto rozšíření se otevře systémové nastavení pod názvem `Správce Internetové informační služby`. Pro správnou funkčnost mServeru je zde nutné nastavit několik položek a parametrů.

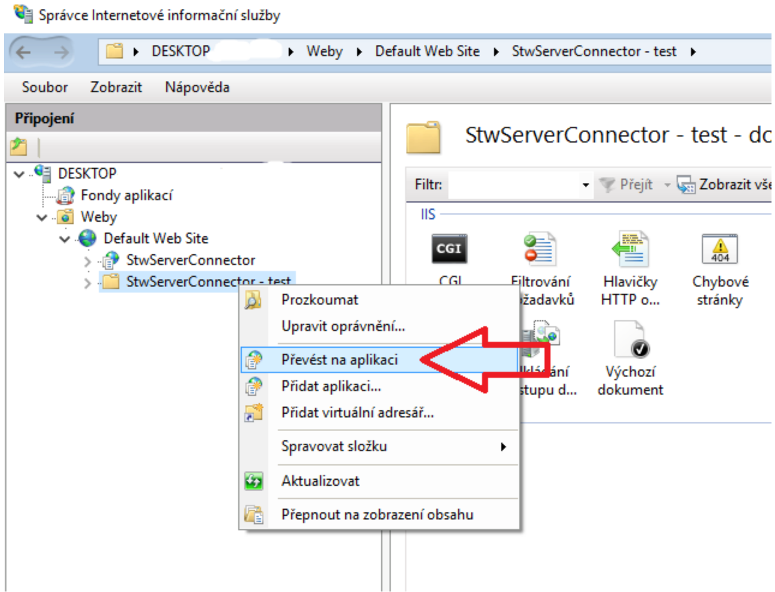
Součástí instalačního balíčku programu POHODA jsou být dva soubory a to `StwServerConnector.dll` a `Web.config`. Tyto dva soubory se nachází ve složce, kde je program nainstalován, případně je možné je stáhnout na oficiálních stránkách [18] v sekci `ASP konektor pro IIS`. Soubory budou

[18] `https://www.stormware.cz/pohoda/xml/mserver/provyvojare`

potřebné při nastavování Internetové informační služby.

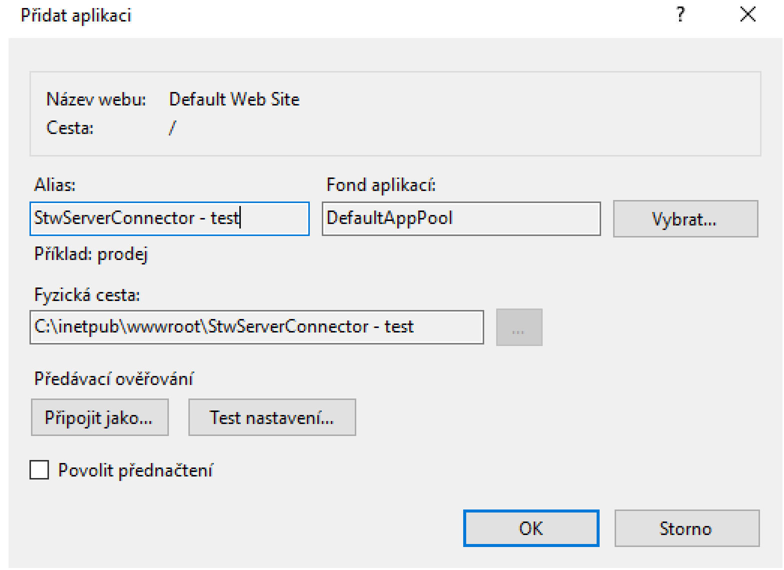
Po otevření Informační internetové služby bude otevřeno okno, zobrazené na obrázku v příloze A Správce IIS. Po kliknutí na záložku `Default Web Site` se v pravé části zobrazí `Akce`. Zde je třeba kliknout na tlačítko `Prozkoumat`. Orevře se průzkumník systému Windows s nastavenou cestou do kořenového adresáře webového serveru. V tomto adresáři musí být ručně vytvořena složka s názvem `StwServerConnector`, do ní budou umístěny dříve zmiňované soubory, kde soubor `StwServerConnector.dll` musí být ještě umístěn do podadresáře `bin` [21].

Předchozí krok vytvoří ve stromu nastavení další záložku jak je vidět na obrázku 3.1 (pro demonstrační účely byla v mém případě vytvořena složka znovu s příponou `- test`). Po kliknutí pravým tlačítkem na požadovanou složku se otevře kontextové menu, kde vybereme položku `Převést na aplikaci`.



Obrázek 3.1: IIS - převedení na aplikaci

Provedení předchozí akce vyvolá otevření okna pro nastavení webového serveru, jak je možné vidět na obrázku 3.2. Nastavení zůstane nezměněné a jen jej potvrdíme tlačítkem `OK`.



Obrázek 3.2: POHODA - Nastavení webového serveru

Konfiguraci HTTP připojení je možné nastavit v souboru `Web.config`. Pro směrování požadavků na `mServer` je nutné vyplnit údaje v uzlu `<StwServer></StwServer>`. Pro testování v lokálním prostředí nastavení vypadá takto:

```
<StwServer>
    <ServerMappings>
        <add serverName="mServer1"
             protocol="http"
             host="localhost"
             port="1111"/>
    </ServerMappings>
</StwServer>
```

Kód 3.1: Ukázka lokálního nastavení mServeru

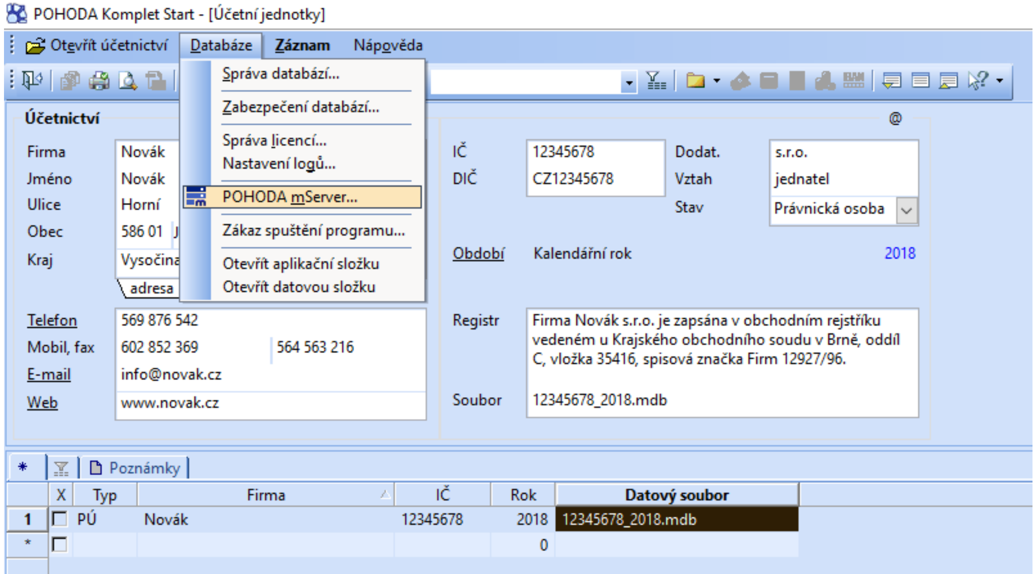
Element `add` má několik atributů a jejich význam je následující:

- serverName - název mServeru, který je vytvořen v programu POHODA
- protocol - komunikační protokol
- host - IP adresa, na které je spuštěn mServer
- port - port, na kterém mServer komunikuje a který byl zadán v konfiguraci programu POHODA

Po tomto základním nastavení bude možné komunikovat se serverem pomocí HTTP protokolu. Pro tento konkrétní příklad to bude na adrese

http://localhost:1111. Detailnější nastavení je popsáno v oficiální dokumentaci [19] od společnosti Stormware.

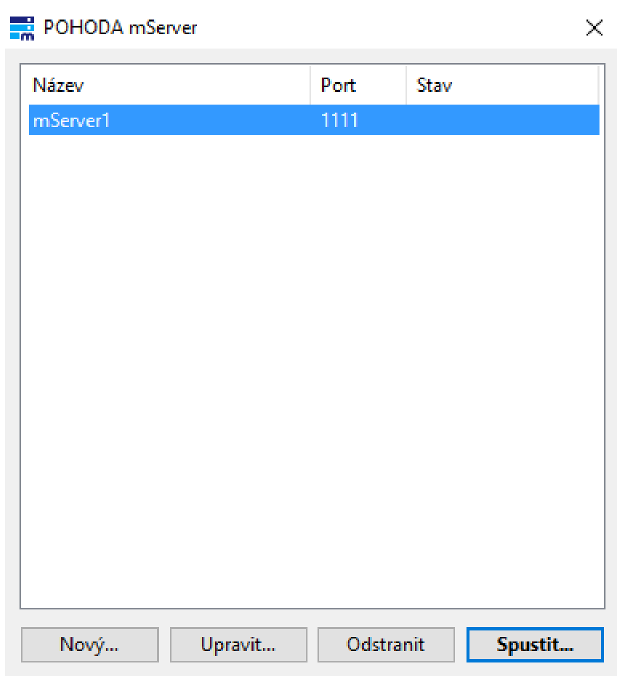
Spuštění POHODA mServeru se provádí přímo v programu. Po spuštění se otevře okno zobrazené na obrázku 3.3. V horním menu se nachází položka Databáze, ve které je možnost POHODA mSever...



Obrázek 3.3: POHODA - Spuštění mServeru

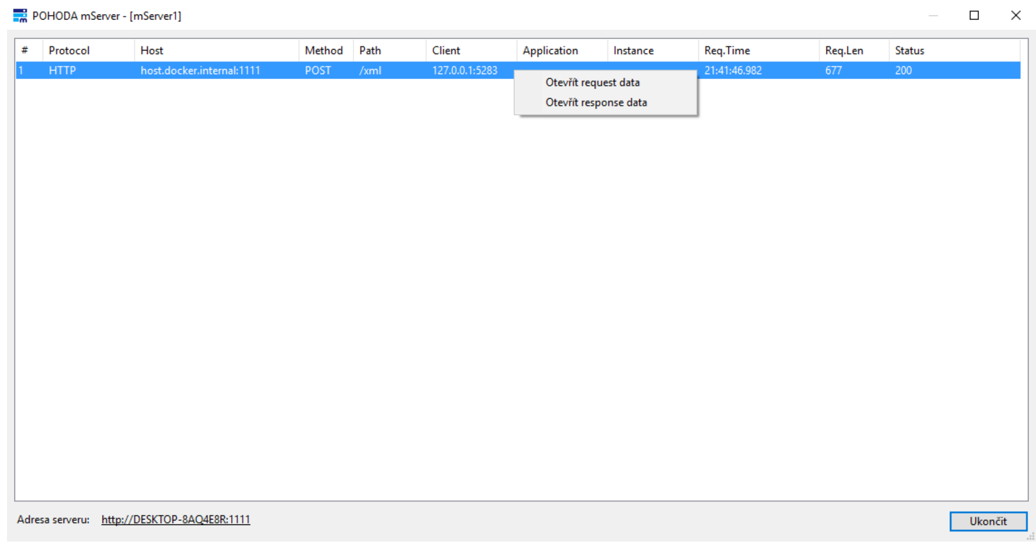
Po kliknutí na POHODA mSever... se otevře okno se seznamen všech nadefinovaných serverů, obrázek 3.4. Po stisknutí tlačítka spustit bude server spuštěn a připraven pro odesílání a přijímaní požadavků. Pokud je mServer spuštěn, není v aktuálně spuštěné instanci programu možné provádět jiné akce. Když bude uživatel chtít s programem dále pracovat, musí být program spuštěn ještě jednou v nové instanci.

