Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

Migrace komplexních projektů mezi web MVC frameworky

Bakalářská práce

Autor: Pavel Vosyka

Školitel: Ing. Martin Čížek, MBA

České Budějovice 2018

Vosyka P., 2018: Migrace komplexních projektů mezi web MVC frameworky [Migrating complex projects between web MVC frameworks, Bc. Thesis, in Czech], Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Bakalářská práce se zabývá migrací komplexního softwarového projektu na nový framework v témže programovacím jazyce, který uplatňuje stejný hlavní návrhový vzor jako framework původní. Vyhodnocuje teoretické i praktické aspekty takové migrace a na části konkrétního projektu demonstruje uplatnění zvoleného postupu migrace ze Zend Frameworku na framework Symfony 3. Práce ukazuje, jak v aplikaci zajistit koexistenci obou frameworků v období přechodu, během něhož musí stále probíhat vývoj nových požadavků. V práci jsou také popsány odlišnosti v uplatňování návrhových vzorů těmito frameworky.

Klíčová slova: Migrace, aplikace, MVC, framework, PHP, Zend, Symfony

Anotation

The bachelor thesis deals with migration of a complex software project to a new framework in the same programming language and with the same main design pattern as the original framework. Theoretical and practical aspects of such migration are evaluated and a part of a concrete project is used to demonstrate the chosen migration process from the Zend Framework to the Symfony 3 framework. The thesis shows how two frameworks can coexist within the application during the transition period when new features have to be delivered continuously. The thesis also describes the differences in how design patterns are applied in both frameworks.

Keywords: Migration, application, MVC, framework, PHP, Zend, Symfony

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby stejnou elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne: …… podpis autora: ……

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Martinu Čížkovi, MBA a týmu Orchitech za trpělivé vedení, cenné rady a čas, který věnovali mé práci. Také děkuji mé rodině, zejména mé shovívavé manželce Kláře, za podporu během studia a během doby dokončování této práce.

Obsah

1 Úvod

1.1 Motivace k migraci

Počítač by měl sloužit svému pánu, ne naopak. Nedílnou součástí takového počítače je software, který splňuje požadavky uživatele. Požadavky a potřeby uživatele se však často mění, a tak se musí vyvíjet i software. A vyvíjet se musí i kvůli požadavkům, které se zdánlivě nemění – například požadavek, aby byl software bezpečný. Bezpečnostní hrozby totiž v průběhu času přibývají.

Dalo by se říct, že téměř všechen software je závislý na jiném softwaru, například na sdílených knihovnách, frameworcích nebo API. Oč více to platí u velkých komplexních projektů! Tento podpůrný software se však také vyvíjí. A čas od času se směr vývoje změní natolik, že vznikne nová verze, která nahradí tu starou. Také se ale stává, že se vývoj zastaví úplně. Je to běžné zakončení životního cyklu softwaru, kterému se v angličtině říká „End-Of-Life“.

V lepším případě je pak software ještě nějakou dobu podporován, alespoň vzhledem k bezpečnostním problémům. Pokud se to stane s knihovnou nebo frameworkem, na kterém je projekt postaven, měl by co nejdříve přejít na jinou podporovanou verzi nebo jiný podporovaný produkt, jinak je vystavován nově vznikajícím bezpečnostním rizikům. Přechod na novou verzi však také často přináší další výhody jako jsou nové funkce, které mohou zjednodušit vývoj a údržbu projektu. Tato práce se zabývá konkrétním příkladem projektu, který se do uvedené situace dostal.

1.2 Cíle práce

Cílem práce je zhodnotit praktické i teoretické aspekty přechodu mezi různými MVC frameworky u komplexního projektu, pokud tyto frameworky používají stejný programovací jazyk a hlavní návrhový vzor.

Dílčí cíle práce jsou:

1. V teoretické části seznámit čtenáře s použitými návrhovými vzory a principiálními důvody pro přechod z jednoho frameworku na druhý.

2. V rešeršní části popsat, jak se frameworky nad stejným programovacím jazykem a uplatňující stejné návrhové vzory prakticky liší. Hodnocenými frameworky jsou Zend Framework a Symfony 3
3. V praktické části navrhnout postupy přechodu a zhodnotit je z pohledu nákladů, čistoty návrhu a udržitelnosti dalšího rozvoje projektu.
4. Vybrat optimální přístup a vytvořit "demo" přechodu pro vybranou část projektu.

Autor po domluvě zadavatelem striktně neodděluje zadanou strukturu práce, ale prolíná teoretickou, rešeršní a praktickou část v kapitolách jinak strukturovaných.

S principiálními důvody k migraci na jiný framework je čtenář seznámen v úvodu práce. Používané návrhové vzory jsou popsány v kapitole druhé.

Ve třetí kapitole autor navrhuje dva základní postupy migrace.

Detailní odlišnosti v uplatňování návrhových vzorů jsou frameworků Zend a Symfony 3 jsou popsány v čtvrté a páté kapitole. V těchto kapitolách také vybírá optimální přístup a vytváří funkční „demo".

Zhodnocení různých praktickických a teoretické aspektů přechodu se prolíná celou prací a je v jejím závěru shrnuto přehlednou tabulkou.

1.3 Význam a aktuálnost zvoleného tématu

Migrace mezi frameworky, nebo alespoň mezi jednotlivými verzemi frameworků, je dlouhodobě zajímavé téma. V dnešní době se projekty staví na frameworcích. Ty se ale vyvíjí a mají jen omezenou dobu podpory. Ta u „dlouho podporovaných" verzí trvá obvykle přibližně 3 roky. S migracemi se tedy počítá. Autor v tom vidí velkou příležitost uplatnění jako programátor na projektech, které vyžadují aktuální verze podpůrných projektů.

1.4 Metodologie použitá při tvorbě bakalářské práce

V jednotlivých částech práce jsou použity v různé míře různé metody.

V úvodu práce začíná autor explanací, tedy vysvětlením problému, který nastal – čili, že skončila podpora pro klíčový podpůrný software. Při hledání obecného postupu pro přepis aplikace je využita analogie, protože se předpokládá, že postupy, které uplatnili jiní autoři s podobnými problémy, budou úspěšně uplatněny i při řešení problémů této práce.

Ve druhé kapitole je čtenář metodou popisu seznámen s projektem IGO Admin 2017. Při hledání postupů pro přepis projektu je hojně používána analýza, kdy je projekt rozložen na menší části, například moduly, controllery, jejich akce a konečně jednotlivé řádky kódu a jejich funkce, které se také analyzují.

Některé oblasti jsou řešeny experimentálně, kdy se záměrně ovlivňuje prostředí a pozoruje chování aplikace – například pokud jde o sessions. Způsoby řešení jiných oblastí jsou vyjádřeny popisem nebo modelováním formou zjednodušeného nákresu.

1.5 Teoretická východiska práce a další rešeršní činnost

1.5.1 Přehled a rozbor literatury

V oblasti informačních technologií je publikováno citelně méně odborné literatury než v jiných vědních oborech. Přesto autor našel dostatek zajímavých podkladů pro zpracování práce.

Nejčastěji používaným pramenem se staly oficiální dokumentace a návody na webu frameworku Symfony. Pro pochopení fungování stávajícího frameworku byla občas použita jeho oficiální dokumentace[1], ale mnohdy bylo rychlejší a přínosnější nahlížet do samotného zdrojového kódu frameworku a kódu projektu.

Velmi zajímavé bylo zjištění, že obecnou otázkou přepsání celé aplikace se zabývali programátoři už před dlouhými léty a své zkušenosti publikovali v článcích na svých webech. Jde například o autory: Martin Fowler, Paul Hammant a Joel Spolsky.

Užitečné byly také články autorů, kteří v nedávných letech stručně publikovali své konkrétní zkušenosti z integrace frameworku Symfony do stávajícího projektu. Byli to autoři: Yannick de Lange a Carlos Buenosvinos.

Zmíněné zdroje jsou dostupné pouze v anglickém jazyce. Jejich volně přeložené citace jsou uvedené v samotné práci a odkazy v kapitole Seznam literatury.

1.5.2 Anglické a další výrazy

Přesto, že je práce psána v češtině, dovolil si autor v práci používat některé anglické výrazy. Věří totiž, že to bude užitečnější pro případné další badatele, protože české překlady by mohly být matoucí. Jde zejména o slovo „controller“. Jeho asi nejvýstižnější český překlad

[1] https://framework.zend.com/manual/1.10/en/manual.html

je zřejmě řadič, ne však kontroler nebo kontrolor, jak by se dalo očekávat. Použití slova „řadič“ by ale mohlo být matoucí v souvislosti s návrhovým vzorem model-view-controller, a tak je v práci používán pouze anglický výraz.

Dále jde například o slova jako:

- Framework
- Model
- View
- Rafactoring
- URL
- request (taktéž česky požadavek)

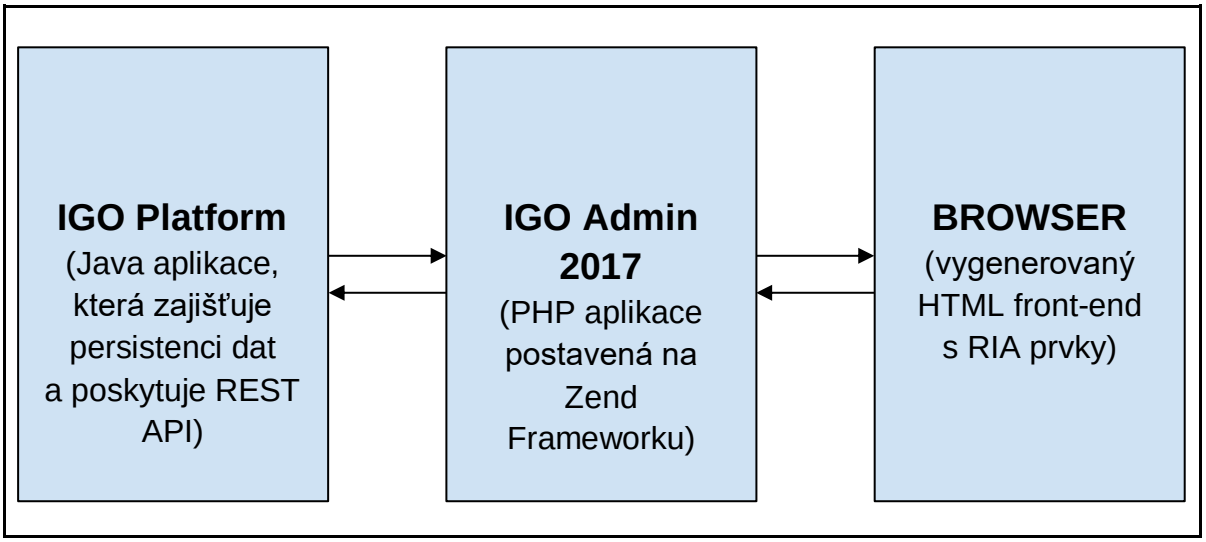
Autor též předpokládá, že čtenář má povědomí o těchto výrazech. V této práci nejsou detailně vysvětleny.

2 Seznámení s projektem IGO Admin 2017

Tato část popisuje konkrétní aplikaci, u které nastala potřeba změnit framework právě z důvodu, že původní framework přestal být podporován, a to včetně bezpečnostních aktualizací.

2.1 Specifika projektu

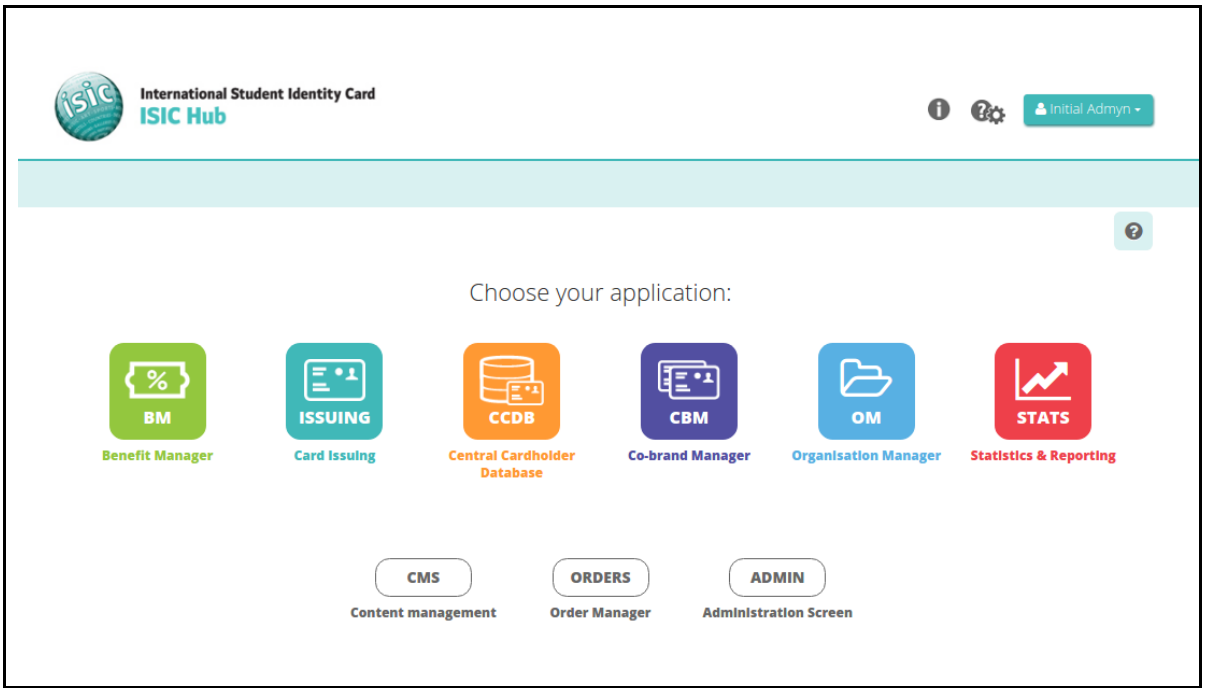
IGO Admin 2017 je webová aplikace napsaná v jazyce PHP a tvoří pouze část projektů ISIC. Jejím úkolem je zajišťovat pohodlné rozhraní pro administraci oprávněným uživatelům. Je neobvyklá tím, že nekomunikuje přímo s databází. Persistenci dat obstarává jiná samostatná webová aplikace napsaná v Javě (IGO Platform), která vystavuje REST API. Tohoto API aplikace IGO Admin 2017 využívá ke komunikaci, a je na něm plně závislá. To znázorňuje následující nákres.



