FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Systém pro zpracování firemní listové pošty
Student:	Ondřej Máca
Vedoucí:	Ing. Petr Špaček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat webový systém pro správu firemní listové pošty, který bude podporovat všechny případy užití existujícího řešení a zároveň eliminuje návrhové nedostatky, kterými současné řešení trpí. Seznamte se s existujícím řešením pro správu listové pošty a posléze navrhněte, pomocí metod softwarového inženýrství a refactoringu, systém nový, který bude mít architekturu typu SPA (Single Page Architecture) a bude dodržovat pravidla dobrého návrhu tak, aby vylepšil udržitelnost a rozšiřitelnost systému jako takového.
Požadované technologie jsou: PHP & Symfony, JavaScript & AngularJS.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 22. prosince 2016

Poděkování

Rád bych poděkoval svému vedoucímu bakalářské práce Ing. Petru Špačkovi Ph.D. za jeho aktivní vedení a velmi hodnotné rady během vytváření této práce.

Prohlášení

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

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

V Praze dne 9. ledna 2019 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Ondřej Máca. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Máca, Ondřej. *Systém pro zpracování firemní listové pošty*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato bakalářská práce se zabývá vytvořením webového systému pro správu firemní listové pošty. Systém je tvořen transformací stávajícího systému a eliminuje původní návrhové nedostatky. Transformace je provedena za využití semi-automatického přístupu, který převod usnadňuje. V první části práce diskutuji nad způsoby a postupy, kterými lze převod provádět za využití současných technologií. V druhé části pak rozebírám postup samotného převodu. Cíl práce se podařilo zcela split. Přínosem této práce jsou poznatky zjištěné při semi-automatickém převodu původního systému do systému postaveného na nových technologiích.

Klíčová slova Špagety kód, udržovatelný kód, transformace, firemní systém, PHP, Symfony, Angular

Abstract

This bachelor thesis deals with the development of a web system for company letter management. The system is created by transformation of the existing system and it is eliminating the original design flaws. Transformation is done by using a semi-automatic approach that facilitates the conversion. In the first part I discuss the ways and procedures by which the transfer can be made using current technologies. In the second part, I discuss the process of the transfer itself. The objective of the thesis was accomplished. The contribution of the thesis is the knowledge of the semi-automatic transfer of the original system into a system based on new technologies.

Keywords Spaghetti code, maintainable code, transformation, company system, PHP, Symphony, Angular

Obsah

Úvod **1**

1 Analýza **3**
- 1.1 Současný stav řešení problematiky . . . 3
- 1.2 Způsoby převodu . . . 9
- 1.3 Možnosti řešení . . . 11
- 1.4 Analýza převodu . . . 13
- 1.5 Dostupné analyzátory . . . 14
- 1.6 Zvolený analyzátor . . . 15

2 Návrh **17**
- 2.1 Použité technologie . . . 17
- 2.2 Konfigurace . . . 20
- 2.3 Rozbor automatické části převodu . . . 24

3 Realizace **27**
- 3.1 Automatická část převodu . . . 27
- 3.2 Manuální část převodu . . . 30

Závěr **43**

Literatura **45**

A Seznam použitých zkratek **47**

Seznam obrázků

1.1 ER Model data databáze 4

2.1 Architektura systému 21

Seznam tabulek

1.1 Funkční požadavky 5
1.2 Nefunkční požadavky 6
1.3 Analýza složek 8
1.4 Analýza souborů 9

2.1 Převod datových typů 23

3.1 Stav controllerů 32

Úvod

Bez informačních systémů se dnešní moderní společnost již jen těžko obejde. Manuální správa listové pošty ve firmách přináši spoustu problémů a proto byl vytvořen informační systém, který práci zjednodušil a zeefektivnil. V současné době je však tento systém zastaralý a proto bude vytvořen systém nový. Při tvorbě bude kladen důraz na to, aby nový systém netrpěl původními návrhovými nedostatky. Docíleno toho bude za pomocí moderních technologií a osvědčených postupů přispívajících k dobré udržitelnosti a rozšiřovatelnosti informačních systémů. Tvorba nového systému bude provedena za pomoci semiautomatizovaného přístupu, který převod usnadní.

Tato bakalářská práce je určena pro všechny, kteří chtějí nahradit nepřehledný kód stávajícího systému moderními technologiemi, přispívajícími ke kvalitnímu a efektivnímu kódu. Téma jsem si vybral, neboť usnadní firmám modernizaci informačních systémů a pomůže zlepšit rozšiřitelnost a udržovatelnost těchto systémů.

V této bakalářské práci se v první části věnuji analýze současného systému, kde probírám jeho strukturu a požadavky, které jsou na tento systém kladeny. Dále diskutuji nad způsoby řešení převodu systému z čehož vyplyne potřeba systém analyzovat a vybrat vhodné technologie pro nový systém a převod samotný. V druhé části pak jednotlivé technologie popisuji více detailněji a píši o jejich použití. V poslední části pak popisuji samotnou realizaci převodu systému. Zvláště hovořím o automatické části převodu a o části, kterou bylo nutné převést manuálně.

Kapitola 1

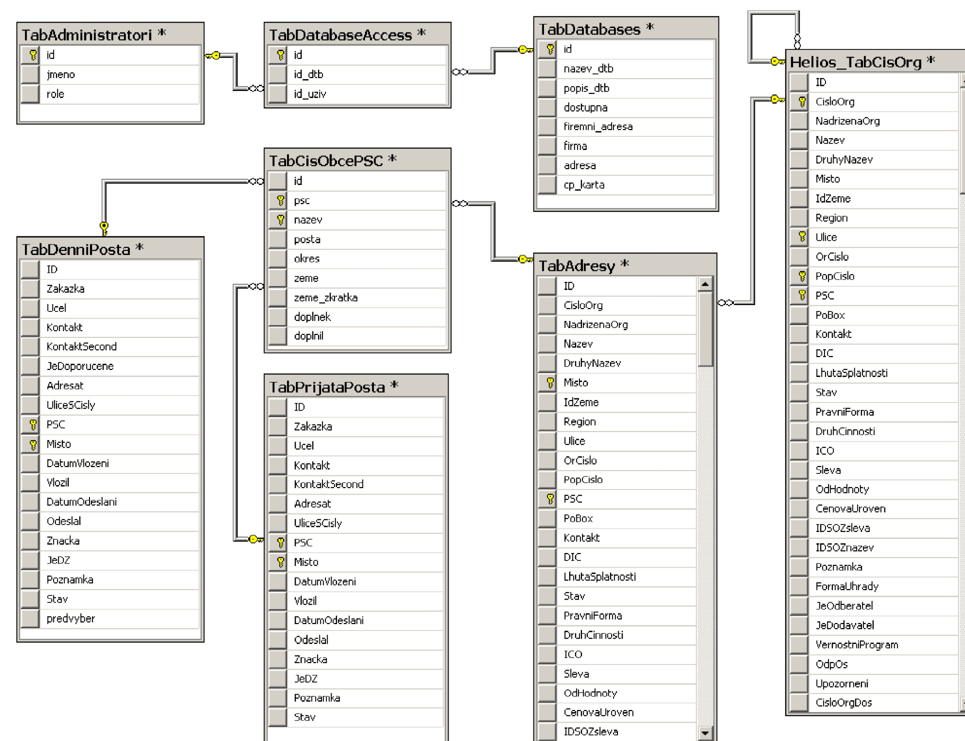
Analýza

1.1 Současný stav řešení problematiky

V současnosti je systém tvořen špagety kódem, což způsobuje nepřehlednost. PHP kód je umístěn přímo uvnitř HTML kódu. Pro změnu v kódu je zapotřebí znát celý kód. V kódu je také velké množství duplicit, což činí udržovatelnost kódu obtížnější. Pokud se kód změní na jednom místě, musí se změnit i na ostatních místech. Musí se také dávat velký pozor, aby byl kód změněn skutečně všude tam kde má být změněn, jinak nastane chyba, kterou může být těžké odhalit. Druhým problémem je i PHP kód, který provádí dotazy nad databází. Je zde tedy míchána samotná logika a práce s daty. Kód pro práci s databází tedy není příliš znovupoužitelný a případná změna technologie související s databází může způsobit velké komplikace. Celkově je tedy udržovatelnost kódu velmi náročná a nese sebou velké riziko vzniku chyb. Rozšířitelnost je na tom velmi podobně. Je tedy třeba se rozhodnout, jak převod započít a jak dále pokračovat. Je potřeba prozkoumat soubory starého systému a určit, který z nich bude třeba převést a který zůstane ve větší či menší míře zachován.

1.1.1 Databáze původního projektu

Databáze původního projektu se skládá z osmi tabulek sloužících pro správu listové pošty. Jak lze vidět na diagramu, nejsou propojeny všechny vzájemně. Nejdůležitější roli zastává tabulka TabCisObcePSC, která je propojena se třemi dalšími tabulkami. Vazba mezi tabulkami však nenastává pomocí primárního klíče ID, ale podle dvojice hodnot, které tvoří klíč. V tabulce TabCisObce jsou onou dvojicí hodnoty psc a nazev, zatímco ve zbývajících třech tabulkách to jsou dvojice Misto a PSC. Dotazovaní nad databází se tak mírně komplikuje, ale nemusí jít nutně ještě o návrhový nedostatek. Problémem však může být rozsáhlost tabulek TabAdresy a Helios_TabCisOrg, které mají kolem 100 sloupců. Druhý problém může být nesoulad v typech mezi jednot-



Obrázek 1.1: ER Model data databáze

livými tabulkami, kdy sloupce určující PSČ nemají stejnou délku a nemusejí tak jít provádět některé operace mezi těmito tabulkami. Možná by tedy stálo za úvahu tabulky refaktorovat. To ale nebylo cílem mojí práce.

1.1.2 Požadavky na systém

Při vymezení cíle práce bylo na systém kladeno několik funkčních a nefunkčních požadavků. Funkční požadavky byly shodné s funkčními požadavky původního systému. Nefunkční požadavky byly do jisté míry stejné jako u původního systému, ale lišili se především v použití nových technologií.

1.1.2.1 Funkční požadavky

Zde je seznam funkčních požadavků, které obsahoval původní projekt a které obsahuje i nový projekt. Všechny funkční požadavky se tedy povedlo splnit. Je tedy možné v novém systému používat stejné funkcionality jako v systému původním, tak jak jsou uživatelé zvyklí.

Tabulka 1.1: Funkční požadavky

Identifikace	**Funkční požadavky**	**Splněno**
FP00	Převod předvybraných záznamů z odeslané pošty do vyřízené	Ano
FP01	Převod všech záznamů z přijaté pošty do vyřízené	Ano
FP02	Převod záznamu z přijaté pošty do vyřízené	Ano
FP03	Smazání PSČ z databáze	Ano
FP04	Uložení PSČ do databáze	Ano
FP05	Smazání adresáta z interní databáze	Ano
FP06	Uložení adresáta do adresáře	Ano
FP07	Přidání datového záznamu do pošty	Ano
FP08	Hromadné přidání adresátů I. fáze	Ano
FP09	Zobrazení přehledu denní pošty	Ano
FP10	Zobrazení přehledu pošty pro přijetí	Ano
FP11	Zobrazení přehledu pošty pro odeslání	Ano
FP12	Zobrazení přehledu přijaté pošty	Ano
FP13	Zobrazení přehledu přijaté vyřízené pošty	Ano
FP14	Zobrazení přehledu odeslané vyřízené pošty	Ano
FP15	Zobrazení přehledu interních adresátů	Ano
FP16	Zobrazení přehledu poštovních směrovacích čísel	Ano
FP17	Zobrazení informací o databázi	Ano
FP18	Editace adresáta	Ano
FP19	Editace poštovního směrovacího čísla	Ano
FP20	Předvýběr všech záznamů pošty pro odeslání	Ano
FP21	Předvýběr vybraných záznamů pošty pro odeslání	Ano
FP22	Předvýběr doporučené pošty pro odeslání	Ano
FP23	Smazání záznamu z denní pošty	Ano
FP24	Smazání záznamu z přijaté pošty	Ano
FP25	Smazání všech záznamů přijaté pošty	Ano
FP26	Smazání všech záznamů denní pošty	Ano
FP27	Hromadné přidání adresátů II. fáze	Ano
FP28	Přidání odesílatele	Ano
FP29	Přidání adresáta	Ano
FP30	Přidání datového záznamu odeslané pošty	Ano
FP31	Přesun denní pošty do seznamu přijaté pošty	Ano
FP32	Přesun přijaté pošty do seznamu pro odeslání	Ano
FP33	Hromadný přesun přijaté pošty do seznamu pro odeslání	Ano
FP34	Hromadný přesun denní pošty do seznamu pro odeslání	Ano
FP35	Vyhledávání PSČ	Ano
FP36	Vyhledávání Adresáta	Ano
FP37	Použití filtrů při vyhledávání	Ano
FP38	Ověření uživatelů	Ano
FP39	Generování PDF souborů	Ano

Tabulka 1.2: Nefunkční požadavky

Identifikace	Nefunkční požadavky	Splněno
NP01	Back-end založen na technologii PHP ve verzi alespoň 7	Ano
NP02	Front-end založen na technologiích HTML, CSS, JavaScript, Angular	Ano
NP03	Odezva systému na HTTP pozadavek max. 5 sekund	Ano
NP04	Validace uživatelského vstupu	Ano
NP05	Ochrana proti cross-site-scriptiong (XSS)	Ano
NP06	Ochrana proti SQL injection	Ano

1.1.2.2 Nefunkční požadavky

Na nový systém bylo kladeno 7 nefunkčních požadavků, které se všechny povedlo splnit viz Tabulka 1.2. Splnění požadavků NP01 a NP02 vyplývá přímo z názvu. Splnění požadavků NP03 závisí na velikosti vstupních dat, počtu uživatelů a odezvě serveru, což jsou parametry, které mnou nebylo možné ovlivnit. Projekt je ale založen na nových technologiích a z testovacích dat, vyplývá, že by nemělo být za běžných obtížné požadavek NP03 splnit. Požadavek NP04 byl splněn tak, že se uživatelská data validují do té míry, v jaké se validovaly v původním systému. Většinou se tedy jedná o kontrolu vyplnění povinných údajů a kontrolu jejich délky. Požadavek NP05 je splněn automicky použitím frameworku Angular. Ten spouští službu zvanou DomSanitizer, která upravuje nedůvěryhodné hodnoty, tak aby byly bezpečné[1]. Požadavek NP06 byl při volání DQL dotazů splněn díky použití členské funkce objektu `\Doctrine\ORM\Query setParameter()`. Při použití členských metod ORM objektů je bezpečnost proti SQL injection zajištěna automaticky pomocí ORM[2].

