FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Webová aplikace Deployment Manager
Student:	Adam Staš
Vedoucí:	Ing. Oldřich Malec
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem této práce je vytvoření webové aplikace, která umožní uživateli mít přehled o nasazených aplikací u svých klientů a jejich verzích a zjednoduší změnu verze již běžících aplikací. Aplikace bude jednou z komponent Apps Manageru, který se kromě této aplikace skládá také z License Manageru a Builderu.

Postupujte v těchto krocích:
1. Proveďte obecnou rešerši potřeb softwarových inženýrů ohledně distribuování různých verzí svého softwaru různým klientům. Dále se zaměřte na konkrétní potřeby společnosti Jagu s.r.o. a popište je.
2. Na základě rešerše vytvořte vhodný návrh aplikace, která bude figurovat jako prostředník mezi již existující aplikací License Manager a vznikající aplikací Builder, na kterých pracují jiní studenti.
3. Na základě návrhu vytvořte použitelný prototyp.
4. Pro vystavené API poskytněte jeho dokumentaci jako OpenAPI specifikaci.
5. Výsledný prototyp řádně otestujte vhodně zvolenými testy.
6. Zhodnoťte vámi dosažené výsledky i celkový systém Apps Manager a navrhněte možná budoucí vylepšení.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 16. ledna 2023 v Praze.

Bakalářská práce

WEBOVÁ APLIKACE DEPLOYMENT MANAGER

Adam Staš

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Oldřich Malec
10. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Především bych rád poděkoval vedoucímu práce Ing. Oldřichu Malcovi, a to za jeho čas, který mi věnoval, cenné rady a zkušenosti. Dále děkuji Ing. Jiřímu Hunkovi, díky němuž vznikla prvotní myšlenka, z níž se později formovalo téma této práce. Mé poděkování si zaslouží i moji kolegové Alena Ježková a Tomáš Sládek, s nimiž jsem během vývoje spolupracoval. Závěrem poděkuji své rodině a přítelkyni za podporu během studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

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

Abstrakt

Tato bakalářská práce se zabývá procesem tvorby webové aplikace, jejíž účelem je zajištění přehledu o nasazených aplikacích a možnost pohodlné změny verze těchto aplikací. Aplikace je schopna komunikace s aplikacemi License Manager a Builder, s nimiž tato práce souvisí. V práci je provedena analýza požadavků na výslednou aplikaci, návrh technologie, architektury a způsobu uložení dat. Dále práce obsahuje dokumentaci vystaveného API, popis implementace samotné aplikace a její testování. V práci také lze nalézt zhodnocení dosažených výsledků a nastínění možných budoucích vylepšení.

Klíčová slova webová aplikace, návrh webové aplikace, změna verze aplikací, automatizace, PHP, Laravel

Abstract

This bachelor thesis deals with the process of creating a web application, the purpose of which is to provide an overview of the deployed applications and the ability to conveniently change the version of these applications. The application is capable of communicating with the License Manager and Builder applications with which this thesis is related. In this thesis, the requirements for the resulting application are analyzed, and the technology, architecture and data storage method are proposed. Furthermore, the thesis includes documentation of the exposed API, description of the implementation of the application itself and its testing. The thesis also includes an evaluation of the results achieved and an outline of possible future improvements.

Keywords web application, web application design, changing version of applications, automation, PHP, Laravel

Seznam zkratek

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CI	Continuous Integration
CD	Continuous Deployment
CSS	Cascading Style Sheets
DM	Deployment Manager
DRY	Don't Repeat Yourself
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifikátor
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LM	License Manager
MVC	Model-View-Controller
ORM	Object-Relational Mapping
PHP	PHP Hypertext Preprocessor
REST	Representational State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Úvod

Každý softwarový vývojář, který se snaží své produkty zdokonalovat a opravovat vzniklé chyby, si nějakým způsobem verzuje zdrojový kód. K tomu slouží verzovací systémy, které nabízí možnost pohodlně vytvářet různé verze produktu nebo vývojové větve pro nově implementované vlastnosti, jež pak vývojář sloučí do hlavního vývojového toku. Avšak jakmile chce vývojář svůj software nasadit klientům, tyto systémy mu už neumožňují udržovat si přehled o tom, jakému klientovi byla nasazena která verze vyvíjené aplikace či tyto verze přehledně měnit. Vývojové společnosti pak mají tendenci svým klientům nasazovat vždy nejnovější produkční verzi a provádět pravidelné aktualizace, jakmile vydají novou verzi, jinak by se z jejich práce mohla stát nepřehledná katastrofa.

Právě problematikou distribuování odlišných verzí aplikací různým klientům se tato práce zabývá. Její výsledky budou užitečné nejen pro společnost Jagu, s. r. o., tedy jejího zadavatele, ale i pro jiné softwarové společnosti, jež se s touto problematikou setkávají. Těm může tato práce sloužit například jako inspirace, jak lze daný problém pojmout a řešit.

Toto téma jsem si zvolil, neboť mě problematika verzování zaujala a rád bych vývojářům usnadnil udržování přehledu o nasazených aplikacích a jejich verzích.

Výsledkem práce je webová aplikace, v níž bude mít správce nasazování aplikací vývojové společnosti možnost evidence nasazených aplikací, jejich verzí a možnost tyto verze změnit.

Výsledná webová aplikace je součástí systému Apps Manager, jenž je dále tvořen aplikacemi License Manager a Builder. Na těchto aplikacích pracují studenti Tomáš Sládek a Alena Ježková v rámci svých bakalářských prací. Protože spolu naše aplikace komunikují, je nutné, abychom spolu na některých částech spolupracovali.

Práce má vytyčeno několik dílčích cílů. Jako první cíl je provedení rešerše potřeb vývojářů týkajících se distribuce odlišných verzí softwaru různým klientům a konkrétní analýzy potřeb společnosti Jagu, s. r. o. Dále mezi cíle práce patří vytvoření vhodného návrhu aplikace, implementace použitelného prototypu této aplikace a tvorba dokumentace API, které bude během návrhu vytvořeno. Závěrečnými cíli jsou otestování vzniklého prototypu vhodně zvolenými testy, zhodnocení dosažených výsledků a návrh možných budoucích vylepšení.

Práce je rozdělena do několika kapitol. V první kapitole popisuji formy výzkumu, provádím obecnou rešerši na téma distribuce odlišných verzí softwaru různým klientům, věnuji se výběru vhodné metodiky vývoje aplikace a zaměřuji se na potřeby zadavatele.

Na získané poznatky navazuji další kapitolou, v níž provádím návrh výsledné aplikace. Zabývám se výběrem vhodné technologie a architektury. Také v této části navrhuji datový model aplikace a rozhraní pro komunikaci aplikací v systému Apps Manager.

Následuje kapitola týkající se samotné implementace webové aplikace. Popisuji, jaký software zvolím pro vývoj i implementační detaily architektury, již jsem vybral v předešlé kapitole. Mimo ně jsou popsány některé důležité služby, které v aplikaci figurují, i technologie starající se o zabezpečení API.

Po popisu implementace následuje kapitola věnovaná testování vzniklého prototypu. V ní zmiňuji typy testů, které se používají, a zejména ty, jež jsou vhodné pro testování webové aplikace. Dále popíši, jak testuji backend aplikace a zmíním i průběh akceptačního testování.

Poslední kapitola hodnotí dosažené výsledky a navrhuje možná budoucí vylepšení.

Kapitola 1
Analýza

Vývoj softwaru začíná analýzou. Jedná se o činnost, jejíž cílem je získání představy, s jakou problematikou se zákazník potýká, specifikace požadavků, jež budou na výsledný software kladeny a identifikace problémových míst či omezení na daný systém. Analýza dále slouží k vyjasnění si zadání, které nám zákazník poskytl, případně jeho vytvoření během komunikace s tímto klientem. [1]

Obsahem této kapitoly bude nejprve popis forem výzkumu a obecná rešerše problematiky distribuce různých verzí softwaru odlišným klientům. Poté shrnu vývojové metodiky. Také popíšu působiště zadavatele této práce a provedu sběr a analýzu požadavků na software, jenž společně s tímto textem vzniká. Kapitolu zakončím specifikací funkčních a nefunkčních požadavků a popisem případů užití svého softwaru.

1.1 Formy výzkumu

Před začátkem řešení problematiky bylo potřeba provést výzkum, pomocí kterého bych zjistil, s jakými problémy se vývojáři setkávají. Konkrétně mě zajímali vývojáři, kteří aktivně vyvíjí nějaký software, v jehož kódu se neustále dělají změny a současně tito producenti softwaru mají vícero klientů. Klíčovou podmínkou byl ovšem fakt, že ne všichni klienti používají stejnou verzi distribuovaného softwaru.

Když chceme provést kvalitní výzkum, je potřeba si nejprve stanovit, o jaký druh výzkumu se bude jednat. Existují dvě formy výzkumů, které popisují, jakým způsobem lze získávat potřebné informace. Jedná se o formy kvantitativní a kvalitativní.

Při kvantitativním výzkumu sbírá výzkumník data pomocí dotazníků, testů či pozorování. Jedná se tedy, jak i plyne z názvu, o metodiku, která má na svém vstupu početnější skupinu lidí. Tento výzkum není příliš časově náročný, protože stačí připravit dotazník a tento odeslat vhodným respondentům. Nutno podotknout, že takový dotazník nezkoumá danou problematiku do hloubky.

Naproti tomu výzkum kvalitativní se zpravidla provádí osobními rozhovory (není to však podmínkou). Pokládané dotazy jsou často otevřenými otázkami, proto je tato forma výzkumu časově náročnější. Pomůže nám ovšem prozkoumat daný problém více do hloubky. [2, 3]

1.2 Distribuce odlišných verzí softwaru různým klientům

V této kapitole popíši, jak jsem při výzkumu postupoval a na co jsem přišel. Podrobně rozpracuji několik možností, pomocí kterých lze distribuovat různé verze softwaru odlišným klientům.

1.2.1 Postup výzkumu

Pro obecnou rešerši potřeb výše zmíněných vývojářů jsem zvolil kvalitativní formu výzkumu, neboť jsem chtěl položit otevřenou otázku menší skupině lidí.

Před položením dotazu bylo nutné zvolit médium, pomocí kterého budu moci získat potřebné informace. Mojí první volbou bylo fórum Stack Overflow. Jedná se o webovou stránku, na níž mohou nejen profesionální programátoři, ale i nadšenci do programování klást dotazy či na ně odpovídat. Toto fórum tedy funguje díky komunitě, která ho aktivně používá a dělá z něj jednu z nejpopulárnějších internetových stránek na světě. [4] Právě proto jsem zvolil tuto možnost. Rozsah probíraných témat je velice široký, jedinou podmínkou je, aby kladený dotaz souvisel s počítačovým programováním. Toto kritétium bezpochyby splňuji.

Dotaz jsem položil v angličtině, jak je na Stack Overflow zvykem. Zeptal jsem se, s jakými problémy se programátoři setkávají, jestliže jejich klienti od softwaru očekávají každý jiné vlastnosti a jak tyto situace řeší. Stejný dotaz jsem položil na fóru Software Engineering, které spadá pod stejnou společnost jako Stack Overflow [5] a které se týká přímo softwarového inženýrství, s nímž je můj dotaz úzce spjatý.

Výše zmíněné portály nabízí při pokládání nového dotazu příručku, jak má takový dotaz vypadat. Mezi tato pravidla patří kupříkladu shrnutí problému do nadpisu o délce jednoho řádku či poskytnutí dostatečného detailu v hlavní části otázky. [6] Ač si myslím, že jsem tato pravidla splnil, mnou položené dotazy byly po pár dnech uzavřeny a skryty. Důvodem skrytí bylo u obou dotazů pravděpodobně jejich záporné hodnocení, které značí, že je dotaz nejasný či neužitečný. Na fórech byl dotaz uzavřený z důvodu, že se jednalo o dotaz zakládající na názorech (centrum pomoci Stack Overflow informuje, že tento typ otázek není pro fórum vhodný [7]) či že dotaz je nutné konkretizovat, neboť nelze jednoznačně určit správnou odpověď.

I přesto, že mnou položené dotazy nebyly na fórech pozitivně přijaty, obdržel jsem dvě odpovědi týkající se tématu. První byla formou krátkého komentáře pod dotazem a v případě druhé se jednalo o rozsáhlejší text.

Současně s položením dotazů na fórech jsem prováděl rešerši vyhledáváním na internetu, neboť jsem tušil, že obecný dotaz na fóru nesesbírá příliš mnoho odpovědí. Navíc každé správné získávání informací by se nemělo skládat pouze z dotazování, ale i ze samotného vyhledávání. Jak informace, jež jsem nabyl z odpovědí na mnou položené dotazy, tak informace, které jsem si sám vyhledal, jsem zužitkoval při psaní následující kapitoly.

1.2.2 Možnosti dodávání softwaru

Významnou potřebou softwarového inženýra při vývoji je způsob, jakým bude svůj produkt dodávat a jak v tomto způsobu zohlední různé verze tak, aby každý klient dostal přesně to, co potřebuje. Těchto možností je mnoho. V této podkapitole popíšu některé z nich společně s jejich výhodami a nevýhodami.

1.2.2.1 Integrace specifických vylepšení do základního produktu

Každý produkt, který je vyvíjen pro vícero klientů a který počítá s budoucími změnami, má nějakou část, která je pro všechny stejná. Té se říká základní produkt a od výsledného produktu se může lišit. Tento pojem si lze představit například na chytrém telefonu. Toto zařízení má několik základních produktů, například schopnost telefonovat, pořizovat fotky nebo přehrávat videa. Zařízení pak dává tyto prvky dohromady, neboť je nelze zakoupit samostatně. [8]

Při dodávání softwaru lze prvky, o které žádá jeden zákazník, integrovat do základního produktu tak, že je budou moci využívat i jiní zákazníci. Takovéto nové prvky je možné zahrnout do základního produktu automaticky či za příplatek, případně lze nechat klienta, aby se sám rozhodl, zda danou funkcionalitu chce. Konkrétní distribuci zvolených vlastností softwaru je pak možné řešit na úrovni vývojáře či až na úrovni klienta.

V případě první možnosti softwaroví inženýři například vytvoří konfigurační soubor, do nějž sami zapíší, které funkčnosti zákazník od produktu požaduje a připraveným skriptem, jemuž na vstupu takový konfigurační soubor dodají spolu se zdrojovými kódy, sestaví výsledný software na míru. Klient tedy dostává produkt s takovými vlastnosti, jaké požadoval, a současně nemusí tyto skutečnosti konfigurovat. [9]

Pokud je distribuce řešena na úrovni zákazníka, je klientovi dodán software, jenž obsahuje veškeré vývojáři naprogramované funkcionality. Samotný zákazník může využít například přepínatelné tlačítko, jímž aktivuje právě takové funkce, jež žádal.

Výhodou tohoto přístupu je skutečnost, že není nutné pro každého klienta vést specifickou vývojovou větev ve verzovacím nástroji, jak tomu je u jedné z dalších možností, kterou popíši níže. Novou funkcionalitu pak vývojář přidá do softwaru jako každou novou aktualizaci, případně přidá i možnost si tuto funkcionalitu zapnout či vypnout nebo upraví sestavovací skript tak, aby bylo možné vytvořit výsledný software s touto novou vlastností. Současně zákazník nepotřebuje mít vlastní tým vývojářů, který se mu o další vývoj bude starat, neboť ho zařídí ten tým, jenž produkt po celou dobu vyvíjí.

Jako nevýhodu spjatou s tímto přístupem uvedu fakt, že pokud do systému neustále přichází nové funkcionality a ne každý klient o ně má zájem, je nutné vytvářet větší množství přepínatelných tlačítek či sofistikovanější sestavovací skript tak, aby byly pokryty všechny nové prvky. S tím je nutné zajistit i jejich kompatibilitu nejen mezi sebou, ale i se základním produktem. [10, 11]

1.2.2.2 Vytvoření nové vývojové větve pro novou vlastnost softwaru

Dalším způsobem distribuování nových změn v softwaru je vytvoření nové vývojové větve ve verzovacím systému pro nově implementovanou funkcionalitu. Každou takovou větev pak můžeme chápat jako kód, ze kterého sestavíme produkt pro jednoho konkrétního zákazníka, jenž si právě tuto vlastnost do programu představoval. To se zpočátku zdá jako jednoduché řešení a tato jednoduchost je výhodou, potýká se však se spoustou problémů.

Při takovémto přístupu mohou nastat dvě různé situace: změny v jedné větvi se buď dají sloučit se změnami ve větvi jiné, anebo sloučení nastat nemůže z důvodu protichůdných funkčností. Pokud se nám změny sloučit nepodaří, máme dvě či více větví, ve kterých se nachází odlišné variace produktu a které je nutné udržovat pro budoucí vývoj. To patří mezi nevýhody tohoto řešení, a to z důvodu, že údržba vícero větví stojí vývojáře více času a vývojářskou společnost více peněz. Toto řešení navíc porušuje tzv. DRY princip, který jednoduše říká, že se nemáme opakovat. [12] Porušujeme ho skutečností, že v každé vývojové větvi máme určitou část kódu, jež je pro všechny větve společná. Tento princip je dobré neporušovat, jelikož zamezuje nepohodlnému a nepraktickému vývoji softwaru (například právě duplikaci kódu).

Samozřejmě se může stát, že po rozdělení vývoje do dvou či více větví se nám podaří větve sloučit do jedné, neboť si jejich nové funkcionality nerozporují, ale to nelze dopředu zaručit. Nelze totiž předpokládat, co bude v budoucnu jaký zákazník požadovat. Proto je tento způsob silně nedoporučován. [10, 11, 13]

1.2.2.3 Prodej samotného kódu

Jednou z možností, ač ne tak častou, je prodat zákazníkovi pouze kód, tedy bez výsledného spustitelného souboru. K tomu je vhodné přiložit dokumentaci kódu a návod, jak kód zkompilovat a výsledný program zprovoznit.

Výhodou a tím hlavním důvodem, proč má tato cesta smysl, je přenechání problematiky distribuce nových funkcionalit na klienta a vyhnutí se komplikacím, které přináší výše zmíněné způsoby.

I tento přístup má však handicap. Zákazník, jemuž je kód prodán, je donucen sehnat si svůj tým vývojářů, jenž na požadované funkčnosti bude pracovat a jenž bude nadále celý systém

udržovat. Dále může nastat ztráta klienta, jelikož veškerý další vývoj a údržba připadne na něj, pokud se zákazník a vývojářská společnost nedohodnou jinak. [10, 14]

1.2.3 Závěr výzkumu

Možností dodávání softwarového produktu je samozřejmě více, ale spousta z nich je pouhou modifikací mnou zmíněných možností či jsou již závislé na konkrétních technologiích.

Obecně lze přístupy i kombinovat a záleží vždy na konkrétním případě, jak moc je to vhodné. Například pokud máme klienta, který je pro nás významnější než ostatní klienti, můžeme pro něj zvážit možnost speciální vývojové větve a nutnost údržby této větve, kdežto ostatním zákazníkům tuto možnost nedávat. [10] I přes to se takovéto chování nemusí vyplatit, pokud je software příliš rozsáhlý a pokud je požadovaná změna od klienta taktéž signifikantní.

Jak plyne z výše popsaných postupů, každý má své výhody a nevýhody a neexistuje tedy jednoznačné nejlepší řešení. Je tedy nutné posuzovat každý softwarový produkt zvlášť. Mimo něj závisí volba vhodné možnosti distribuce i na typu a rozsahu nové funkcionality. Může se stát, že některé zákazníkem navrhované funkce mohou být v rozporu s účelem celého produktu. [15]

V neposlední řadě uvedu poznatek, jenž platí pro většinu výše popsaných způsobů, a sice že čím více různých, paralelně vyvíjených verzí vývojář vytvoří, tím je jejich distribuce a údržba náročnější. [16]

Protože je tato rešerše otázkou spadající i do projektového řízení, považuji za velmi důležité zmínit, že klíčovým faktorem pro výběr vhodné distribuční cesty je i zákazník, pro nějž je produkt vyvíjen. Je v zájmu vývojářů, resp. zaměstnanců dané vývojové společnosti, kteří se věnují komunikaci se zákazníkem, aby byl klient spokojený a tato společnost mohla díky němu a jeho finančnímu toku prosperovat. [17]

1.3 Metodiky vývoje

Pro získání nadhledu k celé tvorbě softwaru zjednodušeně popíšu, jaké kroky vedou k úspěšnému dokončení softwarového projektu. Takový proces tvorby označujeme pojmem softwarový proces a jeho jednotlivými částmi jsou analýza, návrh, implementace, testování a provoz. Existuje vícero způsobů, které popisují, zda tyto kroky provedeme jeden po druhém pouze jednou nebo zda budeme celou činnost opakovat v iteracích. Mezi ty nejpopulárnější patří tzv. vodopád, iterativní a agilní vývoj. [18]

1.3.1 Vodopád

Vodopád si můžeme představit jako skutečný vodopád, kde voda padá dolů a nahoru už se nikdy nevrátí. V tomto přístupu jsou jednotlivé fáze vykonány vždy právě jednou a jsou tedy oddělené.

Výhodou takového přístupu je dobrá predikovatelnost (jak dlouho bude celý proces trvat, jaké budou finanční nároky apod.), díky níž lze i poměrně jednoduše vydefinovat plán a koordinovat práci.

Tato metodika se ale pojí i s řadou nevýhod, jako je například pomalá reakce na změny a nutnost chápat cíle projektu již od samého začátku práce na projektu. [18, 19]

1.3.2 Iterativní

Aby se zlepšila reakce na změny a rychlost dodávky první podoby produktu, přešla řada společností na iterativní metodiku vývoje softwaru. Oproti vodopádu se zde tvoří několik verzí systému s tím, že se tyto verze vyvíjejí sekvenčně. Zákazníkovi je tedy ukázána první podoba produktu a je mu dána možnost změnit směr vývoje pro další iterace.

Mezi výhody mimo zmíněné stále patří definovaný plán a predikovatelnost.

Nevýhoda znalosti, co se po vývojářích chce již od začátku projektu, zůstává, ačkoli oproti vodopádu jsou zde zmíněné iterace. [18, 20]

1.3.3 Agilní

Velkým kontrastem proti zmíněným metodikám je přístup agilní. Nejedná se o jednu metodiku, nýbrž o obrovskou skupinu metodik, které mají společné rysy. Mezi ně patří mnohem kratší iterace oproti iterativní metodice, možnost nedělat každou verzi produkční a změna myšlení celého vývojového týmu.

Výhodou takových metodik je rychlost, tedy za jakou dobu zákazník uvidí první verzi produktu. Současně má klient obrovskou možnost ovlivnit cílovou podobu produktu, a dokonce je donucen toto provádět. Agilní metodiky totiž mohou vydávat nové verze i každý týden a klient, jenž může být přítomen na pracovišti vývojářů, do vývoje zasahuje a sděluje zpětnou vazbu k předešlé verzi a požadavky, jež na produkt klade.

Zmíněná výhoda může být ovšem i nevýhodou. Pokud klient není například z časových důvodů ochoten spolupracovat během vývoje, metodiky pak postrádají smysl. [21, 19]

1.3.4 Zvolený přístup

Protože během vývoje aplikace Deployment Manager budu spolupracovat s dalšími studenty, s čímž se pojí potřeba mít možnost reakce na změny, zvolím agilní přístup. Jeho vlastností využiji i během průběžných konzultací s vedoucím práce.

1.4 Analýza potřeb zadavatele

Pro analyzování potřeb společnosti Jagu, s. r. o., jsem zvolil opět výzkum kvalitativní, neboť bylo zapotřebí kontaktovat pouze dva respondenty. Těch jsem se ptal na velice specifické otázky k jejich představám týkajících se aplikace Deployment Manager. Dotazy byly velmi často otevřenými otázkami, jelikož pro mě jako výsledek výzkumu bylo hlubší porozumění této problematice.

Výzkum byl prováděn formou rozhovorů s vedoucím práce Ing. Oldřichem Malcem a Ing. Jiřím Hunkou. Z našich společných schůzek, kterých se v některých případech účastnili i studenti Alena Ježková a Tomáš Sládek pracující na aplikacích doplňujících Apps Manager, jsem si psal poznámky do svého souhrnného digitálního dokumentu týkajícího se mé bakalářské práce. Do něj jsem si také během analyzování problematiky poznamenával otázky na své kolegy.

1.5 Doména zadavatele

V této kapitole popíši, co jsem zjistil analyzováním potřeb zadavatele.

1.5.1 Aplikace License Manager a Builder

Jak se píše v zadání této práce, systém Deployment Manager je součástí trojice Apps Manager. Ta se dále skládá z aplikací License Manager a Builder. Tyto slouží k automatizaci prodeje a nákupu licencí na produkty a k jejich sestavování.