Nákres specifické komunikace projektu IGO Admin 2017

2.2 Komplexita projektu

Projekt je vlastně jednotným rozhraním pro několik jiných aplikací. Na následujícím snímku obrazovky je vidět 9 základních modulů, které aplikace má.



Obrazovka s výběrem aplikace

Ve skutečnosti obsahuje ještě 5 dalších modulů (default, geo, rest, campaigns, vm). Každý modul má obvykle své controllery, pohledy, modely, formuláře apod. Z následující tabulky je možné si udělat představu o velikosti projektu. Jsou v ní uvedeny pouze soubory specifické pro každý modul, projekt pak obsahuje, kromě knihoven, ještě několik dalších desítek souborů.

Název složky modulu	Počet souborů	Počet řádků
admin (ADMIN 2017)	49	3 974
bm2 (Benefit Manager 2017)	127	14 929
campaigns	84	10 097
ccdb2 (Central Cardholder DB 2017)	136	11 683
cms (CMS 2017)	111	8 872
cobrand (Cobrand Manager 2017)	53	6 663
Default	181	14 111

geo	18	993
oi (Card Issuing 2017)	235	24 689
om (Organization Manager 2017)	88	9752
orders (ORDERS 2017)	208	23 071
rest	8	271
sales (Statistics & Reporting 2017)	69	4 674
vm	6	183
CELKEM ve složce modules	**1 373**	**133 962**

Tabulka s počtem souborů a řádků kódu v jednotlivých modulech z roku 2017.[2]

Na základě těchto informací si autor práce dovoluje označit projekt za komplexní.

2.3 Frameworky a doba podpory

Aplikace v současnosti používá první generaci frameworku Zend. V roce 2016 bylo oznámeno[3], že 28. září téhož roku skončí jeho životnost produktu (tzv. End-of-Life).

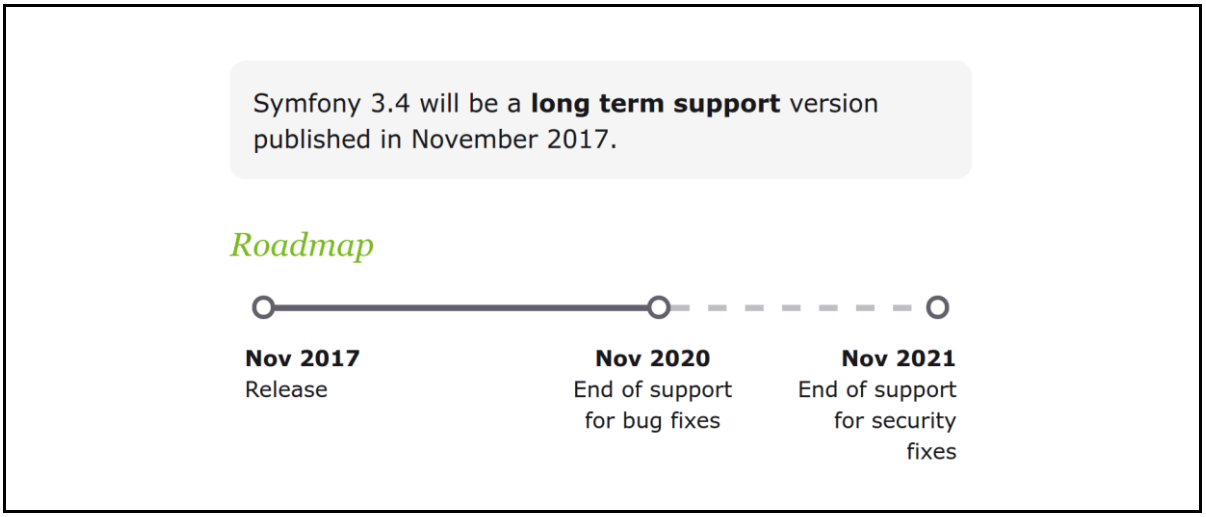
Tým Orchitech navrhl Symfony jako vhodného kandidáta pro nový framework. V době psaní práce byla uvolněna dlouho podporovaná verze[4] 3.4. Konec životnosti je plánován na listopad 2021.[5] To znamená 3 + 1 rok podpory. Navíc je na webu Symfony slíbeno[6], že upgrade na další verze bude snadnější. Tvůrci se prý poučili z předchozích verzí.

[2] Údaje měřeny v dubnu 2018. Odpovídají však hlavní vývojové větvi k datu 7.3. 2017. Měřeno v programu PhpStorm s pomocí doplňku Statistic. V počtu řádků jsou zahrnuty i komentáři či případné prázdné řádky.

[3] O'PHINNEY, Weier. Zend Framework 1 End-of-Life Announcement. In: Zend Framework: Blog [online]. [neuvedeno]: Weier O'Phinney, 2016 [cit. 2018-04-05]. Dostupné z: https://framework.zend.com/blog/2016-06-28-zf1-eol.html

[4] tzv. LTS, Long-Time Support

[5] POTENCIER, Fabien et al. Symfony Roadmap. In: Symfony [online]. [neuvedeno]: Potencier, 2016 [cit. 2018-04-05]. Dostupné z: https://symfony.com/roadmap?version=3.4



Obrázek s přehledem doby podpory nového frameworku z webu Symfony.[7]

2.4 Použité návrhové vzory

Návrhové vzory jsou způsobem řešení často opakovaných problémů a úloh, které je potřeba řešit nehledě na použitý programovací jazyk nebo framework. Zend i Symfony 3 používají jako hlavní návrhový vzor MVC.

2.4.1 MVC – Model View Controller

Autor plně souhlasí s výstižnou definicí uvedenou na české wikipedii. „*Model-view-controller (MVC) je softwarová architektura, která rozděluje datový model aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent tak, že modifikace některé z nich má jen minimální vliv na ostatní.*“[8]

2.4.2 Front controller

Oba frameworky používají také návrhový vzor „front controller“. Tento controller předchází všem ostatním controllerům aplikace. Z jednoho místa obstarává společné prvky controllerů, což je pohodlnější, než je obstarávat v každém controlleru zvlášť.

---

[6] POTENCIER, Fabien et al. Our Backward Compatibility Promise. In: Symfony [online]. [místo neuvedeno]: Potencier, 2013 [cit. 2018-04-05]. Dostupné z: https://symfony.com/doc/3.4/contributing/code/bc.html

[7] viz pozn. 5

[8] Model-view-controller. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2005-. Dostupné také z: https://cs.wikipedia.org/wiki/Model-view-controller

Mnohé frameworky pro jazyk PHP, včetně Zendu, používají jako front controller soubor „index.php“. Symfony tento soubor nazývá „app.php“. A pro striktní odlišení prostředí produkčního a vývojového používá také soubor „app_dev.php“. Ať už se soubor jmenuje jakkoli, vždy je potřeba zajistit, aby na něj webový server směroval potřebné požadavky (většinou všechny, kromě například .js, .css, .jpg, .png).

2.4.3 DI – Dependency Injection

Jeden ze způsobů, jak řešit závislosti na jiných částech programu, uvádí návrhový vzor Dependency Injection, a Symfony jej využívá. Zend jej ve své první verzi nepoužívá a závislosti řeší pomocí klíčového slova new (např. `new MyDependency()`).

Princip DI velmi srozumitelně popsal David Grudl:

„Dependency Injection (dále jen DI) neboli zřejmé předávání závislostí říká: odeberte třídám zodpovědnost za získávání objektů, které potřebují ke své činnosti.

Budete-li psát třídu vyžadující ke své činnosti databázi, nevymýšlejte uvnitř jejího těla, odkud ji získat (tj. ze žádné globální proměnné, statické metody, singletonu, registru atd.), ale požádejte o ni v konstruktoru nebo jiné metodě. Popište závislosti svým API. Nebudete muset tolik přemýšlet a získáte srozumitelný a předvídatelný kód.“[9]

Třída tedy ideálně ve svém konstruktoru popíše své závislosti. Framework třídě následně obstará závislosti, které potřebuje (tzv. autowiring). Tato technika vede k čistšímu a lépe testovatelnému kódu.

[9] GRUDL, David. Co je Dependency Injection?. In: PhpFashion [online]. Praha: Grudl, 2012 [cit. 2018-04-05]. Dostupné z: https://phpfashion.com/co-je-dependency-injection

3 Obecně o kompletním přepisu

Migrace na jiný framework je velký zásah do projektu. Téměř každá část kódu totiž bývá se stávajícím frameworkem pevně svázána. Pro zbavení se této vazby je nutné přepsat téměř celý projekt. To je riskantní úkol, protože přepis bude trvat dlouhou dobu. Proto je dobré poučit se od zkušených vývojářů. Obecně je možné vydat se k přepisu celého projektu dvěma způsoby.

3.1 The Big Rewrite – začít s čistým štítem

První variantou je jednorázový přepis, tzv. Big Rewrite. Znamená to pozastavit průběžný vývoj aplikace, celý kód napsat od znovu, a ve chvíli, kdy je vše hotové a funkční, nahradit novým kódem starou aplikaci. Tímto způsobem není vhodné postupovat u velkých projektů, kde by přechod trval dlouhou dobu, což by mohlo mít negativní dopad z ekonomického a obchodního hlediska.

Joel Spolsky[10] v roce 2000 publikoval svůj možná nejznámější článek (na jeho blogu je umístěn na prvním místě v kategorii TOP 10). Jmenuje se „Věci, které byste nikdy neměli dělat, část I.“. Uvádí tam nešťastné zkušenosti z vývoje internetového prohlížeče Netscape. Rozhodnutí přepsat celý kód od začátku, považuje za osudovou chybu. Přepis trval téměř tři roky a během té doby nevyšla žádná nová verze prohlížeče. Souběžně s přepisem totiž nebylo možné věnovat se vývoji nových funkcionalit. Navíc byly do projektu opětovně zaneseny již dříve opravované chyby. Mezitím je prý konkurence předběhla a podíl jejich produktu na trhu klesl.

V tomto a dalších svých článcích[11] popisuje, že programátoři mají tendence svůj kód dokola přepisovat a že s ním nikdy nejsou spokojení. Také popisuje svůj názor, proč tomu tak je. Nakonec uznává, že některé části kódu je dobré přepsat, ale varuje před jednorázovým odhozením celého kódu. Doporučuje vyhledat části, které například aplikaci zpomalují, a tak i jen jejich zlepšení bude mít velký efekt. Také radí starý kód refactorovat, což je proces

[10] SPOLSKY, Joel. Things You Should Never Do, Part I. In: JOEL ON SOFTWARE [online]. New York City: Spolsky, 2000 [cit. 2018-04-05]. Dostupné z: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

[11] SPOLSKY, Joel. Don’t Let Architecture Astronauts Scare You. In: JOEL ON SOFTWARE [online]. New York City: Spolsky, 2001 [cit. 2018-04-05]. Dostupné z: https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/

krátce popsaný dále. Zanášení nové architektury do projektu, nebo už jen zabývání se jí, považuje za ztrátové, pokud nepřináší žádnou novou funkci pro uživatele.

Velký důraz také klade na včasné doručení produktu[12] (nebo nové verze aplikace). Upřednostňuje dostatečně dobrý produkt doručený včas před produktem s dokonale napsanou vnitřní architekturou. A tuto myšlenku se snaží předat svým zaměstnancům a vůbec všem programátorům.
„You're not here to write code; you're here to ship products." (J. Spolsky)

3.2 Rewriting není refactoring

Zmíněný refactoring se dá definovat[13] jako proces aplikace sady malých úprav kódu bez vnější změny chování programu. Při této pečlivé prací není funkční kód zahozen a záhy nahrazen novým. Jde spíše pouze o přemístění nebo i jen přejmenování toho starého.

Existuje několik způsobů pro refactoring, které změní uspořádání kódu a pomohou k jeho lepší čitelnosti, přičemž funkčnost by po každém kroku neměla být nikterak změněna. Tyto postupy jdou značně snadněji provádět pomocí IDE. Cílem práce není více se rozepisovat o detailech refactoringu. Je zde zmíněn, aby bylo zřejmé, že jde o něco jiného, než je přepsání celé aplikace od začátku.

Během přepisování je nutné myšlenkově pronikat do problematiky veškerých funkcionalit. Proto je z hlediska vyrovnání nákladů velmi žádoucí funkcionalitu zlepšit či rozšířit.

Jak je tedy možné přepsat veškerý kód a současně během přepisování držet krok s průběžným vývojem a uvolňováním nových funkcionalit?

3.3 StranglerApplication – postupný přepis aplikace

Martin Fowler, uznávaný vývojář, který sepsal mnoho architektonických návrhových vzorů, v roce 2004 sepsal jeden právě pro postupný přepis aplikace[14]. Nazval ho „StranglerApplication", v překladu tedy „škrtící aplikace".