[19] https://www.stormware.sk/xml/mServer/navod_Nasazení_HTTP_konektoru_pro_mServer.pdf



Obrázek 3.4: POHODA - Seznam mServerů

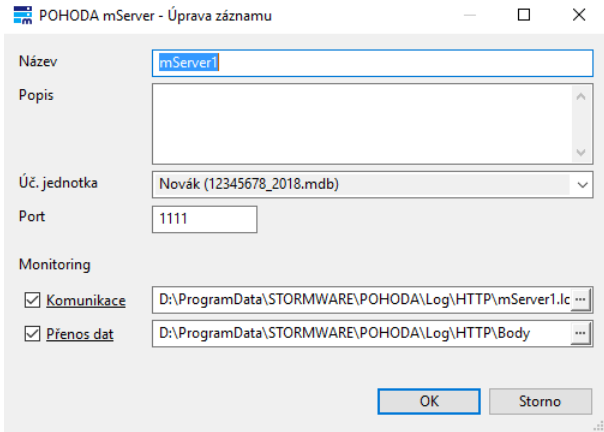
Spuštění mServeru vyvolá nové okno, které zobrazuje přijaté požadavky, obrázek 3.5. U každého požadavku je možnost zobrazit si data požadavku, stejně jako odpovědi. Tato možnost je vyvolána kliknutím pravým tlačítkem myši na konkrétní požadavek a data budou otevřena v defaultním textovém editoru operačního systému.



Obrázek 3.5: POHODA - Seznam požadavků

Při vytváření či upravování mServerů bude otevřeno dialogové okno

(obrázek 3.6), ve kterém je nutné vyplnit několik údajů. Údaje vybraných položek musí souhlasit s nastavením v souboru `Web.config`. `Název` odpovídá položce `serverName` a je to název serveru. `Popis` není povinný a slouží pouze pro informační účely. Položka `Úč. jednotka` udává pro kterou konkrétní firmu (databázi) má být server spuštěn. `Port` odpovídá položce `port` v konfiguračním souboru a udává, na kterém portu bude server spuštěn. Checkboxy `Komunikace` a `Přenos dat` pouze udávají, zda mají být logovány požadavky a data do souboru. Umístění souboru je uvedeno v odpovídajícím textboxu.



Obrázek 3.6: POHODA - Úprava či vytvoření mServeru

Základní pravidla pro komunikaci s mServerem jsou následující:

- POHODA mServer pro přenos dat využívá HTTP protokol. Požadavky i odpovědi jsou složeny z textových hlaviček a binárních dat.
- Přenos dat je realizován pomocí XML. Parametr `Content-Type` v HTTP hlavičce musí být nastaven na `text/xml`.
- Hlavička HTTP požadavku musí obsahovat parametr `Stw-Authorization`. Hodnota parametru je ve formátu jméno a heslo odděleno dvojtečkou a zakódované BASE64. Jméno a heslo odpovídají přístupovým údajům do programu POHODA.
- Hlavička HTTP požadavku může obsahovat parametr `Content-Length` s délkou (počtem znaků), které jsou obsaženy v XML.
- Pro přenos XML dat je povolena pouze metoda POST. Metoda GET pro přenos XML dat není podporována.
- Veškerá XML data jsou uložena v kódování Windows-1250.

- Metodu GET lze použít pro přenos souborů ze složky dokumentů firmy.
- mServer podporuje kompresi přenášených dat a to metodou `gzip/deflate` [22].

Požadavky na XML data, tedy metodou POST, se posílají na adresu `/xml`. Z příkladu uvedeného dříve by tedy výsledná URL vypadala takto: `http://localhost:1111/xml`. Zpracování požadavků probíhá synchronně. V případě odeslání většího počtu požadavků na POHODA mServer najednou, dojde k jejich serializaci a budou zpracovány v takovém pořadí, v jakém na server přišly [22].

Požadavky i odpovědi se velmi liší na základě toho, s čím chce uživatel pracovat a zda chce z POHODA systému data získávat, či je do systému ukládat. U požadavků pro získávání dat je velká část XML pro všechny stejná, jako je například definice verze XML a kódování, definice schémat požadavku, atribut pro IČO nebo atribut `application`. Poté následují položky a elementy specifické pro konkrétní požadavek. Je také možné přídávat filtry, na základě kterých budou nevyhovující záznamy z výsledného XML vyřazeny.

Odpovědi na požadavky pro získání dat se liší skoro vždy. Elementy jednotlivých položek výsledného XML (například položky ve skladové zásobě) se mění na základě toho, jaké atributy mají dané záznamy v programu POHODA vyplněny a kolik záznamů stejného typu se v programu nachází.

Požadavky pro ukládání záznamů do systému jsou velmi podobné jako odpovědi na požadavky pro získání dat. Jsou poměrně proměnlivé, na základě toho, kolik máme informací o konkrétní položce, která má být uložena atd. Odpovědi na tyto požadavky jsou jednoduché, obsahují informaci o tom, zda bylo uložení úspěšné. Pokud uložení neproběhlo podle očekávání, je v odpovědi uvedena chyba.

3.2 ABRA FlexiBee

ABRA FlexiBee je účetní systém, který stejně jako POHODA, nabízí práci s agendami jako účetnictví, faktury, skladové zásoby, mzdy atd. Je určen pro živnostníky a menší firmy. Na rozdíl od systému POHODA, FlexiBee nabízí hrazení paušálně každý měsíc, nebo je zde také možnost provést platbu jednorázově [4].

Cena pro paušální i jednorázové hrazení se liší na základě počtu uživatelů a množství funkcí, které bude systém poskytovat. Cena

pro jednoho uživatele se základní funkčností je udávána 295 Kč/měsíc, nebo 3 950 Kč jednorázově plus stejná částka za každého dalšího uživatele. Pokročilá funkčnost potom 595 Kč/měsíc/uživatel či 6 950 Kč/uživatel jednorázově a plná funkčnost 795 Kč/měsíc/uživatel či 9 950 Kč/uživatel jednorázově. Konkrétní rozdíly v poskytovaných funkcích jednotlivých plánů jsou uvedeny v oficiálním ceníku[20]. Je zde také možnost si program vyzkoušet na měsíc zdarma [4].

Pro vývojáře je poskytována trial verze programu na tři měsíce. Dále je zde možnost využít online demo[21]. Online demo je neomezené a obsahuje přednastavené šablony firem i živnostníka. Na stánkách FlexiBee je uvedeno, že jsou přednastavené firmy jednou za čas obnoveny do původních verzí.

Program FlexiBee je podporován operačními systémy Windows, Mac OS X a Linux. Dále existují dvě možné verze programu. První je kompletní instalační soubor, který umožní instalaci na jediném počítači či na serveru. Druhou možností je odlehčený klient, který pouze přistupuje k datům uloženým na serveru. FlexiBee poskytuje cloudové řešení, kdy společnost využívající tyto služby nemusí mít vlastní serverou instalaci programu a pouze přistupuje na cloud pomocí klienta. Bohužel program neobsahuje možnost zobrazení požadavků ani odpovědí jako tomu je u POHODA systému [3].

FlexiBee poskytuje vývojářům API[22] pro přistup k datům z externích systémů, které je postavené na HTTP protokolu. Server, na kterém je API dostupné, je nainstalován s kompletním instalačním balíčkem programu FlexiBee. Při lokální instalaci bude tedy server nainstalován přímo na daném počítači a přistupovat se k němu bude přes lokální adresu[23]. Položka `c` je pevně definovaná a bude v URL vždy. Další část URL `nazev_spolecnosti` udává zadaný název společnosti při jejím vytváření v programu FlexiBee. Pokud bude instalace programu provedena na serveru, budeme k API přistupovat pomocí adresy serveru, příkladem může být demo[24], kde `demo` udává název společnosti [3].

Další možnost práce s URL je přidávání filterů, pro bližší specifikaci požadovaného výsledku. Filtry mohou být velmi jednoduché, ale také poměrně složité. Pokud do URL[25] přidáme za evidenci parametr, který bude nabývat číselné hodnoty, systém porovná tento číselní parametr s ID záznamů v dané evidenci a vrátí buď jeden nebo žádný záznam z ní. Filtry

[20] https://www.flexibee.eu/cenik/
[21] https://demo.flexibee.eu/c/
[22] Application Programming Interface - Rozhraní pro programování aplikací
[23] http://localhost/c/nazev_spolecnosti
[24] https://demo.flexibee.eu/c/demo
[25] https://demo.flexibee.eu/c/demo/cenik/1

je možné vytvářet i různě kombinované[26] a zároveň lze výsledky řadit podle určitých atributů (parametr `sort=id` bude řadit podle sloupce ID a parametr `dir=desc` udává řazení sestupně), které jsou u dané evidence a také lze omezit počet výsledků (parametr `limit=0` udává neomezený počet výsledků). Ve filtrech fungují porovnávací i logické operátory a jejich kompletní seznam je uveden v oficiální dokumentaci[27]. Existuje i možnost získat sumaci výsledků a to přídáním parametru `$sum` na konec URL [2].
Pro ukládání nových dat do FlexiBee musí být použita HTTP metoda PUT. Tak FlexiBee pozná, že mají být data v systému uložena nebo aktualizována. Aktualizace dat se provede automaticky, pokud již v evidenci existuje položka s ID, které je uvedeno v odeslaných datech pro uložení. Pokud ID v odeslaných datech vůbec nebude, bude vytvořen nový záznam [2].

[26] `https://demo.flexibee.eu/c/demo_sk/cenik/((id = 42) and (nazev like 'test'))?sort=id&dir=desc&limit=0`

[27] `https://www.flexibee.eu/api/dokumentace/ref/format-types`

4 Návrh komponenty pro připojení na účetní systémy

V této kapitole bude popsán návrh celé aplikace. Hlavním cílem je vytvoření knihovny, která bude dostupná přes composer a bude jednoduše ovladatelná a použitelná. Dále by měla odstínit programátory využívající tuto knihovnu, od práce s daty v surovém formátu, který bude odesílán a příjímán z účetního systému, tedy XML či JSON. Zároveň je součástí knihovny rozšíření pro Nette framework, pro snadnou integraci.

Dalším požadavkem bylo, aby měla aplikace co nejmíň závislostí na dalších knihovnách a aby byla pokryta unit testy. Proto bude tato komponenta napsána v čistém PHP vyjma rozšíření pro Nette framework a testů.