License Manager je webová aplikace, která vznikla jako bakalářská práce a jejíž autorem je Viktor Holý. Slouží ke správě licencí vydávaných softwarovými společnostmi. Umožňuje vývojářům přihlásit se jako prodejce a do aplikace vložit svůj software jako nový produkt, klientům přihlásit se jako zákazník a v aplikaci produkt zakoupit. License Manager byl vytvořen tak, aby byl schopen komunikovat s aplikací Builder. [22] I proto, že vzniká aplikace Deployment Manager

a je zapotřebí, aby s aplikací License Manager komunikovala, ujal se jejich úprav student Tomáš Sládek v rámci své bakalářské práce.

Aplikace Builder byla předmětem jiné bakalářské práce, která však nebyla dokončena. Proto bylo toto téma zadáno studentům předmětu *Softwarový týmový projekt 1*, kteří na projektu pokračovali v navazujícím předmětu *Softwarový týmový projekt 2*. Tohoto týmu jsem byl součástí. Aplikace Builder je systém, jenž reaguje na dění v aplikaci License Manager a sestavuje zákazníky zakoupené produkty. Pokud chce zákazník svoji licenci zrušit či mu je z důvodu neuhrazení poplatků deaktivována, o zrušení nebo deaktivaci běžící instance se postará taktéž Builder. Úprav této webové aplikace se ujala studentka Alena Ježková také ve své bakalářské práci.

Považuji za důležité zmínit, že momentálně tyto aplikace nejsou zadavatelem používány, neboť Builder nebyl v rámci našich školních projektů vyvinut do takové úrovně, v níž by byl schopen ostrého provozu. Má práce však předpokládá, že kolegové pracující na zbytku Apps Manageru své práce dokončí a systém bude použitelný jako celek.

Z toho plyne jeden z požadavků na Deployment Manager, a sice poskytnutí API zbylým aplikacím systému Apps Manager. Komunikační vztahy zmíněných aplikací nebyly předem definovány a jejich návrh bude předmětem kapitoly 2.7.

1.5.2 Produkty a jejich komponenty

Zákazník si v aplikaci License Manager zakoupí licenci se specifickou konfigurací (např. počet poboček) na daný produkt. Takový produkt se může skládat z jedné či více komponent. Jako příklad uvedu možnost rozdělení webové aplikace na klientskou a serverovou část, v anglicky psané literatuře označované někdy jako frontend a backend. To jsou dvě komponenty, které spolu komunikují, avšak každá běží samostatně. Ve chvíli, kdy si uživatel zakoupí produkt skládající se z vícero komponent, je nutné sestavit všechny tyto komponenty, jinak by celková aplikace nemusela fungovat korektně. [23]

1.5.3 Změna verze aplikací

Společnost Jagu, s. r. o., své produkty neustále vylepšuje a opravuje nalezené chyby, čímž vznikají nové verze aplikací. Ty je zapotřebí klientům nasazovat. V aktuálním stavu proces změny verze probíhá následovně:

1. Společnost vydá novou verzi komponenty
2. Tato nová verze je nasazena všem zákazníkům majícím starší verzi komponenty

Po vydání nové verze zadavatel mění verzi všem klientům z důvodu, že by v případě klienta, jehož verze by zůstala nezměněna, vznikla nutnost si verze nasazených aplikací někam zaznamenávat. Vést si manuálně vytvářené záznamy popisující stav produktu u všech zákazníků, kterých může být velký počet, není však jednoduchý úkol. Občas se stává, že některý klient není ochoten za poskytnuté služby včas uhradit pravidelný poplatek. Takoví zákazníci patří k těm, kterým by společnost Jagu, s. r. o., chtěla zamezit přístup k aktualizacím jejich produktů. Vývojářům této společnosti vzniká tak potřeba udržovat si přehled o tom, jakou verzi každý klient má, aby nejnovější verzi nebylo nutné nasazovat všem klientům.

Zadavatel pro verzování využívá aplikaci GitLab, do níž ukládá veškeré verze svých aplikací. Pro nasazování aplikací klientům využívá funkcionalitu GitLab CI/CD. Jedná se o automatické sestavení aplikace, spuštění automatických testů a nasazení vzniklé aplikace na server. Tento proces se standardně spouští automaticky po vložení nového kódu do repozitáře (tzv. *push*), uživatel si ale může nadefinovat různá pravidla, kterými lze postup konkretizovat. Lze například použít automatické sestavení a otestování při každém vložení kódu do repozitáře, ale nasazení aplikace jen za manuálně provedených činností (např. vytvoření nového *tagu* v projektu). [24, 25]

Tag zobrazuje neměnný stav repozitáře. Tedy po vytvoření *tagu* v něm již není možné provádět žádné úpravy. Zpravidla se jím označují důležité milníky v projektu jako je vydání nové verze. [26] Tímto způsobem *tagy* využívá i zadavatel, jenž je povětšinou pojmenovává řetězci ve formátu *SemVer* (např. `v1.4.5`, `v1.2.` apod.). Tento formát popíši podrobněji v kapitole 1.5.5.

Samotnou změnu verze aktuálně pracovníci provádí poloautomaticky. Jakmile společnost vydá novou verzi, zodpovědná osoba ručně spustí aktualizaci běžících aplikací v systému GitLab. Do souboru `.gitlab-ci.yml`, jenž popisuje celý průběh automatického procesu spouštěného při úpravě obsahu repozitáře či manuálním spuštěním, musí uvést určité parametry popisující konkrétní instance běžících aplikací u klientů. V případě zadavatele se jedná o tzv. *slug*, neboli unikátní řetězec skládající se například z příjmení zákazníka bez diakritiky, čímž se jednoznačně identifikuje daná instance aplikace. Kupříkladu aplikace, již zadavatel provozuje pro klienta jménem Josef Novák, by se označovala slovem `novak`. V některých případech se pro tvorbu řetězce zvaného *slug* používá název klientovy prodejny či další zadavateli známé údaje.

Tento způsob není příliš pohodlný, neboť je nutné pro každého nového zákazníka vytvářet manuálně nový *slug* a ten je zapotřebí vložit do zmiňovaného konfiguračního souboru v aplikaci GitLab. Zde figuruje lidský faktor, jelikož se člověk, který konfigurační soubor upravuje, může ukliknout či upsat. Zadavatel požaduje zpřehlednění a zjednodušení této činnosti.

1.5.4 Doporučování verzí souvisejících komponent

Dalším diskomfortem pojícím se se změnou verze softwaru je možnost způsobení nesprávného fungování aplikace s tímto softwarem komunikující. S tímto problémem se zadavatel také potýká, jelikož se některé jím vyvíjené aplikace skládají z vícero komponent.

Momentálně je pověřená osoba nucena ručně porovnat verzi běžící aplikace, která má být aktualizována s verzí související aplikace. Při tom je nutné ověřit, zda nově nasazovaná verze aktualizované aplikace umožní správnou komunikaci se související komponentou, neboli zda jsou tyto verze kompatibilní.

Protože v současném stavu dostává novou verzi aplikace každý zákazník zadavatele, tento proces je vykonán jen jednou pro každou novou verzi. Nicméně, jestliže by zadavatel měl možnost nasazovat různým klientům odlišené verze softwaru, bylo by časově náročné tento proces opakovat pro každého zákazníka zvlášť. Dále by se tím zvýšila šance nastání lidské chyby. Proto mezi potřeby zadavatele patří i automatizace kontroly kompatibilit verzí v podobě jejich doporučování, a sice za pomoci symbolů sémantického verzování, které bude popsáno v následující kapitole.

1.5.5 Sémantické verzování

Jak jsem již zmínil v kapitole 1.5.3, zadavatel využívá pro pojmenovávání nových verzí tzv. *SemVer*. Plný název tohoto formátu je sémantické verzování. Obecně ho lze definovat předpisem ve tvaru `MAJOR.MINOR.PATCH`, kde jsou jednotlivá slova nahrazena číslicemi. Jakmile vývojář vydá novou verzi, určí její označení ve formátu sémantického verzování takto:

- `MAJOR` popisuje velké změny v kódu, které způsobily nekompatibilitu API
- `MINOR` popisuje přidání nové funkcionality, která je však zpětně kompatibilní
- `PATCH` popisuje drobné změny v kódu, zpravidla jen opravy chyb, které jsou taktéž zpětně kompatibilní

Formát dále umožňuje přidat za položku `PATCH` bližší označení verze (např. alpha, beta) a disponuje rozsáhlým souborem pravidel, jak lze tyto verze porovnávat.

Také nabízí možnost definování rozsahů. Pro ně využívá speciální symboly jako `~`, `^`, `x` či `*`. Rozsah je způsob, jakým lze popsat množinu verzí, která vývojáři zrovna vyhovuje. Například po vydání nové verze jedné komponenty lze tímto způsobem vydefinovat, jaký tvar musí mít

sémantický formát verze druhé komponenty, aby spolu tyto dvě části byly kompatibilní. Tato skutečnost je využívána ve značné míře. Používají ji známé nástroje pro správu závislostí na softwarových knihovnách či balíčcích, a sice Composer, který spravuje závislosti aplikacím v jazyce PHP, a npm, jenž je správcem balíčků pro software v jazyce JavaScript.

Jako příklad definice rozsahu lze uvést například řetězec `^1.2`. Toto omezení říká, že podporovanými verzemi jsou verze větší nebo rovny verzi `1.2.0` a současně menší než verze `2.0.0`.

Rozsahy lze uvést i zjednodušeně pomocí znamének `=`, `>`, `<` či jejich kombinací. Dále lze dvě verze ve formátu *SemVer* spojit pomlčkou, čímž se taktéž definuje rozpětí verzí. [27, 28, 29]

1.6 Specifikace požadavků

Důležitým výstupem analýzy domény zadavatele je kompletní specifikace funkčních a nefunkčních požadavků. Té se budu věnovat v této kapitole.

Výsledný seznam byl vytvořen po konzultacích se zadavatelem a byl jím i revidován, aby se obě strany ujistily, že od výsledného produktu mají stejná očekávání. Ač byl vývoj aplikace prováděn agilním způsobem, níže uvedené požadavky byly formulovány již na počátku vývoje.

1.6.1 Kategorizace požadavků

Při specifikaci požadavků je vhodné užít některé klasifikační způsoby. Mezi ně patří například kategorizace FURPS a MoSCoW, pomocí nichž každý z požadavků kategorizuji.

1.6.1.1 FURPS

Kategorizace FURPS obsahuje 5 odlišných tříd, do nichž je možné požadavky třídit dle jejich zaměření. Těmito třídami jsou:

- F — Funkctionality — funkčnost
- U — Usability — použitelnost
- R — Reliability — spolehlivost
- P — Performance — výkon
- S — Supportability — podporovatelnost/rozšiřitelnost

Požadavky na *funkčnost* většinou popisují hlavní funkcionality softwaru. *Použitelnost* se zpravidla týká uživatelského rozhraní a jeho parametrů, kterými mohou být například vzhled či konzistence. *Spolehlivost* je kategorie, do níž zařazujeme požadavky jako dostupnost softwaru či jeho schopnost obnovení fungování po selhání. Požadavky na *výkon* mohou charakterizovat dobu odezvy nebo dobu, kterou trvá obnovení systému. Kategorie poslední, *podporovatelnost/rozšiřitelnost*, popisuje, jak dobře lze software testovat, škálovat, jak je či není konfigurovatelný, udržitelný a rozšiřitelný. [30]

1.6.1.2 MoSCoW

Označení MoSCoW je jen mnemotechnická pomůcka pro 4 kategorie, do nichž lze požadavky roztřídit dle jejich priority. Tyto kategorie jsou následující:

- M — Must have — musí mít
- S — Should have — měl by mít
- C — Could have — mohl by mít

- W — Won't have for now — prozatím mít nebude[1]

Kategorie *Must have* se přiřazuje požadavkům, bez nichž by výsledný produkt postrádal smysl. Například internetový obchod bez existence nákupního košíku umožňujícího samotné nakupování.

Should have náleží důležitým požadavkům, které má výsledná aplikace splňovat, ale v případě časového nedostatku mohou tyto požadavky býti oddáleny na budoucí vývoj. Bez jejich splnění ovšem produkt postrádá smysl. Jako příklad uvedu *load balancing*, neboli vyvažování zátěže při velkém náporu uživatelů softwaru jako je například webová aplikace. [31]

Možnost třetí, *Could have*, popisuje vlastnosti, které by bylo dobré mít v případě, že jejich vývoj nebude časově či cenově náročný. Nejedná se však o hlavní smysl produktu a lze se bez nich obejít. Příkladem pro internetový obchod je tzv. *cross-selling*, neboli prodávání podobného či doplňkového zboží zákazníkovi. [32]

Kategorie *Won't have for now* označuje požadavek, který není v aktuálním stavu vývoje potřebný, ale který může být potřebný v pozdější fázi. Může se jednat o funkcionalitu umožňující zákazníkům psát recenze na zakoupené produkty v internetovém obchodě. [33, 34]

1.6.2 Funkční požadavky

V této části práce vyjmenuji a popíši veškeré funkční požadavky, které jsou na Deployment Manager kladeny. U každého požadavku uvedu jeho identifikátor, název, popis a prioritu dle MoSCoW.

U jednotlivých požadavků nebude uvedena kategorie dle FURPS, neboť jsou všechny tyto požadavky kategorie *funkčnost*.

F1: Systém umožní uživateli zobrazit a filtrovat seznam nasazených aplikací

Systém umožní zobrazit seznam aplikací, které byly zakoupeny zákazníky v aplikaci License Manager a následně nasazeny aplikací Builder. Dále bude možné vyfiltrovat konkrétní aplikace podle jejich atributů.

Priorita: musí mít

F2: Systém umožní uživateli změnit verzi zvolené aplikace

Uživatel bude moci změnit verzi aplikace, která je nasazena na serveru. Bude si moci vybrat, na jakou verzi se má aktuální verze změnit. Po volbě bude probíhat sestavování nově zvolené verze aplikace a její nasazování na server, kde běžela aplikace v původní verzi.

Priorita: musí mít

F3: Systém umožní uživateli automaticky zvolit nejnovější kompatibilní verzi komponenty nasazené aplikace

Uživatel bude mít možnost si po změně verze komponenty A nechat automaticky zvolit vhodnou verzi komponenty B takovou, která je s komponentou A kompatibilní a která je současně nejnovější možná. Uživateli bude umožněno v systému zadat informaci o tom, která verze komponenty B je kompatibilní se zadanou verzí komponenty A. Uživateli bude dále umožněno používat symboly sémantického verzování pro speciální požadavky na verzi.

Priorita: mohl by mít

[1] Označení „for now – prozatím“ se používá v agilní vývojové metodice, neboť její časté iterace umožňují změny specifikovaných požadavků.

F4: Systém umožní autentizovat uživatele

Uživatel se za účelem vstupu do systému bude muset přihlásit uživatelským jménem a heslem, aby se ověřilo, že se jedná o oprávněnou osobu.

Priorita: musí mít

F5: Systém umožní uživateli správu skupin nasazených aplikací

Systém umožní shlukovat zvolené nasazené aplikace do skupin, což umožní pohodlněji vykonat totožný úkon na vícero aplikacích jednodušeji. Tyto aplikace musí být instancemi jedné komponenty. Systém nabídne i zobrazení takto vytvořených skupin a možnost změny verze několika běžících aplikací na jednu konkrétní verzi. Mimo to bude uživateli umožněno skupinu odstranit.

Priorita: mohl by mít

F6: Systém bude schopen navázat komunikaci s aplikací Builder

Systém bude schopen zahájit komunikaci s aplikací Builder, jež zajišťuje mimo jiné prvotní nasazení aplikace po zakoupení zákazníkem v aplikaci License Manager.

Priorita: musí mít

F7: Systém umožní uživateli spravovat produkty a jejich komponenty

Systém umožní uživateli správu produktů, tedy přidat a odebrat produkt, zobrazit seznam evidovaných produktů, definovat komponenty, z nichž se produkty skládají, přidávat a odebírat tyto komponenty a definovat kompatibilitu verzí těchto komponent.

Priorita: měl by mít

1.6.3 Nefunkční požadavky

V této podkapitole vypíši nefunkční požadavky na aplikaci. Každému z nich přiřadím identifikátor, název, popis, prioritu dle MoSCoW a kategorii dle FURPS.

N1: Systém bude možné ovládat prostřednictvím grafického uživatelského rozhraní

Uživatel bude moci ovládat systém prostřednictvím grafického uživatelského rozhraní. V tomto rozhraní bude možné zobrazit nasazené aplikace, provádět správu jejich verzí a provádět správu produktů.

Priorita: měl by mít

Kategorie dle FURPS: použitelnost

N2: Systém poskytne rozhraní pro komunikaci aplikacím License Manager a Builder

Systém poskytne API aplikacím tvořící systém Apps Manager, čímž umožní aplikaci LM poslat informaci o zakoupené licenci zákazníkem, kterou systém zpracuje a dále předá aplikaci Builder, která sestaví instanci produktu zakoupeného klientem. Dále tímto umožní aplikaci LM požádat o deaktivaci běžící aplikace.

Priorita: musí mít

Kategorie dle FURPS: podporovatelnost/rozšiřitelnost

Tabulka 1.1 Pokrytí funkčních požadavků případy užití

	F1	F2	F3	F4	F5	F6	F7
UC1				✓			
UC2	✓			✓			
UC3	✓			✓			
UC4		✓		✓			
UC5		✓	✓	✓			
UC6				✓	✓		
UC7				✓	✓		
UC8				✓	✓		
UC9				✓	✓		
UC10		✓			✓		
UC11			✓	✓			
UC12				✓			✓
UC13				✓			✓
UC14				✓			✓
UC15				✓			✓
UC16				✓			✓
UC17				✓			✓
UC18				✓			✓
UC19		✓			✓	✓	

1.6.4 Případy užití

Pro bližší porozumění fungování aplikace k funkčním požadavkům přidám i seznam případů užití. Jedná se o souhrn kroků, které musí uživatel či jiný aktér vykonat v daném pořadí, aby dosáhl nějakého cíle, jejž má aplikace umožňovat. Tyto kroky jsou popsány hlavním scénářem. V některých případech mohou nastat různé situace, proto se u takových případů uvádí i scénář alternativní. [35, 1]

Každý případ užití realizuje alespoň jeden z funkčních požadavků na software, aby se po projití všech případů užití mohlo rozhodnout, zda jsou v nich veškeré požadavky naplněny. Tento přehled je k nahlédnutí v tabulce 1.1. V ní lze vidět, že každému funkčnímu požadavku je přiřazen minimálně jeden případ užití, neboli že všechny funkční požadavky jsou v případech užití realizovány. Současně neexistuje žádný případ užití, jenž by nerealizoval žádný funkční požadavek.

V této podkapitole popíši případy užití a jejich aktéry.

1.6.4.1 Aktéři

Mezi aktéry v případech užití aplikace patří:

Nepřihlášený uživatel Jedná se o osobu, jež bude vznikající webovou aplikaci používat. Tato osoba není běžným uživatelem počítače, naopak se jedná o pokročilého uživatele, a sice administrátora/správce nasazování aplikací. Tento aktér se zatím nepřihlásil do aplikace, proto mu je většina funkcionalit nepřístupná.

Přihlášený uživatel Uživatel, který byl nepřihlášený a úspěšně se přihlásil.

Systém Systémem je samotná aplikace Deployment Manager.

GitLab Aktérem GitLab se rozumí aplikace GitLab, s níž Deployment Manager komunikuje přes její API.

Z výše uvedeného vyplývá, že koncový zákazník nebude se systémem Deployment Manager interagovat. Zákazníci používají pouze aplikaci License Manager, v níž si zakupují licence na produkty. Nebudou tím pádem mít vliv na verze nasazovaných aplikací – ty určuje pouze administrátor/správce nasazování aplikací.

1.6.4.2 Případy užití

Pro přehlednost doplním případy užití jejich diagramem. Ten je z důvodu čitelnosti rozdělen na dvě části a je k nalezení na obrázcích 1.1 a 1.2.

UC1: Přihlášení uživatele

Uživatel se musí přihlásit, aby se ověřilo, že má právo manipulovat s nasazenými aplikacemi spravovanými v systému. Výsledkem je stav, kdy má přihlášený uživatel přístup do systému.

Aktér: Nepřihlášený uživatel
Hlavní scénář:

1. Uživatel otevře systém Deployment Manager.
2. Systém zobrazí uživateli stránku se žádostí o zadání přihlašovacího jména a hesla.
3. Uživatel vyplní požadovaná pole.
4. Systém zkontroluje, zda uživatel zadal správnou kombinaci uživatelského jména a hesla.
5. Jelikož uživatel vyplnil správnou kombinaci, systém zobrazí přehled nasazených aplikací.

UC2: Zobrazení seznamu nasazených aplikací

Uživatel si může zobrazit seznam nasazených aplikací, které si systém eviduje díky komunikaci s aplikacemi License Manager a Builder.

Aktér: Přihlášený uživatel
Hlavní scénář:

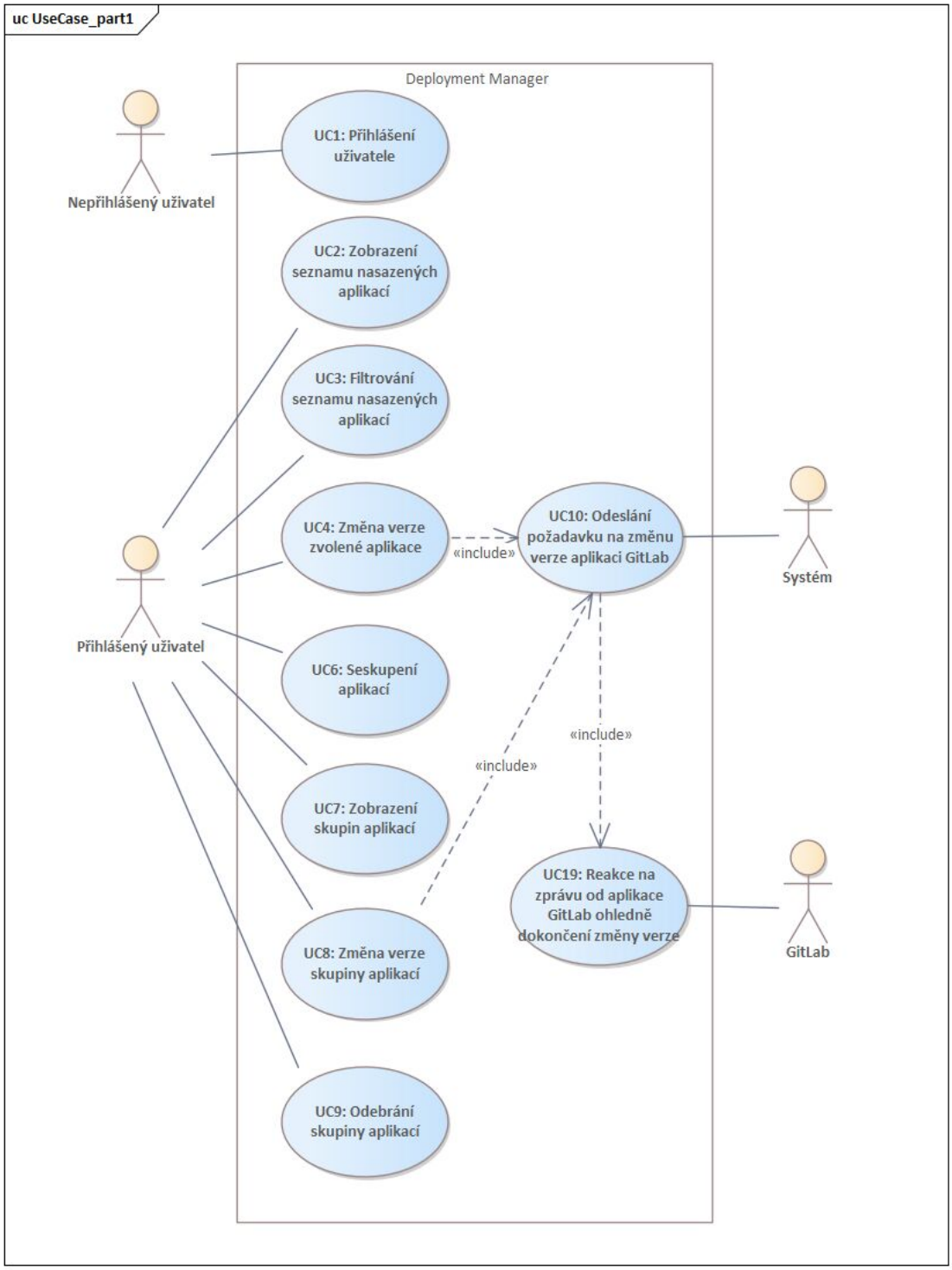
1. Uživatel zvolí možnost pro zobrazení seznamu nasazených aplikací.
2. Systém zobrazí evidované aplikace.