---

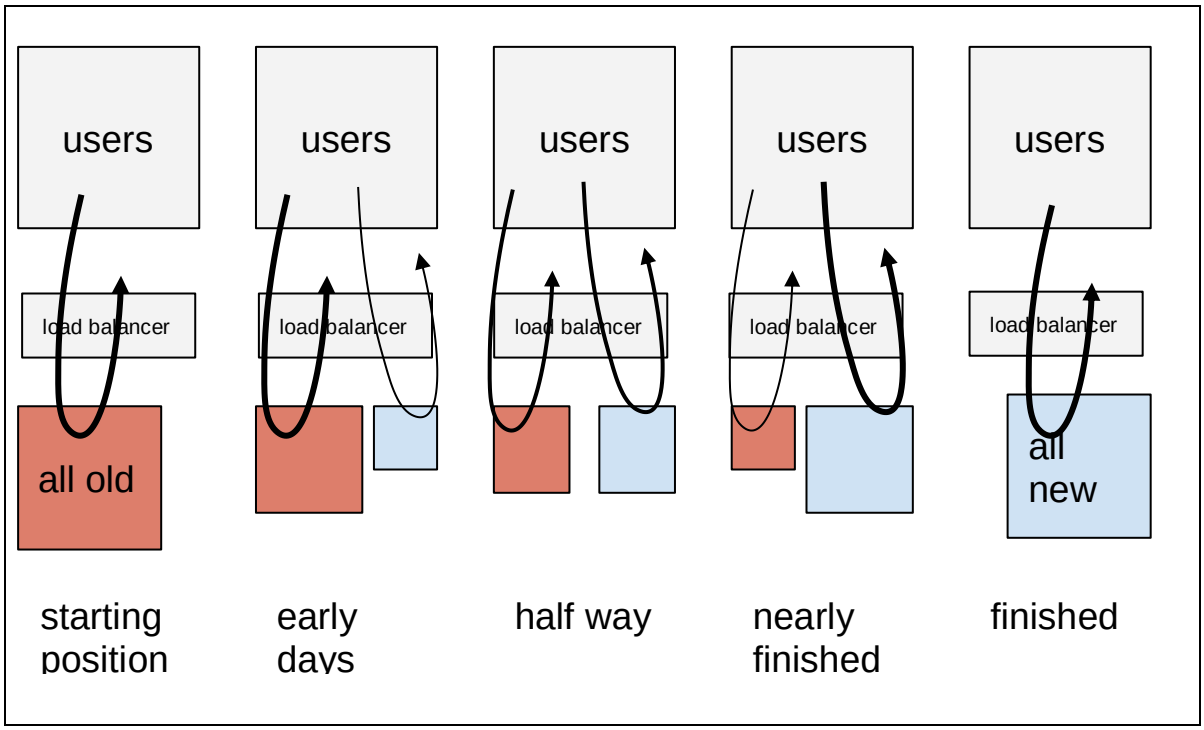
[12] SPOLSKY, Joel. The Duct Tape Programmer. In: JOEL ON SOFTWARE [online]. New York City: Spolsky, 2009 [cit. 2018-04-05]. Dostupné z: https://www.joelonsoftware.com/2009/09/23/the-duct-tape-programmer/

[13] FOWLER, Martin a Kent. BECK. Refactoring: improving the design of existing code. Reading, MA: Addison-Wesley, 1999. ISBN 978-0201485677.

Jde o metaforu s rostlinným škrtičem, který začíná růst na větvích stromu, spouští kořeny až dolů do země, a přitom jimi obepíná hostitelský strom. Původní strom je postupem času uškrcen, ztrouchniví a zůstane pouze škrtič. Podobně při postupném přepisování touto formou mají dlouhou dobu koexistovat obě aplikace až do doby, kdy ta nová plně nahradí starou.

Paul Hammant v roce 2013 publikoval článek,[15] ve kterém se na tento vzor odvolává a popisuje několik případových studií postupného přechodu na novou aplikaci. Případy jsou to velmi různorodé. Popisuje například přechod na aplikaci sjednocující více aplikací do jedné, přechod na úplně jiný programovací jazyk a zmiňuje i přechod na jiný framework v rámci jednoho jazyka.

Na obrázku trefně znázorňuje, jak nově vznikající aplikace nahrazuje tu starou. Do cesty jim postavil „Load Balancer“ – prvek, který zajišťuje správné směrování požadavků (ve webovém prostředí jde o URL adresy) na starou nebo novou aplikaci.



Nákres procesu, kdy nová aplikace postupně nahrazuje starou, až ji nahradí kompletně. [16]

---

[14] FOWLER, Martin. StranglerApplication. In: Martin Fowler [online]. Chicago: Fowler, 2004 [cit. 2018-04-05]. Dostupné z: https://www.martinfowler.com/bliki/StranglerApplication.html

[15] HAMMANT, Paul. Legacy Application Strangulation : Case Studies. In: Paul Hammant's blog [online]. New York: Hammant, 2013 [cit. 2018-04-05]. Dostupné z: https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/

[16] Překresleno z obrázku v článku P. Hammanta (viz pozn. 15).

3.3.1 Doporučení z desetileté praxe

Nakonec doporučuje 6 bodů pro úspěšný růst nové aplikace:

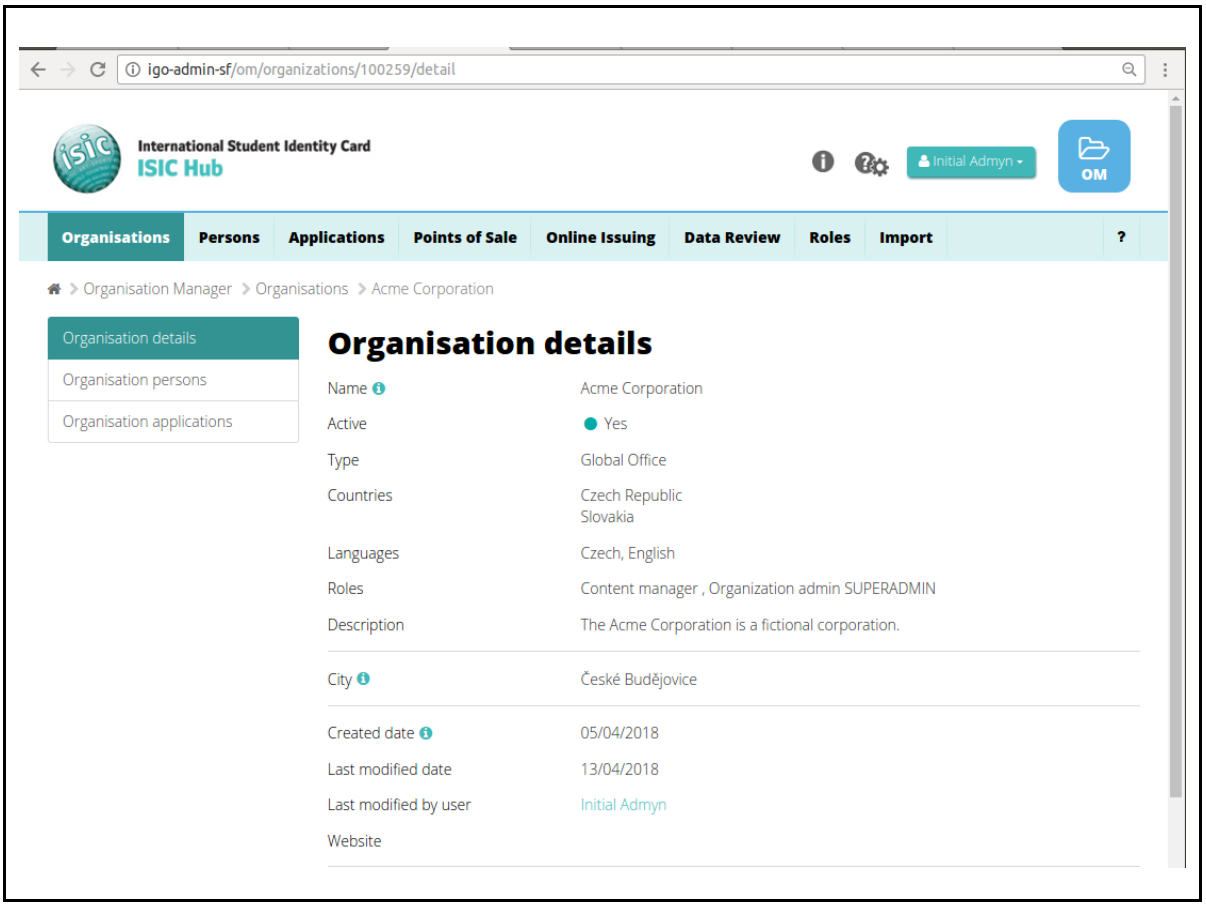
1.) Proces uškrcení si rozfázujte a během něho udržujte aplikaci stále nasaditelnou. První kus práce nasaďte zhruba do měsíce, a poté další části nejdéle po 2 týdnech. V opačném případě projekt nebude mít úspěch, pravděpodobně pro ukončení ze strany sponzora.
2.) Nové funkce nebo zlepšení musí jít ruku v ruce s přepisováním. Během přepisu věnujte podstatnou část času práce těmto zlepšením. Díky tomu budou moci ti, kdo migraci financují, práci považovat za hodnotnou. Pro tyto lidi obvykle není informace o tom, že se přestala používat nějaká technologie, která již pozbyla své životnosti, dostatečným argumentem. Celý produkt se během migrace musí znatelně zlepšovat. A všichni – od vedení po programátory – o této potřebě musí vědět a souhlasit s ní.
3.) Integrační a funkční testy jsou pro úspěch klíčové. Je potřeba zajistit, aby pohyb uživatele mezi starou a novou aplikací fungoval bez problému. Automatizované testy pomohou předejít nepříjemným překvapením.
4.) Pochopte, že obecné nekonkrétní požadavky (typu stabilita, rozšiřitelnost, udržitelnost, bezpečnost...) mohou ohrozit celou iniciativu k migraci. Členové týmu s oprávněním rozhodovat mohou mít oblíbené technologie a různé názory na to, co je dobré použít. To je zkouškou pro celý vývojový tým – zda se shodne nebo ne.
5.) Pracujte agilně. S klasickým vodopádem to nepůjde – trvalo by to dlouho.
6.) Uvědomujte si, že během migrace existuje riziko ztráty nějaké zapomenuté, ale přitom potřebné funkčnosti.

3.4 Vybraný postup pro „demo“ přechod projektu IGO Admin

Projekt IGO Admin se stále aktivně vyvíjí. Migrace na nový framework musí probíhat tak, aby vývoj nebyl pozastaven. Z toho důvodu se jako vhodnější řešení zdá přepis postupný.

Cílem „dema“ je najít postup a zjistit náročnost přechodu na generačně novější framework a z úplně jiné dílny – z první verze Zendu na Symfony ve verzi 3 – v poměrně rozsáhlé aplikaci. Celá aplikace využívá mnoho již nepodporovaných Zend komponent, které se v nové aplikaci používat nebudou.

Půjde o přechod jen části jednoho modulu aplikace, což by mělo stačit k navržení určitého směru, kterým by se mělo postupovat podobně u dalších částí. Konkrétně jde o akci „detail“ v controlleru „organizations“ v modulu „Organizations Manager“. Cílem není kompletně je nahradit, ale spíše zjistit, jakými problémy je nutné se během přechodu zabývat.



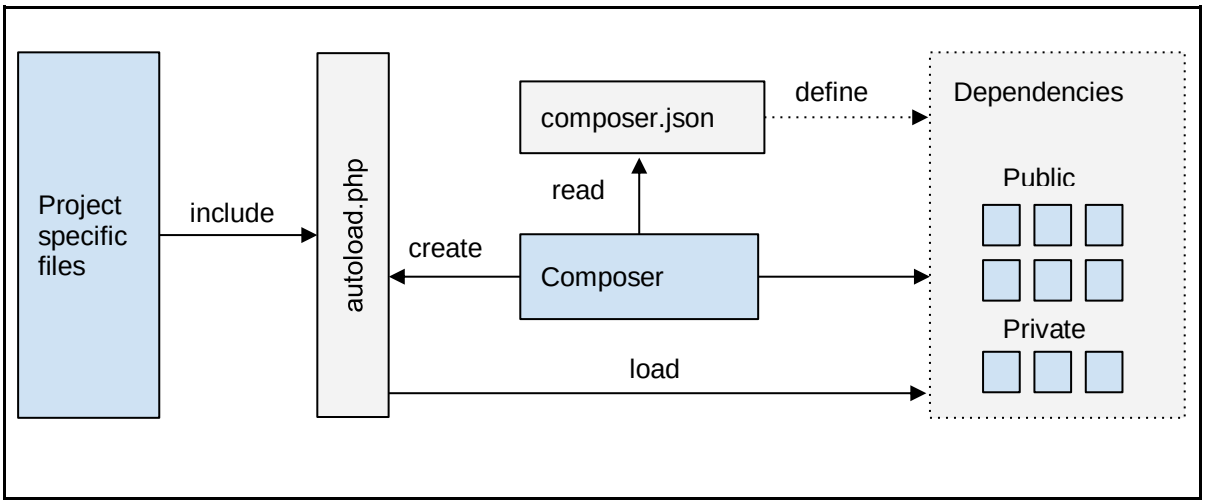
Snímek obrazovky detailu organizace ve výchozím frameworku.

4 První kroky k přepisu

Tato kapitola se zabývá zajištěním souběžné existence obou frameworků. Jsou zde uplatněny některé doporučené postupy z předchozí části.