1.1.3 Seznámení se soubory

Kód starého systému je směsicí PHP kódu a HTML kódu a navíc jsou zde volány i SQL dotazy nad databází. Jak již bylo zmíněno soubory tedy bude nutné převést do frameworku Symfony. Převod musí proběhnout tak, aby nové členění souborů odpovídalo zvyklostem frameworku Symfony. Nebude tedy již docházet k mísení PHP kódu a HTML kódu. Oddělena bude i práce s databází. Projekt navíc bude rozdělen na dvě části, frontend a backend. Přičemž backend bude tvořit REST API právě v Symfony a frontend bude tvořen pomocí frameworku Angular. Soubory jsem si prošel a dospěl k následujícímu zjištění. Původní systém je tvořen několika druhy souborů. Některé obsahují pouze HTML kód, některé pouze Javascript, některé pouze PHP kód a ty ostatní jsou směsicí několika kódů dohromady.

1.1.4 Rozbor souborů

V této části rozeberu jednotlivé soubory a nastíním, co se s nimi bude pravděpodobně dělat. U některých nebyl postup převodu dopředu jasný a objasnil se

až v průběhu převodu. Shrnutí postupu převodu jednotlivých souborů je v tabulce Tabulka 1.3. V prvním sloupci je název souboru či složky a ve druhém popis toho, jak se budou soubor či soubory převádět.

Soubory budu procházet abecedně, přičemž prvně ty v podsložkách a poté ty v kořenu projektu. Nejdříve se tedy podívám do složky css. Zde jsou podle očekávání pouze soubory s koncovkou CSS. Tyto soubory určují styl stránky a není třeba je převádět. Je však nutné zachovat soubory s HTML kódy aspoň do té míry, aby v oněch souborech zůstaly použity původní CSS selektory.

Další složkou je složka Images. Ta obsahuje jak už název napovídá pouze obrázky. Tuto složku stačí víceméně pouze překopírovat do frontendové části nového projektu. Výjimku tvoří pouze několik málo obrázků, které jsou použity v backendové části při generování PDF souborů a budou tedy přesunuty do backendové části.

Další složkou je složka include. V té se nachází soubory s koncovkou js, css a htm. Nejspíše je půjde bez větších změn znovu použít v nové projektu a bude je tedy stačit zkopírovat. Obdobné by to mělo být i se složkou js, která obsahuje pouze JavaScript soubory.

Následuje složka json. Ta obsahuje PHP soubory, které vrací data ve formátu JSON na základě daného SQL dotazu. Soubory jsou velmi podobné. Liší se většinou pouze ve volaném dotazu. Soubory by tedy nemělo být příliš těžké převést.

Další složkou v pořadí je složka log. Do ní se obvykle ukládají informace, které zaznamenávají nějaké dané činnosti při běhu systému. Zde se ukládá pouze čas ke konkrétnímu uživateli, který prováděl na systému nějakou činnost. Takovou složku tedy stačí vytvořit prázdnou a při nasazení je do ní možné podle potřeby zkopírovat původní soubory s logovacími záznamy.

Následuje složka návody, v které je pouze jeden soubor a sice pdf, které obsahuje uživatelský manuál pro obsluhu systému Pošta. Jelikož by u nového systému měla být zachována stejná funkčnost, měl by být návod po převodu stále aktuální. Bude tedy stačit jej pouze zkopírovat.

Dále se dostáváme ke složce pages. Ta obsahuje několik PHP souborů. Některé slouží pro práci s databází a některé pro obsluhu formulářů. Je v nich mísen PHP kód, HTML kód a rovněž i JavaScript kód. Bude je tedy zcela jistě nutné převést.

Další složkou kterou je třeba prozkoumat, je složka sestavy. Nachází se v ní soubory určující fonty a soubory, které generují PDF. Některé bude zřejmě nutné převést, vzhledem k tomu, že obsahují PHP kód, který je potřeba v novém systému správně umístit. Například v některých souborech jsou prováděny dotazy nad databází, které bude navíc nutné přepsat.

Následující složka nese označení temp a obsahuje vygenerované csv soubory. V takto označené složce bývají vždy pouze dočasné soubory. Není je tedy nutné kopírovat. Krom toho framework Symfony, který bude použit pro běh nového systému, má vlastní složku pro dočasné soubory. Složku temp není tedy ani třeba v novém systému vytvářet.

Tabulka 1.3: Analýza složek

Název	**Činnost**
css	zkopirování
datagrid	pravděpodobně bude nahrazeno
images	zkopirování
include	pravděpodobné zkopirování
js	pravděpodobné zkopirování
json	pravděpodobný převod
log	zkopirování
navody	zkopirování
pages	převod
sestavy	pravděpodobný převod
temp	nebude potřeba

Nyní se dostáváme k rozboru souborů, které jsou přímo v kořenové složce projektu. Prvním z nich je `config.inc`, což je konfigurační soubor aplikace. Definuje 3 konstanty platné pro celý projekt. Jedná se o konstanty pro nastavení databáze, složky pro dočasné CSV soubory a o nastavení šířky štítků. Některé konstanty jako třeba nastavení databáze se budou nacházet v jiném konfiguračním souboru, který je pro to zvláště určený. U jiných je možné, že zůstanou v tomto souboru. Zatím ale není třeba věnovat tomu přílišnou pozornost.

Dalším souborem k prozkoumání je soubor `connection.php`. Tento soubor poskytuje základní funkce pro práci s databází, jako je třeba načtení názvu databáze nebo ukládání dat. Tyto a další funkce se volají z několika různých souborů napříč aplikací. Jsou to např. `index.php`, nebo soubory pro práci s databázi ve složce pages. Soubor `connection.php` tedy bude nutné alespoň z části převést. Některé funkce zřejmě převést nebude třeba, jelikož se jedná o funkce, jež sám vnitřně implementuje framework Doctrine.

Následující soubor `exporty.php` je jakousi knihovnou pro vytváření reportů. V podstatě jediné co soubor provádí je, že vkládá do skriptu různé soubory v závislosti na superglobálních proměnných. Soubory, které se vkládají jsou soubory ze složky sestavy. Soubor bude třeba převést, ale větší část souboru by mohla zůstat zachována.

Soubor `forms.php` se skládá ze tří funkcí. Ty obsahují převážně HTML kód, ale i PHP pro přístup k superglobálním proměnným a pro dotazy nad databází. Jedná se o funkce pro vytvoření hlavní stránky, výpis přihlašovacího formuláře a vytvoření patičky stránky. Jistě tedy bude nutné tento soubor převést.

Dalším souborem je soubor `functions.php`. Ten obsahuje přídavné funkce pro různé formátování textu. Některé z nich jsou hojně používané, některé nejsou použity prakticky vůbec. Soubor bude zřejmě stačit pouze zkopírovat

Tabulka 1.4: Analýza souborů

Název	Činnost
config.inc	pravděpodobně bude nahrazeno
connection.php	převod
exporty.php	převod
forms.php	převod
functions.php	zkopirování
index.php	převod
parseCSV.php	převod

na vhodné umístění do struktury Symfony.

Hlavní a také jedním z největších souborů je soubor `index.php`. Jedná se o řídící soubor celé aplikace, který se spouští jako první. Obsahuje kód, který půjde převést automatizovaně. Detailnější rozbor souboru `index.php` je proveden v následující kapitole.

Posledním souborem v kořenovém adresáři projektu je soubor `parse-CSV.php`. Ten slouží jak již název napovídá pro zpracování CSV souborů. Funkce, které obsahuje jsou volány výhradně ze souboru index. Obsahují však převážně PHP kód a jejich převod by tedy neměl být příliš náročný.

1.2 Způsoby převodu

Pro převod systému se nabízí několik způsobů. Já se budu zabývat dvěma relevantními, které rozeberu a popíši jejich výhody a nevýhody.

1.2.1 Porovnání způsobů

Samotný převod lze provést dvěma způsoby. První možností je volat ze Symfony kód starého systému a ten postupně nahrazovat kódem reflektujícím framework Symfony. Během celého vývoje tedy bude systém funkční. Mělo by tedy být snadnější nalézt chyby a celý vývoj by měl být i jednodušší. A to proto, že bude vždy snadno vidět, co je potřeba zrovna převést. Druhou možností je převádět starý systém na nový postupně tak, že začnu s prázdným projektem Symfony a budu do něj postupně přidávat nové a nové funkcionality a až na konci bude systém plně funkční.

1.2.2 Nástin prvního způsobu

Z výše uvedeného důvodu – mít systém funkční během celého vývoje, jsem se rozhodl zkusit první způsob převodu. Bylo tedy třeba nainstalovat Symfony na virtuální počítač, kde mám nainstalované PHP ve verzi 5.3.5. Již toto naznačuje jisté komplikace. Nejvyšší verze Symfony, která lze s verzí PHP 5.3.5 naistalovat je Symfony 2.6. Tato verze Symfony však již není podporovaná a

bylo by více než vhodné ji po dokončení převodu systému přepsat na novější verzi. Symfony sice umožňuje upgradování verzí, ale u takto staré verze by těch upgradů muselo být pravděpodobně více a nejspíše by se i některé převody musely provést manuálně.

Mohl jsem ale také upgradovat PHP na verzi 5.3.9, na kterém by měl původní systém stále fungovat. S touto verzi PHP totiž funguje Symfony ve verzi 2.7. Jedná se také o starší verzi, ale protože je to LTS verze, tak je stále podporovaná. Zkusil jsem tedy provést upgrade PHP stáhnutím nových souborů a zkopírovat je do původní složky PHP. Tento upgrade se ale nepovedl. Dále jsem zkusil upgrade pomocí instalátoru msi. Ani pomocí toho se však upgrade nezdařil.

Rozhodl jsem se tedy zůstat u verze PHP 5.3.5 a nainstalovat Symfony ve verzi 2.6. Instalace ovšem nešla provést pomocí instalátoru, kterým se běžně Symfony v nových verzích instaluje, ale musela se provést ručně pomocí programu Composer. Composer je nástroj na správu závislostí v PHP, který bylo ale třeba nejprve naistalovat. Instalace Composeru byla snadná, pro Windows k tomu existuje instalátor ve formátu exe. Následovala tedy instalace Symfony. Ukázka kódu znázorňuje, co vše bylo třeba provést. Je to docela značný rozdíl oproti dnešní instalaci symfony pomocí instalátoru.

```
php -r "file_put_contents('symfony', file_get_contents('https://symfony.com/
    installer'));"
move symfony c:\xampp\php
cd c:\xampp\php
(echo @ECHO OFF & echo php "%~dp0symfony" %*) > symfony.bat
composer create-project symfony/framework-standard-edition sym 2.6
```

Dále jsem se pustil do samotného převodu. Všechny soubory, které byly vygenerovány při vytvoření projektu Symfony jsem přesunul do složky původního projektu. Přesun byl o to snazší, že se žádný soubor nejmenoval stejně. Nový systém byl tedy uvnitř starého a mohlo začít napojování nového systému na systém původní. Start původního systému začíná tak, jak je to běžné souborem – `index.php`. Start nového projektu v technologii Symfony začíná souborem `app.php` umístěným ve složce web, pokud se jedná o projekt v produkci a nebo souborem `app_dev.php` umístěným ve složce web, pokud se jedná o projekt během vývoje. Dalším krokem tedy bylo vložit include souboru `index.php` do souboru `app_dev.php` a docílit toho, aby projekt při spuštění souboru `app_dev.php` fungoval stejně, jako by se spustil soubor `index.php`. Tento úkol se však ukázal mnohem náročnější než se na první pohled jevil a z neznámých důvodů se jej stále nedařilo splnit. Bylo zřejmě třeba prozkoumat hlouběji závislosti při spouštění původního systému a hlavně závislosti, které se provádí při spouštění každého systému založeného na frameworku Symfony. Toto zkoumání by jistě vyžadovalo značné množství času, ale byla tu ještě druhá věc, která se mi na tomto způsobu převodu nelíbila. A sice ta, že jsem si začínal uvědomovat, že tento způsob převodu je velice složitý a možná až nemožný. Nový systém má totiž v backendové části úplně jinou strukturu. Jednotlivé požadavky z daných URL adres se nezpracovávají pomocí souboru

index.php, ale jsou každý zvlášť zpracovány odpovídajícím controllerem. Tento rozdíl mezi strukturami se mi zdál příliš velký na to, aby šlo nějak rozumně zakomponovat tyto dvě struktury současně do jednoho projektu a postupně tak převádět systém z jedné struktury do druhé. Po konzultaci s vedoucím práce, kterému se tento způsob také příliš nezamlouval, jsem se rozhodl, zvolit druhý způsob převodu. A tedy začít stavět nový projekt od začátku ve frameworku Symfony.

1.3 Možnosti řešení

V následující kapitole je nastíněno, jaké jsou možnosti pro výsledný projekt a z jakých důvodů se pro kterou z nich rozhodnout. Konkrétně se jedná o výběr programovací techniky, výběr frameworku, výběr technologií a o volbu postupu převodu.