4.1 Funkční požadavky

Funkční požadavky na vytvoření komponenty jsou následující:

- Možnost získávání dat webovou aplikací z účetního systému
- Možnost odesílání dat webovou aplikací do účetního systému
- Výměna dat (získvání, odesílání) v rámci alespoň tří evidencí (faktury, objednávky atd.)
- Vytvoření rozšíření pro Nette framework
- Pokrytí unit testy
- Dostupnost přes composer

4.2 Mimofunkční požadavky

Mimofunkční požadavky na vytvoření komponenty jsou následující:

- Podpora verze PHP >= 5.6.0

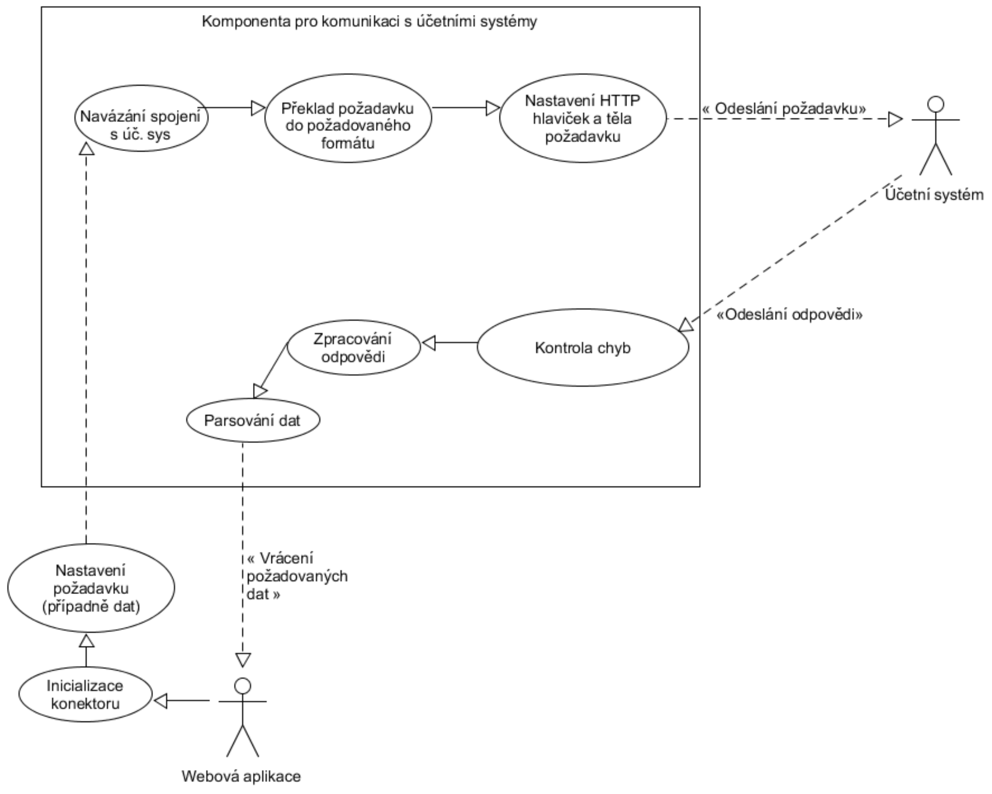
- Podpora alespoň dvou účetních systémů
- Snadné propojení webové aplikace naprogramované v PHP a účetního systému
- Minimum závislostí na externích knihovnách

Grafické uživatelské rozhraní v rámci této práce nebylo řešeno, jelikož se jedná čistě o výměnu dat mezi dvěma systémy. Prezentace dat potom závisí na každém z nich. Stejně tak bezpečnost. Ta závisí na individuálním nastavení jednotlivých systémů, komponenta slouží pouze jako komunikační kanál mezi nimi.

4.3 Obecný návrh

Při návrhu bylo postupováno od nejzákladnějšího aspektu knihovny a to samotné připojení na účetní systémy. Dále bylo nutné zjistit jak vypadají požadavky a odesílání dat na API účetních systémů a jak s daty pracovat, aby byl budoucí uživatel této knihovny odstíněn od práce se surovými daty. Po dokončení požadované funkčnosti musí být kód pokryt unit testy, aby bylo možné odhalit chyby zanesené do kódu. Jako poslední část komponenty bylo vytvořeno rozšíření pro Nette framework.

Obrázek 4.1 zachycuje obecnou představu o funkčnosti a komunikaci systémů. Komunikace bude inicializována vždy na straně webové aplikace, účetní systém sám nikdy nemůže bez podnětu odesílat data do externích systémů. V rámci komponenty je navázáno spojení, provedeno přeložení nastaveného požadavku do systémem podporovaného formátu a jeho odeslání do účetního systému. Účetní systém požadavek zpracuje a vrátí odpověď, která bude opět zpracována komponentou. V první řadě budou zkontrolovány chyby, které mohli nastat na úrovni HTTP požadavku či ve formátu požadavku. Odpověď bude zpracována do čitelného formátu a vrácena do webové aplikace.



Obrázek 4.1: Obecný návrh konektoru

4.4 Připojení na účetní systémy

V analýze bylo zjištěno, že oba účetní systémy využívají k připojení externích aplikací HTTP protokol, bude tedy nutné vytvořit prvek (třídu), který bude zmíněné připojení zajišťovat. Další funkčností, která je žádoucí, je schopnost odesílat požadavky a uchovávat patřičné odpovědi. Nezbytnou součástí programu je zachytit jakoukoli nastalou chybu při komunikaci a uvědomit o této chybě uživatele. Chyba může nastat na úrovni HTTP protokolu (např. nedostupný server atd.) i v rámci formátu dat požadavku (např. neplatný sloupec pro filtrování).

Účetní systémy mají drobné rozdíly co se týče HTTP hlaviček, například parametr pro autentizaci, konkrétně jeho šifrování, formát dat. Oba systémy mají rozdílné podoby požadavků. Proto je vhodné každý konektor umístit do vlastní třídy a společné prvky sjednotit v rodičovské třídě. Požadavky budou vytvářeny pomocí návrhového vzoru tovární metoda.

Využití tovární metody odstíní progamátora od nutnosti vytvářet objekty pomocí klíčového slova `new` z tříd, jež reprezentují požadavky.

Požadavky mají v konstruktoru určité parametry (ne všechny), které nejsou pro programátora důležité, případně je zde možnost je později upravit pomocí settru pokud to bude nezbytné. Atributy požadavku, důležité pro jeho chování, budou předávány přes parametry tovární metody.

Mezi společné prvky obou systémů můžeme zařadit například to, že se k nim vždy přistupuje přes doménovou či IP adresu. Také mají oba systémy právě jeden identifikační údaj, podle kterého lze jednoznačně identifikovat k jaké účetní jednotce se přistupuje. Stejně tak mají oba systémy uživatelské jméno a heslo, pod kterým se uživatel do systému přihlašuje.

4.5 Formát požadavků

Formáty požadavků se pro každý systém liší. V případě systému POHODA to je přístup na URL, která je vždy ve stejném formátu a komunikace přes HTTP probíhá vždy pomocí metody POST. POHODA navíc poskytuje možnost komunikace pouze pomocí XML, se kterým práce v PHP není ideální.

URL pro FlexiBee je, oproti POHODA systému, velmi proměnlivá. Pro získání dat se využívá metody GET a pro odesílání dat do systému metody PUT. Nabízí práci s daty v mnoho formátech, pro PHP je nejpřívětivější formát JSON.

Požadavky do obou systémů je možné rozdělit na dvě skupiny a to na:

- Získávání dat
- Odesílání dat

Skupiny jsou velmi rozdílné jak v rámci jednotlivých systémů, tak i mezi oběma systémy. Požadavky pro získávání dat jsou z principu jednodušší, jelikož neobsahují data, která se do systému odesílají, pouze informaci o evidenci, ze které mají být data získána a případně filtraci. Požadavky pro odesílání dat musí obsahovat data, která mají být v systému uložena. Aby bylo možné odstínit ostatní programátory od práce s těmito daty v formátu XML či JSON, budou vytvořeny entity a kolekce, které budou reprezentovat jednotlivé položky v evidencích. Programátor pouze naplní tuto entitu či kolekci a předá ji do parametru tovární metody příslušného požadavku. Pro každý požadavek bude vytvořena samostatná třída, která jej bude reprezentovat.

4.5.1 Systém POHODA

Požadavky v systému POHODA, jak už bylo zmíněno, jsou ve formátu XML. Každý z nich musí obsahovat IČO firmy a libovolné identifikační číslo. Může obsahovat libovolnou poznámku, pro informační účely bude poznámka obsahovat název třídy, která požadavek vytvořila. Požadavky také musí obsahovat deklaraci jmenných prostorů definovaných společností POHODA. Na základě těchto informací bude vytvořena základní abstraktní třída, ze které budou ostatní vycházet.

Přidávání filtrů je možné pouze u požadavků, které získávají data ze systému. Formát základního filtru je u všech takových požadavků stejný. Proto bude vytvořena další abstraktní třída, která bude obsahovat vytváření filtru a požadavky získávající data budou vycházet z ní.

Požadavky pro odesílání budou sestavovat XML data na základě výše zmíněných entit a kolekcí.

4.5.2 Systém FlexiBee

Požadavky ve FlexiBee budou odesílány a přijímány ve formátu JSON. Získávání dat je v případě tohoto systému jednoušší, jelikož není nutné sestavovat žádná data do požadovaného formátu. Evidence či filtry jež je požadovány jsou zadávány přímo v URL. Filtry mohou být přidávány též v těle HTTP požadavku, ale tato metoda nepodporuje logické a porovnávací operátory, proto bude zvolena možnost filtrování přes URL.

Podobně jako u systému POHODA se budou požadavky pro odesílání dat sestavovat na základě entit a kolekcí.

4.6 Návrh implementace unit testů

Úkolem unit testů je kontrolovat, zda se požadavek správně odešle a bude vrácena správná odpověď. Mělo by být kontrolováno správné vytvoření konektoru, dále jestli je konektor schopný odeslat požadavek. Pokud ano, zkontrolovat odpověď, zda při ní nenastala chyba a případně zkontrolovat data, vůči vzorovým, nebo strukturu dat.

5 Implementace a publikace komponenty

Tato kapitola se zabývá postupnou implementací komponenty. Komponenta je verzována pomocí verzovacího nástroje Git, konkrétně postkytovatelem GitHub[28] a je veřejně dostupná na adrese `https://github.com/soukupm2/acc-sync`. Součástí repozitáře je i uživatelská příručka, vytvořena v rámci komponenty. Finálním krokem je publikace komponenty pro její dostupnost přes composer, což je popsáno v poslední části této kapitoly.

Prvním krokem při začátku vývoje projektu byla inicializace zmiňovaného Git repozitáře a balíčkovacího nástroje composer. Byl proveden průzkum jak v rámci PHP komunikovat skrze HTTP protokol. Na základě průzkumu byly vytvořeny konektory na oba účetní systémy. Každý z konektorů je veden ve vlasní adresářové struktuře pro větší přehlednost. V rámci adresářové struktury jsou uloženy všechny související prvky konektoru, jako jsou požadavky, entity, kolekce a další pomocné třídy pro práci s daty. Mezi finální část implementace patří vytvoření testů, to bude ovšem popsáno v kapitole 6.

5.1 Inicializace projektu

Repozitář byl vytvořen u poskytovatele GitHub skrze webové rozhraní. V rámci vytváření repozitáře byl vytvořen i počáteční commit se souborem `README.md`, aby jej bylo možné ihned klonovat.