4.1 Instalace frameworku a sdílených knihoven

Pro jazyk PHP je v dnešní době standard spravovat závislosti přes Composer[17]. Tento nástroj si přečte seznam závislostí v konfiguračním souboru composer.json a jedním příkazem je stáhne nebo aktualizuje. Kromě toho vytváří soubor autoload.php, který stačí zahrnout do projektu (příkazem `include "vendor/autoload.php";`), a všechny závislosti jsou v tu chvíli načtené a připravené k použití. Funkci composeru popisuje následující nákres.



Nákres principu fungování nástroje Composer.

I nový framework Symfony se instaluje přes tento balíčkovací systém. Yannick de Lange, který na svém blogu radí, jak z frameworku Zend 3 přejít na Symfony 3, uvádí jako první radu zajistit, aby starý projekt uměl pracovat s composerem.[18]

Současný projekt, přestože composer používá k instalaci Zend Frameworku, ale své privátní závislosti přes composer spravované neměl. Jsou spravované ve vlastním repozitáři, takže se stahovaly ručně do složky library, v kódu byl pro ně zajištěný autoloading. S composerem byl autoloading vyřešen snadno pomocí zanesení složky „library/“ do classmap.

[17] https://getcomposer.org/

[18] DE LANGE, Yannick. Migrating your project to Symfony. In: Stovepipe Systems [online]. Amsterdam: Lange, 2016 [cit. 2018-04-06]. Dostupné z: https://stovepipe.systems/post/migrating-your-project-to-symfony

```
    "autoload": {
        "classmap": [
            "library/"
        ]
    }
```

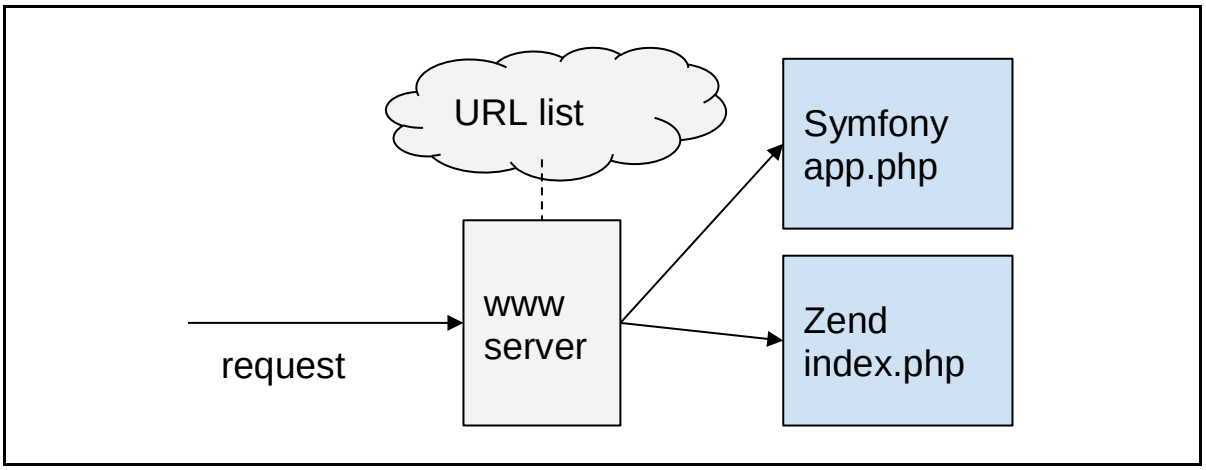
Kód souboru composer.json.

4.2 Směrování URL adres správným controllerům

Zachovat funkční URL adresy je nesporný požadavek[19]. Během přepisu se bude rozšiřovat seznam nových controllerů, které nahrazují ty původní. Je tedy potřeba vybrat vhodnou techniku pro směrování, která bude oním „load balancerem“ z předchozí kapitoly.

4.2.1 Teorie tří nalezených postupů

Směrování je, dá se říct, první krok ke zpracování požadavku. Carlos Buenosvinos publikoval článek o tom, jak si s tím poradili ve společnosti Atrápalo.[20] Rozhodli se tento klíčový krok činit již na úrovni webového serveru (v jejich případě Apache). Předhodili mu seznam adres pro nové controllery, které pak server směroval na vstupní místo Symfony (app.php), ostatní na index.php – vstupní místo starého frameworku.

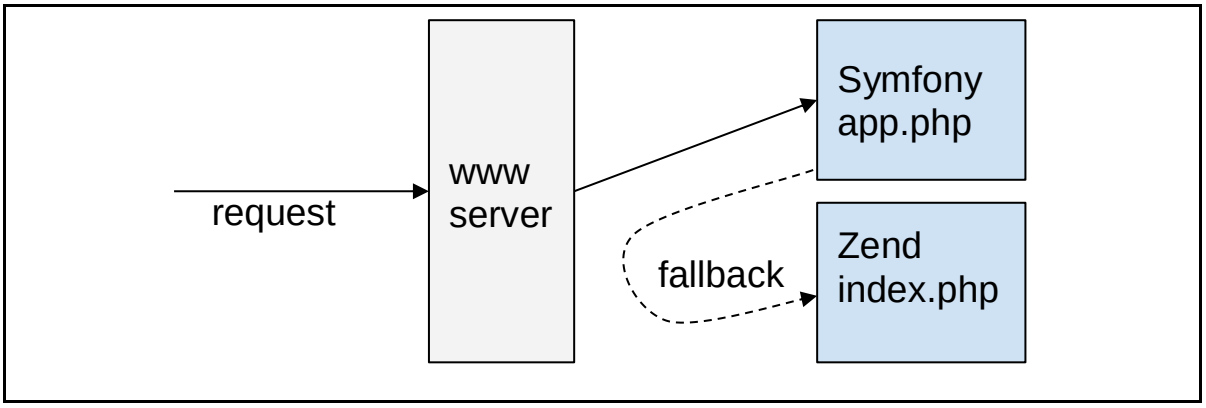


Nákres řešení směrování na úrovni webového serveru.

[19] KNESL, Jiří. Problémy při přepisu aplikace na novou verzi. In: Jiří Knesl [online]. Praha: Knesl, 2014 [cit. 2018-04-06]. Dostupné z: http://www.knesl.com/problemy-pri-prepisu-stareho-systemu

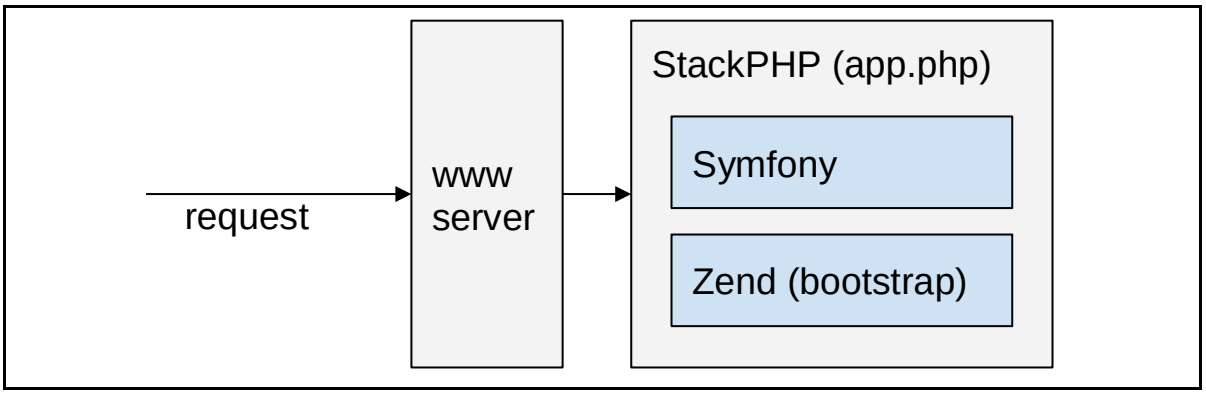
[20] BUENOSVINOS, Carlos. Migrating progressively to Symfony without pain. In: Carlos Buenosvinos [online]. Barcelona: Buenosvinos, 2015 [cit. 2018-04-06]. Dostupné z: https://carlosbuenosvinos.com/migrating-progressively-to-symfony-without-pain/

Jinou technikou je upřednostnění nového frameworku. Popsal jí Yannick de Lange, který už byl výše zmíněn. Pokud nový framework nerozpozná URL požadavek, spustí starý framework a zobrazí jeho výsledek. Je to rozhodně pohodlnější způsob, než řešení na úrovni webového serveru. Na to ostatně později přišel i zmiňovaný Carlos Buenosvinos.



Nákres řešení směrování s fallbackem ve frameworku Symfony.

Již 4 měsíce po vydání svého prvního článku totiž popsal, nedostatky minulého řešení[21]. Spočívaly zejména ve zbytečné složitosti generovat celý soubor adres po každé nově definované routě. Nový návrh spočíval v řešení směrování až na aplikační úrovni, nicméně s využitím dekorátoru StackPHP[22]– tedy prvkem, který přebere zodpovědnost za správné směrování. Ten však vyžaduje aby bootstrap starého frameowrku implementoval rozhraní HttpKernelInterface. To sice není příliš složité, ale vyžadovalo by to zásah do starého frameworku. To by také nemusel být problém, ale autor zatím nebyl přesvědčen o výhodách tohoto řešení.



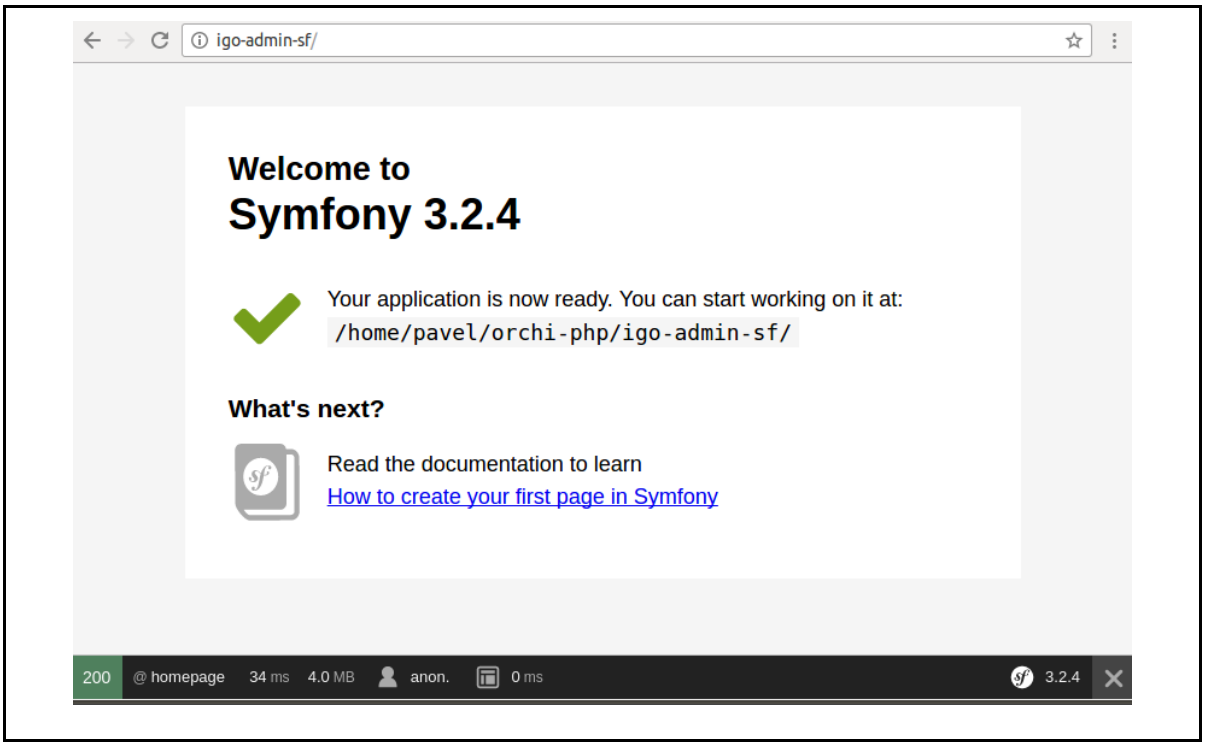
Nákres řešení směrování pomocí StackPHP.

[21] BUENOSVINOS, Carlos. Migrating progressively to Symfony without pain with StackPHP. In: Carlos Buenosvinos [online]. Barcelona: Buenosvinos, 2015 [cit. 2018-04-06]. Dostupné z: https://carlosbuenosvinos.com/migrating-progressively-to-symfony-without-pain-with-stackphp/

[22] https://github.com/stackphp

4.2.2 Vstupní bod a jeho implementace

Výše vybrané řešení vyžaduje směrování všech požadavků na vstupní bod frameworku Symfony. Tím je v produkci soubor app.php (symfony má pro vývojové prostředí zvlášť oddělený vstupní bod, jímž je app_dev.php). Aplikace IGO Admin je provozována na serveru Apache. Pro změnu výchozího vstupního bodu bylo potřeba nastavit v souboru .htaccess nasměrování na soubor app_dev.php. Pokud se nyní aplikace spustí, zobrazí se uvítací stránka Symfony.



Snímek obrazovky s výchozí stránkou Symfony frameworku.

4.2.3 Implementace a ukázka controlleru

Uvítací stránka se zobrazila proto, že ve standardní distribuci Symfony je výchozí AppBundle. Jak bude v praxi implementován vybraný postup s fallbackem? Vystačí dva kusy kódu. V konfiguraci routeru frameworku bude definována jako poslední routa ta, která odpovídá jakýmkoli možným požadavkům, a ta bude směrovat na fallback controller.

```
# This is fallback for any url, that was not specified above.
# It will call the legacy bundle, which starts Zend framework 1.
fallback:
    path: /{path}
    defaults: { _controller: "legacy.controller.fallback:fallback" }
    requirements:
        path: .*
```

Fallback controller, má spustit framework Zend. Původní front controller frameworku Zend (soubor index.php) se v téměř nezměněné podobě stává obsahem tohoto fallbacku.