UC3: Filtrování seznamu nasazených aplikací

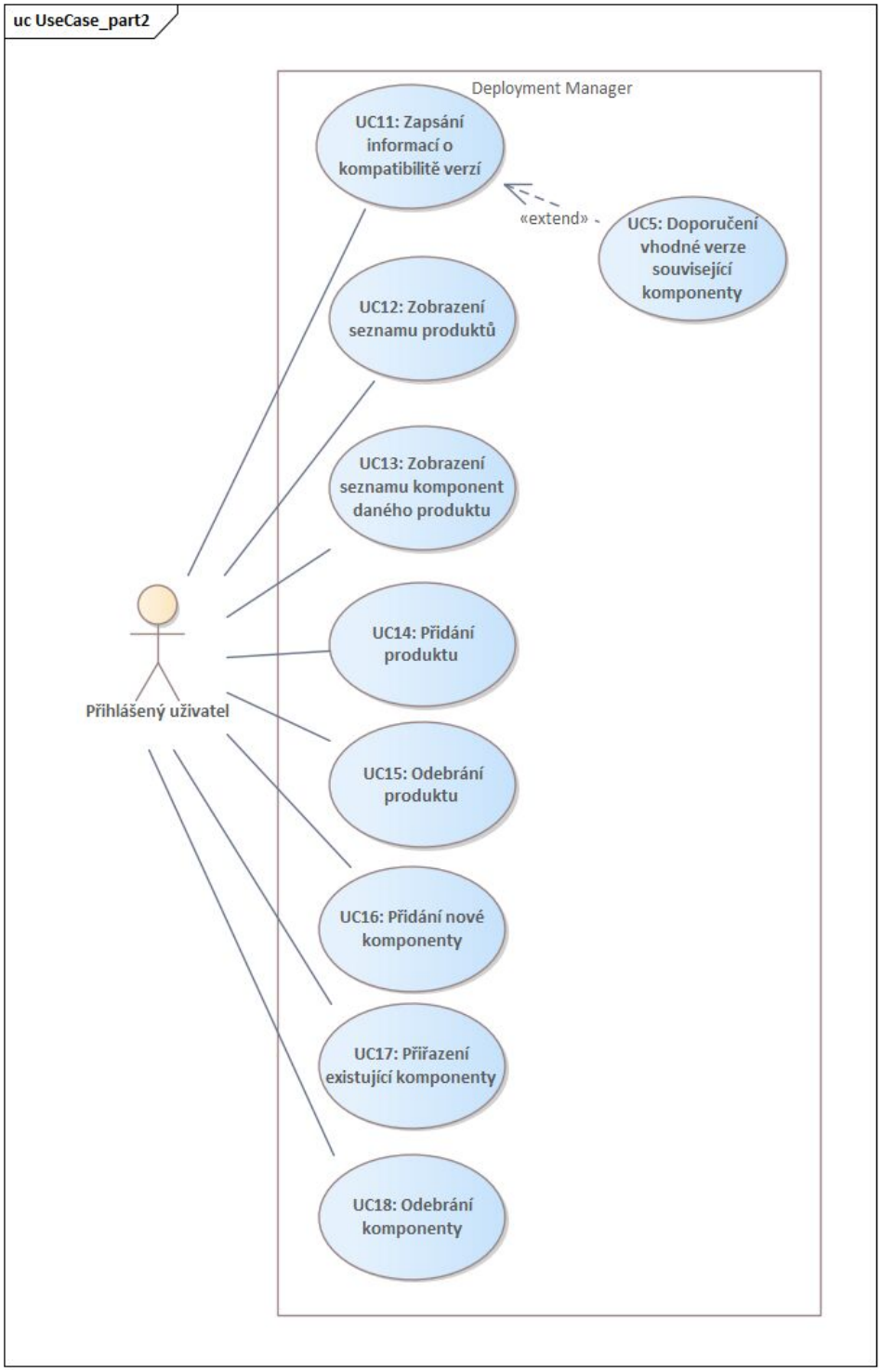
Uživatel může filtrovat seznam nasazených aplikací, které systém eviduje díky komunikaci s aplikacemi License Manager a Builder, dle jejich atributů.

Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro filtrování seznamu aplikací.
2. Uživatel si ze všech aplikací vybere jen ty požadované použitím možnosti pro filtrování.



Obrázek 1.1 Diagram případů užití – první část



Obrázek 1.2 Diagram případů užití – druhá část

UC4: Změna verze zvolené aplikace

Uživatel může změnit verzi některé z aplikací, které systém eviduje. Systém eviduje i dostupné verze dané aplikace. Výsledkem je nová instance dané aplikace ve zvolené verzi. Původní běžící instance bude zastavena.

Celý proces změny verze jsem zachytil na diagramu aktivit, který je k nalezení v příloze A.

Aktér: Přihlášený uživatel

Hlavní scénář:

1. Uživatel zvolí možnost pro změnu verze dané aplikace.
2. Systém zobrazí nabídku dostupných verzí.
3. Uživatel vybere z nabídky cílovou verzi.
4. Systém provede UC10.

UC5: Doporučení vhodné verze související komponenty

Systém umožní uživateli doporučit verzi komponenty, která komunikuje s komponentou, jejíž verzi mění. Systém nabízí sken souvisejících komponent a jejich dostupných verzí. Podmínkou pro tento případ užití je poskytnutí informace o kompatibilitě verzí komponent. Uživatel musí zapsat verze formátem sémantického verzování. Pokud uživatel nabídku přijme, bude výsledkem scénáře změna verze běžící aplikace i její související komponenty. Pokud odmítne, žádná změna verze neproběhne.

Aktér: Přihlášený uživatel

Hlavní scénář:

1. Uživatel provede UC11.
2. Uživatel zvolí aplikaci, jejíž verzi chce změnit.
3. Uživatel zvolí možnost pro změnu verze.
4. Systém vyhledá související komponenty aplikace, jejíž verzi se uživatel chystá změnit. Hledá pouze komponenty, pro něž je definovaná kompatibilita s běžící aplikací.
5. Vyhledáváním systém takovou komponentu nalezne a nabídne uživateli, zda chce změnit i její verzi. Uživateli je nabídnuta i konkrétní verze, kterou systém vyčetl díky UC11.
6. Uživatel nabídku přijímá a nechává si tak sestavit nové instance aplikací v nových verzích nebo nabídku uživatel odmítne.

Alternativní scénář:

1. Uživatel zvolí aplikaci, jejíž verzi chce změnit.
2. Uživatel zvolí možnost pro změnu verze.
3. Systém vyhledá související komponenty aplikace, jejíž verzi se uživatel chystá změnit. Hledá pouze komponenty, pro něž je definovaná kompatibilita s běžící aplikací.
4. Vyhledáváním systém zjistí, že žádná taková komponenta neexistuje a uživateli tedy nic nenabídne.

UC6: Seskupení aplikací

Uživateli je umožněno vytvořit skupinu aplikací, které jsou instancemi téže komponenty. Tuto skupinu je nutné pojmenovat. Výsledkem je vytvořená skupina nasazených aplikací.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro vytvoření skupiny.
2. Systém uživateli nabídne seznam komponent.
3. Ze seznamu komponent uživatel vybere komponentu, jejíž instancemi jsou požadované aplikace.
4. Uživatel zadá název skupiny.
5. Systém vytvoří požadovanou skupinu.
6. Uživatel zvolí možnost pro přidání aplikací do dané skupiny.
7. Systém zobrazí seznam nasazených aplikací, jež jsou instancemi komponenty zvolené při tvorbě této skupiny.
8. Uživatel vybere ze seznamu požadované aplikace.
9. Systém přiřadí požadované aplikace do skupiny.

UC7: Zobrazení skupin aplikací

Systém umožňuje uživateli zobrazit si vytvořené skupiny aplikací, aby nad nimi mohl provádět operace.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro zobrazení skupin aplikací.
2. Systém zobrazí uložené skupiny aplikací.

UC8: Změna verze skupiny aplikací

Systém nabízí uživateli změnit verzi všech aplikací, které tvoří skupinu.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí požadovanou skupinu ze seznamu.
2. Systém nabídne dostupné verze, na které mohou dané aplikace přejít.
3. Uživatel zvolí požadovanou verzi ze seznamu.
4. Systém provede požadavek UC10 pro každou aplikaci ze skupiny.

UC9: Odebrání skupiny aplikací

Systém umožňuje uživateli mimo vytváření a zobrazování skupin tyto skupiny také odebírat.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro odstranění požadované skupiny.
2. Systém danou skupinu odstraní.

UC10: Odeslání požadavku na změnu verze aplikaci GitLab

Systém je schopen odeslat požadavek aplikaci GitLab na změnu verze uživatelem zvolené aplikace. Výsledkem je změněná verze běžící aplikace na serveru.
Aktér: Systém
Hlavní scénář:

1. Systém odešle požadavek aplikaci GitLab.
2. Aplikace GitLab provede změnu verze dané aplikace.
3. Aplikace GitLab provede UC19.

UC11: Zapsání informací o kompatibilitě verzí

Systém umožňuje uživateli zadat informace o kompatibilitě verzí komponent daného produktu. Výsledkem je evidence nové kompatibility verzí v systému.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro zapsání informací o kompatibilitě verzí.
2. Systém zobrazí příslušná textové pole.
3. Uživatel zapíše požadované vztahy mezi komponentami.
4. Systém tyto změny uloží.

UC12: Zobrazení seznamu produktů

Systém umožňuje uživateli zobrazit seznam produktů, které systém eviduje.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro zobrazení produktů.
2. Systém zobrazí seznam evidovaných produktů.

UC13: Zobrazení seznamu komponent daného produktu

Systém umožňuje uživateli zobrazit seznam komponent, které systém eviduje pro daný produkt.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro zobrazení komponent požadovaného produktu.
2. Systém zobrazí seznam evidovaných komponent.

UC14: Přidání produktu

Uživateli je umožněno přidat produkt do systému. Uživatel musí vyplnit název produktu. Výsledkem je evidence nového produktu v systému.
Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro přidání produktu.

2. Systém zobrazí textové pole.
3. Uživatel vyplní název produktu.
4. Systém zaeviduje daný produkt.

UC15: Odebrání produktu

Uživateli je umožněno odebrat produkt ze systému. Tím zaniknou komponenty, ze kterých se produkt skládá, za podmínky, že tyto komponenty nespadají i pod jiný produkt. Výsledkem je odebraný produkt a jeho komponenty ze systému.

Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro odebrání požadovaného produktu.
2. Systém zkontroluje, zda se produkt skládá z komponent, které spadají i pod jiný produkt.
3. Taková komponenta neexistuje, takže systém odebere produkt a jeho komponenty ze své evidence.

Alternativní scénář:

1. Uživatel zvolí možnost pro odebrání požadovaného produktu.
2. Systém zkontroluje, zda se produkt skládá z komponent, které spadají i pod jiný produkt.
3. Taková komponenta existuje, takže systém odebere ze své evidence produkt a jeho komponenty vyjma té, jež spadá i pod jiný produkt.

UC16: Přidání nové komponenty

Uživateli je umožněno přidat komponentu do systému. Uživatel musí zvolit název komponenty. Každou novou komponentu je nutno přiřadit alespoň jednomu produktu, tedy nelze přidat komponentu, která není vázána na žádný produkt. Výsledkem je evidence nové komponenty v systému.

Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro vytvoření nové komponenty k produktu.
2. Systém zobrazí textové pole pro zadání názvu nové komponenty.
3. Uživatel toto pole vyplní.
4. Systém zaeviduje novou komponentu a přiřadí ji k danému produktu.

UC17: Přiřazení existující komponenty

Uživateli je umožněno přiřadit existující komponentu k danému produktu. Uživatel musí zvolit komponentu z nabízeného seznamu. K produktu není možné přidat komponentu, která už je s daným produktem svázána. Výsledkem je evidence vztahu existující komponenty a daného produktu v systému.

Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro přidání existující komponenty k produktu.

2. Systém zobrazí seznam nabízených komponent, jež nejsou s daným produktem spjaty.
3. Uživatel vybere požadovanou komponentu ze seznamu.
4. Systém zaeviduje nový vztah pro daný produkt a zvolenou komponentu.

UC18: Odebrání komponenty

Uživateli je umožněno odebrat komponentu daného produktu. Pokud daná komponenta není svázaná s žádným dalším produktem, je odebrána z evidence komponent. Pokud komponenta patří i k jinému produktu, je odebrána pouze od produktu, v němž uživatel zvolil možnost pro odebrání této komponenty.

Aktér: Přihlášený uživatel
Hlavní scénář:

1. Uživatel zvolí možnost pro odebrání komponenty daného produktu.
2. Uživatel potvrdí svoji volbu a komponenta je tím odebrána dle popisu tohoto případu užití.

UC19: Reakce na zprávu od aplikace GitLab ohledně dokončení změny verze

Systém je schopen zpracovat požadavek přijatý od aplikace GitLab o dokončené změně verze běžící aplikace. Výsledkem je uložení si podstatných informací a odeslání informací o provedené změně verze aplikaci Builder.

Aktér: GitLab
Hlavní scénář:

1. Aplikace GitLab dokončí změnu verze aplikace.
2. Aplikace GitLab pošle systému zprávu, že verze byla úspěšně změněna.
3. Systém zprávu přijme a uloží si potřebné informace.
4. Systém pošle aplikaci Builder informaci, že byl proveden úspěšný přechod na novou verzi dané aplikace.

Kapitola 2
Návrh

Po důkladném analyzování problému přichází další fáze softwarového procesu, a sice fáze návrhová. Jedná se o proces nastavování základů pro pozdější konstrukci aplikace. Zatímco analytická část popisuje, co je cílem softwaru, návrh říká, jak toho lze docílit. Dílčími problémy, které návrhová část řeší, jsou výběr programovacího jazyka, v němž bude aplikace vznikat, popis struktury databáze a volba její technologie. Dále je nutné zvolit vhodnou architekturu, a pokud aplikace komunikuje s jinými komponentami, provádí se i návrh komunikačního rozhraní. [36, 37]

Obsahem této kapitoly bude výběr technologie, v níž bude Deployment Manager implementován. Poté zvolím a popíši architekturu a navrhu samotný proces změny verze aplikace. Dále se budu věnovat výběru databázové technologie a návrhu tříd. Než kapitolu zakončím, popíši návrh API pro komunikaci jak s aplikací GitLab, tak s aplikacemi License Manager a Builder a vysvětlím, jak jsem tvořil uživatelské rozhraní.

Jelikož jsem vyvíjel agilně, prováděl jsem návrh v několika iteracích a jeho dílčí části se měnily. V této kapitole bude pro zachování přehlednosti představena jeho finální podoba.

2.1 Technologie

Technologií pro webové aplikace existuje dnes celá řada. Pokud nechceme vymýšlet, co již bylo vymyšleno, nabízí se možnost sáhnout po nějakém frameworku. Ten představuje strukturu, na níž lze software vybudovat, aniž bychom museli začínat od nuly. Frameworky jsou zpravidla vázány na daný jazyk, což jejich volbu může značně ovlivnit. [38]

Spousta frameworků se zaměřuje jen na backend či jen na frontend. Jelikož je ale vývoj aplikace rozdělené na tyto dvě komponenty zpravidla náročnější a Deployment Manager je má první vyvíjená webová aplikace, zaměřím se na technologie umožňující vzniku full-stack aplikací. Tedy technologie, v nichž lze vytvořit klientskou i serverovou část. [39]

Nyní popíši, jaké frameworky jsem pro implementaci Deployment Manageru zvažoval. Jelikož jsem zvolil framework Laravel (odůvodním v kapitole 2.1.3), bude jeho popis nejobsáhlejší. Také zmíním, že frameworků existuje velké množství a tato práce se nezaměřuje na jejich výčet a srovnání, proto popíši jen pár z nich.

2.1.1 Django

Prvním frameworkem, který popíši, je Django, který zjednodušuje konstrukci softwaru v programovacím jazyku Python. Vývojáři se chlubí tím, že Django je rychlý, bezpečný a rychle škálovatelný framework. Tato technologie disponuje veškerými funkcionalitami pro vývoj plnohodnotné

aplikace. Existují nicméně různé balíčky obsahující další funkčnosti. Těmi jsou například testování či debugování.

Protože webové aplikaci v Djangu jsou psány v jazyce Python, další výhodou je kratší kód, který se i rychleji píše. Jedná se o jazyk vhodný pro začátečníky kvůli jeho „pseudokódové“ syntaxi. [40, 41]

Django nabízí oficiální podporu pro operační systém Microsoft Windows, ale i pro Linuxové distribuce či pro Apple Mac. [42]

Co se využívané architektury týče, jedná se o MVT, neboli Model View Template. Zjednodušeně řečeno, model je používán pro vytváření tabulek a jejich atributů, view je funkce v jazyku Python, která jako parametr přijímá požadavek uživatele a vrací odpověď. Template je šablona obsahující statický obsah jako kód v HTML, CSS či JavaScriptu. [43]

Tato technologie mě velice zaujala a věřím, že i v ní by Deployment Manager mohl dobře fungovat. Nebyl jsem však přesvědčen tolik jako u později zmíněného Laravelu.

2.1.2 Spring

Spring je populární framework nad jazykem Java. Aplikaci v něm lze ovšem psát i v jazycích Kotlin a Groovy. Protože má samotný Spring komplikované spouštění, používá se jeho nadstavba Spring Boot. Ten umožňuje automatické vytvoření a nakonfigurování serveru, k němuž se budou uživatelé webové aplikace připojovat přes svůj webový prohlížeč. Autoři píší, že framework je navrhnutý tak, aby ho bylo možné rozfungovat co nejrychleji a s minimální konfigurací. Jisté zdroje však tvrdí, že řešení některých pokročilejších problémů vyžaduje velkou míru porozumění. [44, 45, 46]

Jazyk Java je populární objektově orientovaný programovací jazyk fungující na spoustě platforem jako je Microsoft Windows, Linux či Apple Mac. V Javě jsou vytvářeny nejen webové aplikace, ale i aplikace mobilní či desktopové nebo například hry. [47]

Tento framework je postaven na architektuře MVC. [44] Ta bude popsána v kapitole 2.2.

Se Springem mám již drobnou zkušenost z předmětu *Technologie Java*, kde jsem s ním byl obeznámen. Během vyvíjení semestrální práce jsem se však potýkal se spoustou problémů a pamatuji si, že dohledávání jejich řešení nebylo vždy jednoduché. Proto jsem tuto technologii nezvolil.

2.1.3 Laravel

Dalším ze spousty frameworků je Laravel, který je využíván vývojáři pracujícími ve skriptovacím jazyce PHP. Laravel se pyšní elegantní syntaxí a možností volby, zda bude vznikající aplikace monolitická či poskytne API klientské části, která bude s aplikací v Laravelu komunikovat. Tato technologie dále nabízí objektově relační mapování pro pohodlnou práci s relační databází (ORM podrobněji popíši v kapitole 2.4), fronty pro zpracování náročnějších požadavků (generování reportů, odesílání e-mailů) na pozadí a má v sobě zabudované techniky starající se o autentizaci. Touto funkcionalitou disponuje nejen pro uživatele samotné webové aplikace, ale i pro aplikace třetích stran komunikující s aplikací v Laravelu přes její API. [48]

V jazyce PHP, který je open sourcovým skriptovacím jazykem, běží k dubnu roku 2023 dle [49] 77,5 % všech webových stránek. Je populární z mnoha důvodů. Jedním z nich je například fakt, že tento jazyk je jednoduchý na naučení. Dále má spoustu nástrojů a frameworků, díky kterým je vývoj v něm ještě jednodušší. [50]

Laravel lze spustit na operačních systémech Microsoft Windows, Apple Mac i Linux. Nejuniverzálnější možností pro inicializaci a fungování projektu je použití Dockeru. Laravel disponuje rozhraním v příkazové řádce, které se nazývá Laravel Sail. To umožňuje spouštět celou aplikaci bez znalosti Dockeru a jeho příkazů. [51]

Další výhodou je velká komunita uživatelů (v lednu roku 2022 byl zvolen nejpopulárnějším backendovým frameworkem [52]) a velmi přehledná dokumentace, která navíc nabízí jednoduché

a vysvětlující ukázkové příklady. Jako začátečník ve tvorbě webových aplikací jsem z ní velmi dobře porozuměl základním pojmům.

Stejně jako Java Spring i Laravel používá architekturu MVC popsanou v kapitole 2.2.

Jak jsem již zmínil v kapitole 2.1, Laravel byla má volba. Přesvědčila mě právě čtivá dokumentace a velká komunita, což zvyšuje pravděpodobnost nálezu řešení častých i méně častých chyb. Dále mám s jazykem PHP ač drobnou, tak kladnou zkušenost ze střední školy. Ocenil jsem i možnost spouštění v Dockeru, který na svém zařízení mám a který mimo jiné zjednoduší možné budoucí nasazení a jeho automatizaci.

Tuto volbu jsem konzultoval s vedoucím práce a jelikož má aplikace neklade žádné neobvyklé nároky na technologii, vedoucí ji schválil.

2.2 Architektura

Jelikož jsem jako technologii pro Deployment Manager zvolil framework Laravel, výběr architektury byl jednoznačný. Laravel totiž implementuje architekturu MVC.

Tato architektura odděluje aplikační či byznysovou logiku od uživatelského rozhraní. Aplikaci rozděluje na tři části, a sice na model, view a controller.

Model se stará o data a jednoduché operace s nimi. Může odpovídat na žádosti o poskytnutí těchto dat nebo data na základě požadavku měnit.

View aplikaci poskytuje uživatelské rozhraní. Stará se o správné zobrazení dat poskytnutých modelem.

Poslední část zkratky MVC, tedy controller, propojuje vstup zadaný uživatelem ve view a model, který provede požadovanou úpravu dat. Také odpověď, již uživatel vidí na své obrazovce, je přenesena právě skrz controller. [53]

Jak lze vidět na obrázku 2.1, uživatel, který má k aplikaci vždy přístup přes view, nemůže přímo manipulovat s datovou vrstvou.

2.3 Návrh procesu změny verze aplikace

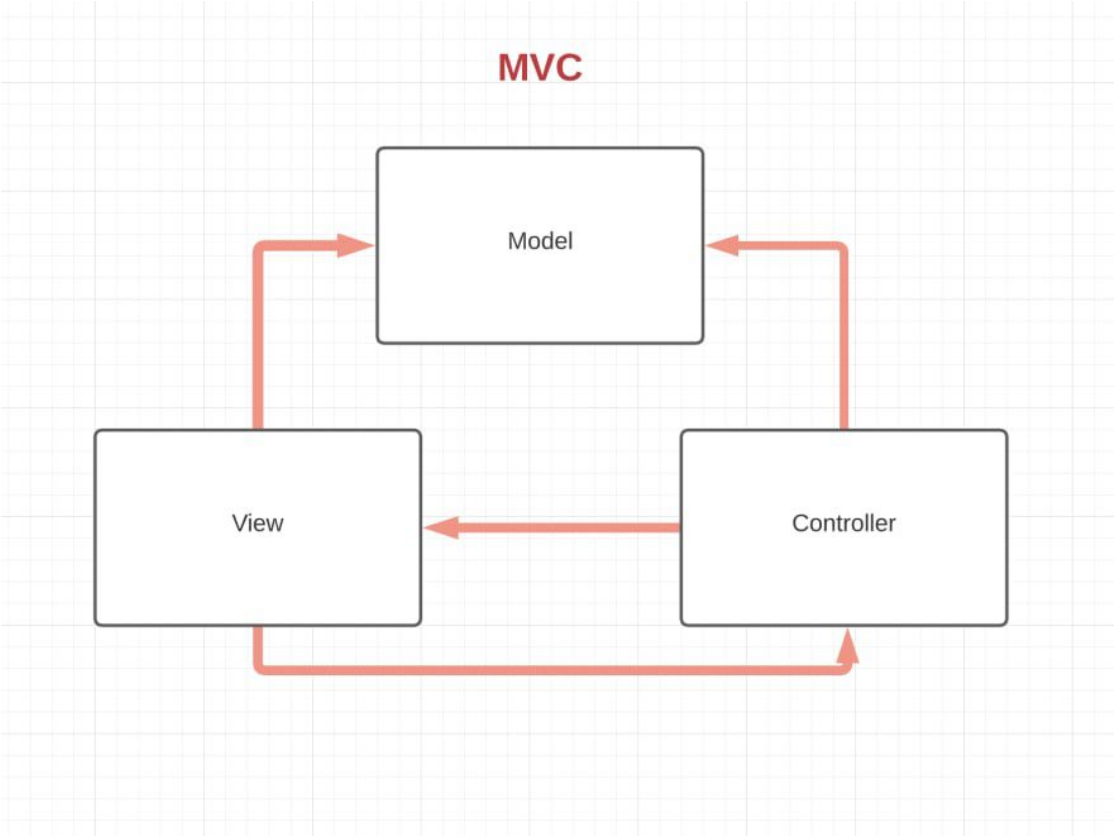
Jak plyne z požadavku F2, systém má umožnit změnu verze uživatelem zvolené aplikace. V této podkapitole tuto funkcionalitu navrhnu.

Nejprve zmíním, jak bude prováděno sestavení aplikací. Jelikož toto bude zajištěno aplikací Builder, podrobný popis bude jistě k nalezení v bakalářské práci Aleny Ježkové. Zmíním jen důležité informace, jež zužitkuji při navrhování procesu změny verze.

