Mendelova univerzita v Brně
Provozně ekonomická fakulta

Reengineering systému Umbrela

Diplomová práce

Vedoucí práce:
Ing. Pavel Turčínek, Ph.D.

Bc. Martin Maroši

Brno 2018

Děkuji Ing. Pavlu Turčínkovi, Ph.D. za vedení diplomové práce. Poděkováni patři také Ing. Stanislavu Mokrému, Ph.D. za konzultace ohledně analýzy a designu nového softwaru.

Čestné prohlášení

Prohlašuji, že jsem práci: **Reengineering systému Umbrela**
vypracoval samostatně a veškeré použité prameny a informace uvádím v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 8. května 2018 .....

Abstract

Maroši M. Reengineering of the Umbrela system. Dimploma thesis. Brno, 2018

The thesis deals with the reengineering of the Umbrela questionnaire system. The text describes the reasons for reengineering, technology selection, application design, description of basic application blocks and implementation. The work also mentions proposals for improvement of the overall project and its further development.

Key words

PHP, ORM, Doctrine, Nette, JavaScript, React, Redux, Marketing research, Continuous integration, Unit testing

Abstrakt

Maroši M. Reengineering systému Umbrela. Diplomová práce. Brno, 2018

Práce se zabývá reengineeringem dotazníkového systému Umbrela. V textu jsou popsány důvody reengineeringu, výběr technologií, návrh aplikace, popis základních bloků aplikace a implementace. V práci jsou také zmíněny návrhy ke zlepšení chodu celého projektu a dalšího vývoje.

Klíčová slova

PHP, ORM, Doctrine, Nette, JavaScript, React, Redux, Marketingový výzkum, Continuous integration, Jednotkové testy

Obsah

1 Úvod a cíl práce

1.1 Úvod

Ústav marketingu a obchodu Provozně ekonomické fakulty na Mendelově univerzitě v Brně se již od prvopočátku zabývá marketingovým výzkumem. Tyto výzkumy jsou realizovány jak v rámci výuky, ale také jako zdroj dat pro různé odborné články, závěrečné práce studentů a další aktivity studentů a zaměstnanců univerzity. Jedním ze základních metod výzkumu je takzvaný kvantitativní výzkum. Jak lze už z názvu odhadnout, základem kvantitativního výzkumu je sběr velkého množství dat. Na ně jsou následně aplikovány nejrůznější statistické a ekonometrické metody, sloužící k získání informací o trhu, výrobcích, spotřebitelích a dalších faktorech. Tyto informace mohou přispět ekonomickým subjektům k jejich rozvoji, nebo pouze informovat společnost o trendech a fenoménech ovlivňující trhy. Nejefektivnějším nástrojem ke sběru právě takových dat je elektronické dotazníkové šetření.

Ústav marketingu a obchodu má už několik let k dispozici aplikaci Umbrela (původně rela). Prostřednictvím této aplikace mohou studenti a zaměstnanci univerzity vytvářet dotazníky, které jsou v online podobě rozesílány respondentům, a získávají tak data pro jejich projekty a závěrečné práce. Poprvé se tato aplikace spustila zhruba před patnácti lety. Od prvopočátku byla vyvíjena a udržována studenty tehdy ještě Vysoké školy zemědělské v Brně. Během této doby se rozrůstala a měnila, avšak kvůli nekonzistentnímu týmu a časté změně jeho členů se ne vždy posouvala správným směrem. Postupem času se začalo vyskytovat mnoho chyb, které snižovaly uživatelskou přívětivost aplikace, a vzrůstala frustrace uživatelů. Popularitě také nepřispěly nové možnosti provádění online dotazníkového šetření jako jsou Google Forms nebo Survio.

Ve vzduchu se tedy začala pohybovat myšlenka o přestavění aplikace Umbrela tak, aby vyhovovala moderním standardům a vytvořila tak pro stávající i nové uživatele aplikace solidní prostředek pro sběr kvantitativních dat. I když v minulosti proběhly určité pokusy o tuto revitalizaci aplikace, ve všech případech se nakonec vedení uchýlilo pouze k menší úpravě některých částí kódu nebo přidání nových funkcí. Nicméně jádro systému zůstalo prakticky nedotčené a díky postupnému nabalováním funkcí a minimální kontrole konzistence a kompatibility těchto změn se Umbrela dostala do bodu, kdy bylo rozhodnuto o celkové obnově celé aplikace od jejich základů. Nová verze by neměla být žádným způsobem závislá na stávající a nebyly kladeny ani žádné limity na zvolené technologie. Jediná podmínka, která musí být splněna, je zachování dat ze současné databáze a zamezení tak přerušení kontinuity stávajících výzkumů.

1.2 Cíl práce

Cílem této práce je reengineering celé aplikace Umbrela. Nebude se jednat pouze o přepsání či modifikaci zdrojových kódu aplikace. V práci budou identifikovány

stávající problémy a chyby v aplikaci. Budou prodiskutovány zvolené technologie, jejich přínosy a nevýhody, a pokud bude nutné tak i výběr jiných. Na základně tohoto průzkumu bude vytvořená nová aplikace bez jakýchkoliv závislostí na starém kódu. Součástí procesu reengineeringu nebude pouze zdrojový kód, ale také uživatelské rozhraní, které už od prvního pohledu bez jakéhokoliv hlubšího zkoumání je evidentně zastaralé a nepřizpůsobené nejnovějším standardům a trendům moderních aplikací. Aby se předešlo stejnému osudu jako pro předchozí verzi, bude vytvořena podrobná dokumentace, která by měla zjednodušit budoucím vývojářům proniknout do zdrojového kódu a zamezit postupnému kolapsu celého systému a opakování současné situace. Musí být také brán ohled na skutečnost, že většinou je vývojářský tým složený z jednoho či dvou členů a vyskytují se i periody, ve kterých dokonce žádný tým neexistuje. Potom není možné předat všechny informace osobně a zajistit tak dostatečné porozumění a zachování procesů a konvencí vývoje, které se mohou značně lišit mezi člověkem, který bude přebírat zodpovědnost za další rozvíjení aplikace. Proto také nesmí chybět doporučení a pomůcky, které by měly napomoci dlouhodobé udržitelnosti a rozšiřitelnosti. Mezi ně mohou patřit například verzování kódu pomocí nějakého softwaru, postup a používání automatického testování a správná organizace úkolů.

Práce není zaměřená pouze na budoucí vývojáře, ale také na uživatele. Zjednodušení aplikace a zaměření se na intuitivnost poměrně složitého uživatelského rozhraní takovým způsobem, aby jej byl schopen používat i technicky méně zdatný uživatel, je také jedním z bodů, kterému bude věnována značná pozornost. To by mělo přispět ke snížení současné frustrace s některými aspekty ovládání softwaru.

Výstupem celého tohoto procesu bude vytvoření stabilní základny, do které bude jednoduché dělat zásahy. Důraz bude také kladen na flexibilitu pro budoucí úpravy a nové funkce. Základní myšlenkou celé práce je zamezení potencionálního dominového efektu nekontrolovaného nekonzistentního vývoje aplikace Umbrela, který by mohl znamenat její zánik a odchod uživatelů k jiným alternativám. Výsledkem by měl být stabilní systém používající moderní technologie, který by měl vydržet několik následujících let bez většího zásahu do jádra aplikace a zajistit tak pokračování poměrně komfortního, flexibilního a levného řešení pro kvantitativní marketingový výzkum na Mendelově univerzitě v Brně.

2 Přehled literatury a pramenů

2.1 Software reengineering

Všechny nástroje, které lidé ve svém životě používají, ať už k práci nebo zábavě, prochází určitou evolucí a neustále se vyvíjejí. Některé mají delší životnost, některé kratší, ale všechny jsou jednoho dne nahrazeny novou, upravenou verzí. To může být zapříčiněno spoustou různých faktorů. Ať už je to opotřebením, technologickým pokrokem nebo jen změnou trendů, každý nástroj je eventuálně zastaralý.

Pokrok na poli informačních technologií je neuvěřitelně rychlý a s rozšiřováním internetu společně s online komunikací a distribucí nebyl vývoj softwaru nikdy jednodušší. Nezávislost na fyzických mediích umožňuje spolupráci programátorů po celém světě a vývoj softwaru se neustále zrychluje. Také poptávka po softwaru roste svižným tempem, a to i díky existenci obrovského množství zastaralých programů a aplikací. Architektury takového softwaru jsou postupně nahrazovány modernějšími. Platformy, na kterých je nasazen, jsou neefektivní a nejsou schopny poskytnout dostatečnou stabilitu a rychlost. Z těchto, i z mnoha dalších důvodů, je dnes reengineering důležitou součástí životního cyklu jakéhokoliv dlouhodobě používaného softwaru (Alam a Padenga, 2010).

V šedesátých letech 20. století si lidská společnost začala rychle uvědomovat, jak vysoký dopad může mít software v mnoha jejich aktivitách. Navíc v důsledku mnoha problémů, s nimiž se potýkal vývoj softwaru, se tehdejší vývojáři všeobecně shodovali na tom, že dostupné techniky by se měly stát méně ad hoc. Místo toho by měly vycházet z teoretických základů a praktických disciplín, které jsou stanoveny v tradičních odvětvích inženýrství. Proto byla v roce 1968 vědeckým výborem NATO zorganizována první konference softwarového inženýrství. Jejím cílem bylo stanovení a následné používání spolehlivých inženýrských principů za účelem vytváření stabilního, efektivního a ekonomicky proveditelného softwaru. Mezi velkým množstvím navrhovaných aktivit byla údržba zařazena mezi post-produkční aktivity (Mens a Demeyer, 2008).

Reengineering je definován jako zkoumání, analýza a změna stávajícího softwarového systému a jeho rekonstrukce do nové podoby. Proces typicky zahrnuje kombinaci více činností, jako je reverzní inženýrství, vytvoření dokumentace, restrukturalizace, překlad do jiného jazyka či implementace nových technologií. Cílem je pochopit stávající software a následně jej znovu implementovat, zlepšit funkčnost, výkonnost nebo implementaci systému při zachování stávajících funkcí a připravit software na pozdější přidání funkčnost. Důležitou vlastností reengineeringu je to, že je k dispozici kód softwaru. Dostupnost předešlé verze kódu jasně určuje funkce systému a nemůže nebo by nemělo dojít k nepochopení a následné chybné interpretaci a realizaci softwaru (Rosenberg a Hyatt, 1997).

Proces reengineeringu není ve všech případech stejný. To, jaký přístup bude použit, závisí na druhu systému, jeho stáří nebo použitých technologiích. Jedním z nejčastěji používaných přístupů je tzv. Big Bang approach. Jak už může název

napovídat, tento přístup je specifický tím, že starý systém je zcela nahrazen novým bez jakékoliv závislosti na starém systému. Není třeba vytvářet žádná rozhraní mezi komponentami. Tento přístup se používá zejména v případě, kdy je potřeba vyřešit problém starého softwaru najednou (přechod na novou architekturu) a nevyplatí se investovat zdroje do jeho postupné renovace (Leach, 2016). Většina autorů se shoduje na riskantnosti tohoto přístupu, avšak ve spoustě případů, kdy se přechází na zcela novou architekturu, je často jediným východiskem. Pokud není průběh vývoje řádně testován a kontrolován, může se stát, že nástupce starého softwaru bude ještě v horším stavu než jeho předek (Pressman, 2005).

Přesným opakem zmíněného přístupu je přístup inkrementální. Stejně jako v ostatních disciplínách informatiky v několika posledních letech je silně prosazován inkrementální přístup, který se vyznačuje postupným nabalováním menších funkčních celků. Po implementaci a řádném otestováním je takový kus kódu přidán k celkovému řešení a následuje přesunutí pozornosti k jiné části softwaru. Využití inkrementálního postupu přináší lepší organizaci kódu, snadnější testování a odhalování chyb. Samotný postup implementace už pak nutí k izolovanosti jednotlivých částí kódu a zvyšuje tak modularitu celého systému a umožňuje jednodušší rozšiřitelnost v budoucnu (Pressman, 2005).

Reengineering je nedílnou součástí životního cyklu jakéhokoliv dostatečně starého softwaru. Vzhledem k neustálému vývoji informačních technologií i samotných principů se Inkrementální přístup se jeví jako nejlepší kandidát říkající, jak při práci postupovat, avšak ne vždy je možné ho plně využít. Pokud však taková situace nastane, a i přes všechnu snahu je jasné, že je zapotřebí radikálních změn, aby se zajistila kontinuita vybraného softwaru, pak nikde není stanoveno a nic nebrání tomu, aby byly prvky inkrementálního reengineeringu zapracovány do procesu. I v případě změny platformy nebo architektury se stále dá řídit pravidly postupného přístupu a kombinací obou přístupů je možné využít všech výhod, které poskytují.

2.2 Reverse engineering

Pojem reverse enginneering a reengineering jsou často chápaný jako jedno a totéž. I když jsou si oba procesy velmi podobné a sdílejí spoustu společných vlastností, často ani samotný vývojář nedokáže rozlišit, který z těchto dvou přístupů používá. Oba pojmy jsou však definovány odděleně. Asi nejběžnější definice reverzního inženýrství je následující. Reverzní inženýrství je proces přepsání zdrojového kódu, který není přístupný, nebo dostupný v průběhu vývoje. Vývojář zná pouze funkce softwaru a snaží se je zreplikovat do nového systému (Wang, 2010).

Zjednodušeně řečeno, reengineering je nová implementace již existujícího softwaru za účelem zlepšení funkcionality, výkonu či udržitelnosti, kdežto podstatou reverse enginneringu je analyzovat existující systém, naučit se jeho funkce a případně vytvořit nový produkt se stejnými čí podobnými vlastnostmi. V obou případech však existuje snaha o vytvoření něčeho lepšího.

2.3 Agilní metodologie

Agilní metodologie vývoje softwaru se objevily v polovině 90. let jako protipól tradičních metod. Vznikly hlavně kvůli omezení, která byla vynucována přísně plánovaným řízením a zadáním. Ty jsou charakteristické u tradičních modelů vývoje softwaru (Abdalhamid a Mishra, 2010).

Agilní řízení projektu se používá zejména pro vývoj softwaru u malých a středních projektů, které potřebují velmi málo dopředného plánování. Agilní přístup je založen na iterativním a inkrementálním stylu, umožňující neustále provádět změny v implementaci softwaru, jsou-li nutné. Na rozdíl od tradičních metod, které vyžadují přesnou specifikaci daného problému, přesné plánování a striktní harmonogram, se agilní přístup skládá z mnoha iterací či etap, které jsou samostatné podprojekty a obsahují všechny fáze vývoje softwaru. Na konci iterace je její výstup zhodnocen, optimalizován a zařazen do celkového projektu (Banica aj., 2016).

Další důležitou součástí projektů řízených pomocí agilního přístupu je neustálý kontakt se zadavatelem nebo koncovým uživatelem. Spolupráce zadavatele a vývojářského týmu je klíčovým faktorem k úspěchu agilních metod. Pomáhá lepé specifikovat problémy a předchází špatné interpretaci zadání. Tyto vlastnosti agilního vývoje vedou k vyšší úspěšnosti projektů oproti tradičním přístupům (Banica aj., 2016).

Tabulka 1: Porovnání úspěšnosti mezi agilním a tradičním vodopádovým projektem (Hastie a Wojewoda, 2015)

Metoda	**Úspěšný**	**Zpochybněný**	**Neúspěšný**
Agilní	39%	52%	9%
Vodopádová	11%	60%	29%

Agilní metodiky se jeví jako jednoznačně lepší přístup pro vývoj malých a středních projektů. Od tradičních metod se primárně liší v oblasti flexibility a zaměření na zákazníka. Veškeré úsilí je vloženo do snahy uspokojení zadavatele a práce je řízená čistě jeho potřebami. Zatímco klasické vodopádové metodologie nedovolují vývojářům návrat do etap, které byly již ukončeny, agilní metodologie jsou založené právě na přizpůsobování se novým situacím bez nutnosti přepsat veškerý zdrojový kód softwaru. Další značná výhoda je testování a odhalování chyb. Tím, že každá etapa je vlastně malý projekt a skládá se ze všech fází, je testování prováděno průběžně a odhalování chybných kusů kódu a jejich oprava je realizována v rámci jednotlivých cyklů. To značně zvyšuje kvalitu a snižuje náklady na vývoj softwaru a více uspokojuje zákazníky (Sengupta aj., 2014).

Agilní vývoj však není jednoduchou odpovědí na všechny problémy v procesu vývoje softwaru. Existují i situace, kdy takový přístup není vhodný, a byly zdokumentovány případy, kdy je vhodnější použíit vodopádové metodologie. Negativní vliv na úspěch agilního přístupu mohou mít velké projekty s více něž dvaceti členy v týmu. S růstem týmu klesá úspěšnost a využití agilního řízení pro takové projekty

je spíše ve fázi výzkumu. Nicméně existují případy, ve kterých se jej podařilo úspěšně implementovat i v rozsáhlých týmech. Dále byly také problémy u týmů, které přecházely z klasických metod na agilní. Pokud členové týmu neakceptují dobrovolně nový styl řízení, skoro určitě bude mít agilní vývoj negativní dopad na výsledky projektu. To je zapříčiněno vyšší odpovědností každého člena a nízkou organizací, někdy až chaosem uvnitř týmu (Landry, 2011).

Agilní přístup je jednoznačně správnou volbou pro vývoj softwaru v malém týmu. U jednočlenného týmu se dá spekulovat o tom, že je to dokonce nutné. Nedostatek dohledu ze strany ostatních vývojářů a zákazníků může vést k nedodržování určitých standardů jako je například psaní unit testů. To nevyhnutelně bude snižovat kvalitu výstupu projektu a může vést i k celkovému neúspěchu. Naopak použití vodopádového přístupu je nereálné, protože jeden člověk nebude nikdy schopen naplánovat celý proces. Průběžné konzultace se zadavatelem a doplňování chybějících znalostí v průběhu projektu, je u jednočlenného týmu nevyhnutelné. To vede k dodatečným změnám a vracení se k předchozím fázím. Dodržování alespoň základních principů jakéhokoliv zástupce z agilních metodik je esenciálním faktorem ke správnému uchopení zadání a úspěšnému dokončení softwaru.

2.4 Údržba softwaru

Jak již bylo řečeno, údržba softwaru je nedílnou součástí jeho životního cyklu. Obvykle doba údržby dokonce několikanásobně převyšuje dobu samotné implementace. Softwarová údržba se navíc značně liší například od údržby hardwaru. Pokud například na serveru dojde místo na disku, jednoduše se přidá další disk a problém je vyřešen. Pokud však v našem programu něco nefunguje tak, jak by mělo, odstranění této chyby není tak přímočaré. Uvedení nějakého algoritmu do původního stavu tak, jak by se opravil hardware, není z jednoduchého důvodu možné. Původní stav byl jednoduše vadný. Chyba se musí najít, což může být samo o sobě náročné. Po identifikaci je nutné vymyslet nové řešení. A pokud je software navržen špatně, z takového nového řešení mohou vzniknou další a další chyby. Nemusí se však jednat pouze o opravy chybného kódu. Optimalizace, vylepšování, přidávání nových funkcí a odstraňování zbytečných. To vše a mnoho dalšího je součástí údržby softwaru (Grubb a Armstrong, 2003).

Existuje spousta různých způsobů jak udržovat software. Většina z nich se dá rozdělit do čtyř kategorií. Žádná údržba, tradiční, diskrétní a průběžná. První z nich je žádná údržba. Může se zdát že jde o nesmysl, nicméně existují případy, kdy je dopředu jasné že údržba nebude možná. V minulosti většina softwaru ani nemohla mít údržbu, protože jakmile se nějaký program jednou naprogramoval a dostal se ke koncovým uživatelům, tak už nebylo možné ho nijak měnit. Takové programy byly podrobeny mnohem striktnější kontrole a počet chyb byl v porovnáním s dnešním softwarem mnohem menší. S rozvojem internetu již však není nutné dodávat tak kvalitní software. Pokud se nějaký nedostatek nalezne, jednoduše se vydá oprava a software se mnohdy automaticky aktualizuje. U některého typu softwaru není

možná údržba v jakékoliv formě. Jednoúčelové programy, které jsou uvnitř nějakého zařízení, nejsou jednoduše dostupné. Asi není příliš obvyklé, aby jednou za rok přišel domů k zákazníkovi údržbář a nahrál mu do pračky nový kus kódu. Poslední variantou, kdy je možné se obejít bez jakékoliv údržby, je případ, kdy se podaří vytvořit perfektní software. To je sice krásná představa, ale v reálném světě je dost nepravděpodobná (Stachour a Collier-Brown, 2009).

Dalším typem je tradiční údržba. Ta je primárně zaměrená na odstraňování konstant, globálních proměnných, správné pojmenování proměnných a další konvence. Kód je psán takovým způsobem, aby při změně jedné části nebyla ovlivněna žádná jiná (Chemuturi, 2010). Tato pravidla by se měla uplatňovat už při tvorbě softwaru a ne až po jeho zveřejnění, nicméně ne vždy jsou dodržována a je nutné je mít stále na paměti.

Diskrétní údržba se od ostatních liší v intervalech a velikosti nových oprav. Mezi jednotlivými aktualizacemi je dlouhá prodleva a každá přináší velké množství úprav. Někdy může být vydána úplně nová verze, která kompletně nahradí původní řešení, jindy takzvaný "patch" přinášející spoustu změn a nových funkcí. Tento způsob může být pro uživatele nepříjemný. Musí dlouho čekat na opravení chybného chování softwaru, a když už si na něj zvykne, přijde nová aktualizace, která vše od základů změní, a on se musí opět učit (Stachour a Collier-Brown, 2009).

V předchozích příkladech se předpokládá, že vývoj softwaru někdy skončí. Stanoví se datum, kdy se vydá plná verze, začne distribuce a program se už prakticky nebude měnit a někdy v budoucnu se vydá jeho následník. To však v dnešní době přestává být pravda.

V posledních letech se objevuje, respektive vrací, takzvaný Software as a Service(SaaS). SaaS je cloudově založený distribuční model pro systémy a software. Na rozdíl od klasického Enterprise řešení jsou v tomto modelu za údržbu, provoz, bezpečnost a další aktivity odpovědní jejich provozovatelé a ne uživatelé. V šedesátých letech IBM a další výrobci sálových počítačů poskytovali podnikatelské služby klientům, kteří neměli zdroje a odborné znalosti k tomu, aby interně začlenili určité podnikové procesy. Nabízeli služby, jako je ukládání dat a využívání výpočetní síly jejich počítačů. Toto je považováno za předchůdce dnešního SaaS. (Hogan, 2017). Aplikace dodávané prostřednictvím modelu SaaS se také často nazývají web-based software. Obliba SaaS je zapříčiněna zejména širokou dostupností internetu a také díky vyloučení nutnosti vlastnit a udržovat vlastní hardware. Hlavně díky absenci hardwaru jsou služby až o 80 % levnější než tradiční modely (Melvin, 2009).

Průběžný model údržby úzce souvisí právě se zmiňovaným SaaS. Vzhledem k nutnosti neustálé podpory a aktualizaci nejrůznějších částí softwaru je samozřejmé, že ve stejném duchu musí probíhat i údržba kódu. Continuous integration (CI), continuous delivery a deployment jsou součástí takového procesu údržby. CI je způsob vývoje a integrace nových funkcí či úprav. Skládá se z osvědčených postupů, které pomáhají minimalizovat nekonzistenci kódu a omezit výskyt nových chyb integrací nového kusu kódu. Před každou integrací je nutné, aby byly spuštěny všechny automatizované testy a hlavně aby všechny byly splněny. Kontroluje se také čitelnost

kódu a dodržování stanovených konvencí. Všechny tyto a další činnosti jsou automatizovány a jakákoliv změna libovolným členem týmu musí projít právě takovou integrací, jinak ji není možné zapracovat do výsledného produktu (Duval, 2008).

Dalším krokem po integraci je nasazení nových změn na servery, aby byly dostupné všem uživatelům. Běžným postupem v minulosti bylo ruční zkopírování nových souborů na produkční server, manuální záloha předchozí verze, databáze a dalších částí softwaru. Tento postup je poměrně časově náročný a i malá chyba může způsobit poměrně značné finanční škody. Continuous delivery a deployment spolu úzce souvisí a přispívají k automatizaci procesu. Ta může dále zjednodušit a zaručit správný chod služby v případě nutnosti provedení změn. To se týká dvojnásobně SaaS vzhledem k tomu, že změny jsou mnohem frekventovanější než u standardního softwaru. Součástí takového procesu mohou být činnosti jako záloha předchozí verze kódu v případě, že by změny způsobily nějaké chyby, migrace a záloha databáze nebo v případě webu v průběhu údržby uživatelům zobrazit stránku s informacemi o průběhu údržby a předpokládané znovuspouštění služeb. Činnosti v rámci tohoto procesu se samozřejmě liší podle charakteru softwaru, ale ve všech případech přináší kvalitnější a plynulejší přechod na novou verzi programu. Naneštěstí Continuous deployment není vždy možné, protože k provedení některých zmíněných činností je nutné mít například administrátorský přístup k serveru, na kterém je služba spuštěná, nebo mohou existovat organizační komplikace (Rossel, 2017).

Je zřejmé, že s nástupem SaaS a webových aplikací, automatizace údržby zdrojových kódu, zejména testování a striktní dodržování konvencí, je důležitým pilířem při stavbě stabilního systému. A i u jiných modelů softwaru není vůbec na škodu se těmito pravidly řídit, protože tato doporučení a pravidla mají bezesporu kladný dopad na celkový výstup většiny projektů. Navíc automatizace jako taková může ušetřit velké množství finančních prostředků. Některé nástroje, jako například Travis CI, jsou dokonce zcela zdarma, takže se ušetří nejen na lidské práci a úsilí může být směřováno na jiné aktivity, ale i na nástrojích, které tuto automatizaci umožňují.

2.5 User experience a uživatelská rozhraní

Neexistuje žádná univerzální definice pro pojem User experience. Definice od různých autorů jsou rozčleněny od zkoumání neurologických fenoménů až po makroekonomické chování nejrůznějších subjektů. Shodují se však na jednom společném prvku, kterým je uživatelské rozhraní. V některých případech takovému rozhraní přiřazují velkou váhu, někdy menší. Dá se však říct, že do jisté míry je jím user experience vždy ovlivněno, protože uživatel nějakým způsobem musí s daným produktem interagovat (Kuniavsky, 2010).

Spousta lidí si pod pojmem návrh uživatelského rozhraní představí estetický vzhled určitých ovládacích prvků. Dobře navržený produkt je takový, na který se hezky kouká nebo je příjemné se ho dotýkat. Další obecně akceptovaná definice designu produktů jsou jejich funkce. Nějaký produkt dělá vše, co má a dělá to dobře. Nůžky, které stříhají, pero co píše. To jsou produkty, které funkcionálně odpovídají

obecným předpokladům společnosti. User experience spojuje oba tyto přístupy do jednoho. Tlačítko na kávovaru musí nejen dobře vypadat, ale musí také perfektně splňovat svojí zamýšlenou činnost (Garrett, 2011).

Jenifer Tidwell(2011) definovala velkou skupinu návrhových vzorů pro uživatelská rozhraní. Tyto vzory se nezbytně nevztahují pouze na software a mohou být inspirací i pro nejrůznější stroje a techniku. Tyto vzory jsou popisem osvědčených postupů vedoucích k lepšímu uživatelskému rozhraní. Zachycují společná řešení obecně známých problémů, se kterými se setkáme takřka vždy při vývoji libovolného softwaru. Nejedná se o přichystané komponenty a jejich implementace se může mírně lišit podle charakteru problému, který je řešen (Tidwell, 2011). Nicméně jejich použití bezpochyby zvýší kvalitu navrženého rozhraní a posune projekt správným směrem.

Dalším prvkem, který je nutné se při návrhu uživatelského rozhraní zohlednit, je samotný uživatel. Je sice hezké, že existují nejrůznější návrhové vzory pro vzhled a umístění prvků na obrazovce, ale co když navrhujeme software pro zrakově postiženého uživatele. Všechny tyto doporučení jsou nepoužitelné. Proto dříve, než se začne rozhraní stavět, je nutné se snažit co nejlépe poznat budoucí či stávající uživatele, aby se nestalo, že jim je doručen produkt, který nemohou využít.

2.6 Marketingový výzkum

Aplikace Umbrela se zabývá sběrem primárních dat pro kvantitativní marketingový výzkum. Marketingový výzkum je disciplína marketingu, která se dívá na různé aspekty uspokojování potřeb spotřebitelů. Takový výzkum pomáhá managementu firmy (nebo jiným zainteresovaným osobám) definovat marketingový mix, zvolit správnou strategii podniku či pomoci k vyrovnání nabídky a poptávky na trhu (Bradley, 2013).

Aby tito jedinci mohli provádět některá rozhodnutí, potřebují informace o daném trhu. Informace mohou z dat získat nejrůznějšími statistickými, ekonometrickými, psychologickými nebo jinými metodami. Tato data jsou sbírána přímo od spotřebitelů a zákazníků. Existují dvě hlavní metody sběru dat. Jedná se o kvalitativní a kvantitativní marketingový výzkum (McDaniel a Gates, 2013).