V adresáři s naklonovaným repozitářem byl vytvořen soubor `composer.json` na stejné úrovní jako `README.md`[29].

Je zde definováno (kód 5.1) jméno balíčku (`name`), popis balíčku (`description`), typ projektu (`type`), licence, autoři atd. Důležitým parametrem je `require`. Ten obsahuje seznam závislostí, které jsou nutné k chodu komponenty. Počáteční fáze vývoje projektu vyžadovala pouze jednu závislost a to na PHP verze 5.6.0 či vyšší. V pokročilejší fázi byli přidány závislosti na knihovnu, díky níž lze vytvořit rozšíření pro Nette framework a na knihovnu, která umožňuje vytváření a spouštění unit testů. Závislosti na těchto knihovnách jsou uchovány v atributu `require-dev`, ve kterém se

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

[29] README.md - soubor, který slouží k popisu repozitáře

definují balíčky pouze pro vývojové účely. Závislot na Nette (`"nette/di": "2.4"`) je v této sekci, protože ne vždy musí být knihovna integrována do Nette a proto není potřeba instalovat všechny balíčky. V případě integrace do Nette je tato knihovna jeho součástí.

```
{
  "name": "soukupm/acc-sync",
  "description": "Package for connecting accounting
     system with ehop.",
  "type": "library",
  "homepage": "https://github.com/soukupm2/acc-sync",
  "license": "MIT",
  "authors": [
    {
      "name": "Miroslav Soukup",
      "email": "miroslav.soukup2@gmail.com"
    }
  ],
  "minimum-stability": "dev",
  "require": {
    "php": ">=5.6.0"
  }
  "require-dev": {
      "phpunit/phpunit": "5.7",
      "nette/di": "2.4"
  }
}
```

Kód 5.1: Obsah souboru composer.json

Následovalo vytvoření přibližné adresářové struktury, která se v průběhu vývoje nepatrně měnila. Její výsledná podoba je uvedena v příloze B Struktura komponenty.

5.2 Konektory na účetní systémy

Nejdůležitější částí projektu je napojení komponenty na účetní systémy. Prvním krokem implementace tedy bylo vytvořit toto napojení.

V rámci průzkumu bylo zjištěno, že ke komunikaci pomocí HTTP protokolu bude nutné využít buď knihovnu `cURL`[30] či zvolit dostupnou nadstavbu nad ní (například Guzzle[31]). Mimonfunkční požadavky udávají co nejmenší množství závislostí na externích knihovnách. Proto byla zvolena knihovna `cURL`[30], která je v PHP nativně podporována.

Jak bylo uvedeno v návrhu (kapitola 4.4), způsob odesílání požadavků má určité společné prvky např. určitou URL a port, na kterou budou požadavky posílány, identifikátor společnosti, přihlašovací údaje. Proto byla vytvořena abstraktní základní třída `Connector.php`, ze které budou

[30] `https://www.php.net/manual/en/book.curl.php`

[31] `https://github.com/guzzle/guzzle`

oba konkrétní konektory vycházet. Obsahuje již zmiňované společné atributy. Součástí třídy jsou i protected atributy, které uchovávají nastavené připojení (curl session), aktuální požadavek a odpověď na něj (v případě, že byl odeslán) a informace o nastalých chybách.

`Connector.php` provádí inicializaci připojení (`curlu`) a to přímo v konstruktoru, proto je důležité v odděděných třídách zavolat rodičovský konstruktor. Při inicializaci připojení jsou nastaveny HTTP atributy v hlavičce, které jsou pro obě připojení stejné. Při zániku objektu, vytvořeného z této třídy je připojení uzavřeno. Konkrétními implementacemi jsou třídy `PohodaConnector.php` a `FlexiBeeConnector.php`.

Každá z těchto tříd rozšiřuje inicializaci připojení o nastavení HTTP atributů v hlavičce, které jsou pro daný účetní systém specifické. Tím je například podoba autorizačních údajů. Pro systém POHODA i FlexiBee se skládají z uživatelského jména a hesla odděleného dvojtečkou. Rozdíl je ten, že POHODA vyžaduje zakódování tohoto řetězce do Base64 a odeslání ve speciálním atributu hlavičky `STW-Authorization`, kdežto FlexiBee využívá nativního HTTP ověřování. Dalším rozdílem je hlavička udávající formát dat, pro POHODU je to XML, pro FlexiBee JSON.

Obě třídy obsahují metody, které kontrolují chyby, které mohly nastat na úrovni HTTP protokolu nebo přímo ve formátu požadavku (např. neplatný sloupec podle kterého je filtrováno). Metoda pro kontrolu chyb na úrovni HTTP je zavolána ihned po odeslání požadavku a pokud chyba nastala, je pro každý konektor definována vlastní výjimka, která je v případě chyby vyhozena. POHODA systém má na úrovni HTTP požadavků definované vlastní chybové hlášky přiřazené k HTTP kódům přijatých v odpovědi. Překlad HTTP kódu na vlastní zprávy je realizován pomocí enumu[32] `EResponseErrorCodes.php`. FlexiBee operuje s defaultně nastavenými zprávami.

Metoda pro získání chyb, které nastaly přímo v datech požadavku (jak již bylo uvedeno, např. neplatný sloupec pro filtraci), je volána až po získání odpovědi, protože chyba je uložena právě tam. Pro získání chyby z odpovědi je vytvořena třída `ErrorParser.php`, ve které jsou definovány statické metody pro oba systémy. Pokud chyba nastala, není jako v předchozím případě vyhozena výjimka, ale chyba je uložena do atributů třídy (jestli chyba nastala či ne a zpráva chyby). Na programátorovi, který využívá komponentu, leží zodpovědnost, aby zkontroloval stav atributů (přístupné jsou pomocí gettrů) a učinil

[32]Enum - výčtový datový typ

patřičná opatření.

Třídy obsahují metody pro nastavení požadavků. Metody jsou založeny na návrhovém vzoru tovární metoda (kapitola 2.3.1) a každý požadavek má metodu vlastní (některé metody mají parametr nezbytný pro iniciální nastavení požadavku). Programátora to odstíní od využívání klíčového slova `new` při vytváření požadavků. Ale i to je možné skrze speciální metodu, která přijímá jako parametr jakýkoliv požadavek pro daný systém. Požadavek je následně uložen do atributu třídy a lze s ním dále pracovat. Požadavky vyžadují nastavení libovolného ID a identifikátoru firmy, kterým je IČO. IČO je předáno v konstruktoru při vytváření konektoru a může být do požadavku rovnou zadáno. ID je generováno v této metodě a první odesílaný požadavek bude mít číslo jedna, u každeho dalšího požadavku se bude toto číslo inkrementovat o jedna. ID požadavku je možné ručně změnit.

Po nastavení požadavku už zbývá jen jediné a to je odeslání požadavku do účetního systému. Existuje pro to jedna metoda, v rámci které jsou provedeny následující úkony.

Pro oba konnektory je vytvořena továrna - `PohodaConnectionFactory` a `FlexiBeeConnectionFactory`. Továrny přijímají v konstruktoru stejné parametry jako konektory, tedy základní URL, uživatelské jméno a heslo, identifikátor firmy a volitelným parametrem je číslo portu. V rámci továren je prováděna validace, zda parametry nejsou prázdné, pokud ano je vyhozena výjimka. Obsahují jednu metodu a to `create()`, která vytvoří instanci konektoru až ve chvíli, kdy s ním potřebujeme pracovat.

- Kontrola, zda byl požadavek inicializován
- Ve FlexiBee konektoru složení URL (POHODA systém přijímá požadavky vždy na stejné URL)
- Odeslání požadavku
- Kontrola chyb na úrovni HTTP
- Parsování odpovědi
- Nastavení errorů
- Vrácení parsovaných dat

Kontrola inicializace požadavku zajistí, že požadavek je opravdu nastaven, pokud ne, bude vyhozena výjimka. Pro získávání dat u FlexiBee je nutné složit URL, tak aby byl jasný jeho význam. Odeslání požadavku

se provádí pomocí `curlu`[33], který byl inicializován při vytvoření instance konektoru. V rámci odeslání jsou nastaveny dodatečné atributy v hlavičce HTTP požadavku a proběhne nastavení těla (data, která mají být odeslána), pokud to je pro požadavek nutné. Proběhne kontrola chyb na úrovni HTTP, popsaná výše. Pokud vše proběhlo bez problémů, budou data rozparsována na objekt třídy `\stdClass`[34]. Proběhne kontrola zda v parsované odpovědi nejsou uvedeny chyby, v případě že ano, jsou nastaveny do atributů. Poté už jsou jen výsledná data vrácena.

5.3 Parsování odpovědí

Parsování dat z odpovědi je v podstatě nezbytné, aby byla práce se získanými daty jednoduchá a v rámci možností přívětivá. Systém POHODA komunikuje pouze pomocí XML a to pro práci v PHP není ideální. Proto byla vytvořena třída `XMLParser.php`, která se stará o parsování XML ze systému POHODA. FlexiBee umožňuje komunikaci ve více formátech, jak již bylo zmíněno a byl zvolen formát JSON. Data jsou parsována na objekt třídy `\stdClass`[34].

Při parsování dat přijatých z POHODA systému z XML do formátu přívětivějšího pro PHP nastal problém, když v XML byly využívány namespaces. Klasické XML na pole, či objekt třídy `\stdClass` lze převést dvěma příkazy (respektive zavolání dvou funkcí `json_decode (json_encode($xml))`). S použitím namespaces je nutné zjistit jaké namespaces jsou v XML použity, XML převést na string a veškeré namespaces odstranit. Po odstranění musí být string převeden zpět na XML. Aplikováním zmíněných funkcí pro klasické XML získáme objekt třídy `\stdClass`. O to se v třídě `XMLParser` stará statická metoda `parseXML`.

Parsování dat z FlexiBee bylo poměrně jednoduché. JSON stačí předat funkci `json_decode ($json)`, která vrátí data v požadovaném formátu, tedy instance třídy \stdClass.

Parsování dat z POHODA systému je možné posunout ještě dál. Struktura dat je předem dána v připravených schématech[35]. Na základě těchto schémat byly vytvořeny entity a kolekce[36], které svými atributy kopírují zmiňovaná schémata. Ovšem nejsou to přesné kopie

[33]Curl - slouří pro komunikaci přes HTTP protokol
[34]\stdClass - obecný objekt v PHP
[35]`https://www.stormware.cz/pohoda/xml/dokladyimport/`
[36]Kolekce - datová struktura

pro zjednodušení struktury. Entity a kolekce jsou využívány i k účelu odesílání dat do POHODA systému.

Byly implementovány 3 požadavky pro získání dat, faktury, objednávky a skladové zásoby. Pro každý z těchto požadavků existuje vlastní parser - InvoiceParser, OrderParser a StockParser. Každý z nich obahuje statickou metodu `parse(\stdClass $data)`, která v parametru přijímá výsledek metody pro odeslání požadavku. Data jsou postupně procházena a mapována do entit a kolekcí. Využití namapování pomocí těchto parserů je volitelné, ale poskytuje výhody při další práci se získanými daty jako například napovídání get metod atributů. Objekt třídy `\stdClass` takovou možnost přístupu k atributům nenabízí a je nutné znát strukturu výsledného XML.