Builder pro sestavování používá prostředí (tzv. *environments*) v aplikaci GitLab. Jedná se o funkcionalitu, která popisuje, kam se nasazuje kód za běhu CI/CD. V prostředí lze vidět název procesu zvaného *job* (*job* je nejmenší jednotka, tzv. krok sestavování, při procesu CI/CD v GitLabu [55]), v jaké vývojářské větvi či *tagu* byl spouštěný proces definován a další údaje. Prostředí lze vytvořit více. Bez problému lze tedy mít pro každou instanci komponenty zákazníka jedno prostředí, což Builder momentálně dělá.

Aktuálně (ve stavu po skončení předmětu *Softwarový týmový projekt 2*, v němž jsem na aplikaci Builder společně s dalšími studenty pracoval) Builder pro identifikaci prostředí využívá speciální řetězce zmíněné v kapitole 1.5.3, které jsou ovšem tvořeny manuálně, jelikož jeho vývoj nebyl dotažen do úplného konce. Toho lze využít a tzv. *slug* generovat v Deployment Manageru, odkud může být zasílán Builderu přes jeho API při žádosti o sestavení produktu, která pochází z License Manageru. Přesněji bude tato komunikace vydefinována v kapitole 2.7.3.1.

Nabízí se tedy jednoduché řešení změny verze, kvůli kterému nebude nutné předělávat fungování aplikace Builder, a sice využití právě GitLab *environments*. Postačí spustit tzv. *pipeline* (proces, který se skládá z několika podprocesů zvaných *job* [55]) v takovém projektu v GitLabu, jehož instancí je aplikace (myšleno jako jedna komponenta produktu), jejíž verzi je potřeba změnit. Jelikož tyto *pipelines* mohou přijímat parametry, je možné jim předat takové názvy prostředí,



Obrázek 2.1 Schéma architektury MVC [54]

pro které chceme změnu verze provést. Abychom GitLabu sdělili, jakou verzi chceme sestavit a nasadit, spustíme *pipeline* v *tagu*, který obsahuje kód té verze. To jednoznačně identifikuje sestavovanou verzi, protože, jak jsem zmínil v kapitole 1.5.3, zadavatel pojmenovává jednotlivé *tagy* řetězci popisující verzi kódu, kterou tyto *tagy* obsahují.

Tento můj návrh byl zkonzultován s Alenou Ježkovou a poté i s vedoucím práce a shodli jsme se, že je pro systém Apps Manager vhodný.

2.4 Uchování dat

Jelikož si bude webová aplikace potřebovat ukládat data, je nutné navrhnout vhodný způsob tohoto uchovávání.

K ukládání dat se běžně používají databázové systémy. Framework Laravel nabízí Eloquent ORM, neboli objektově relační mapování. Díky němu není nutné při práci s daty v jednotlivých modelech přistupovat přímo do příslušné tabulky v databázi. Namísto toho může programátor volat metody objektů či přistupovat k jejich atributům. Tento kód je nezávislý na používané databázové technologii. [56, 57]

Zmíněné Laravelem podporované databáze jsou MariaDB 10.3+, MySQL 5.7+, PostgreSQL 10.0+, SQLite 3.8.8+ a SQL Server 2017+. [57] Z nich jsem zvolil PostgreSQL, jelikož je tento systém preferován zadavatelem. Díky ORM je však případná změna databázového systému jednoduchá a fungování aplikace vůbec neovlivní.

2.5 Návrh tříd

Již jsem zmínil, že data bude Deployment Manager ukládat do databáze. V této sekci popíši návrh konkrétních tříd, jejich atributů a metod.

Výslednou podobu tříd si lze prohlédnout na diagramu tříd v příloze B. Nejsou na něm zachyceny triviální metody, které slouží Eloquentu pro zachycení vazeb mezi daným modelem a modelem jiným.

Nyní popíši takové části již zmiňovaného diagramu, které jsem shledal komplikovanými a vhodnými k vysvětlení.

2.5.1 Komponenta, verze a kompatibilita

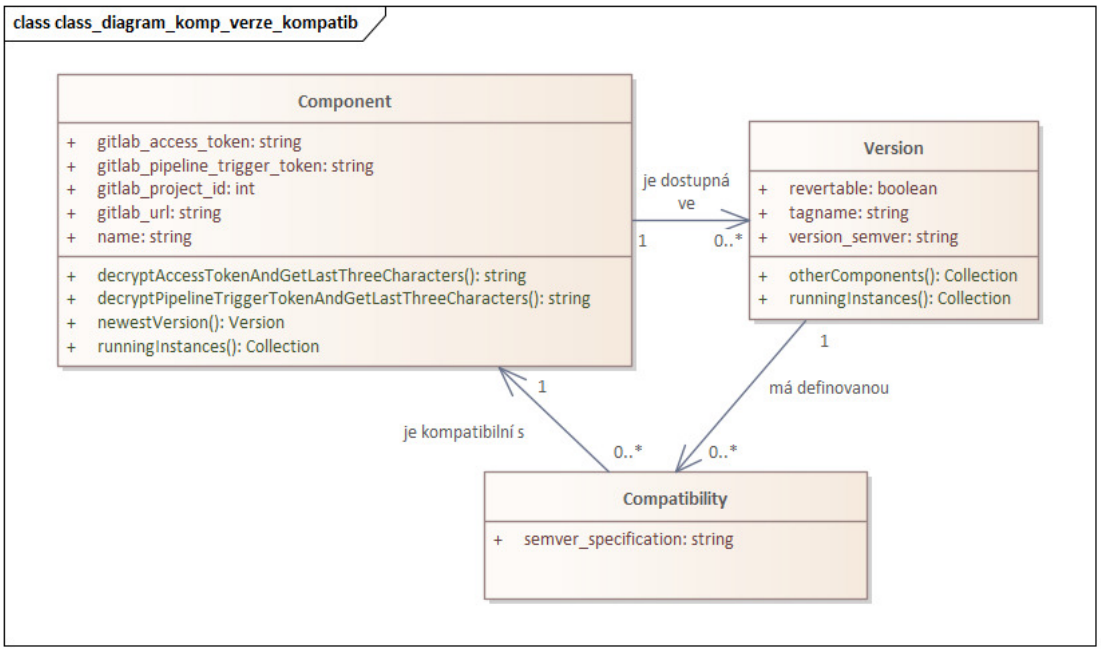
Pro správné fungování aplikace bylo třeba zajistit evidenci jednotlivých komponent, jejich verzí a kompatibilit mezi nimi. Ty bude moci uživatel v grafickém rozhraní vyplňovat. Vztah je zachycen na obrázku 2.2.

Komponenta je reprezentována třídou *Component*, která obsahuje důležité údaje pro identifikaci korespondujícího projektu v GitLabu a pro umožnění přístupu do něj.

Komponenta může být dostupná v různých verzích (třída *Version*). Tyto verze si systém získává z aplikace GitLab automaticky (viz 2.6.1.1). U verze evidujeme kromě názvu odpovídajícího *tagu* v GitLabu i její hodnotu ve formátu sématického verzování a příznak říkající, zda se lze z dané verze vrátit do starší verze neboli zda je verze zpětně kompatibilní. Toto byla jedna z připomínek zadavatele, které měl po některé z vývojových iterací a kterou jsem do systému úspěšně zapracoval.

Verze patří vždy právě jedné komponentě. Pokud máme například verzi se sémantickým označením `1.2.3`, lze zjistit, k jaké komponentě patří. Pokud by existovala jiná komponenta s verzí `1.2.3`, tato verze by byla jiná instance třídy *Version*, jelikož je svázána s jinou komponentou.

Kompatibilita popisuje vztah mezi konkrétní verzí komponenty a jinou komponentou. V aplikaci je reprezentována třídou *Compatibility* a má jediný atribut, kterým je specifikace povolených verzí ve formátu sémantického verzování. Jedná se o jeden řetězec, jenž využívá speciální symboly právě sémantického verzování. Příklady takových specifikací lze nalézt v kapitole 1.5.5.



Obrázek 2.2 Diagram tříd pro komponenty, verze a jejich kompatibility

V aplikaci bude zamezeno uživateli přidávat kompatibilitu nějaké verze s komponentou, které tato verze náleží. Současně platí, že kompatibilitu lze definovat mezi komponentou A a verzí jen v případě, že komponenta B, které tato verze náleží, patří do stejného produktu jako komponenta A.

2.5.2 Aplikace

Třída *Aplikace* reprezentuje jednu aplikaci, která je systémem Apps Manager spravována. Mimo adresy serveru, na níž aplikace běží (pokud již byla nasazena) eviduje Deployment Manager i její *slug* a stav. Detailní popis je na obrázku 2.3.

Stav aplikace je reprezentován výčtem hodnot, a sice `deploymentInProgress` neboli „probíhá nasazování“, dále `running` neboli „běží“ a `versionChangeInProgress`, což lze přeložit jako „probíhá změna verze“.

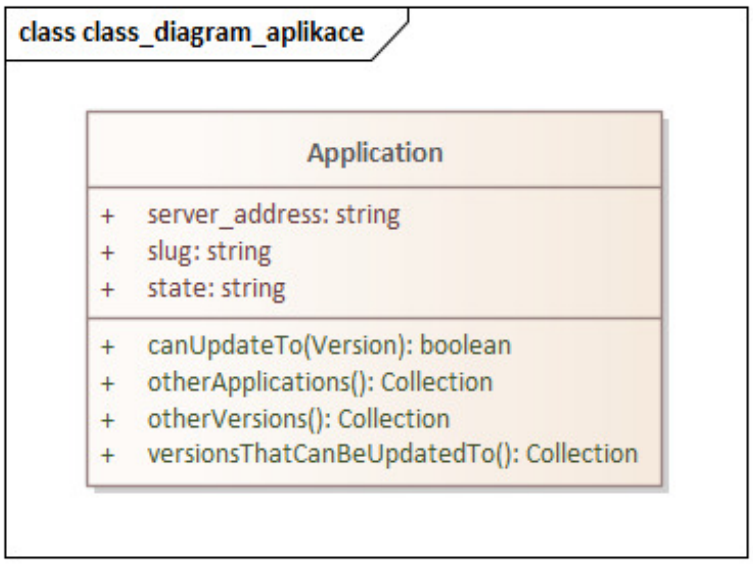
Přechody mezi jednotlivými stavy jsem znázornil pomocí stavového diagramu na obrázku 2.4. Komunikace mezi komponentami systému Apps Manager bude popsána v kapitole 2.7.

2.6 Komunikace s aplikací GitLab

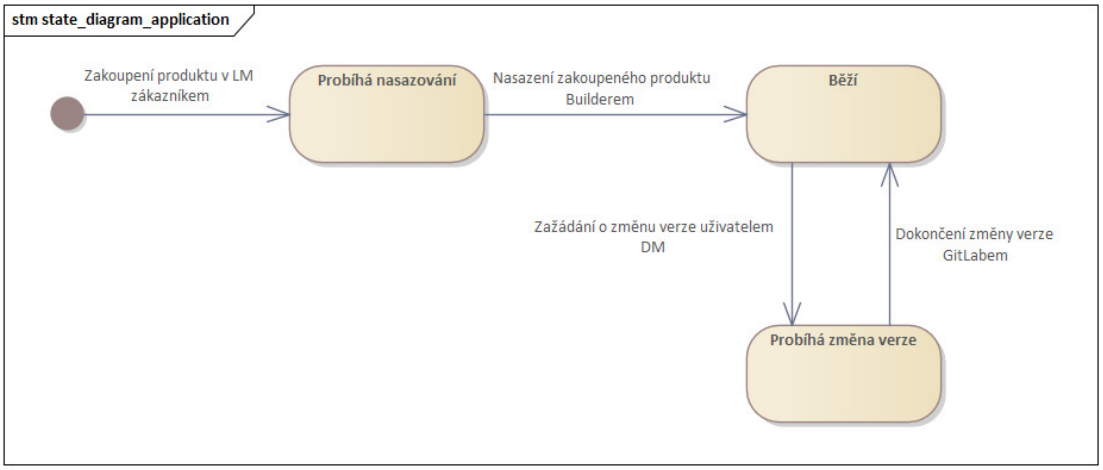
Funkční požadavek F2, týkající se změny verze běžící aplikace, bude pro svoji implementaci vyžadovat komunikaci s aplikací GitLab. V této podkapitole popíši jak API GitLabu, které bude Deployment Manager konzumovat, tak API, které budu poskytovat aplikaci GitLab. U každého zmiňovaného koncového bodu uvedu jeho adresu, HTTP metodu a případně potřebné parametry.

2.6.1 Konzumované API

GitLab nabízí REST API, pomocí kterého lze číst či zapisovat informace do projektů.



Obrázek 2.3 Diagram tříd – třída Aplikace



Obrázek 2.4 Stavový diagram aplikace

Jelikož zadavatel v GitLabu využívá neveřejné projekty, pro přístup k nim je nutné do hlavičky požadavku vložit přístupový token. [58] Ten bude systém po uživateli vyžadovat při tvorbě nové komponenty v databázi.

Těchto tokenů je více typů, já však zvolím tzv. *GitLab access token*, který lze vytvořit pro jednotlivé projekty zvlášť. Jelikož bude tento token využíván pro několik koncových bodů, které vyžadují různá oprávnění, je nutné nastavit tokenu taková oprávnění, aby mohl obsloužit vše potřebné. V praxi to znamená nastavit jeho roli na *Maintainer* a přístup (angl. *access*) na *api*.

2.6.1.1 Získání všech verzí komponenty

Systém Deployment Manager si bude muset u každé komponenty evidovat, jaké verze (resp. názvy *tagů*) repozitář této komponenty poskytuje. K tomu systém využije koncový bod API GitLabu:

Adresa požadavku `{adresa-gitlabu}/api/v4/projects/{project-id}/repository/tags`

HTTP metoda `GET`

adresa-gitlabu URL na instanci aplikace GitLab, tedy například `https://gitlab.com`

project-id ID projektu v aplikaci GitLab [59]

2.6.1.2 Vytvoření *project hooku*

Jelikož si systém bude ukládat vešekeré *tagy* vytvořené v GitLabu, bude zapotřebí vytvořit metody na reakci na zprávu od GitLabu, která sdělí, že byl vydán nový *tag*. Mimo to bude zapotřebí zajistit reakci na úspěšně doběhlou *pipeline* po provedení změny verze, které bylo navrženo v kapitole 2.3.

Aby GitLab věděl, že má zasílat informaci o nově vytvořeném *tagu* či o doběhnuté *pipeline*, je nutné vytvořit tzv. *project hook*. Tím lze nastavit, aby GitLab posílal na určitou adresu informaci o tom, že se něco stalo. [60]

Adresa požadavku `{adresa-gitlabu}/api/v4/projects/{project-id}/hooks`

HTTP metoda `POST`

adresa-gitlabu URL na instanci aplikace GitLab, tedy například `https://gitlab.com`

project-id ID projektu v aplikaci GitLab

Tento požadavek bude volán dvakrát, pokaždé s jiným cílovým URL, jelikož poprvé bude nastaven *project hook* na informaci o novém *tagu*, později pak na informaci o doběhnuté *pipeline*.

Do těla požadavku bude systém vkládat ID projektu a URL, na které mají být informace GitLabem zasílány. Dále se dovnitř vkládá booleovská hodnota popisující, na jaký typ informací má být *project hook* vytvořen. Nakonec je přiložen i token, kterým systém Deployment Manager u přijatého požadavku ověří, zda ho odeslal skutečný GitLab, anebo někdo, kdo se za něj pouze vydává. [61] Zabezpečení API bude popsáno v kapitole 3.4.3.

2.6.1.3 Vytvoření *pipeline trigger tokenu*

Mnou vyvíjená aplikace bude potřebovat měnit verze běžících aplikací. K tomu využije návrh popsaný v kapitole 2.3. Ten říká, že změna verze proběhne přes *pipeline* v aplikaci GitLab. Aby mohla být *pipeline* spuštěna přes API, je k tomu potřeba tzv. *pipeline trigger token*. Ten si lze nechat vygenerovat přes API, což popíši v následujícím požadavku:

Adresa požadavku `{adresa-gitlabu}/api/v4/projects/{project-id}/triggers`

HTTP metoda `POST`

adresa-gitlabu URL na instanci aplikace GitLab, tedy například `https://gitlab.com`

project-id ID projektu v aplikaci GitLab

V těle požadavku je obsaženo ID projektu a popis pro vytvářený token. Tento popis slouží k identifikaci tokenu v přehledu tokenů v GitLabu. [62]

2.6.2 Vystavené API

Z důvodu umožnění reakce na *project hook* je nutné vystavit API, které bude moci GitLab volat.

2.6.2.1 Přijetí informace o vytvořeném *tagu*

Jako odpověď na *project hook* týkající se nově vytvořeného *tagu* bude v systému implementována reakce na následující požadavek:

Adresa požadavku `{adresa-dm}/api/versions`

HTTP metoda `POST`

adresa-dm URL adresa, na které běží Deployment Manager

V těle požadavku GitLab odesílá spoustu informací. [63] Systém si z nich bere pouze název nově vytvořeného *tagu* a z dalších údajů sestaví URL na GitLab. Dále systém čte informaci obsahující ID projektu v aplikaci GitLab. Právě dvojice ID projektu a URL na GitLab jednoznačně definují komponentu v databázi Deployment Manageru. Té pak přiřadí nově vytvořený *tag*.

2.6.2.2 Přijetí informace o doběhlé *pipeline*

Požadavek od GitLabu informující o doběhlé *pipeline* bude odeslán na následující koncový bod:

Adresa požadavku `{adresa-dm}/api/pipelines`

HTTP metoda `POST`

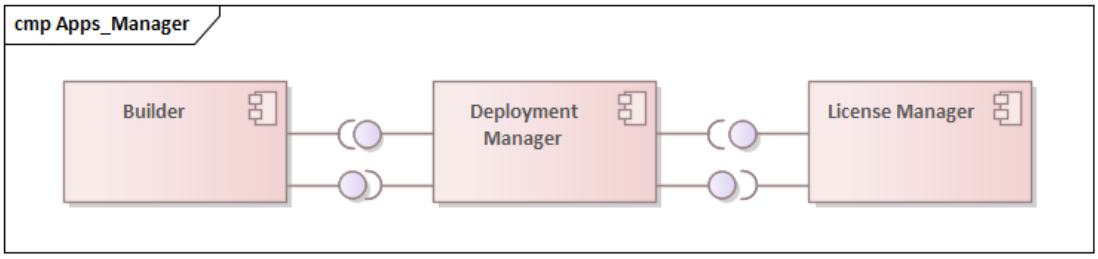
adresa-dm URL adresa, na které běží Deployment Manager

Implementovaná aplikace si přečte informace o doběhnuté pipeline, napasuje ji na měněnou verzi běžící aplikace a upraví svá data tak, aby reflektovala novou verzi. Dále bude zahájena komunikace s aplikací Builder popsaná v podkapitole 2.7.3.2.

2.7 Komunikace v systému Apps Manager

Důležitým bodem vývoje je návrh komunikačního rozhraní mezi všemi komponentami Apps Manageru. Již na prvních schůzích s kolegy a našimi vedoucími jsme se shodli, že ideálním schématem pro komunikaci bude License Manager — Deployment Manager — Builder, neboť právě mnou implementovaný systém zastává roli prostředníka. Tato komunikace mezi komponentami je zachycena na diagramu komponent na obrázku 2.5.

Zvažovali jsme i jiná pořadí, či dokonce možnost, v níž by mohl License Manager komunikovat přímo s Builderem. Tyto možnosti však působily velmi komplikovaně a neprakticky. Ku příkladu zvažovaná varianta License Manager — Builder — Deployment Manager by způsobila to, že by Builder, starající se pouze o sestavování aplikací a dále jejich pozastavování či úplné vypínání, musel řešit, kde běží jaká verze a vše posílat na Deployment Manager.



Obrázek 2.5 Diagram komponent Apps Manageru

Lze namítnout, že změnu verze, kterou momentálně řeší Deployment Manager, by měl dělat Builder, jelikož se jedná o operaci podobnou sestavení aplikace. Takto to bylo dokonce původně v plánu, avšak tuto část Builderu jsem měl řešit já. S kolegy jsme usoudili, že by tento postup byl velice nepraktický, jelikož by dva lidé implementovali jednu aplikaci, na níž probíhá kompletní refaktorizace a vznikl by tím zmatek. Navíc mé zkušenosti s technologií, kterou Builder používá, nejsou příliš velké. Proto jsme tuto funkcionalitu nakonec umožnili pouze ze mnou implementované aplikace.

Jako architekturu API jsme zvolili nám dobře známý REST. Veškerá odesílaná data serializujeme do formátu JSON.

2.7.1 Průběh návrhu

První verze návrhu API mezi komponentami Apps Manageru vznikla v aplikaci Microsoft OneNote, kde jsme při společném online hovoru nás, všech tří studentů, nakreslili tři čtverečky reprezentující naše aplikace a mezi nimi jsme dále kreslili šipky s popisky, co bude která aplikace od jiné potřebovat. V ten moment nikdo z nás neměl promyšlený návrh své aplikace ani provedenou analýzu, proto to pro nás bylo dostačující schéma.

Později, během pracování na našich aplikacích, jsme však zjistili, že je potřeba si API vydefinovat přesně. K tomu jsme, na doporučení našich vedoucích, zvolili aplikaci SwaggerHub.

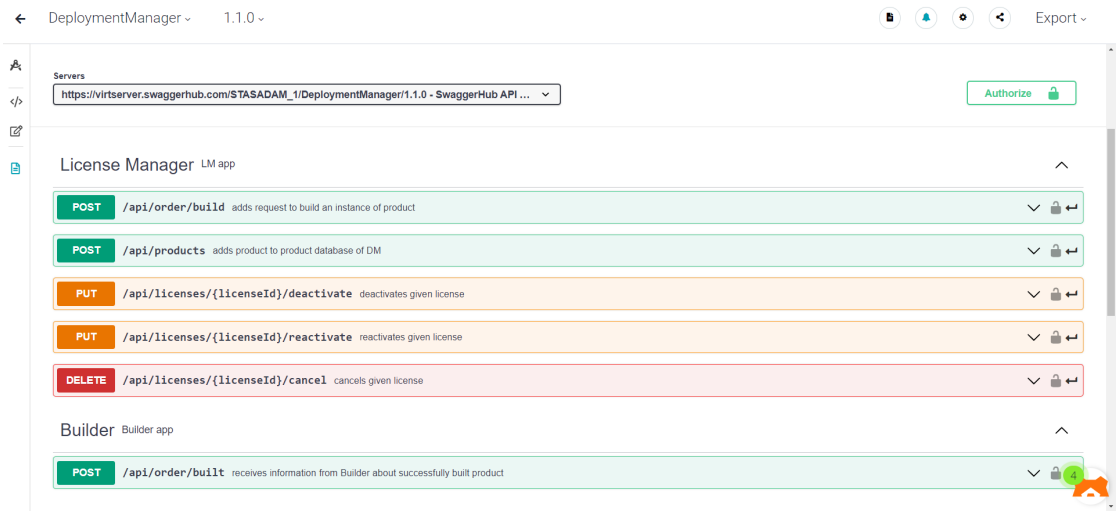
SwaggerHub je aplikace poskytující jednoduchou a intuitivní tvorbu API. Existuje jak verze zdarma, tak verze placená. My jsme si vystačili s tou bezplatnou. Aplikace je v souladu se Swaggerem, dnes označovaným jako OpenAPI specifikace. [64]

SwaggerHub definuje jednotlivá schémata i koncové body pomocí `yaml` souboru. Ač to nemusí působit příjemně, výstup, který SwaggerHub z tohoto souboru vytvoří, naopak působí velice pozitivně. U jednotlivých koncových bodů jsou přehledně barvami odlišeny HTTP metody, jednotlivé cesty si lze rozklikávat, což otevře záložku s parametry, tělem požadavku a návratovými HTTP kódy. Tělo požadavku si lze zobrazit buď jako obecný předpis (který je opět rozkliknutelný), anebo jako ukázkovou hodnotu. Grafické rozhraní aplikace je k nahlédnutí na obrázku 2.6.

Mimo přehledné grafické rozhraní disponuje aplikace také možností vygenerování dokumentace API formou webové stránky v jazyce HTML. Toho jsem na samotném konci procesu definování API využil. Na této webové stránce si lze prohlížet jednotlivé koncové body aplikací systému Apps Manager, zobrazit či skrýt jednotlivá těla požadavků a podobně. Proto je v přiloženém archivu k dispozici kompletní OpenAPI specifikace API celého Apps Manageru.

Upozorňuji, že finální podoba API aplikací License Manager a Builder se mohla změnit, neboť studenti pracující na těchto aplikacích pracovali na jejich vývoji i v době, kdy jsem toto API již testoval. Níže popisované API je v souladu s implementací aplikace Deployment Manager.

U jednotlivých požadavků bude zachována stejná struktura jako u požadavků v kapitole 2.6.