Kvalitativní výzkum je zaměřen na sběr velice detailních a podrobných dat. Jsou vysoce heterogenní a jejich aplikace na širokou veřejnost je takřka nemožná. Výzkumníci nejčastěji vedou nestrukturovaný rozhovor se subjekty a snaží se identifikovat pocity a podvědomé chování spotřebitelů. Samotný rozhovor bývá často prováděn vysoce kvalifikovanými psychology, kteří dokáží respondenta přiznat skutečnosti, které si sám ani nemusí uvědomovat. Přesným opakem kvalitativního výzkumu je výzkum kvantitativní. Ten je zaměřen na sběr velkého množství dat a dá se dobře zobecnit na populaci. Jeho cílem je najít podobné vzory chování za účelem optimalizace nabídky produktů či služeb. Data jsou nejčastěji sbírána formou dotazníků, ať už v papírové nebo elektronické podobě (Aaker, 2013).

Systém Umbrela je webová aplikace, která jejím uživatelům dovoluje vytvořit a spravovat si elektronický dotazník, rozeslat ho respondentům pro vyplnění,

prohlížet si nasbíraná data nebo si je stáhnout pro další potřeby marketingového výzkumu.

3 Metodika

I přes to, že zadavatel vyjádřil svoji nespokojenost se současným stavem aplikace a vyžaduje zcela novou implementaci aplikace, je nutné provést několik aktivit před začátkem jakékoliv práce. Nejdříve bude potřeba analyzovat současný stav aplikace. Po nastudování potřebných materiálů a osvěžení znalostí týkajících se moderních způsobů vývoje webových aplikací bude prozkoumána současná implementace řešení.

Podrobná analýza současného stavu systému dále upřesní požadované funkce a procesy, které bude nutné zreplikovat. Jakékoliv identifikované nedostatky a funkční chyby, mohou pomoci předejít k jejich nechtěné replikaci do nové implementace a zvýšit tak kvalitu výstupu práce.

Dalším krokem bude volba vhodných technologií k implementaci. Systém je již několik let starý a během posledních let došlo ke značné transformaci v celém oboru informačním technologií. Je proto možné, a také vysoce pravděpodobné, že některé stávající technologie jsou již zastaralé a jejich výměna je nutným krokem ke kvalitnější aplikaci.

Po výběru technologií bude nutné si udělat funkční obraz nové aplikace, následně začít s implementací a průběžně testovat zdrojový kód. Po implementaci bude provedeno zhodnocení výsledků.

3.1 Aplikace Umbrela

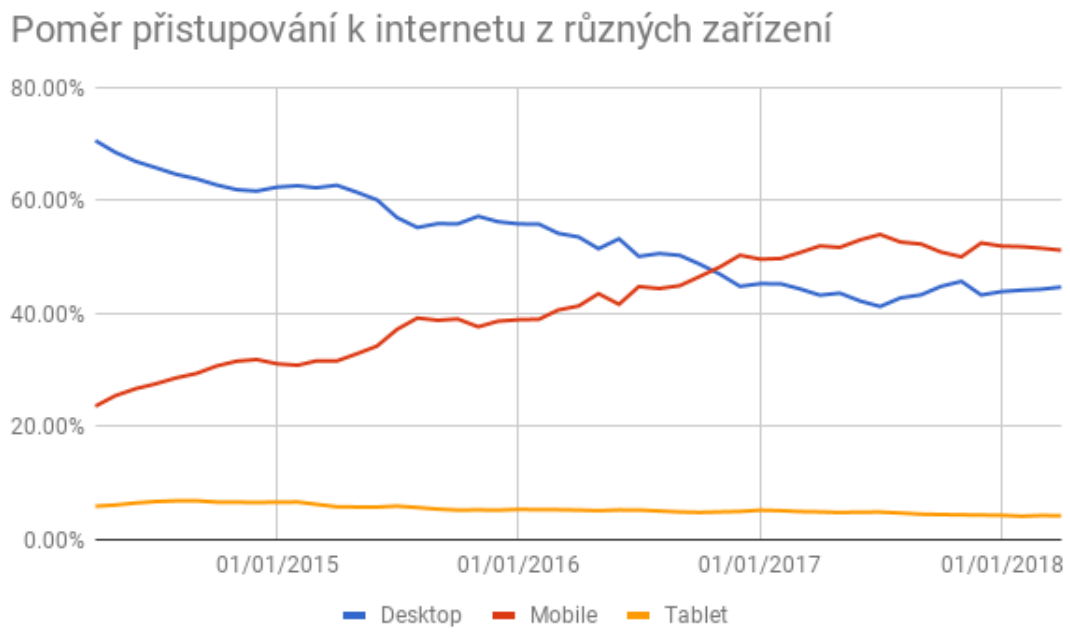
Jak již bylo uvedeno, Umbrela je aplikace, kterou používají převážně studenti a zaměstnanci Mendelovy univerzity v Brně ke sběru primárních dat pro kvantitativní marketingový výzkum. První verze tohoto systému byla vyvinuta před více než patnácti lety tehdejšími studenty univerzity. V této době byl internet ještě poměrně mladý a neexistovalo na něm mnoho aplikací, které by poskytovaly podobné služby. Univerzita se rozhodla tento systém nadále udržovat a používat ho pro studijní a výzkumné účely. O aplikaci se dále starali studenti, avšak po odchodu autorů se poměrně rychle začaly ztrácet technické detaily o aplikaci. Nezkušenost některých vývojářů, špatná dokumentace a absence supervize nad procesem vývoje aplikace vedly k postupné degradaci zdrojového kódu. V aplikaci se začalo vyskytovat velké množství chyb. V nedávné době proběhly pokusy o rekonstrukci celého systému, všechny však neustále stavěly na původním kódu. Ten byl za dobu své existence nepřehledný, stavěl na starých technologiích a architekturách. Všechny tyto pokusy tedy selhaly. I když se některé kritické chyby podařilo odstranit, na jejich místě se objevily nové. Ani kvalita kódu se žádným způsobem nezlepšila, protože při porovnání obou verzí se zjistilo, že prakticky všechen zdrojový kód byl identický. Jediný pozitivní výstup těchto pokusů bylo nové schéma databáze, které odstranilo masivní, nestrukturované tabulky. Tato operace však systému uštědřila další masivní ránu. Tou byla ztráta všech nasbíraných odpovědí. Jediné co zůstalo v databázi, byly uživatelská data a struktura dotazníků. Tento fakt dále znehodnotil aplikaci v očích uživatelů a zapříčinil jejich další odliv. Proto padlo rozhodnutí reengineeringu celé

aplikace. Ten by měl být proveden od úplných základů, aby se dostala na úroveň moderních webových aplikací. V této části práce, budou popsány, nedostatky aplikace a způsob jakým bude následně nová verze implementována.

3.2 Identifikace nedostatků

3.2.1 Uživatelské rozhraní

Uživatelské rozhraní pravděpodobně není nejdůležitější částí aplikace. Krásný a inovativní vzhled není k ničemu, pokud je aplikace postavena na špatných základech a není možné ji bez problému používat. Nicméně je to první aspekt, kterého si každý všimne. Ať už obyčejný uživatel nebo profesionální designér, už na první pohled rozpozná, že i tato stránka byla dlouho dobu zanedbaná. Používání obrázků pro ohraničení nejrůznějších sekcí, chaotické umístění prvků. I logo ve formě obrázku, ke kterému nebyl nikdo schopný vysvětlit spojení s aplikací. Projekt byl vytvořen v době, kdy internet na mobilních telefonech byl raritou a tomu odpovídá i optimalizace zobrazení na menších obrazovkách. Žádná není. V době, kdy objem dat stažených z internetu na mobilních telefonech či tabletech je vyšší než na počítačích, je to obrovský nedostatek a ztráta obrovského množství potenciálních uživatelů (Marcotte, 2014).



Obrázek 1: Poměr přistupování k internetu na různých zařízeních (StatCounter, 2018a).

Bez jakékoliv nutnosti dlouhého přemýšlení je jasné, že celé uživatelské rozhraní bude muset být od základů předěláno.

3.2.2 Funkční chyby

Celá aplikace se skládá z mnoha různých formulářů pomocí, kterých uživatel komunikuje se serverem. Ve většině případů fungovaly správně, a pokud byly nějaké chyby identifikovány, jednalo o menší nedostatky, na které si uživatele v průběhu používání zvykli. Což samozřejmě není omluva jejich existence, ale zásadním způsobem neovlivňovaly průběh používání aplikace. Existují však dvě kritická místa, která jsou nejdůležitější pro celou aplikaci. Jedná se o editor dotazníku a část pro jeho vyplňování. Všechny ostatní funkce viditelné uživatelem nejsou v porovnání s těmito dvěmi důležité. I přesto se hlavně v editoru dotazníku podařilo identifikovat chyby, kde část z nich může způsobit ztrátu i několika hodin práce.

Samotný editor je vysoce interaktivní část aplikace a pro jeho správnou funkčnost bylo zapotřebí použití JavaScriptu. Jedna z prvních verzí byla napsána v čistém JavaScriptu a postupně se přešlo na JQuery. Předposlední verze, implementovaná pomocí JQuery, neobsahovala sice žádné zásadní chyby, ale představovala obrovské bezpečnostní riziko pro celou aplikaci. Dovolovala uživatelům vkládání vlastního HTML kódu do dotazníku, včetně tagů jako jsou iframe nebo vlastní skripty a další. Výsledek se nejenže zobrazoval respondentům, ale také se vše ukládalo do databáze, což údajně v minulosti způsobilo několik problémů. Tato možnost byla odstraněna s nástupem nové verze a celý editor byl napsán pomocí knihovny React. Technologicky se jednalo o obrovský skok dopředu, nicméně to mělo za důsledek několik problémů. Oficiálně tato knihovna byla v té době v beta verzi. Bylo jasné, že se bude rychle měnit, a to se také stalo. V momentě nasazení nové aplikace na produkční servery už byla knihovna v používané verzi zastaralá a prošla mnoha změnami, které již nebyly kompatibilní, a bylo jasné, že celý editor bude nutné aktualizovat, i kdyby fungoval bezchybně.

Dalším problémem byl způsob ukládání průběhu práce na dotazníku. Dotazník byl ukládán nárazově po velkých kusech. V momentě, kdy se z jakýchkoliv důvodů nepodařilo dotazník uložit, i třeba jen kvůli jednomu řetězci v nějakém textovém poli, byl ztracen celý průběh práce. Tomuto faktu ani nepomohla skutečnost, že délka session na serveru byla nastavená na jednu hodinu. Pokud měl někdo zapnutý editor déle jak hodinu, tak po uplynutí toho časového úseku nebylo možné dotazník vůbec uložit. Samotná asynchronní komunikace se serverem neobnovila session a často tak docházelo ke ztrátě dat. Uživatel navíc nebyl po vypršení ani přesměrován na přihlašovací formulář a bez automatického ukládání si mnohdy všiml problému až v momentu, kdy byla práce hotová. On si však nemohl dotazník uložit a o vše přišel. Toto byl jeden z nejčastějších problémů, na který si uživatelé stěžovali. Další problém v editoru bylo chybné ukládání pořadí archů, otázek a možností. Pokud chtěl uživatel změnit pořadí některého z těchto objektů, občas se stávalo, že aplikace přiřadila dvěma stejným objektům stejné pořadové číslo. To je kontrolováno i na úrovni databáze. Tím pádem dotazník nešel uložit a uživatel si toho nemohl nijak všimnout. Aplikace nešla opět uložit a uživatel se opět dostal do stejné situace, která byla popsána dříve. Takových problémů existovalo v editoru více. Špatná validace

na straně klienta vedla k posílání chybných dat, a to znemožnilo uložení celého kusu posílaných dat a k jejich následné ztrátě. Vzhledem ke staré verzi knihovny, žádné dokumentaci a celkově chybné implementaci bylo rozhodnuto, že editor bude muset být navržen a zrealizován znovu. Také by nebylo moudré se inspirovat existujícím kódem, jelikož by to mohlo vést ke stejným chybám.

Vyplňování dotazníku probíhalo vyplňováním klasického formuláře na webu. Několikrát byl hlášen problém s chybným uložením odpovědí do databáze. Nicméně nebyl k dispozici dostatek podkladů a případů, kde by se podařilo identifikovat závažnější problém. Jednalo se pravděpodobně o izolované problémy na straně serveru, které byly ojedinělé. Asi největším problémem této části aplikace byl její vzhled, především absence responsivního designu. To mohlo často odradit respondenty k úspěšnému vyplnění často dlouhých dotazníků. Ale to je společný fakt pro celou aplikaci.

3.2.3 Architektura a konvence aplikace

MVC (MVP, MVW) nebo jeho obdoby je v dnešní době univerzálně akceptovaný návrhový vzor pro tvorbu webových a desktopových aplikací. Poprvé se objevil v roce 1978 a od té doby se stal více méně standardem (Walter, 2008).

Jedná se o logické rozdělení zdrojového kódu na tři hlavní celky. Těmi jsou model, view a controller. Model představuje data a manipulaci s nimi, view se stará o prezentaci těchto dat uživatelům a controller obstarává akce, které může uživatel s aplikací provádět. Existuje obrovské množství frameworků ve spoustě různých programovacích jazyků, které tento model přímo podporují a některé dokonce vynucují (Kunjumohamed aj., 2016).

Tento vzor tedy není žádnou převratnou myšlenkou a zdá se logické ho použít, pokud neexistuje nějaký závažný důvod, který by dokázal, že jeho použití uškodí aplikaci. Ze zdrojového kódu je však patrné, že tento model není dodržen. Není viditelná žádná logická organizace kódu. Můžeme se pouze domnívat, že při původní tvorbě tohoto softwaru nějaká snaha o organizaci existovala, ale v během vývoje a údržby postupně mizela. Část view je jediný blok MVC, který je relativně dobře oddělený od ostatních, nicméně jeho implementace je značně zastaralá. Absence responsivního designu a rozdělení sekcí s pomocí tabulek jsou některé z hlavních nedostatků prezentační části aplikace.

Hranice mezi datovou a funkční částí aplikace je velice propletená. Často není snadné určit o jakou se jedná. V aplikaci existují Data access object (DAO), které se chovají jako rozhraní mezi aplikací a její databází (Jenkov, 2014). Ty jsou rozmístěny napříč celou aplikací, místo aby byly sdruženy na jednom místě. To samo o sobě však nemusí být chybou. Nicméně je pak podivuhodné, že databáze je přímo dostupná z dalších částí aplikace. Controllery (API, Users, Auth...) přímo přistupují k databázi, místo toho, aby přenechávali tuto práci modelům. I jiné části aplikace, které nejsou ani z modelové části, ani controller, také obsahují SQL dotazy, které přistupují k databázi. Dodržování konvencí už na úrovni MVC je důležitý krok, který

je základem k úspěšnému dlouhodobému vývoji takového softwaru. Jejich nedodržování bude mít zcela určitě za následek zmatení nových členů týmu, kteří nebudou vědět, jestli vůbec nějaké konvence existují. Pokud na projektu začne pracovat nový člověk a uvidí, že nezáleží na tom, kam svůj kód umístí, pak se brzy v kódu začne projevovat chaos a bez rychlého zásahu může vést k duplicitám, chybám a frustraci členů týmu. Toto je zásadní problém a s největší pravděpodobností je jedním z hlavních důvodů, který vedl k postupné degradaci celého systému a výskytu nových chyb, až ke konečnému rozhodní o přestavění celé aplikace.

Dalším problémem, který dále zhoršuje kvalitu kódu je nedodržování klasických konvencí. Jedním z takových případů jsou takzvané magické konstanty. Jedná se o přímo vepsané konstanty v kódu, které jsou stejné na více místech, nicméně jejich hodnota není udržována v žádné proměnné. Tyto literály vedou k duplicitám a k chybám, pokud se při nějaké změně nepodaří aktualizovat každý jejich výskyt (Arsenovsky, 2009). I tyto konstanty se v aplikaci bohužel vyskytují na více místech. Jedním z příkladů je definice různých typů otázek. Podivuhodné však je, že i když se ve zdrojovém kódu magické konstanty vyskytují, existuje však i definice těchto typů v souboru "QType.pm", které obsahují stejné řetězce. Přitom však na některých místech nejsou použity. S ohledem na umístění souboru a jeho název se dá předpokládat, že někteří vývojáři o této konvenci ani nevěděli, a proto použili zmiňované magické konstanty. To může být důsledkem více problémů. Nekompletní dokumentace, vývojář nebyl nikým obeznámen s aplikací, neexistovala žádná kontrola nad tím, jaký kód bude zaveden do produkční verze nebo jednoduše neochota konvence dodržovat.

3.2.4 Programovací jazyk Perl

Programovací jazyk, který byl použit pro tuto aplikaci, odpovídá jejímu stáří. Perl byl jedním z prvních programovacích jazyků používaný pro tvorbu webových aplikací. V dnešní době už se od toho jazyku upouští a na poli webových aplikací ho nahrazují zejména jazyky Ruby a Python (Sroka, 2015). Po krátkém hledání na nejrůznějších portálech či článcích dostupných z internetu se mnoho vývojářů shoduje na několika následujících faktech.

Jedním z důvodů, proč se dnes tento programovací jazyk používá pro webové aplikace, jsou takzvané "legacy" aplikace. Právě proto, že byl Perl jedním z prvních programovacích jazyků používaný pro webové aplikace, existuje spousta starých systémů napsaných právě v tomto jazyku. Tudíž je nutné je nějakým způsobem udržovat (Zjr, 2012).

Dalším faktem je stagnace toho jazyka. Poslední verze jazyka, který je aplikací používán, Perl 5, byla zveřejněna v roce 2000. Od té doby se pracuje další verzi, Perl 6, ta je však fundamentálně odlišná a převedení aplikace by znamenalo kompletní přepsání od základů, stejně jako při použití úplně jiného programovacího jazyka. Aplikace tedy běží na dnes 18 let staré verzi bez jakékoliv vidiny aktualizace, která by například mohla vylepšit její výkon (Myhrvold, 2014).

Mezi komunitou také panuje shoda nad tím, že v dnešní době je tento programovací jazyk vhodný pro jiné účely než webové aplikace. Administrace UNIX systému, shell skripty, data mining nebo statistická analýza (Zjr, 2012).

Špatná podpora objektů ve verzi 5, nečitelná syntaxe a překvapivě i volnost, která umožňuje použití velkého množství přístupů k dosažení stejného výsledku. To jsou nejčastěji zmiňované příčiny vývojářů na portálech jako StackOverflow, StackExchange a GitHub, kterým je dáváno za vinu, proč je Perl na ústupu. Mezi vývojáři tento jazyk dokonce vyhrál hlasování o nejvíce nenáviděný programovací jazyk (Claburn, 2017).

Perl tedy zřejmě není nejvhodnější programovací jazyk, který by měl být použitý pro vytvoření nové webové aplikace. Navíc jeho hodnocení ani nepřispívá fakt, že systém má být nadále udržován studenty Mendelovy univerzity v Brně, kde se tento jazyk vyučuje jen v jednom kurzu, a to spíše v kontextu dolování dat a administrace serveru. V zájmu zajištění kontinuity projektu bude proto nutné vyhledat jinou alternativu.

3.2.5 Organizace a vývojový tým

Všechny výše popsané nedostatky jsou pouze důsledky absence dlouhodobého týmu a řádné organizace celého projektu. Podle slov vedení Ústavu marketingu a obchodu první verze aplikace obsahovala podstatně menší množství funkcí, než nyní. Dá se předpokládat, že v té době výše zmíněné nedostatky neexistovaly a že se do aplikace dostaly až dalším vývojem.

Po odchodu původního tvůrce systému byl zamýšlený model vývoje a údržby následující. O systém se měl starat vždy jeden student Mendelovy univerzity, zejména studenti informaticky zaměřených oborů. Takový student by měl po nějakou dobu provádět údržby a případně změny v systému. Po určité době by se měl do týmu připojit další člen, kterému by jeho předchůdce předal všechny znalosti o projektu a postupně i zodpovědnost nad celým projektem. Problém je, že ne vždy se podařilo takové plynulé předání zrealizovat. V takovém případě měla na řadu přijít řádná dokumentace. Ta však neexistovala, až na výjimku občasných komentářů uvnitř zdrojového kódu. Supervize ze strany zaměstnanců Ústavu marketingu a obchodu také nebyla možná, protože jeho dlouhodobí zaměstnanci nemají vzdělání v oboru informatiky. Kromě zamýšlené funkcionality systému nemohou studentům, kteří systém přebírají, poskytnout žádné informace, které by jim pomohly se s projektem seznámit a pokračovat tak v práci stejným způsobem jako jejich předchůdci. Tím se značně narušila konzistence celého projektu a každý vývojář si zaváděl vlastní konvence a buď úmyslně, nebo nevědomky nedodržoval již existující. Tato inkonzistence následně vedla k výše zmíněným problémům a nakonec k rozhodnutí o přepracování celého systému.

Je jasné, že i po úspěšném reengineeringu bude zapotřebí najít řešení i k tomuto problému, jinak se riskuje, že za pár let, možná i měsíců, bude projekt trpět stejnými problémy jako v okamžiku analýzy.

3.3 Výběr technologií

Před začátkem návrhu a implementace je nutné zvážit, které technologie ponechat a případně vybrat nové, které budou v projektu používány. Budou popsány stěžejní technologie. Kromě nich budou používány i další, jako je Git, TravisCI, AsciiDoc a mnoho dalších aplikací či knihoven určené programovacím jazykem. Pokud nebudou zmíněny v implementační části této práce, tak o nich bude řeč v dokumentaci aplikace.

3.3.1 Webový server

Umbrela má pro svůj běh k dispozici virtuální server, na kterém je spuštěny webový server Apache. To ale není nezbytně problém. Apache je kvalitní webový server, který podporuje většinu moderních programovacích jazyků pro webové aplikace. Jazyky jako například Java sice potřebují instalaci dodatečných modulů, ale pokud taková potřeba nastane, není problém tyto doplňky přidat. Samotný webový server nebyl zdrojem žádných problémů, a proto není nutné v tomto ohledu dělat jakékoliv změny.

3.3.2 Databázový systém

Databáze je samozřejmě nedílnou součástí aplikace Umbrela. Jak již bylo řečeno, databáze nedávno proběhla důkladnou rekonstrukcí a je v dobrém stavu. Bude sice nutné na pár místech provést menší změny, to se však dá vyřešit jednoduchou migrací. Součástí zmiňované rekonstrukce byla však ztráta všech odpovědí na tehdejší dotazníky. Takovou situaci vedení ústavu označilo za nepřijatelnou a, je proto nutné zachovat veškerá uživatelská data.

PostgreSQL je flexibilní, spolehlivý, open source a hlavně aktivně vyvíjený databázový systém pro relační databáze. Všechny moderní programovací jazyky mají ovladače, které umožňují komunikaci s touto databází a často je podporována i nejrůznějšími frameworky (Maymala, 2015).

Stejně jako v případě webového serveru tuto technologii není třeba měnit. Navíc databázové systémy od firem Oracle nebo Microsoft jsou vhodné spíše pro masivní aplikace, a navíc jejich licence není zdarma, což by nevyhovovalo zadavatelům. Jedinou alternativou by bylo použití databázového systému MySQL (MariaDB). Nicméně rozdíly mezi těmito systémy jsou minimální a při obyčejném používání databáze jsou tyto rozdíly prakticky nulové.

3.3.3 Programovací jazyk

Pro vývoj webových aplikací je teoreticky možné použít libovolný jazyk, který je schopen pracovat s HTTP protokolem a komunikovat s vybraným databázovým systémem. To však neznamená, že některý z jazyků nemůže být vhodnější než jiný. V této části bude popsáno několik programovacích jazyků, které jsou vhodné

pro vývoj webových aplikací a následně budou vybrány ty nevhodnější pro tento specifický projekt.

JavaScript

JavaScript je skriptovací programovací jazyk, který je určený k vytváření interaktivních webových aplikací. Toho je dosaženo tím, že bez nutnosti neustálého načítání webových stránek umožňuje uživateli měnit její obsah. Zapracování JavaScriptu do webové stránky vylepšuje User experience a webové aplikace se tak stávají více intuitivní, a postupně nahrazují klasické desktopové aplikace (Chapman, 2017).

JavaScript se však v dnešní době nepoužívá pouze k vytváření frontendu webových aplikací. V roce 2009 Ryan Dahl na konferenci v Berlíně představil technologii zvanou Node.JS. Narozdíl od všech předcházejících knihoven, frameworků a dalšího softwaru naprogramovaného pomocí zmíněného jazyka, Node.JS nebyl navržen k tomu, aby běžel v prohlížeči. Jednalo se o technologii, která byla spuštěná na straně serveru. To přineslo ideu toho, že celá aplikace by mohla být implementována v jednom programovacím jazyku. Po stránce výkonu je Node.JS srovnatelný s aplikací používající server Apache a jazyk PHP (Rauch, 2012).

I když je vidina jediného programovacího jazyka pro celou aplikaci lákavá, existují i určité překážky, které mohou takovému rozhodnutí bránit. Jako u každé aktivity z jakéhokoliv oboru se v různých situacích používají různé nástroje. Ani u programování tomu není jinak. Neexistuje univerzální řešení a i každý jazyk a technologie má své výhody a nevýhody, které je nutné nejdříve zvážit. Node.JS je vhodný například pro streamování dat nebo pro aplikaci, která bude čelit velikému počtu vstupních dat v jeden okamžik ze strany klienta. Nicméně pro jakoukoliv webovou aplikaci, která je stavěná na relační databázi, není tato technologie příliš vhodná. Důvodem je prozatím nepříliš kvalitní podpora tohoto typu databází. Vzhledem k tomu, že databáze aplikace Umbrela již existuje a její převedení například do grafové nebo objektové formy by bylo časově náročné a i nepraktické, tak je nutné tuto skutečnost opravdu zvážit (Capan, 2017).

Dalším značným problémem Node.JS jsou výpočetně náročné operace na straně serveru. Vzhledem k tomu, že JavaScript může pracovat pouze s jedním vláknem, jakákoliv operace, která je závislá na výpočetním výkonu procesoru, bude značně pomalejší, než při použití jazyka, který je schopen pracovat s více vlákny (Capan, 2017). To se však tohoto projektu příliš netýká.

JavaScript je nedílnou součástí moderních webových aplikací. Už před začátkem analýzy aplikace Umbrela bylo zřejmé, že je součástí této aplikace a že také nadále její součástí bude. Jediné, co nebylo jasné, je do jaké míry se bude tohoto jazyka využívat. Tento jazyk bude s jistotou využit pro některé komplexní části uživatelského rozhraní aplikace. Editor dotazníku je jedna z částí, u kterého je prakticky jasné, že se využije.

Java

Java není nejvhodnějším kandidátem pro webovou aplikaci tohoto typu. Striktní

typová kontrola, vysoké náklady na provoz, složitá konfigurace, ale vysoká bezpečnost jazyka, jistota dlouhodobé podpory, možnost využívání více vláken procesoru a další vlastnosti tohoto jazyka z něj dělají perfektního kandidáta na velké enterprise projekty, na kterých pracují desítky nebo i stovky lidí najednou. Umbrela však taková aplikace není. Nejedná se sice o malý projekt, který má pár desítek aktivních uživatelů, ale také to není obrovská distribuovaná bankovní aplikace s vysokým provozem, pro kterou je jazyk Java vhodnou volbou. Nicméně je nutné se o takovém jazyku zmínit.

V této práci jazyk Java představuje zástupce nejen programovacích jazyků, ale i jiných technologií, které jsou velice mocnými nástroji. Dá se pomoci nich vypořádat bezpochyby s každým problémem, který by mohl při implementaci tohoto projektu nastat. To však neznamená, že je vhodné je použít. Tyto nástroje mají jistě místo ve světě webových aplikací, ale v úplně jiné sféře, než se pohybuje aktuální projekt. Pokud by byl tento jazyk vybrán, tak by to znamenalo určité zvýšení složitosti, což je nežádoucí, a možná i nutnost vynaložení nových finančních prostředků.

PHP, Python, Ruby

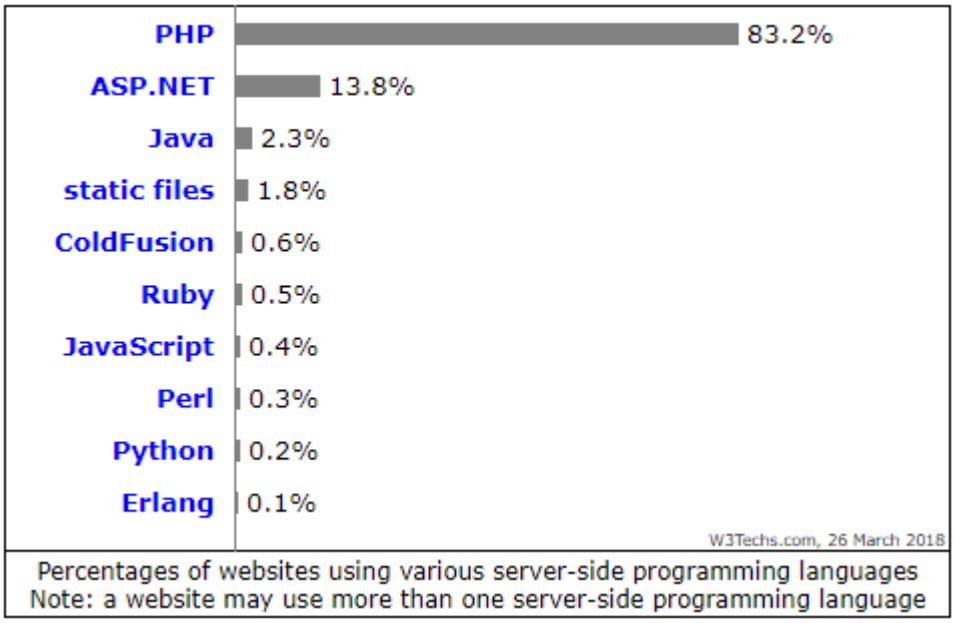
Tyto tři jazyky patří v posledních několika letech mezi nejpopulárnější programovací jazyky pro webové aplikace. Mají také několik společných znaků. Všechny mají slabou typovou kontrolu a jsou používány velkými korporacemi (Facebook, Google, Red Hat) na různé projekty a aplikace. Navíc jsou kompatibilní s již vybranými technologiemi a všechny mají velice velkou a aktivní komunitu vývojářů. Také existuje velké množství frameworků a dalších knihoven, které jsou vedeny jako open source projekty. Jsou používány jak pro menší, tak i pro velké aplikace. Jsou dobře škálovatelné, zdokumentované a je relativně jednoduché vyhledat rady a návody po celém internetu. Uvedené a další vlastnosti dělají z těchto jazyků výborné kandidáty pro novou implementaci aplikace Umbrela. Po porovnání těchto tří jazyků bude vybrán jeden z nich.