FlexiBee tuto možnost nenabízí. Struktura dat v odpovědích je značně proměnná a proto je problematické vytvořit odpovídající entity a kolekce. Data z FlexiBee jsou tedy uloženy vždy v objektu třídy `\stdClass`.

5.4 Požadavky pro získávání dat z POHODA systému

Základní třída požadavku - `BaseRequest.php`, byla implementována jako abstraktní. Konstruktor vyžaduje dva parametry, které musí být v každém požadavku obsaženy a to je libovolné číslo (ID) pro identifikaci požadavku a IČO firmy, podle kterého bude v systému POHODA identifikována účetní jednotka. Konstruktor také obsahuje volání protected[37] metody pro složení základní podoby požadavku, metoda je abstraktní a musí být definována v každém potomkovi, protože tělo XML požadavku se vždy liší. Třída poskytuje možnost vložení vlastního XML, pokud by náhodou definice požadavků nevyhovovala všem potřebám.

Třída obsahuje metody pro přenastavení ID a identifikátoru firmy. Dále pak metodu, která sestaví XML hlavičku, která je pro každý požadavek stejná.

Z této třídy vychází základní třída pro získávání dat `BaseGetDataRequest.php`, která obsahuje navíc pouze metodu pro přidávání filtrů, které jsou u odesílání dat zbytečné. Ta přijímá dva parametry, prvním je název atributu, podle kterého bude filtrace provedena a druhým je hodnota filtru. Filtrování je možné u všech

[37]Protected - přístup k protected metodě či atributu má pouze třída, ve které je definovaná a její potomci

požadavků, ovšem nelze filtrovat podle všech atributů. Příklady filtrace jsou uvedeny na oficiálních stránkách[38].

Konkrétně byly implementovány požadavky pro získání faktur, objednávek a skladových zásob. Vycházejí ze základní třídy pro získání dat. Tomu odpovídají třídy `ListInvoiceRequest.php` pro faktury, `ListOrderRequest.php` pro objednávky a `ListStockRequest.php` pro skladové zásoby. Požadavky pro získávání faktur a objednávek mají navíc o proti základnímu filtru navíc možnost filtrovat podle zadaného rozmezí v čase, podle jména společnosti, pro kterou byla zaevidována faktura či objednávka a poslední nahrazuje jméno z předchozího případu za IČO. Metody pro tyto filtry navíc byly přidány, protože v XML mají jinou strukturu než filtr základní. Navíc mají též v konstukoru parametr, který udává o jaký typ objednávky/faktury se jedná.

Požadavek pro získání skladových zásob také obsahuje několik filtrů navíc, které mají nestandardní formát a museli být implementovány zvlášť. Jedná se o filtraci podle poslední provedené změny ve skladových zásobách, podle ID či názvu skladu a podle ID divize, do které sklad patří.

Příklad požadavku pro získání skladových zásob vypadá následovně:

```
POST http://localhost:1111/xml HTTP/1.1
User-Agent: 123456789
STW-Authorization: Basic QDo=
Content-Type: text/xml
Host: http://localhost:1111
Content-Length: 648

<?xml version="1.0" encoding="Windows-1250"?>
<dat:dataPack
    <!-- definice schemat -->
    id="1" ico="25313142"
    application="HTTP klient"
    version="2.0"
    note="ListStockRequest" >
  <dat:dataPackItem id="1" version="2.0">
    <lStk:listStockRequest
        version="2.0"
        stockVersion="2.0">
      <lStk:requestStock/>
    </lStk:listStockRequest>
  </dat:dataPackItem>
</dat:dataPack>
```

Kód 5.2: Ukázkový požadavek POHODA

Na příkladu výše je možné vidět, že v hlavičce HTTP požadavku jsou uvedeny všechny povinné atributy uvedené v základních pravidlech, viz kapitola 3.1. Hlavička i tělo požadavku jsou vytvořeny konektorem, ale podoba XML v těle vytváří třídy požadavků. První řádek udává metodu, kterou bude požadavek odeslán,URL, na kterou bude požadavek odeslán

[38]`https://www.stormware.cz/pohoda/xml/dokladyexport/`

a verzi HTTP protokolu. V druhém řádku je uveden atribut `User-agent`, který identifikuke odesílatele požadavku, v tomto případě je zde vyplněno IČO firmy.

`STW-Authorization` je atribut, který, jak již bylo uvedeno, uchovává zakódované přihlašovací údaje. `Content-Type` udává formát obsahu těla požadavku, v tomto případě je to XML. Následuje informace o doménovém jménu serveru, atribut `Host`. Jako poslední je uveden `Content-Length` udávající délku těla požadavku.

Poté následuje prázdná řádka a samotné tělo požadavku s obsahem XML dat. Důležitým elementem v XML je `lStk:listStockRequest`, který specifikuje evidenci, ze které mají být získána data. Elementy `dat:dataPack` a `dat:dataPackItem` jsou pevně definovány systémem POHODA a udávají, že se jedná o dotaz na data (nebo jejich odeslání do POHODA systému). Atributy `id, ico, version, note` elementu `dat:dataPack` jsou vyplněny na základě typu požadavku a jeho atributů.

Odpověď programu POHODA na uvedený požadavek vypadá zhruba takto:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=Windows-1250
Content-Length: 1151
<?xml version="1.0" encoding="Windows-1250"?>
<rsp:responsePack version="2.0" id="Z001"
    state="ok" programVersion="12002.7 (26.10.2018)"
    ico="12345678" note="export zasob"
    <!-- definice schemat --> >
  <rsp:responsePackItem version="2.0"
      id="0001" state="ok">
    <lStk:listStock version="2.0"
        dateTimeStamp="2019-05-09T15:23:33"
        dateValidFrom="2019-05-09"
        state="ok">
      <lStk:stock version="2.0">
        <stk:stockHeader>
          <stk:id>1</stk:id>
            ⋮
        </stk:stockHeader>
        <stk:stockPriceItem>
          <stk:stockPrice>
            <typ:id>1</typ:id>
              ⋮
          </stk:stockPrice>
        </stk:stockPriceItem>
      </lStk:stock>
    </lStk:listStock>
  </rsp:responsePackItem>
</rsp:responsePack>
```

Kód 5.3: Ukázková odpověď POHODA

První řádek odpovědi udává verzi HTTP protokolu, následuje kód stavu požadavku a zpráva. Následuje opět formát obsahu a jeho kódování.

Posledním atributem je délka odpovědi.

Elementy `rsp:responsePack` a `rsp:responsePackItem` jsou pevně definovány systémem POHODA a udávájí, že se jedná o odpověď ze systému. Element `lStk:listStock` udává že se jedná o výpis skladových zásob. Každá skladová položka je uložena v elementu `lStk:stock`. Jeho obsah udává atributy položky na skladě.

5.5 Požadavky pro odesílání dat do POHODA systému

Výchozím prvkem pro tyto požadavky je třída `BaseRequest.php`, popsaná v kapitole 5.4. Implementovány byly, stejně jako u získávání dat, požadavky pro odesílání faktur, objednávek a skladových zásob. Představují je třídy `SendInvoiceRequest.php` pro faktury, `SendOrderRequest.php` pro objednávky a `SendStockRequest.php`.

Úkolem těchto tříd je vzít data, která ji budou předána přes konstruktor a poskládat je do validního XML, které bude odesláno do systému POHODA. Data jsou předávána ve formátu kolekce. Jsou využity stejné kolekce a entity jako při optionálním parsování odpovědí (kapitola 5.3). Kolekce a entity musí být naplněny daty. Předaná data se v třídách postupně prochází a na základě vyplněných atributů je skládáno XML. Atributů, které mohou být vyplněny v kolekcích a entitách je velké množství a proto je tato část kódu velmi rozsáhlá, navíc zde probíhá validace jestli jsou data vyplněna a pokud ano, jsou zapsána do XML.

Požadavek pro odeslání dat má stejné hlavičky jako v kódu 5.2. Tělo pak vypadá podobně jako v kódu 5.3 s tím rozdílem, že elementy `rsp:responsePack` a `rsp:responsePackItem` budou přejmenovány na `dat:dataPack` a `dat:dataPackItem`.

5.6 Požadavky pro získávání dat z FlexiBee systému

Stejně jako u systému POHODA i zde je vytvořena základní abstraktní třída `BaseRequest.php`, ze které ostatní požadavky vychází. V případě FlexiBee obsahuje pouze jeden atribut, který udává o jakou se jedná evidenci a jeho getter.

Z této třídy vychází základní třída pro požadavek na získávání dat BaseGetDataRequest.php. Ten obsahuje několik metod. První metodou je nastavení filtru v URL.

Pro vytvoření filtru (podmínka pro získání dat) existuje entita FlexiBeeCondition a helper obsahující statickou metodu FlexiBeeHelper::joinConditions, který slouží ke spojování podmínek, pokud by jich bylo více. Entitě je třeba v konstruktoru předat sloupec, podle kterého se bude filtrovat, operátor a hodnotu. Při spojování podmínek využijeme helperu, ten přijímá jako první parametr operátor a jako, druhý, třetí, čtvrtý atd. jsou podmínky, které mají být spojeny. Takto vytvořený filtr se předá zmiňované metodě na jeho nastavení.

Další metodou je nastavení, zda má být výsledek sumován. Přijímá nepovinný parametr typu boolean, implicitně je nastaveno TRUE. Dalšími dvěmi metodami může být nastaveno řazení, jako parametr přijíma sloupec, podle kterého bude výsledek řazen a směr a limit počtu výsledků, jako parametr je přijímáno číslo s údajem kolik záznamů má být získáno (0 = bez limitu) a druhý nepovinný parametr udává offset (odsazení) od prvního záznamu.

Konkrétními požadavky jsou získání ceníku, získání přijatých a vydaných faktur a získání přijatých a vydaných objednávek. Reprezentují je třídy ve stejném pořadí PriceListRequest, ReceivedInvoiceRequest, IssuedInvoiceRequest, ReceivedOrdersRequest, IssuedOrdersRequest. Tyto třídy obsahují pouze naplnění atributu ze třídy BaseRequest.php, aby bylo možné identifikovat evidenci. Poté se o vše starají metody ze třídy BaseGetDataRequest.php.

Příklad požadavku na získání záznamu z ceníku vypadá takto:

```
GET  https://demo.flexibee.eu/c/demo_sk/cenik/
    ((id = 42) and (nazev like 'test'))
    ?sort=id&dir=desc&limit=0
Accept: application/json
Authorization: Basic QDo=
Host: https://demo.flexibee.eu
```

Kód 5.4: Ukázkový požadavek FlexiBee

V příkladu je možné vidět URL, která byla poslána metodou GET na demo server systému FlexiBee. V parametru Accept je uvedeno, že požadujeme v odpovědi data ve formátu JSON. Dále jsou zde uvedeny autorizační údaje a doménové jméno serveru. Požadavek v příkladu vrátí odpověď ve formátu JSON (kód 5.5)

První řádek odpovědi udává verzi HTTP protokolu, následujé kód stavu požadavku a zpráva. Následuje opět formát obsahu. Posledním atributem je délka odpovědi.

Atribut `winstrom` je pevně definován systémem FlexiBee, `@version` udává verzi systému. Následuje požadovaný výsledek.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 216

{
  "winstrom": {
      "@version": "1.0",
      "cenik": [
          {
              "id": "42",
              "lastUpdate": "2019-05-02T13:58:51",
              "kod": "TESTING",
              "nazev": "testing",
              "cenaZakl": "100.0",
              "cenaZaklBezDph": "100.0",
              "cenaZaklVcDph": "120.0",
              "szbDph": "20.0"
          }
      ]
  }
}
```

Kód 5.5: Ukázková odpověď FlexiBee

5.7 Požadavky pro odesílání dat do FlexiBee systému

Jak již bylo řečeno, v požadavcích pro FlexiBee není využíváno entit ani kolekcí jako v případě POHODA systému. Požadavky jsou vytvořeny tak, aby měli základní atributy, odpovídající základním atributům ve FlexiBee s možností přidat jakýkoliv další atribut. Dalo by se říci, že třída požadavku je v podstatě entitou, kde jsou data nastavovány přímo jemu přes metody set. Data atributů třídy jsou při odeslání namapována do asociativního pole a poté převedena do formátu JSON.