Obrázek 2.6 Grafické rozhraní aplikace SwaggerHub

2.7.2 Komunikace s aplikací License Manager

V této sekci pokryji veškerou komunikaci, která probíhá mezi Deployment Managerem a License Managerem.

2.7.2.1 Přidání produktu do Deployment Manageru

Ač bude mnou vytvářený systém umožňovat přidání produktu přímo uživatelem z grafického rozhraní, s Tomášem Sládkem jsme se dohodli, že umožníme přidání produktu i přes API.

Původně byl našim cílem stav, ve kterém stačí produkt vytvořit pouze v jedné komponentě (License Manager či Deployment Manager) s tím, že do té druhé by se předal přes API. Bohužel toho nelze docílit, neboť LM pro vytvoření produktu vyžaduje popis produktu a jeho konfigurace a DM pro změnu vyžaduje definici komponent, z nichž se produkt skládá. Umožnil jsem nicméně v DM tvorbu produktů bez komponent, které může uživatel přiřadit později. Dodám, že v aplikaci License Manager je možné produkt po vytvoření skrýt, tedy než se v DM dodefinují potřebné komponenty, produkt nebude možné zakoupit.

Požadavek na přidání produktu do databáze mé aplikace vypadá následovně:

Adresa požadavku `{adresa-dm}/api/products`

HTTP metoda `POST`

adresa-dm URL adresa, na které běží Deployment Manager

V těle požadavku LM poskytne ID vytvořeného produktu a jeho název.

2.7.2.2 Žádost o sestavení produktu

Jakmile si zákazník používající aplikaci License Manager objedná nějaký produkt, Deployment Manager obdrží žádost o sestavení tohoto produktu. K tomu slouží následující koncový bod:

Adresa požadavku `{adresa-dm}/api/order/build`

HTTP metoda `POST`

adresa-dm URL adresa, na které běží Deployment Manager

Systém License Manager v těle požadavku poskytne ID objednávky, jež má být sestavena. Objednávka dále obsahuje konfiguraci, která se skládá ze svého číselného identifikátoru a z produktu. Produkt má taktéž číselné ID, dále má název. Konfiguraci lze v budoucnu přidat například počet poboček či další vlastnosti specifické pro daného zákazníka.

Kromě ID objednávky a její konfigurace jsou poskytnuty i údaje o zákazníkovi, a sice jeho ID (vzniká v License Manageru při registraci zákazníka) a jméno.

Deployment Manager si po přijetí takového požadavku uloží zákazníka, pokud uložený ještě nebyl. Aplikace nebude umožňovat přidání zákazníka skrz grafické rozhraní. Dále si systém zaeviduje ID objednávky a ID zákazníka, aby mohl později u sestavené objednávky správně identifikovat, která či které aplikace byly sestaveny.

Poté bude zahájena komunikace s aplikací Builder, kterou podrobně popíši v podkapitole 2.7.3.1.

2.7.2.3 Správa licencí

Systém má připravené API i na správu licencí. Konkrétně se jedná o jejich deaktivaci, reaktivaci a úplné zrušení. Ač jsme se během společného návrhu API s kolegy dohodli, že správu licencí nebudeme implementovat, přidal jsem alespoň prázdné metody čekající na pozdější implementaci. Nyní popíši, jak vypadají jejich koncové body.

Deaktivace licence:

Adresa požadavku `{adresa-dm}/api/licences/{licenseId}/deactivate`

HTTP metoda `PUT`

adresa-dm URL adresa, na které běží Deployment Manager

licenseId ID licence, jež má být deaktivována

Reaktivace licence:

Adresa požadavku `{adresa-dm}/api/licences/{licenseId}/reactivate`

HTTP metoda `PUT`

adresa-dm URL adresa, na které běží Deployment Manager

licenseId ID licence, jež má být reaktivována

Úplné zrušení licence:

Adresa požadavku `{adresa-dm}/api/licences/{licenseId}/cancel`

HTTP metoda `DELETE`

adresa-dm URL adresa, na které běží Deployment Manager

licenseId ID licence, jež má být zrušena

2.7.2.4 Informování LM o dokončení sestavování produktu

Jakmile aplikace Builder dokončí sestavování produktu, je její povinností o tom informovat Deployment Manager. Tento krok bude popsán později v podkapitole 2.7.3.3.

Nyní popíši koncový bod License Manageru, na nějž se Deployment Manager následně obrací a předává mu informace o sestaveném produktu.

Adresa požadavku `{adresa-lm}/api/order/built`

HTTP metoda `PUT`

adresa-lm URL adresa, na které běží License Manager

V těle je obsažena kompletní objednávka tak, jak byla popsána v sekci 2.7.2.2. K ní je přiloženo pole dvojic popisujících URL adresy, na které Builder nasadil aplikace. Tyto adresy se skládají z názvu typu (například *android* či *web*) a ze samotné adresy. Na některých adresách se nenachází přímo běžící aplikace, ale instalační soubor ke stažení. To je třeba u adres pro *android*.

Dále Deployment Manager vkládá do těla další pole dvojic, tentokráte se jedná o nasazené aplikace. To je přeposíláno jen pro jednoduchost implementace aplikace DM, která toto pole aplikací zpracovává. License Manager ho pro své fungování nevyžaduje. Toto pole bude popsáno v podkapitole 2.7.3.1.

2.7.3 Komunikace s aplikací Builder

Zde popíši veškeré koncové body, skrz ně komunikují systémy Deployment Manager a Builder.

2.7.3.1 Žádost o sestavení produktu

Jakmile Deployment Manager obdrží od License Manageru zprávu, že má být sestaven produkt, přeposílá ji na Builder. Ten k tomu má nachystaný následující koncový bod:

Adresa požadavku `{adresa-builderu}/api/license/build`

HTTP metoda `POST`

adresa-builderu URL adresa, na které běží Builder

V těle je přeposlána celá objednávka popsaná v sekci 2.7.2.2. K ní je připojeno pole dvojic *tagů*. Každá dvojice se skládá z ID komponenty evidované v Deployment Manageru a z názvu *tagu*, který má být sestaven. Touto informací se Builder dozvídá, jakou verzi aplikace má nasadit. Momentálně je předávána vždy nejnovější možná verze. Toto pole obsahuje tolik prvků, kolik je komponent v objednaném produktu.

Dalším připojeným polem jsou dvojice identifikátoru komponenty a identifikátoru prostředí zákazníka. Jak jsem již naznačil v kapitole 2.3, Deployment Manager pro každou nasazovanou aplikaci vygeneruje unikátní *slug*. Dvojice se skládají z ID komponenty, které je totožné jako v předchozím odstavci, a z řetězce obsahujícího právě *slug*. Tento pak slouží i jako identifikátor běžící aplikace v komunikaci Deployment Manageru a Builderu při odeslání požadavku s informací o provedené změně verze na Builder.

2.7.3.2 Informování Builderu o provedené změně verze

Protože si Builder eviduje, jaká aplikace běží v jaké verzi, je nutné ho po změně verze o této skutečnosti informovat. Požadavek má následující tvar:

Adresa požadavku `{adresa-builderu}/api/license/version`

HTTP metoda `PUT`

adresa-builderu URL adresa, na které běží Builder

V těle požadavku DM odesílá pole verzí, které se skládá z dvojice *slug* a název *tagu*. Jak jsem již zmiňoval v podkapitole 2.7.3.1, *slug* slouží k identifikaci běžící aplikace. Název *tagu* říká, jaký *tag* byl změnou verzí nasazen.

2.7.3.3 Informování o dokončení sestavování aplikace

Ve chvíli, kdy Builder sestaví aplikaci, posílá tuto informaci na Deployment Manager. Ten má k tomu připravený následující koncový bod:

Adresa požadavku `{adresa-dm}/api/order/built`

HTTP metoda `POST`

adresa-builderu URL adresa, na které běží Deployment Manager

Požadavek se skládá z objednávky popsané v sekci 2.7.2.2, z polí obsahujících URL adresy popsané v podkapitole 2.7.2.4 a z pole dvojic identifikátoru a *slugu* aplikace popsaného v sekci 2.7.3.1.

Deployment Manager na základě přijetí tohoto požadavku posílá zprávu License Manageru, který je díky ní schopen uložit si URL adresy a zobrazit je zákazníkovi.

2.8 Uživatelské rozhraní

Protože bude Deployment Manager disponovat grafickým uživatelským rozhraním, věnuji tuto kapitolu popisu jeho tvorby.

Jak jsem zmínil v kapitole 1.3.4, vývoj jsem prováděl agilně. Proto jsem jednotlivé obrazovky, které uživatel v aplikaci uvidí, navrhoval vždy po jedné. Jakmile jsem navrženou obrazovku zkonstruoval, teprve poté jsem začal navrhovat další.

Obrazovku jsem si nejprve načrtl na papír. Mimo pozice a barvy tlačítek a textových polí jsem si do náčrtu také poznamenával, jaká data budou zobrazena, jaké metody musím svým modelům vytvořit, aby obrazovka dokázala zobrazit správná data, co bude potřeba ošetřit ve formuláři, který je součástí dané obrazovky, a podobně. Návrh tedy sloužil primárně pro mě a pro jiné čtenáře nebyl příliš přehledný. Když jsem zadavateli během vývoje aplikaci prezentoval, vždy viděl obrazovku přímo v prototypu aplikace. Tvorba jednotlivých obrazovek je v Laravelu za použití šablonovacího nástroje Blade a frameworku Bootstrap jednoduchá a vzhled stránky lze rychle změnit. Právě proto jsem se během vývoje nezdržoval představováním svých návrhů zadavateli.

Dalším důvodem pro nevytváření například drátěného modelu a pro nekonzultování mých návrhů se zadavatelem byl fakt, že systém Deployment Manager není určený pro koncového zákazníka, nýbrž pro administrátora. U něj se nepředpokládají velké nároky na uživatelské rozhraní, naopak je preferována správná funkcionalita aplikace.

Kapitola 3
Implementace

Po pečlivě provedeném návrhu následuje implementace. Jedná se o hlavní část práce softwarového vývoje. Současně jde o proces, který jako jediný ze všech procesů týkajících se vývoje softwaru nemůže být vynechán. Jeho cílem je dodání produktu, jenž splňuje požadavky vytvořené během analýzy, neobsahuje chyby a je jednoduše použitelný. [65] V této kapitole popíši, jak konstrukce Deployment Manageru probíhala.

Nejdříve popíši vývojové prostředí, v němž jsem veškerý vývoj provedl. Poté se budu zabývat popisem architektury MVC doplněným o s ním související činnosti či technologie. Zmíním použití objektově relačního mapování i technologií Bootstrap a JavaScript. U controllerů mimo jiné uvedu, jak jsem prováděl validaci vstupu od uživatele. Dále budou popsány výčty a služby, u kterých se zaměřím na ty nejdůležitější. Ty se zabývají automatickým doporučováním verzí, změnou verze aplikací a zajišťují některou komunikaci v rámci Apps Manageru. V neposlední řadě okomentuji zabezpečení jednak API, jednak celé aplikace. Poslední sekce bude věnována frontám.

3.1 Vývojové prostředí

Před započetím implementace bylo zapotřebí nainstalovat si do počítače vše potřebné pro vývoj aplikace v Laravelu. Jak jsem zmínil v kapitole 2.1.3, mnou vybraný framework disponuje možností vývoje v Dockeru. Jelikož jsem Docker na svém počítači již měl zprovozněný, byla to pro mě první volba. Kód jsem psal v aplikaci PhpStorm.

3.1.1 Docker

Nástroj Docker je příkladem tzv. kontejnerizace. Ta přišla jako řešení nedostatků virtualizace neboli vytváření virtuálních zařízení uvnitř jednoho fyzického zařízení. Jejím hlavním znakem je to, že jeden virtuální počítač se chová jako samostatný server s vlastním operačním systémem. Zpravidla ale procesy, pro jejichž chod se virtuální počítač tvoří, nevyžadují celý operační systém, naopak jim stačí jen jeho část. Tím vzniká zbytečná režie.

Kontejnerizace virtualizuje jádro operačního systému, čímž mohou všechny kontejnery běžet v rámci jednoho operačního systému, a tedy sdílet paměť a další potřebné zdroje. Tím nevzniká taková režie jako u virtualizace. Další důležitou výhodou kontejnerizace je možnost kontejnery izolovat a nasadit je v jiném prostředí.

Tím Docker zjednodušuje mimo jiné i nasazení aplikace po vývoji na lokálním zařízení na server, jelikož se stačí připojit na vzdálený server, nainstalovat Docker a stáhnout tzv. *image*,

což je „šablona“, z níž Docker vytvoří výsledné kontejnery. Díky *image* Docker ví, jaké knihovny či další soubory jsou pro správný chod aplikace potřebné. [66]

I když předmětem této práce není nasadit vytvořený prototyp aplikace na server, bude vývoj pomocí Dockeru sloužit alespoň pro usnadnění budoucího vývoje.

Instalace Laravelu s pomocí Dockeru byla velmi prostá. Stačilo se řídit dokumentací a během chvíle mi lokálně běžela webová aplikace.

3.1.2 PhpStorm

Dalším rozhodnutím, kterým jsem se zabýval, byl výběr vhodného softwaru pro psaní a úpravy kódu. Jelikož mám za celé studium velmi pozitivní zkušenost s IDE (integrovanými vývojovými prostředími) společnosti JetBrains, zvolil jsem jejich PhpStorm.

Mezi výhody PhpStormu patří mimo spoustu dalších i možnost pohodlného verzování kódu bez nutnosti psaní příkazů do příkazového řádku. Protože jsem o svůj kód nechtěl přijít v případě poruchy na svém počítači, používal jsem verzovací systém git a v tomto textu již zmíněnou aplikaci GitLab. Verzoval jsem po celou dobu vývoje na GitLabu zadavatele.

3.2 Architektura MVC

V kapitole zabývající se návrhem jsem zmínil, že Laravel využívá architekturu MVC. Nyní popíši, jak její tři části fungují implementačně. Popis těchto částí rozšířím o procesy či technologie, které s nimi souvisí.

3.2.1 Model

Všechny modely v Laravelu dědí ze třídy `Model`. Jeden takový model odpovídá jedné třídě na diagramu tříd nacházejícímu se v příloze B.

3.2.1.1 Objektově relační mapování

Objektově relační mapování poskytované Eloquentem umožňuje programátorovi psát kód pro operace s daty v databázi v jazyce PHP. Na pozadí se tento kód převádí na příslušný kód v dotazovacím jazyce SQL.

Dále je v Laravelu využitý Query Builder pro tvorbu dotazů buď v jazyce SQL, anebo ve své syntaxi, která je pro všechny podporované databáze stejná. [67] Tvorba dotazů v čistém SQL je vhodná zejména pro pokročilé dotazování. Pokud se tedy vývojář vyhne složitým dotazům, za celý vývoj aplikace nemusí napsat žádný kód v jazyce SQL.

Detailněji popíši Eloquent v následující podkapitole.

3.2.1.2 Modelová třída

Jedné databázové tabulce (mimo tabulky popisující pouze vazbu mezi entitami) odpovídá právě jedna modelová třída.

Jelikož Laravel využívá Eloquent, je zapotřebí každému modelu přidat metody, které pak budeme moci volat namísto přímého vstupu do databáze. Tyto metody je zvykem pojmenovávat názvem takového modelu, který lze pomocí nich získat. Pokud metoda popisuje vazbu s vícero instancemi nějakého modelu, název se píše v množném čísle.

Metody popisující vztahy mezi entitami mohou mít následující návratové typy:

- HasOne
- BelongsTo

- HasMany
- BelongsToMany [68]

Jedno z možných použití jsem zachytil ve výpisu kódu 3.1. Tato ukázka obsahuje metodu `appGroups()` s návratovým typem `BelongsToMany`. To znamená, že třída `Application`, v níž se tato metoda nachází, může náležet mnoha instancím třídy `AppGroup`.

Nutno dodat, že zavolání takové metody nevrátí přímo kolekci příslušných modelů. Namísto ní je navrácen typ dědící z typu `Relation`, na který lze buď dále řetězit dotazy, anebo lze metodou `get()` získat kolekci příslušných instancí. [68]

Výpis kódu 3.1 Metoda popisující vztah tříd Application a AppGroup

```
    class Application extends Model
{
    public function appGroups(): BelongsToMany
    {
        return $this->belongsToMany(AppGroup::class);
    }
}
```

Co se atributů jednotlivých entit týče, do modelové třídy se nevkládají. Díky tomu, že modelovou třídu pojmenujeme stejně jako odpovídající databázovou tabulku s jediným rozdílem, že tabulky se pojmenovávají v množném čísle, Laravel daný model automaticky propojí s příslušnou tabulkou. Poté lze k atributům přistupovat pomocí jejich názvu. [56]

Dále lze u modelů vytvářet různé metody, které je možné v kódu volat.

3.2.2 View

Obsah Deployment Manageru bude uživateli přístupný skrze jeho webový prohlížeč. Proto je zapotřebí v kódu softwaru vydefinovat podobu HTML stránek, které jsou uživateli zobrazovány.

View odděluje aplikační logiku aplikace od logiky prezentační. Protože by bylo nepraktické používat čisté HTML, šablony pro views se tvoří za pomocí šablonovacího jazyka Blade. Ten umožňuje například zavolání kódu v PHP pomocí syntaxe `{{ phpkod }}`. Často se tato funkcionalita používá k zobrazování parametrů, které danému view předal controller. [69, 70]

Každý view je uložen v adresáři `resources/views`. Pro přehlednost jsem jednotlivé views rozdělil ještě do podadresářů, které jsem pojmenoval podle modelové třídy, s níž je daný view úzce spjatý.

Jednotlivé views jsou také pojmenované podle modelu, s nímž souvisí. Současně jejich název vždy končí příponou `.blade.php`. Například šablona pro obrazovku s přehledem všech komponent daného produktu je umístěna v adresáři `resources/views/component` a má název `components.blade.php`.

Blade dále nabízí direktivy pro podmínkové toky. Například pro `if` lze použít `@if`, `@elseif`, `@else` a `@endif`. [70] Toho jsem využil při zobrazování chyb, které controller během validování vstupu uživatele nalezl. Pomocí `@if` jsem se dotázal, zda velikost pole v proměnné `@errors` je větší než nula a pokud toto tvrzení bylo platné, chyby jsem vypsal.

Mimo podmínkové toky zmíním i další mnou používané direktivy tohoto šablonovacího nástroje, a to `@foreach` a `@endforeach` uvozující běžný `foreach` cyklus. Pomocí nich jsem ve spoustě views generoval tabulky zobrazující data uživateli.

Výsledné obrazovky si lze prohlédnout v příloze C.

3.2.2.1 Bootstrap

Pro zajištění příjemného vzhledu jednotlivých obrazovek jsem použil Bootstrap. Jedná se o knihovnu obsahující předem navržená tlačítka, karty, navigační lišty a mnoho dalšího. Prvky nabízené knihovnou Bootstrap jsou vytvořeny za pomocí HTML, CSS a JavaScriptu. [71]

Často jsem využíval modály, jejichž zpracování knihovna Bootstrap také nabízí. Modály jsou dialogová okna, jež se otevřou na popředí aktuálně zobrazené obrazovky. Lze je zavřít kliknutím do pozadí a jejich obsahem často bývají různé formuláře či upozornění.

3.2.2.2 JavaScript

V některých obrazovkách jsem použil JavaScript. Zajišťuji pomocí něj mimo jiné i doporučování kompatibilní verze v obrazovce týkající se změny verze aplikace.

Zde musí uživatel nejprve zvolit verzi, na niž chce přejít. Touto volbou se zpřístupní druhý HTML *select*, v němž si uživatel vybere, pro jakou aplikaci si chce nechat doporučit vhodnou verzi. Bez použití JavaScriptu by bylo nutné nejprve odeslat formulář se zvolenou verzí a na ten vrátit jako odpověď obrazovku, v níž by byl zpřístupněný druhý *select* s volbou aplikace.

Následuje samotné doporučení vhodné verze, které funguje přes AJAX. Jedná se o techniku umožňující přístup k serveru bez nutnosti znovu načíst webovou stránku. [72] Uživatel v *selectu* zvolí aplikaci, pro níž si chce nechat doporučit vhodnou verzi. AJAX provede příslušný dotaz na server a jakmile je mu vrácena odpověď, může například vypsat přijatá data. V mém případě je vypisována verze, kterou systém doporučil.

3.2.2.3 DataTables

Jako poslední využití JavaScpritu během implementace Deployment Manageru zmíním DataTables. To je *plugin* pro JavaScriptovou knihovnu jQuery. K obyčejným HTML tabulkám přidává například stránkování, řazení dle sloupců a vyhledávání ve sloupci či v celé tabulce. [73] Tuto možnost jsem zvolil především z důvodu splnění té části požadavku F1, jež se týká filtrování v seznamu aplikací.

DataTables disponují dvěma možnostmi zpracování dat. Tou první je zpracování v prohlížeči uživatele. Znakem tohoto přístupu je načtení všech dat v tabulce při přístupu na stránku. To může způsobit zdlouhavé načítání stránky v případě, že tabulka obsahuje statisíce řádků. Výhodou je, že jakmile se data načtou, uživatel může tabulku filtrovat, vyhledávat v ní apod. a nečeká se na odpovědi serveru, jako tomu je u následující možnosti.

Druhou možností je odesílání AJAX požadavku na server při každé uživatelově změně zobrazení tabulky. Například když uživatel zvolí seřazení tabulky dle identifikátoru jejich položek, je odeslán požadavek na server, který vrátí seřazená data. Pokud je v tabulce nastaveno stránkování, načte se vždy jen tolik řádků, kolik se jich vejde na jednu stránku. Tento přístup je výhodný ve chvíli, kdy tabulka obsahuje velké množství řádků a uživatel si chce zobrazit jen pár z nich. Nevýhodou však je, že každý AJAX požadavek zabere množství času, které závisí na vytíženosti serveru.

Autoři *pluginu* doporučují zvolit první přístup v případě, že velikost tabulky nepřesahuje deset tisíc řádků. Pokud má tabulka nad 100 tisíc řádků, doporučují přístup pomocí AJAX. [74]

Jelikož jsem v době implementace neznal přesné hodnoty uvedené v dokumentaci, pro jistotu jsem zvolil přístup druhý.

Obrazovku s přehledem nasazených aplikací, která DataTables používá, si lze prohlédnout v příloze C.

3.2.3 Controller

Konkrétní data do views předávají controllery. To jsou třídy dědící ze třídy `Controller`.

3.2.3.1 Koncové body

V souborech `routes/web.php` a `routes/api.php` jsou definované jednotlivé koncové body, které aplikace nabízí. Soubor `web.php` obsahuje koncové body pro aplikaci jako takovou. Slouží tedy pro specifikování toho, co se má dít po interakci uživatele s webovou částí aplikace. V souboru `api.php` jsou obsaženy koncové body, které aplikace nabízí v rámci svého API.

U každé specifikace koncového bodu se uvádí HTTP metoda a URI (cesta, jež může obsahovat parametry ve složených závorkách). Poté je uveden controller, který má požadavek na daný koncový bod zpracovat a jeho metoda, která má být zavolána. Pro pohodlné definování, jaký koncový bod má být při dané akci zavolán, nabízí Laravel metodu `Route::name`, pomocí níž je možné koncový bod pojmenovat. [75] Definici koncového bodu si lze prohlédnout ve výpisu kódu 3.2.

Výpis kódu 3.2 Definice koncového bodu pro odstranění skupiny aplikací

```
Route::delete('/appgroups/{appgroup}', [AppGroupController::class,
'removeAppGroup']) -> name('removeAppGroup');
```

Ve chvíli, kdy uživatel klikne na tlačítko, které ho přesměruje na jinou obrazovku nebo kterým odešle data zadaná do formuláře, přistupuje na nějaký koncový bod. Ten, pokud je definován v jednom ze dvou výše zmíněných souborů, je zachycen příslušným controllerem, v němž je zavolána vývojářem zvolená metoda.

3.2.3.2 Validace vstupu

Pokud zadával uživatel nějaká vstupní data, v controlleru zpravidla probíhá jejich validace. K tomu v Laravelu slouží metoda `Request::validate`. Jako parametr jí lze předat pole dvojic klíč – hodnota, kde klíčem je název položky v požadavku a hodnotou je řetězec skládající se z různých omezení, která musí platit, aby byla validace splněna. Pokud neplatí, Laravel uživatele přesměruje zpět a danému view předá v parametru seznam chyb. [76] Validaci jsem zachytil ve výpisu kódu 3.3.

Výpis kódu 3.3 Validace vstupu v controlleru

```
$request->validate([
      'name' => 'required|unique:products',
      'license_manager_id' => 'required|unique:products|numeric'
  ]);
```

Po úspěšné validaci se zpravidla provádí čtení či zápis dat do databáze a poté probíhá přesměrování na daný koncový bod či je navrácen konkrétní view s konkrétními parametry.