Python je moderní programovací jazyk, který se v posledních letech začíná stávat více a více populárním. Programátory je považován za jednoduchý objektový jazyk s čistou syntaxí a je ideální pro začátečníky. Obsahuje mnoho vestavěných datových struktur, typů a funkcí, které dále ulehčují práci. Jedná se o interpretovaný jazyk, což dále urychluje vývoj softwaru (Miller a Ranum, 2007). Vzhledem k tomu, že se jedná o interpretovaný, dynamicky typovaný jazyk, nemusí se podařit při vývoji odchytit všechny možné chyby. Pokud takový případ nastane, můžou se i na produkčním prostředí vyskytovat chyby. Tuto vlastnost sdílí všechny navrhované jazyky a vyžadují mnohem více testování než kompilované jazyky. Výkon je také další vlastnost, ve které interpretované jazyky zaostávají za kompilovanými. To je však důležité spíše u výpočetně náročných programů. Absence ohraničení bloků kódu, a tím pádem velice striktní syntaxe, může vést v některých případech k horší čitelnosti kódu (Krill, 2015). To jsou nejběžnější nedostatky jazyka Python.

Ruby je interpretovaný, dynamicky typovaný programovací jazyk. Je také objektově orientovaný a funkcionální. Dobrá podpora regulárních výrazů, čistá a jed-

noduchá syntaxe. To jsou vlastnosti, kterými je jazyk běžně popisován (Baird, 2007). Tento jazyk se také může pyšnit frameworkem Ruby on Rails, který je širokou veřejností považován za jeden z nejlepších webových frameworků. Horší výkon a menší komunita než u jiných jazyků jsou naopak některé faktory, které reputaci tohoto jazyka poněkud zhoršují. Pak je tu také stáří toho jazyka. Jedná se o relativně nový programovací jazyk a někdy může být velice složité najít programátora se zkušenostmi. Také získání těchto zkušeností může trvat podstatně déle, protože neexistuje tolik kvalitních materiálů jako v případě jiných jazyků (Johnson, 2016).

Posledním z porovnávaných jazyků je PHP. V dnešní době je PHP nejpopulárnějším jazykem pro vývoj webových aplikací. Jedná se opět o objektový, interpretovaný a dynamicky typovaný jazyk. Podle portálu W3tech, až 83.2 % webových aplikací, které tento portál sleduje, používá na straně serveru právě jazyk PHP.



Obrázek 2: Relativní zastoupení programovacích jazyků pro webové aplikace. (W3Techs, 2018)

PHP, stejně jako ostatní jazyky, přichází v několika různých verzích. S vydáním verze 7 se dramatickým způsobem zvedla úroveň bezpečnosti toho jazyka, zmizela spousta chyb, které mohly způsobit zastavení aplikací a také se podstatně zvýšila rychlost programů oproti předchozím verzím. Rychlost aplikací používaných v reálném světe se zvedla o 25 až 70 procent (Prettyman, 2015). PHP se také mírně vzdaluje od dynamického typování. Zavádí návratové typy, typy parametrů funkcí a pomocí anotací lze definovat typy atributů a proměnných (Beak, 2017). Jedná se sice o volitelnou vlastnost, nicméně může do zdrojového kódu vnést vyšší míru organizace a ulehčit tak orientaci a odhalování potenciálních chyb. Stejně jako předchozí kandidáti, oproti kompilovaným jazykům má nižší výkon a pokud nebude použito

možnosti definování typů, mohou se vyskytovat chyby i v průběhu aplikace. Co však nejvíce znehodnocuje tento jazyk, je jeho špatná reputace. Mezi velkým množstvím vývojářům panuje názor, že tento programovací jazyk je jednoduše špatný. Tento názor je z velké části způsoben stářím toho jazyka, respektive množstvím takzvaných "legacy" aplikací. Jedná se o velmi staré aplikace, které nepoužívají žádný framework a jsou postaveny na velmi starých verzích jazyka PHP (Dupertuis, 2016). Kombinací těchto faktorů pak vznikal velice špatný kód a udržování takových aplikací způsobilo nejednu bolest hlavy. Nicméně špatný kód se dá napsat v každém jazyce a u nových aplikací v jazyce PHP tyto problémy nejsou aktuální. Proto se musí brát podobné názory s určitou rezervou.

Z předchozího textu vyplývá, že všechny jazyky jsou si velmi podobné. Sdílejí spoustu vlastností, jak dobrých, tak i špatných. K uskutečnění konečného rozhodnutí bude tedy nutné vzít v potaz i jiné parametry, a to organizační. Jak již bylo několikrát zmíněno, aplikace Umbrela vždy byla a s velkou pravděpodobností bude udržována studenty informaticky zaměřených oborů Mendelovy univerzity v Brně. Tento fakt sebou přináší několik informací o sadě dovedností, které můžeme od takových jedinců očekávat. Podíváme-li se na studijní program takového studenta, zjistíme, že ze zmíněných jazyků je pouze jazyk PHP v průběhu studia ve větší míře vyučován. Student se s tímto jazykem může setkat hned v několika předmětech s přímou navázaností na webové aplikace. I když se navrhované jazyky liší spíše ze syntaktické části, je rozhodně praktičtější, aby potenciální vývojář měl předchozí zkušenost s programovacím jazykem a urychlila se tak jeho aklimatizace do projektu. Z faktu, že je jazyk PHP na univerzitě vyučován, je také zřejmé, že v řadách lektorů jsou osoby s bohatou zkušeností s programování v PHP a mohou poskytnout spoustu užitečných rad, vyskytne-li se vývojář ve složité situaci.

Z těchto důvodů bude jako programovací jazyk pro serverovou stránku aplikace použit jazyk PHP verze 7.1 a vyšší, který by měl nahradit dosavadní zastaralý Perl. Jeho vlastnosti, relativně nedávné vydání poslední verze, zkušenosti akademických pracovníku a přísné dodržování konvencí by měli pomoci k udržení lepší konzistence projektu.

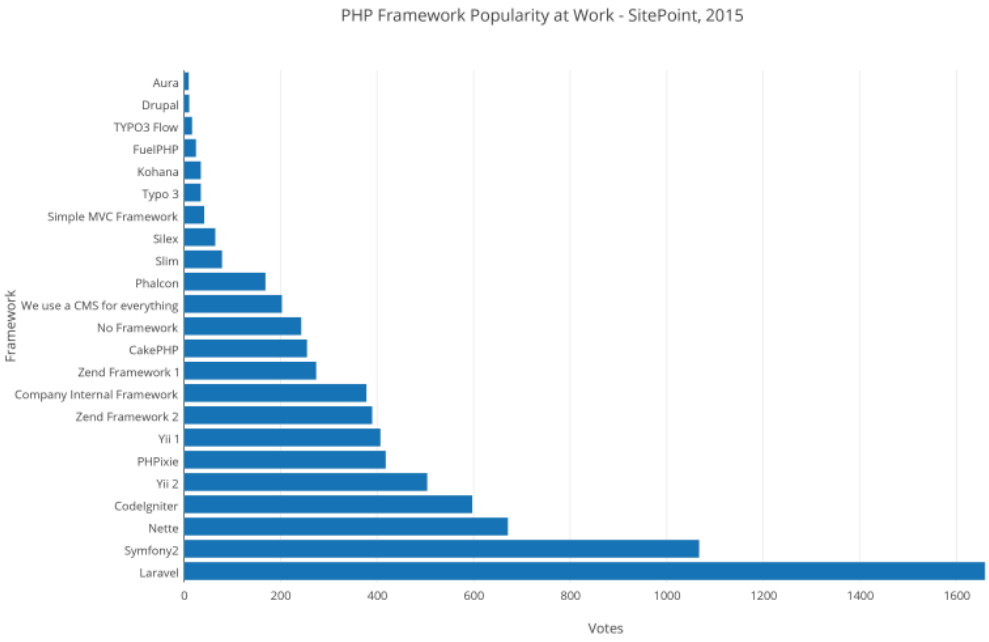
3.3.4 Framework

Webový framework teoreticky není absolutně nutná součást webových aplikací. Nicméně z praktického hlediska je obecně doporučeno nějaký použít. Dobrý framework ušetří spoustu práce a dovoluje vývojářů se soustředit přímo na funkce aplikace, místo toho aby museli nejdříve řešit spoustu rutinních záležitostí.

Výběr frameworku bude samozřejmě omezen již vybranými technologiemi. Bude se jednat o framework pro jazyk PHP kompatibilní s verzí 7.1. Výběr se bude také odvíjet od bezpečnosti, rychlosti, aktivity komunity, debugovacích nástrojů, testovacích nástrojů a samozřejmě dokumentace. Aplikace Umbrela je také celkem specifická. Neexistuje příliš mnoho aplikací poskytující stejné služby. Proto místo komplexních frameworku s velkým množstvím pluginů, které při tvorbě generické

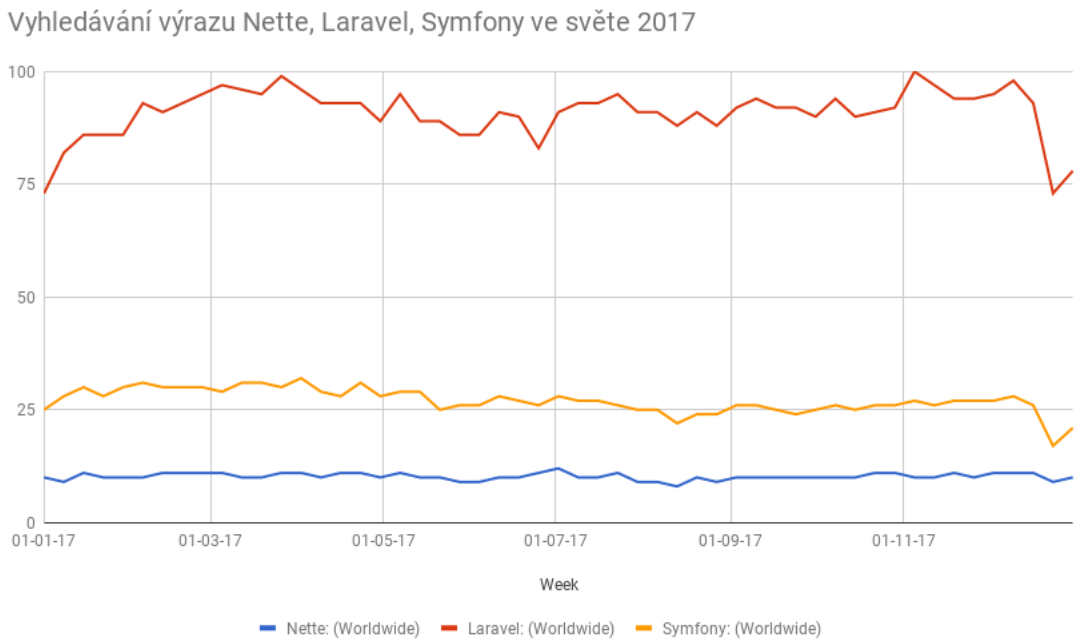
aplikace mohl ušetřit spoustu času, bude preferovaný spíše menší a jednodušší za účelem rychlejšího běhu aplikace.

Jelikož takových rámcových řešení pro jazyk PHP existují snad stovky, bude vybíráno pouze z těch nejpopulárnějších. Na následujícím obrázku jsou vidět nejpopulárnější frameworky podle studie SitePoint. I mnoho dalších zdrojů přichází s velice podobnými výsledky.



Obrázek 3: Popularita webových frameworků pro jazyk PHP. (GeckoandFly, 2018)

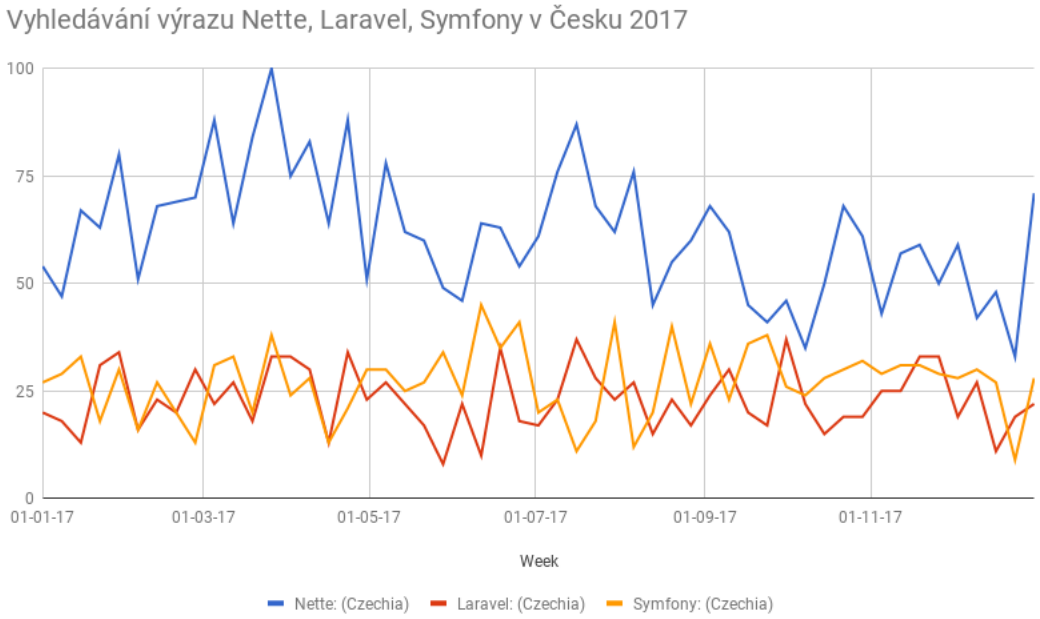
Vybíráno bude z prvních tří frameworků. Těmi jsou Laravel, Symfony2 a Nette. Po krátkém prozkoumání bylo zjištěno následující. Všichni tři zástupci mají aktivní komunitu. Poslední aktualizace projektů na portálu GitHub byly v rámci pár, hodin maximálně jednoho dne. To je dobrá známka aktivity. Není žádoucí, aby byla vybrána jakákoliv alternativa, u které hrozí zastavení vývoje. Všechny frameworky jsou založené na objektovém přístupu. Překvapivě Laravel nepodporuje dependency injection. Místo toho používá statické třídy, což může některým vývojářům vadit. Všechny mají dobrou dokumentaci, šablonovací systém, podporují již vybrané technologie, jsou dobře zabezpečené a i výkonově se drží na předních pozicích. Celkem vzato, každá z těchto alternativ splňuje všechny požadavky. Kdyby je nesplňovaly, pravděpodobně by se neumístily na předních příčkách mezi programátory. Stejně jako u programovacího jazyka tedy spíše záleží na osobní preferenci a bude nutné přihlédnout i k dalším skutečnostem které mohou nastat během a po skončení reengineeringu. Pokud by se jednalo o projekt, který by měl zajištěný dlouhodobý tým, nejspíše by zvítězil framework Laravel, čistě díky jeho světové popularitě a minimálnímu riziku ukončení podpory v blízké budoucnosti.



Obrázek 4: Vyhledávání frameworků ve světě. (Google, 2018)

Vzhledem k tomu že taková situace je nepravděpodobná a i samotný reengineering je prováděn jedinou osobou, budeme se muset zamyslet nad prostředím, ve kterém bude aplikace žít a opět nad profilem potenciálního budoucího vývojáře. Bude se opět jednat o studenta, který své znalosti bude primárně čerpat ze studia. Opět by tedy bylo praktické, aby se se zvoleným frameworkem mohl setkat v průběhu výuky. V tento moment přichází na řadu český framework Nette. Vztah mezi frameworkem Nette a jeho protějškem Laravel může být přirovnán k pověstnému boji mezi Davidem a Goliášem. I přesto, že v porovnání s ostatními alternativami se jedná o malý framework, disponuje všemi moderními vlastnostmi a v některých případech je dokonce napřed před protějšky. Framework Nette byl v nedávné minulosti vyučován přímo na Provozně ekonomické fakultě. Jeho šablonovací systém Latte by se měl i nadále používat k výuce. Další kladnou vlastností je dokumentace v českém jazyce. I když je angličtina dnes považována za samozřejmost, u některých jedinců dále může odstranění této bariéry dále pomoci k rychlejšímu přizpůsobení. Těm, co by mohl český jazyk vadit, je také dostupná alternativa v angličtině. Kromě dokumentace existuje i české fórum, na kterém se aktivně vedou nejrůznější diskuze. Ty mohou dále pomoci při řešení možných problémů. Frameworku Nette také hraje do karet jeho velikost. Jak již bylo řečeno, aplikace je velmi specifická a spousta vestavěných funkcí větších kandidátů by nikdy nebyla využita. Jediným problémem by tedy po volbě nejmenšího ze tří uchazečů mohla být relativně nízká popularita a risk nenadálého ukončení podpory. Pravděpodobnost takové situace je však poměrně nízká. V Česku a sousedním Německu a Rakousku, byl v roce 2017

stále nejvyhledávanějším webovým frameworkem a jak již bylo řečeno, projekt je skoro každý den upravován a rozšiřován.



Obrázek 5: Vyhledávání frameworků v Česku. (Google, 2018)

Pro vývoj aplikace bude tedy použit framework Nette. V případě čistě subjektivního rozhodnutí by pravděpodobně zvítězil framework Laravel. Z důvodu snahy zajistit jednodušší zapojení do projektu budoucích programátorů byla zvolena jiná varianta. To neznamená, že by byla nějakým způsobem horší. Všechny možnosti splňují stanovené požadavky a pro potřeby tohoto specifického projektu jsou všechny více než dostačující. Zvolená varianta má díky svému původu pár pěkných vlastností, které mohou ulehčit práci na projektu.

3.3.5 ORM

ORM neboli object-relational mapping je pojem pro proces převodu strukturovaných dat z relační databáze do objektů. Od té doby, co je objektově orientované programování de facto standardním způsobem psaní softwaru, se musel řešit problém nesourodosti interpretace dat mezi relační databází a zdrojovým kódem. Pokud má být plně využito výhod objektového přístupu, je nutné data převést do instancí tříd nebo použít jiný způsob perzistence dat. První řešení byla vytvářena na míru k určitému softwaru. Vzhledem k rozšířenosti a popularitě jak relačních databází, tak objektového přístupu, se postupně začaly objevovat knihovny pro širokou škálu různých programovacích jazyků. Tyto knihovny řešily problém převodu záznamů z tabulek na objekty (Barcia aj., 2008).

Teď se však naskýtá dobrá otázka. A tou je výkon. Každý převod tabulkový dat na objekty sebou nese nezbytné režijní operace. Také jakákoliv hlubší optimalizace dotazů je závislá přímo na používaném systému. Obecně je známo, že použitím ORM se sníží rychlost komunikace s databází. Kdy je tedy vhodné ORM použít?

Na toto téma existuje spousta debat, které jsou často spíše podloženy subjektivními názory a preferencemi jednotlivců než fakty. Většina vývojářů z obou stran debaty se však shoduje na následujícím. ORM je zaměřeno na doménový model aplikace. Podstatně snižuje množství kódu. Knihovna bývá dobře otestovaná vlastními vývojáři a tím snižuje bezpečnostní rizika a možné výskyty chybného kódu. Některá ORM využívají cache, čímž mohou v některých případech dokonce snížit zatížení databáze. Z tábora odpůrců se často ozývají tyto nevýhody. Zvýšení doby spouštění aplikace. Obtížná optimalizace a debugging dotazů a vysoká učící se křivka (Sasidharan, 2016). Pokud se použitím ORM dramaticky nesníží výkon aplikace a použije se kvalitní mapování, které zaručí bezchybné získání dat, objektově relační mapování je rozhodně užitečné.

Před tím než padne rozhodnutí o jeho využívání, je třeba se také podívat na charakter dat. Pokud je model databáze opravdu jednoduchý a očekáváme obrovskou zátěž, ORM je zbytečné. Samotná konfigurace pravděpodobně zabere delší čas než vytvoření jednoduchého modelu. Navíc převod velkého množství dat na objekty přinese další mezikrok a tím zvýší režii, což přinese vyšší prodlevu mezi požadavkem a získáním dat. Na druhou stranu čím je model složitější a zátěž nižší, tím roste potřeba nějakého mapování. Orientace ve složitém datovém schématu je obtížná a použití mapování může podstatně snížit nároky na vývojáře. Se zvyšující se složitostí modelu roste náročnost konfigurace takového mapování a v extrémních případech je potřeba opravdových expertů (Hadlow, 2012).

Aplikace Umbrela má složitější schéma, nicméně počet požadavků na databázi není nijak zvlášť vysoký a neměl by tedy být ani žádný zásadní problém s výkonem při využití mapování. Před rozhodnutím se znovu musí zvážit charakter celého projektu. Jedním z nejsilnějších argumentů, proč použít ORM, je čas. Na vytvoření kvalitního modelu bez použití knihoven třetích stran jednoduše není dostatek času. Jedním z důvodů celé této práce je zavedení nových standardů a pravidel pro další vývoj. Pomocí nich by měl dojít ke zjednodušení a vyšší organizace zdrojového kódu. Vytvoření opravu kvalitního modelu, který by přinesl stejné výhody jako dobře řešené mapování a navíc optimalizovat SQL dotazy specificky pro tuto aplikaci, by nejspíš zabralo stejně úsilí jako vytvoření celé aplikace. Vzhledem k časovému omezení je to však nereálné. Není ani zaručeno, že by výsledek nebyl mnohem horší než nějaké dostupné řešení. Dalším argumentem jsou schopnosti vývojářů. Nedá se spoléhat, že každý bude ovládat SQL na dostatečně vysoké úrovni. Je proto mnohem bezpečnější a jednodušší nechat programátory pracovat s metodami objektů, než riskovat psaní dotazů a dodatečně měnit schema databáze, což by opět mohlo degradovat celý projekt. Proto bude v této aplikaci použito objektově relační mapování.

Dále je nutné se rozhodnout jakou knihovnu použít. Už je jasné jaký programo-

vací jazyk je nutný. Také dobrá kompatibilita s vybraným frameworkem je důležitá vlastnost. Jako vždy i v tomto případě existuje mnoho alternativ. Nicméně ne všechny jsou dobré. Narozdíl od jiných technologií, u ORM je výběr poměrně jednoduchý. Nejlepším řešení je Doctrine ORM. Má největší komunitu, asi nejlepší implementaci, ale hlavně existuje i plugin, který přímo integruje tuto knihovnu do Nette. Doctrine jako taková je kolekce několika knihoven, které dohromady ulehčují proces persistence dat a vytváření komplexních databázových modelů. Podporuje také databázové systémy PostgreSQL (Dunglas, 2013). Neměl by tedy existovat žádný problém s integrací do aplikace.

3.3.6 React

Už dříve bylo zmíněno, že JavaScript bude nedílnou součásti aplikace. Kromě klasických knihoven pro zjednodušení práce s formuláři nebo pro asynchronní komunikaci, je však také nutné se zamyslet nad modernějším použitím tohoto jazyka. Z prvotní analýzy aplikace vyšlo najevo, že pro vytvoření editoru dotazníku, bylo použito knihovny Reac.JS, tehdy ještě beta verze. Editor dotazníku je nejkomplexnější částí celé aplikace, a proto je nutné použít sofistikovanějšího řešení pro jeho implementaci. Není to však jediný kandidát, kde by se dalo knihoven, jako je React, využít.

Existuje mnoho různých knihoven a v posledních letech všechny konvergují ke stejným paradigmatům. Používají podobné syntaxe, virtuální DOM, one-way data binding a staví na komponentové architektuře. Jediný podstatnější rozdíl je v jejich podpoře a komunitě.

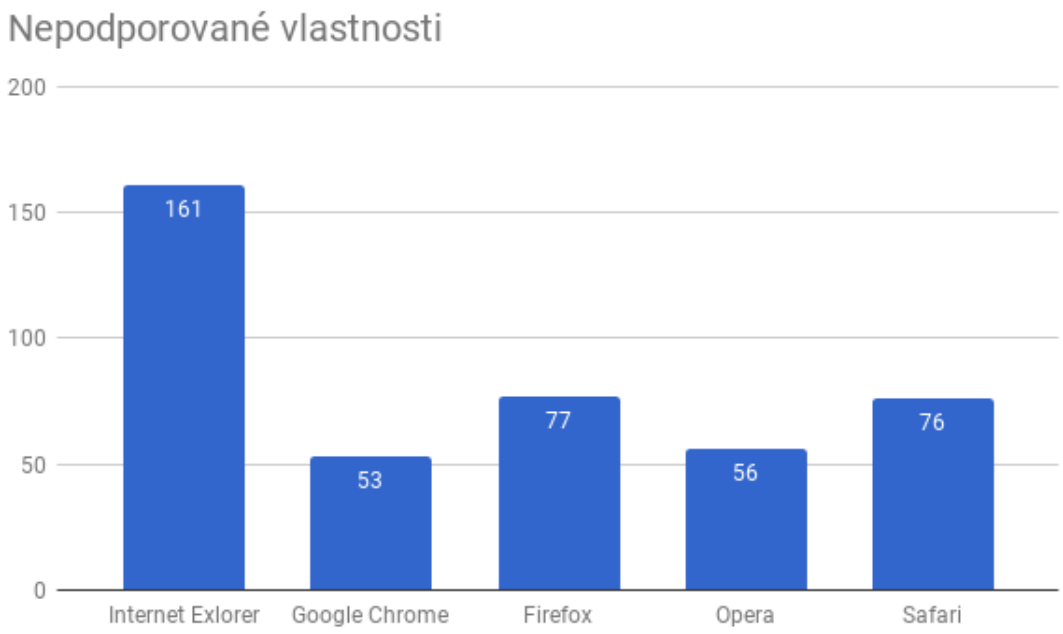
Tabulka 2: Statistiky JS nástrojů pro tvorbu uživatelského rozhraní (Korotya, 2017)

	Angular2	**React**	**Vue.js**	**Ember.js**	**Meteor.js**
Definice	MVC framework	Knihovna	MVC framework	MVC framework	Platoforma pro JS aplikace
Rok vydání	2016	2013	2014	2011	2012
Komunita	392	912	62	636	328
Popularita GitHub	19832	57878	39933	17420	36496

Pokud pomineme starší framework AngularJS, který je nahrazen verzí 2 (a více), ze schématu lze vyčíst jasného vítěze, kterým je React. Nejen velikost komunity a počet spoluúčastníků projektu, ale i fakt že je aktivně vyvíjen a používán společností Facebook, dává dostatečnou jistotu, že tento projekt nebude za pár měsíců opuštěn. Bude se tedy dále používat i v nové realizaci aplikace Umbrela, avšak v aktuální verzi.