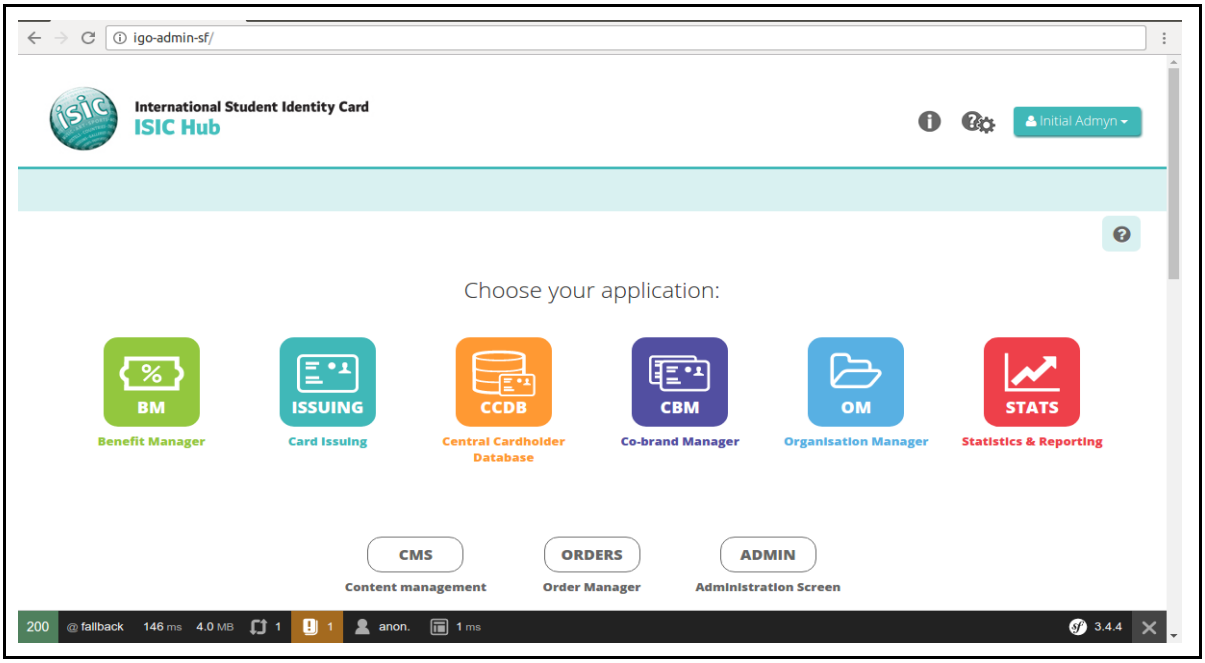
```
/**
 * @Route(service="legacy.controller.fallback")
 */
class FallbackController extends Controller
{
    private $webDir;

    public function __construct($webDir, ContainerInterface
$container)
    {
        $this->webDir = $webDir;
        // Give a Symfony container to the legacy app
        Container::init($container);
    }

    /* Inspired by Zend index.php */
    public function fallback(Request $request, $path)
    {
        // ...
        // Turn on output buffering before startting Zend
application
        ob_start();
        // Create application, bootstrap, and run
        $application = new Zend_Application(
                APPLICATION_ENV,
                APPLICATION_PATH . '/configs/application.ini'
                );
        $application->bootstrap()->run();
        // Get content from buffer and send it as Symfony Response
        $output = ob_get_contents();
        ob_end_clean();
        return new Response($output);
    }
```

Dále byla přepsána routa ukázkové stránky Symfony, a framework se tak po zadání hlavní stránky pokusil přesunout na hlavní stránku IGO Admin. Nastal při tom sice problém se Sessions, ale ten byl pro zatím vyřešen oficiálním postupem, který bude dále popsán.

Poté už se zobrazila stránka staré aplikace. Spodní debugger lišta připomíná, že nad touto aplikací běží Symfony.



Snímek obrazovky s úvodní stránkou IGO Admin 2017 při fallbacku ze Symfony na Zend.

4.3 Sessions a session_start()

Sessions, neboli relace, jsou nezbytným prvkem webových aplikací. Díky nim může aplikace udržet stav, byť je protokol HTTP bezstavový.

V jazyce PHP se práce se sessions spouští příkazem session_start() a na starost si to obvykle berou frameworky. Problémem je, že se příkaz session_start() smí spustit pouze jednou. Oba frameworky ale chtějí být prvními, kdo spustí sessions, a hlásí chybu, pokud se tak nestane.

4.3.1 PhpBridge

Symfony počítá s možností integrace frameworku do již hotové aplikace a nabízí oficiální cestu, jak tuto situaci řešit[23] – phpBridge. Toto řešení bylo uplatněno již výše, jde o rychlou změnu parametrů:

```
framework:
    session:
        storage_id: session.storage.php_bridge
        handler_id: ~
```

[23] http://symfony.com/doc/current/session/php_bridge.html

Ze začátku se vše zdálo být funkční. Tento bridge počítá s tím, že sessions bude spouštět stará aplikace. Pak je ale nutné zajistit, aby se sessions spustily i v případě, že se stará aplikace nespouští (tedy i pokud se nespustí fallback). To autor s určitým časovým odstupem vyhodnotil jako špatné řešení a komplikovalo mu to vývoj. Nakonec se ponořil do zkoumání Zend_Session hlouběji a hledal možnost, jak spustit Zend_Session s již dříve volaným session_start. A našel.

4.3.2 Hack Zend_Session

Zend_Session téměř vždy spouští sessions sám. Jediná výjimka je s veřejnou proměnnou $_unitTestEnabled. Při hodnotě true si Zend uvědomuje, že není první a že je zahrnut do nějakého testovacího frameworku, který sessions startuje dříve. Pak se dá bez chyb volat Zend_Session:start(). Při tomto volání si Zend uloží, že už jsou sessions spuštěny a pro předejití vedlejších efektů se proměnná $_unitTestEnabled nastaví zpět na false. To by snad mohl být způsob, jak se se sessions úspěšně vypořádat.

```
/**
 * Hack for Symfony Zend_Session
 * With this hack Symfony can already start the sessions.
 * It is probably the only one way to work with Zend_Sesions
 * without letting him be the one who starting the sessions.
 */
private function startZendSessionAlthoughSessionAlreadyStarted() {
    Zend_Session::$_unitTestEnabled = true;
    Zend_Session::start();
    Zend_Session::$_unitTestEnabled = false;
}
```

A autor toho využil. Hack se spouští při fallback controlleru těsně před spuštěním Zend aplikace. Díky tomu pak zbytek aplikace funguje, i když sessions spustil Symfony (a ten je může spouštět normálně bez jakéhokoli bridge). V jiných případech se hack nepoužívá. Pokud bychom ze Symfony volali nějaké komponenty, které jsou se Zend_Sessions svázány, hlásily by chybu, protože už sessions odstartoval někdo jiný (právě Symfony).

To je nakonec dobře, protože to programátora vede k tomu, aby při přepisování nepřetahoval staré věci do nové části kódu.

4.4 Bezpečnost

Bezpečnost je klíčová pro celý projekt. Je vlastně i podstatným důvodem pro celou migraci. Kontrola oprávnění se ve starém frameworku provádí na všech úrovních:

- Controllery (formou anotací) – pokud uživatel nemá oprávnění, controller ho dále nepustí.
- Modelová vrstva – předem kontroluje, kdo se ptá, ale přitom ani nemá možnost získat data, pokud k nim přihlášený uživatel nemá přístup. IGO Platform by je neposkytl.
- Vrstva view – podle oprávnění uživatele se upravují některé části šablony, aby uživatel ani neviděl, že existují nějaké odkazy, pokud k nim nemá oprávnění.

V projektu je 934 míst, kde se volá Default_Model_SecurityUtils, což je třídá, která zajišťuje bezpečnost projektu. Je potřeba to tedy vymyslet co nejlépe. Nabízí se 3 varianty:

A) Zpřístupnit starou security komponentu novým kontrolorům jako službu.
B) Napsat novou komponentu a tu všem starým částem předat jako službu.
C) Napsat novou komponentu, která bude s tou starou sdílet stejnou datovou strukturu, uloženou v Symfony sessions.

Varianta A byla během vývoje zamítnuto, protože by se znovu zanášela závislost na Zend komponentách. Varianta B by mohla být zbytečně složitá. Tým Orchitech navrhl a zvolil variantu C. V rámci „dema“ ale nebyla implementována.

5 Přepisování po částech

Doposud byla popsána fáze, kdy je nový framework nainstalován a oba jsou zdánlivě připraveni k vzájemnému soužití. Nastává fáze růstu nové aplikace. První přepisovaný controller má před sebou ještě těžkou cestu. Je sice již spustitelný, ale jeho proces zahrnuje mnoho činností, k nimž mu ještě nikdo cestu neprošlapal.

Musí například získat správná data z modelové vrstvy. Data pak musí umět zobrazit v rozložení stránky, které už pak budou další nové controllery sdílet. Přitom musí mít přístup k různým službám, například ke slovníku.

5.1 Modelová vrstva

Modelová vrstva MVC aplikace má controllerům zpřístupňovat funkce pro práci s daty. Po krátkém úvodu následuje nejprve varianta, jak by teoreticky bylo možné sdílet starou modelovou vrstvu s novými controllery. Tato vrstva je však silně závislá na komponentách starého frameworku, které nejsou dále podporovány. Poté následuje varinta vytvoření nové modelové vrstvy.

5.1.1 Komunikace s IGO Platform

Ke komunikaci se používá již v úvodu zmíněné REST API. Ověření přístupu probíhá pomocí základní HTTP autentifikace. Jméno a heslo pro autentifikaci se zadává při přihlášení do aplikace IGO Admin. Ta si ho trvale nikam neukládá, má ho pouze uložené v session, na které tedy závisí celá modelová vrstva, protože ho potřebuje k přístupu k jakýmkoli datům. Výměnným formátem dat je JSON.

Modul OM má podobně jako mnohé ostatní moduly svojí modelovou vrstvu (ve složce /models). Pro každý modelový „objekt“ (byť tak není formálně reprezentován) má svojí třídu, která zajišťuje práci jako je načtení, uložení apod. Tyto třídy jsou potomky třídy `Om_Model_Rest_BaseRestAbstract`. Podobnou třídu `*_Model_Rest_BaseRestAbstract` má téměř každý modul. A tyto třídy dědí z `Orchitech_Rest_Json_Client_Abstract`, která jako svůj privátní http client používá právě zmíněný `Zend_Http_Client`.

5.1.2 Varianta znovupoužití stávající modelové vrstvy

Autor se nejprve vydal cestou zpřístupnění staré modelové vrstvy novým controllerům. Vedly ho k tomu dva důvody. Jednak chtěl co nejrychleji získat data pro controller a view a tak dokončit koncept „dema“. Za druhé si myslel, že ta vrstva není s frameworkem Zend příliš svázána a že bude bude relativně snadné tuto vrstvu znovupoužít. Tuto variantu nakonec nezvolil, ale podívejme se, jak postupoval.

Data o organizacích zajišťovala třída `Om_Model_Rest_Organizations`.
Z následujícího kódu je zjevné, že je závislá na komponentě Zend_Registry, která poskytuje url adresu datového zdroje.

```
public function __construct() {
    parent::__construct();
    $this->_serviceUrl = Zend_Registry::get('config')->om->url;
    ...
```

Komponenty Zend_Registry je potřeba se v rámci migrace výhledově zbavit. Symfony má samozřejmě svůj způsob pro práci s parametry. Relativně snadné řešení tedy bylo přenést tyto parametry do světa Symfony a postarat se, aby byly zpětně kompatibilní se starým světem, když je bude potřebovat.

Konkrétně tedy do souboru parameters.yml v adresáři app/config:

```
parameters:
    om.url: 'http://igodev:8000/om/'
```

Symfony uplatňuje vzor DI, takže pokud je nějaká třída na něčem závislá, měla by to o sobě dát vědět, ideálně ve svém konstruktoru. Například takto:

```
public function __construct($url) {
    parent::__construct();
    $this->_serviceUrl = $url;
    ...
```

Frameworku musíme ještě říci, aby tuto třídu považoval za službu, a aby jí předával potřebné parametry do konstruktoru. To se dělá v souboru `services.yml` umístěném také v adresáři s konfigurací aplikace:

```
services:
    # ...
    om.organizations:
        class:          Om_Model_Rest_Organizations
        arguments:    [%om.url%]
```

Nyní je komponenta funkční ve světě Symfony. Je potřeba ještě zajistit, aby bylo možno jí volat ve světě Zend aplikace, která DI neumí. Kontejner služeb je však aplikaci zpřístupněn při fallbacku, takže stačilo jej použít a volat třídu skrze něj. To se muselo udělat v 53 souborech. Ta změna vypadala takto:

```
// add
use LegacyBundle\Compatibility\Container;
...
// find
$organizationsRest = new Om_Model_Rest_Organizations();
// replace
$organizationsRest = Container::get("om.organizations");
```

Podobně autor pokračoval ještě s třídou `Om_Model_Rest_Roles`, která je ve vybrané části také používána.