Pro validaci lze použít i vlastní validační pravidla. Ta jsem vkládal do adresáře `app/Rules`. Jelikož bylo zapotřebí v určitých místech zkontrolovat, zda uživatel zadal validní výraz ve formátu sémantického verzování a jelikož toto pravidlo Laravel neobsahuje, rozhodl jsem se, že vytvořím pravidlo vlastní.

Toto pravidlo je zachyceno v ukázce 3.4. Metoda `__invoke()` obsahuje kód, který je vykonán ve chvíli, kdy je toto validační pravidlo zavoláno. V této metodě lze specifikovat i chybovou hlášku, která má být vypsána, když je pravidlo nesplněno. Třída `Constraint` je z knihovny pro sémantické verzování, kterou popíši v sekci 3.4.1. V tomto pravidle se o veškerou logiku okolo formátu *SemVer* stará právě ona.

Výpis kódu 3.4 Definice validačního pravidla

```
use z4kn4fein\SemVer\Constraints\Constraint;

class SemVer implements InvokableRule
{
    public function __invoke($attribute, $value, $fail)
    {
        $constraint = Constraint::parseOrNull($value);
        if($constraint === null) {
            $fail('The :attribute must be valid semver specification.');
        }
    }
}
```

3.3 Výčty

Stavy modelu aplikace, které jsem již zmínil v sekci 2.5.2, jsem v kódu realizoval pomocí výčtů. Protože ale jazyk PHP nabízí výčty (*Enums*) až od verze 8.1 [77] a já během vývoje používal verzi 8.0, bylo nutné si výčty obstarat jiným způsobem.

Pro realizaci stavů jsem zvolil knihovnu *laravel-enum* vytvořenou Benem Sampsonem. K dubnu roku 2023 má tato knihovna přes 6,7 milionů stažení. [78]

Samotné stavy aplikací jsem vložil do třídy `ApplicationState`, kterou jsem umístil do mnou vytvořeného adresáře `app/Enums`. Třída dědí z `BenSampo/Enum` a je zachycena ve výpisu 3.5.

Protože jsem během vývoje zjistil, že verze 8.0 jazyka PHP je zastaralá, přešel jsem na verzi 8.2. Ta již výčty nabízí bez nutnosti stahování externích knihoven, pro jednoduchost jsem však ponechal ve své implementaci výčty pomocí zmíněné knihovny.

Výpis kódu 3.5 Výčet stavů modelu aplikace

```
use BenSampo\Enum\Enum;

final class ApplicationState extends Enum
{
    public const DEPLOYMENT_IN_PROGRESS = 'deploymentInProgress';
    public const RUNNING = 'running';
    public const VERSION_CHANGE_IN_PROGRESS = 'versionChangeInProgress';
}
```

3.4 Služby

V adresáři `app/Services` lze nalézt třídy, které pomáhají modelům s určitými procesy nad daty. Zpravidla jsem do služeb zařadil takové operace, které nebyly jen pouhým přístupem do databáze. Dále jsem do služeb umístil veškeré metody starající se o komunikaci s aplikacemi GitLab, License Manager (`LicenseManagerService`) a Builder (`BuilderService`).

V této sekci popíši některé služby starající se o procesy stěžejní pro Deployment Manager.

3.4.1 Automatické doporučování verzí

Pro implementaci automatického doporučování verzí bylo potřeba zvolit vhodnou knihovnu do jazyka PHP, která mi umožní pracovat se znaky sémantického verzování. Ačkoli jsem knihoven

našel více, žádná z nich, kromě jedné, neměla ve své dokumentaci přehledně popsané veškeré funkce, které jsem od knihovny potřeboval.

Zmíněnou výjimku tvořila knihovna *php-semver*, jejíž autorem je Peter Csajtai, v aplikaci GitHub (v níž se nachází i dokumentace ke knihovně) vystupující pod přezdívkou *z4kn4fein*. Tato knihovna obsahuje velké množství funkcionalit pokrývající vytvoření verze v sémantickém formátu, porovnání takových verzí i seřazení pole verzí. Mimo samotné verze knihovna nabízí i tzv. *constraints* neboli omezení.

Pro definici omezení lze použít nejen znaky menší než, větší než, rovná se a jejich kombinace, ale i rozsahy psané pomocí znaku `-` či další znaky, jimž jsem se věnoval v kapitole 1.5.5. [79]

Ve třídě `VersionService` jsem vytvořil metodu `recommend()`, která díky zmíněné knihovně doporučí pro zadanou verzi a komponentu nejnovější kompatibilní verzi.

3.4.2 Změna verze aplikací

Proces změny verze jsem navrhl v kapitole 2.3. Nyní popíši, jak jsem zmíněný návrh implementoval.

Kód pro změnu verze aplikace jsem umístil do služby `ApplicationService`. Změna verze skupiny aplikací je prováděna ve třídě `AppGroupService`. V těchto třídách se nejprve připraví data pro odeslání požadavku pro spuštění *pipeline* na GitLabu, poté je požadavek odeslán. Změna verze jedné aplikace a skupiny se liší zejména přístupem ke vkládání parametrů do hlavičky požadavku.

3.4.2.1 GitLab

Nejprve zjednodušeně popíšu, jak Builder sestavuje aplikace. Se studentkou pracující na aplikaci Builder jsme si ujasnili, že se implementace procesu nasazení nové aplikace příliš nezmění. Budu tedy vycházet z její aktuální podoby.

Soubor `.gitlab-ci.yml`, který je obsažen v každém produktu spravovaným Apps Managerem, vychází ze šablony, která umožňuje Builderu sestavovat aplikace. Nejprve jsou provedeny činnosti jako sestavení aplikace, její otestování a další (to závisí na sestavované aplikaci). Poté se spouští tzv. *deploy job*.

Ten je v `.gitlab-ci.yml` definován pro každého zákazníka. Takový *job* spustí předem definovaný skrytý *job*, kterému předá pomocí atributu *environment* název prostředí, do kterého má být aplikace nasazena. Aby se proces spouštěl vždy jen pro konkrétního zákazníka, používá se klíčové slovo `rules`. Tím lze definovat pravidla, kdy se má daný *job* spustit. [80]

Na tento postup jsem navázal a pravidla upravil tak, aby umožnila Builderu nasazovat nové aplikace a současně umožnila Deployment Manageru provádět změnu verze. Podobu činnosti s označením *deploy job* pro konkrétního zákazníka před úpravou lze vidět ve výpisu kódu 3.6. Ve výpisu 3.7 jsem zachytil *deploy job* s novými pravidly.

Výpis kódu 3.6 Podoba konkrétního deploy jobu před úpravou

```
frantisek-novak:
  <<: *deploy_job
  environment:
    name: "frantisek-novak"
  rules:
    - if: $CI_PIPELINE_SOURCE!="pipeline" || $DEPLOY=="frantisek-novak"
```

Nová pravidla nyní vysvětlím. Původní pravidlo nastavilo *job* tak, že se spustil jen v případě, kdy byla hodnota proměnné `$DEPLOY` nastavena na identifikátor, kterým byl daný *job* pojmenován, anebo v případě, když byl vydán nový *tag*, přidána nová data do repozitáře a podobně.

Výpis kódu 3.7 Podoba konkrétního deploy jobu po úpravě

```
frantisek-novak:
  <<: *deploy_job
  environment:
    name: "frantisek-novak"
  rules:
     - if: $DEPLOY=="frantisek-novak" || $frantisek-novak=="true"
```

Mým cílem bylo zachovat původní fungování s přidaným pravidlem, které říká, že se *job* spustí, pokud je proměnná `$frantisek-novak` nastavená na `true`. Aby se původní funkce zachovala, toto nové pravidlo jsem přidal k těm stávajícím pomocí logického *OR*.

Protože by pouhé přidání pravidla `$frantisek-novak == "true"` nezabránilo automatickému spouštění *pipeline* v případě, že byl vydán nový *tag* (tedy nová verze aplikace), odebral jsem pravidlo související s `$CI_PIPELINE_SOURCE`, které toto spouštění povolovalo. Automatickému spouštění *pipeline* jsem chtěl zabránit z důvodu, že by se tím narušil aktuální stav nasazených verzí uložený v Deployment Manageru. Pro jeho správné fungování je totiž nutné provádět veškeré změny verze přímo z něj, nikoli automaticky například přes *push* do GitLab repozitáře.

Je nutné upravit fungování Builderu tak, aby kód, který vkládá do `.gitlab-ci.yml` skrz *pipelines* jednotlivých produktů, odpovídal nově nastaveným pravidlům. Dále názvy jednotlivých procesů typu *deploy job* budou nahrazeny identifikátory označenými jako *slug*, které jsou předávány v rámci API Apps Manageru.

3.4.3 Komunikace s GitLabem

V této sekci popíši služby, které zajišťují komunikaci s aplikací GitLab. Celkově se jedná o dvě třídy, a sice `GitLabService` a `ComponentService`.

Ta první slouží k ošetření, zda Deployment Manager nekomunikuje s falešným GitLabem. To je zajištěno, jak jsem popsal v kapitole 2.6.1.2, tokenem. Tento je odesílaný v hlavičce každého požadavku od GitLabu na systém. Služba nejprve zkontroluje, zda je v hlavičce požadavku obsažena položka s názvem `X-GitLab-Token` a pokud tam je, zkontroluje i její hodnotu. Tu porovná s tokenem uvedeným v konfiguračním souboru, do nějž se načte ze souboru `.env`. Metoda `checkToken()`, která se o výše uvedený proces stará, je volána při každé komunikaci z GitLabu do systému.

`ComponentService` zajišťuje vše potřebné pro uložení vytvořené komponenty do databáze systému. Mezi její metody patří `getAllTags()` pro získání všech *tagů* daného projektu v GitLabu. Dále je volána metoda `setupProjectHook()`, která vytvoří *project hook*. Tato metoda je v controlleru `ComponentController` volaná dvakrát: poprvé pro vytvoření *project hooku* ohledně nově vytvořeného *tagu* v projektu, podruhé pro umožnění systému reagovat na doběhnutou *pipeline*. Poslední metodou této služby je `createPipelineTriggerToken()`, jež obstarává vytvoření *pipeline trigger tokenu*.

3.4.4 Komunikace v rámci Apps Manageru

Veškerý kód zajišťující odesílání požadavků z Deployment Manageru aplikacím License Manager a Builder jsem umístil do příslušných služeb. Zahájení komunikace s License Managerem je prováděno třídou `LicenseManagerService`. S Builderem je komunikace započata ve službě `BuilderService`.

Nyní jednoduše popíši jednotlivé metody. Podrobněji je API popsané v kapitole 2.7.

`LicenseManagerService` disponuje metodou `orderBuilt()`, která mimo jiné přepošle License Manageru informaci přijatou Builderem. Tato přeposílaná zpráva obsahuje informace o aplikaci sestavené Builderem.

Třída `BuilderService` obsahuje metodu `buildProduct()`. Ta požádá Builder o sestavení produktu, o jehož zakoupení byl Deployment Manager informován.

Dále je v této třídě obsaženo volání Builderu související s provedenou změnou verze aplikace. Zajišťuje ho metoda `versionChanged()`.

Ke správnému fungování API využívá Deployment Manager fronty. Ty popíši v sekci 3.6. Dále uvedu, že API je zabezpečeno BearerTokenem. Tuto implementaci popíši v kapitole 3.5.1.

3.4.4.1 Ladění API

Protože během vývoje Deployment Manageru probíhal vývoj s ním komunikujících aplikací, nemohl jsem svoji implementaci API ladit přímo na těchto aplikacích. Proto jsem sáhl po aplikacích umožňující odchytávat příchozí požadavky a odesílat požadavky na můj počítač, na kterém veškerý vývoj probíhal.

Pro kontrolu, zda do mým systémem odesílaných požadavků vkládám správná data ve správném formátu jsem používal aplikaci Webhook.site. Stačilo otevřít webovou stránku této aplikace a okamžitě mi byl vygenerován unikátní odkaz, který jsem vložil do služeb komunikujících s License Managerem a Builderem. Aplikace veškerou komunikaci zachytila a pro každý přijatý požadavek zobrazila veškeré detaily.

Pro vyzkoušení reakce na příchozí požadavky jsem užíval aplikaci Postman. Aby mohly požadavky chodit na můj lokální stroj, bylo nutné stáhnout si Postman Desktop Agenta. V této aplikaci jsem si nadefinoval několik požadavků, které jsem si uložil. Ke každému z nich jsem uvedl mimo jiné URL adresu, na kterou má být požadavek zaslán, co má být obsaženo v jeho těle i v jeho hlavičce. Tělo a hlavičku jsem si pro každý požadavek upravoval podle toho, jaká data by dle vydefinovaného API měl Deployment Manager obdržet.

3.5 Middleware

Middleware je mechanismus, jenž zajišťuje kontrolování a filtrování příchozích HTTP požadavků v Laravelu. Vytvořené middlewary se ukládají do adresáře `app/Http/Middleware` a na jednotlivé koncové body se dají aplikovat vícero způsoby. [81] Já jsem využil možnost seskupení koncových bodů a vzniklé skupině jsem přiřadil příslušný middleware. Tento způsob je zobrazen ve výpisu 3.8.

Výpis kódu 3.8 Aplikace middlewaru na skupinu koncových bodů

```
Route::middleware(['LMBearerTokenAuth'])->group(function () {
    Route::post('/order/build', [OrderApiController::class, 'processOrder']);
    Route::post('/products', [ProductApiController::class, 'addProduct']);
});
```

V následujících podkapitolách vysvětlím, k čemu a jak jsem middleware využil.

3.5.1 Zabezpečení API

Aby s Deployment Managerem nekomunikoval falešný License Manager či Builder, bylo potřeba API zabezpečit. Rozhodl jsem se pro zabezpečení pomocí tzv. Bearer tokenu. To je zašifrovaný řetězec zpravidla vygenerovaný serverem pro daného uživatele. Jakmile uživatel přistupuje k neveřejným informacím (v mém případě se jedná o veškerou komunikaci od License Manageru a Builderu), server vyžaduje v hlavičce požadavku `Authorization` Bearer token. Celý formát je `Authorization: Bearer <token>`. [82]

Pro jednoduchost jsem neimplementoval koncový bod, který by po zavolání vygeneroval Bearer token a odeslal ho odesílateli požadavku. Namísto toho lze Bearer token pro obě aplikace

Apps Manageru zvlášť nastavit v souboru `.env`. Pro zvýšení bezpečnosti se do tohoto souboru vkládá pouze hash tokenu. Ten lze vygenerovat pomocí metody Laravelu `Hash::make($token)` [83], kde `$token` je, pro zlepšení zabezpečení, náhodně vygenerovaný řetězec.

Middlewary `LMBearerTokenAuth` a `BuilderBearerTokenAuth` se starají o samotnou kontrolu přítomnosti a následně správnosti tokenu. V metodě `handle()`, která v middlewaru určuje, co má být na příchozích požadavcích zkontrolováno, se provádí kontrola shody hashe přijatého tokenu s hashem, jejž si uživatel definoval do souboru `.env`. Při neshodě či neposkytnutí Bearer tokenu je odesílateli požadavku vrácen kód 401.

3.5.2 Autentizace uživatele

Obsahem požadavku F4 je autentizace uživatele za pomocí přihlašovacího jména a hesla. Jelikož mi bylo sděleno, že Deployment Manager bude určený pro jednotky uživatelů v rámci společnosti Jagu, s. r. o., s vedoucím práce jsme se dohodli, že postačí „zadrátovaný login“. Jinými slovy, v aplikaci má být uloženo jedno přihlašovací jméno a jedno heslo, pomocí nichž se bude uživatel přihlašovat. Zadavatel nevyžaduje od aplikace rozdílné chování pro různé uživatele, tvorba uživatelských účtů tedy nebude přístupná.

3.5.2.1 Hotové řešení nabízené frameworkem

Framework nabízí kompletní řešení autentizace, které si vývojář může modifikovat. Lze si zvolit, jak bude Laravel ověřovat, zda je daný uživatel přihlášený (například pomocí *sessions*), jak bude ukládat informace o všech uživatelích (například do databáze) a další detaily. Dokonce je vývojáři vytvořena i obrazovka pro registraci a pro přihlášení. [84]

Jedná se však o velmi robustní a pro můj problém ne příliš vhodné řešení. Kdybych totiž toto řešení využil, musel bych kupříkladu zakázat přístup na stránku s registrací nového uživatele, pokud už byl nějaký uživatel zaregistrován. Dále by v tabulce všech registrovaných uživatelů byl vždy pouze nejvýše jeden uživatel. Proto jsem pro konstrukci této funkcionality tyto nástroje nepoužil.

3.5.2.2 Mé řešení

Rozhodl jsem se tedy pro implementaci autentizace uživatele pomocí middlewaru. Vytvořil jsem nejprve přihlašovací obrazovku a metodu, která při zadání správné kombinace uživatelského jména a hesla vloží do *session* informaci, že byl uživatel přihlášen. Hash správného uživatelského jména a hesla zadá uživatel při konfiguraci aplikace do souboru `.env`. Tento hash lze, stejně jako hash Bearer tokenů, vygenerovat Laravelem pomocí metody `Hash::make($string)`, kde `$string` je hashovaný řetězec (metodu je potřeba zavolat jednak pro uživatelské jméno, jednak pro heslo).

Pro pohybování se v aplikaci je každý uživatelův odeslaný požadavek kontrolován middlewarem `UsernameAndPasswordAuth`. Ten kontroluje, zda se v uživatelově *session* nachází položka `authenticated`. Pokud tato položka chybí, uživatel je přesměrován na obrazovku s přihlašením.

S vedoucím práce jsme se dohodli, že odhlášení uživatele z aplikace nebude vyžadováno, protože v aplikaci existuje pouze jeden uživatelský účet. Nemá tedy smysl se odhlašovat, neboť se nikdo jiný přihlásit nemůže. Odhlášení však systém provede automaticky po 120 minutách nečinnosti uživatele a jedná se o konfigurovatelný parametr. Časový údaj v minutách lze zadat do souboru `.env` k položce `SESSION_LIFETIME`.

3.6 Fronty

Pro zavolání některých metod služeb je zapotřebí nejprve obdržet požadavek od jiné komponenty Apps Manageru. Tato komponenta očekává brzkou odpověď a není možné nechat ji čekat, než

Deployment Manager provede například komunikaci se zbývající komponentou. Tento problém i spoustu dalších časově náročných požadavků řeší fronty.

Fronty umožňují řešit problémy na pozadí. Proces vykonávaný frontou se označuje slovem *job*. Takový *job* lze vytvořit v adresáři `app/Jobs` a pomocí metody `nazevJobu::dispatch()` se vkládá do fronty. [85] Ta může být zavolána například v controlleru po úspěšné validaci vstupu. Po vložení procesu do fronty může aplikace odpovědět na požadavek, aniž by čekala na dokončení zpracování tohoto procesu.

Pro správné fungování je však nutné nejprve spustit zpracování činností frontou. To lze provést pomocí příkazu `php artisan queue:work` v příkazové řádce běžícího serveru. Fronta tím zpracuje požadavky, které obdržela před tím, než byla zapnuta. Zpracovány budou i požadavky nově příchozí.

Dále bylo před prvním použitím této funkcionality nutné nastavit, jak si bude Laravel ukládat procesy do fronty. To framework nabízí přes technologie Redis či Amazon SQS. Mimo to podporuje i relační databázi. [85] Jelikož relační databázi má aplikace již nakonfigurovanou, zvolil jsem právě toto řešení. Stačilo jen vytvořit potřebné tabulky dle návodu v dokumentaci a fronta fungovala bezchybně.

Mnou vytvořené činnosti zpracovávané frontou volají vždy příslušnou metodu vhodné služby. `BuildProduct` je vložen do fronty, jakmile systém obdrží požadavek na sestavení produktu. `ProductBuilt` je *job* starající se o informování License Manageru, že Builder dokončil sestavení produktu. Poslední třídou je `VersionChangeFinished`. Proces, který je touto třídou reprezentován, obnáší komunikaci s Builderem a je vložen do fronty v momentě, kdy je od aplikace GitLab přijata informace, že byla dokončena změna verze.

Kapitola 4
Testování

Jedním z kroků, kterými má vzniklý prototyp dle softwarového procesu projít, je jeho otestování. [18] V této kapitole popíši, jaké typy testů existují a které z nich jsem pro testování Deployment Manageru zvolil. Dále zmíním, jak testování probíhalo a jaké přineslo výsledky.

4.1 Typy testů

Než představím, jak jsem výsledný prototyp testoval, je nutné zmínit typy testů, jež se běžně provádí. Těch existuje velké množství a lze je klasifikovat do různých kategorií, například na testy funkční a nefunkční. Funkčností se nemyslí fakt, zda test lze či nelze spustit, nýbrž to, zda testuje funkční nebo nefunkční požadavky softwaru. Dále se v tomto textu zaměřím právě na kategorii funkčních testů, neboť byly na Deployment Manager kladeny zejména funkční požadavky. Do této kategorie spadají mimo jiné testy

- jednotkové
- integrační
- systémové
- akceptační [86, 87, 88]

Jednotkové testy provádí vývojář a ověřuje se jimi správná funkčnost malých jednotek kódu jako jsou například funkce či metody. Tento typ testů lze automatizovat. [89, 90]

Aby se ověřilo správné fungování modulů, do nichž bývá software často rozdělen, používá se integrační testování. To ověří funkčnost softwaru jako celé skupiny modulů, čímž se zajistí, že různí programátoři vytvořili moduly tak, aby byla zajištěna kompatibilita. [91]

Systémové testování prověří, zda je software schopen komunikace s externími komponentami a fungování v systému, na němž bude používán v ostrém provozu. [92, 93]

Akceptační testování se oproti výše zmíněným provádí se zákazníkem. S ním se testuje, zda výsledný produkt splňuje zákazníkovy požadavky a zda je připraven na ostrý provoz. [94]

4.2 Testování prototypu aplikace

Před samotným testováním bylo potřeba vhodně zvolit testy. Kromě testů, k nimž vývojáře směruje Laravel a které popíši v kapitole 4.2.1, je vhodné otestovat i uživatelské rozhraní a jeho přívětivost. [95]

Vytvořený prototyp bude používán pouze administrátory, tedy lidmi, u nichž se předpokládá větší zkušenost v oboru a menší náročnost na grafické rozhraní. Jak jsem již psal v sekci 3.5.2, aplikace bude mít pouze jednotky uživatelů. I z toho důvodu jsme se společně se zadavatelem dohodli, že nebudu provádět detailní uživatelské testování s několika respondenty, ve kterém se prochází typické scénáře, ale že klientskou část prototypu otestujeme pouze v rámci akceptačního testování.

V této kapitole popíši, jak jsem samotné testování prováděl.

4.2.1 Testy prováděné na backendu

Jelikož byl k vývoji Deployment Manageru použit framework Laravel, zaměřil jsem se na testy, jež jsou doporučovány samotnými autory této technologie.

Aplikace již ve výchozím nastavení obsahuje adresář `tests`, v němž se nachází adresáře `Unit` a `Feature`. Framework totiž podporuje tyto dva typy testů. Jednotkové testy jsem popsal v kapitole 4.1. *Feature* testy testují větší části kódu, nebo dokonce i celý proces od odeslání požadavku na koncový bod přes jeho zpracování, interakci s databází a odeslání odpovědi na požadavek obsahující správný návratový kód a očekávané informace v těle požadavku. Dokumentace frameworku upozorňuje, že většina vývojáři psaných testů by měla spadat právě do kategorie *feature*, neboť tyto testy jsou schopny zachytit fungování systému jako celku. [96] Proto jsem všechny testy psal právě tímto způsobem.

4.2.1.1 Postup psaní testů

Při psaní testů je vhodné dodržovat strukturu adresáře s testy. Ta se tvoří zrcadlením adresáře s testovanými třídami. [97] Příklad jsem vložil do výpisu kódu 4.1, v němž si lze prohlédnout cestu k souboru `AppGroupControllerTest`, jenž obsahuje veškeré testy třídy `AppGroupController`. Zmíněná cesta koresponduje s cestou k testovanému souboru, kterou jsem zaznamenal do výpisu kódu 4.2.

Výpis kódu 4.1 Cesta k souboru s testy třídy AppGroupController

```
tests/Feature/Http/Controllers/User/AppGroupControllerTest.php
```

Výpis kódu 4.2 Cesta ke controlleru AppGroupController

```
app/Http/Controllers/User/AppGroupController.php
```

Pro každý controller jsem takovýmto způsobem vytvořil soubor, do nějž jsem napsal testy veškerých koncových bodů, které tento controller pokrývá.

Konkrétněji jsem postupoval tak, že jsem si otevřel soubory popisující dostupné koncové body mého prototypu a podle nich jsem nalezl metody, jež jsou při odeslání požadavku na tyto koncové body volány. U každé z nich jsem se zamyslel, jaké možné vstupy může požadavek obsahovat a při následné tvorbě testu jsem se snažil pokrýt co nejvíce případů. Jelikož veškeré funkcionality, které jsem do aplikace implementoval, jsou volány právě zasláním požadavku na korespondující koncový bod, otestoval jsem tímto způsobem téměř většinu funkcí prototypu.