1.3.1 Výběr programovací techniky

Jako řešení, které eliminuje problémy s udržovatelností a rozšiřitelností se nabízí použití techniky objektově orientovaného programování. [3] Díky rozdělení kódu na objekty se stane kód přehlednějším a mnohem více pochopitelnějším. Logicky související celky tedy budou tvořit jeden objekt. Při použití daného objektu proto nebude nutné znát celý kód objektu, ale pouze to, jak se s daným objektem zachází. Dalším přínosem objektově orientovaného programování je zapouzdření. Pomocí něj můžeme označit metody a proměnné objektu jako soukromé a nebude k nim tedy možné přistupovat od jinud než z instancí daného objektu.[4]

Použití objektově orientovaného programování je dnes v informačních systémech zcela běžné a v podstatě se již u nově vznikajících systémů nesetkáme s jiným přístupem.[5]

1.3.2 Výběr frameworku

Podobné je to i s použitím frameworků, které jsou dnes již také běžnou součástí tvorby informačních systémů. Frameworky obsahují knihovny, které zjednodušují typickou práci pro daný programovací jazyk. Dále frameworky obsahují doporučené postupy, jak by se mělo pomocí daného frameworku vyvíjet. Díky těmto postupům dochází k jisté standardizaci a pro ty kteří se s danými postupy seznámí, je poté kód dodržující tyto postupy snáze pochopitelnější. Naopak nedodržování těchto doporučení může vést k různým problémům, které obvykle vedou k řešením způsobujícím problémy s rozšířitelností a udržovatelností.

1.3.3 Výběr technologií

Po zvolení postupu zbývá už jen výběr konkrétních technologií. Požadavek na systém je, aby byl napsán v jazyku PHP. Je tedy třeba zvolit vhodný framework pro PHP, který je dostatečně rozšířen, aby bylo snadné sehnat či zaškolit programátory pro údržbu systému.

Dále je třeba zvolit technologii pro frontendovou část aplikace. Při této volbě je třeba zohlednit, jaká technologie je použita pro backend aplikace. Framework či frameworky pro backend a frontend by se tedy měli vybírat současně. S volbou frameworku souvisí i to, zda budeme chtít klasickou více stránkovou aplikaci, nebo tzv. jednostránkovou aplikace známou též pod označením Single page application (SPA).

Databáze běží na Microsoft SQL Serveru, na což je třeba také brát ohled. Moderní frameworky ovšem nabízí pro práci s databází rozhraní, která nás od konkrétních databázových strojů odstiňují. Umožňují nám totiž s nimi komunikovat univerzálním způsobem, který je nezávislý na zvoleném databázovém stroji.

1.3.4 Volba postupu převodu

Dále je třeba zvolit postup, jak bude stávající systém refaktorován na nový systém. Nejpřímočařejší postup je přepsat celý kód ručně. Tento postup však není příliš zajímavý a obsahuje spoustu opakující se mechanické práce. Vhodnější by bylo použít řešení, které alespoň částečně refaktoring zautomatizuje a eliminuje tak opakující se mechanickou práci.

1.3.5 Zvolené řešení

Postup, který jsem zvolil pro řešení refaktoringu je částečná automatizace. Použiji analyzátor, pomocí něhož rozdělím určitou část kódu na třídy a na další části, díky čemuž se výrazně usnadní celkový převod. Je tedy třeba analyzátor vhodně vybrat a najít ty části kódu, které půjdou převést automatizovaně.

Nejvhodnější bude vybrat takový, který pracuje s tzv. abstraktním syntaktickým stromem (AST). Existují sice i další způsoby zpracování zdrojového kódu než použití analyzátoru založeného na AST, jako např. tzv. proud tokenů, neboli token stream. Jeho generování podporuje PHP nativně, avšak proud tokenů je mnohem více nízkoúrovňový a hodí se tedy spíše k jiným účelům. Umožňuje např. analyzovat přesné formátování textu. Pro řešení komplexnější analýzy je však příliš složitý.[7]

1.3.5.1 Backendová část

Pro řešení backendové části aplikace jsem použil PHP framework Symfony. Jedná se o jeden z nejrozšířejnějších PHP frameworků jak na světě, tak i v Česku. Má rozsáhlou komunitu vývojářu, díky čemuž je framework dále

vyvíjen, má kvalitní dokumentaci a spoustu užitečných funkcí jak pro malé, tak velké projekty.

Symfony je MVC framework, což znamená že datový model aplikace je rozdělen do tří vrstev zvaných: model, pohled, kontroler. Vrstvy jsou na sobě nezávislé, takže změna jedné vrstvy neovlivní další vrstvy. Je tedy oddělen datový model, řídící logika a uživatelské rozhraní. Díky tomuto rozdělení se kód stává přehlednějším a přispívá k rozšiřitelnosti a udržovatelnosti.

Dále bylo rozhodnuto, že backendová část bude zastávat funkci jakéhosi REST API. Backendová část bude tedy oddělena od frontendové a komunikace mezi nimi bude probíhat skrze JSON soubory.

1.3.5.2 Frontendová část

Na frontendovou část aplikace jsem si vybral framework Angular. Jedná se o JavaScriptový framework zaměřený na tvorbu single-page aplikací. Ty mají několik výhod, přičemž hlavní je ta, že se ze serveru nestahuje celý HTML kód, ale pouze data, která se mění. Prohlížeč pak nemusí překreslovat celou stránku[6]. V současné době je Angular nejmodernějším JavaScriptovým frameworkem. Je tedy velmi rozšířený a těší se velké oblibě. Propojení s bakendovou částí aplikace bude probíhat, jak již bylo zmíněno skrze soubory formátu JSON.

1.4 Analýza převodu

1.4.1 Příprava převodu

Nyní je třeba rozvážit, kterými soubory se s převodem začne a do jaké podoby se kód převede. Hlavně je třeba nalézt tu část, která půjde převést automatizovaně a rozmyslet si, jak to provést.

Soubory jsem se rozhodl převádět v pořadí, v jakém se spouští při začátku běhu systému. Nejprve se tedy spustí soubor `index.php`. Ten volá funkce v dalších souborech a ty postupně mohou využívat další soubory.

1.4.2 Rozbor hlavního souboru

Začnu tedy postupně procházet soubor `index.php` a analyzovat, jak kterou část kódu převést. Nejprve soubor zběžně projdu a budu hledat, jestli jsou v něm části, které by šlo převést automatizovaně. Jak už mi i vedoucí práce poradil, tak jedna z částí, které se pro automatický převod nabízí je sekce se dvěma příkazy switch. Tyto příkazy mají stejný parametr a sice superglobální proměnnou `$_GET['action']`. Mezi dvěma příkazy switch je pouze krátká sekce určující barvu stránky. To půjde snadno přesunout do frontendové části, kde se pracuje se vzhledem stránky. Bude tedy možné příkazy switch spojit do jednoho, což bude více než vhodné, protože výsledný spojený switch bude při

převodu nahrazen controllery. Tento převod půjde provést automatizovaně. Jen je třeba nalézt vhodný analyzátor, pomocí něhož automatizovaný převod provedu.

Nyní budu procházet soubor `index.php` více detailněji a zaměřím se tak na každý řádek kódu. Soubor začíná sekcí s příkazy include a require. Ta bude nahrazena sekcemi v jednotlivých souborech (controllerech), do kterých bude soubor `index.php` rozdělen. V Symfony lze pro include použít klíčové slovo use. Dále následuje odhlášení. Na správu přihlášených uživatelů se v Symfony používá zvláštní služba. Není ani třeba přistupovat do superglobálních proměnných. Přístup k superglobálním proměnným je ve frameworku považován za špatnou praxi. Proto se v Symfony obsluha globálních proměnných provádí skrze speciální třídy, jako je třeba třída Post. Dále následuje inicializace databáze a načtení názvu databáze. Ani toto nezůstane v původním stavu. Symfony používá pro konfiguraci databáze speciální konfigurační soubory. Podobné to je i s tzv. přenastavením databáze. To bude nahrazeno přihlašovací třídou. Dále následuje časový filtr - datum, rok, který bude přesunut na frontend. Dále následuje sekce if dlouhá asi 470 řádek. Od řádku 86 se jedná o sekci pro administrátora a od řádku 430 se jedná o sekci pro ostatní uživatele. Hlavní částí této sekce jsou již zmíněné příkazy switch. Před ním jsou pouze příkazy pro nastavení prohledávaného roku a počtu položek na stránku. Ty se přesunou na frontendu, stejně jako změnění barvy stránky na řádku 298, díky čemuž můžu spojit dva zmíněné switche v jeden.

Automatický převod tedy započnu tímto switchem. Většinou obsahuje pouze volání konkrétní funkce v závislosti na proměnné `$_GET['action']` a volání funkce header. Vytvořím controllery automaticky tak, že z každého casu vznikne jeden controller v kterém bude vnitřek casu a funkce, které jsou z casu volané. Jedná se o funkce, které jsou v současnosti ve složce pages.

Za příkazy switch následuje od řádku 431 sekce pro nepřihlášené uživatele. Provádí se zde autorizace a ukládají potřebná data do sessions. Pokud k přihlášení nedojde, zobrazí se akce, které je možné provádět bez přihlášení. Přihlášení bude v novém projektu probíhat přes REST API a většinu současného kódu tedy nebude možné použít. Na konci souboru je již pouze přesměrování pro případ, že není nastavena superglobální proměnná `$_GET['action']`.

1.5 Dostupné analyzátory

Na internetu lze nalézt volně k dispozici spoustu různých analyzátorů, neboli parserů napsaných v různých jazycích a sloužících pro různé účely. Těch které by analyzovali PHP kód není příliš mnoho. Bylo třeba vybrat takový, který by nebyl příliš složitý, měl dobrou dokumentaci a dal se dobře použít pro moje účely. Jako nejjednodušší se na první pohled může jevit použít PHP Simple HTML DOM Parser, což je analyzátor HTML kódu. Klíčová slova jazyka PHP by se tedy musela převést do html tagů. Což by bylo poněkud kostrbaté

a mohli by nastat jisté komplikace. Zkoušel jsem tedy najít analyzátor přímo pro PHP kód.

Jedním z možných parserů, který se nabízel byl phpstan-php-parser od autora phpstan. Parser se nacházel na webovém úložišti GitHub pod licencí MIT. Bylo by jej tedy možné použít. Samotný parser byl však jakýmsi rozšířením pro nástroj PHPStan, neboli PHP Static Analysis Tool. Zkusil jsem tedy najít jednodušší řešení.

Nakonec jsem našel program PHP-Parser od autora nikic, který je rovněž umístěn na úložišti GitHub. Licence dovoluje použití pod podmínkou, že bude zmíněn autor a uvedeny licenční podmínky. Parser má dobrou dokumentaci a zdá se býti vhodný k použití pro moje účely.

1.6 Zvolený analyzátor

PHP-Parser od Nikic je analyzátor pro PHP kód verze 5.2 až 7.2, napsaný v PHP. Jeho účelem je zjednodušit analýzu statických kódů a manipulaci s nimi. Jak autor analyzátoru zmiňuje, PHP nemusí být jazyk vhodný pro rychlou statickou analýzu, ale zpracování PHP kódu je mnohem snazší než zpracování v jiných jazycích, jako jsou jazyky typu C. Krom toho lidé kteří chtějí provádět statickou analýzu PHP kódu, jsou spíše vývojáři jazyka PHP, než vývojáři jazyka C. Syntaktický analyzátor vytváří abstraktní syntaktický strom (AST). Jak strom vypadá, může být nejlépe vidět na příkladu. Z programu `<?php echo 'Hi', 'World';` vznikne AST, který je uložen takto:

```
array(
    0: Stmt_Echo(
        exprs: array(
            0: Scalar_String(
                value: Hi
            )
            1: Scalar_String(
                value: World
            )
        )
    )
)
```

To se shoduje se strukturou kódu: V poli je blok označující funkci echo, která má dva řetězce jakožto výrazy mající hodnoty Hi a World. Z ukázky je také vidět, že AST neobsahuje žádné informace o bílých znacích (i když většina poznámek je uložena). Tento analyzátor tedy nelze použít pro analýzu formátování. V následujícím seznamu jsou uvedeny hlavní funkce této knihovny.

- Parsování PHP 5 a PHP 7 kódu do abstraktního syntaktického stromu (AST).
  - Neplatný kód lze analyzovat do částečného AST.
  - AST obsahuje přesné informace o poloze kódu.
- Vypsání AST v lidsky čitelné podobě.

- Převedení AST zpět na PHP kód.
  - Experimentální: Formátování může být zachováno pro částečně změněné AST.
- Infrastruktura pro přesun a úpravu AST.
- Rozlišení jmenných prostorů.
- Vyhodnocení konstantních výrazů.
- Stavba zjednodušující konstrukci AST pro generování kódu.
- Převedení AST do formátu JSON a zpět. [7]

Kapitola 2

Návrh

2.1 Použité technologie

V následující kapitole se zabývám popisem jednotlivých technologií použitých pro tvorbu nového systému. Vyzdvihuji jejich nejzajímavější vlastnosti a důvody pro jejich užití.

2.1.1 Symfony

Definice 2.1.1. *Symfony je kompletní framework pro PHP, tvořený sadou PHP komponent, který výrazně zjednodušuje tvorbu webových aplikací.*

[8] Symfony je velmi kvalitně objektově navržené a jedná se o jeden z nejvíce světově rozšířených PHP frameworků. Je vhodný na malé i velké projekty a komponenty které využívá, staví na principech MVC architektury.

2.1.1.1 Přednosti Symfony

Existuje mnoho dobrých důvodů, proč si zvolit Symfony jako framework pro vývoj webu v PHP. Zde je 6 těch, které považují tvůrci Symfony za nejdůležitější.