Hlavička opět udává metodu HTTP požadavku, která je v případě odesílání dat do FlexiBee `PUT`. Následují stejné atributy jako u požadavku pro získání dat a v těle se nachází odesílaná data ve formátu JSON.

```
PUT  https://demo.flexibee.eu/c/demo_sk/cenik.json
Accept: application/json
Authorization: Basic QDo=
Host: https://demo.flexibee.eu

{
  "winstrom": {
      "cenik": [
          {
              "id": "42",
              "kod": "TESTING",
              "nazev": "testing",
              "cenaZakl": "100.0",
```

```
                "cenaZaklBezDph": "100.0",
                "cenaZaklVcDph": "120.0"
            }
        ]
    }
}
```

Kód 5.6: Ukázkový požadavek pro odeslání dat do FlexiBee

5.8 Základní kolekce

Kolekce (příloha C) byla v rámci implementace zmíněna několikrát. Standartně není součástí PHP a proto zde byla implementována. Abstraktní třída `BaseCollection.php` představuje základní kolekci[39]. Kolekce je v podstatě pole, které slouží k ukládání objektů a výhodou je, že může být omezena na určitý typ těchto objektů. Zároveň s kolekcí není pracováno jako s polem, ale jako s objektem.

Kolekce implementuje tři rozhraní:

- \IteratorAggregate
- \ArrayAccess
- \Countable

Každé z těchto rozhraní definuje metody, které musí být v rámci kolekce implementovány. \IteratorAggregate definuje metodu `getIterator()`, která umožní iterovat přes instanci budoucího objektu vytvořeného z třídy kolekce. \ArrayAccess definuje metody pro přístup do kolekce, jako do pole, tedy získání prvku na určité pozici v poli, přidáni prvku na pozici do pole atd. Poslední rozhraní `\Countable` definuje jednu metodu, `count`, která vrací počet prvků v kolekci. Implementací těchto metod bude vytvořena funkční kolekce.

5.9 Rozšíření pro Nette framework

Jedním z požadavků bylo vytvořit rozšíření pro Nette framework, pro jednoduchou integraci. Rozšíření je k dispozici v příloze E Rozšíření pro Nette framework.

Rozšíření je třída rozšiřující `\Nette\DI\CompilerExtension`. To umožňuje možnost jeho registrace v konfiguračním souboru frameworku. Dále umožňuje načítání dodatečných parametrů.

[39]Kolekce - datová struktura

Parametry jsou ve třídě definovány jako konstanty a reprezentují hodnoty, které mají být předány továrně, která vytváří připojení k účetnímu systému. Ve třídě je definována metoda (`setUpParams()`), která zajistí načtení parametrů z konfiguračního souboru a zároveň validuje, že jsou vyplněny. Pokud ne bude vyhozena výjimka.

Metoda `loadConfiguration()` volá již zmiňovanou funkci na získání parametrů z konfiguračního souboru. Následuje nastavení továrny, která díky této metodě bude dostupná v dependency injection kontejneru Nette.

5.10 Publikace komponenty

Pro publikaci knihovny byl zvolen portál Packagist.org[40], který byl stručné popsán v kapitole 2.2.

Aby bylo možné komponentu na Packagist.org publikovat, je nutné zde provést registraci. Dále je nutné mít komponentu uloženou v online repozitáři (Git, Svn), který bude veřejně přístupný a komponenta musí mít vytvořený alespon jeden tag[41].

Po splnění nutných požadavků je možné pokračovat v procesu publikace. Registrací se na Packagist.org[42] otevře možnost přidat balíček. Rozkliknutím odkazu na přidání balíčku je zobrazen formulář s jedním textovým polem, do kterého je nutné vyplnit URL adresu požadovaného repozitáře. Odesláním formuláře s validní URL adresou repozitáře bude balíček přidán. Podoba balíčku na Packagist.org[43] je ukázána v příloze D Podoba balíčku v Packagist.org.

V levém horním rohu je název balíčku a hned pod ním příkaz pro jeho instalaci pomocí composeru a krátký popis. Název balíčku a popis jsou uloženy v projektu v souboru `composer.json`. Následují tlačítka, která jsou vidět pouze pro uživatele, který má přidělena práva pro správu balíčku a jsou to:

- **Abandon** - slouží pro opuštění balíčku přihlášeným uživatelem, který již nechce být správce
- **Delete** - smazání balíčku
- **Update** - zkontroluje, zda v registrovaném repozitáři proběhli změny, například přidání nového tagu

[40] `https://packagist.org/`
[41] Git/Svn tag - označení důležitého bodu v repozitáři verzí
[42] `https://packagist.org/packages/submit`
[43] `https://packagist.org/packages/soukupm/acc-sync`

- **Edit** - umožňuje změnit URL adresu repozitáře

Levá dolní část ukazuje závislosti balíčku, licenci, pod kterou je balíček vyvíjen a autora. V pravé horní části jsou statistické informace, odkaz na repozitář atd. Dolní část potom ukazuje všechny tagy v repozitáři.

Zmiňované tlačítko `update` lze u některých poskytovatelů repozitářů nahradit automatickou aktualizací. V případě této knihovny je to poskytovatel GitHub. GitHub umožňuje vytvoření tzv. `Webhooks` pro každý repozitář. Pomocí `Webhooks` můžeme odesílat informaci pro Packagist.org o vytvoření nového tagu v repozitáři. Při obdržení této informace proběhne automatická aktualizace balíčku.

Posledním požadavkem je přidání záznamu do souboru `composer.json`. Tím bude zajištěno, že po instalaci bude knihovna automaticky načtena. `AccSync` je pouze identifikátor a označuje cestu k souborům, které mají být načteny. Podoba záznamu je následující:

```
"autoload": {
    "psr-0": {
        "AccSync": "src/"
    }
}
```

Kód 5.7: Composer.json autoload knihovny

6 Testování

Po implementaci funkční části komponenty, bylo nutné tuto část řádně otestovat. V rámci této kapitoly jsou popsány postupy, které byly využity při testování komponenty.

6.1 Unit testy

Unit (jednotkové) testy jsou takové, kde jsou testovány individuální části kódu. Mohou to být metody či celé třídy a tak dále [18].

Pro vytvoření testů byla použita knihovna PHPUint[43] a to konkrértně verze PHPUint 5. Tato verze již není vývojáři podporována a to z důvodu ukončení podpory PHP 7.0 a nižších verzí. Knihovnu je i přes to stále možné využívat a její instalace se provádí pomocí composeru.

Oba systémy jsou testovány odděleně. Společné však mají testování vytvoření připojení pomocí zmiňovaných továren. Dále je u každého systému testování odeslání požadavku a po té validace odpovědi v rámci možností každého systému. Příklad vytvořeného testu pro kontrolu vytváření připojení pro systém POHODA (obdobně i pro FlexiBee) vypadá takto:

```
public function testCreateConnectionWithNoBaseUri()
{
    $this->expectException(
        \InvalidArgumentException::class
    );
    $this->expectExceptionMessage(
        'BaseUri cannot be empty'
    );
    new \AccSync\Pohoda\PohodaConnectionFactory(
        NULL,
        self::USERNAME,
        self::PASSWORD,
        self::COMPANY_ID,
        self::PORT);
}
```

Kód 6.1: Příklad unit testu na kontrolu vytvoření připojení

Prvním krokem v testu je definice očekávané výjimky a zprávy, kterou má výjimka mít. Následuje vytvoření továrny, která musí mít povinně neprázdný parametr udávající URL systému. Tento test je velmi jednoduchý a stejně jsou testovány i další povinné parametry.

[43] https://phpunit.de

6.1.1 Systém POHODA

V systému POHODA je testována podoba XML požadavků pro získávání dat vytvořených komponentou oproti nadefinovanému validnímu XML požadavku, který je uložen v souboru. Dále je testováno odeslání požadavku do systému a kontrola zda v odpovědi není uvedena chyba. Probíhá i testování odesílání dat do systému, tady probíhá jen kontrola zda při odesílání nastala chyba či ne.

Následuje ukázka testu (kód 6.2), který kontroluje strukturu vytvořeného požadavku na skladové zásoby oproti připravené validní šabloně. V první řadě je vytvořen požadavek s nadefinovaným filtrem. Poté je načtena šablona XML, se kterým má být požadavek porovnán. Následuje porovnání struktury vytvořeného XML proti uložené validní šabloně a kontrola hodnot, kterých XML elementy nabývají.

```
public function testListStockRequestCode()
{
    $request = new ListStockRequest('1', 12345678);
    $request->addFilter('internet', 'true');
    $dom = $this->loadTemplate(
        'PohodaXML/Stock/zasoby_01_v2.0.xml'
    );
    $domRequest = dom_import_simplexml(
        $request->getRequestXml()
    );
    $this->assertEqualXMLStructure(
        $dom->documentElement,
        $domRequest
    );
    $this->checkXmlValues(
        $dom->saveXML(),
        $request->getRequestXml()->saveXML()
    );
}
```

Kód 6.2: Příklad unit testu na kontrolu POHODA požadavku

6.1.2 Systém FlexiBee

V systému FlexiBee možnost kontroly výsledných dat oproti šabloně není. Odpovědi na požadavky pro získání dat jsou velmi proměnné. Proto je provedena kontrola pouze, že je vytvořen správný požadavek. Dále že při nastavení limitu či filtru je tento limit či filtr opravdu dodržen a nejhlavnějším aspektem je otestování, že při požadavku nenastala žádná chyba.