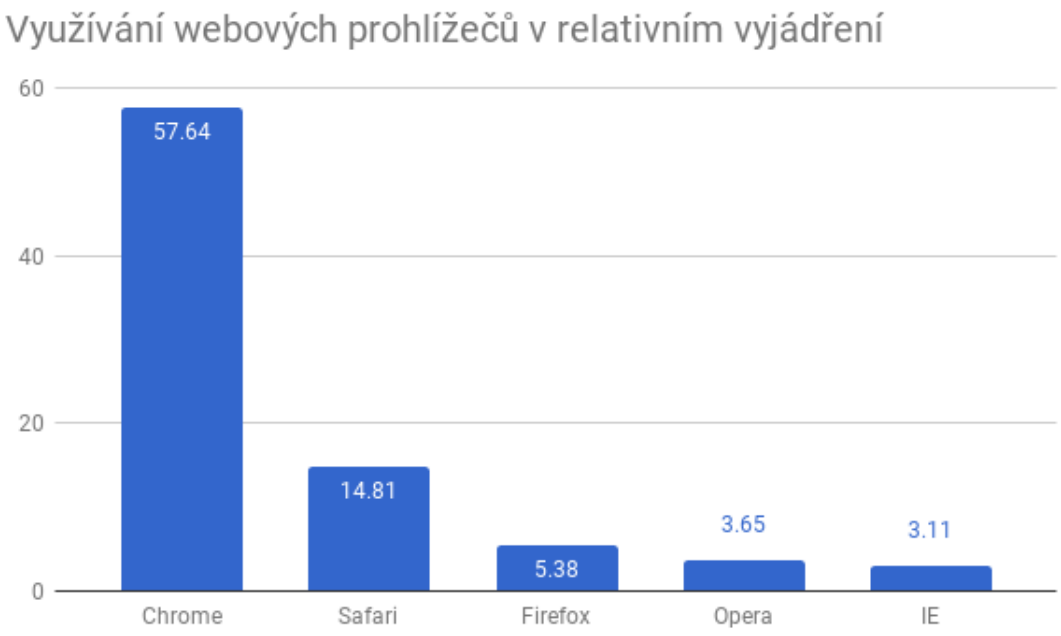
3.3.7 Webový prohlížeč

Jaký webový prohlížeč podporovat? To je otázka, na kterou se nabízí velice jednoduchá odpověď, všechny. Ve skutečnosti to však nemusí být tak jednoduché. Implementace prohlížečů se mohou podstatně lišit a ne každá funkce JavaScriptu či vlastnost kaskádových stylů funguje ve všech prohlížečích stejně. V některých případech dokonce není vůbec podporovaná. Ideální stav by samozřejmě byla podpora všech prohlížečů. To by však znamenalo podstatné omezení možností co se týče JavaScriptu a CSS, a to hlavně díky extrémně zastaralému prohlížeči Internet Explorer. Oproti ostatním prohlížečům, jeho poslední verze 11, nepodporuje obrovské množství vlastností, což by mohlo podstatně komplikovat vývoj front-end části aplikace.



Obrázek 6: Počet nepodporovaných vlastností webovými prohlížeči. (Can i use, 2018)

Po zběžném průzkumu bylo nalezeno přes 160 nepodporovaných vlastností nejen JavaScriptu a CSS, ale i některé HTML5 značky. Dalších vlastnosti, které mají chybnou implementaci, je také podstatné množství. Podpora prohlížeče Internet Explorer 11 by do jisté míry zkomplikovala vývoj a minimálně na straně uživatelského rozhraní by projekt posunula zpět do doby kamenné. Naštěstí ze statistiky používání webových prohlížečů plyne, že jej používá naprosté minimum uživatelů.



Obrázek 7: Míra využívání webových prolížečů. (StatCounter, 2018b)

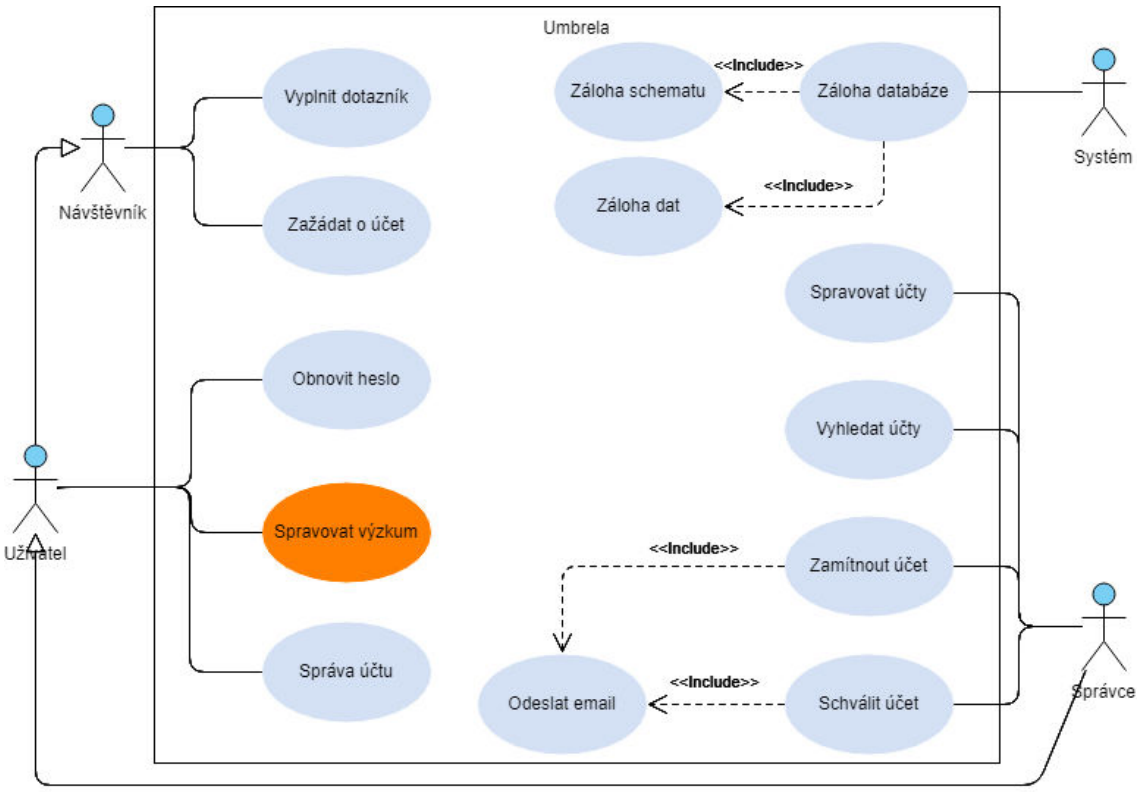
Tato data jsou v celosvětovém měřítku a ze všech typů zařízení, které mají přístup k internetu. Vzhledem k uživatelům aplikace mohou být nepřesná. Naštěstí, díky dlouhodobé existenci aplikace je, k dispozici celkem dobrý profil klasického uživatele aplikace Umbrela. Většina zaměstnanců univerzity používá prohlížeče Firefox a Google Chrome. To stejné platí pro studenty. Protože nejsou sbírána data o respondentech, není zřejmé jaké prohlížeče používají a u tohoto druhu uživatele je nutné se držet veřejně dostupných dat. Budou tedy podporované prohlížeče Google Chrome, Firefox a Safari.

4 Implementace

V této části práce bude popsán průběh implementace nové verze aplikace Umbrela. Kapitola je rozdělena na implementaci backendu a frontendu aplikace, jelikož se od sebe podstatně liší. Moduly v knihovně React mohou být navíc považovány za samostatný projekt. Implementace se drží postupů agilního vývoje softwaru a není rozčleněná na klasické fáze analýzy, návrhu, implementace, testování a nasazení. Rozdělení kapitoly na tyto části slouží čistě k logickému uspořádání a ulehčení orientace. Implementace probíhala na operačním systému Linux distribuce Ubuntu. Kód je spravovaný pomocí softwaru Git.

4.1 Funkce aplikace

Před začátkem psaní zdrojového kódu je nejdříve nutné si alespoň hrubě ujasnit jaké funkce by měl systém mít. Agilní metodiky vývoje softwaru počítají s neustálou změnou požadavků na systém, a proto nemá smysl snažit se o vytvoření detailní specifikace problému. Se zadavatelem byly prodiskutovány hlavní funkce, kterými by měla aplikace disponovat. Na následujícím diagramu je vidět zjednodušený model případů užití, které byly identifikovány jako stěžejní případy užití. Všechny detaily správy dotazníku jsou záměrně vynechány z důvodů přehlednosti a jsou znázorněny v samostatném schématu.



Obrázek 8: Zjednodušený model případů užití

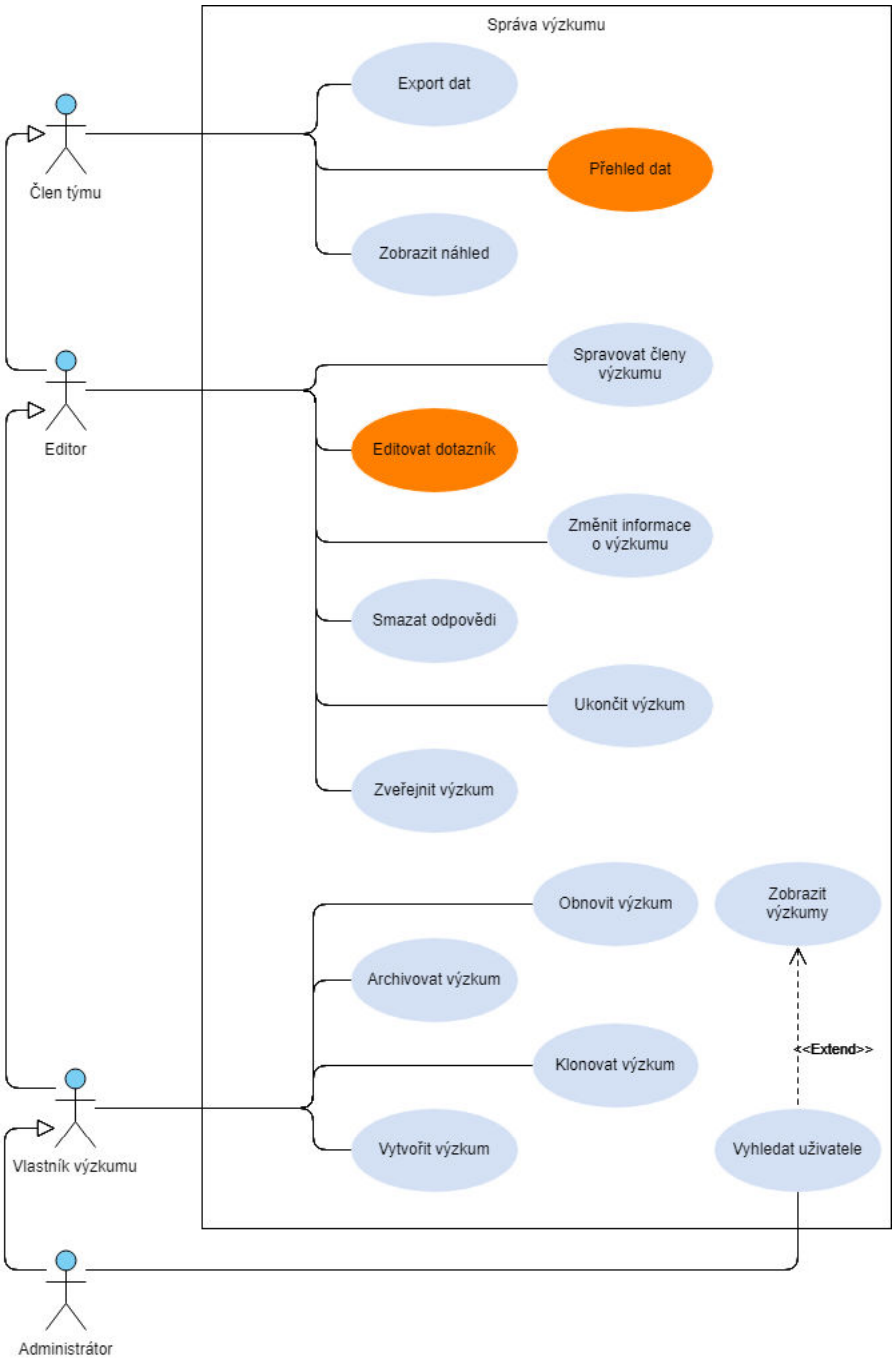
I na takto abstraktní úrovni už existuje jeden rozdíl oproti starší verzi systému. Tím je odstranění vytváření role ”Garant” ze systému. Uživatel typu garant v minulosti sloužil jako dozor nad výzkumem. Mohl sledovat změny v projektu, provádět úpravy v dotazníku a zobrazit si data. Všechna tato práva, kromě úpravy dotazníku, měl však také běžný člen týmu, kterého může vlastník výzkumu přidat. Po krátké diskuzi byla tato role označena za zastaralou a bude kompletně nahrazena přidělením extra práv určitému účastníkovi na výzkumu.

Poměrně zvláštní částí aplikace je vytváření uživatelských účtů. Ústav marketingu a obchodu chce mít kontrolu nad tím, kdo aplikaci používá. Primárně je určena pro zaměstnance a studenty univerzity. Nicméně existují i výjimky, podle kterých si může založit účet i jiná osoba. Tyto podmínky určuje samotný ústav. Pokud by neexistovaly, nejjednodušší způsob autorizace by bylo napojení na LDAP univerzity. Tuto variantu však zadavatel zamítl a vyslovil přání zachovat stávající způsob vytváření uživatelských účtů. Jednotlivec si musí nejdříve zažádat o uživatelský účet. Vyplní registrační formulář a jeho odesláním se v databázi vytvoří žádost o účet. Každý účet musí být schválen správcem (uživatel s nejvyšším oprávněním). Pokud uživatel nesplňuje stanovené podmínky, bude jeho žádost zamítnuta a přístup k aplikaci bude odepřen. Tento způsob není ideální a přidává další administrativní zátěž, nicméně i přes vyslovené námitky, si zadavatel přál ponechat tuto funkcionalitu.

Další případy užití ani role už nemají žádné nestandardní způsoby realizace. Návštěvník (nejčastěji respondent), kromě registrace a vyplnění dotazníku, není autorizován k jakýmkoliv aktivitám. Správce může naopak provádět jakékoliv změny, od schvalování účtu, přes editaci cizích dotazníků až po anonymizaci jednotlivých uživatelů.

4.2 Správa výzkumu

Nyní bude blíže popsána správa jednotlivých výzkumů. Jedná se o jádro celé aplikace a je třeba ho správně pochopit. V průběhu vývoje se editor dotazníku několikrát upravoval a od první iterace implementace prošel podstatnou změnou. Na následujícím diagramu lze vidět případy užití vztahující se ke správě dotazníků.



Obrázek 9: Případy užití správy výzkumu

Výzkum může být spravován až čtyřmi různými typy uživatelů. Vlastník a administrátor jsou jednoznační uživatelé, kteří mohou zasahovat do výzkumu. Dalšími jsou pak člen týmu a jeho speciální druh, editor. Ten nahradil zmiňovaného garanta. To, jestli bude člen výzkumu editorem nebo ne, určuje vlastník výzkumu, případně jiný editor. Oproti předchozí realizaci aplikace zde nalezneme několik dalších rozdílů.

Uživatelům zmizela možnost mazání dotazníku a nahradila ji jejich archivace. To by mělo zabránit nechtěnému smazání výzkumu a ztrátě dat. Výzkum bude samozřejmě možné obnovit.

Export dat také projde menší změnou. Do databáze nyní přibude informace o délce vyplňování dotazníku respondenty, ta bude dostupná právě v exportovaných datech. Dále se také přestane soubor s exportovanými daty ukládat na disk serveru. V minulosti se soubory ukládaly s myšlenkou udržování historického vývoje, nicméně se ukázalo, že tuto možnost uživatelé vůbec nevyužívali. Data si stejně vždy ukládali na svá zařízení. Zadavatel tedy souhlasil, že se exportovaná data nebudou nadále ukládat na server.

Již několikrát zmiňovaný editor dotazníku a také modul pro prohlížení odpovědí projdou velikou změnou. Editor obsahoval značné množství chyb a musí být od základů předělán. Modul pro zobrazování dat fungoval velice dobře, nicméně u většího množství respondentů měl problém s rychlostí, a proto bude nutné upravit způsob nahrávání dat. Oba moduly jsou také ve starších verzích knihovny React a budou aktualizovány. Co se týče funkcí, většina se bude lišit spíše implementací. Za zmínku stojí možnost filtrování otázek. V nové verzi přibude možnost filtrování mezi jednotlivými archy dotazníku, která v minulosti chyběla. Bude také odstraněna možnost vytváření dvou druhů otázek, které uživatelé nepoužívali. Maticovou otázku s textovou odpovědí už nebude možné vytvářet, ale protože v databázi jich již pár existuje a zadavatel nechce uživatele připravit o data, bude její typ zachován. Otevřená otázka s dlouhým textem v zadání bude nyní sloučena s klasickou otevřenou otázkou a pouze se aktualizuje omezení délky textu.

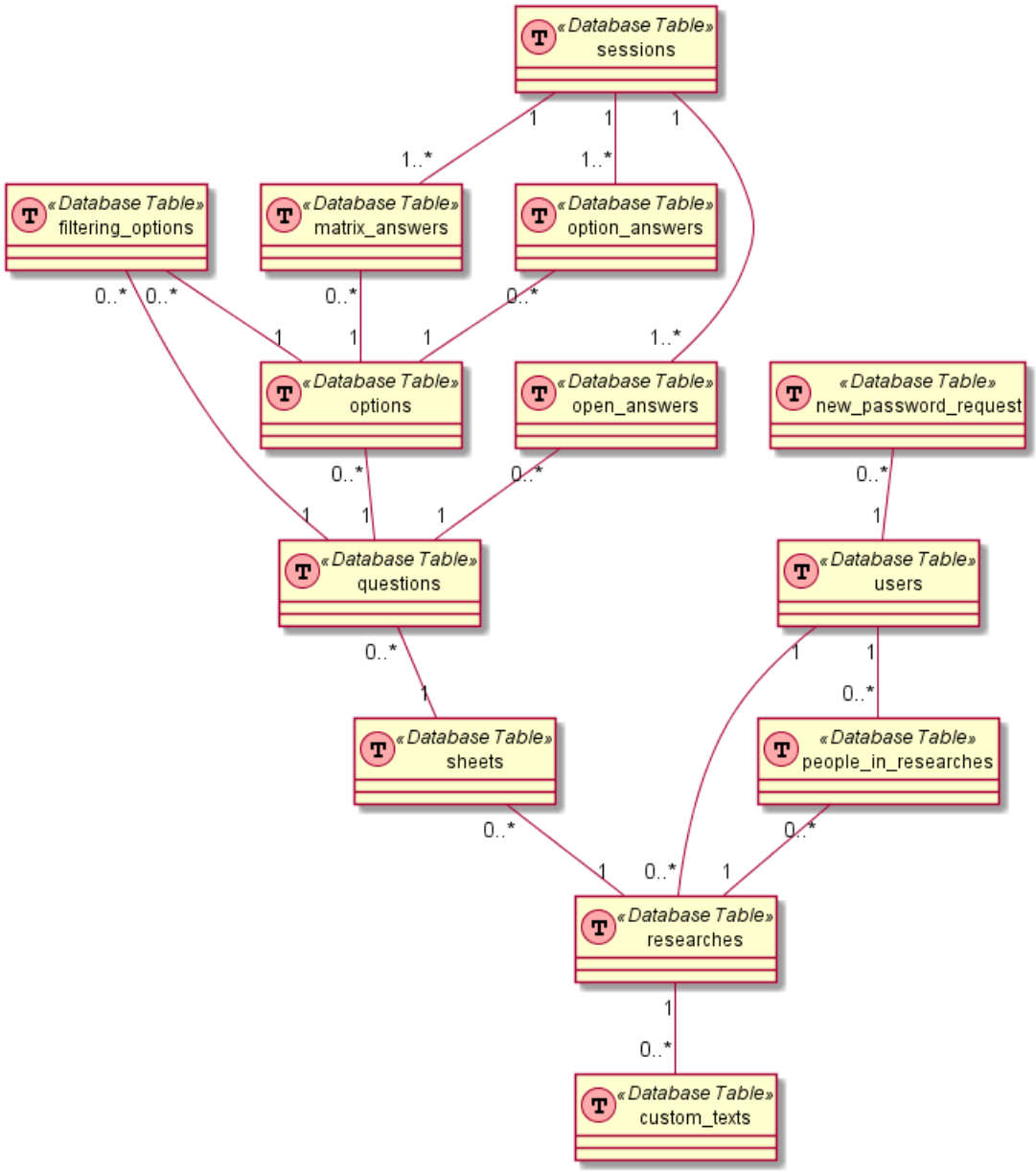
4.3 Schéma databáze

Schéma databáze je na rozdíl od aplikace v dobrém stavu. V minulosti prošlo transformací, která značně vylepšila její stav. I zde je však možné udělat menší úpravy. V tabulce pro žádosti o obnovu hesla chybí primární klí. Dvě tabulky "cached_sheets" a "temp_sessions" budou díky ORM a ukládání stavu vyplňování dotazníku v klientském prohlížeči zbytečné. Mohou se tedy bez problému odstranit.

Kvůli zvýšení rychlosti získávání odpovědí respondentů bude také do tabulky "sessions" přidán cizí klíč do tabulky výzkumů, aby bylo ihned možné přistoupit k těmto datům. Odpovědi jsou sice přístupné díky cizím klíčům, nicméně i pro získání tak triviální informaci, jako je počet respondentů výzkumu, musel být vždy proveden složitý dotaz, který zbytečně prodlužoval odezvu. Přidáním jednoho, zdánlivě redundantního, sloupce se značně sníží složitost všech operací, které v nějaké podobě potřebují data o odpovědích na dotazník. Jednoduchým dotazem lze tento sloupec přidat i existujícím záznamům.

Další, spíše aplikační změnou, bude ukládání otevřených odpovědí respondentů. V tabulce "open_answers" se bude používat sloupec "long_answer", který umožňuje ukládaní řetězce o maximální délce 1500 znaků. V minulosti se používalo sloupce "short_answer" s maximální délkou 255 znaků. To se však postupem času ukázalo

jako nedostatečné. S touto změnou také souvisí sloučení dvou typů otázek ”open” a ”openLong”.



Obrázek 10: Zjednodušený erd diagram

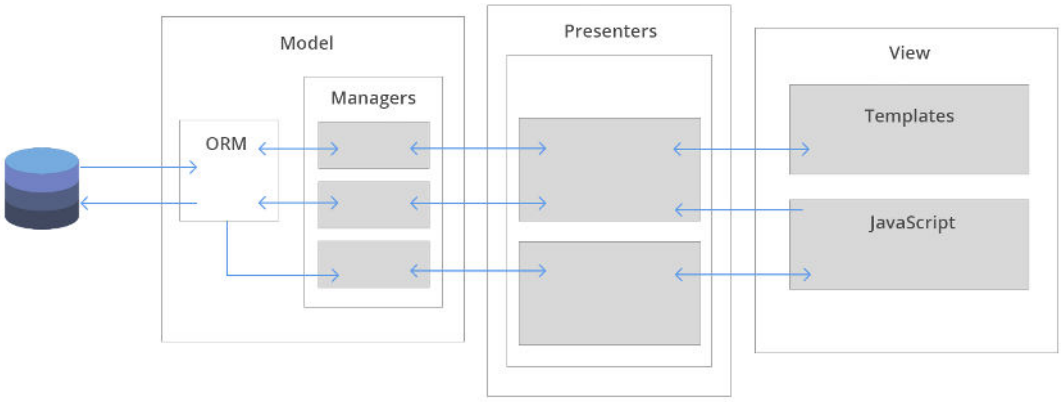
4.4 Backend

Po ujasnění základních funkcí a doladění schématu databáze, je tedy čas připravit vývojové prostředí pro serverovou část aplikace. Vzhledem k tomu, že určité části uživatelského rozhraní budou realizovány pomocí knihovny React, mohla by práce

na těchto dvou částech aplikace probíhat paralelně. Jediné, co by bylo třeba specifikovat, je API pro komunikaci se serverem. V praxi je takovýto postup zcela běžný za pomocí nástrojů k simulování API jako je například Apiary. Protože není k dispozici vícečlenný tým, bude nejdříve dána přednost práci na serverové části aplikace, aby se mohla následně použít k vývoji a doladění klientské části.

4.4.1 Architektura

Pro implementaci byl vybrán programovací jazyk PHP 7.1 a vyšší, webový framework Nette a Doctrine2 ORM pro model. Základní rozdělení zdrojového kódu se bude řídit MVP architekturou. Na následujícím schématu je ilustrováno, jakým způsobem budou mezi sebou jednotlivé části komunikovat.



Obrázek 11: Architektura aplikace

Striktní dodržování navrhovaného způsobu komunikace je důležitou podmínkou k udržení konzistentního zdrojového kódu. Přístup k databázi z presenterů bude možné pouze přes takzvané "Managers". To budou třídy vztahující se k jednotlivým entitám databáze, které s nimi budou manipulovat a připravovat je pro další použití. Díky použití ORM může být někdy hranice mezi modelem a presenterem nejasná. Používání metod nad různými třídami je ekvivalentní s vyvoláním nějakého SQL. Například použití metody *"find(...)"* nad manažerem všech entit je ekvivalentní s SQL dotazem *"SELECT * from table"*. Tento kus kódu by se nikdy neměl objevit v presenteru. K takový metodám tedy bude přistupováno stejným způsobem. Existují i krajní případy, kdy se může zdát přesun na modelovou vrstvu aplikace zbytečný, nicméně by se měl tento návrh dodržovat a používat v presenterech metody které vyvolávají akce nad databází jen minimálně.

4.4.2 Správa závislostí

Součástí aplikace bude samozřejmě spousta knihoven a dalších závislostí. I samotný framework je ve své podstatě jen knihovna, která je v průběhu vývoje využívána. Vzhledem k tomu, jakým směrem se dnes vývoj nejen webových aplikací udává, těchto závislostí mohou být desítky, nebo až stovky. Další skutečností je to, že i samotné knihovny mohou mít další závislosti v několika různých verzích. Spravovat tyto knihovny ručně je nepraktické v některých případech i nemožné. Tyto aktivity jsou naštěstí dnes již automatizované díky takzvaným "Dependency Managers". Správce závislostí je software, který spravuje externí knihovny a balíčky a integruje je tak, aby spolu mohly spolupracovat. Informace o těchto závislostech, jako je jejich verze nebo místo, ze kterého jsou dostupné, jsou obvykle uložené na online repositáři. Správce tyto informace využívá k identifikaci potřebných závislostí a s jejich stažením na zařízení vývojáře (Seun, 2017). To má hned několik výhod jako například jednoduchá organizace závislostí, automatická aktualizace na nejnovější kompatibilní verzi. Také přenášení kódu je mnohem jednodušší. Místo těchto závislostí je potřeba, kromě zdrojového kódu aplikace, přiložit pouze seznam knihoven ve standardizovaném formátu.

Jazyk PHP má svého vlastního správce, který nese jméno composer. Pomáhá definovat potřebné knihovny, jejich instalaci a aktualizaci. Standardně jsou závislosti ukládány pouze lokálně, takže jsou přístupné pouze z konkrétního projektu. Pokud je potřeba, tak mohou být uloženy i globálně, to by se však nemělo nikdy vztahovat na konkrétní závislost projektu (Composer, 2016).

4.4.3 Počáteční konfigurace aplikace

Před začátkem vývoje je také nutné nastavit vývojové prostředí. Instalace jazyka, databázového systému a dalšího nutného softwaru je blíže popsána v dokumentaci přiložené v práci. Jakmile je vývojové prostředí nachystáno, je už možné začít s prvotním nastavením aplikace a začít s vývojem. Aplikace byla sice vyvinuta na operačním systému Linux, ale celý proces by měl být relativně jednoduše proveditelný i na jiných.

Prvním krokem pro vytvoření základu celého projektu je instalace takzvaného "sandboxu", který obsahuje prázdnou aplikaci postavenou na frameworku Nette. Za použití správce závislostí se jedná o jednoduchý příkaz v terminálu operačního systému:

```
1 $ composer create-project nette/sandbox umbrela-server
```

Tento příkaz vytvoří a konfiguruje prázdnou aplikaci, do které je následně možné provádět všechny změny. To, jestli aplikace funguje, lze ověřit spuštěním vestavěného serveru jazyka PHP jednoduchým příkazem. Po přepnutí do nového adresáře aplikace stačí zadat shell příkaz a aplikace bude dostupná na zvolené adrese (Votoček, 2012).

```
$ php -S localhost:<port> -t www
```

Druhým krokem je nahrání schématu databáze. Na systému Linux existuje jednoduchá sekvence příkazů, díky které lze vytvořit databázi a následně nahrát schéma databáze. Pokud v databázi existuje role se stejným jménem jako je uživatelské jméno uživatele v operačním systému a má požadovanou autorizaci, může uživatel zadat následující:

```
$ createdb <dbname>
$ psql <dbname> < umbrela.schema.sql
```

Posledním krokem je propojení aplikace a databáze. Pokud by byl použit tradiční přístup k databázi, stačilo by přepsat správné údaje o databázi a uživateli v konfiguračním souboru a připojení by mělo být navázáno. Protože je v plánu použít Doctrine ORM, bude nutné přidat další nastavení. Za předpokladu, že jsou nainstalované všechny závislosti (viz dokumentace), bude potřeba do konfiguračního souboru "config.neon" přidat následující:

```
extensions:
    doctrine: Kdyby\Doctrine\DI\OrmExtension
    annotations: Kdyby\Annotations\DI\AnnotationsExtension
    console: Kdyby\Console\DI\ConsoleExtension
    events: Kdyby\Events\DI\EventsExtension


doctrine:
    user: 'db-role'
    password: 'role-password'
    dbname: 'db-name'
    host: 127.0.0.1
    driver: pdo_pgsql
    metadataCache: default
    metadata:
        App: %appDir%/model/entities
```

Přidáním této konfigurace se propojí knihovna Doctrine s databází a aplikace bude připravena k dalšímu vývoji.

4.4.4 Objektově relační mapování

Existuje dva základní způsoby jak mapovat objekty na databázové tabulky. Entitu nejdříve definovat třídou v aplikaci, nebo ji vygenerovat přímo z databázového schématu. Protože je schéma již definované i se všemi potřebnými tabulkami, bude použitý druhý způsob. Generování entit z databáze nemusí proběhnout přesně podle představ vývojáře a je nutné výsledek důkladně zkontrolovat, nicméně to může ušetřit spoustu času.