Řešení je to funkční a ukazuje zajímavou myšlenku, totiž jak znovupoužít kód, pokud by jeho jedinou „vadou“ bylo používání Zend_Registry. To však není tento případ. Třída `Om_Model_Rest_Organizations` je totiž potomkem třídy `Om_Model_Rest_BaseRestAbstract` , která je společná pro všechny modely modulu OM, a ta však zase třídy Orchitech_Rest_Json_Client_Abstract, která ale používá komponentu Zend_Http_Client. Té však skončila podpora spolu s frameworkem, takže nebude nadále používána. Komponenty také mají závislost na Zend_Session, a proto se v nové části aplikace bez session hacku nebudou používat.

5.1.3 Varianta nové modelové vrstvy včetně nahrazení http klienta

Vhodným nástupcem komponenty Zend_Http_Client je Guzzle[24], kterou framework Symfony doporučuje. Rozhodnutí o tomto nástupci provedl tým Orchitech.

Guzzle klient je architektonicky trochu jiný než Zend_Http_Client. Je takzvaně „Immutable“. Zajímavou vlastností, kterou potom nabízí, je volat asynchronní requesty.

[24] http://docs.guzzlephp.org/en/stable/

Přechod na tohoto klienta bude probíhat formou vytváření úplně nové modelové vrstvy, která se bude rozrůstat a zlepšovat s každým dalším přepisovaným controllerem.

Nová modelová vrstva bude lepší také v tom směru, že jako datový formát nebude používat obyčejná pole, ale i namísto nich entity. To jsou instance třídy, která definuje atributy a metody pro přístup k nim. Pro data „organizace“ vypadá následovně (ukázka je zkrácena jen na atributy name a active, celkově jich tato entita má 37):

```php
<?php
namespace OmBundle\Entity;
class Organization
{
    /** @var string|null */
    private $name;

    /** @var boolean */
    private $active;

    /**
     * @return string|null
     */
    public function getName(){
        return $this->name;
    }

    /**
     * @param string $name
     * @return Organization
     */
    public function setName($name){
        $this->name = $name;
        return $this;
    }

    /**
     * @return boolean
     */
    public function isActive(){
        return $this->active;
    }

    /**
     * @param boolean $active
     * @return Organization
     */
```

```
    public function setActive($active){
        $this->active = $active;
        return $this;
    }
}
```

Dále byla napsána třída pro práci s touto entitou. V jejím konstruktoru získáváme parametr URL ze symfony kontejneru, dále objekt $session, ve kterém je uloženo jméno a heslo a samozřejmě nového http klienta Guzzle.

```
class OrganizationRepository {
    private $_serviceUrl;
    private $_path = "organizations/";
    private $_auth;
    private $_client;
    public function __construct($url, SessionInterface $session,
GuzzleHttp\ClientInterface $client) {

        $this->_serviceUrl = $url;
        $this->_auth = [
                'auth' => [
                        $session->get ( 'username' ),
                        $session->get ( 'password' )
                ]
        ];
        $this->_client = $client;

    }
```

Snahou je žádat o rozhraní (interface) těchto služeb, aby mohly být snadněji nahrazeny v případě potřeby. Aby Symfony tuto třídu považoval za službu, uměl ji vytvářet a vkládal do ní potřebné závislosti, musíme ji definovat v již zmiňovaném souboru service.yaml:

```
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true
        autoconfigure: true
        public: false
    # ...
    OmBundle\Repository\OrganizationsRepository:
        arguments: ["%om.url%"]
```

```
    GuzzleHttp\ClientInterace:
        class: GuzzleHttp\Client

    OmBundle\Repository\OrganizationRepository:
        class: OmBundle\Repository\OrganizationRepository
        shared:    false
        arguments:
            $url: "%om.url%"
```

Guzzle klient se poté již může použít v metodě zajišťující získání dat organizace na základě jejího organizationId.

```
/**
 * @param int $id
 * @return Organization
 */
public function getOrganizationById($id)
{
    $response = $this->_client->request('GET', $this->_serviceUrl
. $this->_path . $id, [
        'auth' => $this->_auth
    ]);
    return $this->deserialize($response->getBody());
}
```

Metodou deserialize jsou obdržená JSON data přeměna na entitu organization:

```
/**
 * @param $json data
 * @return Organization entity
 */
private function deserialize($json)
{
    $serializer = new Serializer([
        new ObjectNormalizer()
    ], [
        new JsonEncoder()
    ]);
    $organization = $serializer->deserialize($json,
        Organization::class, 'json');
```

```
    return $organization;
}
```

Nová vrstva od té chvíle začíná fungovat.

V nové vrstvě nejsou plně implementovány náhrady za komponenty:

- Zend_Auth
- Zend_Storage
- Zend_Translate (pro překlad hlášek z IGO Platform)
- Zend_Acl
- Zend_Cache

5.2 View – vrstva zobrazení

Data, která vrstva view obdrží musí vhodně zobrazit. V době interaktivních aplikací někdy postačí jen vhodný formát, například json nebo xml a problematika zobrazení se přesouvá na stranu klienta. Na takové situace jsou frameworky připraveny. V ostatních případech musí grafické rozhraní obstarat server.

5.2.1 Šablona

Šablona je nejčastějším případem využití vrstvy view. Na daná místa v html kódu se umístí údaje předané controllerem. Šablona také často obsahuje jednoduchou logiku, která souvisí s uživatelským rozhraním. Nezřídka také potřebuje pracovat s cykly, když dostane více dat stejného typu. Přesto, že právě s tímto účelem vznikal samotný jazyk PHP, vzniklo dnes několik šablonovacích systémů (třeba Twig – podporovaný ve frameworku Symfony, nebo například Latte od českého frameworku Nette). Twig potřebu svého vzniku obhajuje tím, že se jazyk PHP dále nevyvíjel směrem šablonovacích systémů, a tak neobsahuje funkce, které dnešní moderní šablonovací systémy mají.[25]

Přesto je v Symfony možné používat šablony v jazyce PHP. V oficiální dokumentaci je uvedený návod, jak pro tento účel framework nastavit[26]. Komponenta „templating“ umí zpracovávat jak šablony .twig, tak i .php, a ty navíc obohacuje o některé funkce, které PHP v základu nemá. Na stránkách Symfony jsou často vidět příklady ve variantě TWIG i PHP.

[25] https://twig.symfony.com/

[26] https://symfony.com/doc/current/templating/PHP.html

Zend má své šablony v PHP a dále je uvedeno, jak je možné starou šablonu přepsat tak, aby fungovala v Symfony. Konkrétně to bylo zkoušeno na obrazovce pro zobrazení detailu organizace v modulu OM, která navazuje na přepisovaný controller.

Velmi zjednodušená ukázka html šablony pro zobrazení detailu organizace:

```
<div class="detail">
  <!-- Organization -->
  <?php if (!empty($this->organization['name'])): ?>
    <div class="fieldRow">
      <span class="label">
       <?php echo $this->translate('om.organization.name');?>
     </span>
      <span class="field">
      <?php
echo $this->escape($this->attr($this->organization, 'name'));
      ?>
      </span>
    </div>
  <?php endif; ?>
</div>
```

Ztučněné výrazy musíme vyřešit, protože v novém frameworku se k nim přistupuje jinak. Je to vidět na následující ukázce.

```
<div class="detail">
  <?php if (!empty($organization['name'])): ?>
    <div class="fieldRow">
      <span class="label">
       <?php
echo $view['translator']->trans('om.organization.name', [], 'om');
        ?>
      </span>
      <span class="field">
      <?php
echo $view->escape($organization['name']);
      ?>
      </span>
    </div>
  <?php endif; ?>
</div>
```

Rychle použitelné find&replace

Na hromadné vyřešení je možné použít následující regulární výrazy. První se týká překládání, které je vysvětlené ještě dále.

FIND: `\$this->translate\('(.*)'\)`

REPLACE: `\$view['translator']->trans\('$1', [], 'om'\)`

V šabloně je dále nahrazeno:

```
$this->organization      =>   $organization
$this->escape            =>   $view->escape
```

FIND: `\$this->attr\(\$(.*), '(.*)'\)`

např.: `$this->attr`[27]`($organization, 'email')`

REPLACE: `\$$1['$2']`

např.: `$organization['email']`

5.2.1.1 Nahrazení pole na entity

Výše uvedené platí, pokud by modelová vrstva používala stávající formát dat, tedy pole. Jak ale bylo v předchozí kapitole psáno, tento formát se v průběhu práce změnil na objekt s předem definovanými atributy, kterému se běžně říká entita. To si vyžádalo další drobné změny.

Název žádaného atributu nyní již nebude jako klíč v poli. Entita má atributy neveřejné, a podle konvence povoluje přístup skrze metody začínající na get a set (případně is, pokud jde o typ boolean).

Cílem tedy je v šabloně z tohoto:

```
$organization['promena']
```

udělat toto:

```
$organization->getPromena()
```

[27] Metoda attr() zkontroluje, zda hodnota není prázdná a pokud ne, vrátí ji.

Protože se tato práce opakuje pro mnoho atributů, mohou s nalezením a nahrazením pomoci následující regulární výrazy v těchto krocích:

1.) Nejdříve změnit první písmeno klíče na velké[28]

FIND: `\$organization\['(.1*?)`

REPLACE: `\$organization\['\U\1\E`

2.) Potom už jen nahradit zbytek

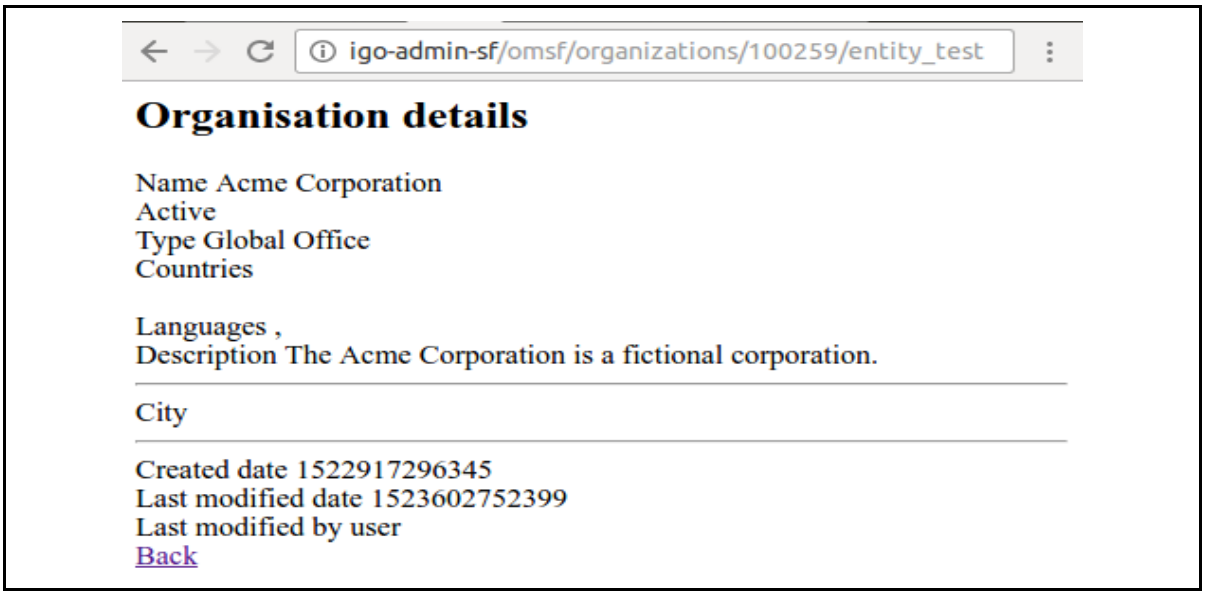
FIND: `\$organization\['(.*?)'\]`

REPLACE: `\$organization->get$1()`

Je zde potřeba si dát pozor na atributy typu boolean, které nemají getter, ale isser. Ve vybrané šabloně jsou to například **active, deleted** či **cardholderEmailsEnabled)**

Při změně z pole na entitu se už také nemůžeme dále ptát, zda je atribut nastaven pomocí isset() nýbrž porovnáním s null ($organization->getVariable() !== null).

Samotná šablona formuláře bez layoutu a css stylů pak vypadá následovně.



Snímek obrazovky s vypsanými daty v šabloně bez layoutu.

[28] Není funkční v Eclipse, použijte např.: https://regex101.com/ viz: https://regex101.com/r/TwfBYJ/1 pak: https://regex101.com/r/6lTkTj/1

Některé problémy nebyly v rámci této experimentální práce řešeny, pouze zaznamenány. Jde o:

- $this->stateImage
- $this->entityReference
- echo $this->entityReference
- $this->city
- $this->date
- $this->url (generuje url v rámci zend aplikace)
- $this->backUrl

5.2.2 Layout – rozložení uživatelského rozhraní

Grafické rozhraní se dá rozdělit na několik částí. Ta nejobecnější část s logem a navigací bývá často sdílená mnoha controllery a říká se jí layout. V nich pak jsou umístěné obrazovky pro konkrétní controller.