4.2.1.2 Přístup k databázi a frontě během spouštění testů

Samotné spouštění testů provádí PHPUnit. Jedná se o testovací framework pro jazyk PHP. Chování tohoto frameworku lze nakonfigurovat v souboru `phpunit.xml`, kde lze mimo jiné nastavit i připojení k databázi, jež se má při testu používat.

Použití databáze tak, jak je zvykem v produkčním prostředí aplikace, vyžaduje vytvoření separátní databáze sloužící jen k testování, jinak by se mohlo stát, že kvůli testům ztratíme data uživatelů. Tvorba separátní databáze nepředstavuje žádný problém, avšak testování aplikace s takovýmto připojením k databázi je časově náročnější, neboť aplikace musí po každé úpravě dat přistupovat na disk a databázi uložit. Proto jsem použil tzv. *in-memory* databázi. Ta ukládá data jako standardní databáze s tím rozdílem, že nepřistupuje k disku zařízení, na němž běží, ale ukládá si je do operační paměti. Jakmile je připojení k databázi ukončeno, veškerá data se ztratí, což ale nevadí, neboť toto se děje automaticky po skončení testu, kdy už tato data nejsou potřeba. [96, 98, 99]

Během vývoje jsem používal databázový systém PostgreSQL, který však oficiálně *in-memory* podporu nenabízí. [100] Jak jsem již zmiňoval v kapitole 2.4, kód v Laravelu můžeme psát stejně pro jakýkoli z podporovaných typů databázových systémů, proto jsem zvolil systém SQLite, jenž touto možností disponuje.

Ve zmíněném souboru `phpunit.xml` jsem také zkontroloval, jak se PHPUnit během vykonávání testů připojuje k frontě. Standardní chování je nahrazení paralelního zpracování procesů zpracováním synchronním, tedy jakmile je vykonán jeden *job*, teprve poté se spustí vykonávání procesu druhého. To mi bohatě stačilo, neboť se testy pouští sériově a fronta tím pádem při jejich vykonávání zpracovává v každý moment nejvýše jeden test.

4.2.1.3 Komunikace přes API během spouštění testů

Jelikož prototyp aplikace poskytuje API, bylo nutné otestovat i reakce Deployment Manageru na příchozí požadavky od aplikací Builder, License Manager a GitLab. Protože není vhodné, aby se při každém spuštění testů (zpravidla se testy spouští při každé nově přidané funkcionalitě) zahajovala komunikace mezi mnou vytvořeným prototypem a aplikacemi, jež poskytované API konzumují, použil jsem metodu `fake`, kterou nabízí Guzzle HTTP Client. Ten je standardně integrován do každé aplikace vytvořené nad frameworkem Laravel. `Http::fake` se použije v testu před odesláním požadavku na koncový bod a specifikuje se v něm, na jaké URL adresy má reagovat a co má zasílat v odpovědi. Lze uvést jak obsah těla odpovědi na požadavek, tak i návratový kód či parametry v záhlaví. [101]

Jedno z mnoha využití při testování prototypu si lze prohlédnout ve výpisu kódu 4.3. Tam jsem metodu použil pro odchycení požadavků na jakékoli URL (symbol `*` označuje libovolný řetězec). Na tyto požadavky Laravel vždy odpoví stavem 201 a v těle odpovědi nalezneme jednoprvkové pole obsahující dvojici s klíčem `id` a hodnotou 50.

Výpis kódu 4.3 Využití metody Http::fake

```
Http::fake([
        '*' => Http::response(['id' => 50], 201)
    ]);
```

4.2.2 Akceptační testování

Akceptační testování, jehož smysl jsem popsal v kapitole 4.1, jsem realizoval s Ing. Oldřichem Malcem, vedoucím práce, a Ing. Jiřím Hunkou (oba zastupují zadavatele) dne 19. 4. 2023 formou osobního setkání.

Nejprve jsem připomenul fungování aplikace a zopakoval důležité pojmy jako *produkt* a *komponenta* a rozdíly mezi nimi. Také jsem zopakoval zadavatelovy připomínky z poslední schůzky, neboť součástí tohoto setkání bylo také představení finální podoby aplikace.

Poté jsem společně se zadavatelem, jemuž jsem sdílel obrazovku pomocí aplikace Google Meet, procházel různé scénáře, které uživatelé Deployment Manageru budou typicky provádět.

Již během průchodu měli oba přítomní dotazy, pomocí kterých jsme si vyjasnili detailní fungování a různé krajní případy.

Poté jsem začal promítat seznam funkčních a nefunkčních požadavků v té podobě, v jaké je uvedený v kapitole 1.6. Se zadavatelem jsme se dohodli, že mimo tvrzení, zda daný požadavek byl či nebyl naplněn, uvede i procentuální hodnocení. To reflektuje celkový dojem z hodnoceného požadavku, v němž je zahrnut například i návrh uživatelského rozhraní a uživatelský prožitek. Kompletní hodnocení požadavků od obou hodnotitelů lze nalézt v tabulce 4.1.

Tabulka 4.1 Hodnocení požadavků Ing. Oldřichem Malcem a Ing. Jiřím Hunkou

ID požadavku	Zjednodušený popis požadavku	Hodnocení OM	Hodnocení JH
F1	zobrazení a filtr aplikací	70 %	60 %
F2	změna verze aplikace	90 %	80 %
F3	automatická volba verze	90 %	85 %
F4	autentizace uživatele	100 %	50 %
F5	správa skupin aplikací	70 %	90 %
F6	navázání komunikace s Builderem	50 %	100 %
F7	správa produktů a komponent	80 %	75 %
N1	grafické uživatelské rozhraní	70 %	70 %
N2	poskytnutí API Builderu a LM	100 %	100 %

Právě uživatelský prožitek byl hlavním problémem, který ubral procenta na celkovém hodnocení. Anglickým termínem je *User Experience* a tento pojem popisuje celkový dojem uživatele z používání aplikace. Nehodnotí se tedy pouze tvar jednotlivých tlačítek či velikost ikon (jako tomu je u hodnocení uživatelského rozhraní, angl. *User Interface*), nýbrž spíše to, kolik kroků musí uživatel vykonat, aby se mu zobrazila jím cílená data. [102] Například při hodnocení požadavku *F7: Systém umožní uživateli spravovat produkty a jejich komponenty* chyběla hodnotitelům možnost zobrazit si správu komponent z pohledu komponenty. Aplikace sice uživateli umožní zobrazit informace o komponentách a zjistit, k jakým produktům jsou přiřazeny, ale je pro to nutné si nejprve ve správě produktů rozkliknout příslušný produkt. Uživateli se poté zobrazí veškeré evidované komponenty toho produktu, nikoli produkty patřící k jím zvolené komponentě.

Dalším důvodem sníženého hodnocení byla rozdílná představa o výsledné aplikaci obou hodnotitelů. Ku příkladu v požadavku *F4: Systém umožní autentizovat uživatele* Ing. Jiřímu Hunkovi chyběla možnost se odhlásit, ačkoli jsme se s vedoucím práce shodli na tom, že tuto funkcionalitu Deployment Manager podporovat nebude (viz sekci 3.5.2.2). Další důvody, kvůli nimž bylo hodnocení sníženo, popíši v kapitole 5.2.2.1.

Mimo procentuální hodnocení zadavatel uvedl, že veškeré požadavky má aplikace splňuje a je tím považována za akceptovanou.

Kapitola 5
Možná vylepšení

Vytyčené cíle byly splněny. Podařilo se mi analyzovat, navrhnout, zkonstruovat a otestovat prototyp webové aplikace zajišťující přehled o nasazených aplikacích a jejich verzích. Změna verze aplikací je také umožněna. Přesto jsem během vývoje narazil na spoustu funkcionalit, které by bylo užitečné v Deployment Manageru mít.

Než popíši tato možná vylepšení, budu část textu věnovat zhodnocení systému Apps Manager a zmíním důležité informace související s budoucím vývojem vytvořeného softwaru.

5.1 Systém Apps Manager

Na Apps Manageru jsme pracovali společně. Protože spolu naše aplikace komunikují, bylo potřeba včas vydefinovat komunikační rozhraní, podle nějž bychom mohli náš software implementovat. To se však nepovedlo tak, jak bylo při definování zadání našich bakalářských prací zamýšleno.

Po dokončení implementace a testování Deployment Manageru byly prováděny změny v API, které mnou zkonstruovaný systém nepodporuje. To značně ovlivňuje výsledné fungování Apps Manageru.

Jednou z výše zmíněných změn je zabezpečení API aplikace Builder. Deployment Manager ve svém aktuálním stavu není schopen požádat Builder o Bearer token, který bude následně odesílat v požadavcích na něj. Pro používání Apps Manageru je možné toto zabezpečení v Builderu deaktivovat. Nejedná se však o rozumné řešení a je namístě do Deployment Manageru chybějící funkcionalitu implementovat. Podobný problém se vyskytuje i v komunikaci mezi Deployment Managerem a License Managerem.

Další takovou změnou jsou nové koncové body v API aplikace Builder. Ty slouží k informování Builderu o nově přidaném produktu a komponentě v aplikaci Deployment Manager. To sice nezpůsobuje nefunkčnost našeho celku, představuje to však jistý diskomfort. Uživatel Apps Manageru je tímto nucen přidávat produkty a jejich komponenty do databáze aplikace Builder ručně.

Apps Manager bude jako celek použitelný, jakmile budou výše zmíněné problémy vyřešeny.

Zmíněné nesoulady byly způsobeny naší špatnou zkušeností s vývojem softwaru a místy nedostatečnou komunikací.

5.2 Budoucí vývoj

Pro budoucí vývoj je vhodné vytvořit příručku pro programátora, aby mu bylo jasné, jak aplikaci zprovoznit pro implementaci nových či opravu starších funkcionalit. Tu jsem vytvořil a vložil jsem ji do repozitáře aplikace GitLab, v němž byl verzován kód aplikace. Lze ji nalézt v souboru

`README.md`. Tento soubor se nachází také v přiloženém archivu ve složce se zdrojovými kódy aplikace.

Dále byla vytvořena dokumentace API Apps Manageru, která zobrazuje koncové body v tom stavu, v jakém byly vytvořeny v době implementace Deployment Manageru. Její tvorbu jsem popsal v kapitole 2.7 a dokumentace se nachází v přiloženém archivu.

5.2.1 Dokumentace kódu

Mimo dokumentaci API jsem vytvořil také dokumentaci kódu celé aplikace, která slouží k tomu, aby vývojář, jenž bude na aplikaci dále pracovat, věděl, k čemu jaká třída či její metoda slouží.

Protože sepisovat takovou dokumentaci ručně je velmi náročná práce, nechal jsem si dokumentaci vygenerovat. K tomu jsem použil nástroj phpDocumentor. Ten umí pracovat s kódem v jazyce PHP a ke tvorbě samotného dokumentu používá specificky formátované komentáře – tzv. DocBlocks.

Ty se píší nejen ke třídám a metodám a shrnují, co daná třída, resp. metoda, dělá i jaký význam mají její parametry. U metod upozorňuje, jaké výjimky mohou být vyhozeny a popisuje význam objektu, jejž metoda vrací. [103] PhpStorm, mnou používáné IDE, umí vygenerovat jejich strukturu, což přináší další zjednodušení práce.

Další výhodou použitého dokumentačního nástroje je přítomnost plnotextového (anglicky *full-text*) vyhledávání ve výsledném textu. Dokumentace je, stejně jako dokumentace API, ve formátu webové stránky a je přiložena v archivu v příslušné složce.

5.2.2 Možná vylepšení

Než uvedu několik návrhů vylepšení aplikace, zmíním, že nejdůležitější budoucí úpravou kódu je odstranění nedostatků popsaných výše v sekci 5.1.

5.2.2.1 Frontend aplikace

V kapitole 4.2.2 jsem mimo jiné zmínil, že aplikace má vady na své klientské části. Jelikož jsem je z důvodu nedostatku času do výsledného prototypu nezapracoval, popíši je nyní jako možnost pro vylepšení aplikace.

Mimo již zmíněný chybějící pohled na správu komponent, který vypíše, k jakým produktům je daná komponenta přiřazena, se nabízí zlepšení obrazovky s přehledem nasazených aplikací, již si lze prohlédnout v příloze C. Nabízenou úpravou je změna polohy vyhledávacích textových polí pod jednotlivými sloupci. Bylo by pohodlnější mít tato pole k dispozici nad jednotlivými sloupci, neboť v tomto případě uživatel nemusí sjíždět na spodek obrazovky.

Momentálně je výchozí řazení tabulky nasazených aplikací dle jejich ID. Možným vylepšením je řadit tabulku dle poslední provedené změny jednotlivých záznamů. Člověk totiž často hledá aplikaci, která byla právě nasazena či u které byla právě provedena změna verze.

Na obrazovce zobrazující komponenty zvoleného produktu je možnost přidání nové komponenty. Jednou z položek vyžadovaných pro tvorbu komponenty je *Přístupový token na GitLab API*. Zde nemusí být uživateli patrné, o jaký token se jedná. Bylo by proto namístě například přidat odkaz na tvorbu tohoto tokenu pro daný projekt.

Na stejné obrazovce má uživatel dále možnost přiřadit k produktu již existující komponentu. Jelikož se spousta aplikací vyvíjených zadavatelem skládá z více než jedné komponenty a tyto komponenty mohou být sdílené mezi produkty, připadá v úvahu další vylepšení. Konkrétně upravit přidávání existující komponenty tak, aby bylo možné zvolit několik komponent naráz.

5.2.2.2 Evidence vývojových větví komponent

Momentálně aplikace podporuje evidenci *tagů* příslušící dané komponentě v jejím repozitáři v systému GitLab. Možným vylepšením je evidování jednotlivých vývojových větví, které zadavatel využívá pro testovací účely či jako ukázku potenciálnímu zákazníkovi, jak daný produkt vypadá a jak funguje.

Jelikož vývojové větve zpravidla nejsou pojmenovávány dle verze ve formátu sémantického verzování, systém by pro tuto funkcionalitu nenabízel automatické doporučování verzí. Pro toto vylepšení by bylo pravděpodobně žádoucí umožnit nasazení aplikací bez nutnosti zakoupení produktu v aplikaci License Manager, neboť by zakoupení vlastního produktu zadavatelem bylo nelogické.

5.2.2.3 Zapínání a vypínání jednotlivých funkcionalit

Produkty zadavatele obsahují různé funkcionality, o které nemá zájem každý zákazník. Tyto funkcionality jsou obsaženy v konfiguracích produktu, které eviduje systém License Manager. Funkcionality, jež si zákazník při objednávce zvolí, ovlivní cenu výsledného produktu.

Možným vylepšením je přidání možnosti do Deployment Manageru, která by zajistila čtení dat z konfigurace zakoupeného produktu a podle nich by aplikaci Builder sdělila, jaké funkce má v sestavované aplikace zapnout a které má naopak ponechat vypnuté. Pro toto vylepšení by bylo nutné upravit i fungování aplikace Builder a provést patřičné úpravy v souboru `.gitlab-ci.yml`, jenž se nachází v repozitářích komponent jednotlivých produktů v systému GitLab.

5.2.2.4 Automatické definování kompatibilit verzí

Nové verze opravující nalezené chyby mohou vznikat často. Ve formátu sémantického verzování, jejž zadavatel používá, se u takových verzí používá označení předešlé verze s inkrementovanou `PATCH` hodnotou. Například opravením drobné chyby komponenty ve verzi `1.5.2` vznikne verze, která se označí jako `1.5.3`.

Deployment Manager ve svém aktuálním stavu podporuje přidávání kompatibilit verzí. Vstupem pro definici nové kompatibility je mimo jiné název komponenty a sémantické označení jedné z jejich verzí. Jako zlepšení této funkce se nabízí přidání podpory alespoň některých symbolů sémantického verzování, díky nimž by uživatel nemusel po každé nově vydané verzi znovu definovat veškeré kompatibility pro tuto verzi.

Pro lepší pochopení uvedu příklad. V systému je evidována verze `1.5.0` komponenty A. Uživatel do aplikace přidá kompatibilitu komponenty A ve verzi `1.5.x` (symbol `x` by byl obsahem vylepšení) s komponentou B, s níž je komponenta A kompatibilní v libovolné verzi či v jejím omezení (omezení jsem popsal v kapitole 1.5.5). Jakmile je vydána nová verze `1.5.1` pro komponentu A, uživatel nemusí znovu definovat její kompatibilitu s komponentou B, neboť výraz `1.5.x` tuto verzi již obsahuje.

Závěr

Cílem práce bylo vytvoření prototypu webové aplikace, jež uživatelům umožní získat přehled o nasazených aplikacích u zákazníků, jejich verzích a dále zjednoduší změnu verze těchto aplikací. Tento cíl byl naplněn.

Nejprve jsem se věnoval analýze a popsal jsem obecné potřeby softwarových vývojářů týkající se distribuce odlišných verzí jejich softwaru různým klientům. Dále jsem formou osobních setkání zanalyzoval potřeby zadavatele, z nichž vyplynul mimo jiné seznam požadavků a případů užití pro následně navrhovanou a implementovanou aplikaci.

Poté jsem zhodnotil, jaká technologie a architektura bude pro implementaci prototypu vhodná. Mimo to jsem navrhl způsob ukládání dat, a jelikož jedním z požadavků bylo provedení změny verze běžící aplikace, rozebral jsem způsob, kterým tento proces výsledná aplikace provádí. Výsledný prototyp je součástí trojice Apps Manager, pro kterou platí, že spolu dílčí aplikace komunikují. Proto jsem provedl navrhnutí API pro tuto komunikaci a výslednou podobu rozhraní jsem zdokumentoval. Poskytl jsem i diagram tříd popisující výslednou podobu databáze aplikace.

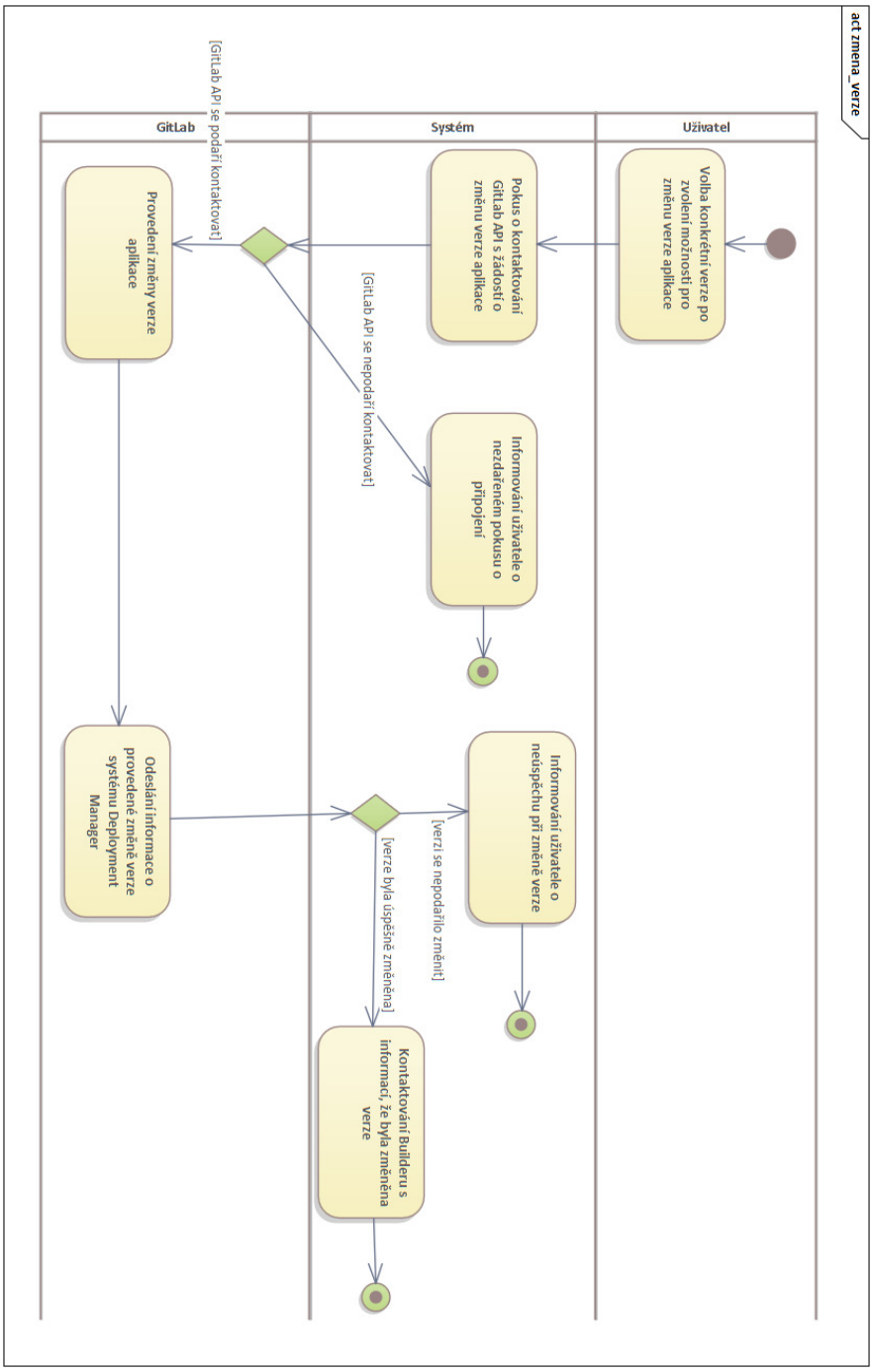
Nabyté informace jsem zužitkoval při implementaci. Prototyp jsem zkonstruoval ve skriptovacím jazyku PHP ve frameworku Laravel, který využívá architekturu MVC. Použitá databázová technologie byla PostgreSQL. Systém je připravený na konzumaci API aplikací Builder a License Manager a je schopen s nimi komunikace zahájit. Z důvodu pozdějšího definování API těchto aplikací však neimplementuje všechny vydefinované koncové body.

Výslednou aplikaci jsem otestoval. Byly provedeny jak testy serverové části, tak testy uživatelského rozhraní. Na závěr jsem se zadavatelem realizoval akceptační testování.

Ač byl cíl práce splněn, během tvorby aplikace jsem si vytyčil několik funkcionalit, díky kterým by výsledné dílo mohlo získat přidanou hodnotu. Mezi ně patří například evidence vývojových větví u ukládaných komponent či možnost automatického definování kompatibilit verzí.