Než bude možné entity úspěšně vygenerovat, je nutné přidat další nastavení. V databázi jsou použity vlastní datové typy, které nejsou součástí databázového systému, a tudíž je není možné v knihovně interpretovat. Je proto nutné vytvořit vlastní definice a přidat je do konfigurace. Všechny nové datové typy jsou v základu datový typ "Enum" (výčet). Bude tedy definována abstraktní třída "EnumType", ze které budou dědit všechny ostatní (Doctrine, 2006). Vytvoření aktuálního typu je už velice jednoduché. Stačí přidat název typu a jeho hodnoty. Například pro typy stavů uživatelských účtů je implementace následující.

```
1  class AccountStates extends EnumType
2  {
3      protected $name = 'AccountStates';
4      protected $values = array(
5       'approved',
6       'disapproved',
7       'pending_request',
8       'approved_disabled'
9      );
10 }
```

Chybějící datové typy byly poslední překážkou před generováním entit z databáze. Nejdříve se musí vytvořit databázové schéma i na straně aplikace. Schéma bude prázdné, protože neexistují žádné definice entit. Nicméně tento krok je nutný, aby se aplikace a databáze synchronizovaly. K vytvoření schématu se použije "ConsoleExtension". V kořenovém adresáři aplikace stačí v konzoli spustit následující příkaz.

```
1 $ php ./www/index.php orm:schema-tool:create
```

Tím proběhne vytvoření schématu v aplikaci a může se přistoupit k vygenerování entit.

```
1 $ php ./www/index.php orm:convert-mapping
2   --namespace="App\Model\Entities"
3   --force --from-database annotation "./app/model/entities"
4
5 $ php www/www/index.php orm:generate-entities
6   ./app/model/entities --generate-annotations=true
```

Tyto příkazy vygenerují entity do lokálního schématu a následně vygenerují i PHP třídy do určeného adresáře. Jednotlivé atributy jsou s databází synchronizované pomocí anotací. Vytvoří se i relace mezi jednotlivými entitami a také základní metody jako gettery a settery. Jak již bylo řečeno, proces nemusí být bezchybný a je nutné vytvořené třídy důkladně zkontrolovat. Zejména vazby mezi entitami mohou být náchylné na chyby.

Posledním krokem před tím než bude možné entity používat ve zdrojovém kódu, je nastavení dědičnosti některých tříd. Některé tabulky v databázi, například tabulka "questions", mohou mít několik různých typů. Údaj, o jaký typ otázky se jedná, je uchován ve sloupci "question_type". Rozdíl mezi jednotlivými typy otázek není příliš velký. Typ otázky spíše rozhoduje o věcech typu jaký text nápovědy zobrazit uživateli, jaký druh odpovědí má daná otázka nebo jakou strukturu odpovědí bude otázka mít. Důležité je, že se od sebe otázky nijak fundamentálně neliší a až na výjimky mají všechny atributy stejné. Proto není nutné otázky oddělovat na úrovni databáze. Avšak na úrovni třídy jejich rozdělení přinese bezpochyby několik benefitů. Využití polymorfismu přinese další zjednodušení kódu. Bude tím odstraněno například velké množství podmíněných příkazů a dalších struktur, které se mohou s případným růstem množstvím typů otázek vymknout kontrole. Pro případ rozdělení jedné tabulky do více tříd se použije takzvané "single table inheritace". Vytvoření takových tříd je velice jednoduché. Jediná podmínka, která musí být splněna, je ta, že musí existovat nějaký atribut, podle kterého lze jednoznačně určit, o jakou třídu se jedná. Například v tabulce "questions" půjde tedy o atribut "question_type". Z vygenerované entity se poté udělá abstraktní třída a pomocí anotací se definuje, který atribut je rozhodující pro dědičnost a které třídy jsou svázané s jakým typem otázky (Barsotti, 2012). Definice abstraktní třídy Question bude následující:

```
1  /**
2  * Questions
3  *
4  * @ORM\Table(name="questions", uniqueConstraints={@ORM\
     UniqueConstraint(name="question_order_uniq", columns={"
     sheet_id", "question_order"})}, indexes={@ORM\Index(name="
     IDX_8ADC54D58B1206A5", columns={"sheet_id"})})
5  * @ORM\Entity
6  * @ORM\InheritanceType("SINGLE_TABLE")
7  * @ORM\DiscriminatorColumn(name="type", type="QuestionTypes")
8  * @ORM\DiscriminatorMap({"close" = "CloseQuestion",
9  *      "closeMulti" = "CloseMultiQuestion",
10 *      "closeWithOpen" = "CloseWithOpenQuestion",
11 *      "closeMultiWithOpen" = "CloseMultiWithOpenQuestion",
12 *      "open" = "OpenQuestion"...
13 */
14 abstract class Question
15 {
16     /**
17      * @var integer
18      *
19      * @ORM\Column(name="question_id", type="integer", nullable
          =false)
20      * @ORM\Id
21      * @ORM\GeneratedValue(strategy="SEQUENCE")
```

```
22      * @ORM\SequenceGenerator(sequenceName="
            questions_question_id_seq", allocationSize=1,
            initialValue=1)
23      */
24     protected $questionId;
25 ...
26 }
```

Pro zkrácení ukázky, byla vynechána většina atributů. Pomocí anotace *DiscriminatorColumn* se určí jaký atribut bude rozhodovat o dědičnosti. Anotace *DiscriminatorMAP* pak namapuje další třídy k individuálním typům. Nové třídy už musí pouze dědit z abstraktní třídy Question. Nyní je možné používat tradičních výhod dědičnosti a polymorfismu. Stejným způsobem byla rozdělena i tabulka "options".

4.4.5 Autentizace a autorizace

Ověření identity uživatele bude probíhat klasicky s využitím uživatelského jména a hesla. Bezpečnost hashovacích funkcí je samozřejmě dlouhodobě diskutované téma. Dobré zakódování uživatelského hesla je bezesporu důležitý krok k ochraně uživatelských dat. Jazyk PHP disponuje rozšířením, které se stará o právě kódování hesel. Od verze 5.5 je považováno za velice bezpečné a je doporučeno ho používat. Dvojice funkcí "password_hash()" a "password_verify()", je zodpovědné za kódování řetězce a jeho porovnání. Obecně je doporučeno také ke každému heslo vygenerovat další náhodný řetězec, takzvaný salt, který je při porovnávání přiřazen k heslu a dále zvyšuje složitost prolomení hesla. Funkce "password_hash" se o tento náhodný řetězec stará na pozadí sama (White, 2015). Je tedy logické používat tento způsob kódování hesel.

Práva uživatelů, neboli autorizace, bude realizována na principu Access control list, neboli ACL. ACL je mechanismus, kdy na základě definovaných uživatelských skupin nebo rolí, jsou přiřazena uživatelům jejich přístupová práva. Každá role má definovaný seznam zdrojů, ke kterým má přístup. Pokusí-li se uživatel přistoupit ke zdroji, ke kterému nemá oprávnění, bude přesměrován na stránku s chybovou hláškou (Chin a Beth, 2011). Framework Nette disponuje vlastní třídou právě pro access control list a umožňuje jednoduché definování uživatelských rolí a přidávání práv jednotlivým skupinám. Třída reprezentující ACL musí implementovat rozhraní Nette \Security \IAuthorizator.

Ve funkci setPermission jsou definované role a jejich práva. Implementace této funkce může vypadat následovně:

```
1    private function setPermissions(){
2        //role types
3        $this->acl->addRole('guest');
4        $this->acl->addRole('user', 'guest');
5        ...
```

```
6
7           //resources
8           $this->acl->addResource('Landing');
9           $this->acl->addResource('Auth');
10          $this->acl->addResource('Error4xx');
11          $this->acl->addResource('Researches');
12          $this->acl->addResource('Research');
13          ...
14
15          //permissions
16          $this->acl->allow('guest', array('Auth', 'Landing', '
                Error4xx', 'Api'));
17          $this->acl->allow('user',array('Account'), 'default');
18          $this->acl->allow('user', array('Researches', 'Research
                ', 'EditResearch', 'Fill', 'DataReview', 'Manual'));
19          $this->acl->allow('root', Permission::ALL);
20      }
```

Metoda ”addRole()” třídy Nette \Security \Permission definuje jméno nové uživatelské role. Druhý parametr této metody znamená, že nová role bude mít všechna práva, jako role uvedená v druhém parametru. Metodou ”AddResource” přidáme názvy prezenterů, které mají spadat pod kontrolu a nakonec jednotlivým skupinám přiřadíme místa, ke kterým mají přístup. Jedná se o velice jednoduchý a pohodlný způsob, jakým zaručit správnou autorizaci uživatelů. V některých případech je bohužel nutné kontrolovat oprávnění jiným způsobem než pouze podle uživatelské skupiny a je nutné přidat další kontroly.

Místo, kde ACL model bohužel nestačí, je rozhodování o přístupu k jednotlivým výzkumům. Protože se jedná o dynamicky tvořený obsah webové aplikace, není možné přístup definovat staticky. Rozhodnutí o možnosti náhledu či úpravy výzkumu, je svěřeno jednomu z implementovaných manažerů. Jedná se o třídu, která spravuje entitu výzkumu. Kromě mnoha dalších, její implementace obsahuje metody canView() a canEdit(), které rozhodují, zdali má uživatel přístup k určité operaci. Implemetance obou metod jsou si velice podobné. Rozhodnutí, jestli může uživatel prohlížet daný výzkum, je založeno na tom, jestli se jedná o vlastníka, člena týmu nebo administrátora. Editovat dotazník, může kromě vlastníka a administrátora, pouze člen týmu, který je označen jako editor.

```php
public function canEdit(int $researchId, int $userId): bool
{
  if($this->canView($researchId, $userId)){
    /**@var User $user*/
    $user = $this->entityManager->getRepository(User::class)->
        find($userId);
    /**@var Research $research*/
    $research = $this->entityManager->getRepository(Research::
        class)->find($researchId);
    /**@var PeopleInResearch $member*/
    $member = $this->entityManager->getRepository(
        PeopleInResearch::class)->findOneBy(array('research' =>
         $research, 'user' => $user));
    if($member && $member->getParticipation() !== 'editor')
        return false;
    return true;
  }
  return false;
}
```

4.4.6 Internacionalizace

Mendelova univerzita v Brně aktivně spolupracuje se zahraničními univerzitami a nabízí tamním studentům možnost studia v Česku. To je jeden z důvodů, proč by měla nová aplikace podporovat jazykovou mutaci. Byl tedy zavedený nový systém překladů, který umožňuje překlad statického obsahu do libovolného jazyka. Bohužel zajistit bezchybný překlad i uživatelsky vytvořeného obsahu je zatím nereálné. Avšak poskytnout uživatelům možnost vytvořit si dotazník v cizím jazyce a sbírat data od jiných než českých respondentů, byla i v minulosti žádaná vlastnost.

Pro překlady textových řetězců bude použito knihovny "Kdyby \translation". Pro správnou integraci je nutné provést dodatečné změny v konfiguračním souboru "config.neon". Je nutné zaregistrovat nové rozšíření a nastavit výchozí jazyk, pokud nebude žádný specifikovaný v URL:

```
extensions:
    ...
    translation: Kdyby\Translation\DI\TranslationExtension
    ...

translation:
    default: cs
    fallback: [cs_CZ, cs]
```

Dále jako potřeba nakonfigurovat masku URL, aby bylo možné rozpoznat parametr jazyka. Toho lze dosáhnout jednoduchým přidáním kódu jazyků do masky jednotlivých rout, v nastavení routingu aplikace ve třídě RouterFactory:

```
$router[] = new Route('[<locale=cs cs|en>/]research/<
    research_id>[/<action>]', 'Research:default');
```

Před tím, než bude možné začít překladač používat, musí být zpřístupněn presenterům. Toho se dosáhne jednoduchým přidáním instance překladače a persistentního atributu s kódem jazyka do třídy "BasePresenter()", ze které dědí všechny ostatní presentery. Persistentní atribut má intuitivní název. Jeho hodnota je uchována po dobu uživatelského sezení a je automaticky vkládána jako parametr do URL. Na základě hodnoty tohoto atributu bude následně vybírán jazyk aplikace.

Po provedení tohoto nastavení je možné používat překlady jak v šablonách, tak ve všech třídách aplikace. Teď už jen zbývá přidat soubory, které obsahují samotné překlady. Ve výchozím nastavení se tyto soubory umístí do adresáře "lang", s názvem "messages.cs_CZ.neon". Kód je samozřejmě odlišný pro různé jazyky. Struktura souboru s překlady je následující:

```
...
forms:
    firstName: "Jméno"
    email: "Emailová adresa"
...
```

Pokud je použitý šablonovací systém Latte, tak lze texty překládat pomocí makra "{_messages.forms.login}", a pokud má být text přeložen přímo ve zdrojovém kódu aplikace, stačí předat kód řetězce překladači následujícím způsobem "$translator->translate('mesages.forms.login')". Aplikace bude prozatím podporovat pouze český a anglický jazyk. Nicméně systém je přichystám takovým způsobem, že pro přidání libovolného jazyka stačí dodat pouze soubor s překlady se stejnou strukturou jako mají již existující soubory a do routeru přidat nový kód jazyka. Jediné, co reálně brání přidání jakéhokoliv jazyka, je nalezení osoby, která by poskytla kvalitní překlad.

4.4.7 Nařízení GDPR

25. května 2018 vejde v platnost nařízení Evropského parlamentu a Rady EU o ochraně osobních údajů fyzických osob. Osobní údaje jsou podle GDPR veškeré informace, podle kterých je možné identifikovat fyzickou osobu (Nezmar, 2017). Podle této definice i aplikace Umbrela ukládá osobní údaje o uživatelích.

Většina povinností tohoto nařízení by měla být implementována na úrovni organizace (v tomto případě na úrovni univerzity). Samotná aplikace musí splňovat tyto požadavky. Při registraci musí uživatel souhlasit se uchováváním osobních údajů a musí být informován jakým způsobem budou tyto údaje zpracovávány. Dále

mu musí být poskytnut kontakt na správce osobních údajů a také by měla existovat možnost na anonymizaci uživatele (Calder, 2016).

Bohužel v době psaní práce neměla Mendelova univerzita v Brně žádným způsobem implementované toto nařízení, a proto nebylo možné plně implementovat všechny požadavky. Vzhledem k tomu že, neexistovala žádná pověřená osoba ani obecné podmínky zpracování osobních údajů v rámci univerzity, nebylo možné vypracovat podmínky pro používání aplikace ani poskytnout kontakt na správce údajů. Anonymizaci údajů je však možné nachystat již s předstihem, než budou všechny detaily vypracovány.

V aplikaci jsou uchovávány tyto osobní údaje, které by mohly vést k identifikaci fyzické osoby. Jméno, příjmení, uživatelské jméno (login) a emailová adresa uživatelů aplikace. Respondenti nemají povinnost jakékoliv registrace či uvádění jakýchkoliv údajů. V minulosti se sice ukládaly i informace o IP adresách respondentů, ale tento údaj byl z databáze odstraněn.

Byly prodiskutovány dva možné způsoby odstranění zmíněných údajů. Smazání uživatele z databáze a zakódování údajů do řetězců pomocí jednostranné hashovací funkce. Smazáním účtů by došlo k ovlivnění i výzkumů a nasbíraných dat. K této variantě se zadavatel příliš nepřikláněn. Proto byla zvolena cesta kódování osobních údajů uživatelů. Byla vytvořena jednoduchá metoda ve třídě "UserManager", která se postará o anonymizaci údajů a deaktivuje účet. Každý údaj je zakódován jednostrannou hashovací funkcí "crypt". Také je k němu přidaný náhodný řetězec, aby se ještě více zvýšila bezpečnost hashovací funkce. Implementace je následující:

```
1  public function hashUserIdentity($userId): User
2  {
3      $user = $this->getUser($userId);
4      $user->setEmail(crypt($user->getEmail(), RandomString::
           randomStr(255)) . '@' . crypt($user->getEmail(),
           RandomString::randomStr(255)));
5      $user->setState('approved_disabled');
6      $user->setLogin(crypt($user->getLogin(), RandomString::
           randomStr(255)));
7      $user->setFirstName(crypt($user->getFirstName(),
           RandomString::randomStr(255)));
8      $user->setLastName(crypt($user->getLastName(), RandomString
           ::randomStr(255)));
9      $this->entityManager->merge($user);
10     $this->entityManager->flush($user);
11     return $user;
12 }
```

Funkce anonymizace je dostupná pouze uživatelům z administrátorské skupiny a mohou k ní přistoupit ze sekce správy uživatelů.

4.4.8 Automatické testování

Jednotkové, neboli unit testy, jsou v dnešní době součástí kteréhokoli kvalitního softwaru. Unit testy jsou zpravidla tvořeny paralelně se zdrojovým kódem. Kód, který je testy dobře pokrytý, je méně náchylný ke vzniku nových chyb při dodatečných změnách, a značně tak ulehčuje dlouhodobý vývoj (Hamill, 2004).

Vzhledem k inkonzistenci týmu pracujícího na vývoji aplikace Umbrela jsou unit testy ještě důležitější. Spouštění testů by mělo v budoucnu předejít kritickým chybám. Samozřejmě nemá smysl testovat triviální metody, jako například "getter" nějaké třídy. 100 % pokrytí kódu ještě neznamená 100 % bezchybnost softwaru. Nicméně všechny kritické metody a funkce by měly být pokryty testy.

K testování serverové části aplikace byl použit balík Nette Tester. Jak již název napovídá, tento balíček je vytvořený na míru k použitému frameworku a měl by značně ulehčit konfiguraci testů a testování některých specifických částí frameworku.

Protože serverová část aplikace je zodpovědná hlavně za správnou manipulaci s databází a předávání dat JavaScriptovým modulům na klientské stránce aplikace, musí být otestovány hlavně zmíněné manažery databázových entit. K tomu je samozřejmě potřeba databáze a správná konfigurace. Jelikož není praktické používat stejnou databázi k vývoji a testování, bude zapotřebí nachystat individuální prostředí i pro unit testy. Nette Tester naštěstí poskytuje celkem dobré nástroje pro konfiguraci testů. V zaváděcím souboru "bootsrap.php" v sekci testů je možné specifikovat separátní konfiguraci aplikace speciálně určenou pro testy (config.test.neon). V té se může specifikovat jiná databáze, která bude použita speciálně pro testovací prostředí. Databázi je také dobré před spouštěním testů vyprázdnit, aby nevznikaly nejrůznější konflikty mezi spuštěními. K tomuto účelu byl vytvořen jednoduchý script, který provede všechny zmíněné aktivity a spustí testy.

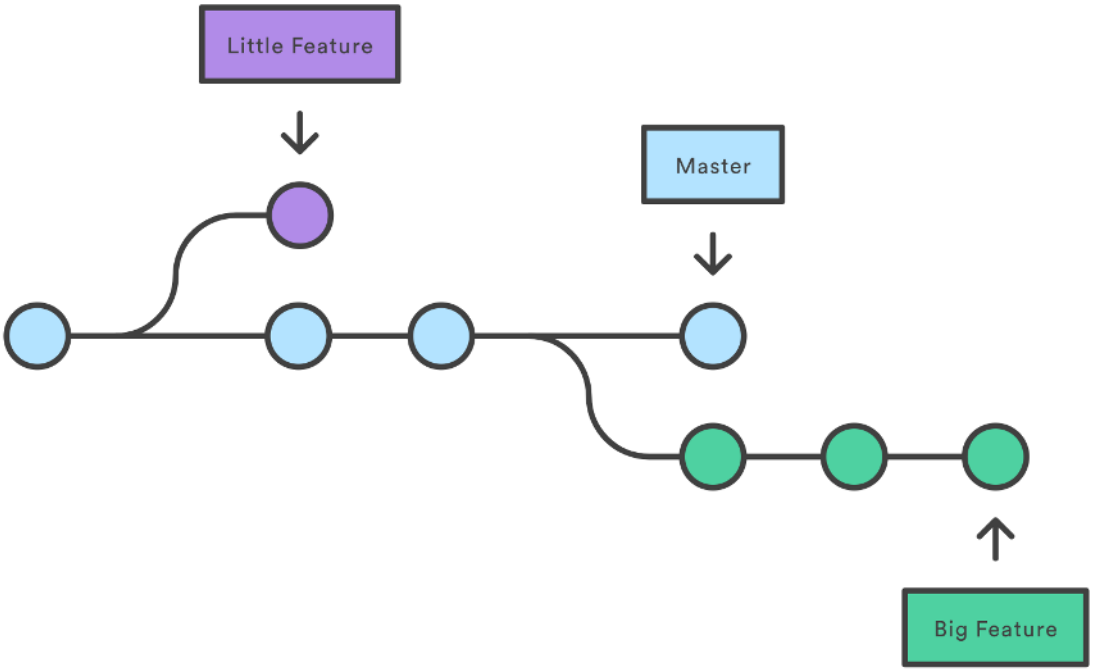
```
1  #!/usr/bin/env bash
2
3  echo "Test database setup"
4  if psql -lqt | cut -d \| -f 1 | grep -qw umbrela-test-db; then
5      echo "Testing database exist"
6      echo "Dropping previous testing database"
7      dropdb umbrela-test-db
8      echo "Testing database removed"
9  fi
10 echo "Creating empty testing database"
11 createdb umbrela-test-db
12 psql umbrela-test-db < umbrela.schema.sql
13 php ../vendor/nette/tester/src/tester -p php -C -w ../
```

Skript smaže existující testovací databázi a vytvoří novou a následně ji naplní prázdným schématem. Poslední řádek pak spouští samotné unit testy.

4.4.9 Continuous integration

Agilní metodiky vývoje softwaru mají jistou nevýhodu v tom, že se kód často mění. Zvyšuje se tak i riziko, že se do produkční verze aplikace náhodou dostane i kód s chybami. Je samozřejmě žádoucí takové situaci předejít. Continuous integration obsahuje sadu pravidel a pomůcek, které takové riziko snižují.

Správný management zdrojového kódu je prvním krokem k pokročilé integraci. Jak již bylo zmíněno, aplikace Umbrela využívá software git ke správě zdrojového kódu, konkrétně webové aplikace BitBucket. Bude používáno klasického schématu větví, kde master větev bude uchovávat produkční verzi kódu a nebudou v ní probíhat žádné změny. Aby se vynutil tento postup i pro budoucí vývojáře, ve větve master bude zakázáno provádět jakékoliv změny.



Obrázek 12: Tradiční schéma vývojových větví. (GitHub, 2007)

Druhým a důležitějším krokem je automatické spouštění testů přímo v repositáři. Stejně jako jiné distribuce softwaru git, i BitBucket nabízí možnost spouštění testů ve virtuálním prostředí. Tato služba se jmenuje Bitbucket pipelines. Jedná se o integrovanou službu CI/CD, která je zabudovaná do repositářů. Umožňuje automaticky vytvářet, testovat a nasazovat kód na základě konfiguračního souboru (Bitbucket, 2017).

Virtuální prostředí je vytvářeno pomocí docker image. Na internetu je volně dostupné nepřeberné množství různých přichystaných obrazů, které je možné použít. Samotná konfigurace automatického spouštění testů je velice jednoduchá. V konfi-

guračním souboru se nastaví obraz prostředí a sekvence příkazů, která se při zápisu kódu do repositáře spustí:

```
image: tripways/php-pgsql

pipelines:
 default:
   - step:
       caches:
         - composer
       script:
         - apt-get update && apt-get install -y unzip
           postgresql-client php7.1-gd
         - apt-get install php-gd
         - curl -sS https://getcomposer.org/installer |
           php -- --install-dir=/usr/local/bin --
           filename=composer
         - composer install
         - mkdir temp
         - mkdir temp/sessions
         - bin/test-ci.sh
       services:
         - postgres

definitions:
 services:
   postgres:
     image: postgres:9.6
     environment:
       POSTGRES_USER: 'postgres'
```

Ke spouštění testů musela být také vytvořená další konfigurace aplikace a shell skript, který bude testy spouštět.

4.4.10 Dokumentace

Součástí této práce je samozřejmě i podrobná dokumentace celé aplikace. Díky charakteru celého projektu byly vytvořeny dva druhy dokumentace. Jedna popisuje zdrojový kód a druhá je jakousi příručkou pro vývojáře, která by jim měla pomoci zapojit se do projektu.

Dokumentace kódu je dostupná ve dvou formách. Přímo v kódu a samostatný HTML soubor, který je generován právě z dokumentace kódu. Toho je dosaženo pomocí komentářů a anotací standardu PHPDoc. Díky této konvenci jsou popsány jednotlivé třídy, jejich atributy, metody, typy parametrů, návratových typů a dal-

ší. O samotné generování dokumentace zdrojového kódu se stará php balík phpDocumentor. PhpDocumentor je nástroj, který umožňuje generovat dokumentaci přímo ze zdrojového kódu PHP. Díky tomu je možné rychle získat detailní informace o funkcích softwaru. Samozřejmě že vygenerovaná dokumentace neposkytne dokonalý obraz o softwaru. Jedná se spíše o doplnění klasické dokumentace.

4.4.11 Nasazení

Díky omezené době běhu virtuálního stroje na repositáři by nebylo praktické využít funkce continuous delivery. Také neexistuje žádný předprodukční server, na kterém by mohly nové verze softwaru nějaký čas běžet a mohly by projít manuální kontrolou nových, či upravených funkcí aplikace. A i kdyby takový server existoval, není k dispozici žádný quality assurance tým, který by mohl nové verze kontrolovat. Chybovat je lidské a bez důkladné kontroly není automatické nahrávání nových verzí bezpečné. To je několik důvodů, proč zatím nebude do projektu zapracována technika continuous delivery. Každopádně by bylo dobré tento proces co nejvíce zjednodušit.

Problémem nahrávání nové verze přímo z repositáře je hlavně rozdílné prostředí, na kterém je produkční aplikace spuštěná. Jméno databáze, přihlašovací údaje databáze nebo třeba režim spuštění aplikace jsou atributy konfigurace, které bezpochyby budou odlišné na vývojovém a produkčním prostřední. Ke správnému nasazení nové verze, byl vytvořen další script. Ten nejdříve smaže starou zálohu aplikace a vytvoří novou z aktuální verze. Následně z repositáře stáhne změny z produkční větve. Protože je v aplikaci hned několik různých konfigurací pro různé situace, existuje také šablona konfiguračního souboru. Ten se nakopíruje a následně se doplní všechny údaje, které jsou specifické pro produkční prostředí. Nakonec se aktualizují všechny závislosti projektu a nastaví se správná práva k některým adresářům a souborům.

Samozřejmě že zavedení continuous delivery by bylo mnohem pohodlnější, ale vzhledem k situaci bylo nutné najít alternativní řešení. Vymyšlený způsob je ale také do jisté míry automatizovaný. Jediné, co musí administrátor udělat navíc, je přihlásit se na server a spustit jeden script. Pokud by však v budoucnu vznikl větší vývojářský tým a celkově by se zlepšily podmínky projektu, plná automatizace procesu nasazení je rozhodně správná cesta vpřed.

4.5 Frontend

V následujících dvou kapitolách bude popsán způsob, jakým se přistupovalo k vývoji uživatelského rozhraní. Bude popsán způsob vývoje stěžejních částí uživatelského rozhraní. Těmi jsou editor dotazníku a modul pro vyplňování dotazníku. Také budou vysvětleny myšlenkové pochody při volbě grafického designu a způsob výměny dat mezi serverovou a klientskou částí aplikace.

4.5.1 Nittro

Single page application je pojem, který je v dnešní době úzce spojen s vývojem webových aplikací. Úkolem SPA je vytvořit aplikace, které mají intuitivnější uživatelské rozhraní a práce s aplikacemi je plynulejší a přirozenější. Tyto aplikace jsou také rychlejší, protože nemusí neustále načítat obsah celého webu, ale pouze mění určité části uživatelského rozhraní přímo v okně prohlížeče, zatímco komunikace se serverem probíhá na pozadí. Tohoto všeho je možné dosáhnout díky JavaScriptu (Monterio, 2014).

Už v předchozí části práce byly zmíněny nejrůznější knihovny a technologie založené na jazyce JavaScript. Bylo také zmíněno, že část rozhraní bude realizována v knihovně React. Tvoření takového uživatelského rozhraní je však časově velice náročné a mnohdy může trvat déle než implementace serverové části. Proto budou tímto způsobem vytvořeny jen ty části rozhraní, kde je to nezbytné. Nicméně výhody SPA není možné ignorovat. Jakákoliv jejich implementace, ať už z technického či uživatelského pohledu, je přínosná. Proto bude použity knihovny Nittro. Tato knihovna je podle jejich tvůrců client-side framework, zaměřený na ulehčení vývoje uživatelského rozhraní webových aplikací postavených na frameworku Nette (Nittro, 2015).

Součástí této knihovny je například validace formulářů nebo optimalizované metody pro REST komunikaci se serverem. Co je však nejdůležitější, je možnost transformace klasického způsobu komunikace a načítání individuálních stránek aplikace. Díky využití takzvaných snippetů, lze pomocí této knihovny velice snadno využít dvou výhod SPA. Těmi jsou komunikace se serverem na pozadí pomocí JavaScriptu a renderování uživatelského rozhraní přímo v prohlížeči.