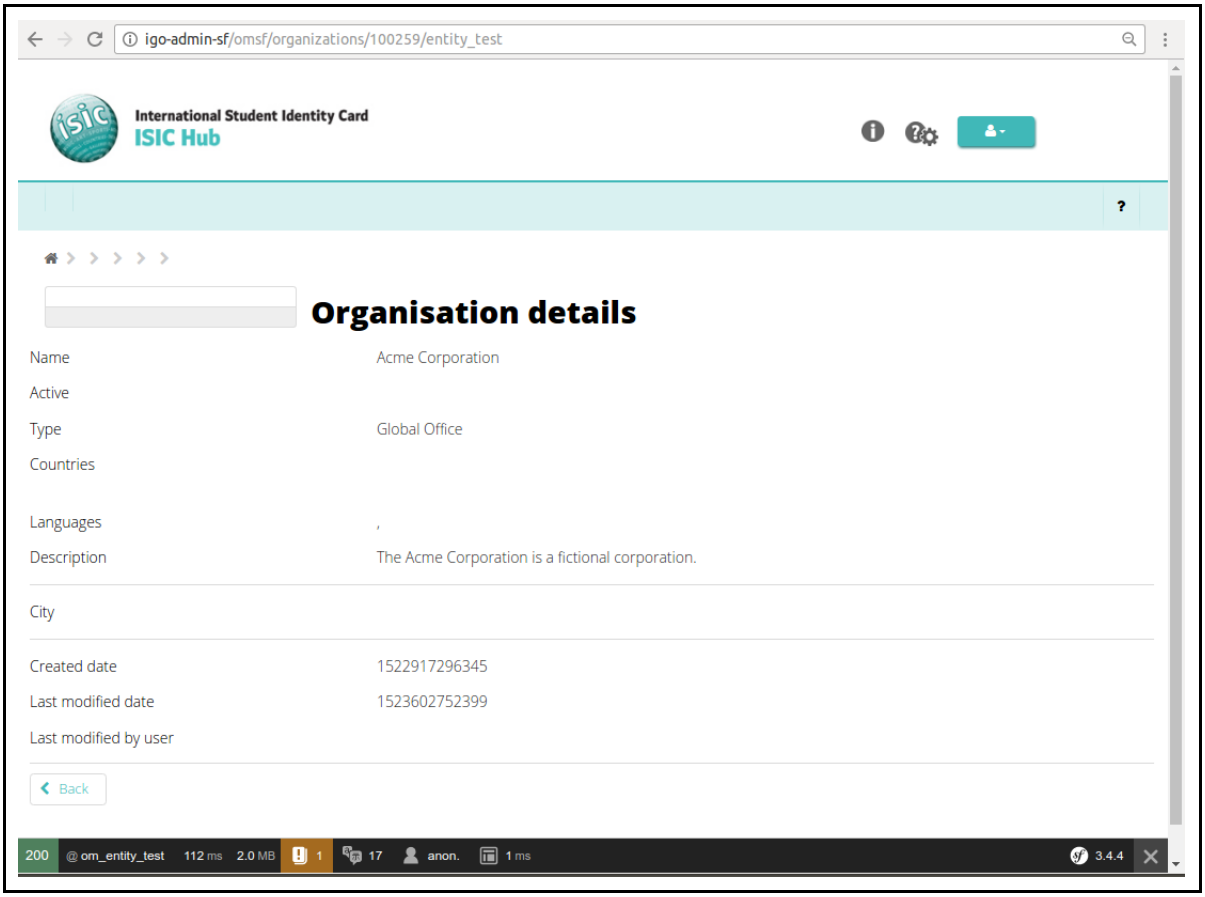
Přístup Symfony frameworku k layoutu se trochu liší od přístupu frameworku Zend. Layout říká, kam vypíše obsah:

```
<?php $view['slots']->output('_content'); ?>
```

Konkrétní komponenta (např. formulář) pak říká, že je obsahem pro nějaký „layout“:

```
<?php $view->extend('layout.html.php') ?>
```

Z následujícího obrázku je vidět, jak byl formulář zabalen do layoutu stránky. Také lze vypozorovat, že detailní problematika zobrazení levého submenu a také další prvky celého layoutu nebyly prozatím řešeny.



Snímek obrazovky s detailem organizace vypsanou v částečně přepsaném laoyutu.

5.2.3 Flash messages

„Flash messages“ jsou hlášky informující například o úspěchu nebo chybě při odeslání formuláře. Jsou to krátkodobé hlášky, které se zobrazí jen těsně po nějaké akci, o které je vhodné uživatele informovat.

Ve frameworku Zend se jejich zobrazení v šablonách volalo takto:

```
<?php echo $this->partial('common/notifications.phtml', 'default',
array('messages' => $this->notificationMessages)); ?>
```

Ve frameworku Symfony je to děláno takto:

```
<?php echo view->render(':scripts:common/notifications.html.php') ?>
```

Samozřejmě je potřeba i patřičná šablona. Ta může v základu vypadat pro framework Symfony následovně:

```
<?php foreach ($view['session']->getFlashes() as $type =>
$flash_messages): ?>
    <?php foreach ($flash_messages as $flash_message): ?>
```

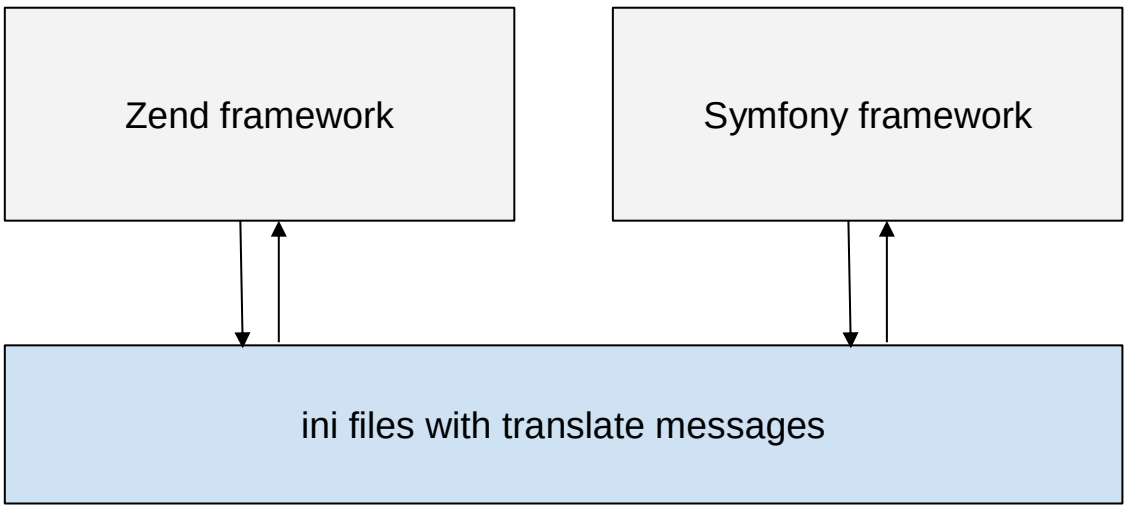
```
        <div class="flash-<?php echo $type ?>">
            <?php echo $flash_message ?>
        </div>
    <?php endforeach ?>
<?php endforeach ?>
```

5.2.4 Translate – jazykový překlad aplikace

Software je přístupnější většímu množství lidí, pokud podporuje více jazyků. To je žádoucí pro mnoho projektů a tak frameworky, na kterých tyto projekty staví, nabízí způsoby, jak vícejazyčnosti dosáhnout. Klíčové je každou zprávu nebo hlášku, která je v projektu použita, nahradit zástupnou hláškou a v šablonách pak používat jen tyto. Pro každý podporovaný jazyk je pak nutné vytvořit slovník, ve kterém bude každá zástupná hláška přeložena do daného jazyka.

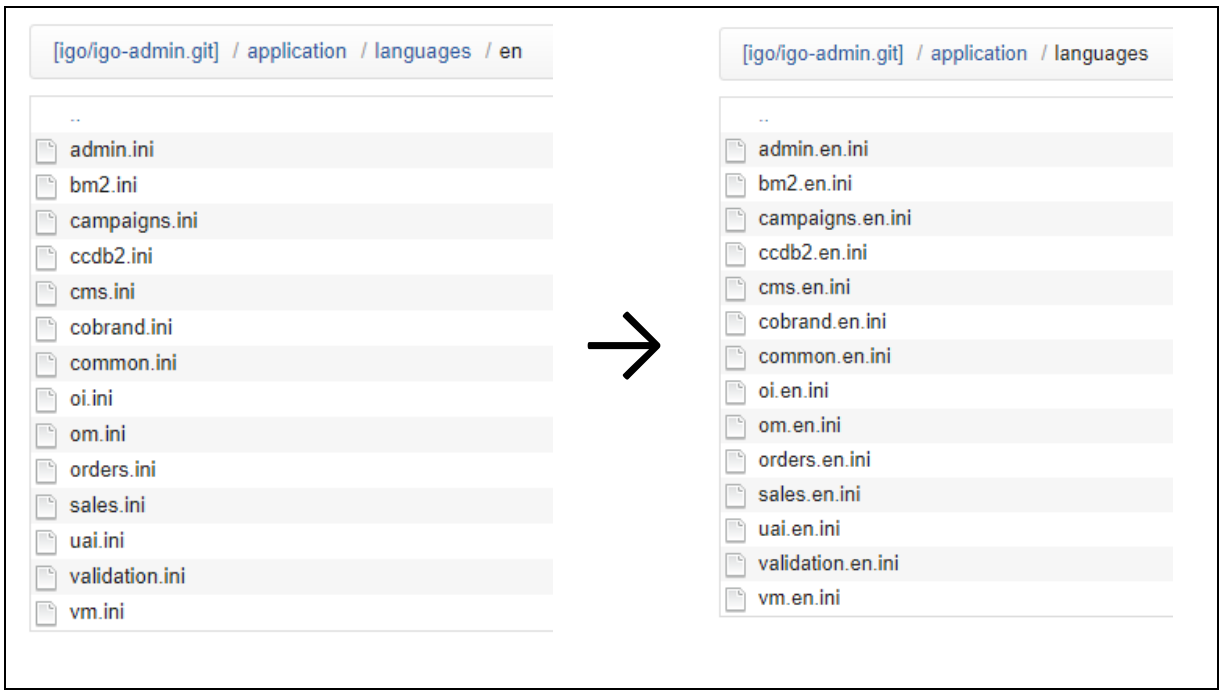
V naší aplikaci se používá pouze angličtina. Zatím zřejmě nebyl požadavek aplikaci překládat do více jazyků. Přesto aplikace používá zástupné hlášky. To je výhodné i z důvodu lepší správy těchto hlášek. Některé se totiž opakují na mnoha místech a takto je snazší předcházet a opravovat překlepy.

Oba frameworky mají zavedený způsob, jak s překládáním pracovat. Bylo by velmi nepraktické duplikovat slovníky pro každý systém zvlášť. Naštěstí komponenta Translation z frameworku Symfony je natolik flexibilní, že podporuje různé druhy slovníků, a tak stačilo pár drobných úprav a oba systémy teď slovníky sdílí, jak je znázorněno následujícím nákresem.



Dva frameworky sdílí stejný slovník.

Z obrázku níže se dají vyčíst ony drobné úpravy:



Úprava umístění a názvů souborů s překlady.

Symfony ve výchozím stavu nebere zkratku jazyka z umístění, ale z názvu souboru, tudíž bylo zapotřebí pojmenovat soubory podle nové konvence a pokusit se vnutit jí Zendu, jinak nechat vše po starém. Ukázalo se, že nová konvence Zendu vůbec nevadí.

```
# /application/configs/application.ini
# ...
resources.locale.default         = "en"
resources.locale.force           = TRUE
resources.translate.adapter      = ini
resources.translate.data         = APPLICATION_PATH "/languages"
resources.translate.locale       = "en"
resources.translate.options.scan = "directory"
# ...
```

Kód: Nezměněná konfigurace překladů ve frameworku Zend

Zajímavostí frameworku Symfony je debugger, který mimo jiné varuje i pokud vypisovaným hláškám chybí překlad. To jistě zpříjemňuje vývoj a pomáhá předcházet chybám.

5.3 Vrstva controllerů

Úlohou controllerů je obsloužit požadavek uživatele.

5.3.1 Jak začít s controllerem a jeho volání pohledu

Controller je vlastně funkce, jejímž vstupem jsou data z HTTP požadavku. Jeho odpovědností je pak zavolání správného pohledu, což je nejčastěji šablona a předaná data získaná z modelu. Jak se v porovnávaných frameworcích liší přístup ke vstupům a výstupu? A jak je přepsat? Dále je po částech popsána snaha o přepsání detailAction z controlleru organizatioins z modulu OM.

5.3.1.1 Vstupní parametry

Parametry z requestu controller v Symfony získává skrze hlavičku controlleru pomocí anotace @Route. Umístění složených závorek v URL adrese určuje, která její část bude použita jako parametr.

Např.:

```
/** @Route("/omsf/detail/{organizationId}") */
function detailAction($organizationId) {
    ...
```

Místo dřívějšího:

```
function detailAction() {
    $organizationId = $this->getRequest()->getParam('id');
    ...
```

Je dobré si také uvědomit, že právě touto anotací je pro framework definována adresa controlleru. Pro dobu vývoje ji autor definoval odlišně od adresy nahrazovaného controlleru (např.: „omsf/detail/{organizationId}“), díky čemuž je možné nadále zobrazovat zároveň původní controller. Ve chvíli dokončení vývoje nového controlleru, bude adresa definována jako adresa původního controlleru. Tím se nový kus kódu funkčně začlení do celého projektu. Starý controller tak již nebude spouštěn a bude možné ho z kódu vyřadit.

5.3.1.2 Závislosti

Již bylo zmíněno, že Symfony framework používá vzor DI. To v praxi pro controllery znamená, že by se v nich již neměli objevovat klíčová slova new. V hlavičce své funkce action() si mohou sepsat vyžadované závislosti.

Controller dostane službu přes konstruktor a může jí používat:

```
public function detailAction($organizationId,
    \Om_Model_Rest_Organizations $organizationRestClient,
    \Om_Model_Rest_Roles $rolesRestClient)
{
    $organization = $organizationRestClient
                        ->getOrganizationById($organizationId);
    $roles = $rolesRestClient->findRoles(...
    ...
```

5.3.1.3 Volání šablony a předání parametrů

Prvky do view se předají v poli v return části.