Ukázkový unit test (kód 6.3) ukazuje kontrolu požadavku pro odeslání dat. Stejně jako u POHODA systému nejdříve vytvoří připojení

k testovacímu serveru. Následuje nastavení požadavku pro odeslání dat (záznam v ceníku) a jsou mu definovány atributy, které mají být uloženy ve FlexiBee. Požadavek je odeslán a je provedena kontrola, že při zpracování nenastala žádná chyba.

```
public function testSendPriceListItem()
{
    $connection = $this->createProperConnection();

    $connection->sendPriceListItem()
        ->setId(42)
        ->setCode('testing')
        ->setName('testing')
        ->setBasePrice(100)
        ->setVatRate(20);

    $connection->sendRequest();

    $this->assertEquals(FALSE, $connection->hasError());
}
```

Kód 6.3: Příklad unit testu na kontrolu FlexiBee požadavku

6.2 Testování pomocí vzorové aplikace

Pro testování funkčnosti byla vytvořena jednoduchá vzorová aplikace v Nette frameworku. Je dostupná v repozitáři `https://github.com/soukupm2/sync_example`, kde jsou uvedeny i instrukce pro instalaci.

Komponenta byla do aplikace nainstalována přes composer. Následně bylo naprogramované rozšíření nastaveno v konfiguračním souboru frameworku. Poté už je možné komponentu začít plně využívat.

V rámci vzorové aplikace byly vyzkoušeny všechny požadavky v surové (JSON, XML) podobě, tzn. data byla prohlížena a pokud neodpovídala byla komponenta řádně upravena.

Posledním krokem bylo v rámci aplikace vytoření GUI[44] pro prezentaci a testování základní funkčnosti.

[44] GUI - Grafické uživatelské rozhraní

7 Závěr

V rámci diplomové práce byla vytvořena komponenta pro přenos dat mezi webovou aplikací (e-shopem) a dvěma účetními systémy - POHODA a FlexiBee. Základní požadavky patřilo, aby bylo možné komponentu využít v co nejširším spektru aplikací, které jsou naprogramovány v jazyce PHP. Proto byla komponenta implementována ve verzi PHP 5.6. Při vývoji byl kladen důraz na dodržování best practices, a aby byla práce s komponentou jednoduchá a intuitivní. To zajišťuje snadné propojení účetního systému a webové aplikace a značné usnadnění práce.

Komponenta zajišťuje výměnu dat inicializovanou webovou aplikací. Výměna dat je pro každý systém možná v rámci tří evidencí - faktury, objednávky a skladové zásoby pro systém POHODA, faktury, objednávky a ceník pro systém FlexiBee. Tato funkčnost například při provedení objednávky ve webové aplikaci umožní okamžité zanesení záznamu také do účetního systému.

Dalším požadavkem bylo, aby byla komponenta dostupná přes balíčkovací nástroj composer. Analýza v teoretické části poskytnula potřebné informace pro naplnění tohoto požadavku. Komponenta je verzována pomocí Git repozitáře, který je přímo napojený na server, díky kterému je dostupná přes zmiňovaný balíčkovací nástroj. Komponentu lze nainstalovat jediným příkazem.

Součástí komponenty je rozšíření pro Nette framework, které umožňuje její snadnou integraci právě pro pojekty využívající tento framework. Po instalaci stačí přidat požadovanou konfiguraci do konfiguračních souborů a komponentu je možné ihned použít.

Aby bylo možné ověřit správnou funkčnost komponenty, jsou v rámci komponenty implementované Unit testy. Ty kontrolují správnost vytváření konektorů na účetní systémy, správnost formátu požadavků a nebo zda požadavky vrací očekávanou odpověď. Díky testům bylo odhaleno značné množství chyb, které by bez jejich pomoci nebylo snadné vypátrat.

Nad rámec diplomové práce byla vytvořena jednoduchá webová aplikace, která má za úkol demonstrovat způsob využívání komponenty. Zároveň slouží jako prezentační vrstva pro komunikační kanál.

V budoucnu by mohla být komponenta rozšířena o další evidence či dokonce o připojení na další účetní systém.

Literatura a použité zdroje

[1] Böhmer, M. *Návrhové vzory v PHP*. Computer Press, 2012. ISBN 978-80-251-3338-5.

[2] flexibee.eu. *FlexiBee dokumentace* [online]. flexibee.eu. [cit. 2019/06/18]. Dostupné z: https://www.flexibee.eu/api/dokumentace/ref/index/.

[3] flexibee.eu. *FlexiBee - ke stažení* [online]. flexibee.eu. [cit. 2019/06/18]. Dostupné z: https://www.flexibee.eu/podpora/stazeni-flexibee/.

[4] flexibee.eu. *FlexiBee ceník* [online]. flexibee.eu. [cit. 2019/06/18]. Dostupné z: https://www.flexibee.eu/cenik/.

[5] github.com. *Composer release date* [online]. Jordi Boggiano, Nils Adermann, 2013. [cit. 2019/05/14]. Dostupné z: https://github.com/composer/composer/releases/tag/1.0.0-alpha1.

[6] nette.org. *Latte* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://latte.nette.org/cs/guide.

[7] nette.org. *Nette komponenty* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://doc.nette.org/cs/3.0/components.

[8] nette.org. *Nette database* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://doc.nette.org/cs/3.0/database.

[9] nette.org. *Nette database explorer* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://doc.nette.org/cs/3.0/database-explorer.

[10] nette.org. *Stažení Nette frameworku* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://nette.org/cs/download.

[11] nette.org. *Vlastnosti Nette frameworku* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://nette.org/cs/#toc-features.

[12] nette.org. *Nette MVC* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://doc.nette.org/cs/3.0/presenters.

[13] nette.org. *Nette NEON* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://doc.nette.org/cs/3.0/neon.

[14] nette.org. *Bezpečnost Nette* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: https://doc.nette.org/cs/3.0/vulnerability-protection#toc-cross-site-scripting-xss.

[15] NETTE.ORG. *Nette Tracy* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: `https://tracy.nette.org/cs/guide`.

[16] NETTE.ORG. *Kdo Nette používá?* [online]. nette.org. [cit. 2019/06/21]. Dostupné z: `https://builtwith.nette.org/`.

[17] SITEPOINT.COM. *Žebříček oblíbenosti frameworků* [online]. sitepoint.com. [cit. 2019/06/21]. Dostupné z: `https://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/`.

[18] SOFTWARETESTINGFUNDAMENTALS.COM. *Unit Testing* [online]. softwaretestingfundamentals.com. [cit. 2019/06/18]. Dostupné z: `http://softwaretestingfundamentals.com/unit-testing/`.

[19] SOURCEMAKING.COM. *Design Patterns* [online]. SourceMaking.com. [cit. 2019/05/21]. Dostupné z: `https://sourcemaking.com/design_patterns`.

[20] STORMWARE.CZ. *Vlastnosti systému POHODA* [online]. stormware.cz. [cit. 2019/06/18]. Dostupné z: `https://www.stormware.cz/pohoda/vlastnosti`.

[21] STORMWARE.CZ. *Nastavení POHODA mServeru* [online]. stormware.cz. [cit. 2019/06/18]. Dostupné z: `https://www.stormware.sk/xml/mServer/navod_Nasazení_HTTP_konektoru_pro_mServer.pdf`.

[22] STORMWARE.CZ. *Vlastnosti POHODA mServeru* [online]. stormware.cz. [cit. 2019/06/18]. Dostupné z: `https://www.stormware.cz/pohoda/xml/mserver/provyvojare`.

Seznam zkratek

AJAX - Asynchronous JavaScript and XML

API - Application Programming Interface

CSRF - Cross-Site Request Forgery

CSS - Cascading Style Sheets

ERP - Enterprise Resource Planning

GUI - Graphical User Interface

HTML - Hypertext Markup Language

HTTP - Hypertext Transfer Protocol

IČO - Identifikační číslo osoby

JSON - JavaScript Object Notation

MVC - Model-View-Controller

PHP - PHP: Hypertext Preprocessor

SQL - Structured Query Language

URL - Uniform Resource Locator

WISIWYG - What You See Is What You Get

XML - Extensible Markup Language

XSS - Cross-Site Scripting

Seznam obrázků

2.1 Nette - životní cyklus presenteru 12
2.2 Nette - Tracy 17
2.3 Nette - Tracy zobrazení informací 17
2.4 Nette - Tracy výpis 18
2.5 Nette - struktura sandboxu 19

3.1 IIS - převedení na aplikaci 30
3.2 POHODA - Nastavení webového serveru 31
3.3 POHODA - Spuštění mServeru 32
3.4 POHODA - Seznam mServerů 33
3.5 POHODA - Seznam požadavků 33
3.6 POHODA - Úprava či vytvoření mServeru 34

4.1 Obecný návrh konektoru 40

Seznam kódů

2.1 Ukázka XSS 10
2.2 Ukázka n:makra v Latte 14
2.3 Ukázkový záznam v NEON souboru 15
2.4 Záznam v NEON souboru pro databázi 15
2.5 Záznam v souboru composer.json 21
3.1 Ukázka lokálního nastavení mServeru 31
5.1 Obsah souboru composer.json 44
5.2 Ukázkový požadavek POHODA 49
5.3 Ukázková odpověď POHODA 50
5.4 Ukázkový požadavek FlexiBee 52
5.5 Ukázková odpověď FlexiBee 53
5.6 Ukázkový požadavek pro odeslání dat do FlexiBee 53
5.7 Composer.json autoload knihovny 56
6.1 Příklad unit testu na kontrolu vytvoření připojení 57
6.2 Příklad unit testu na kontrolu POHODA požadavku 58
6.3 Příklad unit testu na kontrolu FlexiBee požadavku 59