Aby se dalo využívat těchto výhod, musí být nejdříve provedena následující konfigurace na serverové stránce. Do seznamu php závislostí se musí přidat jednoduchá knihovna "nittro/nette-bridges", pomocí které bude možné přizpůsobit komunikaci se serverem pro potřeby SPA a také přidá makra do šablonovacího systému. Do konfiguračního souboru config.neon se musí registrovat nová zmíněná makra:

```
1 latte:
2     macros:
3         - Nittro\Bridges\NittroLatte\NittroMacros
```

Dále se ve třídě "BasePresenter" musí provést pár změn. Musí být přidána proměnná, která bude signalizovat, zdali má být odpovědí na http požadavek klasická webová stránka, nebo jestli se jedná o AJAX komunikaci a je zapotřebí odeslat pouze data. Také musí nastavit výchozí snippety, ve kterých bude uložen dynamický obsah stránek. Třída také nebude dědit z výchozího presenteru Nette, ale z modifikovaného z knihovny Nittro. Na ukázce lze vidět, že není nutná příliš složitá příprava:

```
1  abstract class BasePresenter extends Presenter
2  {
3      ...
4      /**
5       * signal for nittro payloads
6       */
7      private $signalled = false;
8      /**
9       * Is signalled
10      * @return bool
11      */
12     protected function isSignalled()
13     {
14         return $this->signalled;
15     }
16     ...
17     public function startup()
18     {
19    ...
20         $this->signalled = $this->getSignal() !== null;
21         $this->setDefaultSnippets(['header', 'content', 'menu',
               'title']);
22    ...
23
24     }
25     ...
26 }
```

Jediné, co je nutné při vytváření rozhraní v šablonovacím systému latte dodržovat, je správné umístění elementů do správných snippetů a obsah bude dynamicky překreslován. Vyhneme se tak neustálému zasílání nejrůznějších dat, jako jsou CSS styly, obrázky, JavaScriptové soubory a další. Vše bude načteno pouze jednou a pouze skutečný obsah stránek se bude měnit.

4.5.2 Less

K upravování vizuální stránky webových stránek se používá takzvaných kaskádových stylů. CSS není programovací jazyk a neumožňuje vytvářet žádné proměnné nebo funkce, které by se daly opětovně využít. S růstem projektu se stává údržba problémem. Také implementace CSS se u různých prohlížečů liší a je nutné přidávat prefixované názvy atributů, nebo i kompletně rozdílné definice stylů, aby se vzhled stránek nelišil na základě prohlížeče. Tyto problémy částečně řeší CSS preprocesory. Jejich pokročilé funkce pomáhají vytvářet znovupoužitelné, udržovatelné a rozšiřovatelné definice stylů a pomáhají redukovat množství kódu v projektu (Cinarli, 2014).

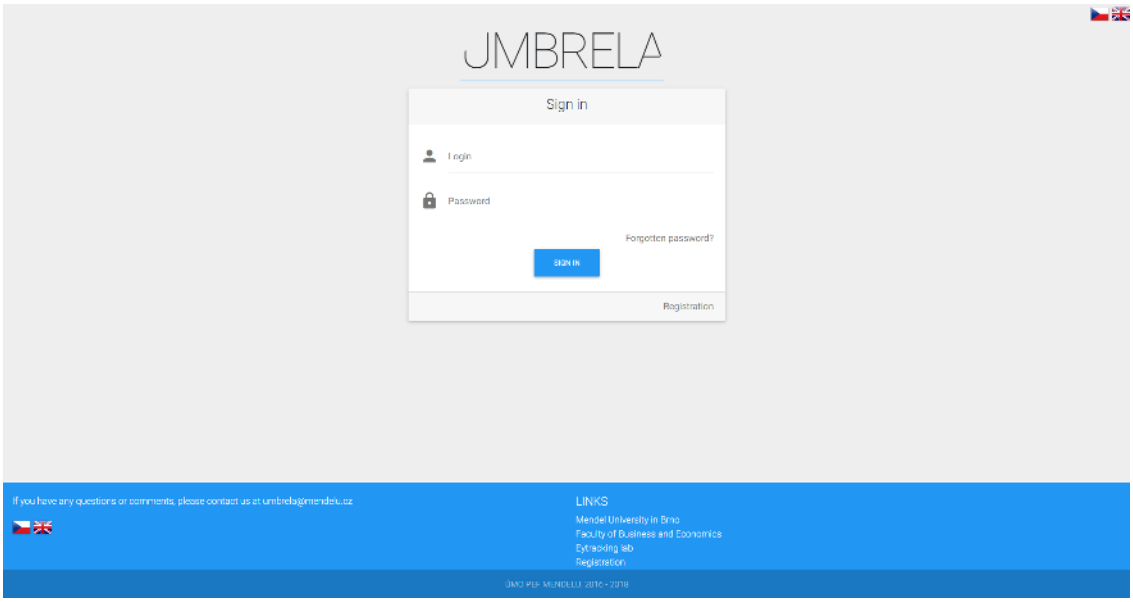
V současnosti jsou nejpoužívanější preprocesory SASS, LESS a Stylus. Při bližším zkoumání funkcí je zřejmé, že se od sebe podstatně neliší. Pro potřeby tohoto projektu, příliš nezáleží na tom, který bude vybrán. Vzhledem k předchozím zkušenostem byl vybrán preprocesor LESS.

4.5.3 Material design

Design uživatelského rozhraní je první věc, které si jakýkoliv uživatel jakékoliv aplikace všimne ihned. Často může rozhodovat o tom, jestli bude uživatel aplikaci vůbec používat. Pokud ho na úvodní stránce přivítá pěkný vzhled, s velkou pravděpodobností na webu zůstane. Naopak špatný design může uživatele ihned odradit od dalšího využívání aplikace a bude hledat alternativy.

Vytvoření opravdu profesionálního designu rozhraní pro jakýkoliv software je neuvěřitelně složitý proces a může zabrat obrovské množství času. Existuje také velké množství literatury, které popisují desítky různých stylů, kterými je možné se vydat. Při hlubším prostudování různých zdrojů a přihlédnutí ke schopnostem vývojáře, které jsou zaměřené spíše na technickou stránku vývoje webových aplikací, bylo jasné, že vybudování designu uživatelského rozhraní od nuly, by byl nezvladatelný úkol. Proto při vytváření vzhledu aplikace bylo využito směrnic firmy Google.

Google směrnici material design poprvé oznámil na I/O konferenci v létě 2014. Založený převážně na starodávných principech vycházejících z teorie barev, animace, tradičního tisku a fyziky, material design poskytuje virtuální prostor, kde mohou vývojáři používat povrch a světlo pro vytváření smysluplných rozhraní a pohybu k vytvoření intuitivních uživatelských rozhraní (Mew, 2015). Původně byl určený hlavně pro mobilní aplikace, nicméně obrovský úspěch tohoto stylu měl za důsledek velice rychlou adopci podobných praktik i na poli webových aplikací. V tomto projektu bylo použito jedné z mnoha interpretací knihovny Bootstrap, která implementuje právě material design. S pomocí této knihovny bylo vytvořeno intuitivní, jednoduché a responsivní uživatelské rozhraní, které by mělo zpříjemnit práci všem stávajícím či budoucím uživatelům aplikace Umbrela.



Obrázek 13: Přihlašovací stránka

4.6 React

V rámci této práce už byla několikrát zmíněna knihovna React. V předchozí verzi aplikace byla také využita a jak již bylo řečeno, neexistuje žádný důvod, proč by se neměla používat i nadále. Oproti uživatelskému rozhraní využívajícímu tradičního značkovacího jazyka HTML, tvorba rozhraní v knihově React nebo jakékoliv její alternativě je mnohem složitější. V této práci bude k modulům využívajícím React přistupováno jako k separátnímu projektu, který bude do aplikace integrován. Jedná se o zcela odlišné principy a technologie, než u zbytku projektu a také to přispěje k lepší organizaci celé práce. Teoreticky by práce na obou částech mohla probíhat paralelně a rozdělení obou částí aplikace do samostatných projektů je běžnou praktikou. Jedinou podmínkou pro bezchybnou integraci je dodržení definic API na obou stranách. V následujícím textu budou popsány stěžejní technologie a praktiky, které byly použity při vývoji modulů pro editor dotazníku, vyplňování dotazníků a přehled dat výzkumu.

4.6.1 Vývojové prostředí

Stejně jako u backendu, i v této části je nutné si správně nachystat vývojové prostředí. Pro tento případ to bude o něco jednodušší. K tomu, aby mohla začít práce na projektu je nutné pouze nainstalovat Node.js. Stejně jako tomu bylo u frameworku Nette, z příkazové řádky lze jednoduše vytvořit prázdný projekt aplikace. Toho lze dosáhnout následující sekvencí příkazů:

```
1 $ npm -g i create-react-app
2 $ create-react-app umbrela-react-modules
```

4.6.2 Webpack

Protože všechny moduly se budou chovat jako samostatné části, je dobré k nim tak přistupovat. Za tímto účelem je použita knihovna webpack. Jedná se především o takzvaný module bundler. Shromažďuje všechny JavaScriptové knihovny, CSS soubory a další zdroje, které byly použity pro vytvoření nějakého programu a vytvoří z nich samostatné balíky (Brown, 2017). Kromě toho může být také využitý k minifikaci zdrojového kódu a pro účely této práce, také k rozdělení jednotlivých částí programu na samostatné celky.

Za účelem optimalizace by měl každý z modulů být v samostatném souboru. Protože všechny moduly budou sdílet určité části kódu, bude také vytvořen soubor, který obsahuje všechny společné části modulů. Protože k vytvoření modulů byl použitý script create-react-app, bude nejdříve nutné si zpřístupnit konfiguraci. Toho lze dosáhnout pomocí příkazu "npm run eject" v terminálu v adresáři projektu. Veškerá konfigurace, které byla doposud součástí knihovny, bude rozbalena do souborů, jako při ruční konfiguraci projektu. Nyní je možné dále upřesnit parametry nastavení kompilace zdrojového kódu do produkčních balíčků.

K tomu, aby bylo možné se k jednotlivým modulům chovat jako k samostatnému celku, je nejdříve nutné definovat vstupní body. To jsou "kořenové soubory", které v sobě uchovávají kořenové komponenty modulu. Aby mohly být při vývoji individuálně přístupní, bude potřeba ke každému z nich přiřadit samostatnou HTML šablonu která bude dostupná na vývojovém serveru. V konfiguračním souboru "webpack.config.dev.js" je nutné upravit následující:

```
...
module.exports = {
  ...
  entry: {
    fillModule:[
      require.resolve('react-dev-utils/webpackHotDevClient'),
      require.resolve('./polyfills'),
      require.resolve('react-error-overlay'),
      paths.umbrelaFillModule,
    ],
    researchEditor:[
      require.resolve('react-dev-utils/webpackHotDevClient'),
      require.resolve('./polyfills'),
      require.resolve('react-error-overlay'),
      paths.researchEditorJs,
    ],
    dataReview:[
      require.resolve('react-dev-utils/webpackHotDevClient'),
      require.resolve('./polyfills'),
      require.resolve('react-error-overlay'),
      paths.dataReviewModuleJs,
    ],
  },
  ...
}
```

V ukázce je nastavení pro každý modul. Ty se liší pouze cestou k zaváděcímu souboru. Teoreticky by bylo možné tyto souboru definovat pomocí jednoduché funkce, aby se ušetřilo pár řádků kódu. Explicitní zadání je však mnohem přehlednější a modulů není velké množství. Automatizovaná konfigurace je vhodnější spíše pro knihovny komponent nebo jiné projekty s velkým počtem vstupních bodů.

Pro nastavení separátních HTML šablon a jejich zpřístupnění na vývojovém serveru se musí přidat následující kód:

```
module.exports {
  ...
  plugins: [
     new InterpolateHtmlPlugin(env.raw),
     new HtmlWebpackPlugin({
          inject: false,
          template: paths.appHtml,
        }),
        new HtmlWebpackPlugin({
          inject: true,
          chunks: ["fillModule"],
          template: paths.umbrelaFillModuleHtml,
          filename: 'fillModule.html',
        }),
        new HtmlWebpackPlugin({
          inject: true,
          chunks: ["researchEditor"],
          template: paths.researchEditorHtml,
          filename: 'researchEditor.html',
        }),
        new HtmlWebpackPlugin({
          inject: true,
          chunks: ["dataReview"],
          template: paths.dataReviewModuleHtml,
          filename: 'dataReviewModule.html',
        }),
    ...
}
```

Kromě šablon modulů bylo také vytvořeno jakési rozcestí, které při spuštění vývojového serveru zobrazí jednoduchý seznam odkazů na specifické moduly. Tato stránka byla přidána čistě pro ulehčení vývoje a není kompilována do produkčních modulů.

Produkční verze modulů potřebuje také specifické nastavení, které je uloženo v souboru "webpack.config.prod.js". Stejně, jako u předchozího případu, je nutné definovat vstupní body. Protože se nebude jednat o vývojové prostředí, není nutné podporovat "hot reloading" ani renderovat debugovací hlášky. Definice vstupního bodu je následující:

```
module.exports = {
  ...
  entry: {
    fillModule:[
      require.resolve('./polyfills'),
      paths.umbrelaFillModule,
    ],
  ...
  }
}
```

Na rozdíl od definic šablon, ze kterých je cílový kód kompilován, produkční verze modulů je o pár vlastností rozšířená. Jedná se zejména o optimalizaci kódu jako jsou minifikace všech zdrojových souborů, odstranění komentářů a výpisů do konzole. To je příklad optimalizací, které jsou prováděny pro produkční verzi balíčků. Posledním rozdílem, oproti výchozímu nastavení kompilace, je sloučení sdílených knihoven a komponent do jednoho souboru, který nese název "vendor". Do části pluginů se přidá následující:

```
module.exports = {
  ...
  plugins: [
    ...
    webpack.optimize.CommonsChunkPlugin({
      name: 'vendor',
      minChunks: module => module.context && module.context.
          indexOf('node_modules') !== -1,
      Infinity
    }),
   ...
  ]
}
```

Všechna popisovaná nastavení mají za cíl vytvořit samostatné balíčky pro každý z vytvářených modulů a navíc jeden sdílený balíček s knihovnami, které budou použity při vývoji. Ostatní parametry budou ponechány ve výchozím stavu.

4.6.3 Redux

Jednou vlastností knihovny React, která je často opomíjena, je fakt, že se nejedná o MVC framework jako je například Angular.js. React je pouze knihovna pro vytváření uživatelského rozhraní a žádným způsobem neřeší ukládání stavu aplikace. I když je tato skutečnost uvedena i v oficiální dokumentaci, spousta vývojářů ji ignoruje a pro ukládání dat využívá interního stavu jednotlivých komponent. Při složitější struktuře dat a nutnosti sdílení stejných dat ve více částech aplikace je však

používání stavu komponent komplikované a komponenty se stávají příliš složité. Také pouhá myšlenka toho, že by například komponenta tlačítka měla mít přístupná veškerá data a uchovávat jejich stav, je poměrně bizarní.

Zejména z těchto důvodů se postupem času začaly objevovat takzvané "state managers" nebo "state containers" knihovny pro JavaScript a Redux je jednou z nich. Pomáhá vytvářet, spravovat a sdílet stav aplikace napříč všemi komponentami, bez toho aniž by komponenty musel být navzájem přímo propojeny (Abramov a Clark, 2015).

Koncept této technologie je postavený na třech základních pravidlech. Stav je udržován v jednom velkém skladu, který je dostupný z celé aplikace, je založený na návrhovém vzoru immutable a je měněn pomocí takzvaných "pure functions".

Myšlenka toho, že stav je jeden velký objekt, se na první pohled může zdát zvláštní. Nebude problém s výkonem? Jak organizovat velké množství dat? To jsou asi nejběžnější dotazy, které se mohou po zjištění toho faktu vyskytnou. Store, který ukládá stav, není nic jiného než obyčejný objekt. Má atributy a ty mají nějaké hodnoty. Tím pádem lze i velice jednoduše stav logicky rozdělit na více částí, které mohou korespondovat s uspořádáním aplikace. Protože objekt v JavaScriptu funguje na principu hashovací tabulky, tak i v případě opravdu složitých struktur je dosaženo dobrého výkonu.

Změny stavu jsou prováděny takzvanými "reducery". Reducer je obyčejná funkce, která poslouchá na definované signály (akce) a na základě typu a obsahu signálu změní stav aplikace. Právě díky reducerům je možné stav rozdělit na logické celky, kdy každý reducer obsluhuje právě jednu část stavu aplikace.

Teď když je jasné, jakým způsobem se bude ukládat stav v aplikaci, je možné přejít k implementaci. K vytvoření instance "store" je nutné nejdříve definovat reducery. Podle popsané definice bylo vytvořeno několik funkcí, které budou ovládat stav aplikace. Implemetace takové funkce může být následující:

```
1  import {
2    UPDATE_RESEARCH_STATE,
3    START_RESEARCH
4  } from '../actions/actionTypes';
5
6  let research = {
7      research: false,
8      activeSheet: null,
9      sheets: [],
10     filters: [],
11 };
12
13 function researchReducer(state = research, action) {
14   switch (action.type) {
15     case UPDATE_RESEARCH_STATE:
16       return {
17         ...action.research,
```

```
18          activeSheet: null,
19          research: true,
20        }
21      case START_RESEARCH:
22        return {...state, startTime: new Date().getTime()};
23      default:
24        return state;
25    }
26  }
27
28  export default researchReducer;
```

Na ukázce je příklad velice jednoduché funkce, která poslouchá pouze na dva typy různých akcí. Jak je vidět, pokud funkce zachytí nějakou akci, provede změny stavu a vrátí jeho novou instanci. Redux využívá návrhového vzoru immutable, aby byl schopen odlišit, kdy se stav skutečně změnil. Je také nutno podotknout, že tyto funkce nejsou v kódu přímo přístupné, ale jsou provolávány pomocí funkce ”dispatch” které je součástí stavu aplikace. Akce, na které funkce čekají, jsou velice jednoduché funkce, které vracejí jednoduché objekty. Příklad definice takové akce může být následující:

```
1  export const someAction = payload => ({
2    type: ACTION_TYPE,
3    payload
4  })
```

Opět se jedná jednoduchou strukturu. Jediným povinným atributem akce je její typ. To jaké a jestli bude mít i jiné atributy, už závisí na situaci, pro kterou je určena. Je nutné také podotknout, že na jednu akci může poslouchat více reducerů a akce může ovlivnit hned několik částí storu najednou. Kromě těchto primitivních funkcí lze také definovat asynchronní akce, které jsou určené ke komunikaci se serverem a i k synchronizaci stavu s databází. Příklad takové asynchronní akce může být následující:

```
...
export function getQuestions(sheetId){
  return dispatch => {
    dispatch(requestQuestions());
    return getQuestionsRequest(sheetId)
    .then(response => response.json())
    .then(json => dispatch(receiveQuestions(json)))
    .then(() => dispatch(finishFetch()))
    .catch((err) => {
      console.log('failed to fetch: ', err);
      dispatch(fetchFailed(QUESTION_FETCH_FAILED))
    });
  }
}
...
```

Tato akce má za úkol získat data o otázkách ze serveru a následně je předat k uložení do stavu. Samotná akce vyvolává hned několik dalších akcí a stav je aktualizovaný hned několikrát. Celý proces lze popsat následovně. První aktivita, po vyvolání akce nějakou událostí ve webovém prohlížeči, je zaslání signálu, že aplikace bude požadovat data ze serveru akcí "dispatch(requestQuestions())". Stav aplikace ji zaregistruje, a to se může projevit například renderováním symbolu pro načítání. Následně je zaslán požadavek na server a čeká se na odpověď. Pokud server bez chyby odpoví, tak jsou data uložena do skladu další akcí "dispatch(receiveQuestions(json)" a následně je vyslán další signál, který informuje o úspěšném získání dat a otázky mohou být vykresleny. Pokud nastane chyba, je odchycena a stav aplikace je aktualizován s informací, že se nepodařilo provést akci. Na chybu může být opět reagováno například renderováním chybové hlášky. Tímto způsobem jsou získávána data do jednotlivým modulů a také je zajištěna synchronizace dat se serverem. Jak je vidět z ukázky, akce na sebe mohou navazovat a jedna událost v aplikaci může vyvolat celou sérii nejrůznějších aktivit.

Pokud jsou definovány reducery, pomocí funkce "combineReduceres(name: reducer)" se spojí do jednoho objektu a mohou se předat funkci createStore(reducer), která vytvoří store pro celou aplikaci. Obě tyto funkce jsou součástí knihovny redux. Jakmile existuje stav aplikace, je potřeba ho také zpřístupnit. K tomu slouží context zvaný Provider.

Context může být chápán jako speciální varianta React komponenty. Pokud by bylo zapotřebí zpřístupnit vlastnost nějaké komponenty jejím potomkům, musela by se jim explicitně předat. Aplikovat toto předávání na všechny akce, reducery a další součásti manažera stavu aplikace by bylo velice nepraktické, protože každá komponenta by měla desítky parametrů. Pokud však komponentě do jejího kontextu přidáme tyto atributy, budou dostupné všem jejím přímým i nepřímým potomkům automaticky (Franklin, 2017). Předefinovaný kontext Provider disponuje všemi po-

třebnými funkcemi a jeho integrace je stejná, jako použití obyčejné komponenty. I když může být umístěn na jakémkoliv místě v aplikaci, je dobré jej umístit jako kořenovou komponentu, aby byl opravu dostupný z celé aplikace:

```
...
export const ResearchEditorModuleWrapper = () => (
  <Provider store={store}>
    Some components
  </Provider>
)
...
```

Při použití kontextu je sice store dostupný pro všechny aplikace, ale bez řádného připojení komponent by mohl být přístupný přímo a tím by se mohla narušit jeho struktura. Je tedy doporučeno stav měnit pouze definovanými akcemi. Ke správnému propojení komponenty, stavu a akcí je tedy nutné použít jednoduchého dekorátoru connect, který je dostupný z knihovny react-redux. Připojení komponenty může vypadat takto:

```
import {connect} from 'react-redux';
import {bindActionCreators} from 'redux'
import {someAction} from 'path/to/action'

const SomeComponent = ({title, onClick})=> (
  <div>
    <label>{title}</label>
    <button onClick={onClick}>Click me!</button>
  </div>
)

const mapStateToProps = ({reducer}) => ({
  title: reducer.someString
})

const mapDispatchToProps = dispatch => ({
  bindActionCreators({
    onClick: someAction
}, dispatch),
})

export default connect(mapStateToProps, mapDispatchToProps)(
   SomeComponent)
```

Komponenta sama o sobě nemá přístup k žádným atributům stavu. Atribut ”title” a akce ”onClick” jsou jí předány pouze jako parametry. Ty jsou k ní připojeny dekorátorem connect, kterému je potřeba definovat dva objekty. MapStateToProps, má velice intuitivní název. Namapuje stav celé aplikace na parametry specifické kom-

ponenty. Je také vidět, že může být zvoleno, které části stavu jí budou zpřístupněny. Může se vybrat i specifický atribut i pod jakým jménem bude přístupný. K ostatním vlastnostem nebude mít komponenta přístup. Druhá funkce mapDispatchToProps není povinná a je nutné ji definovat pouze tehdy, má-li komponenta mít schopnost vyvolat nějaké akce ovlivňující stav aplikace. Jednotlivé akce jsou opět namapovány na klíče odpovídající názvům parametrů komponenty. Oproti proměnným jsou navíc pomocí funkce bindActionCreators zabaleny do funkce dispatch. Ta se postará o dodatečnou režii, ale hlavně o to, že označí funkce jako signály pro reducery.

Nyní se nabízí otázka, jaké komponenty připojit? Pokud by byla připojena pouze jedna, hierarchicky nejvyšší komponenta, použití Reduxu by nedávalo smysl. Nastala by stejná situace, jako při použití stavu komponenty. Pokud by se naopak připojila úplně každá komponenta, zbytečně by se zvyšovala jejich složitost a ne každá by k datům přistupovala. Odpověď je tedy někde mezi těmito extrémy. Pokud má komponenta pár potomků a každý z nich přistupuje k malému počtu dat ze stavu aplikace, je asi lepší připojit jejich rodiče. Pokud by však měla komponenta třeba stovku potomků a každý z nich by vyžadoval jiná data ze storu, nemá cenu vše připojovat k jedné komponentě a následně rozdělovat mezi ostatní (Shinde, 2017). Dalším faktorem je také rychlost překreslování. Každá změnu stavu vyvolá překreslení připojené komponenty, které nějakým způsobem interpretuje změněná data. Pokud například existuje seznam komplexních komponent, které mohou nějakým způsobem změnit svůj vlastní stav, ponechání veškeré režie na rodičovské komponentě by znamenalo negativní dopad na rychlost vykreslování. Protože by všechna data byla uchovávání v jedné připojené komponentě, jakákoliv změna i zanedbatelně malé položky stavu, by znamenala její překreslení a následné překreslení všech komponent seznamu. Bude-li však každá komponenta připojena individuálně, změna vlastního stavu vyvolá překreslení pouze sebe sama, čímž dojde k podstatnému zvýšení rychlosti.

Důkladná správa dat na klientském prohlížeči je čím dál tím více důležitá. S přesunem podstatné části logiky webových aplikací na klientské zařízení se zvyšuje jejich složitost a bez řádné organizace může dojít ke zbytečným chybám a zhoršení kvality, jak zdrojového kódu, tak aplikace jako takové. Implementací tohoto způsobu uchovávání dat se podařilo oddělit data od komponent a vylepšit tak celkově kvalitu všech modulů.

4.6.4 Styled components

V předchozí části práce byly zmíněny CSS preprocesory. Jelikož uživatelské rozhraní vytvořené pomocí knihovny React je také stylované pomocí CSS, ihned se nabízí otázka použití preprocesoru LESS. Existuje však ještě jedna alternativa, která posouvá kontrolu nad vzhledem uživatelského rozhraní na úroveň programovacího jazyka. Díky tomu, že je používaný JavaScript standardu ES6, je možné využít nové syntaxe literálů nazývanou ”template string” (Orendorff, 2015). Jedná se o nový způsob zapisování textových řetězců s možností vkládání proměnných.

Tyto literály jsou využívány právě pro vytváření komponent pomocí knihovny Styled Components. Hlavním rozdílem mezi klasickým přístupem používajícím CSS a komponentami vytvářenými pomocí zmíněné knihovny je fakt, že již nejsou stylovány HTML elementy nebo komponenty založené na CSS selektorech. Jsou vytvářeny přímo nové React komponenty. Každá komponenta má vlastní zapouzdřené styly, které jsou pro ni unikátní a nelze je sdílet mezi sebou (s výjimkou dědičnosti) (Greif, 2017). Vytvoření komponenty je velice jednoduché a podobné jako definice CSS třídy:

```
1  import styled from 'styled-components';
2  import grey from 'material-ui/colors/grey';
3
4  const BaseHeading = styled.h2`
5    font-weight: 300;
6    border-bottom: 1px solid;
7    border-color: ${props => props.boderColor || 'red'}
8    border: ${props => props.withouthBorder ? 'none' : ''};
9    color: ${grey[800]};
10 `;
```

Knihovna má předdefinované všechny HTML elementy a těm jsou přiřazovány styly pomocí template string. Stejně jako u preprocesorů je možné používat proměnné a funkce. Co však posouvá tento způsob stylování uživatelského rozhraní o kus dále, je zavedení dynamický proměnných, neboli props, které umožní dynamicky měnit vzhled komponent. Co by za jiných okolností muselo být řešeno výměnou CSS třídy nebo definováním inline stylů, je možné realizovat předáváním parametrů přímo komponentám. Ty si všechny zmíněné funkce obstarají samostatně. Použití takové komponenty v kódu je identické jako použití HTML elementů. Jen místo klasické HTML značky bude použit název komponenty.

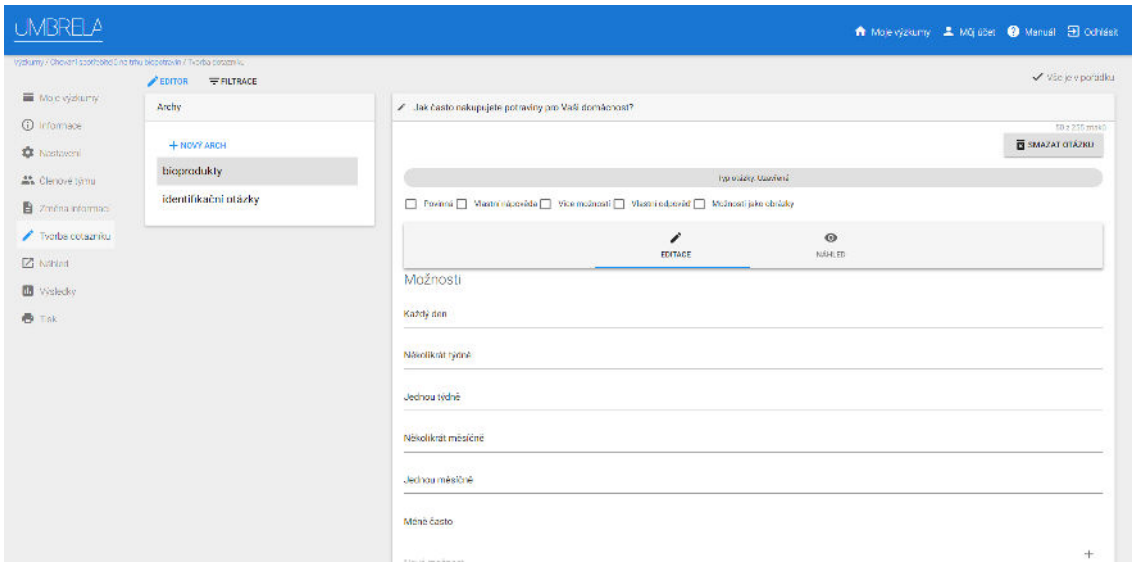
Existuje argument, že je těžké se takové konvenci přizpůsobit, protože místo zavedených HTML značek se používají vlastní definované značky. Takový argument však nemá příliš dobré základy, protože při používání knihovny React vývojář používá z velké části názvy klasických komponent a i kdyby byla dána přednost tradičnímu přístupu konstrukce uživatelského rozhraní, musely by si pamatovat minimálně CSS třídy. Stylované komponenty opět povedou k ušetření podstatné části zdrojového kódu a zjednoduší vývoj dynamicky se měnícího vzhledu uživatelského rozhraní vyvíjených modulů.

4.6.5 Editor dotazníků

První verze editoru byly vytvořena pomocí knihovny JQuery. Fungovala sice bezchybně, ale byla neintuitivní a bylo zapotřebí důkladné vysvětlení každé možné akce, kterou šlo s editorem provést. Druhá verze tyto nedostatky téměř odstranila, ale díky velkému množství chyb v implementaci měla spousta uživatelů potíže dokončit práci. Bylo tedy zapotřebí vytvoření intuitivního designu a bezchybné implementace.

Stejně jako u zbytku uživatelského rozhraní, bylo využito směrnice Material design od korporace Google. Jejich dodržením vznikl editor, který je srozumitelný i pro nové uživatele, avšak umožňuje provádět všechny operace jako v předchozích iteracích.

Většina chyb předchozí verze editoru vznikala ze skutečnosti, že data se nesynchronizovala průběžně, ale na pokyn uživatel se dávkově ukládala na server. Vzhledem k nedostatečné validaci vstupů pak byla zadávána chybná data a nebylo možné dotazníky ukládat a uživatelé přicházeli o svoji práci. Už jen koncept dávkového ukládání struktury dotazníku je v tomto případě chybný. Není možné spoléhat na bezchybnou práci s editorem a také na to, že uživatel třeba nezapomene dotazník uložit. Proto byl navržen velice jednoduchý systém automatického ukládání průběhu práce. Každá individuální změna je ihned synchronizována s databází, a tím pádem není možné přijít o několika hodinové úsilí. Ukládání probíhá pomocí dříve popsaných asynchronních akcí. Mohla by existovat námitka, že komunikace se serverem bude pomalá a zhorší se tak user experience. Tato skutečnost však nenastala. Velikost požadavků málokdy přesáhla velikost desítek kilobajtů. Ve většině případů se pohybovala do jednotek kilobajtů, což je v době dostupného vysokorychlostního internetového připojení zanedbatelné. Doplnění striktní validace vstupů a dostatečně viditelné, ale ne invazivním způsob upozornění uživatelů na chyby, navíc odstranilo nepochopení požadavků pro různé akce editoru.

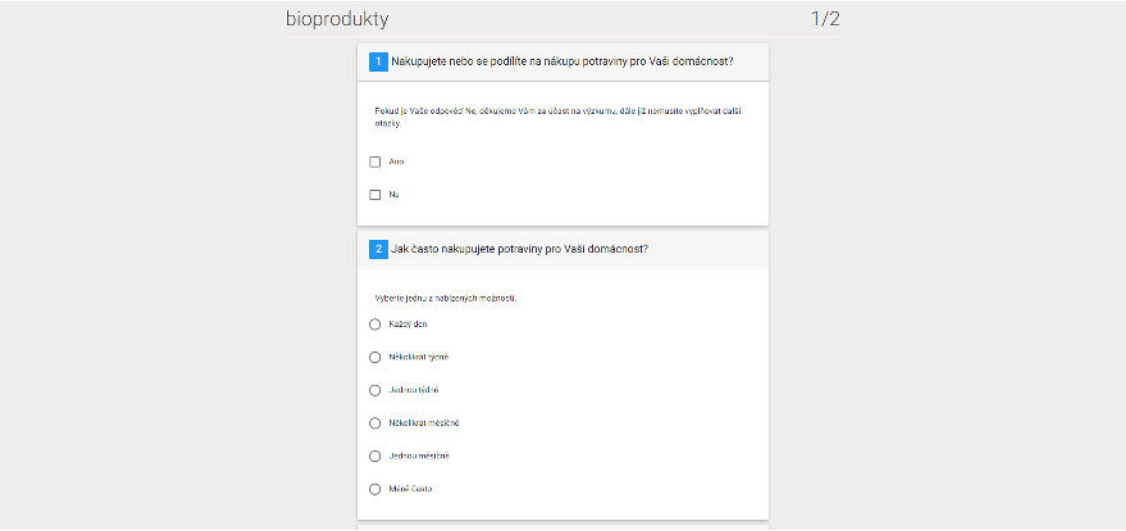


Obrázek 14: Editor dotazníků

4.6.6 Vyplňování dotazníků

Modul pro vyplňování dotazníku je v nové verzi také realizován pomocí knihovny React. V minulosti se jednalo o klasický formulář obohacený JavaScriptem. Vzhledem k tomu, že dotazníky jsou skoro ve všech případech rozděleny minimálně na

dva archy a je nutná jejich okamžitá validace a také zachovávání stavu dotazníku v paměti (dříve se ukládal do session), je tento modul velice vhodnou částí právě pro kombinaci knihoven React a Redux.

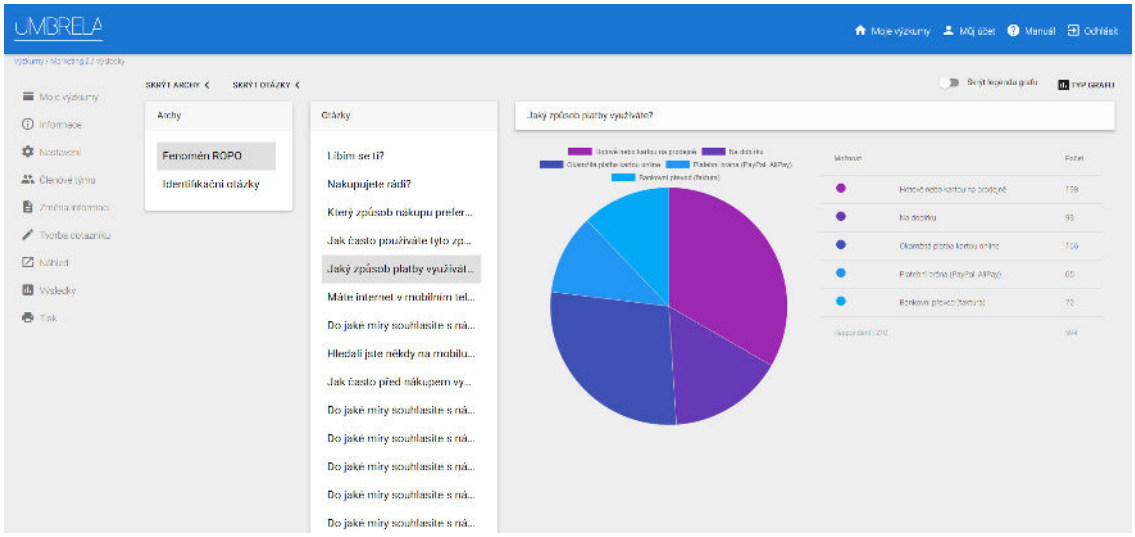


Obrázek 15: Vyplňování dotazníku

4.6.7 Přehled dat

Vizualizace průběžných výsledků výzkumů byla nejnovější část celé aplikace a jedna z mála, která neměla žádné viditelné funkční nedostatky. I na tomto místě lze však najít místo pro zlepšení. Kromě starší verze knihovny a samozřejmě rozdílného designu, byl problém výkon.

V případě výzkumů, které měly tisíce respondentů, vznikal problém s dobou načítání zobrazovaných dat. Data byla nahrávána v jednom celku, takže časová složitost rostla nejen s počtem respondentů, ale i s komplexitou samotné struktury dotazníku. Proto se přistoupilo k nahrávání dat každé otázky individuálně pomocí dříve popsaných asynchronních akcí. Data jsou požadována až v okamžiku jejich použití a zůstávají uložena v klientském prohlížeči. Tento krok přinesl u velkých dotazníků značnou úsporu času a dále přispěl k vylepšení user experience.



Obrázek 16: Přehled výsledků

4.6.8 Testování

Testování frontend modulů se liší testování serverové části. Odpadá nutnost testování databázových manažerů, není třeba testovat požadavky na server a testovací prostředí je mnohem jednodušší. Na druhou stranu je nutno otestovat správnost mutace stavu, asynchronní získávání dat, a protože uživatelské rozhraní je v mnoha případech mnohem komplexnější, musí být otestováno i správné renderování jednotlivých komponent.

K testování renderování komponent uživatelského rozhraní knihovny React byl vyvinut velmi intuitivní nástroj, takzvaný snapshot testing. Ten je součástí testovací knihovny vyvinuté společností Facebook, určené přímo pro React. Knihovna se nazývá Jest. Snapshot test komponety funguje tímto způsobem. Komponenta musí být v testu vyrendrována do virtuálního DOM, a knihovna si z ní vytvoří snímek (snapshot) a uloží jej. Tím se vytvoří jakási šablona, se kterou je následně při dalších testech komponenta porovnávána. Pokud snímky jedné komponenty ve dvou testech neodpovídají, je jasné, že se komponenta nějakým způsobem změnila (Pierotti, 2017). Tento proces pomáhá zachycovat nechtěné změny a špatné renderování komponenty. Samozřejmě, že pokud jsou změny provedeny úmyslně, obraz komponenty lze jednoduše aktualizovat. Tento způsob testování je vhodný zejména pro dynamicky měnící se uživatelské rozhraní a zajistí, že při stejných vstupních datech bude mít výstupní komponenta stejnou podobu. Příklad takového testu může být následující:

```
import React from 'react'
import {DataReviewModuleWrapper} from '../dataReviewModule';
import {shallow} from 'enzyme';
import toJson from 'enzyme-to-json';
import configureStore from 'redux-mock-store';

describe('Data review module wrapper mounting component', ()
   => {
  ...
  const mockStore = configureStore();
  it('should render correctly', () => {
    const store = mockStore({})
    const tree = shallow(
      <DataReviewModuleWrapper store={store}/>
    )
    expect(toJson(tree)).toMatchSnapshot();
  })
  ...
});
```

Další důležitou částí testování je stav aplikace. Protože je používáno knihovny Redux a stav je jeden velký objekt, který je měněn pomocí jednoduchých funkcí (reducer), stačí testovat volání těchto funkcí a porovnávat skutečný stav s očekávaným. Samotný test je velice jednoduchý a může mít následující podobu:

```
 it('should remove question from state on close multi answer',
     () => {
    const questionId = 1;
    const initialState = {
      "1": {
        options: [1,]
      }
    }
    const expectedState = {}
    const payload = {
      optionId: 1,
      questionId,
      checked: false
    }
    expect(reducer(initialState, {
        type: types.CLOSE_MULTI_ANSWER,
        payload
      })).toEqual(expectedState)
  })
```

Testování akcí reduxu je bezpochyby stejně důležité. Většina akcí jsou jednoduché funkce, které vrací nějaké objekty. Stačí jim předat vstup a kontrolovat jestli jejich výstup bude stejný jako očekávaný. Testy jsou velice podobné, jako při testování stavu aplikace. Co je však už o něco složitější, je testování asynchronních akcí. Protože posílají požadavky na server, je nutné tento server nějakým způsobem simulovat. To lze vyřešit pomocí mock knihoven, které simulují server. U těchto funkcí je také běžné řetězení dalších akcí. Proto je potřeba také tyto akce sledovat a kontrolovat jejich korektní volání. Na příkladu bude ukázána událost změny typu otázky, která z nějakých důvodů nemohla být serverem provedena:

```
1  it('should fail to change question type', () => {
2     const questionId = 1;
3     const sheetId = 1;
4     const questionType = 'CloseQuestion';
5     const question = {
6       questionId,
7       title: 'title',
8     }
9     let store = mockStore({
10      questions: {activeQuestion: question},
11      editor: {
12        activeSheet: {
13          sheetId,
14        }
15      }
16    });
17    const expectedActions = [{
18      type: types.CHANGE_QUESTION_TYPE,
19      questionType
20    }, {
21      type: types.REQUEST_QUESTION
22    }, {
23      type: types.QUESTION_FETCH_FAILED,
24    }, {
25      type: types.FINISH_FETCH
26    }]
27    fetchMock.postOnce(`${window.base}synchronize-question`, {
        status: 500});
28    return store.dispatch(actions.changeQuestionType(
        questionType))
29    .then(() => {
30      expect(store.getActions()).toEqual(expectedActions);
31    });
32  })
```

V průběhu implementace se ukázalo, že v mnoha ohledech je testování uživa-

telského rozhraní mnohem složitější než-li testování serverové části aplikace. Je sice pravda, že konfigurace prostředí je mnohem jednodušší a nemusí se například vytvářet žádná databáze, ale uživatelské rozhraní je mnohem náchylnější k chybám, protože je nutné počítat s mnohem více scénáři. To potvrzuje i samotný počet testů. Tyto moduly mají dohromady zhruba 600 různých testů, což je asi dvojnásobek oproti serverové části. Pokud se přihlédne k tomu, že jen část uživatelského rozhraní je implementována v jazyce JavaScript a na složitost implementace, tak se potvrdí, že v dnešní době je vytvoření uživatelského rozhraní progresivních webových aplikací mnohem náročnější, než se může na první pohled zdát.

4.6.9 Continuous integration

Stejně jako server, tak i uživatelské rozhraní může benefinovat ze zavedení Continuouse integration. Simulace produkčního prostředí je oproti serveru triviální. Jediné, co je potřeba, je mít k dispozici Node.js na virtuálním stroji a testování může bez problému začít. V minulém případě bylo použito bitbucket pipelines, nicméně časové omezení a doba průběhu testů i jejich frekvence se ukázaly být překážkou v používání této alternativy. Existuje však i jiná možnost. Tou je služba TravisCI, které je možné používat na repositářích umístěných na portálu GitHub. TravisCI pro veřejné repositáře má neomezenou dobu testování a nebyly nalezeny žádné nedostatky oproti konkurentovi. Problémem však je nutnost zveřejnění zdrojového kódu, nebo zaplacení privátního repositáře. Vzhledem k neochotě financovat další náklady na vývoj, byl vytvořen repositář pro otestování této alternativy pouze pro JavaScriptové moduly uživatelského rozhraní.

Konfigurace prostředí je velice podobná, využívá stejného typu souborů a lze využívat stejných konzolových příkazů jako při vývoji softwaru:

```
1  language: node_js
2  node_js:
3    - 8
4  install:
5    - npm install -g codecov
6  cache: yarn
7  before_script:
8    - yarn
9  script:
10   - yarn test
11   - yarn build
12   - codecov
13 os:
14   - linux
```

4.7 Organizace projektu

Změny technologií a implementace nové verze přispěly k odstranění chyb, inovaci aplikace a pokusu o zavedení nového standardu zdrojového kódu a ulehčení budoucích změn. Bohužel, při zachování aktuálního způsobu vedení projektu, se může velice rychle stát, že veškerá snaha bude k ničemu a aplikace se dostane do stejné spirály, jako její předchozí verze a během pár let bude ve stejném stavu, jako byla na začátku této práce. V průběhu implementace začínalo být jasné, že kromě revize zdrojového kódu, bude nutné zajistit i revizi vedení a organizace celého projektu Umbrela.

Nekonzistentní jednočlenný tým spadající pod vedení, které nemá žádné zkušenosti s vývojem softwaru, je výborný recept na neúspěch. Po vysvětlení aktuální situace a podnětu k jejímu řešení právě vedením ústavu Marketingu a obchodu, se začalo s přípravou převedení správy celého projektu na ústav Informatiky na Provozně ekonomické fakultě Mendelovy univerzity v Brně. Ten by měl nabídnout lepší zázemí pro celý projekt. Má také mnohem lepší podmínky pro vyhledání nových a kompetentních vývojářů, kteří mohou na projektu dále pokračovat. Také by měla být vytvořena pozice supervizora, který bude dohlížet na celý projekt, na dodržování architektury, dodržování konvencí a dalších aspektů projektu. Tento supervizor by měl také pomáhat se zaškolením nových členů týmu, měl by mít dobrou znalost celé aplikace, a zajistit tak chybějící kontinuitu předávání znalostí. Dalším krokem je také převedení zdrojového kódu do privátního systému GitLab (distribuce softwaru git) ústavu. Na něm již neexistují žádná omezení pro continuous integration a otevírá také budoucnost pro zavedení continuous delivery, budou-li tomu vyhovovat podmínky projektu.

5 Diskuse

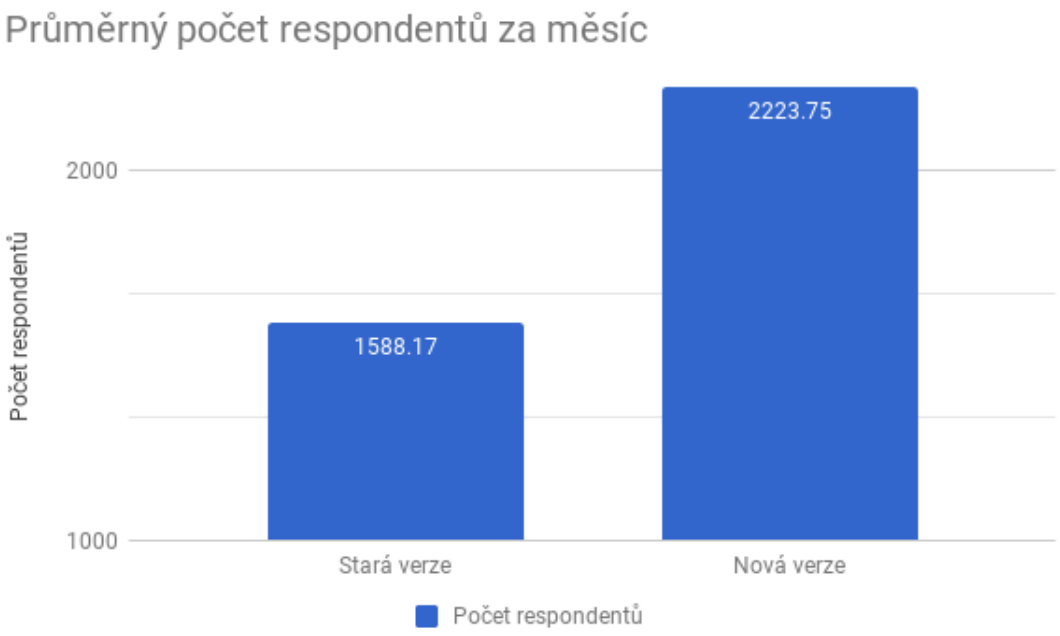
Na začátku této práce byl zadán zdánlivě jednoduchý úkol implementace nové verze aplikace. Postupem času začalo být zjevné, že tomu tak není. Jádrem problému aplikace nebyl pouze starý a nekvalitní kód, ale také prostředí, ve kterém se nacházela. Také velikost celého projektu byla zadavateli hrubě podceněna. Aplikace za svoji dobu existence nepochybně rostla a měnila se. Spravovat jí jedním člověkem je dlouhodobě nereálné. Spíše než odborné znalosti vývojáře, je testována jeho osobnost. Bylo zapotřebí vynaložit obrovské množství úsilí a práce, aby byl projekt včas hotový a měl odpovídající kvalitu. Neustálé změny požadavků a návrhy dalšího rozšíření ztěžovaly vývoj a několikrát se musely různé části aplikace předělávat.

Největším problémem celého průběhu práce na projektu je právě velikost týmu. Vývojář není nikým kontrolován, není se s kým poradit ani nikdo nemůže zpochybnit postup vývoje. Tyto skutečnosti svádí k nedodržování konvencí, nedostatečnému testování kódu, můžou způsobit nepochopení či špatnou interpretaci zadání. Pohled více osob na jeden problém může také někdy přinést unikátní řešení. Veškerá zodpovědnost je přenesena na jednu osobu a to může vyvolat i velkou psychickou zátěž. S ohledem na tyto a další zkušenosti z vývoje na projektu není divu že v minulosti většina studentů na projektu nevydržela pracovat déle než pár měsíců a potom museli projekt opustit.

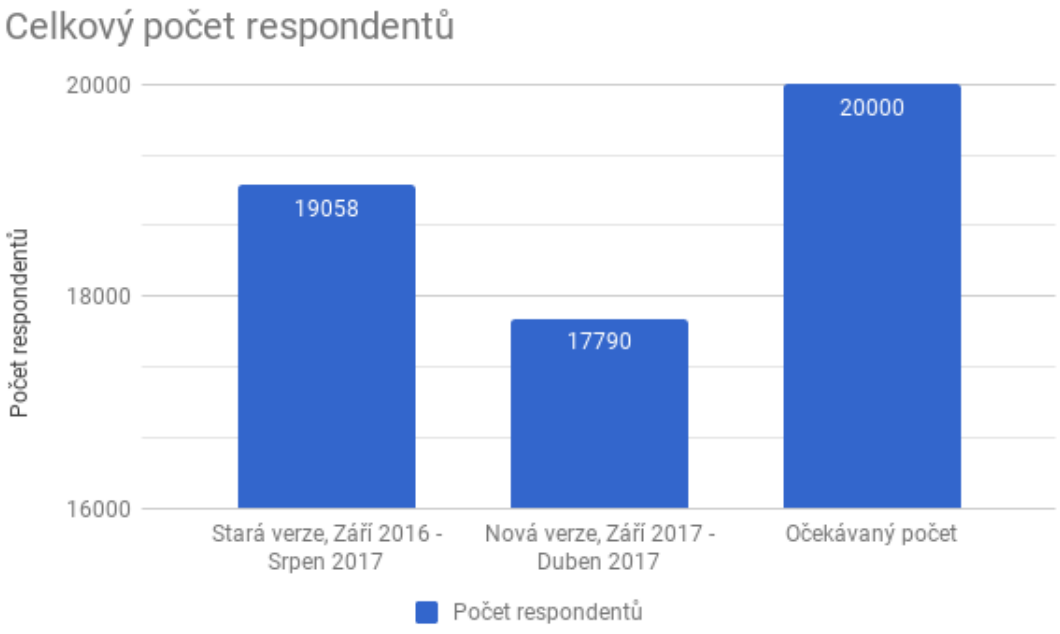
Na druhou stranu tato izolace však vytváří velice unikátní situaci. Vývojář je nucen k rozšíření svých znalostí do relativně širokého spektra. Od konfigurace severu, přes implementaci všech částí aplikace, až i k určitému managementu celého projektu. Získání takových zkušeností je k nezaplacení a jsou bez pochyb velice cenné k dalšímu profesnímu růstu. Mohou jedinci pomoci se posunout nad meze "řadového" programátora a připravit jej na pozici, která vyžaduje schopnost abstrakce problému a vymyslet dobré řešení, které může být za hranicemi myšlení vývojáře zaměřeného pouze na jednu oblast.

Všechny nabyté znalosti v průběhu práce byly využity k vytvoření stabilnějšího a nejen lepšího softwaru, ale i celého projektu. I když se může obyčejnému uživateli zdát, že kromě nového designu uživatelského rozhraní a pár upravených a nových funkcí aplikace, se vlastně nic nezměnilo, pod povrchem nalezneme kompletně odlišný kus programu. Každá zvolená technologie a konvence byla pečlivě vybrána za účelem zjednodušení celého projektu. Byl kladen důraz na to, že o aplikaci se budou v budoucnu starat zejména studenti a bylo pro ně připraveno prostředí, se kterým budou schopni pracovat mnohem lépe, než v minulých verzích systému.

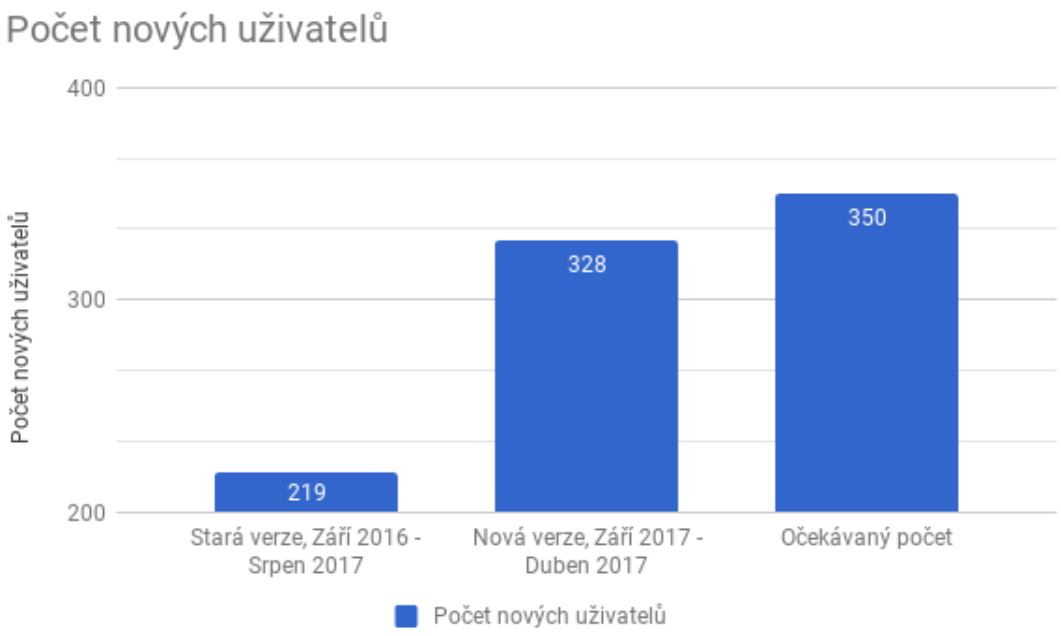
Také z uživatelského hlediska se zdá být reengineering úspěšný. Od nasazení první beta verze v září roku 2017 se popularita aplikace dramatickým způsobem nezhoršila a dá se očekávat i zvýšení frekvence využívání softwaru.



Obrázek 17: Porovnání průměrného počtu respondentů



Obrázek 18: Porovnání celkového počtu respondentů



Obrázek 19: Porovnání celkového počtu nových uživatelů

Samozřejmě, že v průběhu se vyskytlo několik chyb a nedostatků. Zejména v první verzi, která byla nasazená na začátku akademického roku 2017. Vzhledem k časovému rámci, ve kterém musela být tato verze hotová, je to po poohlédnutí se zpět za celým procesem pochopitelné. Během tří měsíců musela být připravena aplikace, na které probíhala práce bezmála celý rok. Tyto chyby však nebyly stejně závažné jako v původní verzi a postupně byly odstraněny. Konečný stav aplikace je více než uspokojivý a od začátku března roku 2018 nebyly uživateli nahlášeny žádné chyby. Komunikace s uživateli má nyní spíše charakter technické podpory, než-li řešení nefunkčních částí aplikace.

Rozhodnutí o omezení podpory webových prohlížečů pouze na uvedenou trojici se také prokázalo jako správné. Nebyly zaznamenány žádné stížnosti, jak uživatelů, tak respondentů o nefunkčnosti v jejich prohlížecích.

Další problém, vyplňování dotazníku na mobilních zařízeních, nebyl vyřešen hned při nasazení. V tomto případě bylo zaznamenáno několik případů, kdy tato funkce chyběla. Vzhledem ke stále se zvyšujícímu využívání internetu právě na mobilních zařízeních, měla tato skutečnost vysokou prioritu, bohužel se ji nepodařilo realizovat ihned a pravděpodobně tak došlo ke ztrátě nezanedbatelného množství respondentů prvních výzkumů zveřejněných na nové verzi aplikace. Optimalizace pro menší obrazovky však přišla velice rychle a měla pozitivní ohlasy.

Uživatelé také pozitivně reagovali na přidání překladu aplikace do anglického jazyka. Vzhledem k zapojení univerzity do mezinárodních studijních programů, byla tato možnost již dlouhodobě požadována, avšak architektura předchozích verzí nedo-

volovala její jednoduchou integraci. Aktuální systém překladů je vystaven takovým způsobem, aby jedinou překážkou před přidáním překladu libovolného jazyka bylo samotné přeložení textů v aplikaci.

Nejdůležitějším krokem pro přežití aplikace je však připravované převedení jeho správy na Ústav informatiky. Je to sice subjektivní názor, nicméně je podložen dlouhodobou degradací předchozí verze aplikace. Žádný z problémů, který dostal systém do stavu, kdy bylo nutné jej od základů předělat, by nenastal, pokud by existoval tým vývojářů a kvalitní dozor nad celým projektem. Špatný kód se dá psát v každém programovacím jazyce. Existují desítky let staré systémy, které jsou kvalitní a soustavným refaktoringem se udržovaly v dobrém stavu a neustále inovovaly používané technologie. Připravované převedení by mělo vytvořit lepší zázemí a hlavně osobu, která bude na aplikaci dlouhodobě dohlížet a měla by zamezit postupnému zhoršování kvality kódu.