Takže místo:

```
$this->view->childOrganizations = $childOrganizations['items'];
$this->view->organization = $organization;
$this->view->organizationWebsites = explode(",",
$organization['websites']);
```

Napsat:

```
return $this->render(':default:detail.html.php', [
    'organization' => $organization,
    'childOrganizations' => $childOrganizations['items'],
    'organizationWebsites' => explode(",", $organization['websites']),
]);
```

Je dobré myslet na to, že v šabloně je pak nutné místo `$this->nazevPredanePromene` volat rovnou `$nazevPredanePromene` (takže místo `$this->organization` tam bude jen `$organization`).

5.4 Další aspekty

Výše uvedené body poukázaly na různé teoretické i praktické aspekty na jednotlivých vrstvách MVC aplikace. Během přepisu se ale musí vzít v úvahu ještě další, méně technická hlediska. Následují zejména rady od Paula Hammanta.

5.4.1 Nové funkce a zlepšení

Je zapotřebí je přinášet. Může tedy být dobré začít přepisovat tu část, kde by se hodilo nějaké vylepšení.

V tomto projektu to může být například ajaxové zobrazení s pokročilým filtrováním, které už některé moduly či controllery mají a jiné ještě ne.

Někteří autoři dokonce doporučovali už nepřidávat žádné nové funkce do starého kódu, ale všechny už zapracovávat do nového.[29] To může zdržet jinak jednoduché požadavky, ale urychlí to celkový přechod.

5.4.2 Doba mezi commity

Doba mezi commity by měla být co nejkratší. Předpokládá se, že první část bude trvat trochu déle, ale cílem je mít projekt co nejkratší dobu v rozpracovaném stavu.

5.4.3 Pokrytí testy

Většina autorů se shodla na tom, že je důležité mít aplikaci co nejlépe pokrytou testy. Tým Orchitech se k tomu musí nějak postavit.

[29] https://jvaneyck.wordpress.com/2015/03/12/the-big-rewrite/

6 Závěr

6.1 Souhrnné zhodnocení praktických a teoretických aspektů migrace

Na následující tabulce je uveden přehled praktických a teoretických aspektů migrace, včetně slovního zhodnocení z hlediska nákladů, čistoty a udržitelnosti projektu. Tím je splněn hlavní cíl práce.

Oblast (aspekt)	**Možnosti (tučně zvýrazněné jsou autorem doporučené)**	**Zhodnocení z hlediska nákladů, čistoty a udržitelnosti projektu**
Styl postupu přechodu (kapitola 3)	Jednorázový	Vzhledem ke komplexnosti projektu a jeho aktivnímu vývoji tento styl migrace autor hodnotí jako neproveditelný.
	Postupný	Migraci je možné rozdělit na mnoho menších částí a aplikace bude funkční během celé doby přechodu. To je výhodné i vzhledem k plánování časových investic.
Směrování URL na danou aplikaci (kapitola 4.2)	Apache URL list	Nevýhodou je nutnost po každé přepsané akci controlleru generovat seznam URL a aktualizovat ho na webovém serveru. Výhodou by mohlo být bezpečné, úplné oddělení těchto aplikací, ale na druhou stranu to komplikuje sdílení komponent.
	Symfony first, fallback to Zend	Nová URL se jednoduše zapíše v anotaci controlleru, a tím nahradí konkrétní akce staré aplikace. Výhodou je snadná nasaditelnost. Umožňuje snadno sdílet nové komponenty starému kódu.
	StackPHP	Řešení nebylo v porovnání s předchozím dostatečně transparentní, ale dá se předpokládat, že by bylo taktéž použitelné.
Session (kapitola 4.3)	PhpBridge	V situaci, kdy je spouštěn Symfony jako první a Zend se spouští jen při fallbacku, je to nevhodné řešešní, protože se musí ručně zajistit spouštění session u nově přepsaných controllerů, což není čisté řešení a vytváří technický dluh do budoucna.
	Zend_Session hack	Toto řešení autor hodnotí jako lepší.

		Symfony si spouští sessions podle potřeby a Zend se s tím vyrovná.
Security (kapitola 4.4)	Sdílet starou komponentu s novým frameworkem	Zamítnuto – jde to proti smyslu migrace. Stará security závisí na Zend komponentách.
	Vytvořit novou komponentu a sdílet ji se starým frameworkem	Zbytečně nákladný a riskantní zásah do celého zbytku projektu vzhledem k tomu, že existuje lepší možnost.
	Vytvořit novou komponentou, která bude Sdílet stejnou datovou strukturu v Symfony session	Vyžaduje přesunout datovou strukturu ze Zendu do Symfony a naučit to starou komponentu. Nicméně je snazší sdílet datový zdroj, než sdílet komponentu.
Modelová vrstva (5.1.2 a 5.1.3)	Znovupoužití staré	Na první pohled možná úsporné řešení, po hlubším prozkoumání však jde proti smyslu migrace.
	Nová s novým http clientem	Pro úspěšnou migraci je nutné nahradit http clienta. Stará může dále fungovat do dokončení přepisu. Stačí vytvářet vrstvu po přepisovaných částech.
Datový formát uvnitř aplikace (5.1.3)	PHP array	Nevýhodou je malý přehled o možných indexech v poli.
	Entita	Zvyšuje přehled o možných atributech. Díky Symfony komponentě Serializer je také snadné možnost implementovat. Vzorem je javovská entita na IGO Platform. Když už se stejně přepisuje nová modelová vrstva, náklady navíc jsou minimální.
Šablony (5.2.1)	PHP	Obě možnosti jsou podporované frameworkem Symfony. PHP šablona by mohla být snazší na přepis, je možné použít připravené regulární výrazy. Přesto bude nutný ruční zásah.
	TWIG	Obě možnosti jsou podporované frameworkem Symfony. Zřejmě vyžaduje vyšší prvotní časovou investici, ale postupně může ušetřit čas a předejít bezpečnostním rizikům pomocí zabudovaných funkcí.

Překladové hlášky (5.2.4)	**Sdílet stejný zdroj dat**	Jednoznačně nejlepší možnost, která předchází nežádoucí duplikaci dat. Minimální náročnost na implementaci.
	Sdílet jednu komponentu v obou aplikacích	Pojí se s tím komplikovaná implementace komponenty na všech vrstvách.
	Duplikovat data v různých formátech	To by bylo náročné na údržbu a tedy náchylné k nekonzistenci.
Layout (5.2.2)	Sdílený	Přepsat layout aby vyhovoval oběma frameworkům může být těžko proveditelné a tím přinášet vysoké náklady. Na stranu druhou neduplikuje kód.
	Zvlášť pro každý framework	Autor zvolil tuto cestu, protože minimálně zpočátku přináší nižší náklady. Dočasně vytváří duplicitní kód, ale jde jen o jeden soubor. Navíc to neohrožuje rozbití navigace v celé aplikaci.
Správa závislostí (5.3.1.2 a 2.4.3)	přímé volání *new*	Je to špinavé řešení s nezřejmými závislostmi a špatně testovatelné.
	Dependency Injection	To je řešení pohodlné, s nízkými náklady, dobře čitelné – udržitelné.

Tabulka zhodnocující některá možná řešení jednotlivých aspektů.

6.2 Zhodnocení splnění dílčích cílů

6.2.1 Dílčí cíl 1

Hned v úvodu práce (1.1 Motivace k migraci) autor čtenáře seznamuje s principiálními důvody pro přechod z jednoho frameworku na druhý, když vysvětlil pojem End-of-Life a potřebu stavět na podporovaném software. V kapitole 2.4 byli čtenáři seznámeni s návrhovými vzory, které frameworky používají – MVC (2.4.1), front-controller (2.4.2) a DI (2.4.3).

6.2.2 Dílčí cíl 2

Odlišnosti v uplatňování návrhového vzoru MVC frameworky Zend Framework a Symfony 3 jsou detailně popsány v kapitole 5, a to na každé vrstvě tohoto vzoru zvlášť – Model (5.1), View (5.2), Controller (5.3). V části 2.4.2 je také uvedena odlišnost v uplatnění vzoru front controller.

6.2.3 Dílčí cíl 3

Obecné možnosti postupu přechodu jsou popsány v kapitole 3, jde o možnost tzv. Big Rewrite (3.1) a způsob postupného přepisu (3.3). Výběr vhodného obecného postupu pro konkrétní projekt IGO Admin 2017 je popsán v závěru této kapitoly (3.4). V kapitole 4.2.1 jsou formou popisu a nákresů teoreticky vysvětleny tři možné způsoby implementace vybraného obecného postupu a jsou též slovně zhodnoceny. Týkají se prvních kroků vedoucích k zajištění koexistence obou frameworků v projektu. V rámci migrace je potřeba zajistit i další postupy týkající se například – sessions (4.3), bezpečnosti (4.4), získávání dat (5.1), přepisování šablon (5.2.1), controllerů (5.3.1). Ty jsou v kapitolách, které se jimi zabývají, navrženy a zhodnoceny z pohledu nákladů, čistoty návrhu a udržitelnosti rozvoje dalšího projektu. Hodnocení je pro přehlednost shrnuto v tabulce v závěru práce (6.1).

6.2.4 Dílčí cíl 4

Některé nejlépe vyhodnocené postupy byly použity pro vytvoření ukázky přechodu vybrané části projektu, jíž je akce „detail“ z controlleru „organizations“ v modulu „OM“. Svědčí o tom klíčové úryvky zdrojového kódu a snímky obrazovky. Ukázka zahrnuje instalaci nového frameworku do projektu (4.1), zajištění koexistence obou frameworků po dobu přechodu pro umožnění aktivního vývoje nových požadavků (4.2), vypořádání se se sessions (4.3), vytvoření a použití nové modelové vrstvy (5.1.3), použití entity místo pole jako datový formát uvnitř aplikace (5.1.3 a 5.2.1.1), přepsání šablony a layoutu (5.2.1 a 5.2.2), sdílení stejného zdroje překladových hlášek novou komponentou (5.2.4), použití anotací pro definici URL adresy controlleru (5.3.1.1) a získávání závislostí pomocí návrhového vzoru DI (5.3.1.2). Některé postupy byly týmem Orchitech použity v praxi.

7 Seznam literatury

BUENOSVINOS, Carlos. Migrating progressively to Symfony without pain with StackPHP. In: Carlos Buenosvinos [online]. Barcelona: Buenosvinos, 2015 [cit. 2018-04-06]. Dostupné z: https://carlosbuenosvinos.com/migrating-progressively-to-symfony-without-pain-with-stackphp/

BUENOSVINOS, Carlos. Migrating progressively to Symfony without pain. In: Carlos Buenosvinos [online]. Barcelona: Buenosvinos, 2015 [cit. 2018-04-06]. Dostupné z: https://carlosbuenosvinos.com/migrating-progressively-to-symfony-without-pain/

DE LANGE, Yannick. Migrating your project to Symfony. In: Stovepipe Systems [online]. Amsterdam: Lange, 2016 [cit. 2018-04-06]. Dostupné z: https://stovepipe.systems/post/migrating-your-projectto-symfony

FOWLER, Martin. StranglerApplication. In: Martin Fowler [online]. Chicago: Fowler, 2004 [cit. 2018-04-05]. Dostupné z: https://www.martinfowler.com/bliki/StranglerApplication.html

FOWLER, Martin a Kent. BECK. Refactoring: improving the design of existing code. Reading, MA: Addison-Wesley, 1999. ISBN 978-0201485677.

GRUDL, David. Co je Dependency Injection?. In: PhpFashion [online]. Praha: Grudl, 2012 [cit. 2018-04-05]. Dostupné z: https://phpfashion.com/co-je-dependency-injection

HAMMANT, Paul. Legacy Application Strangulation : Case Studies. In: Paul Hammant's blog [online]. New York: Hammant, 2013 [cit. 2018-04-05]. Dostupné z: https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/

KNESL, Jiří. Problémy při přepisu aplikace na novou verzi. In: Jiří Knesl [online]. Praha: Knesl, 2014 [cit. 2018-04-06]. Dostupné z: http://www.knesl.com/problemy-pri-prepisu-stareho-systemu

O'PHINNEY, Weier. Zend Framework 1 End-of-Life Announcement. In: Zend Framework: Blog [online]. [neuvedeno]: Weier O'Phinney, 2016 [cit. 2018-04-05]. Dostupné z:

https://framework.zend.com/blog/2016-06-28-zf1-eol.html

POTENCIER, Fabien et al. Our Backward Compatibility Promise. In: Symfony [online]. [místo neuvedeno]: Potencier, 2013 [cit. 2018-04-05]. Dostupné z: https://symfony.com/doc/3.4/contributing/code/bc.html

POTENCIER, Fabien et al. Symfony Roadmap. In: Symfony [online]. [neuvedeno]: Potencier, 2016 [cit. 2018-04-05]. Dostupné z: https://symfony.com/roadmap?version=3.4

SPOLSKY, Joel. Things You Should Never Do, Part I. In: JOEL ON SOFTWARE [online]. New York City: Spolsky, 2000 [cit. 2018-04-05]. Dostupné z: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

SPOLSKY, Joel. Don't Let Architecture Astronauts Scare You. In: JOEL ON SOFTWARE [online]. New York City: Spolsky, 2001 [cit. 2018-04-05]. Dostupné z: https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/

SPOLSKY, Joel. The Duct Tape Programmer. In: JOEL ON SOFTWARE [online]. New York City: Spolsky, 2009 [cit. 2018-04-05]. Dostupné z: https://www.joelonsoftware.com/2009/09/23/the-duc-tape-programmer/

Model-view-controller. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2005-. Dostupné také z: https://cs.wikipedia.org/wiki/Model-view-controller