- **Pověst** - Symfony byla rychle přijata odborníky zabývajícími se touto oblastí a od svého vzniku v roce 2005 se stalo stabilním prostředím, které je velmi oblíbené a uznávané po celém světě.

- **Trvalost** - Za projektem Symfony stojí webová agentura SensioLabs vytvořená před 18-ti lety, která má ve svých referencích mnoho významných partnerů. Symfony je navržené od profesionálů pro profesionály. Trvalost týkající se také dlouhé podpory je zapříčiněna jednak firmou SensioLabs, ale i mnoho dalšími, které do tohoto projektu investují. Udržitelný rozvoj je navíc podpořen i licencí MIT.

- **Reference** - V Symfony vkládá důvěru spoustu vývojářů a firem, ať už jsou jakkoliv velké. Běží na něm jak intranety, sociální sítě, komunitní stránky a mnoho dalších. Jedním z nich jsou např. Yahoo!, Dailymotion, Opensky.com, Exercise.com a dokonce i aplikace jako phpBB nebo Drupal.

- **Inovace** - Symfony má vše, co by člověk od frameworku očekával: rychlost, flexibilitu, znovu použitelné komponenty atd. A pak je tu struktura toho, co bylo vyvinuto a nejlépe se osvědčilo praxí. Krom toho se ale Symfony stále inspiruje u jiných nástrojů a zavádí jejich řešení do praxe. Příkladem může být tzv. dependency injection převzatá z Javy, nebo lišta nástrojů pro ladění převzatá z jiných frameworků.

- **Zdroje** - Ve světě Symfony lze vždy nalézt odpověď na každou otázku. Ať už se jedná o podporu ze strany komunity, nebo podporu formou konzultací a školení od samotné společnosti.

- **Interoperabilita** - Myšlenka za kterou Symfony stojí je, abychom nezůstávali uvěznění pouze v samotném Symfony. Symfony respektuje současné konvence pojmenování a podobné zvyklosti a dovoluje používat stavební bloky aniž by bylo třeba používat framework jako celek. Samotné Symfony je příkladem Interoperability tím, že používá externí softwarové bloky jako třeba ORM Doctrine pro správu databáze, nebo Swiftmailer pro správu emailů.

[9]

2.1.1.2 Doctrine

Symfony nepracuje přímo s databází, ale poskytuje úzké napojení přes knihovnu třetí strany zvanou Doctrine. Pro dotazování lze použít jazyk DQL, který je velmi podobný jazyku SQL. Největší rozdíl oproti SQL je v tom, že je třeba nad dotazem myslet jako nad výběrem objektů, namísto řádků v tabulce. Doctrine se tedy stačí soustředit na objekty a není třeba se starat o ukládání dat do databáze. [10]

Druhá možnost jak přistupovat k databázi je pomocí objektu zvaného QueryBuilder. To je užitečné hlavně, když dotaz závisí na dynamických podmínkách. Pak se totiž kód stává těžko čitelný, jak se spojují řetězce do jednoho DQL dotazu.

Mapování potřebných vazeb mezi entitami se děje pomocí technologie zvané asociace. Nahrazují se tím vazby mezi tabulkami známé z relačních databází. Provádět to lze několika způsoby, přičemž tím nejpoužívanějším jsou anotace. [11]

2.1.2 Angular

Definice 2.1.2. *Angular je MVC framework od Googlu, který vznikl jako reakce na rozrůstající se JavaScript, na který přestala knihovna JQuery stačit.*

Ta má totiž velkou nevýhodu v tom, že míchá dohromady aplikační logiku, zpracování událostí a manipulaci s DOM. U velkých aplikací se tak začne programátor v kódu ztrácet. Angular tento problém řeší právě tím, že je postaven na architektuře MVC. Podle mnohých mezi nejužitečnější koncepty frameworku patří: Two-Way Data Binding, implementace Dependency Injection, testovatelnost, direktivy a znovupoužitelnost komponent. [12]

2.1.2.1 Two-Way Data Binding

Do češtiny lze pojem Two-Way Data Binding přeložit jako "dvoucestná synchronizace dat" a slouží k tomu, že řeší synchronizaci stavů mezi modelem a view. Funguje to tak, že když např. uživatel napíše řetězec do textového pole, tak se tento text přenese do modelu (to je ta první cesta) a z modelu se následně přepíše do ostatních částí view (to je ta druhá cesta). V jiných frameworcích lze toto samozřejmě řešit také, ale v případě Angular je tohle výborně zautomatizováno.

2.1.2.2 Dependency Injection

Dependency injection je návrhový vzor, který řeší závislosti mezi jednotlivými komponentami programu. Vychází z obecnějšího návrhového vzoru Inversion of Control (IoC)[13].

Angular obsahuje zabudovaný subsystém, které řeší Dependency Injection napříč celou vyvíjenou aplikací. Umožňuje tedy pokaždé přesně specifikovat, které komponenty jsou uvnitř konkrétní komponenty používané.

2.1.2.3 Testovatelnost

Testovatelnost patří k jedné z hlavních oblastní, na kterou je Angular zaměřen. Přispívá k ní implementace Dependency Injection a nástroje pro testování, které vznikly pro potřeby frameworku.

Existuje několik nástrojů pro testování, jako např. Testacular, který umožňuje spouštět automatické testy nad různými prohlížeči. Dalším projektem, který vznikl kvůli Angularu je Jasmine-node, který přenesl jeden z nejpoužívanějších frameworků Jasmine do prostředí Node.js. Lze však volně stáhnout šablonu angular-seed v které je předpřipravené prostředí pro jednotkové a end2end testy.

2.1.2.4 Direktivy

Direktivy jsou speciální atributy, které do klasického HTML nepatří. Slouží pro zpracování HTML kódu pomocí Angularu.[14] Můžeme např. říci, že při kliknutí na nějaký textový element se text promění v textové pole a po kliknutí mimo tento element se nad změněnými daty v textovém poli provede nějaká operace, jako např. odeslání dat na server.

2.1.3 Architektura

Názornou ilustraci architektury můžete vidět na obrázku Obrázek 2.1. Architektura nového systému je jak již bylo řečeno rozděla na dvě části.

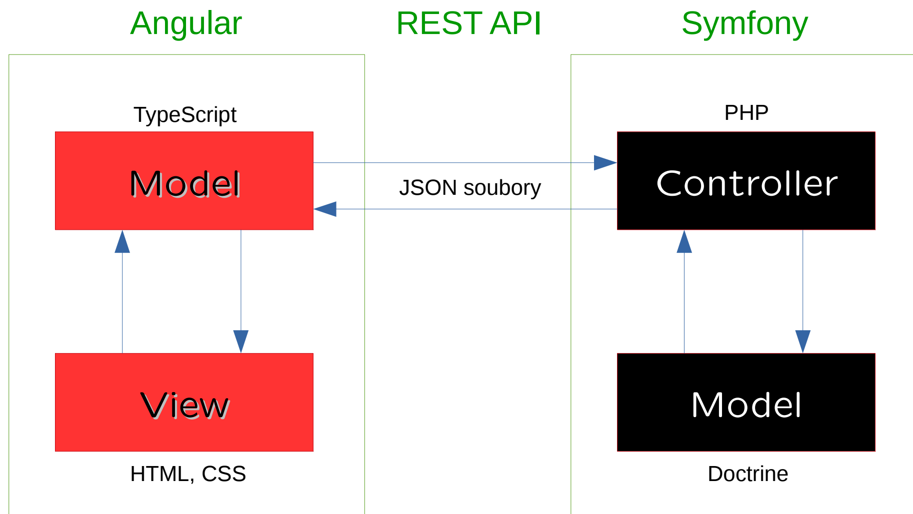
První částí je backendová část, která je napsaná s využitím frameworku Symfony. Tato část je dále rozdělena do vrstev zvaných Controller a Model. Hlavním úkolem vrstvy Model je obstarávání komunikace s databází pomocí frameworku Doctrine. Dále pak vrstva Model komunikuje s vrstvou Controller, od které přijímá požadavky a vrací ji požadovaná data z databáze, nebo od ní naopak přijímá data a zapisuje je do databáze. Vrstva Controller krom komunikace s vrstvou Model poskytuje i rozhraní REST API, pomocí něhož komunikuje s frontendovou částí, tedy frameworkem Angular. Datový přenos u této komunikace probíhá skrz JSON soubory v závislosti na routovacích URL adresách, které směřují k jednotlivým Controllerům a jejich funkcím.

Druhou částí, jak již bylo naznačeno je frontendová část napsaná v jazyku Angular. Tato část je dále rozdělena na vrstvy Model a View. První úlohou vrstvy Model je již zmíněná komunikace s vrstvou Controller v backendové části pomocí rozhraní REST API. Komponenty, které ve frameworku Angular slouží pro tuto komunikaci jsou napsané v jazyku TypeScript. Druhou úlohou vrstvy Model je komunikace s vrstvou View. Získaná data z backendové části vrstva View zpracovává v šablonách a generuje z nich potřebné HTML soubory. Ty potom v kombinaci s CSS soubory zobrazuje v prohlížeči pro uživatele systému.

2.2 Konfigurace

Před započetím samotného programování a převodu, bylo třeba provést několik konfigurací a nainstalovat potřebné nástroje. Konkrétně se jednalo o instalaci Symfony a Angular a jejich konfiguraci, zejména pro rozhraní REST API. Dále se jednalo o konfiguraci databáze přes framework Doctrine.

Konfiguraci, stejně jako samotný převod jsem prováděl střídavě dle potřeby na backendové i frontendové části, ale pro přehlednost zde píší o každém zvlášť.



Obrázek 2.1: Architektura systému

2.2.1 Instalace frameworku Symfony

Symfony jsem se rozhodl nainstalovat ve verzi 3.3, protože se v době začátku konání bakalářské práce jednalo o nejnovější verzi. V průběhu programování jsem se však po nějaké době dostal do stavu, kdy verze 3.3 již nebyla dostatečná pro použití. Stalo se to při instalaci bundlu LexikJWTAuthenticationBundle, který se využívá pro autentizaci a vyžadoval minimálně verzi 3.4. Upgrade na tuto verzi, jak se dočtete i v sekci o autentizaci proběhl bez problému. Seznam verzí a informace o jejich podpoře, lze nalézt na oficiálních stránkách[15] Symfony. Instalace Symfony je velmi jednoduchá a informace o instalaci konkrétní verze lze rovněž nalézt na zmíněných oficiálních stránkách. Instalace se skládá z několika málo kroků. U verzí 3.3 a 3.4 stačí vytvořit adresář, stáhnout do něj installer a nastavit mu práva. Na systémech typu linux, se jedná konkrétně o tyto kroky:

```
sudo mkdir -p /usr/local/bin sudo mkdir -p /usr/local/bin
sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
sudo chmod a+x /usr/local/bin/symfony
```

Následně už stačí projekt pouze vytvořit příkazem:

`symfony new my_project_name version`. Na místo slova `version`, lze napsat konkrétní verzi, jako např 3.3.15, nebo větev verze, tedy např. 3.3. Případně lze i informaci o verzi vynechat a v takovém případě se vytvoří projekt v nejnovější dostupné verzi. Pro spuštění systému lze využít virtuální server, který je součástí Symfony a není třeba pro základní použití nijak konfigurovat. Spouští se v kořenové složce projektu příkazem: `php bin/console server:run` a běží na adrese `http://localhost:8000`.

2.2.2 Nastavení REST API

Pro zprovoznění funkčnosti REST API je zapotřebí použít několik balíčků neboli bundlů. Konkrétně se jedná o tyto bundly:

- **FOS REST Bundle** - obsahuje nástroje pro REST API
- **JMS Serializer Bundle** - usnadňuje transformaci JSON dat na entity.
- **Nelmio CORS Bundle** - slouží ke konfiguraci hlaviček sdílených zdrojů
- **Nelmio API Doc Bundle** - generuje dokumentaci z anotací

Tyto bundly je třeba nejprve zaregistrovat do souboru composer.json, pomocí něhož se bundly nainstalují při spuštění composeru a zároveň zapíší do souboru `app/AppKernel.php`. Mnoho nastavení těchto bundlů lze provést v souboru `app/config/config.yml`. Krom toho je i vhodné upravit soubor `app/config/parameters.yml` a `app/config/parameters.yml.dist` v nichž jsou informace jako např. název a popis API. Do souboru `app/config/routing.yml` je nutné doplnit informace jako např. název souboru, který bude obsahovat cesty ke controllerům. Onen soubor jsem vygeneroval pomocí scriptu při tvorbě controllerů, jak se více dočtete v následující kapitole. Nakonec je vhodné rovněž upravit i soubor `app/config/security.yml` sloužící pro zabezpečení.

2.2.3 Konfigurace databáze

Jak již bylo zmíněno, pro přístup k databázi budu používat framework Doctrine. Ten mapuje objekty zvané entity na konkrétní tabulky dané databáze.

Instalace je velice jednoduchá. Doctrine, jakožto i jakýkoliv jiný bundle v Symfony stačí pouze zaregistrovat pomocí nástroje composer příkazem: `composer require doctrine/doctrine-bundle` a následně vyvolat jeho spuštění pomocí umístění následujícího řádku do souboru `app/AppKernel.php` do pole bundles.

```
new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
```

Poté je třeba databázi vytvořit. Vyplní se konfigurační údaje v souboru `app/config/parameters.yml` jako např. název databáze nebo přihlašovací údaje a spustí se příkaz `php bin/console doctrine:database:create`.

V původní projektu byla použita databáze Microsoft SQL Server. Nejprve tedy musím zjistit jaké obsahuje tabulky a podle toho vytvořit příslušné entity v Doctrine. Pomocí nástroje Microsoft SQL Server Management Studio lze vygenerovat informace o tabulkách příkazem: `Script Table/View as - CREATE To`.