Financování projektu tohoto charakteru není vůbec jednoduché. Už se nejedná o klasický software, který je vyvíjen určitý časový interval a po něm prohlášen za hotový. Aplikace Umbrela má spíše vlastnosti služby, tak jako spousta moderních webových aplikací. Vývoj na takovém softwaru nikdy nekončí a je samozřejmě finančně náročnější. V kombinaci s fakty, že jeho používání není žádným způsobem zpoplatněno a je financován čistě z prostředků univerzity, není divu, že je problém sehnat kvalitní tým vývojářů. Jediná motivace pro studenty je možnost vypracování bakalářských či diplomových prací. Ale i tato možnost je dnes nabízena firmami, kterým v mnoha případech nemůže univerzita konkurovat s platovým ohodnocením. Komercializace produktu by měla být jedním z hlavním cílů univerzity. Podle slov vedení ústavu marketingu a obchodu, i jiné univerzity vyslovily zájem o používání aplikace Umbrela ke studijním účelům. I jiné subjekty údajně měly zájem o tyto služby. Přesun do této sféry se však nikdy neuskutečnil, i přestože evidentně existuje poptávka. Projekt byl navíc také financován pouze ústavem marketingu a obchodu i přes to, že je využíván všemi fakultami univerzity. Finanční zapojení i ostatních částí univerzity využívající aplikaci by mohlo také vést k vytvoření lepších podmínek pro další práci.

První krok pro lepší budoucnost systému Umbrela byl úspěšně dokončen. Teď záleží pouze na vedení, zdali bude mít zájem a ochotu o zavedení dalších opatření ke dlouhodobému zlepšování celého projektu. Zánik Umbrely by znamenal značné komplikace pro studenty Mendelovy univerzity v Brně a museli by často financovat jak semestrální projekty, tak i závěrečné práce z vlastních zdrojů. Přežití projektu je v zájmu všech stran a postupnou prací by se z něj mohl stát konkurent službám jako je Google Forms nebo Survio.

6 Závěr

Stanoveným cílem práce byl reengineering aplikace Umbrela. Aplikace se potýkala s mnoha problémy, které vedly k rozhodnutí o vytvoření její nové verze. Aplikace byla analyzována a byla zjištěna řada nedostatků na všech úrovních, jejichž kombinace měla za následek rozhodnutí, že bude přerušena jakákoliv závislost na původní implementaci. Některé funkce byly také modifikovány nebo přidány. Snaha byla také zaměřena na nepoužívané části aplikace a byly prozkoumány možnosti jejich odstranění.

Technologie byly vybírány na základě jejich užitečnosti, ale také na jejich rozšířenosti a byly také přizpůsobeny potenciálním zkušenostem budoucích vývojářů. Byla také adoptována klasická MVC/MVP architektura pro stavbu nejen webových aplikací. Při implementaci byl kladen důraz na dodržování stanovených konvencí. Zdrojový kód byl dokumentován a průběžně doplňován unit testy. Všechny tyto kroky vedly k vytvoření funkční a stabilní nové verze aplikace Umbrela, jejíž první verze byla nasazena v září 2017. Následoval důkladný refaktoring, odstraňování chyb a další zlepšování celé aplikace.

Také bylo vyjednáno převedení aplikace na Ústav informatiky a vytvoření pozice supervizora. To by mělo přinést zlepšení organizace celého projektu a zabránit zhoršování kvality kódu. Byla také vytvořena jak dokumentace zdrojového kódu, tak i vývojářská příručka, která by měla provést nové vývojáře nastavením vývojového prostředí, seznámením s aplikací a pravidly které by měly být dodržovány během vývoje.

Aplikace se může pyšnit hojným užíváním a očekává se, že bude i nadále využívána nejen studenty Mendelovy univerzity v Brně.

7 Reference

Aaker, D. A. *Marketing research: international student version*. 11. Hoboken, c2013. Irwin/McGraw-Hill series in marketing. ISBN 978-1-118-32181-2.

Abdalhamid, S., A. Mishra *Adopting of Agile methods in Software Development Organizations: Systematic Mapping*. TEM Journal [online]. 2017, 6(4), 817-825 [cit. 2018-02-25]. DOI: 10.18421/TEM64-22. ISSN 22178309.

Abramov, D. a A. Clark *Redux documentation*. Redux [online]. 2015, 2015 [cit. 2018-04-15]. Dostupné z: https://redux.js.org/.

Alam, A. a T. Padenga *Application software reengineering*. Delhi: Pearson, 2010.
ISBN 978-81-317-3185-7.

Arsenovski, D. *Professional refactoring in C# & ASP.NET*. Indianapolis, IN: Wrox, 2009. ISBN 9780470434529.

Baird, K. C. *Ruby by example: concepts and code*. San Francisco: No Starch Press, c2007. ISBN 978-1-59327-148-0.

Banica, L., M. Radulescu a Hagiu, A. *Towards an Agile Approach in Academic Software Development - A Case Study*. Annals of the University Dunarea de Jos of Galati: Fascicle[online]. 2016, 22(2), 88-95 [cit. 2018-02-25]. ISSN 15832074.

Barcia, R., G. Hambrick, K. Brown, R. Peterson a K. Bhogal *Persistence in the Enterprise: A Guide to Persistence Technologies*. 1. 800 East 96th Street, Indianapolis, Indiana 46240: IBM Press, 2008. ISBN 978-0-13-158756-4.

Barsotti, D. *Table Inheritance with Doctrine*. Liip [online]. Schmiedenplatz 5 CH-3011 Bern: Liip, 2012 [cit. 2018-04-08]. Dostupné z: https://www.liip.ch/en/blog/table-inheritance-with-doctrine.

Beak, A. *Php 7 Zend certification study guide*. 1. New York, NY: Springer Science Business Media, 2017. ISBN 978-1-4842-3245-3.

Bitbucket *Bitbucket Support [online]*. Sydney, NSW, Australia: Atlassian, 2017 [cit. 2018-04-11]..

Bradley, N. *Marketing research: tools & techniques*. 3. Oxford: Oxford University Press, 2013. ISBN 9780199655090.

Brown, M. *A Beginner's Guide to Webpack 2 and Module Bundling*. SitePoint [online]. Melbourne, Australia: SitePoint, 2017, 30. 1. 2017 [cit. 2018-04-14]. Dostupné z: https://www.sitepoint.com/beginners-guide-to-webpack-2-and-

module-bundling/.

Calder, A. *EU GDPR & EU-US Privacy Shield: A Pocket Guide*. 1. United Kingdom, Ely, Cambridgeshire: It Governance Publishing, 2016. ISBN 978-1-84928-872-9..

Can I use [online]. Can I use, 2018 [cit. 2018-05-08]. Dostupné z: https://caniuse.com/.

Capan, T. *Why the Hell Would You Use Node.js*. Medium [online]. San Francisco: A Medium Corporation, 2017, [cit. 2018-03-25]. Dostupné z: https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e.

Cinarli, V. *An Introduction to CSS Pre-Processors: SASS, LESS and Stylus*. HTMLMAG[online]. 2014, 2014 [cit. 2018-04-14]. Dostupné z: https://htmlmag.com/article/an-introduction-to-css-preprocessors-sass-less-stylus.

Claburn, T. *Official: Perl the most hated programming language, say devs*. The Register [online]. New York: Situation Publishing, 2017 [cit. 2018-03-11]. Dostupné z: Official: Perl the most hated programming language, say devs.

Composer *Dependecy management*. online]. Compsoer, 2016 [cit. 2018-04-08]. Dostupné z: https://getcomposer.org/doc/00-intro.md#dependency-management.

Doctrine [online]. 2006 [cit. 2018-04-08]. Dostupné z: https://www.doctrine-project.org/projects/doctrine-orm/en/latest/cookbook/mysql-enums.html.

Dunglas, K. *Persistence in PHP with Doctrine ORM*. Birmingham: Packt Publishing, 2013. ISBN 978-178-2164-111.

Dupertuis, E. *The PHP hate*. Eric Dupertuis[online]. http://edupertuis.net/, 2016 [cit. 2018-03-26]. Dostupné z: http://edupertuis.net/2016/02/25/phphate.html.

Duval, P. M. *Continuous integration: Improving software quality and reducing risk*. 1. India: Dorling Kindersley, 2008. ISBN 978-81-317-2291-6.

Franklin, J. *Context in ReactJS Applications*. JS Playground [online]. JS, 2017, 13. 2. 2017 [cit. 2018-04-15]. Dostupné z: https://javascriptplayground.com/context-in-reactjs-applications/.

Garrett, J. *The elements of user experience: user-centered design for the web and beyond*. 2. Berkeley, CA: New Riders, 2011. ISBN 978-032-1683-687.

GeckoandFly [online]. USA: GeckoandFly, 2018 [cit. 2018-05-08]. Dostupné z: https://www.geckoandfly.com/25913/best-php-7-frameworks/.

GitHub [online]. San Francisco: GitHub, 2007 [cit. 2018-05-08]. Dostupné z: https://github.com/.

Google Trends [online]. Mountain View, California, USA: Google, 2018 [cit. 2018-05-08]..

Greif, S. *A 5-minute Intro to Styled Components*. FreeCodeCamp [online]. Medium Corporation, 22. 1. 2017 [cit. 2018-04-15]. Dostupné z: https://medium.freecodecamp.org/a-5-minute-intro-to-styled-components-41f40eb7cd55.

Grubb, P. a A. T. Armstrong *Software maintenance: concepts and practice*. 2. Singapore [u.a.]: World Scientific, 2003. ISBN 9812384251.

Hadlow, M. *When should i use ORM*. Code rant [online]. 2012 [cit. 2018-04-03]. Dostupné z: http://mikehadlow.blogspot.cz/2012/06/when-should-i-use-orm.html.

Hamil, P. *Unit test frameworks: [a language independent overview]* Beijing [u.a.]: O'Reilly, 2004. ISBN 0596006896..

Hastie, S. a S. Wojewoda *Standish Group 2015 Chaos Report*. InfoQ [online]. InfoQ, [cit. 2018-02-25]. Dostupné z: https://www.infoq.com/articles/standish-chaos-2015.

Hogan, P. *Software as a Service: Definition, History, and Benefits*. Bussiness2community [online]. Philadelphia PA: Bussiness2community, 2017 [cit. 2018-03-03]. Dostupné z: https://www.business2community.com/tech-gadgets/software-service-definition-history-benefits-01842113.

Chapman, S. *This Is What JavaScript Is Used For*. ThoughCo. [online]. New York: ThoughtCo, 2017 [cit. 2018-03-25]. Dostupné z: https://www.thoughtco.com/what-is-javascript-used-for-2037679.

Chemuturi, M. *Mastering software project management: best practices, tools and techniques*. Fort Lauderdale: J. Ross Publishing, 2010. ISBN 978-1-60427-034-1.

Chin, S. a S. Beth *Access control, security, and trust: a logical approach*. 1. Boca Raton: Chapman & Hall, 2011. ISBN 978-158-4888-628.

Jenkov, J. *The DAO Design Pattern*. Jenkov.com [online]. Jenkov Aps, 2014 [cit. 2018-03-11]. Dostupné z: http://tutorials.jenkov.com/java-persistence/dao-design-pattern.html.

Johnson, S. *The Disadvantages of Ruby Programming* Techwalla [online]. Santa Monica, CA 90404: Techwalla, 2016 [cit. 2018-03-26]. Dostupné z: https://www.techwalla.com/articles/the-disadvantages-of-ruby-programming.

Korotya, E. 5 Best JavaScript Frameworks in 2017. Hackernoon [online]. San Francisco: Medium Corporation, 2017 [cit. 2018-05-08]. Dostupné z: https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282.

Krill, P. *A developer's guide to the pros and cons of Python: Devotees talk about what's good (ease of use, IoT potential) and not so good (design issues, performance) about the language*. InfoWorld [online]. Framingham, MA 01701-9208: InfoWorld, 2015, [cit. 2018-03-26]. Dostupné z: https://www.infoworld.com/article/2887974/application-development/a-developer-s-guide-to-the-pro-s-and-con-s-of-python.html.

Kuniavsky, M. *Smart things: ubiquitous computing user experience design*. Amsterdam: Morgan Kaufmann Publisher, 2010. ISBN 9780123748997.

Kunjumohamed, S., H. Sattari, A. Bretet a G. Warin *Spring MVC: Designing Real-World Web Applications*. 1. Birmingham B3 2PB, UK: Packt Publishing, 2016. ISBN 978-1-78712-639-8.

LANDRY, N. *Iterative & agile implentations methodologies in business intelligence software development*. 1. United States of America, North Carolina: LuLu Enterprises, 2011. ISBN 978-0-557-24758-5.

Leach, R. J. *Introduction to software engineering*. 2. Boca Raton: CRC Press, Taylor & Francis Group, 2016. ISBN 978-1498705271.

Marcotte, E. *Responsive Web Design*. 2. New York: A Book Apart, 2014. ISBN 978-1937557188.

Maymala, J. *PostgreSQL for Data Architects*. 1. Birmingham B3 2PB, UK: Packt Publishing, 2015. ISBN 978-1-78328-860-1.

McDaniel, C. D. a R. H. Gates *Marketing research*. 9. Hoboken, 2013. ISBN 978-1-118-11271-7.

Melvin, B. *Software as a Service inflection point: using Cloud computing to achieve business agility*. New York: iUniverse, 2009. ISBN 978-1440141966..

Mens, T. a S. Demeyer *Software evolution*. New York: Springer, 2008. ISBN 9783540764397.

Mew, K. *Learning Material Design*. 1. Birmingham B3 2PB, UK: Pack Publishing, 2015. ISBN 978-1-78528-981-1..

Miller B. N. a D. L. Ranum *Computer science: the Python programming language*. Sudbury, Mass.: Jones and Bartlett, c2007. ISBN 978-0-7637-4316-1.

Monterion, F. *Learning Single-page Web Application Development*. 1. Birmingham B3 2PB, UK: Packt Publishing, 2014. ISBN 978-1-78355-209-2..

Myhrvold, C. *The Fall Of Perl, The Web's Most Promising Language*. In: FastCompany [online]. USA: Fast Company &, 2014 [cit. 2018-03-11]. Dostupné z: https://www.fastcompany.com/3026446/the-fall-of-perl-the-webs-most-promising-language.

Nezmar, L. *GDPR: praktický průvodce implementací*. Praha: Grada Publishing, 2017. Právo pro praxi. ISBN 978-80-271-0668-4..

Nittro [online]. Nittro, 2015 [cit. 2018-04-14]. Dostupné z: https://www.nittro.org/.

Orendorff, J. *ES6 In Depth: Template strings*. Mozilla HACKS [online]. Mozilla Corporation, MountainView,California,UnitedStates, 14. 5. 2015 [cit. 2018-04-15]. Dostupné z: https://hacks.mozilla.org/2015/05/es6-in-depth-template-strings-2/.

Pierotti, L. *Snapshot testing React Components with Jest*. Hackernoon [online]. San Francisco: Medium Corporation, 2017[cit. 2018-04-21]. Dostupné z: https://hackernoon.com/snapshot-testing-react-components-with-jest-744a1e980366.

Pressman, R. S. *Software engineering: a practioner's approach*. 6. United States: MCGRAW-HILL COMPANIES (OH), 2005. ISBN 9780073019338.

Prettyman, S. *Learn php 7: object oriented modular programming using HTML5, CSS3, Javascript, XML, JSON, and MYSQL*. 1. New York, NY: Springer Science Business Media, 2015. ISBN 978-1-4842-1729-0.

Rauch, G. *Smashing Node.js: JavaScript everywhere*. Chichester, West Sussex: John Wiley & Sons, 2012. ISBN 978-1-119-96259-5.

Rosenberg, L. H. a L. E. Hyatt *Software Re-engineering*. Software Assurance Technology Center System Reliability and Safety Office Goddard Space Flight Center, 1997, NASA 301-286-7475.

Rossel, S. *Continuous integration, delivery, deployment: Reliable and faster software releases with automating builds, tests, and deployment*. 1.

Birmingham B3 2PB, UK: Pack Publishing, 2017. ISBN 978-1-78728-661-0.

Sasidharan, M. *Should I Or Should I Not Use ORM?*. Medium [online]. San Francisco: A Medium Corporation, 2016 [cit. 2018-04-03]. Dostupné z: https://medium.com/@mithunsasidharan/should-i-or-should-i-not-use-orm-4c3742a639ce.

Sengupta, S., D. Sengupta a R. Titus *Application of Agile Methodologies for Member and Team Role Transformation in Projects*. PM World Journal [online]. 2014, 3(1), 1-16 [cit. 2018-02-25]. ISSN 23304480..

Seun, M. *What are Dependency Managers?*. Medium [online]. San Francisco: A Medium Corporation, 2017 [cit. 2018-04-08]. Dostupné z: https://medium.com/prodsters/what-are-dependency-managers-26d7d907deb8.

Shinde, V. *Why using nested connect(react-redux) components is good?*. Hackernoon[online]. San Francisco: Medium Corporation, 2017, 26. 9. 2017 [cit. 2018-04-15]. Dostupné z: https://hackernoon.com/why-using-nested-connect-react-redux-components-is-good-bd17997b53d2.

Sroka, A. *Which language should I learn to code, Python, Perl or Ruby?*. In: Quora [online]. Quora, 2015 [cit. 2018-03-11]. Dostupné z: https://www.quora.com/Which-language-should-I-learn-to-code-Python-Perl-or-Ruby.

Stachour, P. a D. Collier-Brown. *You Don't Know Jack about Software Maintenance*. Communications of the ACM [online]. 2009, 52(11), 54-58 [cit. 2018-02-25]. DOI: 10.1145/1592761.1592777. ISSN 00010782..

StatCounter [online]. Guinness Enterprise Centre, Taylor's Lane, Dublin 8, Ireland: StatCounter, 2018a [cit. 2018-05-08]. Dostupné z: http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet#monthly-201404-201804.

StatCounter [online]. Guinness Enterprise Centre, Taylor's Lane, Dublin 8, Ireland: StatCounter, 2018b [cit. 2018-05-08]. Dostupné z: http://gs.statcounter.com/browser-market-share.

Tidwell, J. *Designing interfaces*. 2. Sebastopol, CA: O'Reilly, 2011. ISBN 9781449379704.

VOTOČEK, P. *Stažení sandboxu přes Composer*. Planette [online]. 2012 [cit. 2018-04-08]. Dostupné z: https://pla.nette.org/cs/sandbox-composer.

W3Techs [online]. Maria Enzersdorf, Austria: Q-Success, 2018 [cit. 2018-05-08]. Dostupné z: https://w3techs.com/technologies/overview/programming_language/all.

WANG, W. *Reverse Engineering Technology of Reinvention*. Hoboken: CRC Press, 2010. ISBN 9781439806319.

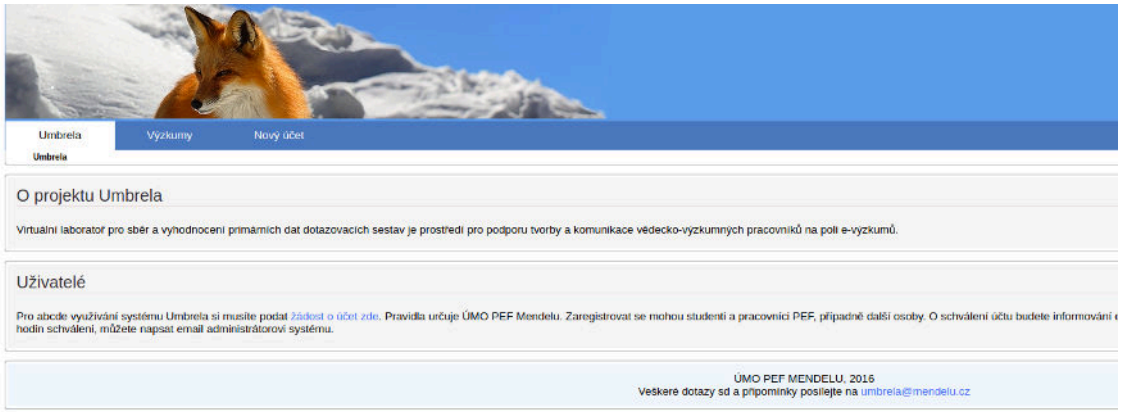
WALTER, S. *The Evolution of MVC*. Stephenwalther.com [online]. STEPHEN WALTHER, 2008 [cit. 2018-03-11]. Dostupné z: http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc.

WHITE, E. *Password security in modern PHP*. IBM[online]. Armonk, North Castle, New York, United States: IBM, 2015 [cit. 2018-04-08]. Dostupné z: https://www.ibm.com/developerworks/library/wa-php-renewed_2/index.html.

ZJR *Is Perl still a useful, viable language?*. In: StackExchange [online]. New York: Stack Exchange, 2012, 2012 [cit. 2018-03-11]. Dostupné z: https://softwareengineering.stackexchange.com/questions/115851/is-perl-still-a-useful-viable-language.

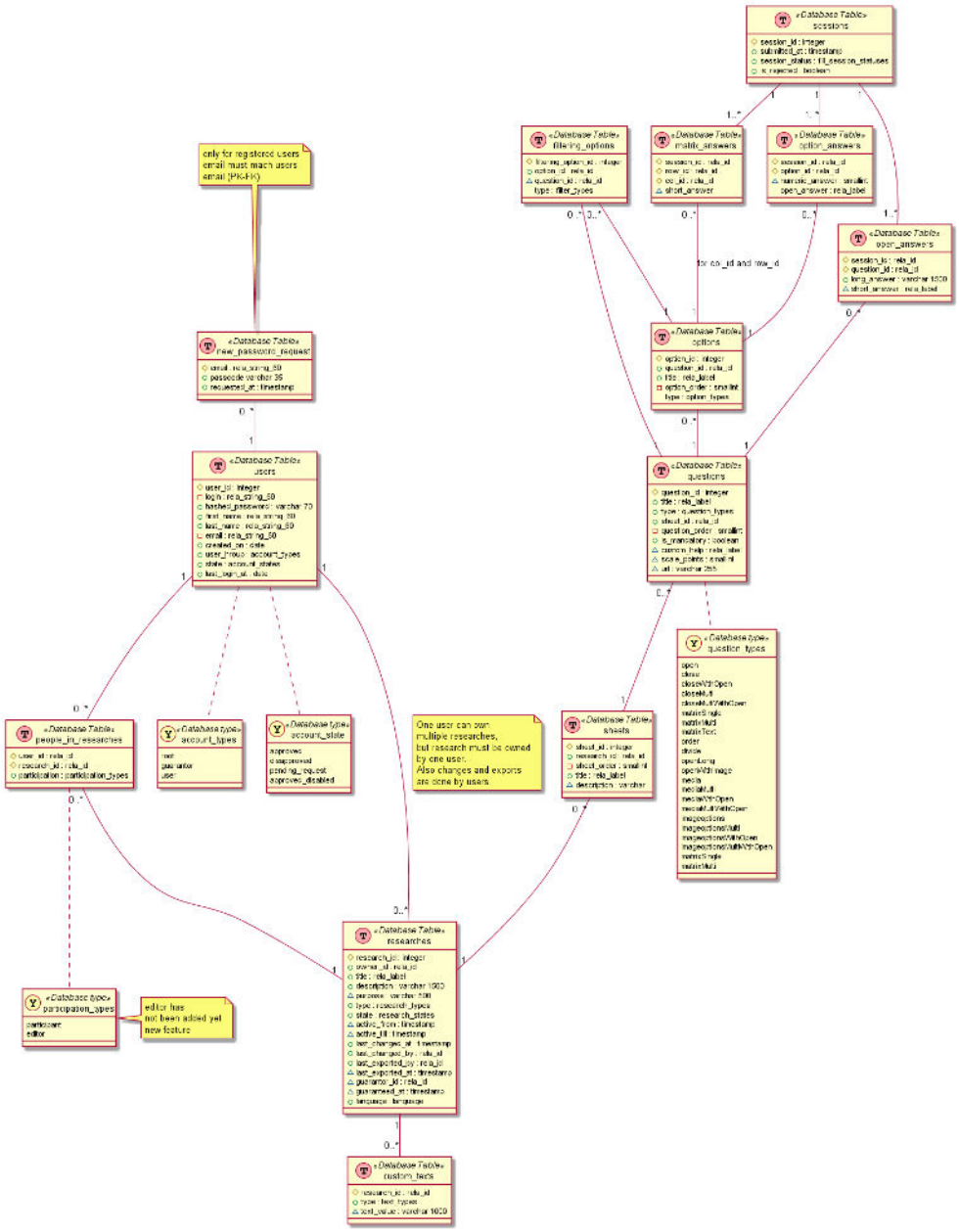
Přílohy

A Původní rozhraní



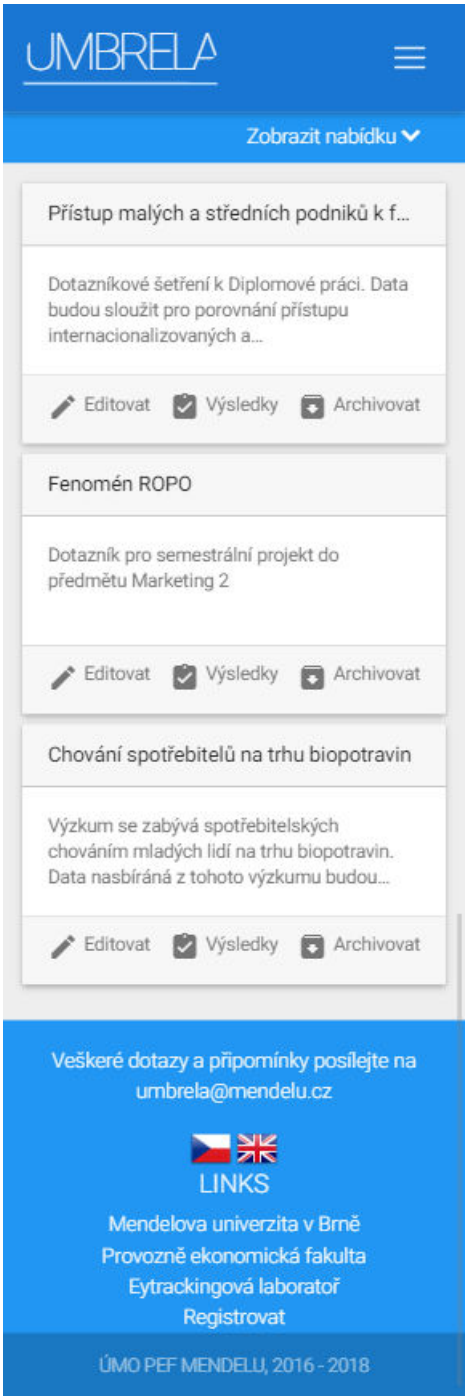
Obrázek 20: Původní uživatelské rozhraní

B ERD diagram



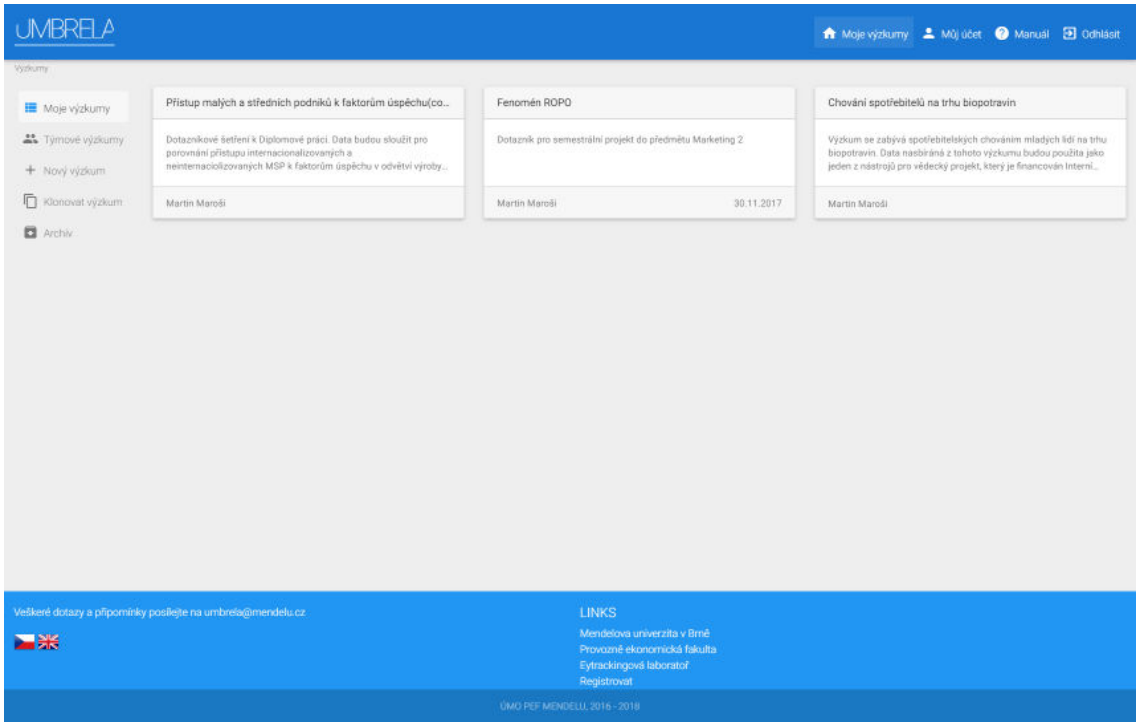
Obrázek 21: ERD diagram

C Mobilní verze uživatelského rozhraní



Obrázek 22: Mobilní verze uživatelského rozhraní

D Desktopová verze uživatelského rozhraní



Obrázek 23: Desktopová verze uživatelského rozhraní