Přílohy

Seznam příloh na CD

Komponenta

- zdrojové soubory vytvořené komponenty

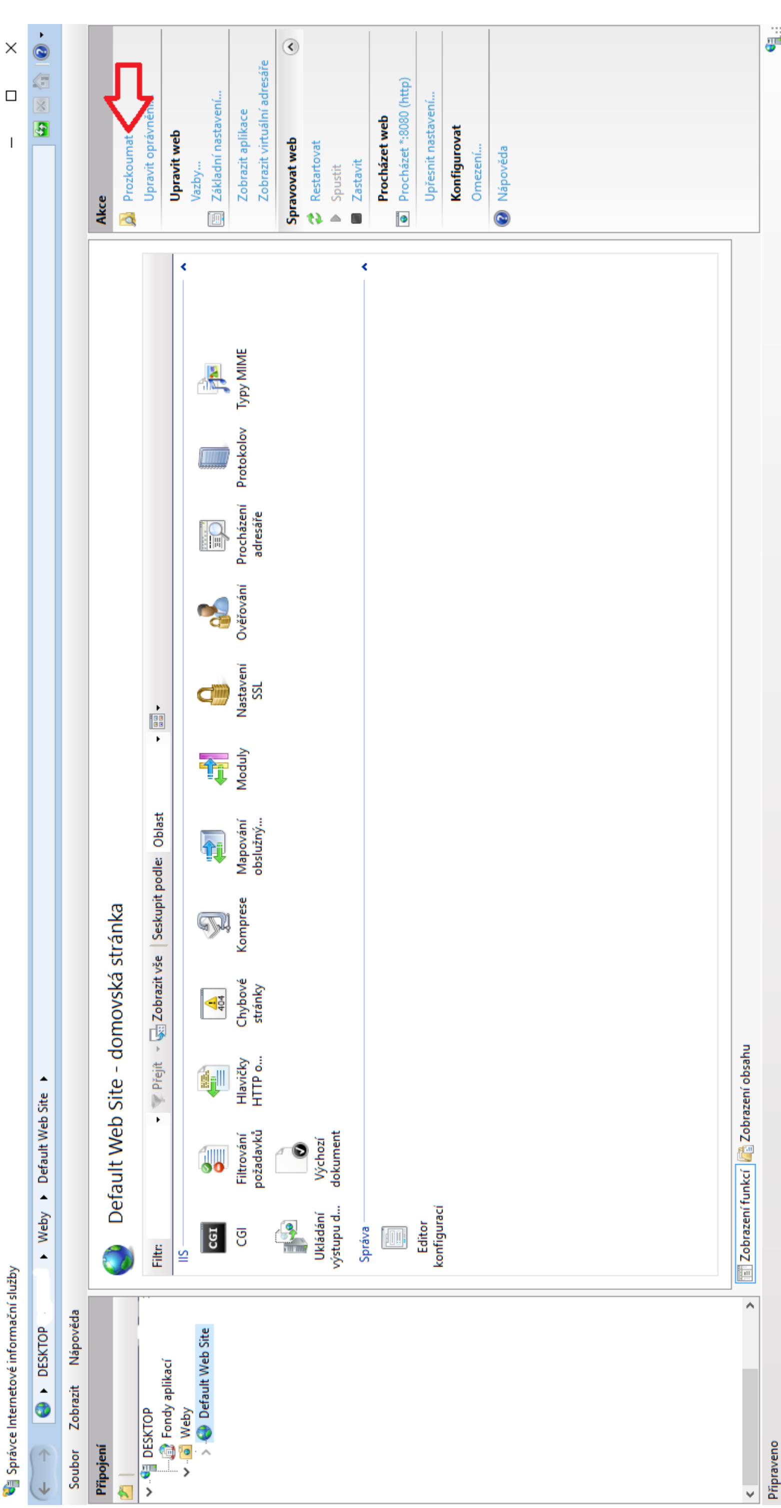
Text diplomové práce

- zdrojové soubory pro LaTeX a vlastní text práce

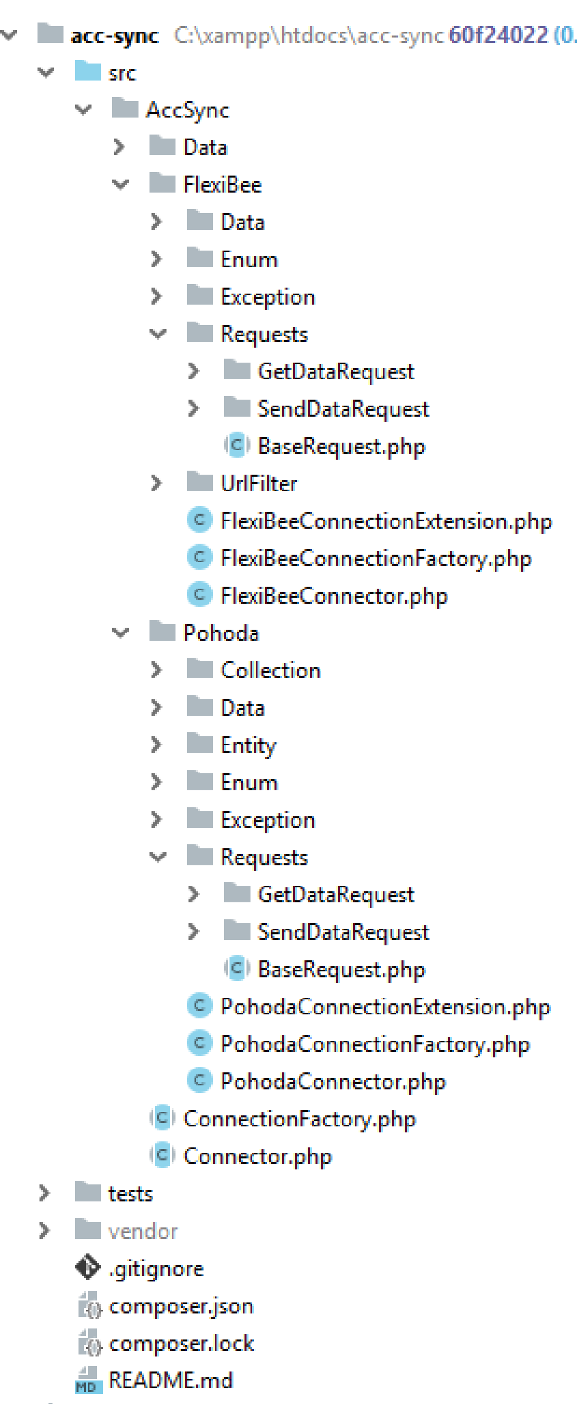
Vzorová aplikace

- zdrojové soubory vytvořené vzorové aplikace

A Správce IIS



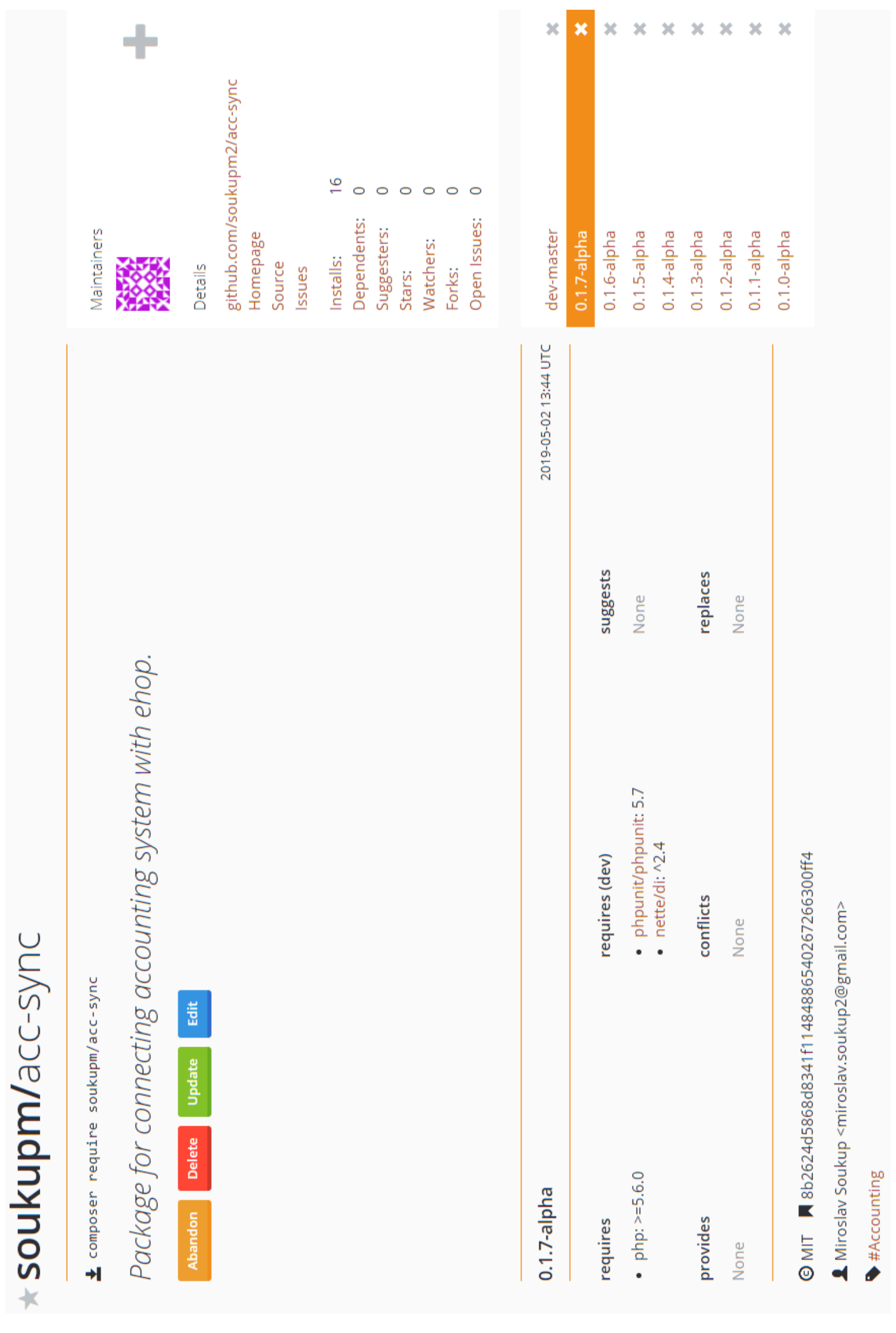
B Struktura komponenty



C Kolekce

```
abstract class BaseCollection implements \ArrayAccess,
    \IteratorAggregate,
    \Countable
{
    /** @var array $collection */
    protected $collection = [];

    public function getIterator()
    {
        return new \ArrayIterator($this->collection);
    }

    public function offsetExists($offset)
    {
        return isset($this->collection[$offset]);
    }

    public function offsetGet($offset)
    {
        return $this->collection[$offset];
    }

    public function offsetSet($offset, $value)
    {
        if (is_scalar($offset))
        {
            $this->collection[$offset] = $value;
        }
        elseif ($offset == null)
        {
            $this->collection[] = $value;
        }
    }

    public function offsetUnset($offset)
    {
        unset($this->collection[$offset]);
    }

    public function count()
    {
        return count($this->collection);
    }
}
```

D Podoba balíčku v Packagist.org



E Rozšíření pro Nette framework

```
class PohodaConnectionExtension
    extends \Nette\DI\CompilerExtension
{
  const BASE_URI = 'baseUri';
  const USERNAME = 'username';
  const PASSWORD = 'password';
  const COMPANY_ID = 'companyId';
  const PORT = 'port';

  const REQUIRED_PARAMS = [
    self::BASE_URI,
    self::USERNAME,
    self::PASSWORD,
    self::COMPANY_ID,
  ];

  /**  @var string $baseUri */
  private $baseUri;

  /** @var string $username */
  private $username;

  /** @var string $password */
  private $password;

  /** @var string $companyId */
  private $companyId;

  /** @var int $port */
  private $port = NULL;

  /**
   * Loads configuration from config file
   *
   * @throws \Nette\Neon\Exception
   */
  public function loadConfiguration()
  {
    $this->setUpParams();

    $builder = $this->getContainerBuilder();
    $builder->addDefinition(
                $this->prefix('pohoda.connector')
            )
            ->setFactory(PohodaConnectionFactory::class)
            ->setArguments([
                $this->baseUri,
                $this->username,
                $this->password,
                $this->companyId,
                $this->port]
            );
  }
```

```
    /**
     * Sets up the parameters from configuration file
     *
     * @throws \Nette\Neon\Exception
     */
    private function setUpParams()
    {
        $configParams = $this->getConfig();

        foreach (self::REQUIRED_PARAMS as $required)
        {
            if (!isset($configParams[$required]))
            {
                throw new ServiceCreationException(
                    'Missing parameter '
                    . $required . ' in configuration file!');
            }
        }

        $this->baseUri = $configParams[self::BASE_URI];
        $this->username = $configParams[self::USERNAME];
        $this->password = $configParams[self::PASSWORD];
        $this->companyId=$configParams[self::COMPANY_ID];

        if (!empty($configParams[self::PORT]))
        {
            $this->port = $configParams[self::PORT];
        }
    }
}
```