Z těchto informaci budu tvořit Entity. Ty se podle konvence ukládají do složky `src/NejakyBundle/Entity`. Lze je psat ručně, nebo můžeme použít interaktivní nástroj, jemuž postupně zadáváme informace o atributech dané

Tabulka 2.1: Převod datových typů

TSQL datový typ	Doctrine datový typ
tinyint	smallint
smallint	smallint
bit	boolean
int	integer
nvarchar	string
numeric	decimal
ntext	text
datetime	datetime
image	blob

entity a on ji vytvoří automaticky včetně getterů a setterů. Tento nástroj se spouští příkazem `php bin/console doctrine:generate:entity`. Při takovémto tvoření entit není třeba zadávat Id, jakožto identifikátor, ten se vytvoří automaticky. V některých případech je však třeba změnit název, nebo typ identifikátoru, jako tomu bylo i zde u několika případů. Lze jej změnit v kódu třídy definující danou entitu.

Datové typy původního projektu, tedy databáze Microsoft TSQL serveru nejsou ale všechny stejné, jako typy Doctrine. V Doctrine např. neexistuje tinyint a některé typy mají jiný název. Typy jsem tedy musel namapovat způsobem, který je uveden v následující tabulce. Přičemž u nchar bylo třeba doplnit do kódu u dané entity parametr `option fixed`.

Nakonec, aby se změny skutečně projevily v databázi, je nutné spustit následující příkaz: `php bin/console doctrine:schema:update --force`

2.2.4 Instalace frameworku Angular

Instalace frameworku Angular probíhá pomocí instalátoru NPM, což je správce JavaScriptových balíčků. Jeho instalace se liší v závislosti na použitém operačním systému. Po instalaci NPM však dále stačí napsat následující příkaz: `npm install -g @angular/cli` a instalace frameworku Angular je hotová. Nový projekt se poté vytvoří příkazem `ng new nazev_projektu`. Poté už stačí pouze spustit virtuální server příkazem `ng serve`, který od této chvíle běží na adrese `http://localhost:4200`.

2.2.5 Routování na straně frontendu

Pro zprovoznění routování je nejprve třeba v souboru `index.html` přidat do sekce `<head>` element base: `<base href="/">`. Značí to, že všechny adresy budou začínat od této cesty.

Dále je třeba naimportovat RouterModule do souboru app.module.ts. Přidá se tedy do tohoto souboru následující řádek:
import { RouterModule, Routes} from '@angular/router';.

Dále jsem vytvořil konstantí pole appRoutes, typu Routes, do kterého jsem postupně umisťoval URL adresy a příslušné komponenty, které s danou URL adresou souvisí. Jeden prvek pole může vypadat např. takto:
{ path: 'PREHLED_POSTA', component: PrehledPostaComponent},

Nakonec je toto pole třeba vložit do parametru metody forRoot objektu RouterModule do pole imports v @NgModule:
RouterModule.forRoot(appRoutes),.

Nyní už lze vytvářet url adresy a mapovat je pomocí pole appRoutes. Přešel jsem tedy do souboru app.component.html a upravil původní URL adresy, které se nacházeli v původním projektu. Není již třeba mít v url adresách text 'index.php?action=', ale stačí aby v url zůstal daný text za znakem rovná se. Takováto nyní již zbytečná předpona se nachází u většiny odkazů v app.component.html a lze ji hromadně smazat pomocí většiny editorů. Další z věcí, které bylo dále třeba změnit byl parametr href tagu a, který se změnil na parametr routerLink. Celkové rozdíly mezi url adresou původního a nového systému můžete vidět na následující ukázce:

```
Kod pred upravou:
<li><a href="index.php?action=PREHLED_POSTA" title="Prehled denni posty">
     Denni posta</a></li>

Kod po uprave:
<li><a routerLink="/PREHLED_POSTA" title="Prehled denni posty">Denni posta
</a></li>
```

Zároveň lze ze souboru prehled-posta.component.ts smazat řádek
selector: 'app-prehled-posta'
a obdobně u všech ostatních nových komponent, jelikož nebudou potřeba. Nakonec je třeba vložit tagy <router-outlet> a </router-outlet> na místo, kde chceme, aby se zobrazovaly dané šablony jednotlivých stránek. V tomto případě to bylo téměř na konec souboru app.component.html.

2.3 Rozbor automatické části převodu

Hlavní myšlenka automatického převodu stojí na tom, že dva příkazy switch ze souboru index.php budou převedeny do mnoha controllerů frameworku Symfony. Tento převod lze provést díky tomu, že v původním systému slouží příkazy switch ke směrování jednotlivých URL adres na dané podstránky systému. Pokud tedy uživatel zadá do webového prohlížeče např. adresu: nazev_domeny/PREHLED_ ADRESATU, tak následně příkaz switch vyhodnotí řetězec PREHLED_ADRESATU, který získá z proměnné $_GET['action'] a podle toho, který příkaz case odpovídá danému řetězci, vyhodnotí se výrazy uvnitř příkazu case, mezi nimiž je vždy i přesměrování na jinou stránku.

Funkci tohoto přesměrování však může plnit i `controller` v `Symfony`. A to tím způsobem, že jednotlivé vnitřní části každého příkazu `case` budou převedeny do svého `controlleru`. Spolu s nimi bude v daném `controlleru` i obsah stránky na kterou daný příkaz `case` přesměrovává. Tato stránka však bude většinou obsahovat `HTML` kód, který se přesune do frontendové části a častokrát i `SQL` kód, který se přesune do repozitářů `Doctrine`. Tento následný přesun `HTML` kódu a `SQL` kódu proběhne ručně, nicméně zkopírování vnitřních částí příkazů `case` a jejich odkazovaných stránek lze provést automatizovaně. O detailech tohoto automatizovaného přesunu se více dočtete v následující kapitole. Na následující ukázce lze na jednoduchém příkladu vidět, jaký může být rozdíl mezi příkazem `case` v původním systému a mezi výsledným `controllerem` v novém systému po manuálních úpravách.

Puvodní systém:

```
case 'PREHLED_ADRESATU':
    //volani formulare
    prehledAdresatu();
break;
```

Nový systém:

```
/**
 * Class PrehledadresatuController
 * @package AppBundle\Controller
 * RouteResource("prehledadresatu")
 */
class PrehledadresatuController extends FOSRestController implements
    ClassResourceInterface
{
    public function getAction() {
        return $this->getDoctrine()
        ->getRepository(TabAdresy::class)
        ->findPrehledAdresatu();
    }
}
```

V tomto zjednodušeném příkladu obsahuje původní příkaz `case` pouze volání funkce `prehledAdresatu()`. Tato funkce obsahuje `HTML` kód, který byl přesunut do frontendové části a `SQL` kód, který byl přesunut do repozitáře `TabAdresy` do funkce `findPrehledAdresatu()`. Kódy jsou tak úspořádány do více souborů a stávají se přehlednějšími. Celkově byl asi 350-ti řádkový `switch` přeměněn na 36 `controllerů`, což už samo o sobě naznačuje značné zpřehlednění. V případě potřeby tak nemusíme zkoumat jeden dlouhý soubor, ale stačí se podívat do potřebného `controlleru`.

To která `URL adresa` bude odpovídat kterému `controlleru` se nastavuje na dvou stranách. Jednak je to na frontendové straně, kde se ve frameworku Angular přiřazují jednotlivé `URL adresy` k určitým routovacím adresám, a dále na backendové straně v Symfony se tyto routovací adresy přiřazují k jednotlivým `controllerům`. Konkrétně se toto přiřazení provádí v souboru `app/config/routing_api.yml`, kde ke každé routovací adrese je přiřazen odpovídající `controller`, jak lze vidět na ukázce níže. Prvním parametrem je routovací adresa, druhým typ routování, což je v našem případě vždy `rest` a

třetím parametrem je cesta k odpovídajícímu `controlleru`. Tento soubor lze vytvořit automatizovaně při vytváření jednotlivých `controllerů`.

```
# app/config/routing_api.yml

prehledposta:
    type: rest
    resource: AppBundle\Controller\PrehledpostaController

prehledpostaproprijeti:
    type: rest
    resource: AppBundle\Controller\PrehledpostaproprijetiController

  ...
```

V `controllerech` jsou jednotlivé funkce označené příponou `Action` a předponami `get, post, put, patch` a `delete`, které určují typ volané akce. Název funkce může zároveň spoluutvářet název routovací adresy. V případě potřeby lze konkrétní podobu jednotlivých routovacích adres zjistit příkazem `php bin/console debug:router` v kořenové složce projektu pomocí terminálu.

Kapitola 3

Realizace

3.1 Automatická část převodu

Realizace, jak již bylo zmíněno se dělí na automatickou a manuální část převodu. V této kapitole se nejprve zaměříme na automatickou část převodu a posléze přejdeme k manuální části převodu. Pro automatický převod jsem použil analyzátor `PHP-Parser` od vývojáře vystupujícího pod pseudonymem `nikic` z webového úložiště `GitHub`. Tento alazyzátor převede kód původního projektu do controllerů, které bude třeba následně manuálně upravit.

3.1.1 Nástroj pro tvorbu scriptu

Pro začlenění analyzátoru `PHP-Parser` do vlastního scriptu, stačí includovat složku `vendor` a definovat použití potřebných tříd.

Analyzátor vytváří složitou stromovou strukturu uzlů v níž jsou znázorněny všechny informace o struktuře kódu. Pro zjednodušení práce s touto strukturou poskytuje `PHP-Parser` mnoho užitečných funkcí. Mezi ty hlavní, které jsem použil patří funkce:

- **prettyPrint()** - převede daný uzel zpět na PHP kód
- **getType()** - zjistí typ daného uzlu (deklaraci funkce, volání funkce, sekce)
- **dump()** - vypíše abstraktní syntaktický strom

Dále jsem častokrát využíval přístup ke členským proměnným tříd, které jsou součásti analyzátoru `PHP-Parser` a pomocí nichž jsem zjišťoval hodnotu a název daných uzlů.

3.1.2 Příprava před tvorbou scriptu

Převod jsem započal tím, že jsem se rozhodl spojit dva příkazy switch v souboru `index.php` a z nich poté vytvořit controlery, ke kterým bude přístup

pomocí REST API ze stejných adres, jaké byly v příkazech case původních switchů. Každý controller bude obsahovat vnitřek příkazu case a funkci, která se v příkazu case volá. Pokud se v casu žádná funkce nevolá, vznikne nový controller, který bude obsahovat vnitřek case. Každý controller bude ještě třeba upravit, aby se extrahoval HTML kód a provedla se celková úprava odpovídající standardům Symfony a objektového přístupu.

3.1.3 Tvorba scriptu

Tvorbu scriptu jsem započal tím, že jsem si pomocí programu `PHP-Parser` vytvořil asociativní pole, kde klíčem byl název funkce a hodnotou název souboru, kde se daná funkce nachází. Bylo tedy nutné projet scriptem všechny soubory, jejichž kód měl být umístěn v controllerech. Konkrétně se jednalo o soubory ve složce pages.

Dále jsem si oba zmíněné switche spojil v jeden switch a ten přesunul do samostatného souboru. Switch jsem následně analyzoval pomocí programu `PHP-Parser` a vytvořil abstraktní syntaktický strom. Ten jsem uložil do proměnné a pomocí dumpování oné proměnné zjistil, v jaké konkrétní části struktury se nachází kód přikazu switch. Poté jsem iteroval přes jednotlivé sekce příkazu `case` a pro každou jsem tvořil controller s názvem odpovídajícím názvu dané sekce a s routovací adresou shodnou s tentýž názvem. Jak vypadá část parsovacího scriptu můžete vidět na následující ukázce.

```
function addFunction($case, $functionFiles, $origCase = NULL, $name = NULL) {
    foreach($case->stmts as $line) {
        $origCase = ($origCase) ? $origCase : $case;
        if(isset($case->cond->value)) {
            $name = mb_strtolower($case->cond->value);
            $name = str_replace('_', '', $name);
        }
        if($line->getType() === 'Expr_FuncCall') {
            $key = $line->name->parts[0];
            if(array_key_exists($key, $functionFiles)) {
                $file = $functionFiles[$key];
                newController($key, $file, $upperPart, $name, $origCase);
            }
        }
        else if($line->getType() === 'Stmt_If'){
            addFunction($line, $functionFiles, $originalCase, $name);
        }
    }
}
```

Jedná se o funkci, která získává potřebná data pro vytvoření jednoho controlleru. První parametr s názvem `$case` je uzel daného příkazu `case` ze kterého se extrahují data pro vytvoření controlleru. Druhým parametrem je již zmíněné asociativní pole, které obsahuje název funkce a k němu odpovídající soubor, ve kterém se daná funkce nachází. Třetí parametr se využije dále při rekurzivním volání a obsahuje uzel příkazu `case`, pro který byla funkce původně volána. Čtvrtý parametr také slouží pro rekurzivní volání a obsahuje název daného příkazu `case`, pro který je funkce volána, protože při následném vnoření už nelze název nijak jinak získat. Na prvním řádku příkaz `foreach` iteruje přes pole `$case->stmts`, které obsahuje jednotlivé řádky daného příkazu

case. Na dalším řádku podmínka zjišťuje, jestli je proměnná case prázdná a pokud ano, přiřadí ji uzel současného příkazu case. Později se pak tato hodnota využije pro vytvoření controlleru. Na dalším řádku podmínka zjišťuje, jestli daný uzel obsahuje název controlleru. Pokud bychom byli vnořeni v rekurzi, dotazovaná proměnná $case->cond->value by neexistovala. Uvnitř podmínky je pak dále třeba název upravit, jelikož název bude použit pro routovací adresu a ta podle pravidel Symfony nemůže mít velká písmena a podtržítka. Dále následuje podmínka, která zjišťuje, jestli se jedná o volání funkce. Pokud ano, zjistí se název funkce a přiřadí do proměnné $key. Dále se pak pomocí dříve vytvořeného pole $functionFiles zjistí, zda se jedná o uživatelskou funkci, tedy zda daná funkce existuje v některém ze souborů ve složce pages. Pokud ano, zavolá se funkce newController s potřebnými daty v parametrech, která vytvoří přímo soubor s controllerem. Pokud se v dřívější podmínce nejednalo o volání funkce, ale šlo o příkaz if, zavolá se rekurzivně funkce addFunction. Pro další zkoumání je totiž třeba přistoupit do poduzlu příkazu case, což je uzel if. A právě tento poduzel se předává jako první parametr funkce addFunction.

Při tvorbě scriptu jsem zároveň vytvořil i soubor routing_api.yml, což je konfigurační soubor poskytující informace o routovacích adresách a příslušných controllerech. Soubor jsem rozšiřoval postupně, vždy při tvorbě nového controlleru.

V průběhu převodu systému jsem však zjistil, že některé sekce case příkazu switch byly nešťastně pojmenovány a sice takovým způsobem, že existovaly URL adresy PREHLED_POSTAPRIJATA a PREHLED_POSTA_PRIJATA. To mělo za následek to, že při vytváření názvu controllerů se použil stejný název pro vytvoření dvou controllerů. Díky tomu se podle očekávání prvně vytvořený controller přepsal tím, který byl vytvořen jako druhým. Bylo tedy nutné vytvořit přepsaný controller ručně. Pro přehlednost jsem novou URL adresu pojmenoval PREHLED_POSTAPRIJATAVYRIZENA, jelikož se tak jmenovala i funkce, která byla v původní sekci příkazu case volána. Controller byl pojmenován stejným názvem, ale bez podtržítka a s malými písmeny.

3.1.4 Zhodnocení výsledku automatické části převodu

Pomocí scriptu pro automatický převod bylo vytvořeno celkem 39 controllerů a soubor routing_api.yml. 9 ze zmíněných 39-ti controllerů se však ukázalo jako nepotřebných, jelikož byly později spojeny s jinými, nebo obsahovali pouze HTML kód, který byl přesunut do frontendové části. Dále v průběhu převodu bylo vytvořeno ručně už pouze 6 controllerů. Díky automatickému převodu se tak ušetřilo značné množství práce, jelikož nebylo třeba kopírovat kód z těla sekce switch do controllerů, kopírovat funkce volané z jednotlivých sekcí příkazů case do controllerů, kopírovat use segmenty controllerů a vytvářet soubor routing_api.yml.

Controllery bylo ale třeba následně upravit a to převážně z důvodu obsahujícího HTML kódu a kódu sloužícího pro práci s databází, což zabralo také značné množství času. Celkově tedy asi nebyl přínos automatického převodu takový, jaký se očekával, jelikož nějaký čas zabrala i tvorba scriptu pro automatický převod.

3.2 Manuální část převodu

Manuální část převodu začínala úpravou controllerů. Konkrétně přesunem kódu uvnitř controlleru na frontend a do repozitářů pracujících s entitami obsluhujícími databázi. Před samotnou úpravou controllerů jsem nejprve zkopíroval soubor `routing_api.yml` do složky `root_projektu/app/config`, aby fungovalo routování URL adres.

Jak controllery vypadaly po převodu lze nalézt ve složce `analyzator/controllery_po_prevodu` v přiložených souborech, nebo je lze získat spuštěním scriptu `parse.php`. Úprava mnohých controllerů byla do jisté míry obdobná. U některých bylo třeba vyjmout HTML kód a přesunout jej na frontend, u jiných šlo o převod TSQL kódu na DQL kód. Prvním controllerem, kterým jsem začal byl `adrhromadnecsvController.php`. Popíši tedy postup převodu na něm.

3.2.1 Postup při úpravě controllerů

Nejdříve je potřeba upravit neideálně strukturované formátování, které vzniklo při automatickém převodu. Hlavně zarovnání funkcí a vnitřku funkcí. Poté je třeba smazat řádek začínající příkazem case a řádek s příkazem break. Vývojové prostředí(PHP Storm) však hlásí stále nějaká varování. Prvním z nich je, že nezná konstruktor parseCSV(). Podíval jsem se tedy do původního projektu a zjistil, že konstruktor se podle očekávání nachází v souboru `parseCSV.php`, který je umístěn v kořenové složce projektu. Zkopíroval jsem jej tedy do složky src/AppBundle v novém projektu.

Soubor `parseCSV.php` však bude třeba upravit. Nejprve jsem do něj připsal namespace `AppBundle\Controller`, tedy stejný, jaký mají všechny controllery. Díky tomu je tedy možné volat funkce tohoto souboru ze všech ostatních controllerů. V souboru `parseCSV.php` však bude potřeba ještě upravit funkci `ParseData()`. Ta se však v mém prvním vybraném controlleru nevolá a proto se k této úpravě vrátím později.

Nyní přejdu zpět do controlleru `adrhromadnecsvController.php`. Vyskytuje se v něm funkce `posta_pridejAdresatHromadne()`, která obsahuje pouze HTML a JavaScript kód. Bude ji tedy třeba přesunout na frontend. Funkci, která je dočasně pojmenovaná Action, přejmenuji na postAction, protože funkce `posta_pridejAdresatHromadne()` má nejblíže k HTTP metodě POST. V controlleru zbyl nyní pouze jeden if, který bude přijímat csv soubor

získaný z požadavku ze strany frontendu. V jiných controllerech je navíc potřeba přesunout kód pro práci s databází do repozitáře, upravit jej do syntaxe DQL nebo použít Entity manager a nastavit volání těchto kódů. Některé z vytvořených controllerů se nakonec ukázaly jako nepotřebné, protože obsahovaly pouze HTML a JavaScript kódy. Takovéto controllery byly smazány a jejich kód převeden na frontend. Níže je uveden kompletní seznam převedených, či nepřevedených controllerů.

3.2.2 Převod databáze

Pro obsluhu entit sloužících pro správu databáze nabízí framework `Doctrine` dva způsoby. Prvním je použití jazyka `DQL`, který je dosti podobný jazyku `SQL`. Místo názvů tabulek se však v zápisu používají názvy entit. Druhým způsobem je přistupovat k entitám pomocí třídy zvané `Entity manager`. Tento způsob nabízí více možností a některé příkazy, jako třeba příkaz `Insert` lze provádět jen pomocí něj. Navíc tento přístup zabezpečuje aplikaci proti `SQL injection`. Jazyk `DQL` je však pro uživatele neznalé technologie `Doctrine` snažší k pochopení a jelikož je i podobný jazyku `TSQL` v němž se přistupuje k databázi v původním projektu, zvolil jsem pokud to šlo zápis příkazů v jazyku `DQL`.

Vzhledem k značné podobnosti jazyků `DQL` a `TSQL` by měl jít kód převést mezi těmito jazyky do jisté míry automatizovaně. Nenalezl jsem však žádný dostupný nástroj, který by tento převod prováděl a jelikož se to zdálo býti dosti komplikované, rozhodl jsem se s ohledem na efektivnost vynaloženého úsilí pro manuální převod.

3.2.2.1 Převod běžných operací

Pro spojování tabulek se v `Doctrine` místo cizích klíčů používají tzv. `asociace`. Fungují tak, že pokud se jedná např. o asociaci 1:N, nad proměnnou která má nahrazovat cizí klíč se napíše výraz `ManyToOne`. Prvním jeho parametrem je `targetEntity`, která udává cílovou entitu, přes kterou má asociace nastat a druhým parametrem je označeno, přes kterou proměnnou se má asociace vykonat. Dále lze použít několik volitelných parametrů, jako např. `JoinColumn(nullable=true)` který značí, že asociace má nastat, i když odpovídající entita nebude existovat.

V kódu původního systému se však vyskytovaly i některé funkce, které `Doctrine` nativně nepodporuje a bylo třeba je přidat pomocí rozšíření. Jednou z nich byla funkce `YEAR`, která z proměnné typu datetime extrahuje rok. Obdobně na tom byly i ostatní funkce pracující s datumem.

Celkově je práce s databází pomocí Doctrine jednodušší než s pomocí Open MS SQL, které bylo použito pro přístup k databázi v původním projektu. Není např. třeba používat funkce jako `mssql_free_result` nebo `mssql_close()`, protože jejich funkčnost zastává `Doctrine` automaticky.

Tabulka 3.1: Stav controllerů

Controller	**Akce**
EditaceadresatController.php	Upraven
EditacepscController.php	Upraven
PrehledadresatuController.php	Upraven
PrehledpostaController.php	Upraven
PrehledpostaodeslanaController.php	Upraven
PrehledpostaprijataController.php	Upraven
PrehledpostaproodeslaniController.php	Upraven
PrehledpostaproprijetiController.php	Upraven
PrehledpscController.php	Upraven
PresunpostajednotliveController.php	Upraven
PresunpostaprijatajednotliveController.php	Upraven
PresunpostaprijatavseController.php	Upraven
PresunpostavseController.php	Upraven
PridejadrhromadneController.php	Upraven
PridejadrjednotliveController.php	Upraven
PridejadrjednotlivedzController.php	Upraven
PridejodesjednotliveController.php	Upraven
PridejodesjednotlivedzController.php	Upraven
SmazatadresatController.php	Upraven
SmazatpostajednotliveController.php	Upraven
SmazatpostaprijatajednotliveController.php	Upraven
SmazatpostaprijatavseController.php	Upraven
SmazatpostavseController.php	Upraven
SmazatpscController.php	Upraven
UlozadresatController.php	Upraven
UlozpscController.php	Upraven
VyrizenapostajednotliveController.php	Upraven
VyrizenapostaprijatajednotliveController.php	Upraven
VyrizenapostaprijatavseController.php	Upraven
VyrizenapostavseController.php	Upraven
AdrhromadnecsvController.php	Smazán
AdrhromadnecsvuploadController.php	Smazán
AdrjednotliveController.php	Smazán
AdrjednotlivedzController.php	Smazán
NovepscController.php	Smazán
NovyadresatController.php	Smazán
OdesjednotliveController.php	Smazán
OdesjednotlivedzController.php	Smazán
PrehledpostaproodeslanipredvyberController.php	Smazán

DatabaseController.php	Vytvořen
ExportyController	Vytvořen
PredvyberdoporuceneController	Vytvořen
PredvybervseController	Vytvořen
PredvybervybraneController	Vytvořen
PrehledpostaprijatavyrizenaController.php	Vytvořen

Problém však nastane, pokud je třeba použít speciální příkazy konkrétní databáze, jako třeba v mém případě u procedury `EXEC`. Tento problém bylo nutné vyřešit přepsáním procedury `EXEC` do příkazů `Doctrine`. Stejně tak ani příkaz `VIEW` není v Symfony podporován. Dále není možné použít jazyk `DQL` pro funkci `INSERT`, ale je nutné vytvořit entitu, vložit do ní data pomocí setterů a následně ji členskou funkcí `persist` objektu `Entity manager` uložit.

3.2.2.2 Převod příkazu VIEW

Při převodu příkazu `VIEW` nešlo použít žádnou konstrukci frameworku `Doctrine`, jelikož `Doctrine` tento příkaz nepodporuje. Byly dvě možnosti, jak tento problém obejít. Buďto rozepsat příkazy `VIEW` do každého dalšího příkazu ve kterém se volají, a nebo vytvořit speciální entity, které by příkazy `VIEW` simulovaly. Druhá možnost by ale vyžadovala vytvoření příkazu `VIEW` v samotné databázi a systém by tak již byl databázově závislý. Z toho důvodu a také proto, že příkazů, které by používaly příkaz `VIEW` nebylo mnoho, jsem se rozhodl pro první řešení, tedy rozepsání příkazů `VIEW` v těch příkazech, které jej volají.

Při rozepisování jsem narazil na několik rozdílů, které se v ostatních příkazech systému nevyskytovaly. Tím prvním bylo zřetězení několika názvů sloupců spolu s mezerami či středníky. V našem případě při použití `Doctrine` se jedná o zřetězení členských proměnných dané entity. Toto zřetězení nelze provést stejným způsobem, jako tomu bylo původně v jazyku `TSQL`, ale bylo třeba použít funkci `CONCAT`. Rozdíl mezi oběma zápisy můžete vidět na následující ukázce. V druhém případě je před každou proměnnou zkratka entity, kterou je třeba v jazyku `DQL` vždy před proměnnou psát.

```
 # puvodni zapis
nazev + '; ' + psc + ' ' + posta

 # novy zapis
CONCAT(psc.nazev, '; ', psc.psc, ' ', psc.posta)
```

Další rozdíl byl už pouze u použití funkce `ltrim`. Ta slouží pro odebrání bílých znaků ze začátku řetězce. V `Doctrine` však tato funkce není podporovaná a bylo tedy nutné použít funkci `trim`, která odebírá bílé znaky i z konce řetězce. Je to tedy nepatrně méně efektivní, ale pro praktické použití by to neměl být problém.

3.2.2.3 Přepsání procedur EXEC

Dalším problémem na který jsem při převodu databáze narazil, byly procedury EXEC. Tyto procedury jsou součástí jazyka TSQL a jejich kódy jsou uloženy přímo v databázi. Jelikož ale nechceme databázově závislý systém a ani nemáme přístup do databáze, musíme si vystačit s možnostmi, které nabízí Symfony. Jedinou možností, kterou lze tedy použít je přepsání procedur EXEC přímo do repozitáře Symfony.

Kromě několika deklarativních příkazů, které jsou v původních procedurách EXEC uloženy a nyní už nejsou potřebné, je nutné zaměřit se na několik rozdílů. Jedním z nich jsou transakce. Ty mají v Doctrine výrazně jiný zápis než v jazyku TSQL. Konkrétní rozdíl můžete vidět na ukázce, kde první zápis je v jazyku TSQL a druhý v Doctrine.

```
BEGIN TRAN T1;
    ...
COMMIT TRAN T1;
```

```
$this->_em->getConnection()->beginTransaction();
try {
    ...
    $this->_em->getConnection()->commit();
} catch (Exception $e) {
    $this->_em->getConnection()->rollBack();
    throw $e;
}
```

Z ukázky lze odvodit, že Doctrine používá pro transakce členské funkce objektu Connection. Instanci tohoto objektu získáme voláním instance objektu EntityManager. Poté už můžeme používat funkce jako beginTransaction a commit v případě úspěšné transakce a v případě neúspěšné, můžeme vrátit databázi do stavu před transakcí pomocí funkce rollBack.

Dále krom rozdílů o kterých jsem hovořil dříve je v jazyku TSQL oproti zápisu Doctrine rozdíl ještě v použití cyklu. Pro ten se v jazyku TSQL používá zvláštní příkaz CURSOR. Ten lze ale v Doctrine nahradit klasickým cyklem, jako je např. cyklus for a v něm volat potřebné funkce, které byly volány v příkazu CURSOR. Poslední věcí, která byla odlyšná bylo použití funkce ISNULL. Ta má dva parametry a slouží v jazyku TSQL k tomu, že pokud hodnota v prvním parametru je NULL, výsledná hodnota bude nahrazena druhým parametrem. Tuto funkci Doctrine neposkytuje, ale nebyl příliš velký problém nahradit hodnotu NULL pomocí zápisu v jazyku PHP.

3.2.3 Vytvoření komponent frameworku Angular

Jednou z prvních věcí, které je potřeba na frontendové straně, neboli ve frameworku Angular udělat, je vytvoření komponent. Komponenta je ve frameworku Angular část programu, která tvoří jakýsi celek několika propojených souborů. Minimální verze komponenty se skládá z jednoho HTML souboru a jednoho TypeScript souboru. Další části, které může obsahovat, jsou CSS

soubor a druhý `TypeScript` soubor s příponou `spes.ts`, který slouží pro unit testy.

Komponenty lze tvořit buď ručně a nebo pomocí nástroje Angular-cli. Při druhém způsobu stačí zadat pouze příkaz `generate component` s příslušnými parametry. Jeho syntaxe pak vypadá následovně:

```
ng generate component nazevKomponenty
```

Soubory pro komponentu se poté vytvoří do složky `src/app` či případně do další podsložky, pokud byla zadána v názvu komponenty. Ve výchozím nastavení se vytvoří celkem 4 soubory. Tedy jeden `HTML`, jeden `CSS` soubor a dva `TypeScript` soubory. Nástroj také poskytuje různé přepínače, jako např `--spec false`, která zajistí, že se nevytvoří TypeScript soubor pro unit testy. Krom vytvoření komponenty také příkaz `generate` aktualizuje soubor `app.module.ts`, ve kterém jsou jednotlivé komponenty registrované.

Komponenty jsem se rozhodl vytvářet pomocí tohoto nástroje a pojmenovával jsem je názvem shodným s `URL` adresou, která je bude spouštět. Negeneroval jsem však soubor pro unit testy a vygenerované soubory sloužící pro `CSS` styl jsem smazal, jelikož nebylo třeba definovat styl pro konkrétní komponenty.

3.2.4 Vytvoření šablon

Nyní přejdu k popisu postupu vytváření šablon. Popis tvorby provedu pro stránku `prehled_posta`. Tvorba ostatních šablon byla dosti podobná.

Šablonu vytvořím tak, že zkopíruji `HTML` kód z controlleru `PrehledpostaController` a vložím jej do `HTML` souboru `prehled-posta.component.html`. Kromě `HTML` kódu obsahoval `controler` také `PHP` kód pro dotazování se nad databází, který byl následně upraven a přesunut do repozitáře.

Úprava šablony probíhá tak, že se zbylý `PHP` kód nahrazuje za tzv. Angular direktivy a doplňují se proměnné z `TypeScript` souboru komponenty. Ty lze v kódu přímo vypsat pomocí zapisu do dvojitých složených závorek, např: `{{proměnná}}`. A nebo v sekcích jako je `if` a `for` se píše proměnná pouze do uvozovek. Syntaxe direktiv je taková, že klíčová slova začínají předponou `ng` nebo `*ng` a musí být uvnitř `HTML` nějakého tagu. Někdy se způsob převodu zdál být poměrně prostý. Např. příkaz `if` z `PHP` kódu se nahradil direktivou `*ngIf`.

U příkazu `if` však bylo většinou třeba změnit i ostatní části příkazu. Např. v `REST API` není vhodné používat `session`, a nebo ve frameworku `Angular` nelze použít `if` pro nastavení `HTML` atributů, tak jako se tomu dělo v původním projektu pomocí `PHP`. Ve frameworku `Angular` totiž nelze použít příkaz `*ngIf` uvnitř `HTML` tagu. Místo toho se ale používá directiva s hranatými závorkami, která umožní přiřazovat do atributů hodnoty pomocí ternárního operátoru. Jako druhý parametr ternárního operátoru lze použít hodnotu `null` a atribut se pak nepoužije. Na následujícím příkladu je vidět rozdíl mezi zápisem kódu v původním systému a zápisem kódu v novém systému. Přičemž oba kódy mají stejnou funkčnost.

```
Puvodni system - PHP kod
<?php if(editace) { disabled = 'disabled' }

Novy system - Angular kod
[disabled]="editace ? 'disabled' : null"
```

Celkově si byly původní šablony dosti podobné. Ne však ale natolik, aby se dali nějak snadno vměstnat do jediné, která by plnila funkci více šablon. Pravděpodobně by se ale do budoucna nějaká taková úprava provést dala. Častokrát šlo totiž pouze o generování formulářů a tabulek.

3.2.5 Tvorba formulářů

Častým úkolem při tvorbě frontendové části bylo vytváření formulářů. Pro zpracování formulářů pomocí frameworku Angular existují dvě možnosti. První možností jsou formuláře typu `Template-driven` a druhou formuláře typu `Reactive`. První možnost typ `Template-driven` umožňuje jednodušší tvorbu, jeho použití produkuje méně kódu a lze použít jednoduchou validaci. Druhá možnost typ `Reactive` je naopak dobrá pro složité formuláře a poskytuje více validační logiky.

Nejprve jsem zkusil použit formuláře typu `Template-driven`. Jelikož se ale některé funkčnosti zdáli býti obtížné k vyřešení, raději jsem se rozhodl zkusit přejít na formuláře typu `Reactive`. To se brzy ukázalo jako dobrá volba, jelikož práce s tímto typem formulářů nebyla ani tak složitá, jak jsem očekával a byl jsem tedy schopen vyřešit všechny požadavky na formuláře tímto způsobem.

Kroky, které je třeba provést k vytvoření formuláře typu `Reactive`, můžete vidět na následující ukázce pro třídu NovepscComponent.

```
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

export class NovepscComponent implements OnInit {
    myForm: FormGroup;

    constructor(private http: HttpClient, private fb: FormBuilder) {
    }

    ngOnInit() {
        this.myForm = this.fb.group({
            psc: ['', [
                Validators.required,
                Validators.minLength(3),
            ]],
            posta: ['', [Validators.required, ] ],
        });
    }

    ...
```

Nejprve je třeba naimportovat příslušné třídy z balíčku `@angular/forms`. V tomto případě se jedná o třídy `FormBuilder`, `FormGroup` a `Validators`. Na prvním řádku třídy si deklarujeme třídní proměnnou `myForm` typu `FormGroup`. Ta bude později obsahovat všechna data formuláře. V konstruktoru si pomocí nástroje zvaného `dependency injection` definujeme třídní proměnnou `http`, která bude sloužit pro komunikaci s rozhraním `REST API` a proměnnou `fb`,

která bude obstarávat propojení mezi formulářovými prvky v `HTML` kódu a proměnnými v `TypeScript` kódu. Dále ve funkci `ngOnInit`, která se volá po inicializaci všech objektů v konstruktoru můžete vidět definování názvů formulářových prvků a jejich požadovaných vlastností. Funkce `group` vyžaduje jako parametr objekt. Ten má proměnné `psc` a `posta`, které obsahují pole. První hodnotou pole je výchozí hodnota odpovídajícího formulářového prvku. Zde je u obou proměnných prázdná, ale hodí se např. využít v případech, kdy chceme nějaká data editovat. Druhou hodnotou pole je pole validátorů. Na příkladu lze vidět validátor `required`, který označuje formulářový prvek jako povinný a validátor `minLength`, který určuje minimální délku řetězce v daném formulářovém prvku.

Jak vypadá následné propojení s `HTML` kódem lze vidět níže.

```
<form [formGroup]="myForm" [hidden]="formSentOk">
    <input formControlName="psc">
    <td class="alert alert-danger" *ngIf="psc.touched && psc.invalid">
        Polozka psc nemuze byt prazdna! Minimalni delka je 5 znaku
    </td>
    <button type="submit" [disabled]="myForm.invalid" (click)="sendData()"
        title="Ulozit PSC">
</form>
```

Na prvním řádku formulář obsahuje direktivu `[formGroup]`. Ta má jako parametr již dříve zmíněnou třídní proměnnou `myForm` typu `FormGroup`. To zapříčiňuje, že následný prvek `input` může být propojen s objektem `myForm`, pomocí atributu `formControlName` a parametru odpovídajímu proměnné tohoto objektu. Na prvním řádku kódu lze dále vidět direktivu `[hidden]`, která zapříčiní, že pokud proměnná v jejím argumentu bude vyhodnocena jako `true`, formulář se v prohlížeči skryje. Toho lze využít např. po odeslání formuláře.

Dále v `HTML` tagu `td` je direktiva `*ngIf`, která zapříčiní, že pokud je splněna podmínka v jejím parametru, celý tag `td` se zobrazí. Podmínka v tomto případě zjišťuje z členských proměných objektu `psc`, zdali bylo na element spjatý s tímto objektem kliknuto a zároveň není validní. Tyto proměnné se nastavují díky frameworku Angular automaticky. Aby k nim však bylo možné v direktivě `*ngIf` přistupovat, je v `TypeScript` souboru nutné vytvořit pro ně `get` metody. Jak taková `get` metoda vypadá, můžete vidět v kódu níže.

```
get psc() {
    return this.myForm.get('psc');
}
```

V dalším elementu `button` direktiva `[disabled]` zapříčiňuje, že na tento element nelze kliknout, pokud je splněna podmínka v argumentu. Tou podmínkou je zde členská proměná `invalid` objektu `myForm`, která se nastaví na `true`, pokud alespoň jedno z pravidel v nastavených validátorech není splněno.

Z ukázky je patrné že validace se provádí pomocí `Angular` directiv a kódu v `TypeScript` souboru ihned při vyplňování jednotlivých formulářových prvků. Oproti tomu v původním systému tato validace probíhala pomocí `JavaScript` kódu umístěného v `HTML` souboru a to až při odeslání formuláře.

Posledním z prvků, které jsou v ukázce HTML kódu součástí frameworku Angular je akce (click). Ta spouští funkci v TypeScript souboru, která je uvedena v argumentu akce. Funkce může mít také argumenty, ale zde nebyly třeba. Jak vypadá implementace této funkce můžete vidět v kódu níže.

```
sendData() {
    const postArray = this.myForm.getRawValue();

    this.http.post(environment.ROOT_URL + '/ulozpscs', postArray).subscribe(
        (data: any) => {
            this.formSentOk = true;
        },
        (err: HttpErrorResponse) => {
            this.formSentError = true;
        }
    );
}
```

Na prvním řádku funkce se do proměnné postArray přiřazují data. Ty se získají z objektu myForm voláním členské funkce getRawValue. Tyto data jsou v objektu ihned po vyplnění příslušných polí, což je zapříčiněno funkcionalitou Angularu Two-way data binding. Dále toto pole vložíme do funkce post objektu HttpClient, přičemž prvním parametrem funkce post je routovací url adresa, na kterou budou data odeslána. Routovací adresa se zde skládá z proměnné ROOT_URL, což je tzv. proměnná prostředí a je umístěna v souboru environment.ts. Proměnná prostředí slouží k tomu, abychom jednu hodnotu nemuseli definovat na více místech, což by způsobovalo problémy s udržovatelností. Dále pak lze pomocí metody subscribe získat odpověď ze serveru, zdali se požadavek POST zdařil či nikoliv. Poté lze např. nastavit odpovídajícím způsobem příslušné proměnné, které mohou zobrazit informativní hlášku o úspěchu či neúspěchu odeslání formuláře, či třeba skrýt formulář, jak bylo vidět dříve v ukázce HTML kódu. Tyto proměnné opět díky funkcionalitě Two-way data binding zapříčiní automatické skrytí formuláře či zobrazení hlášky okamžitě, hned jak jsou nastaveny na příslušnou hodnotu true nebo false.

3.2.6 Generování PDF souborů

Generování PDF souborů je jednou z věcí, která byla dosti odlišná od ostatních požadavků, které většinou pouze získávali, nebo ukládali data do databáze a proto stojí za to se zde o ní zmínit.

3.2.6.1 Generování na backendové straně

V původním systému je samotné generování PDF souborů napsané v PHP kódu. A proto pro co nejsnazší převod a tedy zachování co největšího množství původního kódu jsem musel zvolit možnost generovat PDF soubor na backendové straně pomocí Symfony a výsledný soubor poté odeslat na frontendovou část pomocí rozhraní REST API.

V původním systému se pro generování každého `PDF souboru` používala samostatná třída, která svými funkcemi určovala co v které části `PDF` souboru bude zobrazeno. Tyto třídy vždy dědily od třídy `FPDF`, která obsahuje všechny potřebné funkce k samotnému generování `PDF` souboru. Tento způsob generování pomocí několika tříd bylo možné použít i v novém systému s tím, že se muselo několik věcí upravit.

První taková úprava bylo přepsání veškerého kódu do třídních funkcí. V původní systému bylo totiž na konci souboru třídy několik řádků kódu, které se automaticky volali při inkludování tohoto souboru. Tyto řádky jsem vždy částečně přepsal do nové funkce třídy v které se původně nacházeli a částečně do `controlleru` ze kterého jsem poté volal i onu funkci.

Dále bylo třeba vytvořit správně konstruktor u nadtřídy `FPDF`, jelikož jeho starý zápis, který se používal v `PHP` dříve již nešel použít. Rozdíli mezi starým a novým zápisem můžete vidět na ukázce níže.

```
# stary zapis
function FPDF($orientation='P', $unit='mm', $format='A4')

# novy zapis
public function __construct($orientation='P', $unit='mm', $format='A4')
```

Další úpravou, kterou bylo potřeba učinit, bylo nahrazení globální proměnné za třídní proměnnou. Použití globálních proměnných není obecně dobrou praxí, což se ukázalo i zde, kdy globální proměnná podtřídy přepisovala proměnnou nadtřídy.

Poslední část úpravy na straně backendu spočívala v převodu kódování. V některých případech text generovaný do `PDF` souboru obsahoval nesprávné znaky a musel tak být převeden. Převod probíhal pomocí `PHP` funkce `iconv` ze znakové sady `UTF-8` do znakové sady `windows-1250`, jelikož s touto znakovou sadou pracovala nadtřída `FPDF`.

Nakonec bylo třeba `PDF` soubor odeslat jako odpověď na požadavek `REST API`. Pro odeslání souboru přes toto rozhraní stačí v Symfony vytvořit objekt `BinaryFileResponse`, tomu při vytvoření zadat do konstruktoru jako parametr název souboru s plnou cestou a tento objekt následně vrátit jako odpověď.

```
return new BinaryFileResponse($fileName);
```

3.2.6.2 Stažení PDF souboru

Pro stažení `PDF` souborů jsem vytvořil zvláštní službu, jejíž zápis můžete vidět na následující ukázce.

```
downloadFile(url): Observable {
    const responseType = {responseType: 'blob'};
    return this.http.get(url, responseType).map(res => res);
}
```

Za zmínku stojí nastavení typu odpovědi v proměnné `responseType`. Zkratka `blob` značí `Binary large object`, tedy libovolný objekt binárních dat.

Dále jsem vytvořil funkci `downloadPdf`, která jako parametr přijímá koncovou část routovací `url` adresy a stáhne `PDF` soubor, který získá jako odpověď při zaslání požadavku na zvolenou routovací adresu. Její implementaci můžete vidět na ukázce níže.

```
downloadPdf($url: string) {
    this.download.downloadFile(environment.ROOT_URL + $url).subscribe(
        data => {
            let downloadLink = document.createElement('a');
            downloadLink.href = window.URL.createObjectURL(new Blob([data],
                {type: 'application/pdf'}));
            document.body.appendChild(downloadLink);
            downloadLink.click();
        },
        (err) => {
            console.log("Download Error:", err);
        }, () => {
            console.log("Download done");
        }
    )
}
```

Na prvním řádku funkce se volá dříve zmíněná služba `downloadFile` s routovací adresou v parametru. Aby bylo možné `PDF` soubor stáhnout, vytvoří se `HTML` element `a`, který bude sloužit jako odkaz. Data získaná z `REST API` se vloží do pole a poté do konstruktoru objektu `Blob`. Dále se nastaví typ dat, což je `pdf`. Následně se vytvořený link přidá do těla `HTML` kódu a klikne se na něj, díky čemuž se požadovaný soubor stáhne.

3.2.7 Autentizace

Jednou z nejdůležitějších částí na tvorbě nového systému bylo zajištění autentizace. V původním systému probíhalo ověřování uživatelů pomocí `session`, což ale v novém systému při použití REST API není možné. Pro zajištění zabezpečené komunikace bylo nutné prováděn ověření pomocí tzv. `tokenů`. Unikátní `token` získá uživatel po úspěšném přihlášení a identifikuje se s ním při zasílání každého požadavku.

3.2.7.1 Autentizace na backendové straně

Pro autentizaci na straně `Symfony` je nejběžnější použít balík `FOSUserBundle`. Ten lze snando nainstalovat z oficiální dokumentace Symfony[16]. Není tedy třeba zde postup instalace příliš detailně rozebírat. Dobré je ale podle mého názoru zmínit, že krom klasické instalace balíčku bylo třeba vytvořit entitu `User`, která sloužila pro ukládání autentizačních údajů uživatelů do databáze. Dále bylo třeba upravit konfigurační soubory: `security.yml`, `config.yml` a `routing.yml`. Nakonec pak aktualizovat `Doctrine` schéma pro vytvoření tabulky `user`.

Dále bylo třeba nainstalovat balíček `LexikJWTAuthenticationBundle` z webového úložiště `GitHub`[17], který slouží pro získávání tokenu. V průběhu instalace jsem však zjistil, že verze Symfony 3.3 již není podporovaná pro tento `Bundle` a byl jsem proto nucen aktualizovat Symfony na verzi 3.4. Aktualizace

proběhla zcela bez potíží. Po instalaci balíčku je dále třeba vytvořit soukromý a privátní `ssh` klíč. Tyto klíče je pak třeba uložit do složky projektu a nastavit k nim cesty do konfiguračního souboru `config.yml`. Nakonec je pak třeba upravit soubor `security.yml`, ve kterém lze nastavit, která routovací adresa bude sloužit pro přihlášení, která bude veřejně dostupná a která bude vyžadovat ověření pomocí tokenu.

Následně už stačí pouze vytvořit uživatele. K tomu lze použít terminál, jehož použití je velmi snadné. V základním provedení se stačí přepnout do kořenové složky projektu a zadat následující příkaz.

```
php bin/console fos:user:create testuser test@example.com p@ssword
```

Místo řetězce `testuser` se zapíše konkrétní uživatelské jméno pro daného uživatele, za jménem konkrétní uživatelův email a heslo.

3.2.7.2 Autentizace na frontendové straně

K zprovoznění autentizace na frontendové straně bylo nejprve nutné vytvořit stránku přihlášení. Ta obsahovala přihlašovací formulář, po jehož vyplnění se zavolala funkce `authentication`. Její definici můžete vidět níže.

```
authentication() {
    const credentials = this.myForm.getRawValue();
    let data = {
        "username" : credentials['name'],
        "password" : credentials['password'],
    }

    this.http.post(environment.ROOT_URL + '/api/login_check',data).subscribe(
        (data: any) => {
            localStorage.setItem('userToken', data.token);
            localStorage.setItem('username', credentials['name']);
            this.router.navigate(['']);
            location.reload();
        },
        (err: HttpErrorResponse) => {
            this.loginError = true;
        }
    );
}
```

Nejprve se získají data z formuláře a nastaví se do objektu `data` ve formátu, jaký vyžaduje REST API. Následně se tyto data odešlou `post` metodou na adresu `/api/login_check`, kde proběhne ověření uživatele. Pokud byly data, která uživatel do formuláře zadal chybná, nastaví se proměnná `loginError` na hodnotu `true`, což způsobí zobrazení chybové hlášky v odpovídajím elementu `HTML` kódu. Pokud byly odeslané údaje správné, v proměnné `data` nalezneme jako odpověď REST API uživatelův token, který bude později uživatelem využíván pro jeho ověření při každé komunikaci s REST API. Nyní token uložíme do tzv. `localStorage`, což je úložiště v prohlížeči klienta. Z tohoto uložiště budeme token později při každém požadavku získávat, abychom nemuseli stále posílat požadavek na REST API, což by bylo dost neefektivní. Dále pak uživatele přesměrujeme na výchozí stránku webu a znovu ji načteme. Tím se

přenačte komponenta `app.component`, která tak zpřístupní položky v menu, které byly před přihlášením nedostupné.

Dále jsem vytvořil třídu `LoginInterceptor`, která zajistí, že každý požadavek odeslaný na REST API bude obsahovat hlavičku s uživatelským tokenem. V této třídě je nejdůležitější funkce `intercept`, jejíž definici můžete vidět níže.

```
intercept (req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<
    any>> {
    const token = localStorage.getItem('userToken');
    const authReq = req.clone({
        headers: req.headers.set('Authorization', 'Bearer ' + token)
    });

    return next.handle(authReq).pipe(
        tap(event => {
            }, error => {
                if (error.status === 401) {
                    console.log(error);
                    localStorage.removeItem('userToken');
                    this.router.navigate(['/LOGIN']);
                }
            }
        )
    )
}
```

Ve funkci se nejprve získává uživatelův token z úložiště `localStorage` a ten se poté nastaví do hlavičky pod označením `Authorization`. Spolu s tokenem se do hlavičky přidá předpona `Bearer`, která je vyžadovaná od REST API. Následně pokud je požadavek na REST API neúspěšný, např. kvůli chybějícímu, nebo expirovanému tokenu, token se odebere z úložiště `localStorage` a uživatel bude přesměrován na stránku přihlášení.

Aby však třída `LoginInterceptor` mohla takto fungovat, je nutné ji zaregistrovat jakožto poskytovatele do souboru `app.modelu.ts` do sekce `@NgModule`. Konkrétní zápis můžete vidět níže.

```
providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: LoginInterceptor,
    multi: true
}],
```

Závěr

Cílem této bakalářské práce bylo vytvořit webový systém pro správu firemní listové pošty. Systém byl tvořen transformací stávajícího systému s důrazem na odstranění návrhových nedostatků. Transformace probíhala semi-automatickým způsobem za pomocí nových technologií. Nový systém byl rozdělen na backendovou a frontendovou část, komunikující mezi sebou přes REST API. V práci bylo dále pojednáno o analýze stávajícího systému a o návrhu řešení pro převod do systému nového.

Cíl práce se podařilo splnit, jelikož systém splňuje všechny funkční požadavky, které splňoval i systém původní. Přínosem této práce jsou poznatky zjištěné při semi-automatickém převodu původní systému do systému postaveného na nových technologiích.

Budoucí možné pokračování této práce spočívá v napsání automatizovaných testů, které by usnadnili pozdější rozvoj aplikace. Dalším možným vylepšením se nabízí, aktualizovat frameworky Symfony a Angular na jejich nejnovější verze.

Literatura

[1] How Angular Protects Us From XSS Attacks?. Dor Moshe [online]. Aug 8, 2017 [cit. 2019-01-02]. Dostupné z: https://dormoshe.io/articles/how-angular-protects-us-from-xss-attacks-19

[2] ORM Security. Doctrine [online]. [cit. 2019-01-02]. Dostupné z: https://www.doctrine-project.org/projects/doctrine-orm/en/2.5/reference/security.html

[3] Základní principy a pojmy objektově orientovaného programování [online]. [cit. 2018-12-31]. Dostupné z: http://pehapko.cz/oop/uvod

[4] Java tutoriál - Objektově orientované programování (3. díl) [online]. [cit. 2018-11-26]. Dostupné z: http://programujte.com/clanek/2007043001-java-tutorial-objektove-orientovane-programovani-3-dil/

[5] What is the actual expansion for OOPs? [online]. [cit. 2019-01-02]. Dostupné z: https://www.quora.com/What-is-the-actual-expansion-for-OOPs

[6] Single Page Application (SPA) [online]. [cit. 2018-12-22]. Dostupné z: https://rychlikoderi.cz/single-page-application-spa/

[7] PHP-Parser/doc/0_Introduction.markdown [online]. [cit. 2018-12-30]. Dostupné z: https://github.com/nikic/PHP-Parser/blob/master/doc/0_Introduction.markdown

[8] 1. díl - Úvod do Symfony frameworku pro PHP [online]. [cit. 2019-01-09]. Dostupné z: https://www.itnetwork.cz/php/symfony/zaklady/uvod-do-symfony-frameworku-pro-php

[9] Six good reasons to use Symfony [online]. [cit. 2018-02-16]. Dostupné z: https://symfony.com/six-good-reasons

[10] Databases and the Doctrine ORM [online]. [cit. 2019-01-01]. Dostupné z: https://symfony.com/doc/3.4/doctrine.html

[11] How to Work with Doctrine Associations / Relations [online]. [cit. 2019-01-01]. Dostupné z: https://symfony.com/doc/3.4/doctrine/associations.html

[12] Začínáme s AngularJS [online]. [cit. 2018-02-16]. Dostupné z: https://www.zdrojak.cz/clanky/zaciname-s-angularjs/

[13] Dependency Injection: motivace [online]. [cit. 2018-02-16]. Dostupné z: https://www.zdrojak.cz/clanky/dependency-injection-motivace/

[14] 1. díl - Úvod do AngularJS [online]. [cit. 2018-02-16]. Dostupné z: https://www.itnetwork.cz/javascript/angular/javascript-tutorial-uvod-do-angularjs/

[15] The Release Process [online]. [cit. 2018-02-16]. Dostupné z: https://symfony.com/doc/current/contributing/community/releases.html

[16] FOSUserBundle. Symfony [online]. [cit. 2019-01-09]. Dostupné z: https://symfony.com/doc/current/bundles/FOSUserBundle/index.html

[17] Lexik JWT Authentication Bundle. GitHub [online]. [cit. 2019-01-09]. Dostupné z: https://github.com/lexik/LexikJWTAuthenticationBundle

Příloha A

Seznam použitých zkratek

PHP Hypertext Preprocessor

MVC Model, View, Controller

REST REpresentational State Transfer

API Application Programming Interface

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

JSON JavaScript Object Notation

ORM Object-Relational Mapping

URL Uniform Resource Locator

Obsah přiloženého flash disku

```
readme.txt ........................... stručný popis obsahu flash disku
src
  impl ................................. zdrojové kódy implementace
    backend ........................ implementace backendové části
    frontend ........................... implementace frontend části
    parser ................................ implementace analyzátoru
  thesis .................... zdrojová forma práce ve formátu LaTeX
    images ................................ obrázky použité v práci
text ................................................... text práce
  thesis.pdf ......................... text práce ve formátu PDF
```