Příloha A

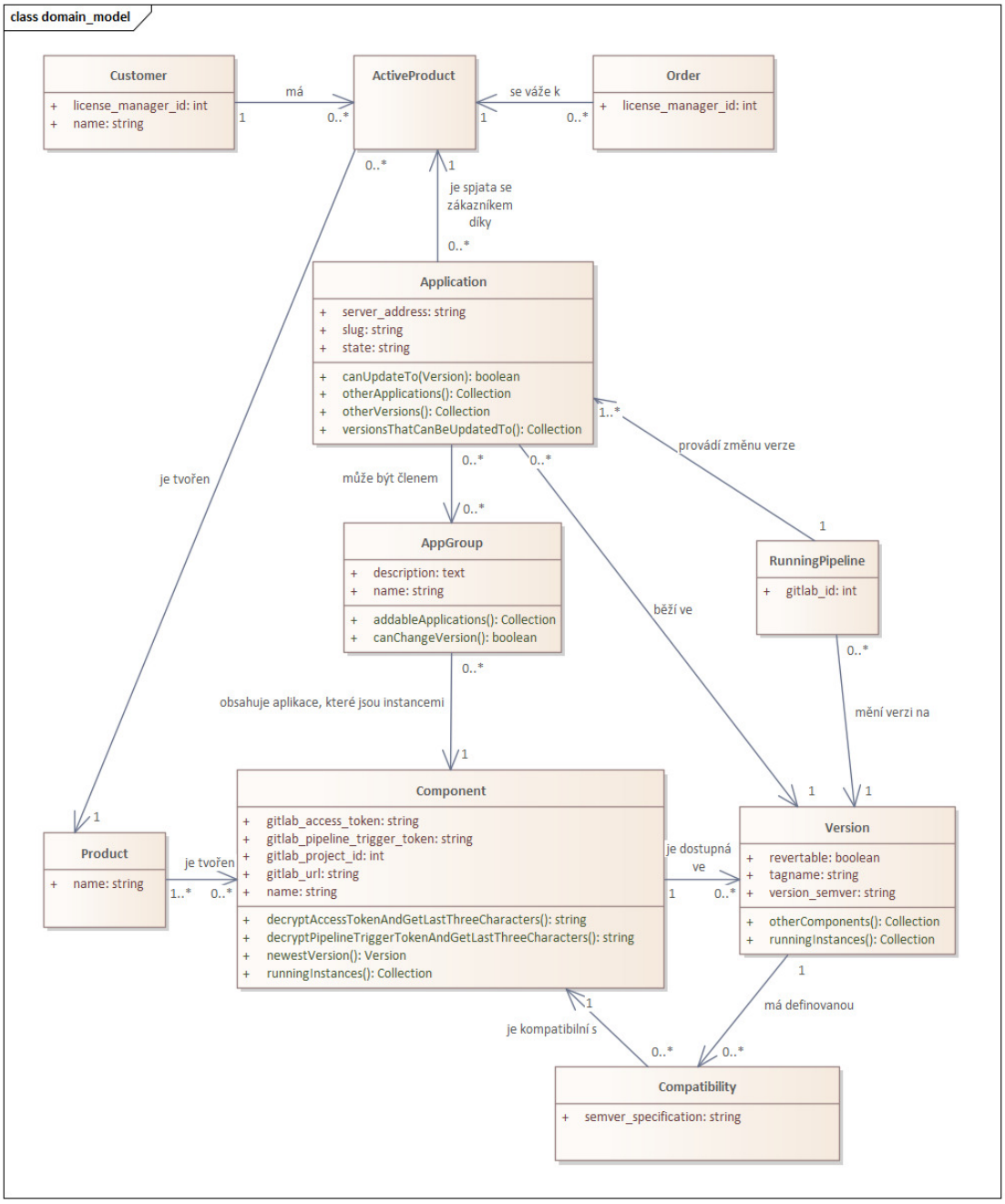
Diagram aktivit



Obrázek A.1 Diagram aktivit změny verze aplikace

Příloha B

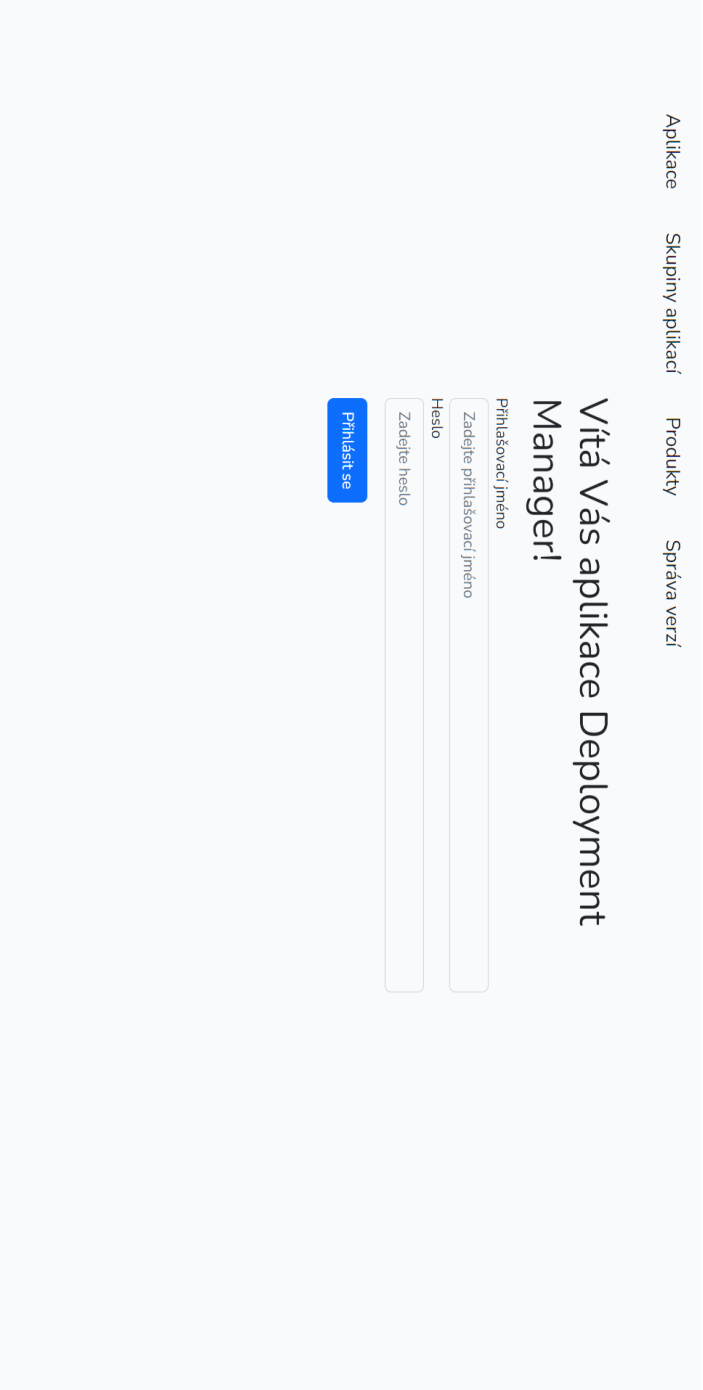
Diagram tříd



Obrázek B.1 Diagram tříd aplikace Deployment Manager

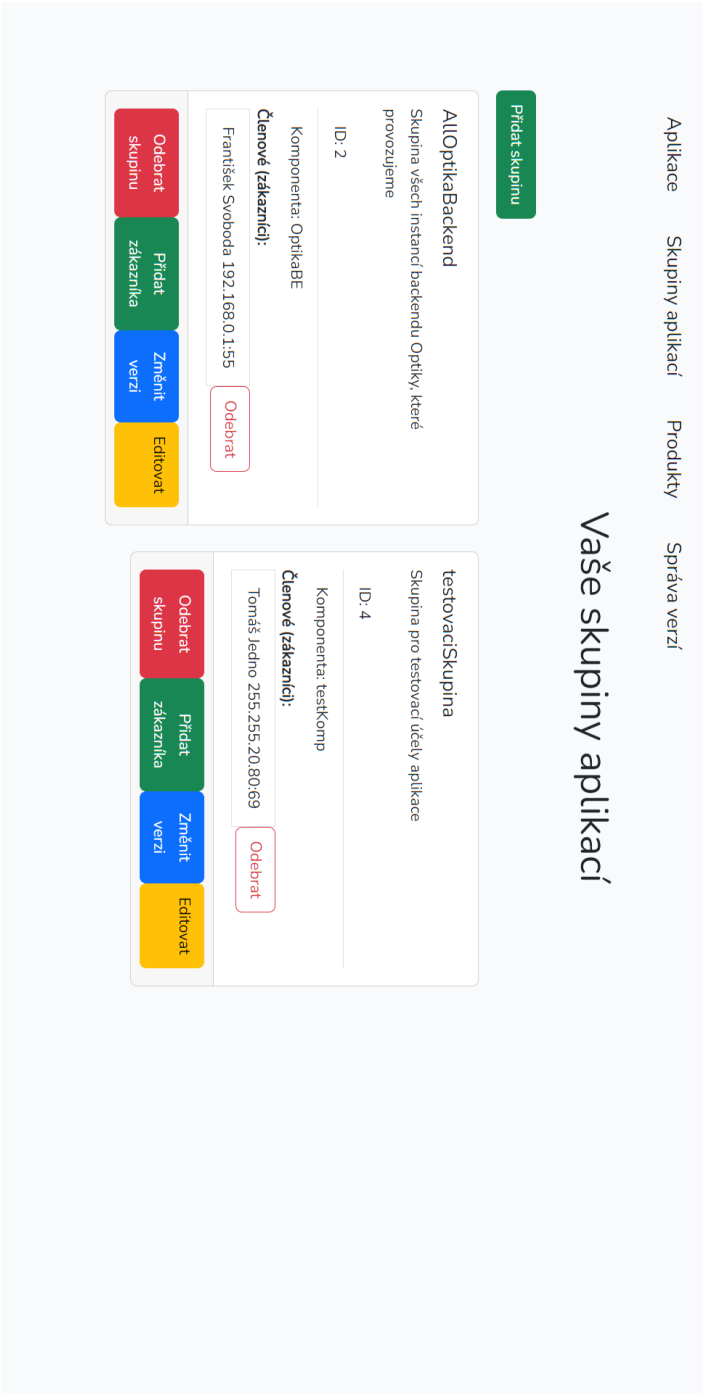
Příloha C

Snímky obrazovek aplikace



Obrázek C.1 Přihlašovací obrazovka

Obrázek C.2 Přehled nasazených aplikací



Obrázek C.3 Přehled skupin aplikací

Obrázek C.4 Komponenty zvoleného produktu

Obrázek C.5 Přehled vztahů mezi verzemi

Bibliografie

1. MLEJNEK, Jiří. *Analýza a sběr požadavků* [online]. 2022. [cit. 2023-04-19]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/506241/mod_resource/content/7/03.prednaska.pdf.
2. HENDL, Jan. *Kvalitativní výzkum: základní metody a aplikace*. Praha: Portál, 2005. ISBN 80-7367-040-2.
3. SURVIO. *Kvantitativní výzkum vs. kvalitativní výzkum - Survio* [online]. 2020-10. [cit. 2023-04-16]. Dostupné z: https://www.survio.com/cs/blog/jak-vytvorit-dotaznik/kvantitativni-vyzkum-kvalitativni-vyzkum.
4. STACK EXCHANGE, Inc. *Stack Overflow* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://stackoverflow.co.
5. STACK EXCHANGE, Inc. *The world's largest programming community is growing* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://stackexchange.com/about.
6. STACK EXCHANGE, Inc. *Ask a public question - Stack Overflow* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://stackoverflow.com/questions/ask.
7. STACK EXCHANGE, Inc. *What does it mean if a question is "closed"? - Help Center - Stack Overflow* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://stackoverflow.com/help/closed-questions.
8. TEAM, Indeed Editorial. *What is a core product? (definition, uses and examples) | indeed.com UK* [online]. 2022-07. [cit. 2023-04-19]. Dostupné z: https://uk.indeed.com/career-advice/career-development/core-product.
9. PSHVED. *versioning - How to maintain different, customized versions of the same software for multiple clients - Software Engineering Stack Exchange* [online]. 2011-03. [cit. 2023-04-19]. Dostupné z: https://softwareengineering.stackexchange.com/questions/60393/how-to-maintain-different-customized-versions-of-the-same-software-for-multiple.
10. DEE, Robbie. *Maintaining different versions of SW to different clients* [online]. 2022-10. [cit. 2022-10-24]. Dostupné z: https://softwareengineering.stackexchange.com/questions/441831/maintaining-different-versions-of-sw-to-different-clients.
11. ÔREL. *Maintaining different versions of SW to different clients* [online]. 2022-10. [cit. 2022-10-24]. Dostupné z: https://stackoverflow.com/questions/74180547/maintaining-different-versions-of-sw-to-different-clients.

12. MULDROW, Lyn. *What is DRY development? | DigitalOcean* [online]. 2020-12. [cit. 2023-04-19]. Dostupné z: https://www.digitalocean.com/community/tutorials/what-is-dry-development.
13. CHROMATIC. *Shooting yourself in the foot with customer branches* [online]. 2009-03. [cit. 2023-04-19]. Dostupné z: http://www.modernperlbooks.com/mt/2009/03/shooting-yourself-in-the-foot-with-customer-branches.html.
14. CARTER, Thomas. *How to Buy Source Code - Lower App Development Costs - App Empire* [online]. 2018. [cit. 2023-04-19]. Dostupné z: https://appempire.com/wholesale-source-code/.
15. POSADA, Barbara. *7 tips for handling crazy requirements proposed by clients* [online]. 2016-03. [cit. 2023-04-19]. Dostupné z: https://www.linkedin.com/pulse/7-tips-handling-crazy-requirements-proposed-clients-posada-men%C3%A9ndez.
16. QUICKLY_NOW. *versioning - How to maintain different, customized versions of the same software for multiple clients - Software Engineering Stack Exchange* [online]. 2011-03. [cit. 2023-04-19]. Dostupné z: https://softwareengineering.stackexchange.com/questions/60393/how-to-maintain-different-customized-versions-of-the-same-software-for-multiple.
17. SATYABRATA, Jena. *Customer satisfaction in Software development - GeeksforGeeks* [online]. 2022-09. [cit. 2023-04-16]. Dostupné z: https://www.geeksforgeeks.org/customer-satisfaction-in-software-development/.
18. HLAVATÝ, Martin. *Softwarový proces* [online]. 2018. [cit. 2023-04-19]. Dostupné z: https://profinit.eu/wp-content/uploads/2018/10/BISI2_1_SoftwareProcess.pdf.
19. TOWNER, Joe. *What is the Difference Between Agile and Waterfall?* [online]. 2022-06. [cit. 2023-04-19]. Dostupné z: https://www.forecast.app/blog/difference-between-agile-waterfall.
20. PAULA. *Iterative Development vs Agile Development* [online]. 2022. [cit. 2023-04-19]. Dostupné z: https://premieragile.com/agile-vs-iterative-model/.
21. WRIKE. *What Is Agile Methodology in Project Management?* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.wrike.com/project-management-guide/faq/what-is-agile-methodology-in-project-management/.
22. HOLÝ, Viktor. *License manager – webová aplikace*. 2021. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií.
23. LOTTE. *What Are Frontend and Backend in App Development?* [online]. 2021-09. [cit. 2023-04-19]. Dostupné z: https://lizard.global/blog/what-are-frontend-and-backend-in-app-development.
24. GITLAB. *GitLab CI/CD | GitLab* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://docs.gitlab.com/ee/ci/.
25. GITLAB. *'.gitlab-ci.yml' keyword reference | GitLab* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://docs.gitlab.com/ee/ci/yaml/index.html.
26. GITLAB. *Tags | GitLab* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://docs.gitlab.com/ee/user/project/repository/tags/.
27. PRESTON-WERNER, Tom. *Semantic Versioning 2.0.0 | Semantic Versioning* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://semver.org/.
28. NPM. *semver - npm* [online]. 2023. [cit. 2023-04-22]. Dostupné z: https://www.npmjs.com/package/semver.
29. GETCOMPOSER. *Versions and constraints - Composer* [online]. 2023. [cit. 2023-04-22]. Dostupné z: https://getcomposer.org/doc/articles/versions.md.

30. EELES, Peter. Non-functional requirements. *IBM Software Group*. 2005.

31. F5 NGINX. *What Is Load Balancing? How Load Balancers Work* [online]. 2023. [cit. 2023-04-22]. Dostupné z: https://www.nginx.com/resources/glossary/load-balancing/.

32. HAYES, Adam. *What Is Cross-Selling?* [online]. 2022. [cit. 2023-04-22]. Dostupné z: https://www.investopedia.com/terms/c/cross-sell.asp.

33. KUHN, Janet. Decrypting the MoSCoW analysis. *The workable, practical guide to Do IT Yourself*. 2009, roč. 5.

34. AIRFOCUS. *What Is Moscow Prioritization? Definition, How-to & FAQ* [online]. 2023. [cit. 2023-04-22]. Dostupné z: https://airfocus.com/glossary/what-is-moscow-prioritization/.

35. GUTHRIE, Georgina. *Create a use case scenario: how to think like users to improve products | Nulab* [online]. 2022. [cit. 2023-04-22]. Dostupné z: https://nulab.com/learn/design-and-ux/how-to-create-a-use-case-scenario-to-improve-products/.

36. MLEJNEK, Jiří. *Návrh softwarových systémů* [online]. 2022. [cit. 2023-04-23]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/532107/mod_resource/content/6/05.prednaska.pdf.

37. MINDBOWSER. *Why Software Design Is Important? l MindBowser* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.mindbowser.com/why-software-design-is-important/.

38. CODEACADEMY TEAM. *What Is a Framework?* [online]. 2021. [cit. 2023-04-23]. Dostupné z: https://www.codecademy.com/resources/blog/what-is-a-framework/.

39. MONGODB, INC. *What Is Full Stack Development? | A Complete Guide | MongoDB | MongoDB* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.mongodb.com/languages/full-stack-development.

40. DJANGO SOFTWARE FOUNDATION. *The web framework for perfectionists with deadlines | Django* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.djangoproject.com/.

41. DAFTARI, Shiv. *Top 10 Pros of using Django framework for back-end web development* [online]. 2020. [cit. 2023-04-23]. Dostupné z: https://www.kellton.com/kellton-tech-blog/why-django-web-development-with-python-for-backend-web-development.

42. DJANGO SOFTWARE FOUNDATION. *How to install Django | Django documentation | Django* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://docs.djangoproject.com/en/4.2/topics/install/#installing-official-release.

43. SCALER. *Understanding Django Architecture | MVT Architecture - Scaler Topics* [online]. 2022. [cit. 2023-04-23]. Dostupné z: https://www.scaler.com/topics/django/django-architecture/.

44. STŘECHA, Tomáš. *Lekce 1 - Úvod do Spring Boot frameworku v Javě* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.itnetwork.cz/java/spring-boot/zaklady/uvod-do-spring-boot-frameworku-pro-javu/.

45. SPRING. *Spring | Web Applications* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://spring.io/web-applications.

46. NASTASE, Alexandru. *Spring Boot: The GOOD and the BAD | by Alexandru Nastase | Medium* [online]. 2022. [cit. 2023-04-23]. Dostupné z: https://medium.com/@alexthedev/spring-boot-the-good-and-the-bad-20be1b409f2.

47. W3SCHOOLS. *Introduction to Java* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.w3schools.com/java/java_intro.asp.

48. LARAVEL LLC. *Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://laravel.com/.
49. W3TECHS. *Usage Statistics and Market Share of PHP for Websites, April 2023* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://w3techs.com/technologies/details/pl-php.
50. ZEND. *Developing Web Applications in PHP | Zend* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.zend.com/resources/developing-web-applications-php.
51. LARAVEL. *Installation - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://laravel.com/docs/10.x#laravel-and-docker.
52. STATISTICS AND DATA. *Most Popular Backend Frameworks – 2012/2022* - [online]. 2022. [cit. 2023-04-23]. Dostupné z: https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/.
53. PUSHER LTD. *How Laravel implements MVC and how to use it effectively | Pusher blog* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://pusher.com/blog/laravel-mvc-use/#what-is-mvc.
54. MORBY, Graham. *MVC and creating it in Laravel 8 - DEV Community* [online]. 2021. [cit. 2023-04-24]. Dostupné z: https://dev.to/grahammorby/mvc-and-creating-it-in-laravel-8-2a6b.
55. RYZYCKI, Marcin. *GitLab: understanding pipelines, stages, jobs and organising them efficiently for speed and feedback loop | by Marcin Ryzycki | Medium* [online]. 2018. [cit. 2023-04-24]. Dostupné z: https://medium.com/@ryzmen/gitlab-fast-pipelines-stages-jobs-c51c829b9aa1.
56. LARAVEL. *Eloquent: Getting Started - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://laravel.com/docs/10.x/eloquent.
57. LARAVEL. *Database: Getting Started - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://laravel.com/docs/10.x/database.
58. GITLAB. *REST API | GitLab* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.gitlab.com/ee/api/rest/index.html.
59. GITLAB. *Tags API | GitLab* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.gitlab.com/ee/api/tags.html.
60. GITLAB. *Webhooks | GitLab* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.gitlab.com/ee/user/project/integrations/webhooks.html.
61. GITLAB. *Projects API | GitLab* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.gitlab.com/ee/api/projects.html#add-project-hook.
62. GITLAB. *Pipeline trigger tokens API | GitLab* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.gitlab.com/ee/api/pipeline_triggers.html#create-a-trigger-token.
63. GITLAB. *Webhook events | GitLab* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.gitlab.com/ee/user/project/integrations/webhook_events.html.
64. SMARTBEAR SOFTWARE. *SwaggerHub | API Design and Documentation with OpenAPI* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://swagger.io/tools/swaggerhub/.
65. MLEJNEK, Jiří. *Implementace* [online]. 2022. [cit. 2023-04-26]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/532120/mod_resource/content/4/08.prednaska.pdf.

66. GRYGAŘÍKOVÁ, Michaela. *Docker, Kubernetes a kontejnery. Jak fungují a proč je chtít* [online]. 2019. [cit. 2023-04-27]. Dostupné z: https://www.master.cz/blog/docker-kubernetes-kontejnery-jak-funguji-proc-je-chtit/.

67. LARAVEL. *Database: Query Builder - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://laravel.com/docs/10.x/queries.

68. LARAVEL. *Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-28]. Dostupné z: https://laravel.com/docs/10.x/eloquent-relationships.

69. LARAVEL. *Views - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-28]. Dostupné z: https://laravel.com/docs/10.x/views.

70. LARAVEL. *Blade Templates - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-28]. Dostupné z: https://laravel.com/docs/10.x/blade.

71. BOOTSTRAP. *Bootstrap · The most popular HTML, CSS, and JS library in the world.* [online]. 2023. [cit. 2023-04-28]. Dostupné z: https://getbootstrap.com/.

72. W3SCHOOLS. *AJAX Introduction* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://www.w3schools.com/js/js_ajax_intro.asp.

73. DATATABLES. *DataTables | Table plug-in for jQuery* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://datatables.net/.

74. DATATABLES. *Data* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://datatables.net/manual/data/.

75. LARAVEL. *Routing - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://laravel.com/docs/10.x/routing.

76. LARAVEL. *Validation - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://laravel.com/docs/10.x/validation.

77. KARUNARATNE, Ayesh. *Enums - PHP 8.1 • PHP.Watch* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://php.watch/versions/8.1/enums.

78. SAMPSON, Ben. *GitHub - BenSampo/laravel-enum: Simple, extensible and powerful enumeration implementation for Laravel.* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://github.com/BenSampo/laravel-enum.

79. CSAJTAI, Peter. *GitHub - z4kn4fein/php-semver: Semantic Versioning library for PHP.* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://github.com/z4kn4fein/php-semver.

80. GITLAB. *'.gitlab-ci.yml' keyword reference | GitLab* [online]. 2023. [cit. 2023-04-30]. Dostupné z: https://docs.gitlab.com/ee/ci/yaml/index.html#rules.

81. LARAVEL. *Middleware - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://laravel.com/docs/10.x/middleware.

82. SMARTBEAR SOFTWARE. *Bearer Authentication* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://swagger.io/docs/specification/authentication/bearer-authentication/.

83. LARAVEL. *Hashing - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://laravel.com/docs/10.x/hashing.

84. LARAVEL. *Authentication - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://laravel.com/docs/master/authentication.

85. LARAVEL. *Queues - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-05-01]. Dostupné z: https://laravel.com/docs/10.x/queues.

86. HAMILTON, Thomas. *What is Functional Testing? Types & Examples* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://www.guru99.com/functional-testing.html.
87. HAMILTON, Thomas. *What is Software Testing? Definition* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://www.guru99.com/software-testing-introduction-importance.html.
88. JAVATPOINT. *Functional Testing - javatpoint* [online]. 2021. [cit. 2023-04-19]. Dostupné z: https://www.javatpoint.com/functional-testing.
89. JAVATPOINT. *Unit Testing - javatpoint* [online]. 2021. [cit. 2023-04-19]. Dostupné z: https://www.javatpoint.com/unit-testing.
90. HAMILTON, Thomas. *Unit Testing Tutorial – What is, Types & Test Example* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.guru99.com/unit-testing-guide.html.
91. HAMILTON, Thomas. *Integration Testing: What is, Types with Example* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.guru99.com/integration-testing.html.
92. PP_PANKAJ. *System Testing - GeeksforGeeks* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.geeksforgeeks.org/system-testing/.
93. HAMILTON, Thomas. *What is System Testing? Types with Example* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.guru99.com/system-testing.html.
94. SOFTWARE TESTING HELP. *What is Acceptance Testing (A Complete Guide)* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.softwaretestinghelp.com/what-is-acceptance-testing/.
95. BOSE, Shreya. *Front End Testing: A Beginner's Guide | BrowserStack* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://www.browserstack.com/guide/front-end-testing.
96. LARAVEL. *Testing: Getting Started - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://laravel.com/docs/10.x/testing.
97. BERGMANN, Sebastian. *5. Organizing Tests — PHPUnit 9.5 Manual* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://docs.phpunit.de/en/9.5/organizing-tests.html.
98. MCCREARY, Jason. *Start testing your Laravel applications* [online]. 2019. [cit. 2023-04-19]. Dostupné z: https://jasonmccreary.me/articles/start-testing-laravel/.
99. AMAZON WEB SERVICES, INC. *What Is an In-Memory Database?* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://aws.amazon.com/nosql/in-memory/.
100. SHADID, Bilal. *Run PostgreSQL in Memory Only | Delft Stack* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.delftstack.com/howto/postgres/run-postgresql-in-memory-only/#postgresql-support-using-a-database-in-memory-or-not.
101. LARAVEL. *HTTP Client - Laravel - The PHP Framework For Web Artisans* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://laravel.com/docs/9.x/http-client#testing.
102. COURSERA. *UI vs. UX Design: What's the Difference? | Coursera* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://www.coursera.org/articles/ui-vs-ux-design.
103. *phpDocumentor* [online]. 2023. [cit. 2023-05-03]. Dostupné z: https://docs.phpdoc.org/guide/getting-started/what-is-a-docblock.html.

Obsah přiloženého archivu

- `readme.txt` — stručný popis obsahu archivu
- `documentation`
  - `api`
    - `builder` — dokumentace API aplikace Builder
    - `deployment-manager` — dokumentace API aplikace Deployment Manager
    - `license-manager` — dokumentace API aplikace License Manager
  - `app` — dokumentace zdrojových kódů aplikace
- `src`
  - `deployment-manager` — zdrojové kódy aplikace
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF