ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Photographix – webová služba pro fotografy
Student:	Bc. Marek Erben
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2021/22

Pokyny pro vypracování

Cílem této práce je návrh a realizace webové služby pro fotografy. Jelikož se jedná o komplexní službu, je cílem práce kromě návrhu a dílčí implementace i řízení týmu studentů bakalářského studia v rámci předmětů BI-SP1 a BI-SP2.

Pro postup viz níže maximalizujte využití studentů týmových projektů, práci koordinujte a sám se aktivně zapojte do vývoje. Tyto činnosti řádně zdokumentujte.

Postupujte v těchto krocích:
* Analyzujte potřeby cílových uživatelů webových služeb pro fotografy.
* Na základě analýzy proveďte vhodný návrh. Pro návrh využijte vhodné modely a postupy.
* Proveďte minimálně prototypovou implementaci ve vámi vhodně zvolených technologiích.
* Podrobte implementaci vhodným testům.
* Na základě testů vylepšete prototyp minimálně do úrovně použitelné beta verze.
* Shrňte přínosy webové služby a navrhněte budoucí směřování projektu pro maximalizaci produkčního užití.

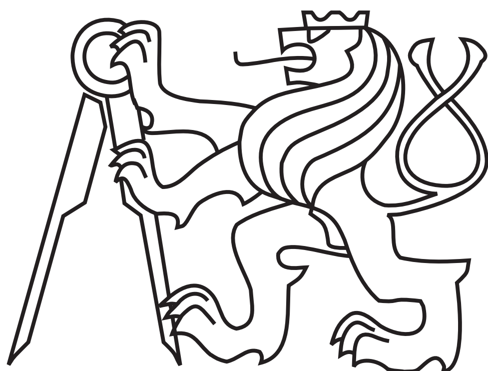
Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 11. září 2020



Diplomová práce

Photographix – webová služba pro fotografy

Bc. Marek Erben

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

7. ledna 2021

Poděkování

Následující řádky chci věnovat poděkování těm, bez kterých bych neměl příležitost psát tuto práci. Především pak svému vedoucímu práce, Ing. Jiřímu Hunkovi, za trpělivé vedení diplomové práce a za záštitu nad řízením projektu v rámci předmětů BI-SP1 a BI-SP2. Zároveň se zde sluší poděkovat celému kolektivu pedagogů a ostatním pracovníkům Fakulty informačních technologií ČVUT v Praze, skrze které jsem mohl pět let čerpat znalosti a zkušenosti z oblasti softwarového inženýrství. I přes to, že to nebylo jednoduchých pět let, budu na ně vždy vzpomínat jako na skvělé období života.

Veliký dík si rovněž zaslouží všichni členové vývojového týmu, kteří mi byli během vývoje implementační části práce nápomocni, jmenovitě se jedná o Michaelu Weberovou, Davida Fencla, Jana Cvrčka, Tadeáše Pálu, Daniela Bartoníčka, Richarda Boldiše a Richarda Kvasnicu.

Tato práce by však nevznikla, nebýt všech, kteří mě po celou dobu studia podporovali – proto můj dík patří především celé mojí rodině, mé milované přítelkyni, Bivojovi, Teovi a všem přátelům a kolegům za jejich neutuchající podporu, jež mě doprovázela po celou dobu mého 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 principu 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 předpisu. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na jejímž základě se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ustanovení § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisu.

V Praze dne 7. ledna 2021 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Marek Erben. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Erben, Marek. *Photographix – webová služba pro fotografy*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a částečnou implementací webové služby pro fotografy. Její přínos spočívá především v usnadnění procesu distribuce digitálních a tištěných fotografií zákazníkům.

První kapitoly práce jsou věnovány analýze a návrhu celé webové služby. Analýza vychází z rozhovorů s fotografy, se kterými jsem zmapoval jejich pracovní postupy. Na jejich základě bylo vytyčeno několik základních požadavků, které by měla webová služba obsahovat. Na základě těchto požadavků byl proveden podrobný návrh. Nemalá část práce se zabývá popisem vývojové infrastruktury, jejíž jádro tvoří nástroj Docker. Její jednotná podoba bývá doceněna v případěch, kdy se na vývoji podílí více lidí současně. Jedna z kapitol je věnována projektovému řízení. V ní popisuji organizaci vývojového týmu, který se skládal ze studentů bakalářských předmětů BI-SP1 a BI-SP2. Poslední kapitoly práce jsou pak věnovány samotné implementaci webové služby. Implementace je zde rozdělena na frontendovou a na backendovou část.

Klíčová slova návrh, implementace, projektové řízení, kontinuální integrace, fotografování, Vue.js, Symfony, Docker, GitLab

Abstract

This diploma thesis deals with the analysis, design and partial implementation of a web service for photographers. Its benefit lies primarily in facilitating the process of distributing digital and printed photos to customers.

The first chapters are devoted to the analysis and design of the entire web service. The analysis is based on interviews with photographers, with whom I mapped their work procedures. Based on them, several basic requirements were set, which the web service should contain. Based on these requirements, a detailed proposal was made. A large part of the work is devoted to the description of the development infrastructure, the core of which is the Docker tool. Its uniform form is usually appreciated in cases where more people participate in the development at the same time. One of the chapters is devoted to project management. In it, I describe the organization of the development team, which consisted of students of bachelor's courses BI-SP1 and BI-SP2. The last chapters of the thesis are then devoted to the actual implementation of the web service. The implementation is divided into frontend and backend part.

Keywords design, implementation, project management, continuous integration, photography, Vue.js, Symfony, Docker, GitLab

Obsah

Obsah xiii

Seznam obrázků xv

Seznam tabulek xix

Úvod 1

1 Analýza 3
- 1.1 Sběr požadavků 3
- 1.2 Definice požadavků 6

2 Návrh 15
- 2.1 Případy užití 15
- 2.2 Architektura služby 35
- 2.3 Doménový model 37

3 Vývojová infrastruktura 47
- 3.1 Docker 48
- 3.2 JetBrains IDE 57
- 3.3 Verzovací systém 64
- 3.4 Vzdálený repozitář zdrojových kódů 68

4 Projektové řízení 77
- 4.1 Metodiky vývoje software 77
- 4.2 Softwarový projekt 83
- 4.3 Nástroje 89

5 Implementace 95
- 5.1 Backend 95

5.2 Frontend 118

6 Závěr **127**

Zdroje **129**

A Seznam použitých zkratek **133**

B Slovník pojmů **135**

C Konfigurační soubory vývojového prostředí **137**

D Ukázka programátorské dokumentace **145**

E Obsah přiloženého média **147**

Seznam obrázků

2.1 Diagram případů užití registrace, přihlášení a obnovy hesla 16
2.2 Diagram případů užití správy uživatelů 18
2.3 Diagram případů užití úpravy uživatelského profilu 19
2.4 Diagram případů užití správy papírů 20
2.5 Diagram případů užití správy tisknutelných formátů 22
2.6 Diagram případů užití správy digitálních formátů 24
2.7 Diagram případů užití správy úrovní kvality 25
2.8 Diagram případů užití správy událostí a sekcí 27
2.9 Diagram případů užití správy fotografií 29
2.10 Diagram případů užití správy objednávek 32
2.11 Ukázka adresářové struktury ZIP archivu nazvětšovaných fotografií 35
2.12 Architektura služby 35
2.13 Doménový model uživatele 37
2.14 Doménový model papírů, úrovní kvality a formátů 39
2.15 Doménový model událostí, sekcí a fotografií 42
2.16 Doménový model objednávek 44

3.1 Výsledky měření výkonnosti Docker kontejneru s různými úrovněmi konzistence sdílených svazků 53
3.2 Výsledky měření výkonnosti Docker kontejneru s využitím NFS pro sdílené svazky 54
3.3 Porovnání mediánů měření výkonnosti Docker kontejneru s různými úrovněmi konzistence sdílených svazků 56
3.4 Zvýraznění a kontrola syntaxe v PHPStorm IDE 58
3.5 Ukázka chytrého našeptávání v PHPStorm IDE 59
3.6 Ukázka integrace VCS do WebStorm IDE 60
3.7 Příklad revize zdrojového souboru, kde se změnilo pouze formátování kódu 60
3.8 Příklad full-textového vyhledávání mezi soubory v PHPStorm IDE 62
3.9 Propojení vývojového prostředí PHPStorm s databází 63

3.10 Každá revize se *vypočte* na základě změn jednotlivých souborů, typické pro SVN *Subversion* 65
3.11 Každá revize je „snímek“ celého projektu, který obsahuje odkazy na soubory aktuální pro danou verzi, typické pro SVN *Git* 66
3.12 Využití diskusí v rámci *merge requests* 70
3.13 Ukázka struktury *pipeline* v rámci *Gitlab CI/CD* náležící k *backendové* části aplikace 71
3.14 Výstup jednotky práce pro výpočet procentuálního pokrytí zdrojového kódu testy 72

4.1 Cyklus vývoje metodou Waterfall 78
4.2 Cyklus vývoje metodou Agile 80
4.3 Inzerát lákající studenty zapojit se do týmu vývojářů webové služby pro fotografy 85
4.4 Diagram popisující pracovní postupy při práci s nástroji *Redmine* a *GitLab* 87
4.5 Tabulka získaných bodů za jednotlivé úkoly v předmětu BI-SP2 . . 89
4.6 Ukázka typického úkolu v systému *Redmine* 91
4.7 Ukázka komunikace v aplikaci *Slack* 92

5.1 Ukázka adresářové struktury projektu *backendové* části aplikace implementované s pomocí frameworku *Symfony* 97
5.2 Ukázka vygenerované dokumentace REST API naší služby 107
5.3 Ukázka práce s nástrojem *Postman* 108
5.4 Sekvenční diagram zachycující způsob autentizace pomocí API tokenu 109
5.5 Diagram zachycující průběh automatického testování a nasazení *backendové* části aplikace 114
5.6 Ukázka zprávy o pokrytí kódu kontrolerů testy 119
5.7 Ukázka stránky pro správu formátů, světlá varianta 123
5.8 Ukázka stránky pro správu formátů, tmavá varianta 123
5.9 Diagram zachycující průběh automatického testování a nasazení *frontendové* části aplikace 125

D.1 Úryvek programátorské dokumentace `README` *backendové* části projektu popisující postup vytváření databázové entity přes příkazovou řádku 145

Seznam ukázek kódu

3.1 Konfigurační soubor Docker kontejnerů potřebných pro chod backendové části aplikace . . . 50
3.2 Konfigurační soubor Docker kontejnerů potřebných pro chod frontendové části aplikace . . . 51
3.3 Konfigurační soubor s pravidly pro formátování kódu frontendové části aplikace . . . 61
3.4 Doplňkový konfigurační soubor pro JetBrains IDE s pravidly pro formátování kódu frontendové části aplikace . . . 61
5.1 Ukázka metody pro získání detailu papíru . . . 110
5.2 Ukázka strukturálního API testu pro ověření správné struktury odpovědi na dotaz ohledně získání kolekce papírů . . . 117
5.3 Ukázka bezpečnostního API testu pro ověření zabezpečení přístupového bodu před nepřihlášenými uživateli . . . 118
5.4 Ukázka jednoduché Vue *single-file* komponenty . . . 121
C.1 Konfigurační soubor Dockerfile definující podobu kontejneru *php* pro chod *frontendové* části aplikace . . . 137
C.2 Konfigurační soubor Dockerfile definující podobu kontejneru *php* pro chod *backendové* části aplikace . . . 138
C.3 Konfigurační soubor webového serveru . . . 139
C.4 Shell skript pro vytvoření NFS serveru . . . 140
C.5 Konfigurační soubor Docker Compose pro spuštění Docker kontejnerů využívajících NFS pro propojení sdílených svazků . . . 141
C.6 Konfigurační soubor pro kontinuální integraci a kontinuální dodávku pro nástroj GitLab CI/CD *backendové* části projektu (1. část) . . . 142
C.7 Konfigurační soubor pro kontinuální integraci a kontinuální dodávku pro nástroj GitLab CI/CD *backendové* části projektu (2. část) . . . 143

C.8 Ukázka konfiguračního souboru nástroje *Deployer* využívaného k nasazení *backendové* části aplikace na cílové prostředí 144

Seznam tabulek

4.1 Obdržená známka vzhledem k počtu získaných Redmine bodů v předmětu BI-SP2 89

Úvod

Volba koníčků lidí každého věku v dnešní době často padne na fotografování. Základna fotografů se neustále zvětšuje ruku v ruce s tím, jak se fotografické vybavení stává dostupnějším. Velmi často tato vášeň přeroste v celoživotní zálibu, někteří pak mají to štěstí, že se fotografováním uživí. Fotograf je k tomuto řemeslu motivován mnohými důvody – některý si libuje v technických nastaveních fotoaparátu, jež ovlivňují výslednou podobu fotografie, někdo miluje ten pocit, kdy stiskne spoušť fotoaparátu, a tím dá vzniknout záznamu okamžiku, který už nikdy znovu nenastane.

Častým problémem mezi fotografy však bývá způsob, jakým pořízené fotografie distribuují mezi své zákazníky. Často je tak distribuují pouze v digitální podobě ve formě archivů, které rozesílají konkrétním zákazníkům. Zákazník si zpravidla nemůže touto formou vybrat pouze takové fotografie, o které opravdu stojí či si konkrétní fotografie přímo objednat k tisku.

V rámci této práce popisuji vývoj webové služby, do níž se bude po jejím dokončení moci přihlásit libovolný uživatel, jenž disponuje internetovým prohlížečem, a pomocí ní bude moci nabízet své fotografie zákazníkům formou jednoduchého e-shopu. Zaregistrovaný fotograf si bude moci definovat množství digitálních i tisknutelných formátů, v nichž budou jeho fotografie zákazníkům k dispozici. Důraz je kladen na co možná největší automatizaci a flexibilitu celého procesu distribuce fotografií.

Tato práce se zabývá postupně kompletní analýzou, návrhem a dílčí implementací služby z pohledu fotografů. S vývojem mi byli nápomocni studenti Fakulty informačních technologií ČVUT v Praze, kteří se k vývoji připojili v rámci předmětů BI-SP1 a BI-SP2 a kteří implementovali některé dílčí části celé webové služby. Formou práce na společném projektu byly studentům osvětleny základní postupy vývoje software, zároveň se seznámili s technologiemi, které jsou během vývoje webové služby využívány a které jsou v této práci postupně popsány. Jelikož se tato práce zaobírá návrhem služby pouze z pohledu fotografa, otevírá se studentům, kteří se na vývoji podíleli, možnost

navázat na další vývoj v rámci svých závěrečných prací.

Velký důraz je v této práci kladen na popis doporučených postupů, které by se během vývoje neměly opomíjet, ať už jde o nastavení jednotného vývojového prostředí v rámci projektové infrastruktury, nebo o nastavení minimálních standardů, jež klademe na podobu zdrojového kódu.

Kromě znalostí, jež jsem nabyl během studia na Fakultě informačních technologií ČVUT v Praze, jsem během vývoje uplatnil i své zkušenosti s fotografováním, kterému jsem se aktivně věnoval již na střední škole.

Kapitola 1

Analýza

Na následujících řádcích shrnuji své poznatky ze zkoumání toho, jak by měla naše aplikace vypadat. To v prvé řadě závisí na funkcích, kterými by taková aplikace měla disponovat. Jaké funkce budou pro naši aplikaci nezbytné a jaké bychom mohli oželet, bude předmětem podkapitoly 1.1.

Na úvod si dovolím znovu připomenout, že celá práce se zabývá analýzou, návrhem a implementací služby z pohledu *fotografů*. Návrh a implementaci služby z pohledu *klientů* bude předmětem samostatné bakalářské práce, o které více referuji v závěru této práce v kapitole 6.

1.1 Sběr požadavků

V úvodu této podkapitoly mi dovolte zmínit, že mám s probíraným tématem bohaté osobní zkušenosti. Od střední školy jsem se několik let aktivně věnoval fotografování a sám jsem ve své době hledal možnosti, jak dostat hotové fotografie mezi své zákazníky, ať už v elektronické formě, nebo jako vytištěné fotografie. To mi, dle mého názoru, poskytuje poměrně dobrou startovní pozici pro návrh software, kterým se v této práci zabýváme. Považoval jsem však za zůležité oslovit fotografy, kteří mohou mít k procesu zpracování a distribuce fotografií odlišný přístup. Rozbor pohledu na věc každého z nich sepisuji v samostatné podkapitole 1.1.1.

1.1.1 Kvalitativní výzkum

Před výběrem metody, pomocí které budu shlukovat klíčové požadavky pro návrh naší aplikace, jsem stál před rozhodnutím, zda použít tzv. *kvalitativní* či *kvantitativní* metodu výzkumu. Dle [1] lze *kvantitativní* metodu výzkumu charakterizovat jako *extenzivní* šetření zkoumané skutečnosti, tedy snahu o porozumění problému pouze povrchně, zato však na velkém vzorku respondentů.

Podle stejného zdroje [1] *kvalitativní* metoda výzkumu probíhá na mnohem menším vzorku respondentů, zato však jde v tématu mnohem více do

hloubky, sběr požadavků touto metodou je však časově více náročný. Na druhou stranu se touto metodou dá vyspecifikovat mnohem více konkrétních požadavků, které bychom *kvantitativní* metodou vůbec neměli šanci získat.

Není proto zřejmě žádné překvapení, že pro sběr požadavků byla v našem případě zvolena *kvalitativní* metoda. Kvalitativní výzkum v kontextu sběru požadavků v praxi probíhal jako přátelský rozhovor s fotografy, kteří mi popisovali, jakým způsobem s fotografiemi pracují, jaký je jejich hlavní předmět fotografování, jakým způsobem distribuují hotové fotografie cílovým zákazníkům, atd.

Za účelem získání co možná největší škály názorů jsem oslovil tři fotografy, z nichř každý k fotografování přistupuje jinak. Jejich charakteristikou se zabývají následující řádky:

Začínající fotografka, 17 let Fotografka se zabývá fotografováním již několik let, nicméně kvůli školní docházce ještě nedostala tolik příležitostí k zakázkovému focení a se svou kariérou teprve začíná. I přes to má za sebou několik fotografování svateb a maturitních plesů.

Fotografie vyhotovené na zakázku vždy odevzdávala pouze v elektronické podobě nasdílením komprimovaných fotografií přes služby typu *Google Drive* či *Ulož.to*, a to jednomu ze zadavatelů zakázky do soukromé zprávy. To, jakým způsobem se její fotografie dostanou ke *všem* návštěvníkům fotografované akce, již neřeší.

Pokud si zákazník přeje některé fotografie vyhotovit v papírové podobě, odkáže ho fotografka na firmy v okolí, které jsou schopny fotografie vytisknout. Nad přímým zprostředkováním prodeje papírových fotografií zatím nepřemýšlela. Pokud by však existoval nástroj, který by jednoduše dokázal dostat její fotografie mezi cílové zákazníky tak, aby si zákazník mohl fotografie objednat k vytištění, případně si konkrétní fotografie přímo stáhnout v plném rozlišení, jistě by ho zařadila do své softwarové výbavy fotografa. Jak sama říká: *„Až se fotografování začnu věnovat na plný úvazek, bude takové řešení nutností“.*

Fotografie upravuje pomocí software *Adobe Lightroom*. Svými fotografiemi se prezentuje skrze sociální sítě *Instagram* a *Gurushots*.

Fotograf, pro něhož je fotografování hlavně koníčkem, 29 let Fotograf se zabývá fotografováním od mládí, kromě digitální fotografie aktivně fotí i tradičními metodami, kdy si fotografie vyvolává ve své domácí temné komoře. Fotografování nicméně není jeho hlavní zdroj příjmů. Je to spíše koníček s tou výhodou, že si s jeho pomocí občas přivydělává fotografováním koncertů a jiných společenských událostí.

Fotograf je při předávání fotografií zvyklý pracovat se službou *Google Drive* a *Dropbox*. V rámci nich si vede adresářovou strukturu s fotografiemi. K jednotlivým adresářům s fotografiemi je schopný generovat odkazy, které poté zasílá zákazníkovi, jenž si dané fotografie může prohlédnout a stáhnout.

Nebylo by podle něj od věci nějakým způsobem provázat navrhnovanou službu právě s úložišti typu *Google Drive* či *Dropbox*.

Sám fotograf aktivně nenabízí zákazníkům možnost objednat si fotografie přímo v papírové podobě. Pokud se tak se zákazníkem domluví, není pro něj problém dát fotografie vytisknout některou firmou, jež se tím zabývá. Vytištění fotografií zprostředkuje pouze za cenu nákladů samotného tisku, sám si ze zprostředkování zpravidla žádnou provizi nebere.

Na fotografiích jako takových ho zajímá jejich technická stránka, ocení informace o parametrech fotografie jako např. clonové číslo, hodnota citlivosti ISO[1] či ohnisková vzdálenost objektivu.

K úpravě fotografií využívá rovněž software *Adobe Lightroom*. V souvislosti s použitým softwarem zmínil jednu funkcionalitu, kterou v *Adobe Lightroom* postrádá – barevné schéma aplikace (*light mode*, *dark mode*) se nedokáže automaticky přizpůsobit zvolenému systémovému nastavení.

Profesionální fotografka s dlouholetou praxí, 37 let Profesionální fotografka má ze všech respondentů největší zkušenosti a na množství podnětů od ní získaných je to znát.

Za svou dlouholetou kariéru má za sebou fotografování nespočtu svateb, rodinných oslav, plesů, zakázkových focení v ateliéru a v neposlední řadě fotografování školek a škol před letními prázdninami.

Fotografka disponuje vybaveným ateliérem, ve kterém nabízí skupinové focení vhodné pro jednotlivce, páry i rodiny. Takové fotografie jsou pak častým dárkem pod vánočním stromečkem. Fotografie je schopna tisknout na kvalitní foto tiskárně přímo ve svém ateliéru, maximálně však do rozměru, který odpovídá formátu A4[2]. Větší tisky si nechává na zakázku vyhotovit ve společnosti, která se tiskem velkých formátů zabývá.

Na moji otázku *jaké nástroje používá k distribuci hotových fotografií cílovým zákazníkům* odpověděla, že v minulosti sama vyhledávala službu, skrze kterou by nahrávala náhledy svých fotografií. Mezi jejími požadavky byla i možnost objednat konkrétní počet kusů fotografie v určitém rozměru. Shrnutí takové objednávky by jí přišlo na její e-mail. Ve té době ale žádnou službu nenašla, a tak si na zakázku nechala vyhotovit jednoduchý webový systém, který její požadavky splňoval. Tento systém v současné době však již jejím požadavkům neodpovídá (nereprezentativní design, drahá údržba, omezené nastavení nabízených rozměrů fotografie, . . .). Náhradu za tento stále používaný systém prozatím nenašla.

Díky této zkušenosti mi předala několik cenných poznatků, které bych během návrhu naší služby neměl rozhodně opomenout. Mezi základní patří možnost shlukovat fotografie do jakýchsi *kolekcí*, které budou obsahovat fotografie z jedné události (svatba, maturitní ples, . . .). Tyto kolekce by bylo

[1] The ISO (International Organization of Standards) 12232:2019 standard

[2] 297 × 210 mm

dobré nějakým způsobem *chránit*, aby její obsah nemohl vidět každý. Je však třeba myslet i na možnost mít kolekce *veřejně přístupné*, v rámci nich by se nabízely fotografie z veřejně přístupných společenských akcí.

Často v její praxi dochází k situacím, kdy nafotí velké množství fotografií, které před finálním předáním zákazníkovi vyžadují poměrně zdlouhavé úpravy a retuše. Dříve upravovala bez výjimky všechny vybrané fotografie. Zákazník si však z těchto vybraných a upravených fotografií ve finále vybral pouze zlomek k vytištění. Podstatná část upravených fotografií tedy přišla vniveč. Fotografka by tedy ocenila mechanismus, který by umožňoval vystavit náhled neupravené fotografie, jež by poté šla objednat „k úpravě“. Teprve po úpravě a retuších by bylo možné danou fotografii odeslat zákazníkovi, případně takovou fotografii přímo vytisknout.

Jelikož fotografka některé fotografie tiskne na vlastní tiskárně a některé musí dávat k vytištění další firmě, ocenila by i rozdělení nabízených formátů fotografií na takové, které je schopna vytisknout sama a které ne. Někteří zákazníci vyžadují i specifický papír, na který má být fotografie vytištěna. Takový papír může mít specifickou strukturu (např. hladká, perleť, …) nebo povrchovou úpravu (lesk, mat, …).

Neméně podstatné byly i její připomínky k cenotvorbě nabízených fotografií. V její praxi totiž nastávají v podstatě dvě věci:

- zákazník zaplatí za fotografování předem, tím pádem mají *všichni* nárok na *digitální* kopie fotografií zdarma, za objednané *tištěné* fotografie platí zákazník navíc,
- zákazník nezaplatí předem a fotograf je odměněn pouze na základě prodeje *digitálních* i *tištěných* fotografií.

Je tedy třeba brát ohled na dobrý návrh tvorby cen nabízených fotografií, aby se výše zmíněné požadavky zohlednily.

Fotografie upravuje především pomocí software *Adobe Lightroom*, pokročilé retuše a doplňkové grafické práce pak pomocí *Adobe Photoshop*. Jako fotografka se prezentuje na vlastních webových stránkách a sociálních sítích *Facebook* a *Instagram*.

1.2 Definice požadavků

V následující podkapitole 1.2.1 přetavuji výše zmíněné poznatky ve funkční požadavky, které by měly definovat všechny klíčové funkce budoucího systému. Z funkčních požadavků vychází několik požadavků nefunkčních, ty jsou rozepsány v rámci podkapitoly 1.2.2.

1.2.1 Funkční požadavky

Následující podkapitola obsahuje soupis funkčních požadavků, jež by mohly být vyžadovány typickými uživateli našeho systému. Velká část z nich vychází z kvalitativního výzkumu, jenž je popsán v podkapitole 1.1.1. Během formulace funkčních požadavků jsem však zúročil i své letité zkušenosti s fotografováním a své zkušenosti s distribucí hotových fotografií cílovým zákazníkům – ať už ve vytištěné, či digitální formě.

FR 1 Registrace uživatelů

Každý příchozí návštěvník naší služby má možnost založit si účet. Při *registraci* vyplňuje základní uživatelské údaje, tedy jméno, příjmení, kontaktní e-mail a přihlašovací heslo. Úspěšná registrace bude podmíněna jejím potvrzením, které je uživateli zasláno na e-mail. Během vyplňování atributů, které z logických důvodů musí být unikátní (*e-mail*, *uživatelské jméno*), se na pozadí provádí jejich validace v reálném čase.

FR 2 Přihlášení uživatelů

Uživatel má možnost se do služby *přihlásit* pomocí zaregistrovaného uživatelského jména a hesla. Množství pokusů o přihlášení je omezen na malý počet, aby se eliminovala možnost prolomení hesla k danému uživatelskému jménu hrubou silou. Z přihlašovacího formuláře jsou uživateli zobrazovány odkazy na stránky s obnovením hesla a s registračním formulářem. V budoucí verzi by mělo být možné provádět přihlášení skrze existující poskytovatele autentizace, např. Google.

FR 3 Úprava údajů přihlášeného uživatele

Uživatel může *upravit své údaje*, které uvedl při registraci. Navíc si může nahrát svého avatara a změnit jazykovou mutaci. Některé informace v profilu nejde změnit z bezpečnostních důvodů s okamžitou platností, konkrétně se jedná o atribut *e-mail* a *heslo*. Během změny atributů, které z logických důvodů musí být unikátní (*e-mail*, *uživatelské jméno*), se na pozadí provádí jejich validace v reálném čase. Uživatel v rámci svého profilu může nahrát svůj vodoznak, se kterým poté mohou být generovány náhledy fotografií.

FR 4 Správa papírů

Uživatel si má možnost definovat libovolný počet *papírů*, na které má možnost tisknout fotografie vybrané zákazníkem. Papíry se dají dle svého formátu dělit na dva základní druhy:

Arch je list papíru s předem danými pevnými rozměry obou stran, typickým zástupcem může být papír A4 s rozměry 297×210 mm.

Role je balení papíru určené pro tiskařské plotry, u něhož se většinou udává pouze jeho *šířka*, např. 600 mm. Na roli se fotografie tisknou v sadě, poté je nutné ji rozřezat.

Důležitým parametrem papíru je jeho gramáž, která je udávána v jednotkách hmotnosti na plochu, typicky g/m^2. Aby mohla naše služba automaticky počítat ceny nabízených fotografií, je nutné u papírů evidovat rovněž jeho cenu, ta se udává typicky v jednotkách Kč/m^2 a tvoří základní stavební kámen výpočtu výsledné ceny fotografie.

Papíry obsahují mimo jiné atributy, které udávají, o jaký typ *povrchové úpravy* a *textury* se v daném případě jedná. Hodnotu těchto atributů uživatel vybírá z předdefinovaných hodnot.

FR 5 Správa tisknutelných formátů

Uživatel definuje formáty, v nichž může být fotografie *vytištěna* na daný papír. U nich je klíčovým parametrem jeho rozměr. Ten lze definovat dvěma způsoby:

Pevný rozměr má pevně zadanou šířku i výšku tisknuté fotografie. Tohoto typu formátu se využívá v případě, kdy je potřeba výslednou fotografii oříznout na pevný rozměr bez ohledu na její originální poměr stran.

Flexibilní rozměr umoňuje zadat délku pouze jedné strany. Uživatel zadá délku a zvolí, zda se daná délka týká delší či kratší strany fotografie. Délka druhé strany se pak dopočítá vždy automaticky na základě poměru stran originální fotografie.

Tisknutelné formáty obsahují libovolný počet papírů, na které může být výsledná fotografie vytištěna.

Vzhledem k tomu, že ne každý fotograf disponuje vlastní tiskárnout, může si uživatel „vypůjčit“ formáty od jiných uživatelů, kteří tuto možnost mají. Výpůjčku formátu daným uživatelem musí schválit majitel formátu. Fotografie objednané ve vypůjčeném formátu se zobrazí i majiteli formátu jako nové objednávky.

FR 6 Správa digitálních formátů

Kromě tisknutelných formátů má uživatel možnost definovat *digitální* formáty. Fotografii objednanou ve zvoleném digitálním formátu lze stáhnout jako digitální soubor v zadaném rozlišení, které je udáváno v jednotkách *megapixel* (MPx), případně je možné fotografii stáhnout v původním rozlišení. V rámci digitálních formátů lze také rozhodnout, zda se má digitální fotografie stáhnout s vodoznakem či bez něj. Nesmí chybět ani cena, kterou zákazník platí za možnost stáhnout fotografii nabízenou v daném formátu.

FR 7 Správa úrovní kvality

Uživatel definuje libovolný počet tzv. *úrovní kvality*. Jeho jádrem je číselný koeficient, jenž je vždy alespoň 1. Úroveň kvality lze přiřadit k celé události nebo jen konkrétní fotografii. Daným číselným koeficientem úrovně kvality je pronásobena cena za papír, čímž vznikne výsledná cena za tištěnou fotografii. Uživatel má tak možnost některé fotografie pomocí tohoto mechanismu nacenit více oproti jiným.

FR 8 Vytváření událostí

Uživatel vytváří *události*, tedy jakási fotoalba, v nichž se shlukují fotografie z jedné fotografické akce. Událost obsahuje kromě názvu i stručný popis a datum konání události. Mezi další klíčové atributy patří *stav* události, jenž může nabývat následujících hodnot:

Aktivní stav zapříčiní, že událost je viditelná uživatelům, kteří k ní mají přístup.

Uzavřený stav skryje událost před všemi uživateli, ať k dané události mají či nemají povolen přístup.

Každé události lze nastavit úroveň jejího *zabezpečení*, jež může nabývat jednoho z následujících stavů:

Veřejná úroveň umožňuje prohlížet obsah události *všem* uživatelům. Takový uživatel nemusí být ani přihlášen do aplikace pod svým účtem.

Chráněná úroveň uživatele před vstupem do události zavazuje k zadání unikátního PIN kódu. Tento PIN kód je generován s vytvořením každé události automaticky. Uživatel přistupující k *chráněné* události nemusí být přihlášen do aplikace pod svým účtem.

Soukromá úroveň zabezpečení události umožňuje uživateli nastavit, který konkrétní uživatel má oprávnění vidět obsah dané události. Uživatel je s událostí spojen na základě svého e-mailu, na který je zároveň zasláno upozornění s informacemi o vpuštění do události. Z toho vyplývá, že uživatel přistupující k *soukromé* události musí být přihlášen do aplikace pod svým účtem.

FR 9 Vytváření sekcí událostí

Každá událost obsahuje libovolný počet *sekcí*. Slouží pro lepší logické rozřazení fotografií v rámci události. Sekce může být pojmenovaná a lze jí nastavit libovolné pořadí v rámci události.

FR 10 Nahrávání fotografií

V rámci jednotlivých sekcí události je možné nahrávat libovolný počet fotografií. Tyto fotografie jsou v rámci sekce řazeny fixně dle svého originálního názvu. Akceptovány jsou fotografie ve formátu JPEG[3].

FR 11 Přehled nahraných fotografií

V detailu události se nachází přehled všech jejich sekcí, v rámci kterých se zobrazují miniatury nahraných fotografií. V přehledu fotografií se dá přepínat velikost zobrazení jednotlivých miniatur. Dále se dá nastavit, zda se má miniatura zobrazit v originálním poměru stran, či zda má vyplnit celý jí přidělený prostor.

Detail nahrané fotografie obsahuje základní informace, které je možno vyextrahovat z EXIF[4] metadat nahrávaného souboru. Mezi základní informace, které jsou v detailu fotografie zobrazeny, patří:

- rozměry,
- velikost,
- clonové číslo objektivu,
- expoziční čas fotoaparátu,
- ohnisková vzdálenost objektivu,
- hodnota citlivosti ISO,
- výrobce a model fotoaparátu,
- výrobce a model objektivu.

V detailu fotografie ji lze stáhnout originální soubor v plném rozlišení.

FR 12 Nahrávání revizí fotografie

K jednotlivým nahraným fotografiím je možné nahrát jejich revizi. Uživatel má možnost nastavit, zda se dá objednat *jakákoliv* verze dané fotografie, či zda se dá objednat pouze verze *poslední*. Motivace pro definici tohoto funkčního požadavku se dá demonstrovat na následujícím příkladu:

Nabídka pouze poslední verze může být využita fotografem tehdy, když zveřejní fotografie, ale dodatečně přidá nějakou její úpravu (lepší korekce barev, dodatečné doostření, ...).

[3]Skutečným názvem typu souboru je JFIF, což znamená JPEG File Interchange Format. Zkratka JPEG znamená Joint Photographic Experts Group, což je vlastně konsorcium, které tuto kompresi navrhlo.

[4]Exchangeable Image File Format

Nabídka všech verzí najde uplatnění v případě, kdy fotograf vytvoří více variant úprav jedné fotografie (jiné barevné podání, jiný ořez, ...), z nichž si může zákazník vybrat.

FR 13 Přehled objednávek

Uživatel má přehled o objednaných fotografiích, které se shlukují v *objednávkách*. U objednávek se evidují údaje zákazníka, zvláště pak jméno, příjmení, e-mail a případně telefonní číslo. Objednávka obsahuje také informace o objednaných fotografiích včetně množství objednaných fotografií pro daný formát a papír. Každá objednávka disponuje také předpočítanou celkovou cenou k úhradě a případnou poznámkou od zákazníka.

FR 14 Export objednávek

Export objednávek je možné provést dvojím způsobem:

Strukturované informace o objednávce v jednom ze standardních formátů pro reprezentaci tabulkových dat, konkrétně XLSX či CSV. Exportovaný dokument obsahuje veškeré informace o objednaných fotografiích včetně údajů o zákazníkovi, který konkrétní objednávku provedl. Exportovat lze více objednávek najednout, v takovém případě je vygenerován archiv, jenž obsahuje pro každou objednávku samostatný XLSX/CSV soubor.

Nazvětšované fotografie[5] v adresářové struktuře, která je dána volbou uživatele. Nazvětšované fotografie lze stáhnout seskupené podle *zákazníka*, *formátu*, *papíru*, *události* a *sekce události*. Uživatel může takto vygenerovaná data přímo zadat k tisku a nemusí se jejich nazvětšováním zabývat na svém počítači.

1.2.2 Nefunkční požadavky

Z výše formulovaných funkčních požadavků vyplývá i několik požadavků *nefunkčních*. Ty zpravidla vyžadují, abychom se v rámci jejich formulace zamysleli nad výkonnostní efektivitou, škálovatelností, spolehlivostí, rozšiřitelností či bezpečností naší aplikace.

NFR 1 Lokalizace

Služba by měla být lokalizována do několika jazyků, na což bude během vývoje brán zřetel.

[5] Ve skutečnosti jde zpravidla o jejich *zmenšení*. Termín *nazvětšovat* pochází z dob, kdy se fotografie vyvolávaly v temné komoře a bylo je potřeba, fotgrafové pracující s digitální fotografií tento temín přejali a používají ho v kontextu úpravy fotografie na kýženou velikost.

NFR 2 Horizontální škálovatelnost

Vzhledem k nahrávání velkého množství dat, především ve formě fotografií, je potřeba myslet na vhodou volbu řešení, které se nám postará o schopnost horizontálního škálování, co se kapacity úložiště týče.

NFR 3 Generování náhledů originálních fotografií

Je nepraktické zobrazovat nahrané fotografie zpět uživateli v jejich původním rozlišení. Takové soubory mohou být totiž velice datově náročné. Během implementace je potřeba klást důraz na efektivní a jednoduché generování náhledů nahraných fotografií. Zároveň je třeba myslet na požadavek generovat náhledy fotografií s případným vodoznakem uživatele.

NFR 4 Asynchronní provádění vybraných akcí

Z výše vypsaných poznatků již dopředu vyplouvá na povrch několik případů, kdy je třeba vyčlenit některé sekvence kódu do tzv. *asynchronních procesů*. Ty se provádějí nezávisle na hlavním procesu. U hlavního procesu se obvykle klade důraz na rychlou odpověď na zadaný požadavek. Akce, které potřebují pro své dokončení velkou porci času, budeme mít snahu vyčleňovat právě do asynchronních procesů. Typické zástupce takových akcí představuji níže:

Mazání fotografií, sekcí, událostí s sebou vždy nese operaci smazání fyzického souboru fotografie ze souborového systému. Vzhledem k nefunkčnímu požadavku na dobrou škálovatelnost úložiště je velmi pravděpodobné, že toto úložiště bude na úplně jiném serveru. Každé smazání fotografie s sebou tudíž nese minimálně jeden HTTP požadavek. Při snaze o smazání většího počtu fotografií v rámci sekce či události se bavíme o řádově desítkách až stovkách takových HTTP požadavků. Při synchronním mazání velkého množství fotografií by byl klient[6] vystaven neúnosně dlouhé době, než by se od serveru dočkal odpovědi.

Export objednávek spočívá v načtení fotografií ve vysokém rozlišení, dále v jejich následném naškálování na objednaný rozměr a nakonec v zabalení výsledných souborů do archivu. Z toho vyplývá, že tento proces bude jistě velmi časově náročný.

Odesílání e-mailů na první pohled nezní jako nikterak časově náročná činnost. Problém opět nastává ve chvíli, kdy je potřeba e-mailů poslat najednou více, což se může ve službě našeho typu lehko stát.

[6] V kontextu architektury klient–server

NFR 5 Zabezpečení aplikace

Jelikož jednu instanci naší aplikace bude využívat najednou velké množství uživatelů, je třeba dbát na důslednou izolaci jejich identity a dat, tedy aby jeden uživatel nemohl neoprávněně vystupovat jako uživatel jiný, a aby měl každý přístup výhradně ke zdrojům, které patří pouze jemu (v našem případě např. originální soubory nahraných fotografií či informace o objednávkách). Tento závazek lze spolehlivě kontrolovat některou formou automatizovaných testů.

NFR 6 Webová aplikace

Základním předpokladem toho, že bude naše služba využívána mnoha uživateli je její přístupnost prostřednictvím internetu. A to ideálně bez nutnosti instalace dalšího doplňkového software. Aplikace by tedy měla být přístupná z internetu pomocí běžného webového prohlížeče.

Kapitola 2

Návrh

Nyní je ten správný čas přetavit znalosti nabyté v předchozí kapitole 1 do formálního návrhu naší služby. V rámci podkapitoly 2.1 popisuji jednotlivé *případy užití*. Na základě sepsaných *případů užití* a *funkčních* i *nefunkčních* požadavků není od věci zamyslet se nad architekturou celé aplikace, ta je podrobněji rozebrána v podkapitole 2.2. V samostatné podkapitole 2.3 pak představuji návrh *doménového modelu*.

2.1 Případy užití

Před výpisem samotných případů užití není od věci definovat základní uživatelské role, jež budou v systému figurovat. Všichni uživatelé nemusí disponovat pouze jednou rolí, ale mohou jich mít zpravidla více. To je výhodné zvlášť pro kombinaci rolí *fotograf* a *tiskárna*.

Administrátor Administrátor má základní přehled o dalších registrovaných uživatelích. Zároveň je mu umožněna jejich další správa, smí tedy vytvářet nové uživatele, či jiným uživatelům odepřít přístup do systému.

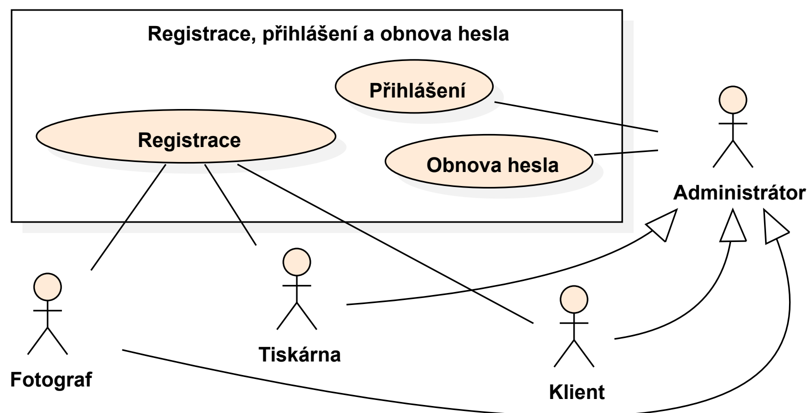
Fotograf Fotograf má možnost nahrávat své fotografie v rámci *událostí* a tyto fotografie nabízet *klientům*, případně všem neregistrovaným návštěvníkům. Fotografie nabízí ve formátech, které si sám vytvořil, případně si je *vypůjčil* od jiného uživatele s rolí *tiskárna*.

Tiskárna Tiskárna je specifická role, jež umožnuje uživateli definovat si vlastní *papíry* a z nich odvozené *formáty*, jež může využít uživatel s rolí *fotograf* a nabízet je spolu s fotografiemi *klientům*.

Klient Registrovaný uživatel, jenž má právo prohlížet a objednávat fotografie, ke kterým má přístup.

2.1.1 Registrace, přihlášení a obnova hesla

Diagram případů užití pro registraci, přihlášení a obnovení hesla uživatele je zobrazen na obrázku 2.1.



Obrázek 2.1: Diagram případů užití registrace, přihlášení a obnovy hesla

UC 1 Registrace

Uživatel se může do služby registrovat vyplněním registračního formuláře. Registrace není umožněna uživatelům s rolí *administrátor*. Základní údaje potřebné při registraci jsou:

- jméno,
- příjmení,
- uživatelské jméno,
- e-mail,
- heslo,
- zaměření.

Volbu zaměření vybírá uživatel z nabídky předpřipravených možností:

Fotograf zaručuje, že bude uživatel zaregistrován s rolí *fotograf*.

Tiskárna zaregistruje uživatele s rolí *tiskárna*.

Fotograf s možností tisknout nastaví uživateli po registraci role *fotograf* i *tiskárna*.

Registraci je nutno potvrdit kliknutím na potvrzující odkaz, jenž je po registraci zaslán uživateli na daný e-mail. Registrace uživatele s rolí *klient* je nad rámec návrhu této práce.[7]

UC 2 Přihlášení

Uživatel se do služby registruje pomocí uživatelského jména a hesla.

UC 3 Obnova hesla

V případě, že uživatel zapomene své heslo, je pro něj k dispozici formulář, pomocí něhož může své heslo obnovit. Samotný proces obnovy hesla se skládá ze dvou fází:

Zaslání žádosti o změnu hesla způsobí odeslání odkazu na zadaný e-mail. Tento odkaz přesměruje uživatele na formulář, v němž vyplní své nové heslo.

Zadání nového hesla probíhá v rámci formuláře, na který je uživatel přesměrován odkazem, jenž obdržel v předchozí fázi.

2.1.2 Správa uživatelů

Diagram případů užití správy uživatelů je k vidění na obrázku 2.2. Všechny akce spojené se správou uživatelů jsou k dispozici pouze uživateli s rolí *administrátor*.

UC 4 Zobrazení seznamu uživatelů

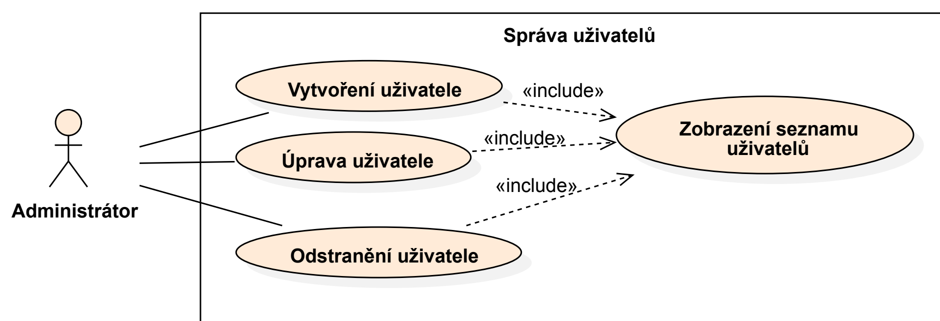
Uživatel s rolí *administrátor* může vidět seznam všech uživatelů, kteří se v naší službě nacházejí. Vzhledem k potenciálně velkému množství uživatelů je nutno myslet na stránkování záznamů.

UC 5 Vytvoření uživatele

Administrátor může vytvořit uživatele. Pro vytvoření uživatele je nutné vyplnit následující atributy:

- jméno,

[7] Tato práce se zabývá především návrhem a implementací administrační části služby pro role *fotograf*, *tiskárna* a *administrátor*.



Obrázek 2.2: Diagram případů užití správy uživatelů

- příjmení,
- uživatelské jméno,
- e-mail,
- heslo,
- seznam rolí, kterými bude uživatel disponovat.

UC 6 Úprava uživatele

Administrátor má možnost upravit existujícího uživatele. Atributy jsou shodné s případem užití pro vytvoření uživatele. Je zde však omezení, že administrátor nesmí upravovat údaje jiného administrátora.

UC 7 Odstranění uživatele

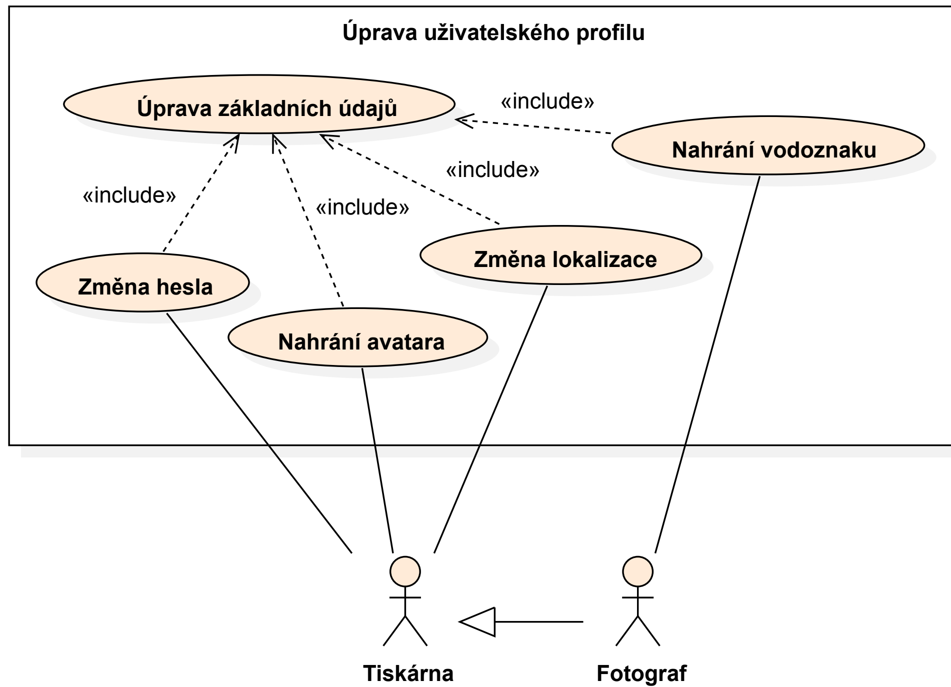
Administrátor má možnost odstranit libovolného uživatele, jedinou výjimkou je opět případ, kdy ani administrátor nemůže odstranit jiného administrátora. Odstraněním uživatele budou ztracena všechna uživatelské data včetně jeho *událostí* a nahraných *fotografií*.

2.1.3 Úprava uživatelského profilu

Diagram případů užití pro úpravu uživatelského profilu je k vidění na obrázku 2.3. Úprava uživatelského profilu je pro všechny uživatelské role podobná, liší se jen v detailech.

UC 8 Úprava základních údajů

Atributy upravované v této sekci jsou:



Obrázek 2.3: Diagram případů užití úpravy uživatelského profilu

- jméno,
- příjmení,
- uživatelské jméno,
- e-mail.

Během vyplňování nového *uživatelského jména*, respektive nového *e-mailu* dochází k validaci unikátnosti těchto atributů v reálném čase, neboť tyto atributy musí být v rámci naší služby unikátní.

UC 9 Změna hesla

Uživatel má možnost změnit si v rámci úpravy profilu své heslo.

UC 10 Nahrání avatara

Uživatel může nahrát svého avatara. Tohoto avatara může odstranit nebo nahradit nahráním avatara nového.

UC 11 Změna lokalizace

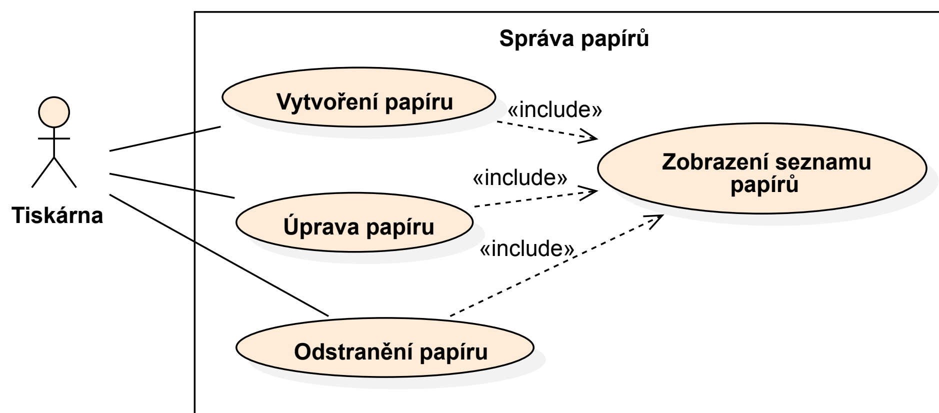
Uživatel má v rámci uživatelského profilu možnost změnit lokalizaci svého profilu. Po jeho změně se mu uživatelské rozhraní „přepne“ do zvoleného jazyka.

UC 12 Nahrání vodoznaku

Tato funkce je exkluzivní pro uživatele s rolí *fotograf*, jenž má v rámci uživatelského profilu možnost nahrát vlastní vodoznak, s nímž budou následně generovány miniatury jeho fotografií.

2.1.4 Správa papírů

Správa papírů je výhradní doménou uživatelů s rolí *tiskárna*. Diagram případů užití správy papírů je k vidění na obrázku 2.4.



Obrázek 2.4: Diagram případů užití správy papírů

UC 13 Zobrazení seznamu papírů

Uživatel s rolí *fotograf* má možnost zobrazit si seznam papírů ve formě karet, každá karta obsahuje kompletní informace o daném papíru.

UC 14 Vytvoření papíru

Uživatel s rolí *fotograf* může vytvořit nový papír. Jakým způsobem jsou u papíru definovány jeho rozměry, závisí na jeho *typu*. Typy papíru mohou být následující:

Arch vyžaduje zadání rozměru šířky i výšky papíru.

Role vyžaduje zadání pouze svojí šířky.

Další atributy papíru jsou:

- název,
- cena v jednotkách Kč/m^2,
- hmotnost v jednotkách g/m^2,
- povrchová úprava,
- textura.

Hodnotu atributu *povrchová úprava* uživatel vybírá z předpřipravených možností:

- lesk,
- polomat,
- mat.

Hodnotu atributu *textura* uživatel vybírá z předpřipravených možností:

- hladká,
- pearl,
- strukturovaná.

UC 15 Úprava papíru

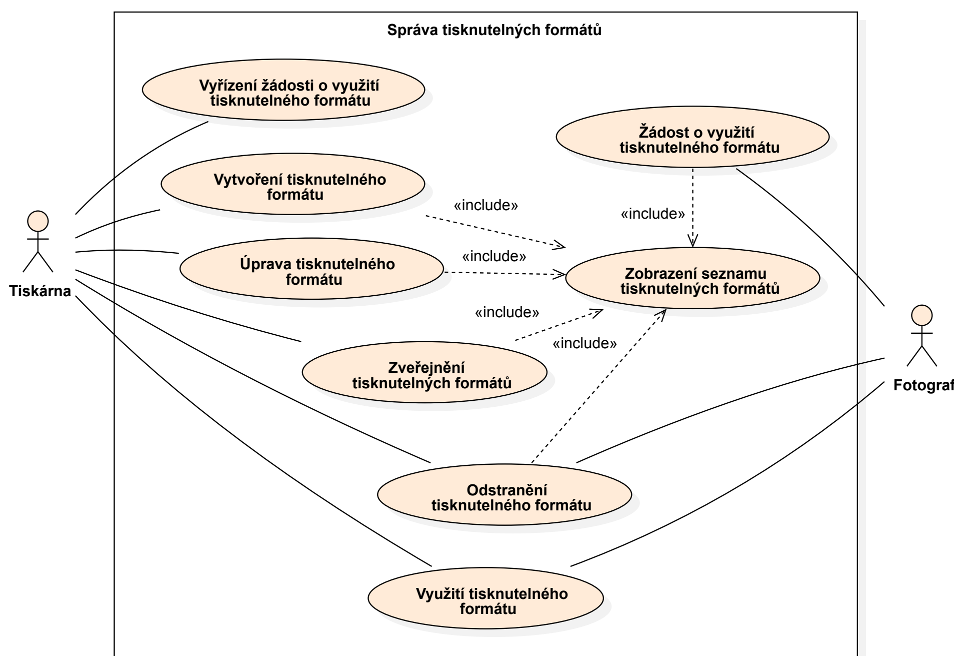
Uživatel s rolí *tiskárna* může všechny své papíry upravovat. Úprava papírů probíhá se stejnými atributy, které jsou popsány v předchozím případu užití pro *vytvoření papíru.*

UC 16 Odstranění papíru

Uživatel s rolí *tiskárna* může smazat libovolný z jím vytvořených papírů. Odstranění papíru logicky zapříčiní i jeho odstranění z *formátů*, které daný papír využívají.

2.1.5 Správa tisknutelných formátů

Grafické znázornění diagramu použití správy tisknutelných formátů je k vidění na obrázku 2.5. Vytvářet je smějí výhradně uživatelé s rolí *tiskárna*, nicméně využít je (díky sdílení) mohou i *fotografové.* Tisknutelné formáty definují, v jakém konkrétním rozměru a na jakém konkrétním papíru se dá fotografie objednat k *vytištění.*



Obrázek 2.5: Diagram případů užití správy tisknutelných formátů

UC 17 Zobrazení seznamu tisknutelných formátů

Uživatel s rolí *tiskárna* má možnost zobrazit seznam tisknutelných formátů. Ty jsou zobrazeny ve formě karet s kompletními informacemi o daném tisknutelném formátu.

Uživatel s rolí *fotograf* si může zobrazit pouze takové formáty, které nastavili ostatní uživatelé s rolí *tiskárna* jako *veřejné*.

UC 18 Vytvoření tisknutelného formátu

Uživatel s rolí *tiskárna* vytváří tisknutelný formát. Klíčovým atributem tisknutelného formátu je jeho *rozměr*, jenž závisí na *typu* tisknutelného formátu. Typ tisknutelného formátu může nabývat hodnot:

Pevný rozměr vyžaduje vyplnění kratší i delší strany formátu.

Flexibilní rozměr očekává pouze zadanou délku jedné ze stran. Uživatel musí definovat, zda zadává délku kratší, čí delší strany.

Další atributy tisknutelného formátu jsou:

- název,

- papíry.

UC 19 Úprava tisknutelného formátu

Uživatel s rolí *tiskárna* může upravovat své tisknutelné formáty. Upravovány jsou ty stejné atributy, které jsou popsány v předchozím případu užití *vytvoření tisknutelného formátu.*

Jakmile jsou upraveny údaje tisknutelného formátu, který je sdílen s alespoň jedním *fotografem*, je takovému uživateli zasláno oznámení upozorňující na provedené změny.

UC 20 Odstranění tisknutelného formátu

Uživatel s rolí *tiskárna* může odstranit jím vytvořené tisknutelné formáty. Odstraněním tisknutelného formátu dojde logicky k jeho odstranění i z *událostí*, potažmo *fotografií.*

Po odstranění tisknutelného formátu dojde k zaslání upozornění těm *fotografům*, kteří daný formát využívají.

Uživatel s rolí *fotograf* může odstranit „vypůjčený“ tisknutelný formát. Tím ztratí právo daným formátem nadále disponovat a v případě zájmu o jeho opětovné využívání je nutné podat žádost o využití tisknutelného formátu znovu.

UC 21 Zveřejnění tisknutelných formátů

Uživatel s rolí *tiskárna*, jenž disponuje svými tisknutelnými formáty, může tyto formáty nabídnout k „pronájmu“ jiným uživatelům s rolí *fotograf.* Uživatel se může podělit buď se všemi formáty, nebo žádným.

UC 22 Žádost o využití tisknutelných formátů

Uživatel s rolí *fotograf* může zažádat o využití konkrétního tisknutelného formátu jeho „majitele“. Majiteli poté přijde upozornění ohledně nové žádosti o využití konkrétního formátu daným *fotografem.*

UC 23 Vyřízení žádosti o využití tisknutelných formátů

Uživatel s rolí *tiskárna* vyřizuje žádosti *fotografů* o využití formátu. Žádost je vytvořena ve stavu *čeká.* Danou žádost může uživatel s rolí *tiskárna* označit jako:

- schváleno,
- zamítnuto.

Stav žádosti o využití formátu může majitel daného formátu kdykoliv změnit, a tím žádost dodatečně schválit/zamítnout.

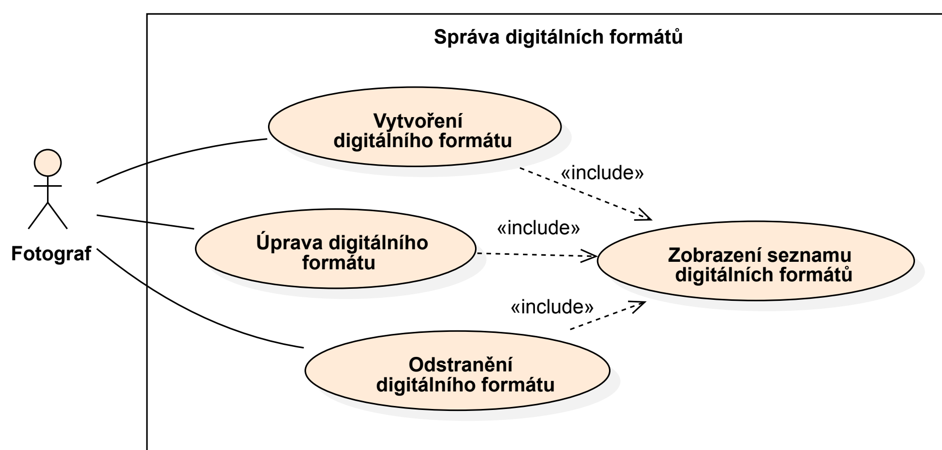
UC 24 Využití tisknutelného formátu

Uživatel s rolí *fotograf*, jemuž bylo povoleno využívat některé z veřejných tisknutelných formátů, může v těchto formátech nabízet své fotografie.

Uživatel, jenž disponuje rolemi *fotograf* a *tiskárna* zároveň, může samozřejmě své tisknutelné formáty využívat přímo.

2.1.6 Správa digitálních formátů

Grafické znázornění případů užití je k vidění na obrázku 2.6. *Digitální* formáty může, narozdíl od *tisknutelných* formátů, vytvářet pouze uživatel s rolí *fotograf*. Dalším rozdílem je, že digitální formáty nejsou sdílitelné napříč uživateli. Slouží k definici, v jakém rozlišení se dá stáhnout digitální kopie fotografie.



Obrázek 2.6: Diagram případů užití správy digitálních formátů

UC 25 Zobrazení seznamu digitálních formátů

Fotograf může zobrazit seznam digitálních formátů, které sám vytvořil. Digitální formáty jsou zobrazeny ve formě karet, které obsahují veškeré potřebné informace o něm.

UC 26 Vytvoření digitálního formátu

Fotograf může vytvořit vlastní digitální formát. Ten vyžaduje vyplnění následujících atributů:

- název,
- rozlišení udávané v jednotce MPx,

- příznak, zda se má fotografie s daným formátem stáhnout s vodoznakem, či bez něj,
- cena.

UC 27 Úprava digitálního formátu

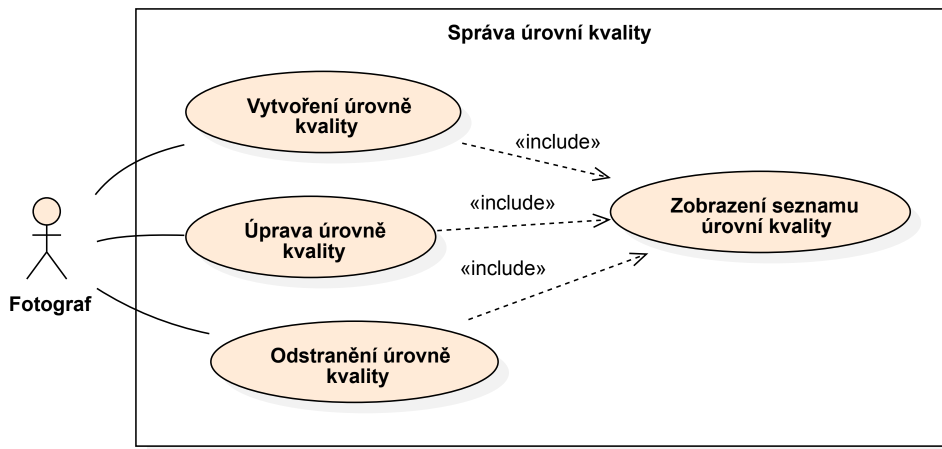
Fotograf může upravit libovolný z jím vytvořených formátů. Úprava spočívá ve změně stejných atributů, které jsou vyžadovány při jeho vytvoření, viz. případ užití *vytvoření digitálního formátu*.

UC 28 Odstranění digitálního formátu

Uživatel s rolí *fotograf* může odstranit libovolný z jím vytvořených digitálních formátů. Tyto digitální formáty logicky nebude možné nadále nabízet při objednávání fotografií.

2.1.7 Správa úrovní kvality

Diagram případů užití správy úrovní kvality je vyobrazen na obrázku 2.7. Právo spravovat úrovně kvality má výhradně uživatel s rolí *fotograf*.



Obrázek 2.7: Diagram případů užití správy úrovní kvality

UC 29 Zobrazení seznamu úrovní kvality

Fotograf může zobrazit kompletní seznam svých úrovní kvality. Ty jsou zobrazeny ve formě karet se všemi potřebnými informacemi.

UC 30 Vytvoření úrovne kvality

Fotograf může vytvořit novou úroveň kvality, jež sestává z následujících atributů:

- název,
- úroveň kvality.

Hodnota atributu *úroveň kvality* je desetinné číslo, které musí nabývat hodnoty alespoň 1.

UC 31 Úprava úrovně kvality

Fotograf má právo upravit všechny jím vytvořené úrovně kvality. Úprava spočívá ve změně stejných atributů, které jsou potřebné v případu užití *vytvoření úrovně kvality.*

UC 32 Odstranění úrovně kvality

Fotograf může odstranit jím vytvořené úrovně kvality. Tyto už nebude možné využít při prodeji tištěných fotografií.

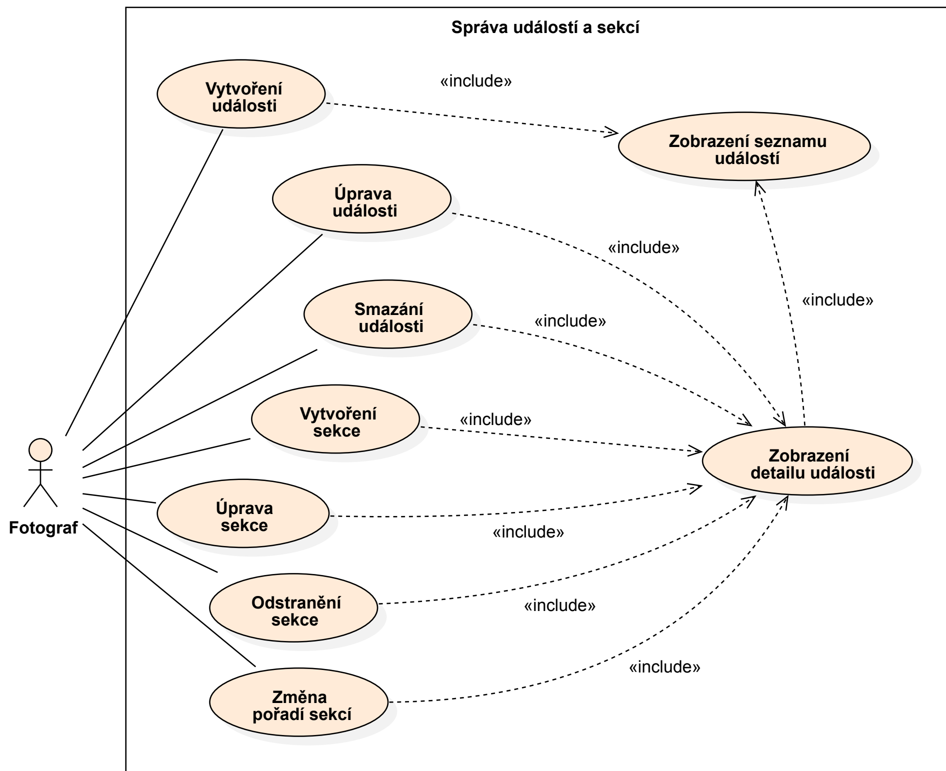
2.1.8 Správa událostí a sekcí

Diagram případů užití pro správu událostí a sekcí je k vidění na obrázku 2.8. Správu událostí a sekcí má ve své gesci uživatel s rolí *fotograf.*

UC 33 Zobrazení seznamu událostí

Fotograf má právo zobrazit seznam událostí, které sám vytvořil. Tyto události jsou zobrazeny ve formě karet, které obsahují všechny potřebné informace o události. Seznam událostí je fixně seřazen podle data události sestupně. Události je možno filtrovat dle některých jejich parametrů:

- název,
- datum,
- stav (aktivní/uzavřená),
- viditelnost (veřejná/chráněná/soukromá).



Obrázek 2.8: Diagram případů užití správy událostí a sekcí

UC 34 Vytvoření události

Fotograf může vytvořit novou událost. Ta pro své vytvoření vyžaduje některé mandatorní parametry:

- název,
- datum,
- popis,
- stav (aktivní/uzavřená),
- viditelnost (veřejná/chráněná/soukromá),
- formáty,
- úroveň kvality,
- uživatelé.

Hodnoty atributu *formáty* jsou takové formáty, které si uživatel sám definoval a nebo si je „vypůjčil“ od jiných uživatelů.

Hodnotami atributu *uživatelé* jsou e-mailové adresy. Jedině uživatelé s danou e-mailovou adresou mají přístup k dané události. Tento atribut bude možné vyplnit pouze v případě, že je *viditelnost* události nastavena jako *soukromá.*

UC 35 Úprava události

Fotograf, jenž vytvořil nějakou událost, má právo tuto událost upravit. Úprava spočívá ve změně stejných atributů, které jsou popsány v předchozím případu užití *vytvoření události.*

UC 36 Smazání události

Fotograf vlastnící nějakou událost má možnost tuto událost smazat. Smazáním události dojde k odstranění i všech fotografií, které jsou v rámci této události, respektive v rámci všech jejich sekcí, nahrány.

UC 37 Zobrazení detailu události

Fotograf si může zobrazit detail události, jiš předtím vytvořil. V rámci detailu je kromě všech podstatných atributů události k vidění i *seznam sekcí*, které jsou vytvořeny v rámci události.

UC 38 Vytvoření sekce

Uživatel s rolí *fotograf* může v rámci události vytvořit libovolný počet *sekcí.* Tato sekce sestává pouze z atributu *název.*

UC 39 Úprava sekce

Fotograf může vytvořenou sekci upravit. Její úprava spočívá pouze ve změně jejího *názvu.*

UC 40 Odstranění sekce

Fotograf může sekci odstranit. Tím dojde zároveň k odstranění všech fotografií, které byly v rámci sekce nahrány.

UC 41 Změna pořadí sekce

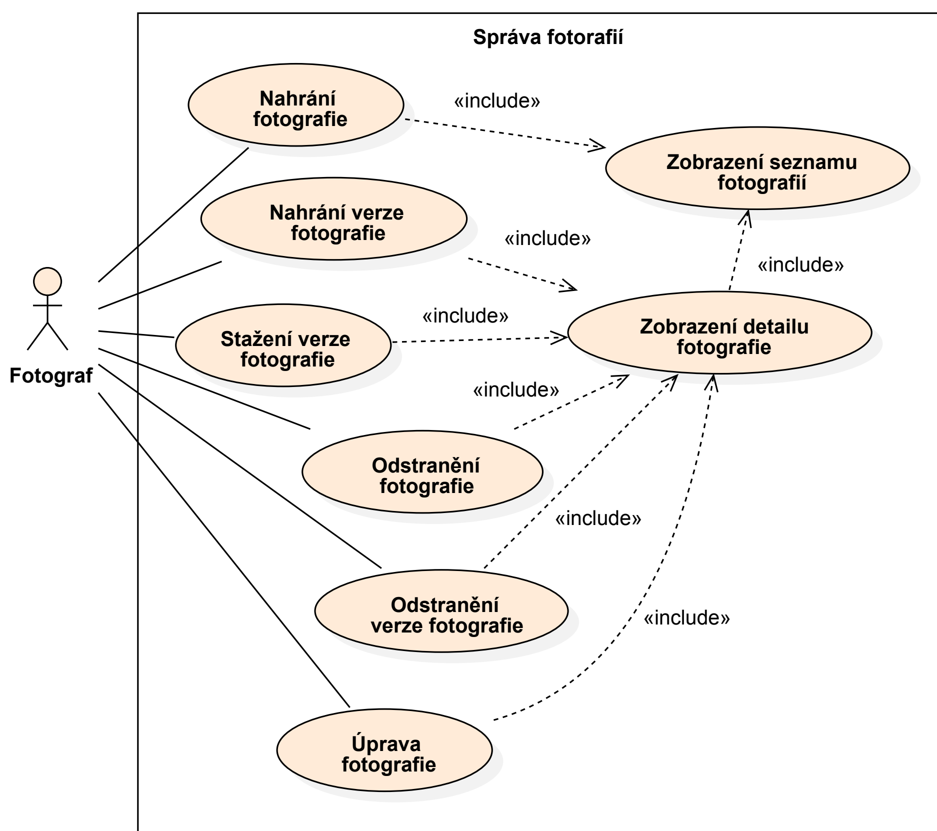
Fotograf může měnit pořadí jednotlivých sekcí. Přeskupení jednotlivých sekcí se provádí systémem *drag and drop.*

2.1.9 Správa fotografií

Diagram případů užití správy fotografií je k vidění na obrázku 2.9. Fotografie může uživatel spravovat pouze v rámci dané *sekce*, z toho vyplývá, že právo na správu fotografií má pouze uživatel s rolí *fotograf*. Zde není od věci připomenout rozdíl mezi *fotografií* a *verzí fotografie*:

Fotografie může mít jednu nebo více *verzí*, které se od sebe příliš neliší a zpravidla zachycují jeden unikátní okamžik.

Verze fotografie je představována konkrétním souborem, jenž se nějakým způsobem odlišuje od jiných *verzí*, např. jinou úpravou barev, přidanými retušemi, jiným ořezem, atd. Všechny verze jedné fotografie však vznikly zpravidla „jedním stiskem spouště fotoaparátu".



Obrázek 2.9: Diagram případů užití správy fotografií

UC 42 Zobrazení seznamu fotografií

Fotograf má možnost si v rámci dané *sekce* zobrazit seznam fotografií. Fotografie lze zobrazit expanzí karty dané sekce. Velikost jednotlivých fotografií lze nastavit uživatelsky na $^1/_{12}$, $^1/_6$, $^1/_4$, $^1/_3$ nebo $^1/_2$ šířky stránky. V rámci přehledu fotografií lze uživatelsky nastavit, zda se má fotografie zobrazit ve svém originálním poměru stran či zda se má přizpůsobit vyhrazenému místu na stránce.

UC 43 Nahrání fotografie

Fotograf může v rámci konkrétní *sekce* nahrát libovolný počet fotografií najednou. Po nahrání fotografie je dané fotografii vytvořena zároveň první *verze fotografie*.

UC 44 Zobrazení detailu fotografie

Fotograf může zobrazit detail konkrétní fotografie. Detail zobrazí fotografii ve své *poslední* verzi. Lze však přepnout na libovolnou verzi dané fotografie. V rámci detailu fotografie jsou zobrazeny základní EXIF data konkrétní *verze fotografie*:

- rozměry,
- velikost,
- clonové číslo objektivu,
- expoziční čas fotoaparátu,
- ohnisková vzdálenost objektivu,
- hodnota citlivosti ISO,
- výrobce a model fotoaparátu,
- výrobce a model objektivu.

UC 45 Nahrání verze fotografie

Uživatel s rolí *fotograf* má možnost v rámci konkrétní *fotografie* nahrát její novou *verzi*. S nahráním nové verze fotografie je možné k verzi připojit *komentář*, v čem je daná verze jiná, než předchozí.

UC 46 Úprava fotografie

Fotograf může konkrítní fotografii nastavit, zda se bude klientovi nabízet pouze *poslední verze* fotografie, či zda si bude moci zvolit z *libovolné verze* dané fotografie. Dalšími atributy, které je možné u fotografie upravit, jsou *formáty* a *úroveň kvality*. Těmito atributy lze „přepsat“ nabízené formáty, respektive úroveň kvality, které by jinak fotografie „zdědila“ z formátů, respektive úrovně kvality, které jsou definovány v rámci *události*.

UC 47 Stažení verze fotografie

Fotograf může stáhnout aktuální *verzi fotografie* v plném rozlišení.

UC 48 Odstranění fotografie

Fotografii může *fotograf* smazat. Smazáním fotografie se smažou i všechny *verze fotografie* vytvořené v rámci dané fotografie.

UC 49 Odstranění verze fotografie

Fotograf může mazat i jednotlivé verze fotografie, přičemž je nutné, aby každá *fotografie* měla za všech okolností alespoň jednu *verzi*.

2.1.10 Správa objednávek

Diagram případů užití pro správu objednávek je vynesen na obrázku 2.10. Správy objednávek se zúčastňuje primárně uživatel s rolí *fotograf*, který obstarává objednávky *svých* fotografií. Uživatel s rolí *tiskárna* má přístup k objednávkám, v nichž figurují fotografie objednané ve *formátech*, jichž je majitelem.

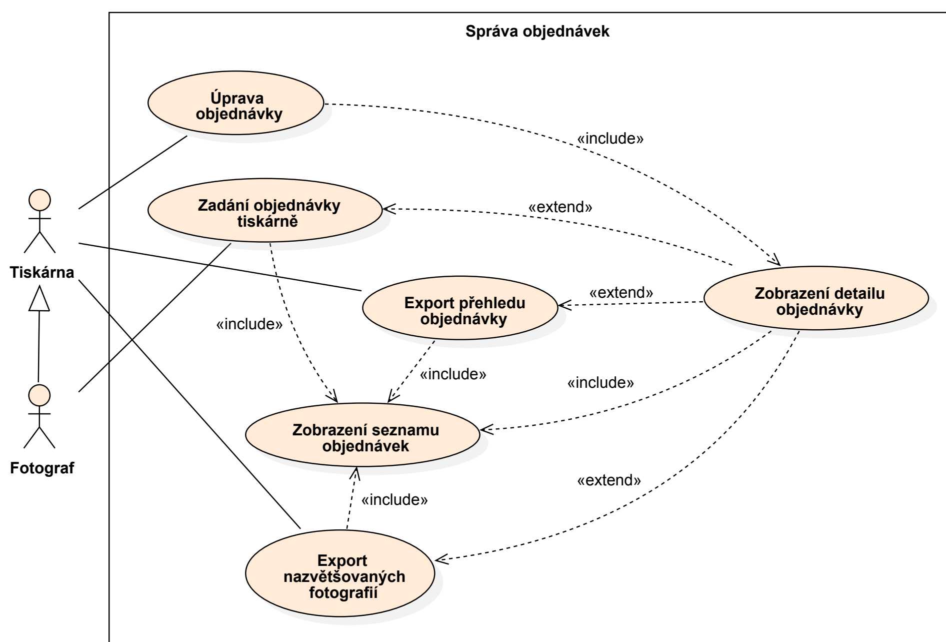
UC 50 Zobrazení seznamu objednávek

Uživatel s rolí *fotograf* i *tiskárna* mají přístup k objednávkám. *Fotograf* může zobrazit všechny objednávky svých fotografií. Oproti tomu uživatel s rolí *tiskárna* má právo zobrazit takové objednávky, v nichž se nacházejí fotografie objednané ve *formátu*, které vlastní daný uživatel. V těchto objednávkách má pak přístup pouze k takovým fotografiím.

Přehled objednávek je vynesen v datové tabulce. Záznamy v tabulce se dají řadit dle libovolného atributu a podle libovolného atributu je možné tabulku i filtrovat. S objednávkami lze provádět i vybrané hromadné akce, jako je např. změna jejich *stavu*, jejich *export* či hromadné *zadání objednávky tiskárně*.

UC 51 Úprava objednávky

Fotograf má právo upravit veškeré údaje své objednávky. Může tedy měnit *počet* objednaných fotografií, objednané *formáty*, *stav platby* objednávky, a v ne-



Obrázek 2.10: Diagram případů užití správy objednávek

poslední řadě i *stav objednaných fotografií* a celkový *stav objednávky*. Uživatel s rolí *tiskárna* může objednávku editovat jen omezeně, manipulovat může se *stavem* objednaných fotografií a s celkovým *stavem objednávky*.

Stav objednané fotografie může nabývat jednoho z následujících stavů:

- nová,
- v tisku,
- vytištěná.

Na základě stavu jednotlivých objednaných fotografií se mění i stav celkové objednávky, jenž může nabývat následujících hodnot:

Nová, pokud jsou *všechny* fotografie v rámci objednávky ve stavu *nová*. Tento stav je nastaven při vytvoření objednávky.

Připravuje se, pokud jsou *všechny* fotografie ve stavu *v tisku* nebo *vytištěná*. Tento stav je nastaven systémem *automaticky*.

Zkompletovaná, pokud jsou *všechny* fotografie již vytištěné a uživatel tyto fotografie shromáždil na jednom místě (zabalil do jedné obálky, ...). Tento stav nastavuje uživatel *ručně*.

Předaná, pokud jsou *všechny* fotografie předány uživateli. Tento stav nastavuje uživatel *ručně*.

UC 52 Zadání objednávky tiskárně

V případě, že se v objednávce daného *fotografa* nacházejí takové fotografie, které jsou objednány ve formátu, jehož majitelem je *jiný* uživatel s rolí *tiskárna*, má možnost *fotograf* tyto fotografie zadat dané *tiskárně*. Tím dojde k vytvoření nové objednávky, ta však bude sdílet objednané fotografie s objednávkou současnou. Adresátem nově vzniklé objednávky bude zadávající uživatel s rolí *fotograf*.

UC 53 Zobrazení detailu objednávky

Jak *fotograf*, tak i uživatel s rolí *tiskárna* mají možnost zobrazit detail objednávky. Zatímco *fotograf* vidí *všechny* objednané fotografie, uživatel s rolí *tiskárna* má v detailu objednávky přístup pouze k fotografiím, které byly objednány v jeho formátu. V detailu objednávky vidí uživatel všechny podstatné informace o ní. Mimo jiné údaje o zákazníkovi:

- jméno a příjmení,
- e-mail,
- telefonní číslo,
- adresa.

Podstatné jsou však především informace o objednaných fotografiích a formátech, v nichž jsou jednotlivé fotografie objednané. Objednané fotografie jsou shlukovány dle *událostí*, v nichž byly dané fotografie publikovány.

- náhled fotografie,
- název fotografie,
- číslo verze,
- název sekce události,
- objednaný tisknutelný/digitální formát.

Pro každý objednaný tisknutelný/digitální formát dané fotografie se zobrazuje:

- počet objednaných kusů,
- cena za jednu fotografii v daném formátu,

- celková cena objednaných fotografií v daném formátu.

V rámci detailu objednávky je zahrnuta také kalkulace výsledné ceny, včetně rozepsaných cen za jednotlivé formáty. Následuje přehled celkového *počtu* fotografií pro dané formáty.[8]

UC 54 Export přehledu objednávky

Jak *fotograf*, tak uživatel s rolí *tiskárna* mají možnost exportovat základní data o objednávce, případně více objednávkách najednout. Tyto informace jsou exportovatelné ve formátech CSV či XLSX. Exportovaná data na svých řádcích obsahují informace o objednané fotografii v daném formátu. Dalšími údaji jsou kompletní informace o zákazníkovi. Tento formát exportovaných dat má sice za následek velikou redundanci dat, nicméně je velmi vhodný pro podrobné uživatelské filtrování nad exportovaným souborem.

UC 55 Export nazvětšovaných fotografií

Uživatel s rolí *fotograf* či *tiskárna* si mohou nechat systémem vygenerovat nazvětšované fotografie dle objednaných formátů. Tyto fotografie budou po vygenerování k dispozici ke stažení v podobě ZIP archivu. Adresářová struktura, v rámci které jsou fotografie organizovány, je uživatelsky definovaná. Jednotlivé úrovně adresářové struktury uživatel volí z následující nabídky:

- zákazník,
- událost,
- sekce,
- formát,
- papír,
- počet objednaných kusů.

Příklad, jak by mohla vypadat struktura exportovaného ZIP archivu s nazvětšovanými fotografiemi přibližuje obrázek 2.11. V tomto případě uživatel při exportu zvolil[9] následující parametry pro seskupení fotografií:

- událost,
- formát,
- počet.

Uživatel má při exportu možnost označit hromadně exportované fotografie jako *v tisku*.

[8] Užitečné při kompletaci objednávky.

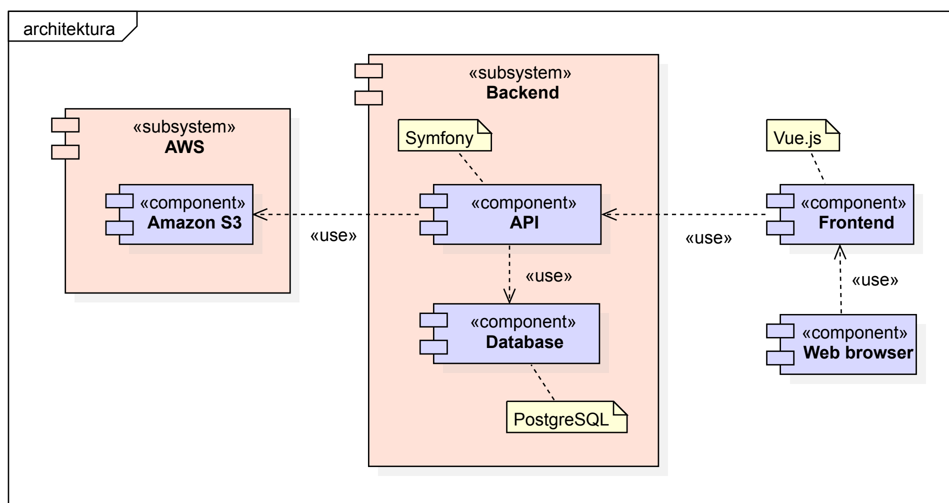
[9] V tomto pořadí.

```
/ ............................................ kořenový adresář archivu
└── maturitni-ples/ ................................... adresář události
    └── 15-10_maly-format/ ............................. adresář formátu
        ├── 5/ ........................... adresář počtu objednaných kusů
        │   ├── IMG_0001.jpg ........... soubor s fotografií v dané velikosti
        │   └── IMG_0002.jpg
        └── 10/
            ├── IMG_0001.jpg
            ├── IMG_0002.jpg
            └── IMG_0003.jpg
```

Obrázek 2.11: Ukázka adresářové struktury ZIP archivu nazvětšovaných fotografií

2.2 Architektura služby

Nyní, když jsme již poměrně do detailu obeznámeni se všemi *funkčními* i *nefunkčními* požadavky, je ten nejvyšší čas zamyslet se nad architekturou naší služby. Diagram komponent zachycující základní rysy architektury je vynesen na obrázku 2.12.



Obrázek 2.12: Architektura služby

Jeden z podstatných nefunkčních požadavků nás zavazuje přemýšlet nad architekturou naší služby jako nad *webovou aplikací*. Toto rozhodnutí v sobě skrývá nemalé výhody:

- aplikace neklade na počítač klienta vysoké nároky, podstatné operace

jsou prováděny na webovém serveru,

- aplikace se tímto stává multiplatformní, neboť *každý* běžně používaný moderní operační systém disponuje webovým prohlížečem,
- distribuce nových verzí aplikace spočívá v jejich nasazení na webový server, klientovi se nová verze stáhne automaticky vždy, když k aplikaci přistoupí přes webový prohlížeč.

Komponenta zapouzdřující *obchodní logiku* se nachází na webovém serveru. V dnešní době je nemyslitelné začít vyvíjet takou komponentu „na zelené louce“. Naštěstí je možné pro tyto účely využít celou řadu *frameworků*, které nám s našimi požadavky na systém dokáží ve větší či menší míře pomoci. Rozhodnutí ohledně volby konkrétního frameworku padlo na PHP framework *Symfony*. Jeho použití v projektu se pak věnuje celá podkapitola 5.1.

Aplikace, která se v dnešní době nedokáže integrovat s jinými službami, jako by nebyla. Základním předpokladem integrace je komunikace s aplikací pomocí API[10]. V našem případě jsem se rozhodl o implementaci architektury rozhraní, jež se nazývá REST[11]. O návrhu REST API naší služby však více pojednává samostatná podkapitola 5.1.2. Rozhodnutí poskytovat REST API naší služby libovolnému návštěvníkovi s sebou nese výzvy, které se týkají jeho *zabezpečení*. I jemu je však věnována samostatná podkapitola 5.1.3.

Díky rozhdonutí poskytovat REST API se nám nepřímo nabídla možnost využít toto rozhraní ke komunikaci té části aplikace, jež obsahuje *obchodní logiku*, s prezenční vrstvou, skrze kterou aplikaci ovládá cílový uživatel. Pro tuto část aplikace opět využijeme nějaký framework, v našem případě se jedná o moderní javaskriptový framework *Vue.js*.

Samozřejmostí je popřemýšlet o komponentě, která by měla za úkol ukládání dat aplikace. V našem případě jsme sáhli po osvědčeném řešení v podobě databáze *PostgreSQL*.

Vzhledem k požadavku na *horizontální škálovatelnost úložiště*, který vyvstal po potřebě ukládat velké množství dat (v našem případě fotografií), jsem se rozhodl „přenechat“ odpovědnost za ukládání souborů externí službě. V našem případě volba padla na službu Amazon S3[12], která je součastí AWS[13]. Konkrétní implementace propojení služby Amazon S3 s naší službou však dovoluje využití široké palety vzdálených úložišť. O tom se ale podrobněji rozepisuji v rámci podkapitoly 5.1.4.

---

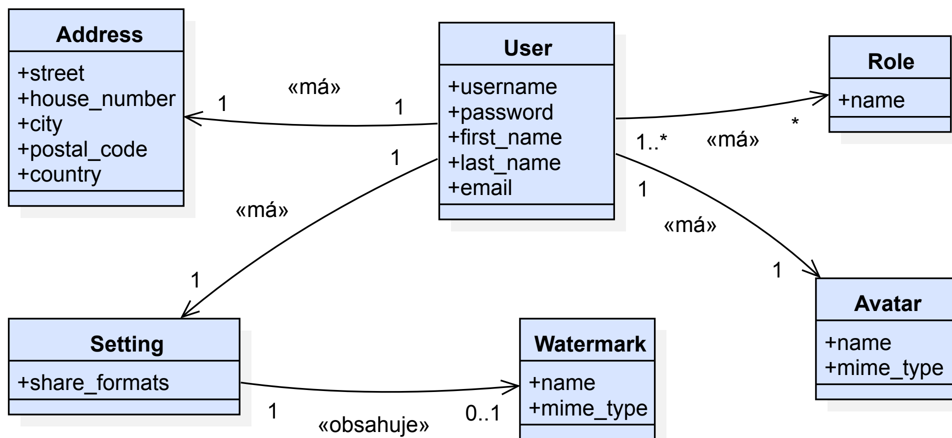
[10]Application Programming Interface
[11]Representational State Transfer
[12]Simple Storage Service
[13]Amazon Web Services

2.3 Doménový model

Další fází návrhu našeho systému je *doménový model*. Ten je zpravidla vytvářen právě po ujasnění veškerých požadavků a z velké části vychází z již dříve probíraných *případů užití*. V rámci doménového modelu se pokusím zachytit klíčové entity, jejich důležité atributy a vzájemné vyzby, jež budou tvořit páteř obchodní logiky naší aplikace. Tyto entity jsou v případě *doménového modelu* modelovány jako platformě nezávislé a bez ohledu na jejich budoucí datové typy. V následujících doménových modelech také nebudu vyobrazovat různé technické atributy entit (primární klíče, cizí klíče, ...).

2.3.1 Uživatelé

Středobodem naší aplikace budou svým způsobem uživatelé. Jelikož aplikaci navrhujeme jako službu, téměř všechna vytvořená data (ať už v rámci databáze či na souborovém systému) se budou vázat na konkrétního uživatele. Doménový model zachycující třídy spojené s uživatelem je vynesen na obrázku 2.13.



Obrázek 2.13: Doménový model uživatele

`User`

Entita `User` reprezentuje konkrétního uživatele. Její struktura je z části dána konvencemi, které vycházejí z implementace uživatelů v rámci frameworku *Symfony*. Entita obsahuje následující atributy:

- `username` – uživatelské jméno, jež identifikuje uživatele při přihlášení,

- `password` – heslo uživatele ve formě jeho otisku[14],
- `first_name` – křestní jméno uživatele,
- `last_name` – příjmení uživatele,
- `email` – e-mailová adresa uživatele.

`Role`
Entita `Role` uchovává uživatelské role, které se v systému nacházejí. O role se poté dělí jednotliví uživatelé, kteří jich mohou mít i více, nejméně však jednu. Entita obsahuje pouze jeden atribut:

- `name` – název role.

`Avatar`
Entita `Avatar` reprezentuje profilový obrázek uživatele. Obsahuje následující atributy:

- `name` – název souboru obrázku,
- `mime_type` – identifikátor typu internetového média MIME[15].

`Address`
Entita `Address` reprezentuje adresu, které je zapotřebí v případě doručení zásilky konkrétnímu klientovi. Entita obsahuje následující atributy:

- `street` – ulice,
- `house_number` – číslo domu,
- `city` – město,
- `postal_code` – poštovní směrovací číslo,
- `country` – země.

`Settings`
Entita `Settings` sdružuje uživatelská nastavení. V první verzi návrhu obsahuje pouze následující atribut:

- `share_formats` – přiznak, zda uživatel dovoluje sdílení svých *formátů* s jinými uživateli.

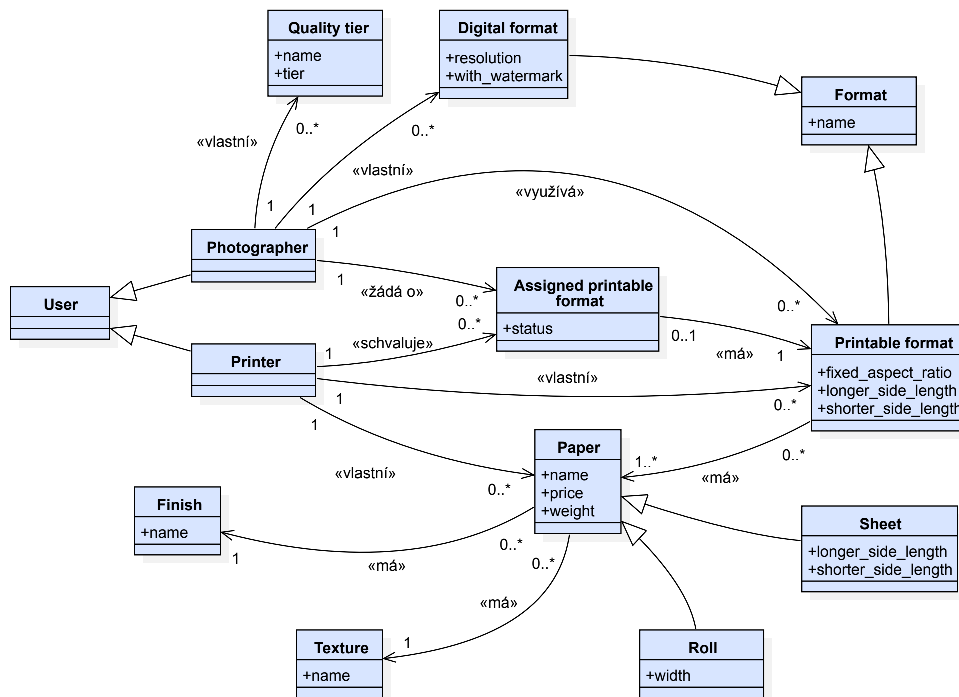
`Watermark`
Entita `Watermark` reprezentuje vodoznak, který může být posléze automaticky vkládán do náhledů fotografií daného uživatele. Entita obsahuje následující atributy:

- `name` – název souboru vodoznaku,
- `mime_type` – identifikátor typu internetového média MIME.

[14] Tzv. *hash*, tedy výstup *kryptografické hashovací funkce*
[15] Multipurpose Internet Mail Extensions

2.3.2 Papíry, úrovně kvality, formáty

Mezi základní stavební kameny naší služby bezesporu patří správa digitálních i tisknutelných formátů, papírů a v neposlední řadě úrovní kvality. Doménový model zachycující tyto entity a vztahy mezi nimi je vyobrazen na obrázku 2.14.



Obrázek 2.14: Doménový model papírů, úrovní kvality a formátů

`Photographer, Printer`
Tyto entity představují specializaci uživatele s rolí *fotograf*, respektive *tiskárna*.

`Quality tier`
Entita `Quality tier` reprezentuje úroveň kvality, jež svým způsobem spoluvytváří celkovou cenu za objednanou tištěnou fotografii. Entita obsahuje následující atributy:

- `name` – název úrovně kvality,
- `tier` – koeficient úrovně kvality.

`Paper`
Entita `Paper` reprezentuje papíry, které má uživatel k dispozici. Papír má následující základní atributy:

- `name` – název papíru,
- `price` – cena papíru udávaná v jednotkách Kč/m^2,
- `weight` – hmotnost papíru udávaná v jednotkách g/m^2.

Dále se papíru přiřazují atributy, jež definují jeho specifické vlastnosti. Hodnoty těchto atributů jsou sdružovány v samostatných entitách:

`Finish`
Entita `Finish` představuje povrchovou úpravu papíru. Obsahuje následující atribut:

- `name` – název povrchové úpravy.

`Texture`
Entita `Texture` reprezentuje druh struktury papíru. Obsahuje následující atribut:

- `name` – název papírové textury.

Papír ze své podstaty dělíme na dva druhy, oba jsou zde reprezentovány samostatnými entitami:

`Roll`
Entita `Roll` reprezentuje typ papíru, který se nazývá *role*. U této entity evidujeme následující atribut:

- `width` – šířka role papíru.

`Sheet`
Entita `Sheet` představuje papír typu *arch*. Tato entita obsahuje následující atributy:

- `longer_side_length` – délka delší strany papíru,
- `shorter_side_length` – délka kratší strany papíru.

`Format`
Entita `Format` představuje obecný formát, jenž definuje, v jakých variantách může zákazník objednat fotografii, k níž je daný formát přiřazen. Obsahuje pouze následující společný atribut:

- `name` – název formátu.

Obecný formát se dělí na dva druhy, z nichž oba jsou zde reprezentovány samostatnými entitami:

`Printable format`
Entita `Printable format` představuje formát, v jakém lze objednat fotografii k *tisku*. Tento formát využívá entity `Paper`, aby uživatel věděl, na jaký *papír* je potřeba fotografii vytisknout, kromě toho entita `Printable format` obsahuje navíc následující atributy:

- `fixed_aspect_ratio` – příznak, zda má být daný formát definován pomocí délek kratší i delší strany, či zda bude zadána pouze strana jedna a délka druhé strany se odvodí z poměru stran fotografie,
- `longer_side_length` – délka delší strany,
- `shorter_side_length` – délka kratší strany.

`Digital format`

Entita `Digital format` reprezentuje digitální formát, který dává zákazníkovi možnost objednat zvolenou fotografii, kterou si může posléze stáhnout do počítače. Tato entita obsahuje následující atributy:

- `resolution` – rozlišení digitálního formátu udávané v jednotkách Mpx,
- `price` – cena, za níž je možné stáhnout digitální verzi fotografie,
- `with_watermark` – příznak, zda se má fotografie vygenerovat s vodoznakem, či bez něj.

`Assigned printable format`

Entita `Assigned printable format` vytváří vazbu mezi *formátem* a *uživatelem*, který má daný formát *zapůjčený* a využívá jej při nabídce svých fotografií. Tato entita obsahuje následující atribut:

- `status` – stav přiřazeného formátu, jenž definuje, zda může žadatel daný formát použít či nikoliv.

2.3.3 Události, sekce, fotografie

Po návrhu papírů, formátů a dalších nepostradatelných entity souvisejících s finální podobou objednané fotografie přichází na řadu návrh samotných *fotografií*. Ty jsou sdružovány v *událostech*, respektive v jednotlivých *sekcích* události. Doménový model zachycující vztahy mezi událostmi, sekcemi a fotografiemi je k vidění na obrázku 2.15.

`Photographer`

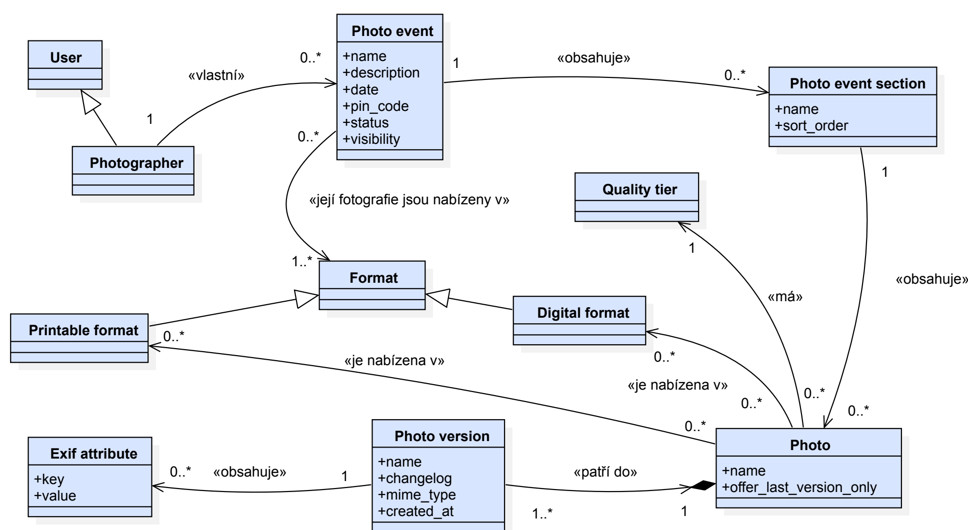
Entita `Photographer` představuje specializaci uživatele s rolí *fotograf*.

`Printable format, Digital format, Quality tier`

Tyto entity představují *tisknutelné*, respektive *digitální* formáty a *úrovně kvality*, které byly vymodelovány dříve v rámci podkapitoly 2.3.2.

`Photo event`

Tato entita reprezentuje událost, v rámci níž jsou sdružovány jednotlivé sekce události. Entita obsahuje následující atributy:



Obrázek 2.15: Doménový model událostí, sekcí a fotografií

- `name` – název události,
- `description` – popis událsti,
- `date` – datum konání události,
- `pin_code` – PIN kód pro vstup do události v případě, že je *viditelnost* události nastavena jako *chráněná*,
- `status` – stav události (aktivní/uzavřená),
- `visibility` – viditelnost události (veřejná/chráněná/soukromá).

`Photo event section`

Entita reprezentuje sekci události, v rámci které jsou dále shromažďovány jednotlivé fotografie. Entita obsahuje následující atributy:

- `name` – název sekce,
- `sort_order` – pořadí sekce v rámci události.

`Photo`

Entita reprezentuje fotografii, která se ve své podstatě skládá z jednotlivých verzí dané fotografie. Entita obsahuje následující atributy:

- `name` – název fotografie,
- `offer_last_version_only` – příznak, zda bude koncovému zákazníkovi umožněno objednat pouze poslední verzi fotografie, či bude možné objednat verzi libovolnou.

Photo version

Entita reprezentuje konkrétní verzi fotografie. Obsahuje následující atributy:

- `name` – název verze fotografie,
- `changelog` – seznam změn, kterými se liší daná verze fotografie od verze předchozí,
- `mime_type` – identifikátor typu internetového média MIME,
- `created_at` – časová značka vytvoření verze.

Exif attribute

Entita shromažďuje podporované EXIF atributy, které je možno získat z metadat nahrané fotografie. Entita obsahuje následující atributy

- `key` – klíč EXIF atributu,
- `value` – hodnota EXIF atributu.

2.3.4 Objednávky

Poslední důležitou oblastí, jíž je třeba věnovat pozornost v návrhu doménového modelu jsou *objednávky*. Doménový model zachycující vztahy mezi entitami, které se objednávek účastní, je k vidění na obrázku 2.16.

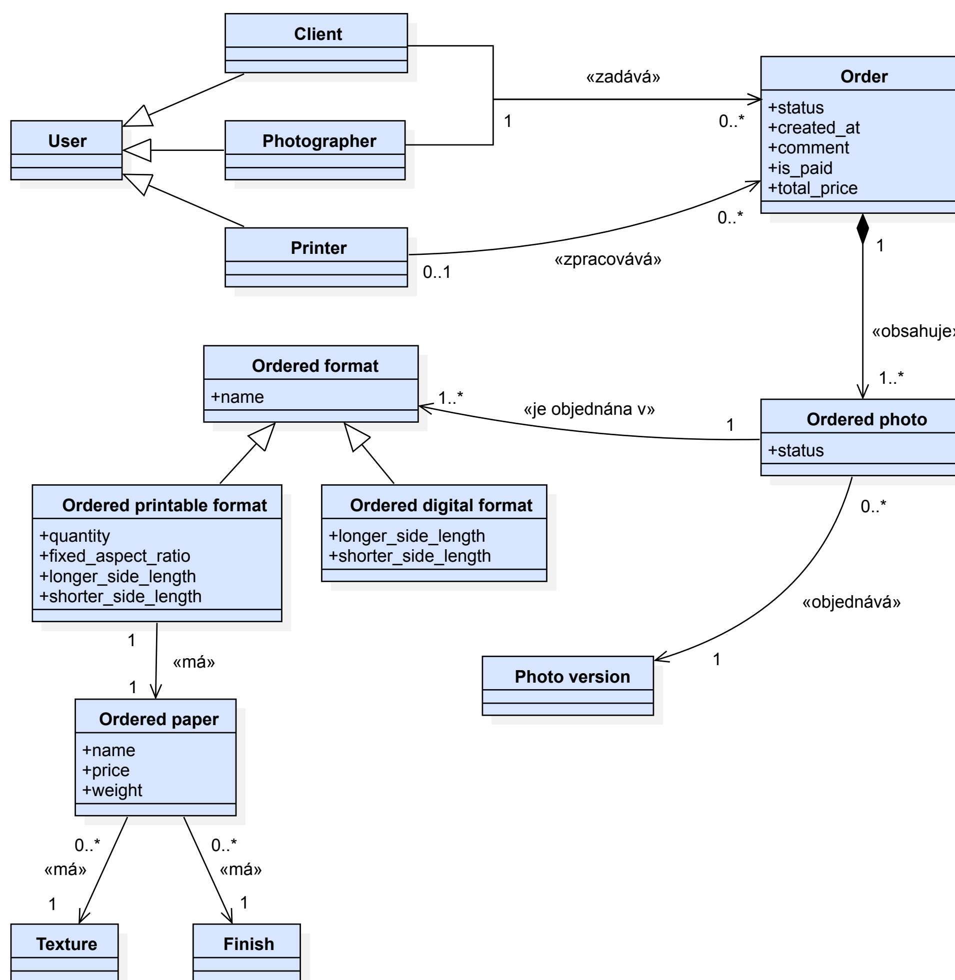
Zde není od věci upozornit, že se v doménovém modelu nacházejí entity, které mají téměř stejnou strukturu, jako entity již dříve modelované. V našem případě se jedná o entity:

- `Ordered format` – kopie entity `Format`,
- `Ordered printable format` – kopie entity `Printable format`,
- `Ordered digital format` – kopie entity `Digital format`,
- `Ordered paper` – kopie entity `Paper`.

Proč vytvářet nové, téměř identické entity, když podobné již v našem doménovém modelu máme? Může se snadno stát, že fotograf nabídne své fotografie v určitých formátech/papírech, klient si tyto fotografie v nějakém z vystavených formátů/papírů objedná a fotograf posléze některé z údajů nabízených formátů/papírů pozmění. Pokud se při vytvoření objednávky pouze *referencovala* entita `Format`, respektive `Paper`, došlo by po úpravě těchto entit ke změnám i ve všech objednávkách, které tyto formáty/papíry využívají. Z toho důvodu je nutné s každou objednanou fotografií zreplikovat veškerá data nesoucí informace o formátech/papírech.

Client, Photographer, Printer

Entity představují specializaci uživatele s rolemi *klient*, respektive *fotograf*, respektive *tiskárna*.



Obrázek 2.16: Doménový model objednávek

`Photo version`

Entita reprezentuje konkrétní verzi fotografie, její struktura byla více rozebírána v samostatné sekci 2.3.3.

`Texture, Finish`

Entity představují druh struktury papíru, respektive povrchovou úpravu papíru. Struktury obou entit byly podrobněji představeny v rámci sekce 2.3.2.

`Order`

Entita `Order` reprezentuje samotnou objednávku. Ta se ve své podstatě

skládá z jednotlivých objednaných fotografií a obsahuje následující atributy:

- `status` – stav objednávky (nová/připravuje se/zkompletovaná/předaná),
- `created_at` – časová značka vytvoření objednávky,
- `comment` – komentář zadavatele objednávky,
- `is_paid` – příznak, zda je již objednávka zaplacena, či nikoliv,
- `total_price` – celková cena objednávky.

`Ordered photo`

Entita reprezentuje jednu konkrétní objednanou fotografii. Obsahuje následující atribut:

- `status` – stav objednané fotografie (nová/v tisku/vytištěná).

`Ordered format`

Tato entita je spolu se svými podtypy svým způsobem přesnou kopií entity `Order` a jejích podtypů. Z toho důvodu bych si pro více informací o jejich atributech dovolil odkázat na sekci 2.3.2. Jediným atributem navíc je následující atribut v entitě `Ordered printable format`:

- `quantity` – počet kusů objednané fotografie v daném formátu.

`Ordered paper`

Entita `Ordered paper` je opět téměř přesná kopie entity `Paper`. Proto bych pro více informací pouze odkázal na sekci 2.3.2.

Kapitola 3

Vývojová infrastruktura

Mít dobře rozmyšlené a navržené vývojové prostředí je při vývoji software základ. Obzvlášť, když se na procesu vývoje podílí více osob, je žádoucí, ba přímo nutné, aby byly jednotlivé nástroje, které vývojové prostředí definují, popsány na jednom centrálním místě spolu se sepsanými postupy, jak tyto nástroje použít ve společném projektu.

Pod pojmem *vývojové prostředí* si zřejmě většina z nás představí software, který v sobě integruje funkce usnadňující psaní kódu, jeho spuštění, podporu pro kontrolu jeho syntaxe, aj. Pro takové aplikace se vžila zkratka IDE[16]. Pro projekt Photographix jsem zvolil sadu nástrojů od společnosti JetBrains, o nich se více rozepisuji v podkapitole 3.2.

Nejen IDE však definuje vývojové prostředí, dle [2] se vývojové prostředí skládá „z procesů a nástrojů, které se používají k vývoji zdrojového kódu nebo programu“. Zvlášť pokud, jakožto vývojáři, pracujeme najednou na více projektech, kdy každý projekt vyžaduje kupříkladu jinou verzi našeho oblíbeného programovacího jazyka či databáze, chceme tyto projekty a jejich závislosti mít nějakým způsobem oddělené. Možností, jak takovéto „izolace“ dosáhnout, je více. Já jsem v tomto projektu sáhnul po nástroji *Docker*. Více o něm pojednává podkapitola 3.1.

Vývojové prostředí by nebylo úplné bez zmínky o verzovacím systému. Vývoj software si bez něj snad žádný programátor neumí ani představit, jeho přínos pak mnohem více vynikne tehdy, kdy se na jednom projektu podílí více vývojářů najednou. Volba verzovacího systému padla na nástroj Git [3], o kterém blíže pojednává podkapitola 3.3.

Volba verzovacího systému programátory prakticky zavazuje využít i vzdálený repozitář zdrojových kódů, jenž neslouží jen jako centrální úložiště zdrojových kódů, ale i rozšiřuje možnosti verzovacího systému a může přidat i některé užitečné funkce navíc. Využívaným funkcím zvoleného vzdáleného repozitáře zdrojových kódů GitLab se pak blíže věnuje podkapitola 3.4.

[16]Integrated Development Environment

3.1 Docker

V dnešní době, kdy je komplexita psaní aplikací vyšší než kdykoli dříve, vyžaduje vývoj jedné aplikace snoubení mnoha programovacích jazyků, frameworků, databází, či napojení aplikace na různá programová rozhraní. Právě k usnadnění konfigurace prostředí slouží open-source nástroj Docker [4]. Myšlenka Dockeru tkví v izolaci procesů uvnitř tzv. *kontejnerů*. Podle [5] je kontejnerizace virtualizací jádra operačního systému. To znamená, že všechny kontejnery běží v rámci jednoho operačního systému a sdílejí paměť, knihovny a další zdroje. Všechny kontejnery navzájem pak mohou sdílet své jednotlivé vrstvy, což podstatně snižuje i nároky na úložiště.

Kromě Dockeru existuje řada nástrojů[17] pro kontejnerizaci. Docker mezi nimi ale beze sporu dominuje a podle [6] obstarává 83 % kontejnerů.

Abychom lépe porozuměli následujícím řádkům, připomeňme zde některé základní pojmy ze světa Dockeru [7]:

Docker image
: Tvoří základ pro *kontejnery* a pro další *Docker image*, jsou neměnné (*read-only*) a nemají stav. K jeho vytvoření slouží soubor *Dockerfile*.

Dockerfile
: Je to soubor, který popisuje kroky nutné k vytvoření Docker image. Každá instrukce v Dockerfile vytvoří ve výsledném *Docker image* vrstvu. Jednotlivé vrstvy se cachují, díky tomu není nutné po změně v *Dockerfile* sestavovat celý *Docker image* znovu, ale pouze ty jeho vrstvy, které následují až po změněné instrukci v *Dockerfile*.

Docker kontejner
: Kontejner je běžící instance *Docker image*. Po spuštění lze měnit jeho stav, ten se však ztratí po jeho opětovném vypnutí. Pokud chceme některá data zachovat i po vypnutí kontejneru, musíme je vyčlenit skrze tzv. *bind mounts*, případně tzv. *data volume* kontejnery.

Docker Compose
: Nástroj *Docker Compose* usnadňuje definici a spouštění sady jednotlivých *Docker kontejnerů*, o kterých se zde mluví jako o *službách*. Tyto služby mohou být propojeny jednou virtuální sítí, díky které mohou jednotlivé služby mezi sebou komunikovat.

3.1.1 Využití v projektu

Výhody, které nám Docker poskytuje, doceníme v plné míře při snaze o spuštění *backendové* části projektu na počítači vývojáře. Zatímco bez podobných

[17] CoreOS RKT, Mesos Containerizer, Linux Containers LXC, ...

nástrojů by byl vývojář nucen všechny závislosti na svůj hostující počítač instalovat a řešit (ne)kompatibilitu verzí růných závislostí, které potřebuje pro vývoj jiných projektů, Docker mu pomocí několika málo příkazů dokáže připravit instantní vývojové prostředí na míru ušité pro daný projekt.

Jen namátkou si dolovím vypsat několik služeb, které jsou vyžadovány pro korektní běh *backendové* části aplikace a které jsou díky Dockeru programátorovi přístupné bez nutnosti jejich přímých instalací, o konfiguraci všech těchto služeb, které mohou být pro korektní běh aplikace zcela zásadní, nemluvě:

- PHP 7.3,
- Xdebug,
- GD Graphics Library,
- PostgreSQL 13,
- Nginx server.

Dobrý přehled o všech službách, které pro jednotlivé části aplikace využíváme, nám dají konfigurační soubory *docker-compose.yaml* 3.1 a 3.2. Jak je patrné z konfiguračního souboru 3.1, jsou zde definovány čtyři služby, které jsou nutné pro chod *backendové* části aplikace:

`php` Služba `php` definuje kontejner mající základ v Docker image `php:7.3-fpm`. Do tohoto základního Docker image jsou instalovány další závislosti potřebné k programování v PHP (nástroj *Composer* pro práci se závislostmi třetích stran, PHP knihovny *GD Graphics Library* pro práci s grafickými funkcemi, ...). Je zde také provedena konfigurace samotného PHP nahráním vlastního souboru `php.ini` a `xdebug.ini`. Kompletní definice kontejneru `php` nabízí pohled do `Dockerfile` dostupného v příloze C.2.

`nginx` Služba `nginx` slouží k definici kontejneru obsahující webový server. Ten ve zkratce přeposílá obdržené HTTP požadavky na zaváděcí soubor samotné Symfony aplikace, která tyto požadavky nadále zpracovává. Konfugurace webového serveru je dostupná v příloze C.3.

`postgres_dev` Služba `postgres_dev` má již podle názvu co do činění s databází, konkrétně s PostgreSQL. Využívá oficiální Docker image `postgres:alpine` a pomocí proměnných prostředí definuje i přístupové údaje k databázi, která se spuštěním kontejneru iniciuje. Tento kontejner slouží jako databáze pro vývojové účely.

`postgres_test` Služba `postgres_test` je na první pohled obdoba výše zmíněné služby `postgres_dev`, avšak pro testovací účely. O nutnosti mít dvě oddělené databáze pojednává samostatná kapitola o testování *backendové* části aplikace 5.1.6.

```yaml
version: '3'
services:
    php:
        build: php-fpm
        ports:
            - '9000:9000'
        volumes:
            - ..:/var/www/app:cached
            - ./php-fpm/php.ini:/usr/local/etc/php/php.ini
            - ./php-fpm/xdebug.ini:/usr/local/etc/php/conf.d/xdebug.ini
        depends_on:
            - postgres_dev
            - postgres_test
    nginx:
        image: nginx:latest
        ports:
            - '8000:80'
        volumes:
            - ..:/var/www/app:cached
            - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
        depends_on:
            - php
    postgres_dev:
        image: postgres:alpine
        ports:
            - '5432:5432'
        environment:
            POSTGRES_USER: dev
            POSTGRES_PASSWORD: pass
            POSTGRES_DB: devdb
    postgres_test:
        image: postgres:alpine
        ports:
            - '5433:5432'
        environment:
            POSTGRES_USER: test
            POSTGRES_PASSWORD: pass
            POSTGRES_DB: testdb
```

Ukázka kódu 3.1: Konfigurační soubor Docker kontejnerů potřebných pro chod backendové části aplikace

Díky tomu, že jsou všechny kontejnery definované v rámci jednoho konfiguračního souboru `docker-compose.yaml`, jsou také součástí jedné virtuální sítě. V rámci této sítě má každá služba svojí IP adresu, která je však maskována za názvem kontejneru. Naše PHP aplikace, která ke svému chodu potřebuje databázi, může k databázi `postgres_dev` poslouchající na portu `5432` přistoupit přes URL v podobě `postgres://postgres_dev:5432`.

Popis služeb, které jsou potřebné pro bezproblémový chod *frontendové* části aplikace je popsán souborem 3.2. Oproti *backendové* části je na první

pohled skromnější a spočívá prakticky pouze ve využití oficiálního *Node.js* Docker image. Pro úplnost odkažme i na příslušný *Dockerfile* dostupný v příloze C.1.

```
version: '3'
services:
    vue:
        build:
            context: .
            dockerfile: Dockerfile
        volumes:
            - .:/var/www/app
            - /var/www/app/node_modules
        ports:
            - 8080:8080
```

Ukázka kódu 3.2: Konfigurační soubor Docker kontejnerů potřebných pro chod frontendové části aplikace

Programátor, jenž potřebuje snadno a rychle takto připravené Docker prostředí využít při vývoji, nepotřebuje znát návrh jednotlivých služeb do detailu, aby je mohl používat. Stačí znát sadu několika málo příkazů pro práci s kontejnery:

`docker-compose build`
: Tento příkaz zapříčiní sestavení kontejnerů. To se děje na základě dodaného souboru `Dockerfile` pro jednotlivé kontejnery.

`docker-compose up`
: Jak už název napovídá, jedná se o příkaz k nastartování všech služeb. S přepínačem `-d` dojde ke spuštění kontejnerů na pozadí. Zároveň se vytvoří i příslušná virtuální síť.

`docker-compose run`
: Tento příkaz využijeme ke spuštění nástrojů a programů uvnitř samotných kontejnerů. Při spuštění musíme definovat kontejner, nad kterým chceme příkaz vykonat. Pro spuštění programu `bash` uvnitř kontejneru `php` využijeme tento příkaz ve tvaru `docker-compose run php bash`.

`docker-compose down`
: Po skončení práce se spuštěnými kontejnery je nutné celé Docker prostředí opět vypnout. Na tento úkon se často zapomíná, opomenuté spuštěné kontejnery však na pozadí stále spotřebovávají prostředky a blokují navázané porty.

Nemalou roli v projektu hraje využití připravené konfigurace Docker kontejnerů v *kontinuální integraci*, tu ale podrobněji rozebírám v samostatné podkapitole 3.4.1.2.

3.1.2 Zvýšení výkonu v macOS

Vlastnosti Dockeru, jež jsem popisoval v přechozí podkapitole, jsou bezesporu jeho silnými stránkami a může se zdát, že jeho použití na projektu skýtá jen samá pozitiva. Není však všechno zlato, co se třpytí. Na jedné straně využití Dockeru snížilo potřeby programátora provádět složité konfigurace vývojového prostředí, na straně druhé s sebou využití Dockeru nese za určitých okolností nepříjemné zvýšení nároků na výkon, se kterými jsem se musel vypořádat i já.

Jak je psáno i v oficiální Docker dokumentaci [8], při použití Dockeru (zvláště pak použití tzv. *volumes* – sdílených svazků mezi *hostujícím počítačem* a *Docker kontejnerem*) na operačním systému Linux nevzniká prakticky žádná režie navíc. Čtení a zápisy prováděné buď na hostujícím počítači nebo v kontejneru se v druhém prostředí projeví okamžitě díky využití nativních *asynchronních* Linuxových událostí `inotify` a `FSEvents` pro manipulaci se souborovým systémem. Nicméně, na jiných platformách, jako je macOS či Windows, vznikají v zájmu udržení dokonalé konzistence mezi hostujícím počítačem a kontejnerem velké režijní náklady, neboť zprávy popisující akce v souborovém systému se musí mezi hostujícím počítačem a kontejnerem předávat *synchronně*.

Malé projekty zvýšené režijní náklady nemusejí příliš trápit. Vývoj našeho projektu, jenž se skládá z mnoha závislostí na knihovnách třetích stran, to však od určité fáze nepříjemně limitovalo ve vývoji.

Vývojáři nástroje Docker si jsou těchto omezení dobře vědomí a snaží se poskytnout vývojářům takové nástroje, pomocí kterých lze výkon sdílených svazků zvýšit. Docker od verze `17.04` zavádí [8] u definice sdílených svazků tzv. *úrovně konzistence*. Ty ovlivňují, zda je obsah sdílených svazků zaručeně konzistentní, či jsou mezi nimi dovoleny dočasné nesrovnalosti:

`consistent`
: Zaručuje konzistenci souborů mezi hostujícím počítačem a kontejnerem, obě strany mají v jednu chvíli stejný obsah sdílených svazků. Tato hodnota je nastavena jako výchozí úroveň konzistence, pokud není uvedena přímo.

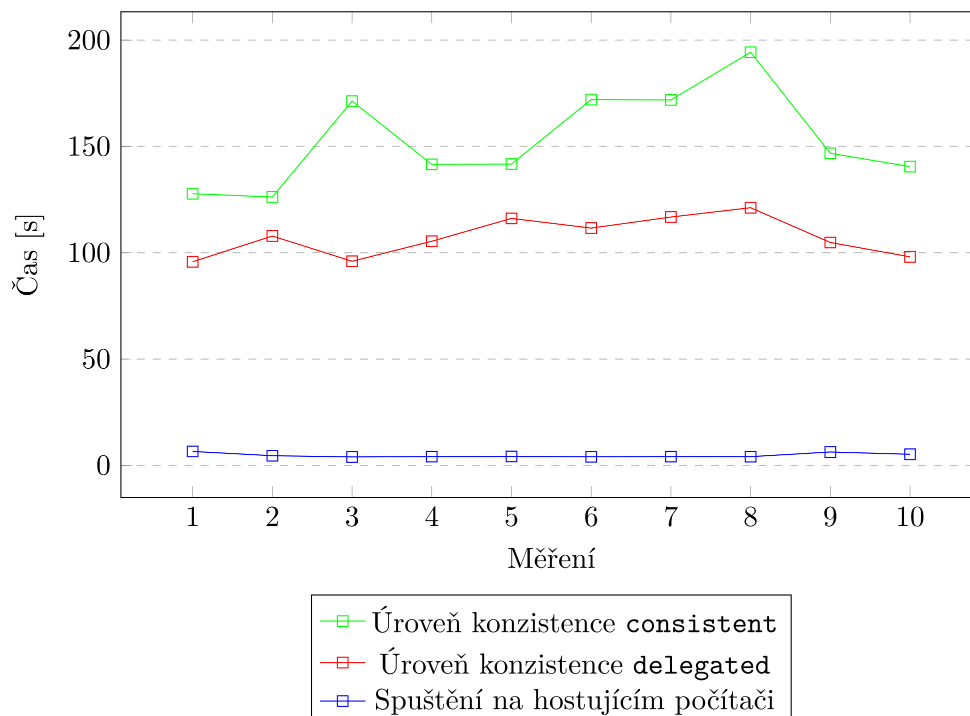
`cached`
: Obsah sdíleného svazku na straně hostujícího počítače je rozhodující, obsah svazku v kontejneru nemusí být vždy konzistentní s tím na straně hostujícího počítače.

`delegated`
: Obsah sdíleného svazku na straně kontejneru je rozhodující, zatímco obsah svazku na straně hostujícího počítače nemusí být vždy konzistentní s tím v kontejneru.

Výkonnost aplikace s různě nastavenými *úrovněmi konzistence* jsem se rozhodl ověřit experimentálně. Zvýšené nároky na režii se projeví při velkém

množství přesunu souborů v rámci sdíleného svazku mezi hostujícím počítačem a kontejnerem. Proto jsem pro účely měření zvolil proces *smazání a zahřátí cache* Symfony aplikace. Při této operaci dochází k mazání a vytváření velkého množství souborů, které jsou vytvářeny za účelem rychlejší odezvy Symfony aplikace.

Měření probíhalo na počítači s operačním systémem macOS 10.15.7 Catalina s procesorem Intel i5 2,7 GHz, s 8 GB RAM a s SSD s udávanou rychlostí čtení/zápisu 1 200 MB/s.



Obrázek 3.1: Výsledky měření výkonnosti Docker kontejneru s různými úrovněmi konzistence sdílených svazků

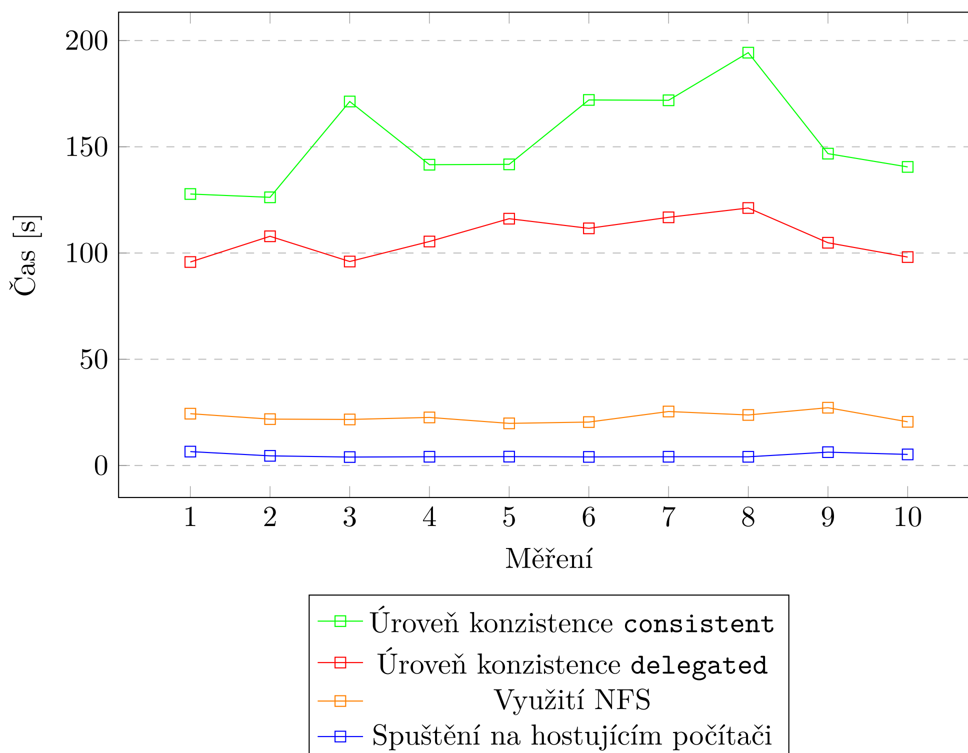
První měření, jehož výsledky jsou vyneseny v grafu 3.1, dopadlo dle očekávání. Spuštění náročné procedury pro smazání a inicializaci cache na hostujícím počítači (modrá osa), tedy bez použití Dockeru, nám dává jakousi dolní mez času, kterého můžeme při nejlepším dosáhnout i s použitím Dockeru. Medián z deseti naměřených časů je 4,1 s. Spuštěním této procedury s pomocí Dockeru na platformě Linux bychom dosáhli srovnatelného času.

Čas potřebný pro smazání a inicializaci cache uvnitř Docker kontejneru na platformě macOS má však k času naměřeném na hostujícím počítači daleko. V základní konfiguraci, tedy s použitím úrovně konzistence `consistent` (ze-

lená osa), je medián naměřených časů 144,3 s. To je 36krát více času, než kolik ho potřebovala procedura bez použití Dockeru. Toto zpomalení způsobuje, že Docker na platformě macOS je v základní konfiguraci pro větší projekty, které během svého vývoje potřebují přesouvat mezi sdílenými svazky velké množství souborů, prakticky nepoužitelný.

Jistou naději pro dosažení lepších časů skýtá využití slabší úrovně konzistence `delegated` (červená osa). Medián časů s touto konfigurací je 106,7 s. Ta má již na první pohled lepší výkonnost, oproti úrovni konzistence `consistent` je medián času dokonce o 26 % lepší. Stále je však tato konfigurace asi 25krát pomalejší než běh čistě na hostujícím počítači.

Zlepšení tedy nastalo, zrychlení ze začátku bylo znatelné a nějaký čas bylo možné se slabší úrovní konzistence sdíleného svazku pokračovat ve vývoji. S tím, jak základna zdrojového kódu projektu rostla, stalo se však i použití `delegated` jako nedostatečné pro pohodlný vývoj a bylo potřeba najít cestu k dalšímu zlepšení odezvy Docker kontejnerů.



Obrázek 3.2: Výsledky měření výkonnosti Docker kontejneru s využitím NFS pro sdílené svazky

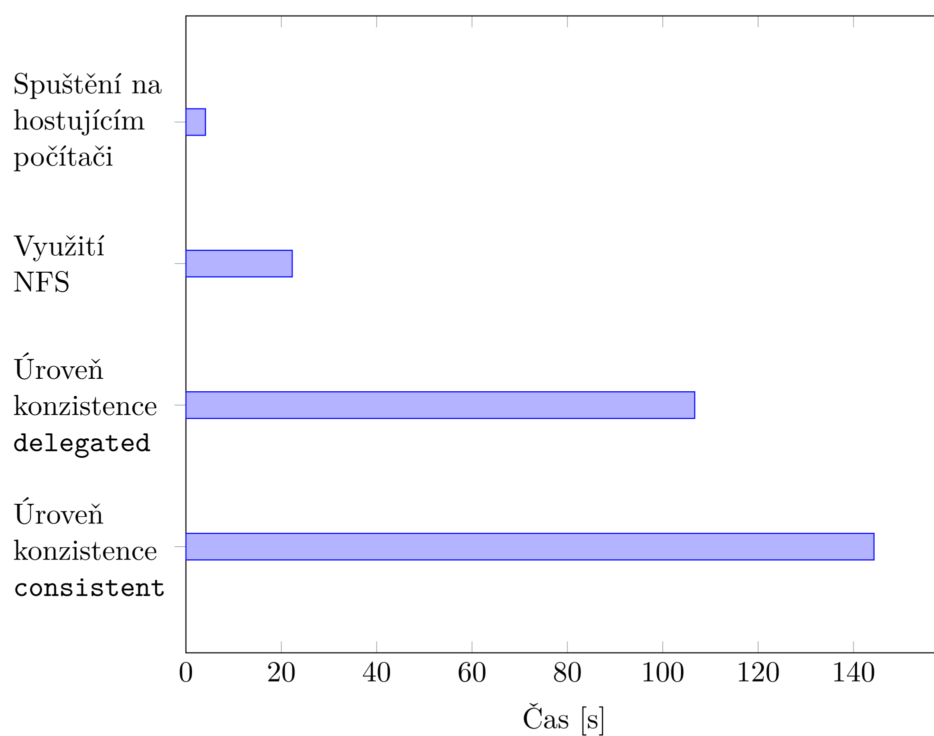
Jako řešení se ukázalo využití NFS[18]. O jeho využití ve spojení s Dockerem pojednává článek [9], který slibuje dramatický nárůst výkonu sdílených svazků. NFS je internetový protokol určený pro vzdálený přístup k souborům přes počítačovou síť. V praxi je potřeba definovat na straně hostitelského počítače *NFS server*, jenž bude definovat adresáře sdílené přes NFS. K jeho nastavení slouží soubor `/etc/exports`. Pro jeho automatickou konfiguraci jsem využil skript, který je dostupný v příloze C.4.

Kromě definice sdílených adresářů bylo potřeba říci Dockeru, že má připravené adresáře začít využívat. K tomu bylo nutné upravit konfigurační soubor *Docker Compose*, který je dostupný v příloze C.5. Klíčová je v něm definice nového sdíleného svazku, jenž využívá právě NFS.

Jak si však tato konfigurace vede v praxi v porovnání s různými úrovněmi konzistence[19]? Pohled na graf 3.2 dává autorům článku [9] za pravdu – nárůst výkonu je opravdu znatelný, dle mých měření (oranžová osa) se čas potřebný na smazání a inicializaci cache oproti využití úrovně konzistence `consistent` zlepšil o bezmála 85 %. Zároveň se jedná jen o pětkrát horší čas, než bez použití Dockeru. S touto konfigurací se již vývoj aplikace stal bezproblémový. Graf 3.3 na závěr shrnuje všechny mediány naměřených časů pro různá nastavení úrovně konzistence, včetně využití NFS.

[18]Network File System

[19]`consistent`, `delegated`



Obrázek 3.3: Porovnání mediánů měření výkonnosti Docker kontejneru s různými úrovněmi konzistence sdílených svazků

3.2 JetBrains IDE

S procesem vývoje jakéhokoli programu – tedy jeho zdrojového kódu – je neodmyslitelně spjatý nástroj pro jeho rychlé a efektivní psaní. K dispozici ke stažení je nesčetná řada takových nástrojů. Moje základní požadavky na IDE by se daly shrnout do následujících bodů:

- podpora syntaxe vybraných programovacích jazyků,
- chytré našeptávání kódu a kontrola sémantiky,
- možnost propojení s vybraným VCS[20],
- formátování kódu dle stanovených pravidel

Všechny tyto požadované vlastnosti naštěstí splňují nástroje od společnosti JetBrains [10], která nabízí nejen desítky vývojových prostředí pro různé programovací jazyky, ale i nástroje pro podporu vývoje v podobě projektového řízení nebo nástroje pro práci s databázemi. S těmito nástroji jsem se důvěrně sžil během mého vysokoškolského studia, použít IDE od JetBrains lze navíc pro školní projekty zcela zdarma. Pro účely vývoje byly použity následující nástroje:

WebStorm
IDE specializující se na vývoj aplikací v programovacím jazyku JavaScript pro frontendovou část aplikace,

PHPStorm
IDE specializující se na vývoj aplikací v programovacím jazyku PHP pro backendovou část aplikace,

DataGrip
IDE pro správu databází.

Na následujících řádcích se snažím o obhájení této volby vzhledem k požadavkům, které byly na volbu vývojového prostředí kladeny.

3.2.1 Podpora syntaxe programovacích jazyků

Tato podmínka si nárokuje schopnost IDE rozumět *syntaxi* daného programovacího jazyka, v našem případě se jedná především o jazyk PHP pro backendovou část projektu a JavaScript pro jeho frontendovou část. Je však potřeba myslet i na podporu jiných jazyků, které se používají pro šablony (Twig, HTML), pro různé konfigurační soubory (YAML, JSON), pro definování struktury databáze (SQL) či pro psaní dokumentace (Markdown). IDE by tedy

[20]Version Control System

mělo podporovat zvýraznění syntaxe programovacích jazyků, správné dodržení syntaxe a v případě porušení syntaxe programátora upozornit a říci, na které pozici v souboru se stala chyba. Demonstrace takového chování IDE je vidět na obrázku 3.4, kde zvýraznění syntaxe přehledně odlišuje klíčová slova jazyka (`public`, `function`, ...), proměnné (`$response`, ...) a třídní atributy (`$this->textureService`) a kde kontrola syntaxe kódu varuje programátora před chybějícím znakem „)“.

```
public function getTextures(): View
{
    $response = [];
    foreach ($this→textureService→getAll() as $texture) {
        $response[] = $this→textureService→serialize($texture;
    }                                                         Expected: )
    return View::create($response);
}
```

Obrázek 3.4: Zvýraznění a kontrola syntaxe v PHPStorm IDE

3.2.2 Chytré našeptávání kódu a kontrola sémantiky

Základním kamenem rychlého a pohodlného vývoje v jakémkoliv IDE je jeho předpovídání toho, co bude chtít programátor napsat. Pro tuto funkcionalitu musí IDE znát kontext (s proměnnou *jakého* typu právě pracuji, *jaká* je návratová hodnota konkrétní funkce, *jaké* veřejné metody má konkrétní objekt, ...). Na základě tohoto kontextu jsou programátorovi doporučovány další kroky, které může v tu chvíli provést, aby neporušil *sémantiku* programu. Ověřování sémantiky může programátora včas upozornit na nevalidní volání, aniž by byl program vůbec spuštěn. Ukázka kontroly sémantiky a funkce našeptávání na základě znalosti kontextu je dobře patrná z obrázku 3.5. Programátorovi jsou zde nabídnuty všechny metody, které lze volat nad instancí objektu v proměnné `$this->textureService`. Zároveň je proměnná `$texture` zabarvena šedě, což značí, že není nikde použita.

Nemalou měrou se na schopnosti jakéhokoliv IDE dobře analyzovat sémantiku programu podílí dodržování striktního typování programového kódu. To v případě PHP není nikterak samozřejmé, nicméně pravidla pro statickou analýzu kódu nás zavazují k tomu, abychom striktní typování důsledně dodržovali.

V případě JavaScriptu, pomocí kterého je psána frontendová část aplikace, je však dodržení striktního typování problematické, což komfort vývoje

Obrázek 3.5: Ukázka chytrého našeptávání v PHPStorm IDE

v tomto případě poměrně narušovalo. O podrobnější rozbor tohoto problému se snaží podkapitola 5.2.1.3.

Porozumět sémantice programu není pro IDE jednoduchý úkol a leckteré z nich tuto funkcionalitu ani nenabízejí.[21]

3.2.3 Možnost propojení s vybraným VCS

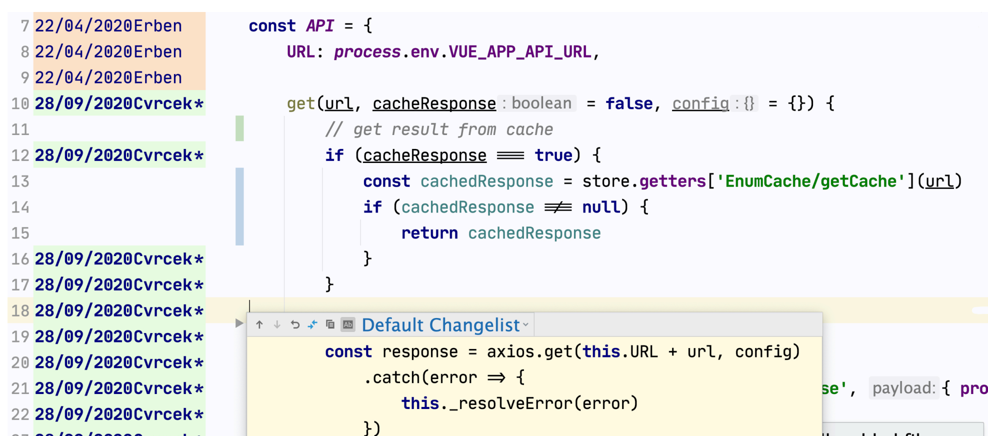
Propojení IDE s VCS je v dnešní době spíše samozřejmostí. Během vývoje dává programátorovi nenásilnou formou dobře najevo, jak se kód změnil od poslední revize. Jedním kliknutím lze zobrazit kompletní historii změn nejen celého souboru, ale i konkrétních řádků. Integrace VCS do nástrojů JetBrains umožňuje nejen provádění základních operací (stažení kódu z repozitáře, nahrání kódu do repozitáře), ale nesmírně usnadňuje i pokročilejší funkce – neocenitelný přínos má v případě řešení konfliktů při slévání dvou vývojových větví. O konkrétním VCS vybraném pro naši aplikaci pojednává více podkapitola 3.3.

Dobrá ukázka integrace VCS do IDE je vyobrazena na obrázku 3.6, kde lze vidět, kdy a kým byl naposledy změněn konkrétní řádek, zelenou barvou jsou označeny řádky, které byly od poslední revize *přidány*. Modrou barvou jsou znázorněny ty, které byly od poslední revize *upraveny*. Lze si také zobrazit ty řádky, které byly od poslední revize smazány.

3.2.4 Formátování kódu dle stanovených pravidel

Jednou z metrik kvality kódu je nejen jeho ověřená funkčnost, ale i jeho jednotný vzhled – tedy formátování – napříč soubory v celém projektu. Pokud je programový kód projektu spravován jedním programátorem, není až takový problém jednotný styl kódu zachovat, každý programátor dospěje časem

[21] Např. hojně rozšířený SublimeText



Obrázek 3.6: Ukázka integrace VCS do WebStorm IDE

k určitému stylu, který mu vyhovuje a takový styl má v ideálním případě nakonfigurovaný ve svém IDE, které mu k dodržení jednotného vzhledu kódu může dopomoci funkcí automatického formátování.

Problém nastává ve chvíli, kdy zdrojové kódy tvoří více programátorů. Každý by v tu chvíli přenesl do kusu kódu svůj programovací styl a celkový vzhled by nebyl jednotný. Nejednotná úprava kódu je pak velmi problematická ve spojení s libovolným VCS. Pokud na jednom souboru se zdrojovým kódem pracuje více programátorů, z nichž každý má nastavené jiné formátování kódu, a oba pravidelně ukládají svoji práci do jednoho VCS repozitáře, mohou si navzájem s každou změnou – která může být třeba jen na jeden řádek – přeformátovat celý upravovaný soubor. Jednotlivé revize kódu pak na první pohled vypadají, že se v rámci nich manipulovalo se všemi řádky souboru, ačkoliv programátor mohl reálně upravit pouze jeden řádek a zbytek řádků se změnil pouze kvůli rozdílnému nastavení automatického formátování kódu. Příklad takové revize, kde se změnilo pouze formátování zdrojového kódu, je vidět na obrázku 3.7.

Obrázek 3.7: Příklad revize zdrojového souboru, kde se změnilo pouze formátování kódu

Cílem je tedy donutit programátory podílející se na projektu, aby dodr-

žovali jednotné formátování kódu. Dobrým nástrojem, který se jednotný styl kódu snaží definovat, je *EditorConfig* [11]. Ten umožňuje definovat základní pravidla, kterých by se mělo jakékoliv IDE držet při automatickém formátování. Mezi taková pravidla patří mj. styl odsazení (mezery, nebo tabulátor) či povinnost mít na konci každého souboru prázdný řádek. Jednotlivá pravidla jsou definována v souboru `.editorconfig`. Ukázková konfigurace pro formátování kódu je k nahlédnutí v ukázce 3.3. Jedná se o konfiguraci pro frontendovou část projektu, konfigurační soubor pro backendovou část vypadá velmi podobně.

```
1  ; top-most EditorConfig file
2  root = true
3
4  ; Unix-style newlines
5  [*.{js, jsx, ts, tsx, vue}]
6  charset = utf-8
7  end_of_line = LF
8  indent_style = space
9  indent_size = 4
10 trim_trailing_whitespace = true
11 insert_final_newline = true
```

Ukázka kódu 3.3: Konfigurační soubor s pravidly pro formátování kódu frontendové části aplikace

Nástroj *EditorConfig* však nedovoluje konfigurovat spoustu pravidel specifických pro konkrétní jazyk. Naštěstí však všechna vývojová prostředí od JetBrains mají možnost uchovávat podrobná nastavení formátování v samostatném souboru ve formátu XML. Soubor 3.4 ukazuje, o jaká pravidla bylo nutno doplnit konfiguraci nástroje *EditorConfig*, aby automatické formátování kódu v IDE splnilo všechny naše požadavky na jednotný vzhled zdrojového kódu.

```
1  <code_scheme name="Photographix" version="173">
2    <JSCodeStyleSettings version="0">
3      <option name="USE_SEMICOLON_AFTER_STATEMENT" value="false" />
4      <option name="SPACE_BEFORE_FUNCTION_LEFT_PARENTH" value="false" />
5      <option name="USE_DOUBLE_QUOTES" value="false" />
6      <option name="SPACES_WITHIN_OBJECT_LITERAL_BRACES" value="true" />
7      <option name="SPACES_WITHIN_IMPORTS" value="true" />
8    </JSCodeStyleSettings>
9  </code_scheme>
```

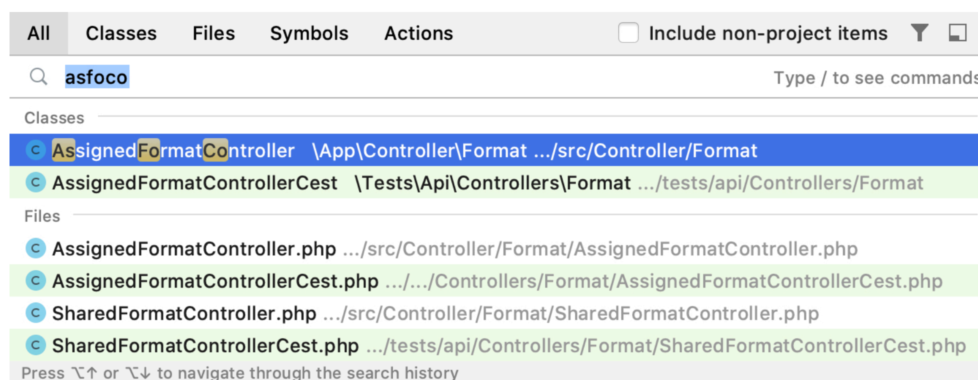
Ukázka kódu 3.4: Doplňkový konfigurační soubor pro JetBrains IDE s pravidly pro formátování kódu frontendové části aplikace

Všechny tyto poznatky by však byly v praxi k ničemu, kdyby jejich dodržo-

vání nebylo efektivně kontrolováno. Více o automatické kontrole formátování kódu pojednává samostatná podkapitola 5.1.6.

3.2.5 Další nepostradatelné funkce

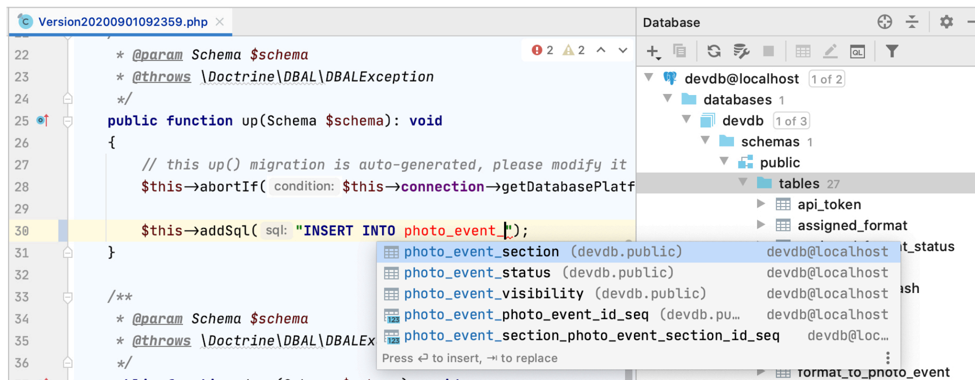
Nedílnou součásti každého IDE by mělo být vyhledávání mezi soubory. V projektech, které v průběhu času nabobtnají na velké množství souborů, je rychlejší nalézat kýžené soubory pomocí full-textového vyhledávání, než procházením adresářové struktury projektu. Vývojová prostředí od JetBrains navíc umožňují vyhledávat mezi soubory i podle fragmentů jejich názvu. Takové chování je ilustrováno na obrázku 3.8.



Obrázek 3.8: Příklad full-textového vyhledávání mezi soubory v PHPStorm IDE

Samozřejmostí je pak i vyhledávání pomocí regulárních výrazů a to nejen podle názvu souborů, ale i podle jejich obsahu. Ruku v ruce s vyhledáváním jde i nahrazování textu. V této oblasti nástroje od JetBrains nabízejí užitečnou funkci nahrazení textu se zachováním velkých a malých písmen.

Při vývoji projektu, který obsahuje databázi, je neocenitelnou výhodou její propojení s vývojovým prostředím. Díky tomu můžeme nejen nahlédnout do její struktury a dat, která obsahuje, ale získáme tím i výhodu našeptávání názvů tabulek, atributů a datových typů při psaní SQL dotazů. To v našem případě oceníme především při psaní databázových migrací. Ukázka propojení vývojového prostředí s databází je dobře patrná z obrázku 3.9.



Obrázek 3.9: Propojení vývojového prostředí PHPStorm s databází

3.3 Verzovací systém

Každý seriózní softwarový projekt se v dnešní době jen těžko obejde bez využití verzovacího systému, zkáceně VCS[22]. Doba, kdy se zdrojové kódy mezi jednotlivými členy vývojového týmu sdílely např. přes sdílený síťový adresář nebo se v horším případě kód distribuoval e-mailem, je naštěstí pryč. Využití libovolného VCS nástroje s sebou nese několik zásadních výhod:

- jednotlivé verze musí být popsány, a tak je na první pohled jasné, jakou funkci daná revize přináší,
- kooperace nad zdrojovými soubory s více lidmi,
- „přepnutí se" na libovolnou revizi z minulosti,
- zálohování kódu.

Systémů pro správu verzí je celá řada, některé se mezi sebou liší jen poměrně málo, každý z nich však spadá do dvou následujících skupin [12]:

Centralizované systémy správy verzí
Umožňují uchovávat verzované soubory na jednom centrálním *serveru*. Jednotliví *klienti* si ze serveru vyžádají jednotlivé soubory, které se chystají upravovat. Tyto vyžádané soubory v té chvíli přejdou do „uzamčeného" stavu, díky čemuž s daným souborem smí v jednu chvíli pracovat pouze jeden klient. Díky tomuto přístupu nehrozí, že mezi jednotlivými verzemi vznikne konflikt, který by se musel ručně vyřešit. Kvůli tomu, že je systém *centralizovaný*, lze server, který uchovává veškeré verzované soubory, označit za tzv. *SPOF*[23], kdy během výpadku takového serveru nelze verzovat vůbec. Dojde-li navíc k jeho fatální poruše, může dojít i ke ztrátě všech dat na něm. Mezi zástupce těchto systémů patří např. nástroj *Subversion*.

Distribuované systémy správy verzí
Mají oproti centralizovaným systémům výhodu v možnosti spolupráce více klientů nad stejnými soubory v jednu chvíli. Nedochází tu tedy, jako v případě centralizovaných nástrojů, k jejich uzamykání. To s sebou ovšem nese zodpovědnost některého z klientů za ošetření potenciálních konfliktů. Ty mohou vzniknout, když dva klienti upraví v jednom souboru stejný řádek. V případě distribuovaných systémů nedochází k potížím s *SPOF*, jako v případě centralizovaných systémů. Klient si totiž nestahuje jen nejnovější verze souborů, se kterými hodlá pracovat, ale

[22] Version Control System
[23] Single point of failure

stáhne si celou lokální kopii repozitáře, ze které v případě poruchy centrálního serveru lze provést obnovu. Mezi zástupce takového systému patří např. *Git* nebo *Mercurial*.

Společným znakem všech VCS je jejich efektivita, co se týče textových souborů. Pokud klient změní v textovém souboru jediný řádek a tuto změnu zapíše do repozitáře, uloží se opravdu jen změna onoho *jednoho řádku*, nikoliv celý upravovaný soubor. Jakmile se verzují i binární dokumenty, např. obrázky, stává se jejich verzování, co se nároků na úložiště týká, „drahé“, neboť s každou nepatrnou změnou v souboru se musí s novou revizí uložit *celý* soubor.

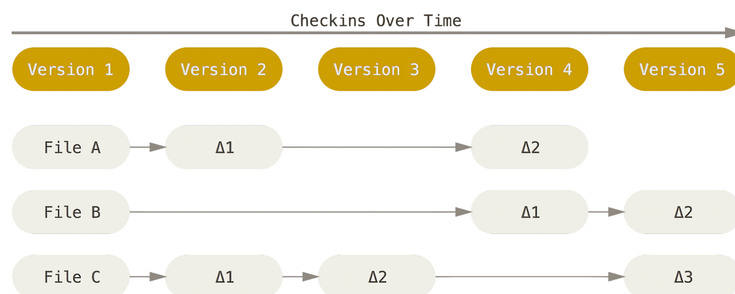
Čtenáře znalé této oblasti asi nepřekvapí, že v případě našeho projektu padla volba na verzovací systém *Git*.

3.3.1 Git

Git je zástupce *distribuovaných* systémů pro správu verzí, jehož historie sahá do roku 2005, kdy jeho vývoj inicioval Linus Torvalds, který se na jeho vývoji i aktivně podílel. Před jeho vznikem byly vytyčeny hlavní body, které měl nový systém splňovat, z nichž nejvýznamnějšími byly:

- distribuovanost,
- vysoká efektivita a rychlost,
- zaručení konzistence všech verzí, ať už z důvodu nehody či záměru.

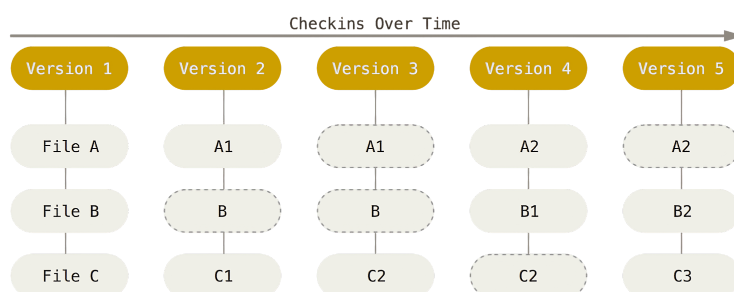
Jiné VCS systémy, např. *Subversion*, ukládají každou revizi jako seznam změn souborů. Pokud se klient potřebuje „přepnout“ na libovolnou verzi, musí se pro danou verzi *vypočítat* podoba souborů ze všech jejich změn. Toto chování je demonstrováno na obrázku 3.10.



Obrázek 3.10: Každá revize se *vypočte* na základě změn jednotlivých souborů, typické pro SVN *Subversion* [12]

Narozdíl od ostatních VCS nástrojů pracuje Git s rozdílnou filozofií, jak se na verzované soubory dívat. Ten o datech uvažuje jako o „sadě snímků“,

kdy každá revize (`commit`) představuje jeden snímek. Každý tento *snímek* zaznamenává, jak vypadají všechny soubory pro danou revizi. Soubory, které se od poslední revize nezměnily, nejsou v rámci efektivity znovu ukládány, jen je odkázáno na jejich poslední podobu. Vypočte jejich podobu a uloží je do databáze. Tento přístup na jednu stranu zvyšuje nároky na úložiště, na stranu druhou umožňuje téměř okamžité přepnutí mezi verzemi, neboť se soubory pro danou verzi nemusejí *vypočítávat*. Tato filozofie Gitu je dobře patrná z obrázku 3.11.



Obrázek 3.11: Každá revize je „snímek“ celého projektu, který obsahuje odkazy na soubory aktuální pro danou verzi, typické pro SVN *Git* [12]

Pro každý soubor, který je verzovaný pomocí SVN Git, platí, že se nachází v jednom z následujících stavů:

Staged
: Soubor byl od posledního `commitu` upraven programátorem.

Modified
: Soubor byl předán k zapsání, ale samotné zapsání (`commit`) ještě neproběhlo.

Commited
: Soubor byl zapsán v lokální Git databázi.

Můžeme si tedy představit, že z hlediska Gitu je projekt rozdělen do tří částí:

Working directory
: Neboli *pracovní adresář*. Jedná se o jednu konkrétní revizi projektu, v rámci které pracujeme se soubory.

Staging area
: Neboli *oblast připravených změn*. Obsahuje informace o změněných souborech, které jsou určeny k další revizi, tedy takové, které budou součástí dalšího `commitu`.

Repository

Neboli *Git repozitář*. V praxi se jedná o adresář s názvem `.git`, který v sobě uchovává veškerá data (databáze verzí, metadata, ...).

Typický pracovní postup pro verzování souborů v rámci Gitu je tedy přibližně následující:

1. změna souborů v pracovním adresáři,
2. příprava změněných souborů k revizi vložením do *oblasti připřavených změn*,
3. zápis změn (`commit`), který převede soubory přidané do *oblasti připřavených změn* na další revizi.

Git disponuje celou řadou příkazů, z nich pro základní práci stačí ovládat dobře jen pár z nich:

`git init`

Vytvoří prázdný Git repozitář.

`git status`

Vypíše aktuální informace o *pracovním adresáři* a *oblasti připravených změn.*

`git add`

Přesune vybrané soubory z *pracovního adresáře* do *oblasti připravených změn.*

`git commit`

Vytvoří revizi ze souborů umístěných v *oblasti připravených změn.*

`git checkout`

„Přepne“ pracovní adresář na konkrétní revizi, typicky pomocí heše, který identifikuje určitý `commit`. Pomocí tohoto příkazu se lze také přepnout do jiné větve.

`git merge`

Sloučí danou větev s větví současnou.

`git tag`

Pro daný `commit` vytvoří „štítek“[24], tedy lidsky čitelný název `commitu`.

`git log`

Vypíše historii `commitů` v současné větvi.

[24] Např. v1.0.5

`git pull`
: Stáhne změny ze vzdáleného repozitáře a sloučí případné rozdíly.

`git push`
: Uloží změny do vzdáleného repozitáře.

Každý `commit` je identifikován svým hešem, což je řetězec čtyřiceti hexadecimálních znaků, který je vypočten z aktuálně změněných souborů i ze všech minulých revizí. Při jakékoliv operaci s danou revizí Git kontroluje přepočtením heše jeho integritu, čímž je zaručeno, že historii revizí nelze jednoduše měnit – ať už záměrně jako útočník, či nechtěně jako nezkušený uživatel.

Konkrétní systém pro správu verzí úzce souvisí s volbou vzdáleného repozitáře zdrojových kódů, který musí vybrané VCS podporovat.

3.4 Vzdálený repozitář zdrojových kódů

V předchozí podkapitole 3.3 byly nastíněny základní rozdíly mezi centralizovanými a distribuovanými systémy. Zvolený verzovací systém *Git* spadá sice do kategorie distribuovaných systémů, i v jeho případě však můžeme mluvit o jeho jakési *centrální instanci*, v jazyce Gitu nazývané *remote*, neboli *vzdálený repozitář*. Tato verze Git repozitáře je zpravidla dostupná z internetu nebo např. z interní firemní sítě.

Git ve svém základu počítá s možností umístit si vlastní repozitáře na námi spravovaný server. Ve zkratce předpokládejme, že k takovému serveru, jenž nese název `git.example.com`, má přístup uživatel `user` přes SSH[25] protokol. Na tomto serveru chceme mít v adresáři `repositories` umístěný Git repozitář s názvem `project.git`. Potom můžeme získat lokální kopii takového repozitáře s pomocí následujícího příkazu:

```
git clone user@git.example.com:/repositories/project.git
```

K našemu štěstí není v dnešní době nutné spravovat vlastní server, který by sloužil jako úložiště pro naše Git repozitáře. S rozmachem nástroje Git se na světlo světa dostaly služby, které ve své podstatě vytváření a správu vzdálených Git repozitářů zprostředkují za nás. Takových služeb je dnes celá řada, sluší se zde mínit alespoň ty nejrozšířenější, mezi které patří *GitHub*, *Bitbucket* nebo *GitLab*.

Naše nároky by však byly poněkud nízké, pokud bychom od služeb tohoto typu očekávali *pouze* hostování našich repozitářů. Mezi základní funkce vzdáleného repozitáře zdrojových kódů by se měly řadit minimálně následující body:

- správa projektů a projektových skupin,

[25]Secure Shell

- správa uživatelů a jejich práv,
- zakládání tématických diskusí,
- mechanismus ke slučování větví kódu,
- možnost kontinuální integrace a kontinuální dodávky.

Z výše zmíněných nástrojů byl pro náš projekt zvolen nástroj *GitLab*, který podrobněji popisujeme v následující podkapitole 3.4.1.

3.4.1 GitLab

GitLab je samotnými autory služby popisován jako *kompletní platforma pro DevOps* [13]. Pod označením *DevOps*, jenž vzniklo spojením slov pro vývoj (*Development*) a provoz (*Operations*), si můžeme představit propojení procesů, technologií a lidí mající za cíl průběžné doručování – v našem případě softwarových – produktů.

Toto je poněkud abstraktní definice, která na první pohled nemusí být každému jasná. V praxi se jedná o snahu co nejvíce automatizovat jednotlivé kroky vývoje. Vše musí být verzováno a testováno, to neplatí jen pro zdrojové kódy, ale i pro podpůrné skripty či konfigurační soubory. Průběžnému testování aplikace, zvanému *kontinuální integrace*, je věnována podkapitola 3.4.1.2. Kromě vývoje se počítá i s automatizací samotného nasazení aplikace na všechna cílové prostředí. Tento proces se označuje *kontinuální dodávka* a ve zkratce se mu věnuje podkapitola 3.4.1.3.

Služba *GitLab* započala svůj vývoj teprve v roce 2014. I přes svou relativně krátkou dobu existence již přilákala více než 30 milionů registrovaných uživatelů, což z ní dělá jednu z nejpoužívanějších služeb tohoto druhu.

Výjmenovat všechny užitečné funkce by bylo na dlouho, dovolte mi zmínit se tu o těch základních, které využíváme v našem projektu.

3.4.1.1 Merge requests

Bez nadsázky nejpotřebnější nástroj, jenž přidává *GitLab* základnímu *Gitu* je nástroj pro správu a slučování jedné větve kódu do druhé. Tomuto mechanismu se vžil název *merge requests*[26], tedy volně přeloženo *žádost o sloučení.*

Žádost o sloučení musí být založena pro dvě větve. Jedna je zdrojová („source“), druhá cílová („target“). Založená žádost o sloučení je pak přidělena uživateli, který by ji měl schválit.

GitLab sám při založení žádosti automaticky zkontroluje, zda je možné sloučení dvou větví provést. Pokud by mezi danými dvěma větvemi nastal konflikt, je na vybraném programátorovi, aby konflikty vyřešil a do zdrojové větvě poslal *commit*, jenž tyto konlikty řeší.

[26]Konkurenční *GitHub* i *Bitbucket* označují podobnou funkcionalitu jako *pull request*

Tohoto mechanismu se v praxi využívá v případě, kdy je potřeba do hlavní větve přidat změny z vedlejší větve, která s sebou nese např. nějakou novou funkcionalitu, či opravu chyby. Založený *merge request* s žádostí o začlenění nové funkcionality do hlavní větve však může být autorovi vrácen s konkrétními připomínkami k přepracování. Po zapracování takových připomínek je dodatečný *commit* přidán do již existující žádosti o sloučení a dříve vytvořené připomínky mohou být označovány jako *vyřešené*. Ukázka využití žádostí o sloučení je znázorněna na obrázku 3.12.

Obrázek 3.12: Využití diskusí v rámci *merge requests*

Pokud je v projektu souběžně nastavena nějaká forma kontinuální inte-

grace, lze nastavit správcem projektu, aby nemohlo dojít k přijmutí žádosti o sloučení, pokud kontinuální integrace neskončí bez chyby. Samozřejmostí je i možnost definovat, který uživatel má pro danou cílovou větev právo tyto žádosti o sloučení schvalovat.

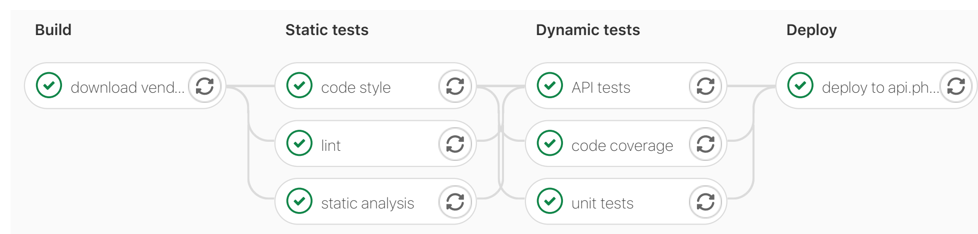
3.4.1.2 Kontinuální integrace

Kontinuální integrace je automatizovaný proces, jenž s každým novým *commitem* a jeho uložením do vzdáleného repozitáře spustí sérii operací, které mají za úkol prověřit současný stav kódu.

S každým novým *commitem* by měl mít jeho autor, ale i kterýkoli jiný člen týmu přehled o tom, v jaké „kondici" se v dané verzi kód nachází. Ke změření takové kondice se využívají nejrůznější metriky, ty jsou však podrobně probrány v samostatné podkapitole 5.1.6.

Ve spojení se službou *GitLab* se zde sluší zmínit, *jakým způsobem* se v rámci ní dá dříve definovaná kvalita kódu sledovat v čase a napříč různými revizemi. Takový pohled je důležitý nejen pro samotné vývojáře, ale v praxi i pro vedoucího vývojáře, či manažera projektu, který na základě něj může rozhodovat, jakým směrem se bude ubírat další vývoj.

Konkrétní konfigurace kontinuální integrace je k nahlédnutí v 1. a 2. části konfiguračního skriptu `.gitlab-ci.yaml` pro *GitLab CI/CD*[27] v přílize C.6 a C.7. Tato konfigurace odpovídá struktuře, která je dobře patrná z obrázku 3.13, popisu této struktury se věnuji níže.



Obrázek 3.13: Ukázka struktury *pipeline* v rámci *Gitlab CI/CD* náležící k *backendové* části aplikace

S pomocí výše zmíněného konfiguračního skriptu se definují *jednotky práce*, tzv. *jobs*. Takovou jednotkou může být např. otestování kódu statickou analýzou, či provedení jedné sady dynamických testů. Prakticky vzato se jedná o spuštění nějakého *programu* či *skriptu*. Na základě návratového kódu takového skriptu či programu se pak celá jednotka práce vyhodnotí jako úspěšně dokončená, jako skončená s chybou, či může být za jistých předem definovaných okolností přeskočena.

[27] GitLab Continuous Integration & Continuous Delivery

Jednotlivé *jednotky práce* se shlukují do *etap*, tzv. *stages*. Každá etapa obsahuje skupinu jednotek práce, které mají něco společného. Např. spadají do jedné kategorie testů, zajišťují sestavení aplikace, apod.

Všechny etapy společně tvoří jednu tzv. *pipeline*. Příklad jedné takové je znázorněn na obrázku 3.13. Na něm můžeme vidět *pipeline* pro *backendovou* část projektu, jež se skládá ze čtyř *etap – build*, *static tests*, *dynamic tests* a *deploy*. Každá z etap pak obsahuje i několik *jednotek práce*. V případě etapy *static tests* se jedná o *lint*, *code style*, *static analysis*.

Grafická podoba *pipeline* dává dobrý přehled o tom, jak jsou její jednotlivé části na sobě závislé. Jednotky práce z jedné etapy se začnou vykonávat jen tehdy, pokud všechny jednotky práce z etapy předchozí neskončily s chybou. Jednotky práce v rámci jedné etapy se mohou vykonávat paralelně, z toho vyplývá i to, že pokud v rámci jedné etapy selže jedna jednotka práce, ostatní se i přes to provedou.

Všímavému čtenáři jistě neujde fakt, že jedna z *jednotek práce* v rámci etapy *dynamic tests* se nazývá *code coverage*. Tato část má za úkol vypočítat procentuální pokrytí kódu testy. Tato důležitá metrika se dá sledovat v čase a celý tým tak může mít dobrý přehled o vývoji pokrytí kódu testy. Ukázka výstupu procentuálního pokrytí testy je dobře vidět na obrázku 3.14.

Obrázek 3.14: Výstup jednotky práce pro výpočet procentuálního pokrytí zdrojového kódu testy

Důležité je zde zmínit, v jakém prostředí všechny výše popsané procesy běží. Programátora znalého *GitLab CI/CD* nepřekvapí, že všechny *jednotky práce* se provádějí v rámci *Docker* kontejnerů. Tento pracovní *Docker* kontejner se dá definovat pro všechny jednotky práce v rámci konfiguračního souboru `.gitlab-ci.yaml`, zpravidla hned na jeho prvním řádku. Na tomto místě můžeme využít libovolné *Docker* kontejnery, se kterými pracuje naše konfigurace vývojového prostředí, které bylo podrobněji popsáno v podkapitole 3.1.

My jsme však v našem projektu šli ještě o krok dál a pro použití v rámci *GitLab CI/CD* jsme z *Docker* kontejnerů, které používáme při lokálním vývoji, vytvořili *Docker* image. Díky tomu máme jistotu, že prostředí pro vývoj i pro kontinuální integraci jsou prakticky identická a na obou prostředích se bude náš spuštěný kód chovat stejně. Samotný *GitLab* má pro tento případ užití podporu v podobě *container registry*, tedy jakéhosi repozitáře pro *Docker* images, jež lze poté použít v rámci *GitLab CI/CD* daného projektu.

3.4.1.3 Kontinuální dodávka

Dalším logickým krokem v automatizaci vývoje naší aplikace je *kontinuální dodávka*, anglicky pak *continuous delivery*, ve zkratce *CD*. Zde si hned na začátek dovolím osvětlit rozdíl mezi dvěma často zaměňovanými pojmy, které si jsou na první pohled velmi podobné, každý z nich však znamená něco trochu jiného, ikdyž se oba termíny schovají za stejnou zkratku *CD*.

Continuous delivery
: Neboli *kontinuální dodávka*. Navazuje na *kontinuální integraci*, po které je aplikace automaticky sestavena. Takto sestavená aplikace je připravena k nasazení na cílové prostředí, spuštění tohoto procesu pak probíhá *manuálně*, typicky na pokyn manažera projektu.

Continuous deployment
: Neboli *kontinuální nasazení*. Opět navazuje přímo na *kontinuální integraci* a i zde se aplikace sestavuje do „nasaditelného“ stavu. Nasazení sestavené podoby aplikace však v tomto případě probíhá *automaticky* a neustále.

Obě varianty mají svá pro a proti. Na první pohled se může přístup *kontinuálního nasazení* zdát výhodnější. Pro malé projekty snad. Je potřeba myslet na to, aby ruku v ruce s *kontinuálním nasazením* byly aktivovány doprovodné obchodní procesy, které může nasazení nové verze aplikace ovlivnit – musí se např. aktualizovat dokumentace či notifikovat systémy třetí strany, které jsou na naši aplikaci napojeny, aby se přizpůsobily změnám, které byly právě nasazeny. Tento proces nemusí být vůbec triviální a vyžaduje téměř kompletní automatizaci v celé podnikové struktuře.

I z toho důvodu se častěji zůstává u varianty *kontinuální dodávky*. Ne jinak je tomu i v případě našeho projektu.

Konfigurace kontinuální dodávky je v našem případě také součástí skriptu pro *GitLab CI/CD*. Její nastavení je k vidění v příloze C.7. V našem případě byla pro nasazení zvolena kombinace nástrojů *GitLab CI/CD* a *Deployer*[28].

Deployer je jednoduchý PHP nástroj, který podle zadaného konfiguračního souboru, v řeči nástroje *Deployer* se mu řídá *recipe*, nasadí sestavenou podobu aplikace na cílové prostředí. Díky možnosti definovat doprovodné skripty provede i její automatizovanou konfiguraci. Typicky se v takovém případě jedná o následující akce:

- definice *sdílených adresářů a souborů*, které se s každým dalším nasazením nepřepíší,[29]

[28] https://deployer.org

[29] Typicky se jedná o konfigurační soubory prostředí, či o adresáře obsahující data, které generuje samotná aplikace (obrázky, cache, ...).

- spuštění databázových migrací,
- smazání a inicializace *cache* aplikace.

Ukázka konfigurace nástroje *Deployer*, která se používala během vývoje k nasazení na testovací prostředí je k vidění v příloze C.8.

3.4.1.4 Projektová dokumentace

Sebelepší softwarový produkt je z hlediska vývoje nepoužitelný, pokud nejsou na jednom místě udržovány informace o tom, jak s projektem nakládat. U projektu, na kterém kooperuje více lidí, je to nutnost.

Základní body, jež by měla kýžená dokumentace pokrývat, jsou mimo jiné:

- instalace aplikace,
- lokální nasazení aplikace,
- nastavení lokálního vývojového prostředí,
- základní projektové pracovní postupy.

Samotný nástroj *GitLab* nabízí pro účely sepisování projektové dokumentace v základu dva mechanismy, oba jsme využili i v našich projektech:

- renderování Markdown souborů s názvem `README` umístěných v kořenovém adresáři projektu,
- projektové a skupinové *Wiki* stránky.

V rámci každého projektu byl udržován soubor s názvem `README`[30], jenž obsahuje instrukce k lokálnímu nasazení daného projektu pro účely vývoje. Případně jsou v něm popsány *osvědčené postupy* použité při vývoji, mezi které se může řadit např. organizace adresářové struktury projektu, odkazy na definice stylu formátování kódu, aj. Neměla by tu chybět ani zmínka o tom, jak spustit aplikační testy, či kam se obrátit, pokud při vývoji narazíme na nějakou chybu. Úryvek jedné sekce ze souboru `README` *backendové* části projektu, jenž popisuje *osvědčený postup*, jak vytvářet databázové entity přes příkazovou řádku, je k nahlédnutí v příloze D.1.

Pokud projekt sestává z jednoho repozitáře, lze ve výše zmíněném souboru `README` udržovat i informace o projektových pracovních postupech, nastavení vývojového prostředí, aj. V opačném případě – pokud projekt sestává z více repozitářů, jak tomu je i v případě naší aplikace – se hodí mít tyto informace separátně oddělené v samostatné sekci. Právě pro tyto účely nabízí *GitLab* projektové a skupinové *Wiki* stránky. Ty lze udržovat, stejně jako soubory

[30]Obsah psán jazykem *Markdown*

`README`, v jazyku *Markdown*. V rámci *GitLab Wiki* lze tvořit hierarchie stránek, odkazovat se z jedné stránky na jiné či přikládat ke stránkám přílohy. V případě naší aplikace udržujeme *GitLab Wiki* stránky se souhrny projektových pracovních postupů a s obecnými postupy pro nastavení vývojového prostředí, jež jsou společné pro více projektů.

Zdrojové soubory *GitLab Wiki* jsou interně spravovány jako samostatné *Git* repozitáře. Je tedy možné celou dokumentaci vyvíjet lokálně a nikoliv pouze skrze webové rozhraní.

V obou případech je podporována speciální verze značkovacího jazyka *Markdown*, jež se nazývá *GitLab Flavored Markdown*. Ten kromě základní stylizace textu umožňuje vkládání obrázků, vytváření tabulek či vkládání úryvků kódu, které mohou být posléze vyrenderovány se zvýrazenou syntaxí daného programovacího jazyka.

Kapitola **4**

Projektové řízení

Právě na úvod této kapitoly se sluší říci, že jsem celou dobu nebyl na vývoj služby sám. Využil jsem nabídky Ing. Jiřího Hunky, který na Fakultě informačních technologií ČVUT pravidelně zaštiťuje několik projektů v rámci předmětů BI-SP1 a BI-SP2. Právě vybraní studenti, kteří si tyto předměty zapsali, mi pomáhali na projektu s dílčími úkoly. O organizaci předmětu více pojednává podkapitola 4.2.

Na počátku vývoje kteréhokoliv softwarového projektu je nutné vymezit si alespoň rámcově metodiku projektového řízení, od které se později odvíjí způsob kooperace jednotlivých členů týmu. Popisu alespoň těch základních a nejčastěji uplatňovaných se věnuje podkapitola 4.1.

Praktické dopady koronavirové pandemie, jež se na celém světě projevila hned na začátku vývoje naší aplikace, měly za následek mimo jiné výhradně distanční formu spolupráce mezi všemi členy týmu. Efektivita naší spolupráce v takové situaci stojí a padá se zvolenými softwarovými nástroji. O nich se zmiňuji v podkapitole 4.3.

4.1 Metodiky vývoje software

V této podkapitole připomenu dva základní směry vývoje software[31], jimiž jsou model *Waterfall* a metodiky *Agile*, což je pouze zaštiťující název pro metody jako je *Scrum*, *Extrémní programování*, atd. Na úvod je třeba říci, že tyto metodiky jsou do jisté míry pouze teoretické s jejich implementací v jejich čiré podobě se prakticky nesetkáme. Řízení konkrétního projektu však může z daných metodik vybrat takové vlastnosti, jež jsou pro daný projekt vhodné.

[31] A ne jenom software

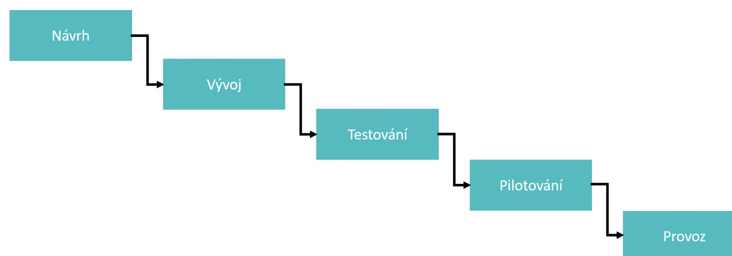
4.1.1 Waterfall

Waterfall je klasický, *sekvenční* proces vývoje, který je charakteristický tím, že se zpravidla žádná z jeho fází neopakuje. Teoreticky je dovoleno se vrátit vždy jen o jeden krok zpět. Paradoxem je, že *Waterfall* byl představen [14] jako příklad *špatného* modelu vývoje. Princip vodopádu je totiž mnohem bližší oborům, které nepracují s tak „tvárným materiálem“, jakým je software. Mezi takové obory můžeme počítat např. stavební průmysl, kdy před stavbou domu je zpravidla nutné mít vyhotovenou veškerou potřebnou dokumentaci a obvodové zdivo lze stavět až ve chvíli, kdy máme pevné základy.

Jednotlivé fáze vodopádového modelu aplikovaného na vývoj software jsou potom následující:

- analýza požadavků a návrh,
- vývoj,
- testování,
- pilotní provoz,
- ostrý provoz a údržba.

Vizualizace průběhu vývoje projektu, u kterého bylo využito metodiky *Waterfall*, je k vidění na obrázku 4.1.



Obrázek 4.1: Cyklus vývoje metodou Waterfall [15]

Sekvenční přístup prakticky vylučuje možnost pracovat na více vývojových fázích současně a k další fázi je vždy nutné přejít až ve chvíli, kdy je současná fáze kompletně vyřešena. Tento přístup velmi omezuje pružnost alokace lidských zdrojů pro daný projekt. *Waterfall* model do jisté míry razí heslo „dvakrát měř, jednou řež“, často je však obtížné každou z fází vývoje odladit k dokonalosti, aniž by se na základě ní začalo pracovat na fázi následující.

Pro tento model však hovoří fakt, že pokud je drtivá většina chyb v softwarovém projektu odhalena v raných fázích vývoje, je jejich oprava relativně levná. McConnel [16] tvrdí, že „odstranění chyby v požadavcích, kterou se nepodaří odhalit až do fáze implementace nebo údržby, stojí 50 krát až 200 krát

více, než kdyby se taková chyba odhalila a napravila již v etapě specifikace požadavků".

Každodenní realita vývoje softwarových produktů nám však ukazuje, že málokdy jsou *všechny* požadavky známy již v době zadání od zákazníka, velmi často přicházejí požadavky stále nové a nové. S takovou situací neumí tento model dobře pracovat.

4.1.2 Agile

Agilní přistup k vývoji software vznikl jako protiklad ke klasickým metodikám, mezi které patří např. výše zmíněný *Waterfall*. „Odlehčené" agilní metodiky, jež si kladly za cíl především zefektivnit vývoj, se začaly objevovat na počátku devadesátých let. Souhrn těchto odlehčených metodik se začal nazývat *agilními* poté, co dala skupina autorů[32] vzniknout manifestu agilního programování [17]. Ten definuje několik priorit, kterými se vymezují proti klasickým metodikám. Tyto si zde dovoluji přímo odcitovat ze stránek manifestu:

- jednotlivci a interakce před procesy a nástroji,
- fungující software před vyčerpávající dokumentací,
- spolupráce se zákazníkem před vyjednáváním o smlouvě,
- reagování na změny před dodržováním plánu.

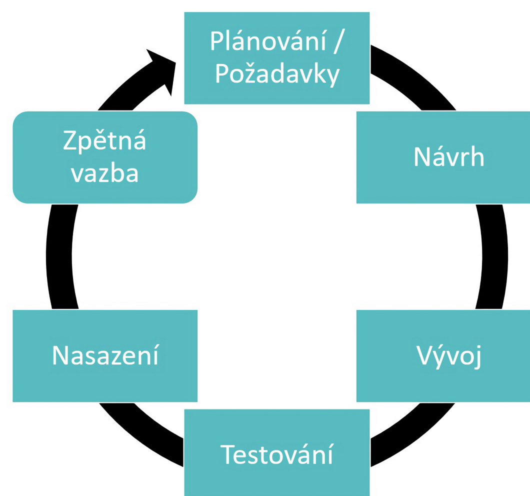
Manifest rovněž definuje několik základních principů:

- Naší nejvyšší prioritou je vyhovět zákazníkovi časným a průběžným dodáváním hodnotného softwaru.
- Vítáme změny v požadavcích, a to i v pozdějších fázích vývoje. Agilní procesy podporují změny vedoucí ke zvýšení konkurenceschopnosti zákazníka.
- Dodáváme fungující software v intervalech týdnů až měsíců, s preferencí kratší periody.
- Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu.
- Budujeme projekty kolem motivovaných jednotlivců. Vytváříme jim prostředí, podporujeme jejich potřeby a důvěřujeme, že odvedou dobrou práci.

[32] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland a Dave Thomas

- Nejúčinnějším a nejefektnějším způsobem sdělování informací vývojovému týmu z vnějšku i uvnitř něj je osobní konverzace.
- Hlavním měřítkem pokroku je fungující software.
- Agilní procesy podporují udržitelný rozvoj. Sponzoři, vývojáři i uživatelé by měli být schopni udržet stálé tempo trvale.
- Agilitu zvyšuje neustálá pozornost věnovaná technické výjimečnosti a dobrému designu.
- Jednoduchost – umění maximalizovat množství nevykonané práce – je klíčová.
- Nejlepší architektury, požadavky a návrhy vzejdou ze samo-organizujících se týmů.
- Tým se pravidelně zamýšlí nad tím, jak se stát efektivnějším, a následně koriguje a přizpůsobuje své chování a zvyklosti.

Agilní metodiky jsou zpravidla *iterační* a jsou založeny na inkrementálním vývoji produktu. Pro lepší představu jsou jednotlivé části vývoje, při němž je využita agilní metodika, znázorněny na obrázku 4.2.



Obrázek 4.2: Cyklus vývoje metodou Agile [15]

4.1.2.1 Scrum

Dle [18] není *scrum* proces zaměřený na praktiky a pravidla, je to empirický proces, který se snaží co nejvíce reagovat na potřeby vnějšího světa. Hlavním cílem celé metody je dodávka produktu v relativně krátkých cyklech tak, aby

tým vývojářů získával průběžnou zpětnou vazbu. V rámci celého vývojového procesu zavádí *Scrum* tři role, jež dohromady tvoří tzv. *Scrum team*:

- Product development team – sebeorganizující se tým lidí, kteří jsou na konci každého *sprintu* schopni dodat použitelný *inkrement*,
- Scrum master – stará se o to, aby *product development team* fungoval tak, jak by měl,
- Product owner – stará se o prioritizaci jednotlivých úkolů v rámci *sprintů*.

Kromě výše zmíněných rolí je pro *Scrum* klíčový *Product backlog*, tedy proritizovaný seznam funkcionalit, jež chceme zákazníkovi dodat. Funkcionality jsou dodávány v krátkých iteracích, jež se nazývají *sprinty*. Sprint má zpravidla fixní délku, obvykle kolem dvou týdnů. Každý pracovní den obvykle začíná *Daily Scrum* schůzkou, na které se probírají cíle aktuálního sprintu. Konec každého *Sprintu* by měl přinášet tzv. *inkrement* – nový funkční přírůstek do rozpracovaného produktu, na který v rámci *Sprint review* získává vývojový tým zpětnou vazbu od zákazníka.

Nedílnou součástí celé metodiky *Scrum* je pak *retrospektiva*, v rámci které celý tým identifikuje procesy, ve kterých má ještě rezervy a ve kterých by se mohl do budoucna zlepšit.

4.1.2.2 Extrémní programování

Oficiální příručka extrémního programování [19] charakterizuje tuto metodiku jedním základním heslem: „Pokud se nějaké postupy v minulosti osvědčily, pak tyto postupy provádíme neusátle.“ Prakticky se tak dovádějí do extrému postupy, které jsou známé z běžného procesu vývoje software.

Dobrým příkladem takového extrému je *revize kódu*. Pakliže se revize kódu v minulosti osvědčila, pak se reviduje *neustále*. Toho může být docíleno např. *párovým programováním*, kdy jeden kód společně v daný čas vyvíjejí dva programátoři – jeden píše kód[33], druhý po něm kód kontroluje[34]. Tito programátoři si role pravidelně vyměňují.

Osvědčilo se v minulosti pokrytí kódu testy? V případě extrémního programování se tak testuje *neustále*. Obvykle se ke každému přírůstku užitečného kódu píší zároveň minimálně *jednotkové testy*.

Jednotlivé fáze vývoje v rámci extrémního programování jsou potom následující:

Zadání

Základem je soupis *User stories* – typických scénářů toho, jak se bude výsledný produkt používat a to na základě profilů typických uživatelů.

[33] Tzv. *Driver*
[34] Tzv. *Observer*

Počítá se s tím, že zákazník může v průběhu vývoje měnit své požadavky, pokud toto nastane, musí se celý cyklus vývoje opakovat.

Plánování
V rámci plánování se vytváří časový plán vývoje na základě vyspecifikovaných dílčích úkolů. Na jejich základě je projekt rozdělen na iterace, na konci každé se vydávají nové změny. Často se plánují i pravidelné rychlé schůzky vývojového týmu.

Návrh
Klíčem k rychlému a kvalitnímu návrhu je návrh pouze takových funkcionalit, které požaduje zákazník. Důležité je časté refaktorování návrhu a to zpravidla od všech členů týmu.

Implementace
Z návrhu víme, co by měla implementovaná funkcionalita dělat, proto je vhodné pro danou funkcionalitu nejdříve připravit jednotkové testy, na základě kterých je prováděna implementace. Programový kód se píše ve dvojicích, předchází se tak potencálním chybám již v době psaní kódu. Dochází k častému refaktorování programového kódu. Optimalizace kódu přichází na řadu až jako poslední.

Testování
Všechen napsaný kód by měl být pokryt alespoň jednotkovými testy. Objem kódu vynaloženého pro testy bývá zpravidla vyšší, než pro užitečný kód. Pokud je v programu nalezena chyba, je pro ni napsán nový jednotkový test. Extrémní programování rovněž klade velký důraz na jednotné standardy kódu, mezi které patří jednotný vzhled či jednotné jmenné konvence pro pojmenování proměnných, metod, funkcí nebo tříd.

Integrace
Extrémní programování klade důraz na co možná nejvíce automatizovaný proces integrace nových funkcí do zbytku aplikace. Základním předpokladem pro spolehliou automatickou integraci je co možná největší pokrytí přírůstku kódu automatickými testy.

Díky tomu, že extrémní programování vede tradiční činnosti do extrému, dokáže se vývoj softwaru pod vedením této metodiky lépe přizpůsobit měnícím se požadavkům ze strany zákazníka a je tak dodáván software vyšší kvality.

4.1.2.3 Feature driven development

Feature driven development, neboli „vývoj řízený funkcemi“, někdy pouze zkráceně jako *FDD*, je metodika vývoje software, jejíž základem je propojení iterativního přístupu a klasických praktik, jež se osvědčily v jiných průmyslových

odvětvích. Metodika se zaměřuje především na návrh a implementaci, v neposlední řadě pak na neustálé monitorování stavu projektu a kvalitu odvedené práce v průběhu vývoje. Jedním z cílů jsou také rychlé dodávky zákazníkovi.

Základem této metodiky je rozložení celého systému na jednotlivé *funkce*. Ty jsou identifikovány postupnou analýzou implementovaného systému. Na základě této analýzy se identifikované funkce rozplánují do jednotlivých iterací. Zákazník má možnost podobu výsledného produktu ovlivňovat již v průběhu vývoje, což zaručuje dostatečnou flexibilitu metodiky, jež je připravena i na potenciální upřesňování zadání v průběhu vývoje.

Jednotlivé fáze vývoje by se daly shrnout následujícím seznamem. Nutno podotknout, že poslední dvě fáze jsou iterativní:

- vytvoření celkového modelu,
- vypracování seznamu vlastností,
- plánování podle vlastností,
- *návrh podle vlastností*,
- *implementace podle vlastností.*

FDD je zajímavou metodikou, která slučuje prvky jak klasických přistupů (důkladná analýza a prvotní návrh), tak moderních agilních metodik (iterativní přístup k vývoji a reagování na změny).

4.2 Softwarový projekt

Jak již bylo řečeno v úvodu této kapitoly, nebyl jsem jediný, kdo se aktivně podílel na vývoji celé naší služby. Pod záštitou Ing. Jiřího Hunky jsem dostal příležitost podílet se na vedení týmu studentů v rámci předmětů BI-SP1 a BI-SP2. Jedná se o dva předměty ze studijního programu *Informatika* Fakulty informačních technologií ČVUT v Praze.

Sám jsem těmito předměty před několika lety prošel v rámci práce na portálu pro podporu výuky předmětu BI-DBS, organizaci tohoto konkrétního běhu předmětu popisuje ve své bakalářské práci kolega Malec [20]. Nemohu se zde tajit tím, že mi byla jeho práce, co se projektového řízení týče, velikou inspirací. Pozorný čtenář proto možná najde výrazné společné rysy ve způsobu vedení, které zde popisuji já a které ve své práci zmiňuje kolega, neboť pod jeho vedením jsem se tuto disciplínu *de facto* učil.

4.2.1 Organizace předmětu

Práce na projektu byla organizována v rámci dvou předmětů BI-SP1[35] a BI-SP2[36]. Studenti si mohou před začátkem semestru zvolit, jaký softwarový tým si zvolí. Zpravidla je vypsáno několik připravených témat, není však výjimkou ani zformování skupiny studentů, kteří si až poté vlastní téma vymyslí. Obecnou náplň práce studentů v těchto dvou předmětech dobře vystihuje sylabus předmětu BI-SP1:

Studenti si prakticky vyzkouší analýzu, návrh a prototypovou realizaci rozsáhlejšího softwarového systému. Teoretickou podporou jim bude současně probíhající předmět BI-SWI, kde se seznámí s potřebnými technikami a teorií. Studenti budou pracovat ve 4 až 6-ti členných týmech na konkrétním projektu. Vedoucím týmu a projektu bude učitel, který bude pravidelně (formou cvičení) s týmem konzultovat formální i věcnou správnost jejich návrhu. Výsledek práce bude dále rozvíjen a dokončován v rámci předmětu BI-SP2. [21]

V našem případě bylo téma celého projektu známé dopředu, mohli jsme si tedy dovolit nalákat potenciální zájemce na seznam témat, technologií a požadovaných znalostí, o kterých jsme předpokládali, že budou v případě našeho projektu používány a vyžadovány. Pro ilustraci přikládám přesné znění nabídky na účast v našem softwarovém týmu, jež je k vidění na obrázku 4.3.

4.2.2 Organizace vývojového týmu

Počet členů vývojového týmu se v čase měnil. Na počátku běhu předmětu BI-SP1 jsme začínali jako skupina sedmi vývojářů (šest členů z řad studentů a já). Tři členové se po dokončení BI-SP1 rozhodli směřovat svoji energii jinam, na druhou stranu nám na začátku běhu předmětu BI-SP2 přibyl člen nový, pokračovali jsme tedy jako tým dohromady pěti vývojářů.

Během vývoje jsme striktně nedodržovali žádnou z metodik, o kterých píši v předchozí podkapitole 4.1, s některými měl však náš způsob vývoje společné rysy:

Párové programování

Tuto metodu jsme převzali z metodiky *extrémního programování*. Ta spočívá ve vzájemné kontrole napsaného kódu. Vzhledem k distanční formě výuky a faktu, že tým nesdílí jednu kancelář, nemůžeme předpokládat, že si budou programátoři během vývoje koukat přes rameno. Vzájemná kontrola kódu probíhala alespoň na dálku.

Každému úkolu na *Redmine*, jež byl předán *k vývoji*, byl přiřazen *řešitel* a případně[37] i *kontrolór kódu*. Kontrolór poté v prostředí *GitLab* může

[35] LS 2020/21
[36] ZS 2020/21
[37] Pokud byl výstupem úkolu zdrojový kód

Návrh a realizace služby prodeje fotografií pro fotografy

Cílem projektu je návrh a implementace webové služby usnadňující fotografům, kteří pravidelně fotí svatby či jiné velké akce, distribuci jejich fotografií mezi cílové zákazníky. Kromě možnosti stažení také tvorba tiskových materiálů u třetích stran apod.

Aktuální stav projektu:

Projekt na zelené louce.

Přínosy a perspektiva:

- Prosazování vlastních nápadů, nepojede se podle návodu
- Možnost pracovat na projektu, u kterého je velká šance, že se bude reálně používat
- Možnost pokračovat na projektu v rámci bakalářské práce
- **Zkušený vývojář bude připraven konzultovat s Vámi postup, zaškolit, poradit..**

Technologie:

- Vue.js (frontend)
- PHP Symfony (backend) (není nutné)
- PostgreSQL
- Git
- GitLab
- CI
- Redmine/Trello či jiný nástroj pro vedení projektů a bug tracking systém
- Unit testy, testování API (Codeception)

Organizace

Ideální by byla skupinka 3-5 kamarádů, kteří už se znají a umí spolupracovat. Individuálně pak bude domluven termín pravidelných schůzek (60 minut 1x za týden).

Obrázek 4.3: Inzerát lákající studenty zapojit se do týmu vývojářů webové služby pro fotografy

zkontrolovat implementaci v rámci přiřazeného *merge requestu*. Projde-li *merge rqeuest* kontrolou, označí ho kontrolór jako *schválený*.

Testování
Z *extrémního programování* jsme převzali i další praktiku – neustálé testování. Všechny klíčové prvky jsou průběžně pokrývány jednotkovými testy, velký důraz je kladen i na testy ověřující celkovou funkčnost. Výsledky těchto testů jsou pravidelně sdíleny se všemi členy týmu např. formou zprávy o pokrytí kódu testy.

Refaktorování kódu
Každý kód je zpravidla podroben drobnému „učesání“. Nejde ani tak o opravu jeho funkčnosti či zefektivnění, spíše o jeho přepsání tak, aby splňoval určité *osvědčené postupy*, jež není možné ověřovat automaticky či o jeho zjednodušení. Refaktorování vždy provádí zkušenější vývojář

nad kódem vývojáře méně zkušeného. Touto formou je začínajícím vývojářům poskytnuta zpětná vazba a dá se předpokládat, že příště by odevzdaný kód již podobnou refaktoraci nepotřeboval.

Jednotné standardy kódu
Pracuje-li nad jedním kódem více programátorů, měly by se zavést takové postupy, aby od všech vypadal kód podobně. Jedná se o další z principů *extrémního programování* a o jeho praktické dosažení v tomto projektu se budu zmiňovat v dalších kapitolách[38].

Modelování systému
Z *feature driven development* jsme převzali princip prvotního namodelování naší aplikace před její přímou implementací. Ta spočívala především v návrhu prvotního *doménového modelu*, jednoduchého *lo-fi prototypu* uživatelského rozhraní a ujasnění si úloh všech komponent *architektury* aplikace.

Nutno však podotknout, že prvotní návrh *doménového modelu* a *lo-fi prototypu* sloužil studentům především pro procvičení látky probírané v rámci předmětu *BI-SI1*. Pro účely dalšího vývoje byl vypracován např. podrobnější doménový model, jenž je prezentován v kapitole 2 této práce.

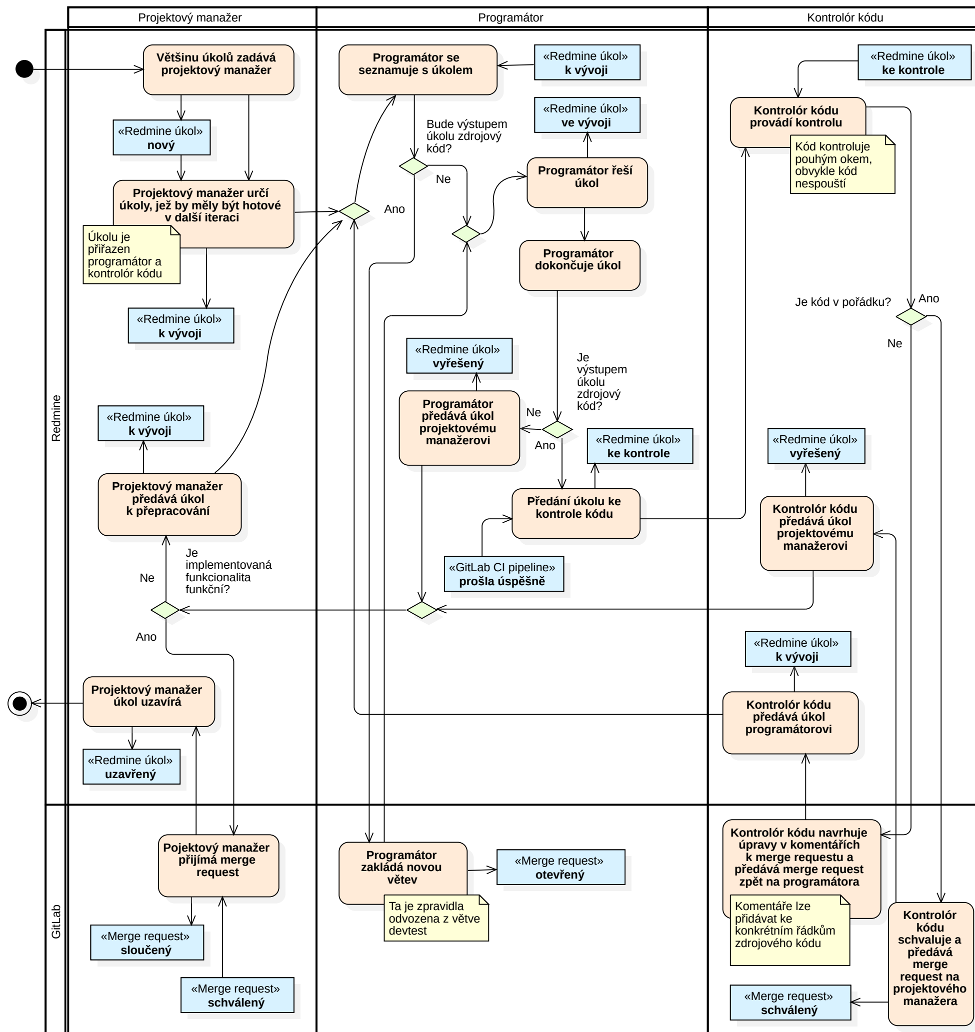
Pokud se na vývoji jakékoli aplikace podílí víc než jeden programátor, je vždy dobré zavést v rámci efektivity vývoje jednotné pracovní postupy pro takové procesy, které se budou v rámci vývoje neustále opakovat a které budou tvořit páteř celého vývoje. V našem případě se jednalo především o definování postupů v rámci systémů *Redmine* a *GitLab*, které mohou být pro nováčka z prvopočátku zmatečné. Tyto pracovní postupy jsou vyneseny na obrázku 4.4.

Jednotlivým členům týmu nebyly přisuzovány žádné konkrétní *role*. Každý se v průběhu vývoje stal na chvíli *architektem*, *designérem*, *vývojářem* nebo *testerem* – záleželo vždy na konkrétním úkolu, kterého se daný člen ujal. Samozřejmě se časem jednotliví členové vyprofilovali spíše do role vývojářů *frontendu*, neboť jim byl bližší jazyk *Javascript*, někteří se spíše ujímali úkolů implementující testy. Jediná role, která byla pevně stanovena, byl *projektový manažer*, jenž celý projekt řídil a jehož povinností jsem se ujal sám.

Členové týmu se setkávali[39] formou pravidelných schůzek každý týden. Na nich se probíralo další směřování projektu, určovaly se úkoly, jež by měly být vypracovány v rámci další iterace a procházely se úkoly vyřešené. Z počátku byla velká část času věnována zaškolení méně zkušených členů týmu do nových technologií a postupů, které se v projektu využívají. Všechny postupy a další tipy byly postupně sepisovány do dokumentace, o které se blíže zmiňuje podkapitola 3.4.1.4.

[38] Podkapitoly 5.1.6 a 5.2.1.4

[39] Kvůli koronavirové pandemii však pouze na dálku



Obrázek 4.4: Diagram popisující pracovní postupy při práci s nástroji *Redmine* a *GitLab*

4.2.3 Hodnocení

Předměty BI-SP1 a BI-SP2 mají předepsaný způsob zakončení formou *klasifikovaného zápočtu*. Forma zakončení tedy nezávisí na složení zkoušky, každý student je ohodnocen za základě jeho práce v semestru. Abych zde mohl popsat systém hodnocení v našem projektu, je nutné nejdříve osvětlit kreditovou dotaci, která se u předmětů BI-SP1 a BI-SP2 liší.

BI-SP1

Předmět samotný má kreditovou dotaci, jež odpovídá 4 ECTS[40] kreditům. Do práce na softwarovém projektu je však zahrnut i zápočet předmětu BI-SI1[41], kreditová dotace tohoto předmětu činí celých 5 ECTS kreditů. Celkem v rámci softwarového týmu studenti bojují dohromady o 9 ECTS kreditů.

BI-SP1

Předmět má kreditovou dotaci 4 ECTS kredity.

Dle [22] odpovídá jeden ECTS kredit časové dotaci 25–30 hodin, které by měl student studiu věnovat. Pokud má tedy student zapsán předmět, za který obdrží celkem 4 ECTS kredity, měl by danému předmětu věnovat přibližně 100–120 hodin svého času.

V hodnocení studentů z našeho softwarového týmu hraje tento systém zásadní roli. Dílčí úkoly, na nichž studenti po celé dva semestry pracovali, byly zaznamenávány do systému *Redmine*. Za každý splněný úkol bylo možné získat určitou sumu *bodů*. Jeden *bod* odpovídá zhruba jedné *hodině*, kterou člen týmu řešením úkolu strávil. Zde je potřeba upozornit na to, že do celkového součtu hodin samozřejmě patří i čas strávený na společných schůzkách.

Jelikož se jedná o *týmový* projekt, body za vyhotovené úkoly nejdou na vrub pouze tomu členovi týmu, jenž daný úkol vyřešil. Místo toho se získané body rozdělí mezi všechny členy týmu. Tento princip má za úkol v jednotlivých členech týmu posílit uvědomění, že se jedná o společné dílo, za které mají odpovědnost společně. Zároveň je tím možné kompenzovat rozdílnou úroveň znalostí jednotlivých členů týmu.

Tento princip však může být potenciálně zneužit takovými členy týmu, kteří by se jen chtěli vézt na vlně odvedené práce ostatních členů. Pro tyto účely bylo zavedeno *interní hodnocení* jednotlivých členů vedoucím týmu. Ten má právo na konci každé iterace přerozdělit body, které tým získal z úkolů z dané iterace tak, aby výsledná bilance takto přerozdělených bodů dávala v součtu nulu. Ve zkratce tedy to, co si některý z členů na bodech přidal, musí být ostatním členům ubráno.

Veškeré získané body byly zaznamenávány do sdílené tabulky 4.5. Jak je z tabulky[42] patrné, na základě získaných bodů je každému studentovi interaktivně vypočítána známka, kterou má právo obdržet. Výsledná známka, jež odpovídá součtu bodů obdržených za úkoly v *Redmine*, se řídí tabulkou 4.1. Předmětu BI-SP1 by se měl student tedy věnovat ideálně $9 \times 25 = 225$ hodin, navazujícímu předmětu BI-SP2 potom pouze $4 \times 25 = 100$ hodin.

[40] European Credit Transfer and Accumulation System

[41] Softwarové inženýrství 1

[42] Zde v podobě, v jaké byla tabulka během poslední iterace předmětu BI-SP2

Začátek týdne	Týden	Iterace	Ideální bodový zisk jednoho člena	Body z úkolů celkem	Michaela Weberová		David Fencl		Jan Cvrček		Tadeáš Pála		Bilance interního hodnocení
					Získané body	Interní hodnocení	Získané body	Interní hodnocení	Získané body	Interní hodnocení	Získané body	Interní hodnocení	
1. července 2020	☀️⛱️💦	1	20,00	64,50	7,50	-8,63	18,00	1,88	19,50	3,38	19,50	3,38	0,00
21. září 2020	1. týden												
28. září 2020	2. týden												
5. října 2020	3. týden												
12. října 2020	4. týden	2	20,00	60,75	17,99	2,81	8,27	-6,92	20,99	5,81	13,49	-1,70	0,00
19. října 2020	5. týden												
26. října 2020	6. týden												
2. listopadu 2020	7. týden	3	20,00	13,50	0,00	-3,38	7,50	4,13	0,00	-3,38	6,00	2,63	0,00
9. listopadu 2020	8. týden												
16. listopadu 2020	9. týden												
23. listopadu 2020	10. týden	4	20,00	49,50	12,38		12,38		12,38		12,38		0,00
30. listopadu 2020	11. týden												
7. prosince 2020	12. týden												
14. prosince 2020	13. týden	5	20,00	79,50	19,88		19,88		19,88		19,88		0,00
21. prosince 2020	🎄🎁❄️												
28. prosince 2020													
4. ledna 2021	Zkouškové												
				Body celkem	**58**		**67**		**73**		**72**		
				Nárok na známku	**E**		**D**		**C**		**C**		

Obrázek 4.5: Tabulka získaných bodů za jednotlivé úkoly v předmětu BI-SP2

Redmine body		
Spodní hranice	Horní hranice	Známka
90	100 a více	A
80	89	B
70	79	C
60	69	D
50	59	E
0 a méně	49	F

Tabulka 4.1: Obdržená známka vzhledem k počtu získaných Redmine bodů v předmětu BI-SP2

4.3 Nástroje

Celý proces projektového řízení lze opřít o celou řadu nástrojů, jež nám s jeho organizací pomohou. Základem sestavy takových nástrojů je v našem případě *Redmine*, o kterém se krátce zmiňuji v podkapitole 4.3.1.

Nástrojem, jenž zastupuje centrální úložiště pro zdrojové kódy, je v našem případě *GitLab*, o něm jsem se však již podrobněji zmiňoval v podkapitole 3.4.1 a zde ho již nebudu rozebírat.

Zásadní je rovněž nástroj pro rychlou a efektivní komunikaci mezi jednotlivými členy týmu. K těmto účelům jsme využívali nástroj *Slack*, jehož přednosti

vyzdvihuji v podkapitole 4.3.2.

Okolnosti, jež se nedaly ovlivnit, nás nutily ke schůzkám na dálku, pro tento účel bylo hojně využíváno nástroje *Google Meet*, o kterém referuji v podkapitole 4.3.3.

4.3.1 Redmine

Jádro organizace projektového řízení tvoří nástroj *Redmine*[23]. Jedná se o open-source projekt pro řízení projektů a *bug tracking system*. V rámci něj je možno vytvářet jednotlivé *projekty*, z nichž každý může mít libovolný počet *podprojektů*. Do jednotlivých projektů lze zvát uživatele, z nichž každý disponuje určitou *rolí*, na základě které lze zpřístupnit jen určité funkce systému.

Klíčovou částí jsou ovšem *úkoly*. Každý z úkolů zpravidla popisuje malou část problému k implementaci. Úkolům lze nastavovat jejich *prioritu*, pomocí které lze posléze vytřídit takové úkoly, jež by se měly implementovat nejdříve. Úkoly se dají drobit na jednotlivé podúkoly. Je zde možné také využít *vztahy* mezi jednotlivými úkoly – můžeme tak např. definovat, že jeden úkol *blokuje* úkol jiný, atd. Úkol, jehož typická podoba je k vidění na obrázku 4.6, obsahuje několik atributů, jež jsou definovatelné správcem projektu. V našem případě jsme využívali zejména následující pole:

- název – název úkolu,
- popis – detailní popis úkol,
- stav – jenž v našem případě může nabývat hodnot *nový*, *k vývoji*, *ve vývoji*, *ke kontrole*, *vyřešený* a *uzavřený*,
- cílová verze - rozlišení, zda se úkol týká *frontendové* či *backendové* části projektu,
- % hotovo – postup řešení úkolu vyjádřený v procentech,
- získané body – kolik bodů bylo přiznáno týmu za úspěšné vyhotovené daného úkolu,
- merge request – URL merge requestu, jenž je spjatý s daným úkolem,
- přiřazeno – člen týmu, od něhož se čeká další aktivita ohledně daného úkolu,
- code review – člen týmu, jež je v daném úkolu zodpovědný za kontrolu kódu.

Připomeňme dále některé méně používané, přesto však velmi užitečné funkcionality systému *Redmine*:

Požadavek #4058 Upravit Přidat čas Sledovat Kopírovat Odstranit

Response přiřazeného formátu by měla vracet i vlastníka přiřazeného formátu « Předchozí | 6/84 | Další »

Přidáno uživatelem Marek Erben před 1 měsíc(ů). Aktualizováno před 1 minuta.

Stav:	Uzavřený	**Začátek:**	19.10.2020
Priorita:	Vysoká	**Uzavřít do:**	
Přiřazeno:	Marek Erben	**% Hotovo:**	100%
Cílová verze:	Backend	**Odhadovaná doba:**	
		Strávený čas:	1.50hod
Body:		**Merge request:**	https://gitlab.jagu.cz/ph...
Získané body:	2.00	**Code review:**	Jan Cvrček

Popis Citovat

Cílem je úprava serializace přiřazeného formátu: metoda `serialize` třídy `AssignedFormatSerializer` tak, aby vracela formát včetně jeho "vlastníka", tedy atributu `user` třídy `Format`.

Po této úpravě bude nutné opravit i dokumentaci API a možná některé testy (nevím jistě), aby korespondovaly s touto úpravou.

Dílčí úkoly Přidat

Související úkoly Přidat

blokuje Photographix – ~~Požadavek #3950~~: Detail sdíleného formátu	Uzavřený	17.07.2020	

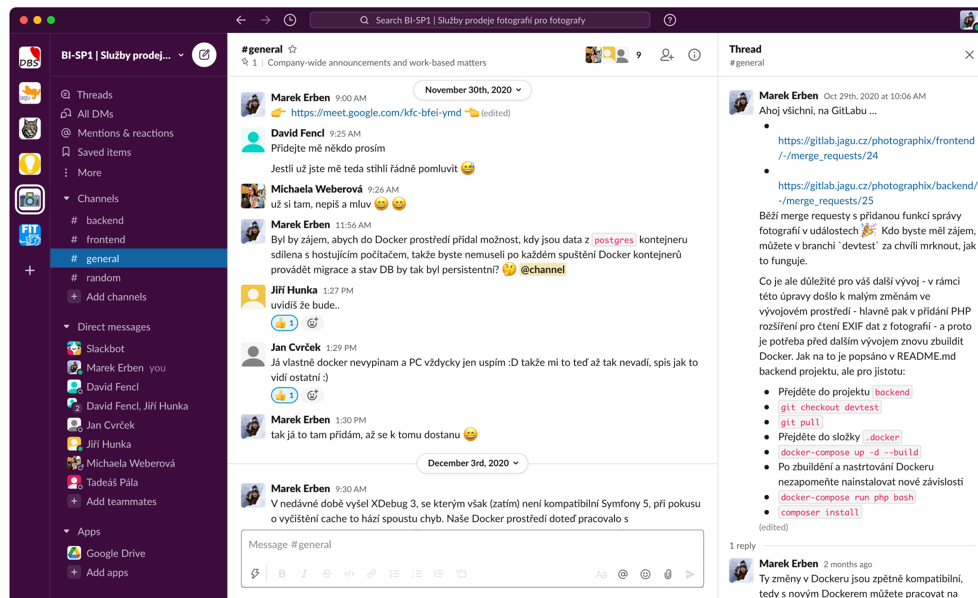
Obrázek 4.6: Ukázka typického úkolu v systému *Redmine*

- přehled aktivit – zvláště užitečný pro projektového manažera, jenž může v tomto přehledu vidět, na čem se v současnosti pracuje,
- strávený čas – přehled stráveného času jednotlivých členů týmu,
- plán – přehled úkolů pro jednotlivé cílové verze, v jednoom pohledu se zde můžeme dozvědět, kolik úkolů je ještě nutno implementovat v rámci přístušné verze.

4.3.2 Slack

Pro instantní komunikaci mezi všemi členy týmu byl využit nástroj *Slack*[24]. Ten dovoluje vytvořit pro skupinu uživatelů pracovní prostor, jenž se nazývá *tým*. V rámci týmu lze vytvářet libovolný počet tématických *kanálů*. Na jednotlivé zprávy pak lze reagovat vytvořením samostatných *vláken*. Možné je samozřejmě také přímá komunikace s jinými členy týmu. *Slack* lze používat prostřednictvím nativní aplikace, či skrze webový prohlížeč.

Velký přínos vidím ve funkcích, které jsou na míru ušité pro komunikaci mezi programátory – zprávy ve *Slacku* podporují bohaté formátování textu včetně neproporciálního písma a podpory pro syntaxe mnoha programovacích jazyků. Při použití bezplatné verze aplikace je však nevýhodou omezená historie, jež činí 10 000 zpráv. Ukázka prostředí aplikace *Slack* je k vidění na obrázku 4.7.



Obrázek 4.7: Ukázka komunikace v aplikaci *Slack*

4.3.3 Google Meet

Období koronavirové pandemie poznamenalo téměř všechny obory lidské činnosti, školství nevyjímaje. Na vysokých školách byla nastolena distanční výuka a všichni členové týmu byli odkázáni ke komunikaci na dálku. Troufám si tvrdit, že obor IT byl touto situací postižen snad nejméně ze všech, i přes to shledávám osobní kontakt s týmem oproti videohovorům jako mnohem efektivnější. Nastolené situaci se však bylo třeba podřídit a bylo nutné najít nástroj, pomocí kterého bychom v rámci týmu mohli pravidelně komunikovat. Požadavky na takový nástroj byly prosté:

- možnost použití v rámci internetového prohlížeče,
- použití zdarma,
- skupinové videohovory,

- možnost videohovor moderovat,
- možnost sdílet obrazovku/konkrétní okno aplikace s ostatními členy,
- možnost omezit kvalitu příchozího videa pro snížení datového přenostu, pokud by účastník využíval ke spojení mobilní připojení.

Abychom co nejvíce omezili škálu využívaných nástrojů na nezbytné minimum, prověřili jsme nejdříve již používaný komunikátor *Slack*. Ten však ve své nezpoplatněné verzi skupinové videohovory nenabízel.

Vzhledem k faktu, že všichni účastníci disponovali účtem u společnosti *Google*, byl jako primární nástroj pro videohovory vybrán *Google Meet* [25]. Ten veškeré výše uvedené funkce splňuje do puntíku.

Výhodou byl také fakt, že za provozováním *Meet* stojí právě *Google*, což do jisté míry zaručuje funkčnost i v době extrémně vysoké poptávky po podobných službách, která byla vyvolána právě probíhající koronavirovou pandemií [26].

Kapitola 5

Implementace

Následující řádky popisují zásadní témata související s implementací naší služby. Vzhledem k rozmyšlené architektuře celé aplikace rozdělím tuto kapitolu na témata související s *backendovou* částí (5.1), jež se stará o manipulaci s daty a obchodní logiku, a na témata spjatá s *frontendovou* částí (5.2), se kterou interaguje cílový uživatel.

5.1 Backend

Jako srdce naší aplikace lze bezesporu označit tu komponentu, kterou zde nazývám *backend*. Jedná se o komponentu sdružující obchodní logiku aplikace, rozhoduje o autentizaci uživatelů, přímo manipuluje s daty v databázi, atd.

Základem této komponenty jsem zvolil PHP framework *Symfony*, ten podrobněji přibližuji v podkapitole 5.1.1.

Komponenta *backend* nemá žádné přívětivé uživatelské rozhraní a ostatní komponenty s ní komunikuji výhradně skrze *REST API*. Jeho návrhu a implementaci se pak věnuje samostatná podkapitola 5.1.2.

Každá webová služba by měla myslet na svoje *zabezpečení*, ani u této komponenty to není jinak. Zabezpečení *backendové* části aplikace se blíže věnuje podkapitola 5.1.3.

Analýza požadavků na naši aplikaci nás zavazuje k uchovávání poměrně velkého množství dat. Ukládat tato data na stejném serveru, na němž samotná aplikace běží, by bylo nerozumné a do budoucna by to představovalo problém. Jak byla otázka úložiště vyřešena? To popisuje podkapitola 5.1.4.

Najít vhodné řešení, jak fotografie ukládat, je jedna věc. Jak tyto fotografie efektivně předávat na stranu klienta je věc druhá. O tomto tématu se krátce zmiňuje podkapitola 5.1.5.

Nemalou část energie při vývoji aplikace jsem věnoval jejímu důkladnému testování. Jaké testy jsem považoval za klíčové pro ověření dostatečné kvality kódu popisuji v podkapitole 5.1.6.

5.1.1 Symfony framework

Symfony [27] je PHP framework vycházející z návrhového vzoru MVC[43]. Jednotlivé části frameworku jsou distribuovány v podobě samostatných, znovupoužitelných komponent. Tyto komponenty lze zpravidla použít samostatně s jakoukoliv PHP aplikací.

Podstatná síla frameworku *Symfony* tkví v možnosti využívat pluginy, v řeči *Symfony* známé jako *bundles*. Dovolte mi zde připomenout některé, které jsem shledal pro naše účely nepostradatelné:

`FOSRestBundle`
: Bundle pro usnadnění implementace REST API. Více se o něm zmiňuji v rámci podkapitoly 5.1.2.2.

`MakerBundle`
: Bundle, jenž poskytuje interaktivní konzolové rozhraní pro snadné generování tříd, jež reprezentují entity přímo mapované na záznamy tabulek relační databáze.

`DoctrineBundle`
: Bundle, který integruje *Doctrine ORM* framework, pomocí kterého můžeme pracovat s databází skrze *ORM*[44] vrstvu.

`DoctrineMigrationsBundle`
: Bundle pro generování a vykonávání databázových migrací. Ty jsou automaticky generovány na základě vytvořených entitních tříd.

`DoctrineFixturesBundle`
: Bundle pro programatickou tvorbu testovacích dat.

`OneupFlysystemBundle`
: Bundle, jenž integruje knihovnu `League\Flysystem` [28] pro abstrahování souborového systému. Dále ho zmiňuji v podkapitole 5.1.4.

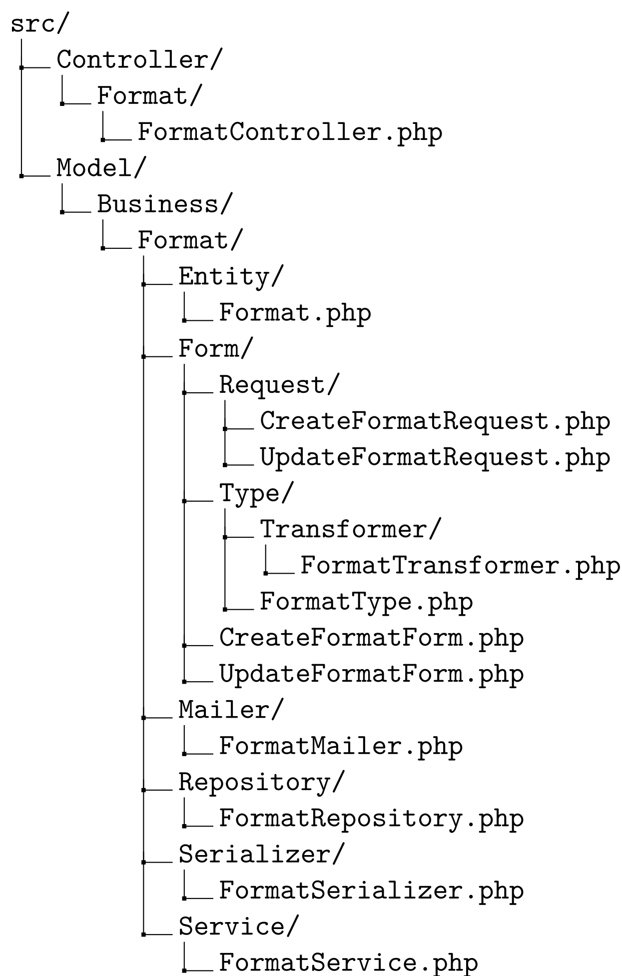
`LiipImagineBundle`
: Bundle pro efektivní práci s obrázky, v našem případě jeho přínos oceníme v souvislosti s generováním jejich náhledů. Blíže jeho použití popisuji v podkapitle 5.1.5. Příkladně funguje s výše zmíněným *OneupFlysystemBundle*.

`NelmioApiDocBundle`
: Bundle pro přímé generování API dokumentace. Více jeho přínos popisuji v podkapitole 5.1.2.3.

[43] Model-View-Controller

[44] Object-relational mapping

Jednou z hlavních rysů jakéhokoliv frameworku je zpravidla daná adresářová struktura, jíž je dobré během vývoje dodržovat. Tento přístup má nespornou výhodu v orientaci programátora v projektu. Symfony není výjimkou – v základu nás pobízí sdružovat např. veškeré kontrolery v adresáři `src/Controller`, všechny entitní třídy v adresáři `src/Entity`, atd. Já jsem pro naše účely shledal jako vhodnější variantu organizaci tříd dle jejich *business* zaměření. Tedy např. všechny třídy, jež nějak souvisejí s *formáty*, shlukuji v rámci adresáře `src/Model/Business/Format`. Ukázková jednotná adresářová struktura, kterou jsem pro náš projekt upravil, je k vidění na obrázku 5.1, kde je struktura demonstrována pro *formáty*.



Obrázek 5.1: Ukázka adresářové struktury projektu *backendové* části aplikace implementované s pomocí frameworku *Symfony*

Následuje detailnější popis jednotlivých souborů v rámci výše zmíněné adresářové struktury:

`FormatController.php`
Kontroler sdružující přístupové body API pro *business* doménu *formátů.*

`Format.php`
Entitní třída reprezentující jeden záznam v relační tabulce formátů.

`CreateFormatRequest.php`
DTO třída, jež po úspěšném zpracování formuláře pro *vytvoření* formátu obsahuje hodnoty odeslané v rámci těla HTTP požadavku.

`UpdateFormatRequest.php`
Totéž jako v předchozím bodě, avšak pro *úpravu* formátu.

`FormatTransformer.php`
Třída zodpovědná za instancování třídy `Format` z jejího unikátního identifikátoru a naopak.

`FormatType.php`
Třída reprezentující znovupoužitelný formulář pro manipulaci s formátem, jenž může být využit v jiných formulářích vedle ostatních formulářových typů.

`CreateFormatForm.php`
Definuje strukturu formuláře pro *vytvoření* formátu, jeho atributy a k nim odpovídající formulářové typy.

`UpdateFormatForm.php`
Totéž jako v předchozím bodě, avšak pro *úpravu* formátu.

`FormatMailer.php`
Třída zodpovědná za zasílání notifikačních e-mailů, které souvisejí s formáty.

`FormatRepository.php`
Třída zodpovědná za získávání dat z databáze skrze *DQL* dotazy.

`FormatSerializer.php`
Třída obsahující metody pro serializaci entity, aby ji bylo možné bezpečně poslat v rámci HTTP odpovědi jako JSON objekt.

`FormatService.php`
Srdce obchodní logiky, vytváří a upravuje nové entity, sdružuje serializovací třídy a repozitáře.

5.1.2 REST API

Na následujících řádcích představuji implementaci REST API v rámci *backendové* části naší aplikace. V rámci podkapitoly 5.1.2.1 rozebírám architektonický styl REST teoreticky, ve kterém definuji základní pojmy a osvědčené postupy, které je záhodno během implementace REST API dodržovat.

O začlenění REST API v rámci naší aplikace se podrobněji rozepisuji v podkapitole 5.1.2.2.

REST API jakékoliv služby by bylo zcela k ničemu, pokud by k němu neexistovala odpovídající dokumentace, o ní pojednává samostatná podkapitola 5.1.2.3.

5.1.2.1 Architektura rozhraní REST

Jak již bylo nastíněno v podkapitole 2.2, klíčovou technologií pro komunikaci klientů s naším aplikačním serverem je REST[45] API. Koncept REST představil ve své disertační práci [29] Roy Fielding – zároveň jeden ze spoluautorů protokolu HTTP[46]. Z toho důvodu nepřekvapí, že REST má s HTTP mnoho společného a silně závisí na jeho standardech.

REST definuje několik hlavních omezení, které je nutno dodržovat. Pokud jsou všechna z nich dodržena, může se takové rozhraní, jež tato omezení implementuje, označovat jako *RESTful*:

Klient-server architektura
: Rozdělení aplikace na samostatný server a klienta umožňuje vyšší rozšiřitelnost a přenositelnost aplikace. Neméně podstatnou výhodou je možnost lepší škálovatelnosti jednotlivých serverových komponent.

Bezstavovost
: Každý požadavek od klienta na server musí obsahovat všechny informace potřebné k vyhodnocení požadavku a nesmí jakkoli využívat uloženého kontextu na serveru. Stav relace je proto zcela ponechán na klientovi.

Využití mezipaměti
: Veškeré odpovědi ze serveru by měly být odpovídajícími technikami označeny jako vhodné/nevhodné pro cachování na straně klienta.

Jednotné rozhraní
: Pokud mezi všemi komponentami systému zavedeme jednotné rozhraní, zvýší se tím přehlednost komunikace mezi nimi. REST přichází se čtyřmi základními omezeními na rozhraní:

  Identifikace zdrojů
  : Jednotlivé zdroje jsou identifikovány pomocí jejich URI[47].

[45] Representational State Transfer
[46] Hypertext Transfer Protocol
[47] Uniform Resource Identifier

Manipulace se zdroji skrze jejich reprezentace
Klient by měl být schopen od serveru získat veškerá data potřebná pro manipulaci s daným zdrojem, tedy aby na jejich základě mohl vyslat validní požadavek pro jejich úpravu. Zároveň klient nepracuje přímo se zdroji, ale pouze s jejich reprezentací – typicky se serializovnou podobou objektu v různých formátech (JSON, XML, …).

Samopopisné požadavky a odpovědi
Každý požadavek na server, stejně jako odpověď ze serveru by měla být samopopisná, tedy aby server, respektive klient mohl jednotlivé zprávy spolehlivě rozkódovat. K tomu se zpravidla používají hlavičky `Content-type`, apod.

HATEOAS
Klient se serverem komunikuje skrze dynamicky poskytované *hypermédium*. Toto hypermédium v sobě obsahuje odkazy na další dostupné zdroje. Tyto linky může klient použít při dalším požadavku, tím definuje svůj stav v rámci aplikace. Typickým příkladem je obsah hlavičky `Location` v rámci odpovědi serveru po vyhodnocení `POST` požadavku na vytvoření nového zdroje či odkazy na předchozí/další stránku ve stránkovaném výpisu zdrojů.

Vrstevnatost
Architektura aplikace může být složena z více vrstev. Každá vrstva by neměla znát žádné implememtační detaily vrstvy jiné. Mezi klientem a koncovým serverem může být spousta zprostředkujících serverů, které mohou zlepšit odezvu systému rozložením zátěže mezi více instancí, či efektivním využitím cachovaného obsahu.

Jak již bylo řečeno, REST definuje jednotný přístup k cílovým zdrojům jednak pomocí jejich URI a jednak pomocí HTTP metody odpovídající operaci CRUD[48]. Kvůli požadavku na nutnost efektivně cachovat zdroje na straně klienta jsou tyto metody rozděleny na základě jejich *datové bezpečnosti*:

Datově bezpečné metody
Datově bezpečná metoda nemění stav cílového zdroje. Obvykle u operací „pouze pro čtení“. Klienti mohou odpovědi ukládat do mezipaměti a mezipaměť libovolně aktualizovat.

Datově nebezpečné metody
Dotazovaný zdroj se vlivem volání *datově nebezpečné operace* může změnit.

Na základě efektu opakovaného volání určité metody nad jedním zdrojem se operace rozlišují na:

[48]Create, Read, Update, Delete

Idempotentní metody
Opakované volání *idempotentní operace* nad jedním zdrojem má vždy stejný efekt. Typicky se jedná o *výpis*, či o *úpravu* atributu zdroje.

Neidempotentní metody
Opakované volání *neidempotentní operace* nad určitým zdrojem může mít rozdílný efekt. Jako příklad se nabízí volání metody pro *vytvoření* nového zdroje.

Vybaveni základními teoretickými poznatky o REST můžeme připomenout, jak jsou v REST definovány jednotlivé CRUD operace:

HTTP metoda `POST`
Metoda `POST` odpovídá v jazyce CRUD metodě *create*. Slouží tedy k vytvoření nového zdroje. Obsah nově vytvořeného zdroje se posílá v serializované podobě v *těle* požadavku. Po úspěšném dokončení požadavku je dobrým zvykem v rámci odpovědi posílat nově vytvořený zdroj. To lze dvěma způsoby:

- odpověď obsahuje v hlavičce `Location` *odkaz* na nově vytvořený zdroj,
- odpověď obsahuje v těle požadavku celý *serializovaný objekt* nově vytvořeného zdroje.

Nedílnou součástí každé odpovědi je *stavový HTTP kód*. Ten je v případě úspěšného vykonání `POST` operace *201 Created*. V případě, že tělo požadavku obsahuje chyby (např. chybí některý z povinných atributů), vysílá server odpověď se stavovým kódem *400 Bad Request*. Pokud není možné požadovaný zdroj vytvořit z důvodu narušení logiky aplikace, odpovídá server stavovým kódem *409 Conflict*.

Metoda `POST` je z hlediska datové bezpečnosti *nebezpečná*, z hlediska idempotence pak *neidempotentní*.

HTTP metoda `GET`
Metoda `GET` z hlediska CRUD odpovídá metodě *read*. Ta slouží k *získání* informací o požadovaném zdroji. Serializované zdroje jsou součástí těla odpovědi.

K uspokojení požadavku na efektivní cachování lze v rámci `GET` metody použít některé standartní mechanismy definované ve standardu HTTP. Typickými zástupci těchto mechanismů mohou být:

- hlavička `Cache-Control` uvnitř odpovědi serveru, která definuje, zda a po jakou dobu může klient daný obsah cachovat,

- hlavička `ETag` uvnitř odpovědi serveru, jež obsahuje heš serializovaného zdroje, o který bylo daným požadavkem zažádáno. Uchováním tohoto heše na straně klienta lze další požadavky na stejný zdroj provádět s vyplněnou hlavičkou `If-None-Match`. Server v takovém případě vrátí kýžený obsah pouze v případě, že se hodnota ETag na straně serveru a na straně klienta liší. Pokud server data nevrátí, použije klient data ze své cache,
- hlavička `Last-Modified` v odpovědi serveru obsahuje časovou značku poslední úpravy požadovaného zdroje. Klient vysílá požadavky s vyplněnou hlavičkou `If-Modified-Since` s poslední známou časovou značkou poslední úpravy zdroje. Server vrátí data pouze v případě, že se od posledního požadavku data změnila. V opačném případě klient opět využije cachovaná data.

Typickým stavovým kódem odpovědi na `GET` požadavek je v případě úspěšného vyhodnocení žádosti *200 OK*. Metoda `GET` je z hlediska datové bezpečnosti *bezpečná* a navíc je i *idempotentní*.

HTTP metoda `PUT`

Metoda `PUT` odpovídá v kontextu CRUD metodě *update*. Součástí těla požadavku posílaného metodou `PUT` jsou data, pomocí kterých má být dotazovaný zdroj upraven. Metoda `PUT` nahrazuje *všechna* data v dotazovaném zdroji. V případě, že dotazovaný zdroj zatím neexistuje, měl by ho server implementující tuto metodu vytvořit. V případě, kdy dotazovaný zdroj *upravuji*, je nutné v rámci těla dotazu dodat *všechny* požadované atributy.

Z hlediska datové bezpečnosti se jedná o *nebezpečnou* metodu, avšak z phledu idempotence se jedná o metodu *idempotentní*.

V případě *vytvoření* nového zdroje po úspěšném provedení metody `PUT` je dobrým zvykem vracet v rámci odpovědi statový kód *201 Created*. Pokud provedením `PUT` metody dojde k vytvoření nového zdroje, je nutno vrátit stavový kód *204 No Content*.

HTTP metoda `PATCH`

Metoda `PATCH` je v jistém slova smyslu podobná předchozí metodě `PUT`. Metoda `PATCH` však nedovoluje vytvořit dotazovaný zdroj, pokud neexistuje. Součástí těla dotazu této metody jsou pouze takové atributy, které mají být upraveny. Atributy zdroje, které nejsou v rámci těla dotazu zníněny, zůstávají beze změn.

Z hlediska datové bezpečnosti se jedná o *nebezpečnou* metodu, navíc je to i metoda *neidempotentní*.

Stavový kód indikující úspěšné provedení této metody by měl být *204 No Content*.

HTTP metoda `DELETE`

A konečně metoda `DELETE` odpovídá stejnojmenné operaci v kontextu CRUD. Jejím úkolem je *odstranit* dotazovaný zdroj.

V případě úspěšného provedení dotazu je v rámci odpovědi vrácen stavový kód *204 No Content*, případně *200 OK*. V takovém případě je ovšem doporučeno vrátit v rámci těla odpovědi i serializovanou podobu právě odstraněného zdroje.

5.1.2.2 Implementace REST API

Veškeré teoretické poznaty o architektuře REST API, které byly probrány v podkapitole 5.1.2.1, jsem převedl i do samotné implementace REST API v rámci *backendové* části aplikace. Pro snadnou implementaci REST API v rámci Symfony aplikace byl využit *bundle* [30] přímo určený pro tento účel.

Tento *bundle* poskytuje programové rozhraní mimo jiné pro získání užitečných dat (HTTP hlavičky, HTTP body, ...) z příchozích požadavků od klienta či pro sestavování odpovědí[49], které jsou klientovi posílány zpět.

Jednotlivé přístupové body REST API jsou v rámci Symfony aplikací reprezentovány konkrétními metodami v rámci kontrolerů. Aby naše aplikace mohla rozhodnout, která metoda se po zaslání konkrétního HTTP požadavku zavolá, je potřeba toto specifikovat pomocí speciálních *anotací*. Každá z anotací odpovídá dané HTTP metodě. Pro obsluhu `GET` požadavku s URL `/api/users/1` je nutné anotovat konkrétní metodu kontroleru pomocí následující anotace:

```php
<?php
/**
 * @Rest\Get(
 *     "/api/users/{userId}",
 *     requirements={"userId"="\d+"},
 *     name="user_user_get"
 * )
 */
public function getUser(int $userId): View { /* ... */ }
```

Přístupové body REST API, které vracejí *kolekci* záznamů mohou teoreticky v odpovědích na HTTP požadavky zasílat velké množství dat. Pro implementaci klienta, jenž by naše API konzumoval, může být výhodné počet záznamů v dané kolekci omezit. K tomu se využívají nejčastěji tři mechanismy:

Stránkování

Tento způsob omezování počtu záznamů v rámci odpovědi na jeden dotaz si můžeme představit jako listování knihou, v rámci kterého můžeme

[49] V našem případě serializovaných ve formátu JSON

definovat *kolik* záznamů očekáváme na *konkrétní* „stránce“. Součástí řetězce dotazu jsou doplňkové parametry (`page`, `perPage`), které nám kýženou podmnožinu záznamů definují.

URL HTTP GET požadavku, který požaduje výpis konkrétní podmnožiny uživatelů, pak může s nastaveným stránkováním vypadat následovně:

```
/api/users?page=3&perPage=10
```

Backendová část naší aplikace podporuje stránkování u všech přístupových bodů, které vrací *kolekci* záznamů. Pro rychlé a efektivní stránkování je nutná jeho implementace na úrovni dotazů do databáze.

Filtrování

Počet záznamů *kolekce* lze efektivně omezit vhodně implementovaným filtrováním. To umožňuje vyhledávat v záznamech podle předem definovanách atributů.

Pro lepší filtrování lze pracovat s *operátory* a *datovými typy*. Ty nám dovolují vytvářet komplexnější dotazy. Operátory, se kterými lze provádět filtrování jsou:

- `eq` – Ekvivalent operátoru `=` v SQL,
- `like` – Ekvivalent operátoru `LIKE` v SQL,
- `gte` – Ekvivalent operátoru `>=` v SQL,
- `lte` – Ekvivalent operátoru `<=` v SQL.

Datové typy jsou pak k dispozici následující:

- `string` – hodnota filtru je explicitně přetypována na `string`,
- `number` – hodnota filtru je explicitně přetypována na `float`,
- `date` – z hodnoty filtru je vytvořena instance třídy `Carbon` [31] pro snažší manipulaci s datumem a časem.

Pro lepší představu ukažme praktický příklad. HTTP požadavek s následující URL vybere takové papíry, jejichž cena je 150–200 Kč/m^2 a jejichž název obsahuje podřetězec „pap“:

```
/api/papers
    ?filters[p.price][0][value]=150
    &filters[p.price][0][operator]=gte
    &filters[p.price][0][dataType]=number
    &filters[p.price][1][value]=200
    &filters[p.price][1][operator]=lte
```

```
&filters[p.price][1][dataType]=number
&filters[p.name][0][value]=pap
&filters[p.name][0][operator]=like
&filters[p.name][0][dataType]=string
```

Filtrování podle předem definovaných atributů podporují všechny přístupové body, které vracejí *kolekci* záznamů.

Řezení

Pro lepší orientaci v záznamech je vhodné poskytnout klientovi možnost jejich získání v předem definovaném pořadí. Jednotlivé záznamy lze řadit podle stejných atributů, podle kterých lze provádět *filtrování*. Řadit lze samozřejmě podle více atributů najednou a to sestupně i vzestupně.

HTTP požadavek s následující URL vrátí informace o papírech, které budou seřazeny podle ceny sestupně a jednotlivé skupiny papírů se stejnou cenou budou druhotně řazeny podle názvu sestupně:

```
https://example.com/api/papers
    ?sort[p.price]=asc
    &sort[p.name]=desc
```

Filtrování, stránkování a řazení je v rámci URL definováno parametry řetězce dotazu[50]. S tím, jaký přístupový bod podporuje konkrétní *query* parametry, nám pomáhá opět již dříve zmíněný *bundle* [30] pro snažší integraci REST API v rámci Symfony aplikace. Jednotlivé *query* parametry je možné definovat opět pomocí *anotací* např. následujícím způsobem:

```php
<?php
/**
 * @Rest\QueryParam(name="page", default=1, requirements="\d+")
 * @Rest\QueryParam(name="perPage", default=100, requirements="\d+")
 * @Rest\QueryParam(name="filters", map=true, default={})
 * @Rest\QueryParam(name="sort", map=true, default={})
 */
public function getPapers(): View { /* ... */ }
```

5.1.2.3 Dokumentace REST API

Sebelepší implementace (nejen) REST API by byla naprosto k ničemu, kdyby programátor neposkytl návod, jak takové API používat. Samozřejmě se můžeme opřít o základní principy architektury REST, s tou však nemusí být každý programátor do podrobna seznámen a proto je při nejmenším nutné poskytnout k API i odpovídající dokumentaci.

[50] Tzv. *query*

Ta by měla poskytovat seznam všech přístupových bodů, textový popis jednotlivých metod a popis jejich parametrů. Nezbytnou součástí by pro každý přístupový bod měl být výčet *všech* HTTP stavových kódů, které mohou nastat. U požadavků prováděných HTTP metodou `GET` by měla dokumentace obsahovat přesnou strukturu těla *odpovědi*. Naopak u požadavků prováděných HTTP metodou typu `POST`, `PUT` či `PATCH` by měla být popsána přesná struktura těla *požadavku*.

Jedním ze způsobů, jak takovou dokumentaci psát, je její manuální specifikace pomocí některého ze standardních nástojů k tomu určených. V dnešní době je takovým nástrojem např. *Swagger*[32]. *Swagger* ve své podstatě implementuje specifikaci *OpenAPI*, jakýsi standard, který popisuje, jak by se správně mělo navrhovat webové REST API. Nástroj *Swagger* umožňuje REST API navrhnout a hlavně i následně *vizualizovat*.

Při letmém zamyšlení nad tím, jak takovou dokumentaci psát a hlavně *udržovat*, vyvstávají některé otázky ohledně *správnosti* výsledné dokumentace. Kdo mi zaručí, že ručně napsaná dokumentace opravdu přesně odpovídá implementaci samotného API? Z počátku vývoje to nemusí být problém, s dodatečnými úpravami kódu hrozí, že se bude API chovat s každou úpravou trochu jinak a pomyslné nůžky mezi podobou dokumentace a reálnou implementací API se budou stále více rozevírat, až se stane dokumentace pro programátora prakticky nepoužitelnou.

Aby se co nejvíce eliminovaly chyby mezi implementací API a dokumentací, vyvstala potřeba kýženou dokumentaci generovat přímo z kódu. Ideálně do nějakého standardního formátu, jenž by bylo možné následně vizualizovat pomocí např. výše zmíněného nástroje *Swagger*. K našemu štěstí existuje *bundle*[33], který dokáže spolupracovat s *bundle* pro tvorbu REST API a který dokáže danou dokumentaci přímo vizualizovat či vygenerovat ve formátu JSON.

Ukázka vygenerované dokumentace se zobrazeným detailem přístupového bodu pro vytvoření nového papíru je k vidění na obrázku 5.2.

Možnost vizualizace dokumentace není jediným přínosem využití zmíněného *bundle* pro generování dokumentace. Výstup vygenerované dokumentace lze reprezentovat i ve formátu JSON. Takto vygenerovaný soubor podporuje celá řada klientů, které mohou po importu souboru sloužit k ozkoušení a komplexnímu testování daného API. Za všechny klienty si zde dovolím zmínit nástroj *Postman*[34], jenž mi během vývoje posloužil k testování a ladění REST API naší služby. Po nahrání specifikace naší dokumentace do nástroje *Postman* vznikne v rámci něj nová *kolekce* HTTP požadavků. Ty již mají předvyplněny HTTP metody, URL, parametry a těla požadavků podle struktury uvedené v dokumentaci. Ukázka práce s nástrojem *Postman* s importovanou API dokumentací je k nahlédnutí na obrázku 5.3.

Generování specifikace API je nasazeno na testovacím serveru *backendové*

Paper

GET /api/papers

POST /api/papers

Parameters Try it out

No parameters

Request body required application/json

Example Value | Schema

```
{
  "name": "FomeiJet PRO Gloss 265",
  "price": 850.5,
  "weight": 265,
  "sheet_longer_side_length": 297,
  "sheet_shorter_side_length": 210,
  "roll_width": null,
  "finish": "GLOSS",
  "texture": "SMOOTH",
  "paper_type": "SHEET"
}
```

Responses

Code	Description	Links
201	Paper successfully created Headers: Name: Location — Description: URL of newly created paper — Type: string	No links
400	Paper could not be created	No links
403	Access denied, ROLE_PRINTER required	No links

GET /api/papers/{paperId}

DELETE /api/papers/{paperId}

PATCH /api/papers/{paperId}

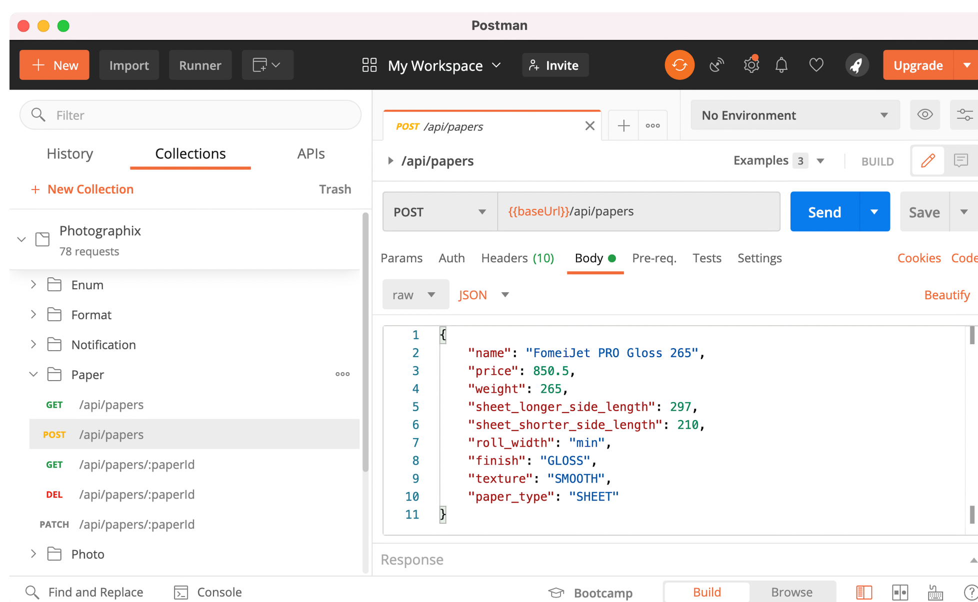
Obrázek 5.2: Ukázka vygenerované dokumentace REST API naší služby

části aplikace jak pro účely přímé vizualizace[51] API s pomocí nástroje *Swagger*, tak i pro vygenerování specifikace ve formátu JSON pro přímé vygenerování klientů konzumující dané API[52].

5.1.3 Zabezpečení

Jelikož je REST API vystaveno na internetu, má k němu přístup prakticky každý. Jelikož naše služba pracuje s citlivými daty více uživatelů, je nutno tato data chránit tak, aby se k nim byl odepřen přístup neoprávněným uživatelům.

[51] https://photographix.jagu.cz/docs
[52] https://photographix.jagu.cz/docs.json



Obrázek 5.3: Ukázka práce s nástrojem *Postman*

5.1.3.1 Autentizace pomocí API tokenu

Základem zabezpečení API je v našem případě *API token*. Jedná se o unikátní heš, který je každému uživateli vygenerován s každým novým přihlášením. Token má platnost jednu hodinu od poslední manipulace s ním. S každým novým požadavkem na server s daným tokenem je tedy jeho životnost nastavena znovu na jednu hodinu.

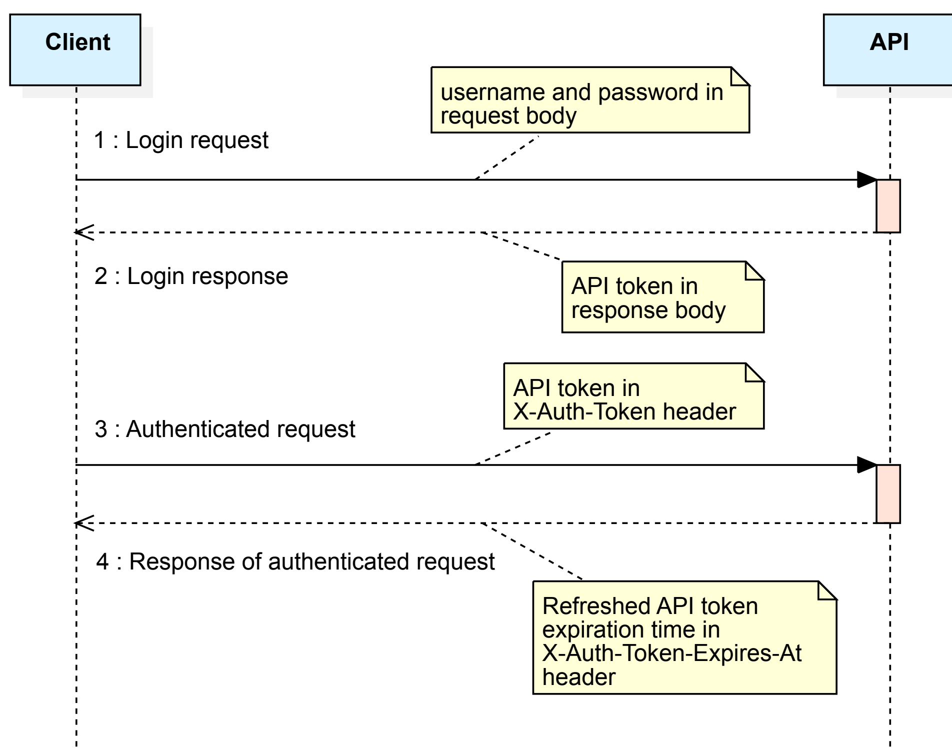
Klientovi je token zaslán v rámci odpovědi na požadavek přihlášení do systému pomocí uživatelského jména a hesla.

Backend dále očekává předávání tokenu v rámci HTTP požadavkuů ve formě vlastní hlavičky `X-Auth-Token`.

Pokud byl požadavek na server zaslán s platným API tokenem ve výše zmíněné hlavice `X-Auth-Token`, je v rámci odpovědi vždy zaslána hlavička `X-Auth-Token-Expires-At` s aktualizovanou časovou značkou expirace tokenu.

Tento proces je graficky znázorněn na obrázku 5.4.

Pokud by časem vyvstala potřeba provádět autentizaci uživatelů i jiným způsobem, není v případě Symfony frameworku nic jednoduššího, než implementovat novou třídu rozšiřující `AbstractGuardAuthenticator`.



Obrázek 5.4: Sekvenční diagram zachycující způsob autentizace pomocí API tokenu

5.1.3.2 Autorizace přístupových bodů

Jednotlivé přístupové body lze díky speciálním *anotacím* zabezpečit tak, že k nim mohou přistoupit pouze uživatelé s danou rolí. Na příkladu metody pro získání detailu papíru 5.1 je na řádku 9 ukázáno použití *anotace* `@IsGranted`, jež zaručí, že danou metodu smějí volat pouze uživatele disponující rolí *tiskárna*. V případě pokusu o přistup k metodě neoprávněným uživatelem je zaslána odpověď se stavovým kódem *403 Forbidden*.

Další úrovní zabezpečení je kontrola, zda se užvatel neprávem nedotazuje na data jiných uživatelů se stejnou rolí. Je kupříkladu nemyslitelné, aby uživatel s rolí *tiskárna* mohl manipulovat s papíry *jiného* uživatele s rolí *tiskárna*. Ukázka kódu 5.1 na řádku 21–23 zachycuje kontrolu, zda příhlášený uživatel (`$this->getUser()`) je ten samý uživatel, jenž vytvořil požadovaný papír (`$paper->getUser()`). Pokud se v obou případech nejedná o stejného uživatele, je opět zaslána odpověď obsahující stavový kód *403 Forbidden*.

```php
<?php

/**
 * @Rest\Get(
 *   "/api/papers/{paperId}",
 *   requirements={"paperId"="\d+"},
 *   name="paper_paper_get"
 * )
 * @IsGranted(Role::PRINTER)
 *
 * @param int $paperId
 * @return View
 */
public function getPaper(int $paperId): View
{
    $paper = $this->paperService->getById($paperId);
    if ($paper === null) {
        return $this->sendNotFound();
    }

    if ($this->getUser() !== $paper->getUser()) {
        return $this->sendAccessDenied();
    }

    return View::create($this->paperService->serialize($paper));
}
```

Ukázka kódu 5.1: Ukázka metody pro získání detailu papíru

5.1.3.3 Autorizace fotografií

Ošetření přístupu k fotografiím vyžaduje poněkud jiný přístup. Zabezpečení přístupových bodů předpokládá, že součástí HTTP požadavku je i API klíč, pomocí kterého se uživatel identifikuje a na základě kterého dovede aplikace posoudit, zda má, či nemá k požadovanému zdroji přístup.

Požadavky na stažení fotografie však s sebou informaci o právě přihlášeném uživateli nést nemohou. Naštěstí existuje standardní mechanismus, který dokáže podobné zdroje zabezpečit. Jedná se o tzv. *signed URLs*, což jsou klasické URL, které navíc disponují dvěma *query* parametry:

- `expires` – časová značka expirace odkazu,
- `signature` – podpis, pomocí něhož může server ověřit, že se klient nesnaží přistoupit k chráněnému zdroji se změněnou hodnotou atributu

expires.

K podepisování URL byla využita knihovna [35] vhodná pro tyto účely.

5.1.4 Úložiště fotografií

Z analýzy požadavků na naší službu vychází potřeba ukládat relativně velký objem dat v podobě fotografií. Každý z uživatelů může nahrát velké množství fotografií, z nichž každou je nutné[53] uchovávat ve vysokém rozlišení. Každá z fotografií může být v průběhu své životnost několikrát naškálována na menší rozlišení[54].

Již v kapitole 2.2 bylo nastíněno řešení využít jako perzistentní úložiště dat některou službu třetí strany, v našem případě *Amazon S3*. *AWS*[55], jehož je *Amazon S3* součástí, nabízí přímo *SDK* pro PHP, pomocí kterého je manipulace se soubory uloženými v cloudu z pohledu libovolné PHP aplikace hračka.

Bylo by však krátkozraké „napojovat“ se s naší aplikací na *Amazon S3* napřímo. Pokud bychom se totiž v budoucnu rozhodli již dále nevyužívat *Amazon S3* a místo něj zvolit alternativní službu[56], bylo by nutné všude, kde se v kódu pracuje s *SDK* od *AWS*, nahradit volání API tohoto *SDK* nějakými novými metodami. To by mohlo být značně komplikované a velmi negativně by to mohlo ovlivnit budoucí rozvoj aplikace.

Elegantním řešením tohoto problému je abstrahovat souborový systém do samostatné vrstvy. Jde o ukázkový příklad využití návrhového vzoru *Adaptér*. Navrhneme *rozhraní* disponující základními metodami pro manipulaci se soubory na běžném souborovém systému. Mezo takové metody patří minimálně:

- `write()` – pro zapsání souboru do daného adresáře na souborovém systému,
- `read()` – pro přečtení souboru uloženého na souborovém systému,
- `delete()` – pro smazání souboru uloženého na souborovém systému,
- `copy()` – pro vytvoření kopie daného souboru,
- a jiné.

Pro každou službu třetí strany, kterou bychom teoreticky chtěli využít jako poskytovatele úložiště, bychom potom implementovali novou třídu implementující výše popsané rozhraní. To, jakou konkrétně instanci by naše aplikace v danou chvíli využívala, by už záleželo na konfiguraci aplikace.

[53] Kvůli možnosti nabízet fotografie cílovým zákazníkům v původním rozlišení
[54] Generování náhledů, příprava nazvětšovaných fotografií k tisku, ...
[55] Amazon Web Services
[56] *Azure File Storage*, *Amazon Cloud Drive*, vlastní FTP server, ...

Použití univerzálního adaptéru nám teoreticky dovoluje nastavit každému uživateli *jiného* poskytovatele úložiště. To byl mimochodem jeden z poznatků jednoho z fotografů během sběru požadavků z podkapitoly 1.1.1.

K našemu štěstí však nebylo nutné tento adaptér implementovat „od nuly“. Přímo pro Symfony existuje *bundle* [36], jenž Symfony aplikacím obdobný adaptér pro práci se souborovým systémem zpřístupňuje. Ten umí v základu pracovat s celou řadou poskytovatelů, mezi nimi i s *Amazon S3*.

Využití výše zmíněného adaptéru má i další výhody. Jen s pomocí rozdílné konfigurace aplikace pro různá prostředí[57] můžeme přepínat mezi dvěma instancemi adaptéru:

- `amazon_filesystem` – adaptér napojený na službu *Amazon S3*, určen pro produkční prostředí,
- `local_filesystem` – lokální adaptér, veškeré soubory jsou ukládány lokálně, tento adaptér je určen pro vývojové a testovací prostředí.

5.1.5 Generování náhledů fotografií

V API naší aplikace se nachází několik přístupových bodů, jež po vyslání HTTP požadavku vrátí binární soubor požadovaného obrázku. Originální zdrojový soubor obrázku bývá zpravidla – co se rozlišení týče – veliký. Webový *frontend* naší aplikace však prakticky nikdy nepotřebuje tyto soubory získat v plném rozlišení, místo toho *frontend* tyto fotografie zpravidla zobrazuje jen jako malé náhledy.

S ohledem na vylepšení celkové odezvy systému je nezbytné, aby fotografie na požadovaný rozměr škáloval samotný *backend*, přičemž se zde uplatňuje princip tzv. *lazy loading*. Ten spočívá ve vygenerování náhledu fotografie až v okamžiku, kdy o něj požádá klient. Samotné generování je v kontextu času potřebného ke zpracování HTTP požadavku časově náročné. Proto je více než žádoucí vygenerované náhledy ukládat vedle originálních fotografií. Díky tomu nebude nutné při dalším požadavku na zobrazení daného náhledu tento náhled znovu generovat, postačí „sáhnout“ po již dříve připraveném.

Přístupové body API, které fotografie zobrazují, podporují zaslat dotaz složený mimo jiné z následujících *query* parametrů:

- `width` – *šířka* vygenerovaného náhledu fotografie, pokud není zadána, nastaví se automaticky dle zadané *výšky* a poměru stran fotografie,
- `height` – *výška* vygenerovaného náhledu fotografie, pokud není zadána, nastaví se automaticky dle zadané *šířky* a poměru stran fotografie,
- `quality` – označuje míru komprese vygenerovaného náhledu dané fotografie.

[57]V závislosti na hodnotě proměnné prostředí `APP_ENV`

URL pro zobrazení konkrétní fotografie, která má šířku 500 px s kvalitou komprese 80 %, by pak mohla vypadat následovně:

```
/api/photos/35/versions/186/show?width=500&quality=80
```

Tento přístup je zvláště výhodný, pokud je na straně *frontendu* implementováno zobrazení obrázku pomocí atributu `srcset` [37] HTML tagu `img`.

5.1.6 Testování

Před samotným vývojem jsem si vytyčil neskromný cíl průběžně kód testovat automatickými testy, které by zaručily určitou úroveň kvality kódu a potvrdily by, že kód funguje přesně dle očekávání.

Testovat kód lze na několka úrovních. Snad každému programátorovi se vybaví *jednotkové testy*, jež testují správnou funkčnost drobných *jednotek* kódu. Dalším zástupcem testů mohou být např. *integrační testy*, které ověřují korektní spolupráci více nezávislých komponent.

Některé testy jsou však specifické pro konkrétní programovací jazyk, kterým je kód psán. *Backendová* část naší aplikace je postavena na jazyku PHP. PHP je jazyk *skriptovací*, který před svým spuštěním nevyžaduje samostatnou kompilaci, může být tedy náchylný k chybám, které se projeví až za běhu. Tyto *runtime* chyby se dají potlačit kvalitní statickou analýzou kódu.

Třešničkou na dortu je potom možnost tyto testy spouštět v rámci kontinuální integrace, kterou poskytuje v základu nástroj *GitLab*. O automatizovaném provádění testů, respektive nasazení kódu na cílové prostředí však pojednávají samostatné kapitoly 3.4.1.2, respektive 3.4.1.3. Průběh automatických testů v rámci kontinuální integrace vykonávaných nad kódem *backendové* části aplikace je k vidění na obrázku 5.5. Zde je podmínkou pro spuštění samotných testů úspěšné sestavení aplikace. Pokud všechny testy projdou bez chyby, následuje automatizované nasazení aplikace.

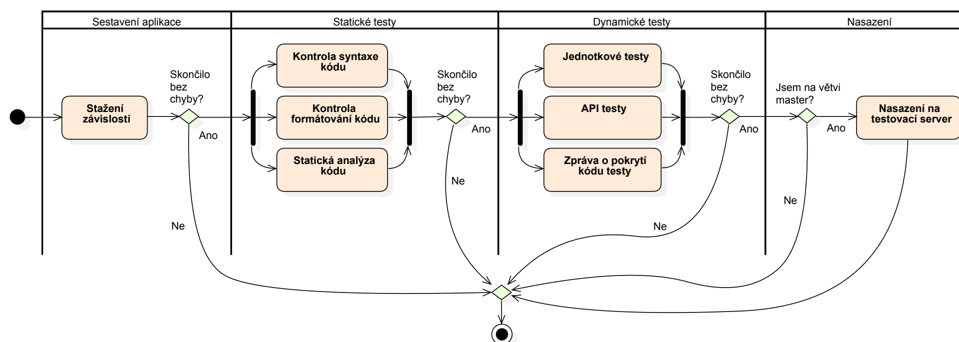
Na následujících řádcích osvětluji smysl jednotlivých částí automatického testování.

5.1.6.1 Kontrola syntaxe kódu

Jelikož je jazyk PHP *skriptovací*, nepotřebuje pro své spuštění svou kompilovanou podobu, jako např. jazyky C, či Java. Kompilátor v případě těchto dvou jazyků odhalí veškeré syntaktické chyby.

V případě PHP se tyto chyby při nejhorším projeví až při jeho spuštění. Nástrojů na odhalení těchto chyb je naštěstí celá řada. Obecně se takovým nástrojům říká *linter*, v našem případě využíváme následující:

- *YAML linter* – validace syntaxe konfiguračních souborů *YAML*, součást frameworku Symfony,



Obrázek 5.5: Diagram zachycující průběh automatického testování a nasazení *backendové* části aplikace

- *Twig linter* – validace syntaxe souborů šablon psaných v jazyce *Twig*, součást frameworku Symfony,
- *Symfony container linter* – validace volání Symfony komponent spočívající především v kontrole, zda programátor iniciuje komponenty s parametry s odpovídajícími datovými typy, součást frameworku Symfony,
- *PHP linter* [38] – validatce syntaxe PHP skriptů.

5.1.6.2 Kontrola formátování kódu

Potřeba nastavení jednotné „podoby“ kódu u projektů, na kterých pracuje více programátorů, je dobře obhájena v rámci podkapitoly 3.2.4. V ní jsem se zabýval nastavením jednotného formátování kódu během lokálního vývoje. Je ale možné jednotné formátování kódu nějak automaticky testovat? Naštěstí ano.

Pro automatickou kontrolu formátování využívám oblíbeného open-source nástroje *PHP_CodeSniffer* [39]. Ten nabízí možnost nejen odhalení prohřešků proti definovaným pravidlům, jak by měl kód vypadat, ale rovněž také nabízí nástroj pro automatickou opravu takového kódu. Sluší se zde zmínit, že soubor pravidel, které definují podobu kódu, jsem převzal z veřejného repozitáře [40] a částečně upravil k obrazu svému.

Jen namátkou zde vypíši některá zásadní pravidla, která v kódu *backendové* části aplikace vyžaduji:

- až na výjimky striktní dodržování standardu *PSR-2* [41],
- název třídy by měl být stejný, jako název souboru,
- jmenný prostor třídy by měl odpovídat umístění souboru v adresářové struktuře,

- zákaz používání funkcí a metod označených jako *depricated*,
- využívat zkráceného zápisu pro deklaraci pole pomocí novějšího `[]` namísto zastaralého `array()`,
- odhalování nepoužívaného kódu (nepoužívaný vložený kód pomocí `use`, nepoužívané třídní atributy či proměnné, ...),
- vynucení striktního typování pomocí `declare(strict_types=1)`,
- povinnost psát návratové hodnoty funkcí a metod,
- povinnost uvádět datové typy argumentů funkcí a metod,
- a mnoho dalších.

5.1.6.3 Statická analýza kódu

Posledním z testů spadající do kategorie statických testů, jež nepotřebují ke svému provedení spuštění testovaného kódu, je *statická analýza kódu*. Ta má za úkol odhalit potenciální *runtime* chyby, které mohou být zapříčiněny následujícími důvody:

- volání funkcí a metod se špatným počtem argumentů nebo s argumenty jiného datového typu,
- přistupování k neexistujícím nebo nedefinovaným proměnným a třídním atributům,
- reálné návratové hodnoty funkcí jsou jiné, než ty, ke kterým jsme se zavázali v předpisu metody,
- chybějící návratové hodnoty funkcí a metod,
- manipulace s proměnnou, která může být v době běhu deklarována na jiný datový typ, než je v kódu očekáváno,
- a mnoho dalších.

Pro *statickou analýzu* kódu využívám oblíbený nástroj *PHPStan* [42], jenž se dá použít pro libovolné aplikace psané v programovacím jazyce PHP. *PHPStan* je navíc možné volitelně rozšířit o dostupné pluginy [43][44] určené pro aplikace psané pomocí frameworku Symfony a Doctrine. Díky tomu je *PHPStan* schopen odhalit potenciální chyby specifické pro tyto frameworky – je např. schopen hlídat správné získávání služeb (*Services*) ze Symfony kontejnerů (*Containers*), či validovat *DQL* dotazy.

5.1.6.4 Jednotkové testy

Prvním ze zástupců dynamických testů, jež ke svému provedení již vyžadují spuštění samotného testovaného kódu, jsou *jednotkové testy*. Jak již vyplývá z názvu, tyto testy mají za úkol ověřit funkčnost pouze jedné *jednotky*. Touto jednotkou může být nějaká funkce, či třída.

Jako základ pro jednotkové testy využívám framework *Codeception* [45]. Testovány jsou především *DTO*[58] třídy, jež reprezentují databázové entity, či třídy, jež serializují jednotlivé *DTO* pro potřeby jejich posílání v rámci HTTP odpovědi.

5.1.6.5 API testy

Dalším ze zástupců dynamických testů jsou *API testy*, což jsou ve své podstatě *funkcionální testy*, jež mají za úkol ověřit komplexní chování naší aplikace.

Pro každý přístupový bod API je psáno několik testů, které lze rozdělit do dvou základních kategorií:

Strukturální testy
: Strukturální testy ověřují srávné chování jednotlivých přístupových bodů, co se podoby jednotlivých HTTP požadavků a HTTP odpovědí týče. Ověřuje se, zda *backend* správně zareaguje, snažíme-li se vytvořit zdroj s neúplnou strukturou (např. v těle dotazu chybí nějaký z atributů, ...), či zda odpověď ze serveru na určitý dotaz obsahuje opravdu všechna kýžená data.

  Ukázka jednoho z mnoha testů, jenž ověřuje strukturu odpovědi na dotaz ohledně získání kolekce papírů, je k vidění v ukázce kódu 5.2.

Bezpečnostní testy
: Bezpečnostní testy mají za úkol pohlídat ochranu konkrétních přístupových bodů API před uživateli, kteří k danému přístupovému bodu nemají práva.

  Ukázka typického bezpečnostního testu, jenž má za úkol ověřit, že se k danému přístupovému bodu nedostane nepřihlášený uživatel, je vidět v ukázce kódu 5.3.

Jako základ pro API testy používám opět framework *Codeception*. Pro každý přístupový bod je v našem případě psáno zpravidla kolem dvou *strukturálních* API testů a zhruba kolem čtyř *bezpečnostních* API testů.

Vykonávání komplexních API testů zahrnuje mimo jiné i ověření schopnosti manipulovat s daty v databázi. Jedny ze základních pravidel pro psaní automatizovaných testů jsou:

[58] Data Transfer Object

```php
<?php

public function testGetPapers(ApiTester $I): void
{
    $I->wantToTest('get papers successfully');

    $I->amLoggedInAsPrinter();

    $I->sendGET('/api/papers');

    $I->seeResponseCodeIs(HttpCode::OK);
    $I->seeResponseIsJson();
    $I->seeResponseMatchesJsonType([
        '_pagination' => 'array',
        'items' => 'array'
    ]);
    $I->seeResponseMatchesJsonType([
        'id' => 'integer',
        'name' => 'string',
        'price' => 'float|integer',
        'weight' => 'float|integer',
        'finish' => 'array',
        'texture' => 'array'
    ], '$.items[0]');
}
```

Ukázka kódu 5.2: Ukázka strukturálního API testu pro ověření správné struktury odpovědi na dotaz ohledně získání kolekce papírů

- opakovatelnost – každý test by měl skončit se stejným výsledkem, nehledě na počet jeho opakovaných vykonání,
- nezávislost – každý test by měl být nezávislý na testech předchozích, jakožto i na pořadí, v jakém jsou jednotlivé testy vykonávány.

Testovací framework *Codeception* nám poskytuje rozhraní pro definování akcí, jež se mohou volitelně provést např. před každým testem či před každou skupinou[59] testů. Pro dodržení výše zmíněných principů *opakovatelnosti* a *nezávislosti* jsme pro testovací prostředí využili databázi umístěnou v rámci samostatného *Docker kontejneru* `postgres_test`. Tato databáze se před spuštěním libovolné skupiny testů naplnila předem definovanými testovacími daty.

[59] V řeči *Codeception* se mluví o tzv. *test suits*.

```php
<?php

public function testGetPapersUnauthorized(ApiTester $I): void
{
    $I->wantToTest('get papers as unauthorized');

    $I->sendGET('/api/papers');

    $I->seeResponseCodeIs(HttpCode::UNAUTHORIZED);
}
```

Ukázka kódu 5.3: Ukázka bezpečnostního API testu pro ověření zabezpečení přístupového bodu před nepřihlášenými uživateli

Každý jednotlivý test se poté nad připravenou databází vykonával v rámci samostatné *transakce*, jež byla po provedení testu odvolána. Díky tomu se po vykonání každého testu vrátí databáze do výchozího stavu, čímž je princip *opakovatelnosi* a *nezávislosti* zaručen. Volbou databáze na základě aktuálního prostředí[60] je zároveň docíleno elegantního oddělení databáze využívané během vývoje a během testování, kdy nedochází s každým spuštěním testů nad *testovací databází* k přemazání dat *vývojové* databáze.

5.1.6.6 Zpráva o pokrytí kódu testy

Abychom měli přehled o tom, jak dobře[61] je náš kód otestován, hodí se nechat si čas od času vygenerovat zprávu o pokrytí kódu testy.

Tuto možnost nám naštěstí framework *Codeception* nabízí. Díky nutnosti spouštět tyto testy s rozšířením *XDebug* však generování reportu trvá násobně déle, než provedení samotných *unit* a *API* testů.

Zpráva o pokrytí nám dává přímo informace o tom, jaký řádek byl pokryt kterým testem nebo z kolika procent jsou konkrétní třídy, metody či funkce pokryty. Na základě této zprávy se poté dá přemýšlet o napsání testů pro ty části kódu, jež zatím nejsou pokryty. Konkrétní podoba zprávy o pokrytí je k vidění na obrázku 5.6.

5.2 Frontend

Obchodní logika naší aplikace, jíž jádrem je bezpochyby *backendová* část, kterou jsem popsal v předchozí podkapitole 5.1, by byla naprosto k ničemu, kdybychom běžným uživatelům neposkytli způsob, jak s naší aplikací interagovat.

[60] *Testovací* vs. *vývojové* prostředí
[61] Jaké množství kódu je pokryto testy

	Code Coverage					
	Lines		Functions and Methods		Classes and Traits	
Total	79.49%	779 / 980	44.83%	52 / 116	11.11%	2 / 18
Enumeration	100.00%	36 / 36	100.00%	8 / 8	100.00%	1 / 1
Format	85.54%	142 / 166	31.25%	5 / 16	0.00%	0 / 3
Notification	89.36%	42 / 47	57.14%	4 / 7	0.00%	0 / 1
Paper	88.46%	46 / 52	33.33%	2 / 6	0.00%	0 / 1
Photo	69.90%	137 / 196	20.00%	3 / 15	0.00%	0 / 2
PhotoEvent	85.11%	120 / 141	25.00%	3 / 12	0.00%	0 / 2
QualityTier	90.38%	47 / 52	33.33%	2 / 6	0.00%	0 / 1
Registration	91.67%	33 / 36	50.00%	2 / 4	0.00%	0 / 1
Security	87.50%	35 / 40	25.00%	1 / 4	0.00%	0 / 1
User	52.34%	67 / 128	37.50%	6 / 16	0.00%	0 / 3
AbstractController.php	85.88%	73 / 85	71.43%	15 / 21	0.00%	0 / 1
IndexController.php	100.00%	1 / 1	100.00%	1 / 1	100.00%	1 / 1

Obrázek 5.6: Ukázka zprávy o pokrytí kódu kontrolerů testy

Navržená architektura naší aplikace nás zavazuje pro tyto účely k vytvoření samostatné *komponenty*. Tuto komponentu budeme na následujících řádcích nazývat *frontend*.

5.2.1 Technologie

Základní stavební kameny *backendové* části aplikace se za dlouhou dobu vývoje technologií, ze kterých se tato komponenta skládá, stačila ustálit a za poslední dobu v této oblasti nedocházelo k příliš velkým změnám, neřkuli převratným inovacím.

Škála technologií, které může programátor využít pro vývoj *webového* frontendu, však v posledních letech zažívá nebývalý rozmach. Dnes si může programátor vybrat z celé řady *frameworků* a knihoven, pomocí kterých je vývoj aplikace mnohem jednodušší, než v dřívějších dobách.

Několik zásadních milníků, které dnešnímu stavu předcházely, shrnuje článek [46]. Na počátku stálo ustanovení standardu HTML v roce 1991 a jeho doplnění o kaskádové styly[62] v roce 1994. V roce 1995 vyšla na světlo světa specifikace skriptovacího jazyka, jenž známe dnes jako *JavaScript*.

Počátek tisíciletí dal na základě technologií CSS a Javascript vzniknout knihovnám, které v sobě obsahovaly komplexnější znovupoužitelné funkce, bez kterých se žádný moderní web v té době neobešel. Za všechny takové knihovny se zde sluší zmínit minimálně *jQuery* [47]. Objevily se rovněž centrální repozitáře, ze kterých měl programátor možnost takové knihovny stahovat. Mezi ně patří dodnes hojně používané *NPM registry* [48], či *Bower* [49].

S rozmachem podílu mobilních zařízení a se stále se zvyšující komplexitou webových přišly javascriptové *frameworky*. Ty s sebou přinesly možnost psát aplikace ve formě *SPA*[63], kdy jsou při první iniciaci webové stránky staženy

[62] CSS
[63] Single-page application

všechny potřebné skripty, následný průchod tím pádem dovoluje rychlé vykreslení uživatelského rozhraní, dodatečná data jsou typicky stahována na pozadí ze vzdáleného API. Dalším revolučním přínosem javascriptových frameworků budiž uveden tzv. *Two-way data binding*, tedy automatická provázanost dat, jež jsou zadávána uživatelem v rámci *DOM*[64] dokumentu HTML a jejich obrazu v podobě atributů javascriptového objektu. Jako zástupce takových frameworků bych zde měl zmínit minimálně *AngularJS* [50] či *React* [51].

Programátor musel v prvopočátcích[65] počítat s tím, že některé vlstnosti HTML, CSS a JavaScriptu se chovaly v různých prohlížečích jinak. Část těchto neduhů řeší tzv. *polyfil* – část kódu, jež zajišťuje kompatibilitu starších prohlížečů s funkcionalitami nových verzí programovacího jazyka.

5.2.1.1 Vue.js

Pro účely vývoje *frontendové* části aplikace jsem v našem případě zvolil framework *Vue.js* [52]. Tento framework navrhl a implementoval v roce 2014 Evan You. Na rozdíl od frameworku *React*, který je velmi přizpůsobivý, nebo frameworku *Angular*, který naopak razí velmi striktní postupy, se *Vue.js* snaží jít tou střední cestou. Kompletní framework se skládá z několika komponent, jejichž kombinací vzniká kompletní ekosystém pro vývoj moderních javascriprových aplikací. Jde především o následující komponenty:

Vue Loader

Jedná se o *loader* pro nástroj *Webpack* [53]. Ten umožňuje vytvářet tzv. *single-file* komponenty. Tedy soubory, typicky s koncovkou `.vue`, jež v sobě obsahují zpravidla tři[66] části:

- `template` – oblast definující šablonu komponenty, její syntaxe vychází z HTML, pomocí speciálních direktiv však lze renderovat dynamická data, podmíněně renderovat jednotlivé části, apod.,
- `script` – jádro logiky komponenty, obsahuje data a metody potřebné pro správnou funkci komponenty,
- `style` – oblast pro deklaraci kaskádových stylů.

Právě z velkého množství takovýchto komponent se zpravidla skládá typická *Vue* aplikace. Ukázka velmi jednoduché komponenty je pak k vidění v ukázce kódu 5.4

Vue Router

Vue router je oficiální plugin pro zpřístupnění dynamického routování v rámci *Vue* aplikace.

[64] Document Object Model
[65] A někdy i dnes, ovšem již ne v takové míře
[66] Může jich však být více, nebo i méně, povinná je pouze část `<script/>`

Vuex

Vuex je knihovna, jež do *Vue* aplikací zavádí *state management pattern*. Pomocí něj lze mezi všechny komponenty zavést jednotné úložiště. Toho lze využít v případě, kdy komponenta A změní nějaký svůj atribut, který potřebuje zobrazit zcela nezávislá komponenta B. Zároveň lze tuto knihovnu využít jako dlouhodobé úložiště lokálních dat (např. informace o přihlášeném uživateli, apod.). *Vuex* představuje abstrakci nad lokálním úložištěm, které může být např. *local storage*, či *session storage*.

Vue CLI

Nástroj *Vue CLI* je, jak již vyplývá z názvu, nástroj pro *příkazovou řádku* pro vytváření a správu *Vue* aplikací a jejich rozšíření. Kromě prostředí pro příkazovou řádku obsahuje i grafické rozhraní *Vue UI*.

```
1  <template>
2      <div class="greeting">
3          <p>Hello, {{ name }}!</p>
4      </div>
5  </template>
6
7  <script>
8  export default {
9      name: 'MyComponent',
10     data: () => ({
11         name: 'world'
12     })
13 }
14 </script>
15
16 <style lang="scss" scoped>
17 .greeting {
18     color: grey;
19 }
20 </style>
```

Ukázka kódu 5.4: Ukázka jednoduché Vue *single-file* komponenty

5.2.1.2 Vuetify

To, jak bude naše aplikace vypadat navenek neurčuje ani tak volba javascriptového frameworku, jako spíš kombinace HTML tagů, na ně navěšených kaská-

dových stylů. Nic programátorovi nebrání postavit si takovou sadu UI[67] komponent – jak se říká – od lesa. Proč však plýtvat energii a čas na vývoj něčeho, co již vyvinul někdo před námi a dal celou sadu užitečných UI komponent k dispozici?

Příklad knihovny, která podobné UI komponenty poskytuje a kterou zároveň v našem projektu využíváme, je knihovna *Vuetify* [54], jež defacto implementuje designový koncept *Material Design* [55].

Tato knihovna nabízí v základu celou řadu užitečných komponent, za všechny zde připomenu jen některé:

Základní komponenty struktury stránky
: Základní rozložení stránky, které spočívá v připravené hlavičce, patičce, prostoru pro navigaci či místě pro samotný obsah stránky. Nutno zde ocenit i příkladnou připravenost všech komponent pro zobrazení jak na velkém displayi stolního počítače, tak i na malém displayi mobilního telefonu.

Formuláře a formulářová pole
: Sada formulářových polí pro různé datové typy, mezi kterými nechybí *textová* pole, *date picker*[68], *checkbox* a *radio button*, *select field*[69], apod.

Datové tabulky
: Komplexní komponenta, jež spočívá v možnosti zobrazit sadu záznamů ve formě tabulky. Obsah samotné tabulky lze zpravidla filtrovat, řadit a v neposlední řadě i stránkovat.

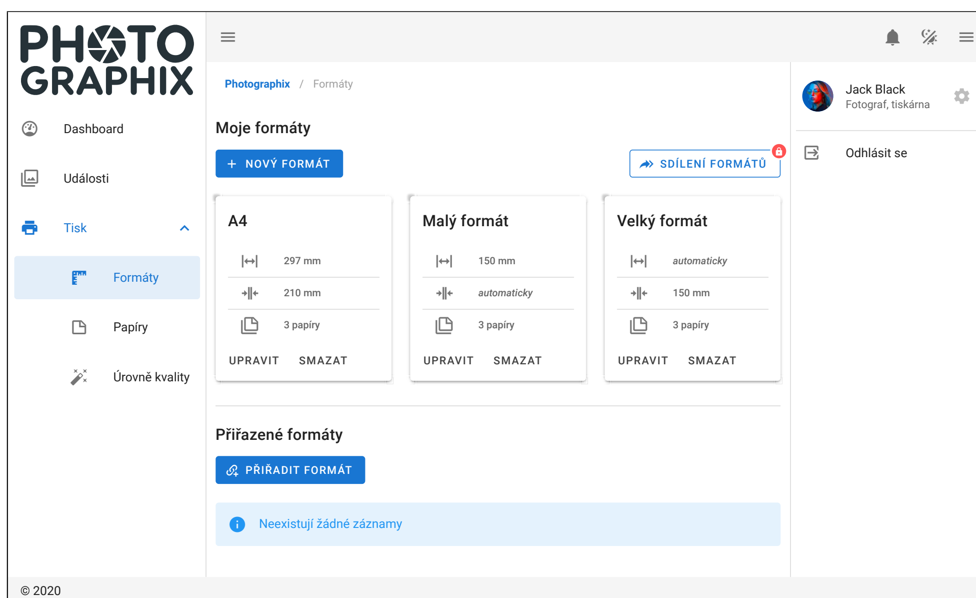
Ukázka začlenění komponent poskytovaných knihovnou *Vuetify* do *frontendové* části naší aplikace je k vidění na obrázku 5.7, ta samá stránka, avšak v tmavém režimu, je pak k vidění na obrázku 5.8. Konkrétně na této stránce uživatel spravuje své *formáty*.

5.2.1.3 JavaScrip vs. TypeScript

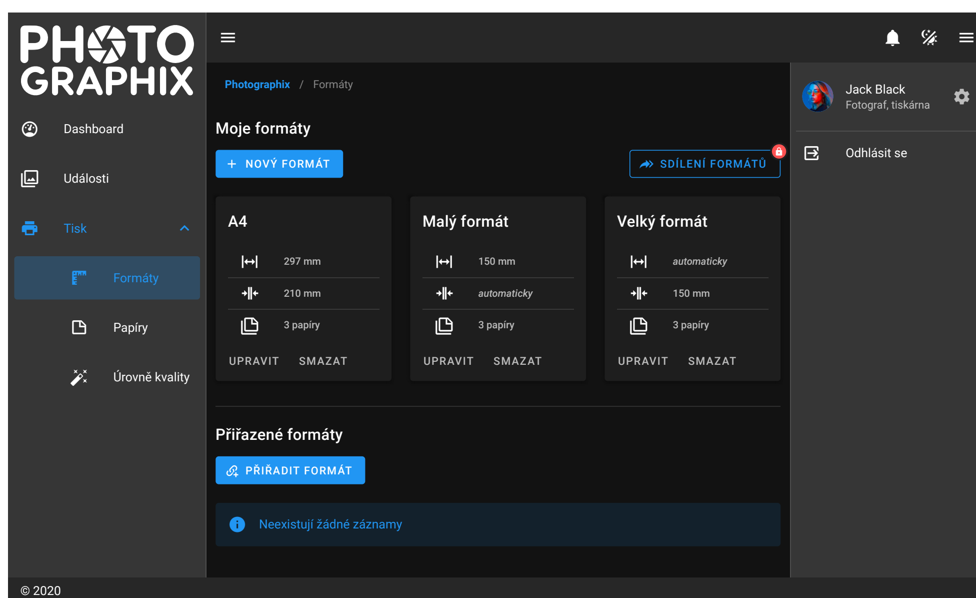
Volba technologie pro implementaci *frontendové* části aplikace padla na *javascriptový* framework Vue.js. Jeho základem je implementace standardu *ECMAScript*, jež se nazývá *JavaScript*. *JavaScript* je skriptovací, objektový, událostmi řízený jazyk, který dnes nachází uplatnění především jako jazyk, pomocí kterého jsou psány komplexní webové aplikace, které běží jak ve webovém prohlížeči, tak na webovém serveru.

Důležitým rysem jazyka je, stejně jako u většiny skriptovacích jazyků, jeho dynamické typování. *JavaScript* nemá, narozdíl např. od jazyků *C* nebo *Java*, struktury jako jsou *třída* či *rozhraní*. S tím souvisí skutečnost, že *JavaScript*

[67]User interface
[68]Pro výběr datumu
[69]Pro výběr hodnoty z předpřipravené nabídky



Obrázek 5.7: Ukázka stránky pro správu formátů, světlá varianta



Obrázek 5.8: Ukázka stránky pro správu formátů, tmavá varianta

nemá žádné mechanismy pro rozumnou kontrolu datových typů argumentů a návratových hodnot funkcí. Tento fakt dovoluje programátorovi v malých

aplikacích velkou míru svobody a podporuje rychlý počáteční vývoj, jak však bylo již zmíněno v podkapitole 5.1.6.3 o statické analýze kódu PHP aplikací, je tato vlastnost jazyka zdrojem mnoha potencionálních chyb, které se zpravidla projeví až za běhu aplikace. Mimo jiné je během vývoje kvůli tomu programátor ochuzen o schopnost IDE našeptávat proměnné s vhodným datovým typem a v reálném čase kontrolovat jejich validitu.

Pomyslným řešením tohoto problému je rozšíření jazyka *JavaScript*, jež se nazývá *TypeScript* [56]. Ten přidává do *JavaScriptu* některé zásadní funkce, pomocí kterých je možné kvalitu kódu mnohem lépe kontrolovat a které podporují lepší organizaci kódu v projektu:

Anotace datových typů
: *TypeScript* umožňuje deklaraci datových typů prostřednictvím anotací. Ty mohou být primitivní[70], nebo vlastní – vycházející z předpisů vlastních tříd či rozhraní.

Třídy a rozhraní
: *TypeScript* rozšiřuje *JavaScript* o třídy a rozhraní zde standardu *ECMAScript 6*.

Moduly
: *TypeScript* podporuje organizaci a zapouzdření tříd, rozhraní a funkcí do vlastních jmenných prostorů.

Je zde třeba také zmínit, že pro spuštění programu psaném v jazyku *TypeScript* je nutné využití *transkompilace*, což je proces převedení zdrojového kódu z jednoho jazyka do jiného. V tomto případě je nutné převést zdrojové kódy psané v jazyce *TypeScript* do jazyka *JavaScript*.

Framework *Vue.js* ve verzi 2.6 díky nástroji *Vue CLI* psaní kódu v jazyku *TypeScript* dovoluje, bohužel nebylo této možnosti při založení projektu využito. To nyní značně stěžuje statickou kontrolu kódu aplikace. Možnost přejít na *TypeScript* v průběhu vývoje samozřejmě možná je[71], vyžadovalo by to však nemalé úpravy všech komponent tak, aby využívaly všech výhod, jež *TypeScript* nabízí.

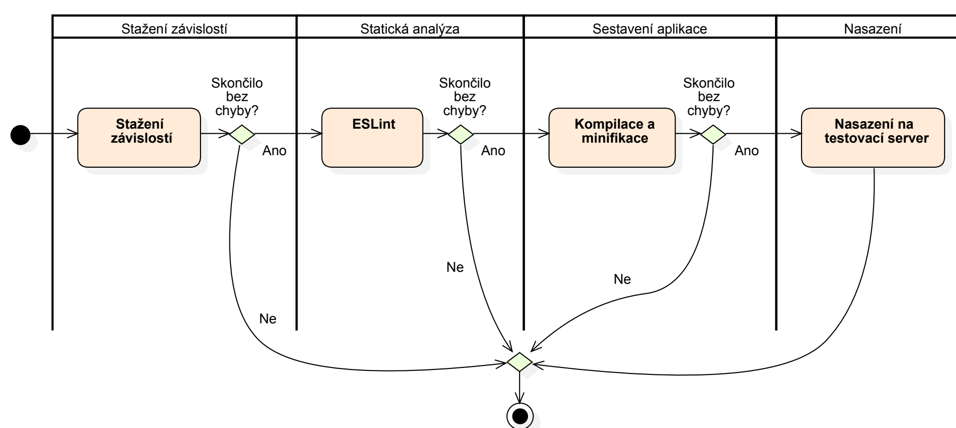
Převedení zdrojových kódů *frontendové* části aplikace do jazyka *TypeScript* je jedna z hlavních výzev, které na tento projekt čekají v budoucnu. Otázkou tak je pouze to, zda se tak stane ještě nad Vue ve verzi 2.6, či až po zmigrování na verzi 3.0, jež psaní kódu v *TypeScriptu* dovoluje již v základu. S třetí verzí frameworku *Vue.js* však prozatím není kompatibilní knihovna UI komponent *Vuetify*. Z těchto důvodů je současný záměr čekat na uvolnění aktualizované verze *Vuetify*, jež bude kompatibilní s třetí verzí *Vue.js* a spolu s ní přepsat zdrojové kódy naší aplikace do jazyka *TypeScript*.

[70] `string`, `number`, `boolean`, `any`

[71] https://vuejs.org/v2/guide/typescript.html

5.2.1.4 Testování

Jádro testování *frontendové* části naší aplikace spočívá ve *statické analýze* kódu. Zprvu jsem přemýšlel i nad částečným pokrytím aplikace *jednotkovými testy*, jelikož je však celá aplikace prakticky celá poskládaná pomocí UI komponent z knihovny *Vuetify*, jež jsou samy o sobě dobře otestovány, rozhodl jsem se od jednotkových testů ustoupit. Jistě by nebylo od věci celou aplikaci pokrýt alespoň částečně tzv. *E2E*[72] testy, jež by měly ověřit základní průchod systémem a to včetně interakce s *backendovou* částí aplikace. Pokrytí aplikace takovými testy by však téměř vydalo na samostatnou práci, a proto nechávám tuto možnost otevřenou pro budoucí vývoj. Diagram popisující proces automatického testování a nasazení aplikace v rámci kontinuální integrace je k vidění na obrázku 5.9.



Obrázek 5.9: Diagram zachycující průběh automatického testování a nasazení *frontendové* části aplikace

Stažení závislostí

Tato část spočívá ve stažení knihoven třetích stran, které jsou nezbytné pro chod aplikace. Konkrétní závislosti a jejich požadované verze jsou definovány v souboru `package.json`.

ESLint

Skripty v této části mají za úkol provedení statické analýzy nad zdrojovým kódem. Vzhledem k problémům popsaných v podkapitole 5.2.1.3 se dá provádět statická analýza kódu jen v omezené míře. Pro analýzu kódu byl použit nástroj *ESLint* [57] spolu s oficiálními pravidly pro kontrolu kvality kódu [58], které jsou doporučovány přímo autory frameworku *Vue.js*.

[72] End-to-End

Kompilace a minifikace

Veškeré zdrojové kódy procházejí procesem tzv. *minifikace*. Ta má za následek značné snížení velikosti výsledného souboru obsahujícího zdrojový kód. Výsledné zdrojové soubory potřebné pro spuštění aplikace jsou vygenerovány do složky `dist/`.

Nasazení na testovací server

Tento bod spočívá v automatickém nasazení připravené aplikace na produkční prostředí.

Kapitola 6

Závěr

Hlavní vytyčené cíle na začátku této práce byly splněny – především pak byla provedena analýza a detailní návrh nové webové služby, na jehož základě byla služba z veliké části naimplementována a řádně otestována automatickými testy. V době dokončování této práce zbývá doimplementovat pouze obchodní logiku, která se týká *objednávek*. Ani dokončení této zbývající funkcionality však neznamená, že vývoj na aplikaci skončí.

Zbývá uvést službu do stavu, kdy ji bude možné reálně používat. To v sobě zahrnuje především vyhotovení klientské části aplikace, přes kterou bude možné vystavené fotografie objednávat cílovými zákazníky. Návrhu a implementaci této komponenty se již během práce v rámci předmětů BI-SP1 a BI-SP2 chopil David Fencl, jenž obojí rozvede ve své bakalářské práci, na níž již pilně pracuje. Spolu s tím by bylo vhodné podrobit hotovou implementaci frontendové části aplikace uživatelskému testování, jež by mělo přispět ke zkvalitnění uživatelského zážitku při používání naší služby. Jednou z výzev pro nadcházející vývoj je také zakomponování jazyka TypeScript do frontendové části aplikace.

Není vyloučeno, že se na tomto projektu bude pokračovat s novým týmem v rámci nového běhu předmětů BI-SP1 a BI-SP2. Veškeré procesy a podpůrné nástroje jsou zavedené, adaptace nového týmu na aktuální stav projektu by tedy neměla být problém.

Vadou na kráse celého procesu byla bezesporu situace spojená s pandemií koronaviru, která prakticky celý[73] svět připravila o možnost osobních setkání. To poznamenalo i pravidelné schůzky vývojového týmu, které se přesunuly do online prostředí videohovorů. Tím se poněkud snížila efektivita, s jakou tým kooperoval na společných úkolech. Změna způsobu výuky všech předmětů na naší fakultě kladla na studenty zvýšené časové nároky na studium, přesto však celý tým věnoval tomuto projektu velké množství svého času, za což jim patří můj dík.

[73]Nejen akademický

Zdroje

1. WEISS, Klára. *Kvalitativní vs. kvantitativní metody výzkumu v úvodní části designového procesu* [online]. 2019 [cit. 2020-11-18]. URL: https://medium.com/design-kisk/kvalitativní-vs-kvantitativní-metody-výzkumu-v-úvodní-části-designového-procesu-d19b532dedd9.
2. TECHOPEDIA.COM. *What is Development Environment - Definition from Techopedia* [online]. 2020 [cit. 2020-09-25]. URL: https://www.techopedia.com/definition/16376/development-environment.
3. TORVALDS, Linus. *Git* [online] [cit. 2020-09-25]. URL: https://git-scm.com.
4. DOCKER, INC. *Docker: Empowering App Development for Developers* [online] [cit. 2020-10-05]. URL: https://www.docker.com.
5. GRYGAŘÍKOVÁ, Michaela. *Docker, Kubernetes a kontejnery. Jak fungují a proč je chtít* [online]. 2019 [cit. 2020-10-05]. URL: https://www.master.cz/blog/docker-kubernetes-kontejnery-jak-funguji-proc-je-chtit.
6. CARTER, Eric. *2018 Docker usage report* [online]. 2018 [cit. 2020-10-05]. URL: https://sysdig.com/blog/2018-docker-usage-report.
7. DOCKER, INC. *Docker overview* [online] [cit. 2020-10-05]. URL: https://docs.docker.com/get-started/overview.
8. DOCKER, INC. *Performance tuning for volume mounts (shared filesystems)* [online] [cit. 2020-10-08]. URL: https://docs.docker.com/docker-for-mac/osxfs-caching.
9. VIVA IT LIMITED. *Docker for Mac Performance using NFS (Updated for macOS Catalina)* [online]. 2019 [cit. 2020-10-10]. URL: https://vivait.co.uk/labs/docker-for-mac-performance-using-nfs.
10. JETBRAINS S.R.O. *JetBrains: Developer Tools for Professionals and Teams* [online] [cit. 2020-10-30]. URL: https://www.jetbrains.com.

11. EDITORCONFIG TEAM. *EditorConfig* [online] [cit. 2020-10-31]. URL: https://editorconfig.org.
12. CHACON, Scott; STRAUB, Ben. *Pro Git*. 2. vyd. Apress, 2014. ISBN 978-1-4842-0077-3.
13. GITLAB INC. *The first single application for the entire DevOps lifecycle – GitLab* [online] [cit. 2020-11-11]. URL: https://about.gitlab.com.
14. RERYCH, Markus. *Wasserfallmodell* [online] [cit. 2021-01-04]. URL: http://cartoon.iguw.tuwien.ac.at/fit/fit01/wasserfall/entstehung.html.
15. ŠIMŮNEK, David. *Jaký je rozdíl mezi Waterfall a Agile přístupem* [online] [cit. 2021-01-04]. URL: https://www.davidsimunek.com/post/jaky-je-rozdil-mezi-waterfall-a-agile.
16. MCCONNELL, S. *Rapid Development: Taming Wild Software Schedules* [online]. Microsoft Press, 1996 [cit. 2021-01-04]. Best Practices Series. ISBN 978-1-55615-900-8. URL: https://books.google.cz/books?id=qM4Yzf8K9hwC.
17. BECK, Kent et al. *Manifesto for Agile Software Development* [online] [cit. 2021-01-04]. URL: http://agilemanifesto.org.
18. SCRUMGUIDES.ORG. *Scrum Guides* [online] [cit. 2021-01-05]. URL: https://scrumguides.org.
19. BECK, K.; GAMMA, E. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000. An Alan R. Apt Book Series. ISBN 978-0-201-61641-5.
20. MALEC, Oldřich. *Řízení projektu a infrastruktury portálu pro podporu výuky předmětu BI-DBS* [online]. 2017 [cit. 2021-01-05]. URL: https://dspace.cvut.cz/handle/10467/69399. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
21. FIT ČVUT. *Popis předmětu – BI-SP1* [online] [cit. 2021-01-05]. URL: http://bk.fit.cvut.cz/cz/predmety/00/00/00/00/00/00/03/46/22/p3462206.html.
22. EUROPEAN COMMISSION. *European Credit Transfer and Accumulation System (ECTS)* [online] [cit. 2021-01-05]. URL: https://ec.europa.eu/education/resources-and-tools/european-credit-transfer-and-accumulation-system-ects.
23. LANG, Jean-Philippe. *Redmine* [online] [cit. 2021-01-06]. URL: https://www.redmine.org.
24. SLACK TECHNOLOGIES. *Slack: Where work happens* [online] [cit. 2021-01-06]. URL: https://slack.com.
25. GOOGLE, INC. *Google Meet* [online] [cit. 2021-01-05]. URL: https://meet.google.com.

26. WARREN, Tom. *Microsoft Teams goes down just as Europe logs on to work remotely* [online] [cit. 2021-01-05]. URL: https://www.theverge.com/2020/3/16/21181300/microsoft-teams-down-outage-europe-remote-working-coronavirus.
27. SENSIO LABS. *Symfony, High Performance PHP Framework for Web Development* [online] [cit. 2020-12-29]. URL: https://symfony.com.
28. JONGE, Frank de. *League Flysystem* [online] [cit. 2020-12-29]. URL: https://github.com/thephpleague/flysystem/tree/1.x.
29. FIELDING, Roy Thomas. *REST: Architectural Styles and the Design of Network-based Software Architectures* [online]. 2000 [cit. 2020-12-08]. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Dis. pr.
30. FRIENDSOFSYMFONY. *FOSRestBundle* [online] [cit. 2020-12-28]. URL: https://github.com/FriendsOfSymfony/FOSRestBundle.
31. NESBITT, Brian. *Carbon* [online] [cit. 2020-12-28]. URL: https://github.com/briannesbitt/Carbon.
32. OPENAPI INITIATIVE. *Swagger* [online] [cit. 2020-12-28]. URL: https://swagger.io.
33. NELMIO. *NelmioApiDocBundle* [online] [cit. 2020-12-28]. URL: https://github.com/nelmio/NelmioApiDocBundle.
34. POSTMAN. *Postman | The Collaboration Platform for API Development* [online] [cit. 2020-12-28]. URL: https://www.postman.com.
35. DE DEYNE, Sebastian. *Create secured URLs with a limited lifetime* [online] [cit. 2020-12-28]. URL: https://github.com/spatie/url-signer.
36. SCHMID, Jim; GREMINGER, David. *OneupFlysystemBundle* [online] [cit. 2020-12-29]. URL: https://github.com/1up-lab/OneupFlysystemBundle.
37. MOZILLA FOUNDATION. *Responsive images* [online] [cit. 2020-12-31]. URL: https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images.
38. ONDERKA, Jakub. *PHP Parallel Lint* [online] [cit. 2020-12-29]. URL: https://github.com/php-parallel-lint/PHP-Parallel-Lint.
39. SHERWOOD, Greg. *PHP_CodeSniffer* [online] [cit. 2020-12-29]. URL: https://github.com/squizlabs/PHP_CodeSniffer.
40. SLEVOMAT. *Slevomat Coding Standard* [online] [cit. 2020-12-29]. URL: https://github.com/slevomat/coding-standard.
41. *PSR-2: Coding Style Guide* [online] [cit. 2020-12-29]. URL: https://www.php-fig.org/psr/psr-2.

42. MIRTES, Ondřej. *PHPStan - PHP Static Analysis Tool* [online] [cit. 2020-12-29]. URL: https://phpstan.org.
43. MIRTES, Ondřej. *PHPStan Symfony Framework extensions and rules* [online] [cit. 2020-12-29]. URL: https://github.com/phpstan/phpstan-symfony.
44. MIRTES, Ondřej. *Doctrine extensions for PHPStan* [online] [cit. 2020-12-29]. URL: https://phpstan.org.
45. CODECEPTION. *Codeception - PHP Testing framework - PHP unit testing, PHP e2e testing, database testing* [online] [cit. 2020-12-29]. URL: https://codeception.com.
46. WANYOIKE, Michael. *History of front-end frameworks* [online] [cit. 2021-01-02]. URL: https://blog.logrocket.com/history-of-frontend-frameworks .
47. RESIG, John. *jQuery* [online] [cit. 2021-01-02]. URL: https://jquery.com.
48. Z. SCHLUETER, Isaac. *NPM* [online] [cit. 2021-01-02]. URL: www.npm js.com.
49. TWITTER, INC. *Bower – A package manager for the web* [online] [cit. 2021-01-02]. URL: https://bower.io.
50. GOOGLE, INC. *AngularJS – Superheroic JavaScript MVW Framework* [online] [cit. 2021-01-02]. URL: https://angularjs.org.
51. FACEBOOK INC. *React – A JavaScript library for building user interfaces* [online] [cit. 2021-01-02]. URL: https://reactjs.org.
52. YOU, Evan. *Vue.js – The Progressive JavaScript Framework* [online] [cit. 2021-01-02]. URL: https://vuejs.org.
53. KOPPERS, Tobias. *Webpack* [online] [cit. 2021-01-02]. URL: https://webpack.js.org.
54. VUETIFY TEAM. *Vuetify – A Material Design Framework* [online] [cit. 2021-01-02]. URL: https://material.io/design.
55. GOOGLE, INC. *Material Design* [online] [cit. 2021-01-02]. URL: https://vuetifyjs.com.
56. MICROSOFT, INC. *TypeScript: Typed JavaScript at Any Scale* [online] [cit. 2021-01-03]. URL: https://www.typescriptlang.org.
57. C. ZAKAS, Nicholas. *ESLint – Find and fix problems in your JavaScript code* [online] [cit. 2021-01-03]. URL: https://eslint.org.
58. NAGASHIMA, Toru. *ESLint plugin for Vue.js* [online] [cit. 2021-01-03]. URL: https://eslint.vuejs.org.

Příloha A

Seznam použitých zkratek

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSS	ascading Style Sheets
DOM	Document Object Model
DQL	Doctrine Query Language
DTO	Data Transfer Object
ECTS	European Credit Transfer and Accumulation System
EXIF	Exchangeable Image File Format
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
MVC	Model-View-Controller
NFS	Network File System
ORM	Object-relational mapping
REST	Representational State Transfer
SDK	Software Development Kit
SPOF	Single point of failure
SSH	Secure Shell
UI	User interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VCS	Version Control System

Příloha **B**

Slovník pojmů

Backend Komponenta aplikace, jež je zodpovědná za obchodní logiku a za ukládání dat do databáze.

Bundle Bundle je doplňková softwarová komponenta, kterou lze volitelně nainstalovat do Symfony aplikace.

Clonové číslo Clonové číslo je poměr ohniskové vzdálenosti optické soustavy a průměru vstupní čočky.

EXIF Exif, neboli *Exchangeable image file format*, je specifikace pro formát metadat, vkládaných do souborů digitálními fotoaparáty. Může obsahovat např. informace o modelu fotoaparátu, datumu a času pořízení snímku, clonové číslo, ...

Expoziční čas Expoziční čas je doba, po kterou je závěrka fotoaparátu otevřena a umožňuje tak světlu dopadat na obrazový snímač nebo film ve fotoaparátu.

Framework Framework je software, který slouží jako podpora při programování a vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny API, podporu pro návrhové vzory nebo doporučené postupy při vývoji.

Frontend Komponenta aplikace, který bývá zpřístupněna přímo koncovému uživateli. Ten ji ovládá pomocí GUI.

ISO	V kontextu fotografování se jedná o zkratku pro označení číselné citlivosti filmu/snímače. Čím vyšší toto je, tím více světla je schopný film/snímač pohltit a fotoaparát tak může exponovat v horších světelných podmínkách s kratším časem.
JavaScript	Multiplatformní, objektově orientovaný, událostmi řízený skriptovací jazyk.
macOS	Operační systém vyvíjený společností Apple.
Ohnisková vzdálenost	Ohnisková vzdálenost nebo obrazová vzdálenost je vzdálenost čočky nebo zakřiveného zrcadla od jejich ohniska.
Open-source	Open-source software, neboli *otevřený software*, je počítačový software s otevřeným zdrojovým kódem, který může být využíván a dále šířen spolu s přiloženou licencí.
PHP	Hypertext Preprocessor, skriptovací programovací jazyk.
Single point of failure	Taková část systému, která – pokud selže – způsobí zastavení celého systému.
Twig	Šablonovací jazyk pro PHP vycházející z HTML
Vodoznak	Vkládání vodoznaku je technika, která do digitálního dokumentu vkládá dodatečnou informaci tak, že je obtížné ji najít či odstranit. Fotografové jako vodoznak s oblibou volí své logo.
YAML	Jazyk pro serializaci dat honě využívaný jako formát konfiguračních souborů.

Příloha C

Konfigurační soubory vývojového prostředí

```
1  FROM node:14-alpine
2
3  # set working directory
4  WORKDIR /var/www/app
5
6  # add `/var/www/app/node_modules/.bin` to $PATH
7  ENV PATH /var/www/app/node_modules/.bin:$PATH
8
9  # install and cache app dependencies
10 COPY package.json /var/www/app/package.json
11 RUN yarn install
12 RUN yarn global add @vue/cli
13
14 # start app
15 CMD ["yarn", "serve"]
```

Ukázka kódu C.1: Konfigurační soubor Dockerfile definující podobu kontejneru *php* pro chod *frontendové* části aplikace

```
FROM php:7.3-fpm

RUN apt-get update

RUN apt-get install -y zlib1g-dev libpq-dev git \
        libicu-dev libxml2-dev libzip-dev \
        libpng-dev libjpeg-dev libfreetype6-dev \
    && docker-php-ext-configure intl \
    && docker-php-ext-install intl \
    && docker-php-ext-configure pgsql -with-pgsql=/usr/local/pgsql \
    && docker-php-ext-install pdo pdo_pgsql pgsql \
    && docker-php-ext-install zip xml \
    && docker-php-source delete \
    && docker-php-ext-configure gd \
        --with-freetype-dir=/usr/include/ \
        --with-jpeg-dir=/usr/include/ \
    && docker-php-ext-install gd

RUN docker-php-source extract \
    && pecl install xdebug redis \
    && docker-php-ext-enable xdebug redis \
    && docker-php-source delete

RUN pecl install apcu
RUN echo "extension=apcu.so" > /usr/local/etc/php/conf.d/apcu.ini

# Disable PHP memory limit
RUN echo "memory_limit = -1" > /usr/local/etc/php/conf.d/20-extra.ini

# Composer
RUN curl -sS https://getcomposer.org/installer \
    | php -- --install-dir=/usr/local/bin --filename=composer

# Symfony
RUN curl -sS https://get.symfony.com/cli/installer | bash \
        && mv /root/.symfony/bin/symfony /usr/local/bin/symfony

# Timezone
RUN rm /etc/localtime \
    && ln -s /usr/share/zoneinfo/Europe/Prague /etc/localtime \
    && date

RUN rm -rf /tmp/*

CMD ["php-fpm", "-F"]

WORKDIR /var/www/app
```

Ukázka kódu C.2: Konfigurační soubor Dockerfile definující podobu kontejneru *php* pro chod *backendové* části aplikace

```
server {
    listen 80;
    server_name web;
    root /var/www/app/public;

    location / {
        try_files $uri /index.php$is_args$args;
    }

    location ~ ^/index\.php(/|$) {
        fastcgi_pass unix:/var/run/php7.3-fpm.sock;
        fastcgi_pass php:9000;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        fastcgi_param DOCUMENT_ROOT $realpath_root;
        fastcgi_buffer_size 128k;
        fastcgi_buffers 4 256k;
        fastcgi_busy_buffers_size 256k;
        internal;
    }

    location ~ \.php$ {
        return 404;
    }

    error_log /var/log/nginx/project_error.log;
    access_log /var/log/nginx/project_access.log;
}
```

Ukázka kódu C.3: Konfigurační soubor webového serveru

```bash
#!/usr/bin/env bash

OS=$(uname -s)

if [ $OS != "Darwin" ]; then
  echo "This script is macOS-only. Please do not run it on any other Unix."
  exit 1
fi

if [[ $EUID -eq 0 ]]; then
  echo "This script must NOT be run with sudo/root." 1>&2
  exit 1
fi

if ! docker ps >/dev/null 2>&1; then
  echo "== Waiting for docker to start..."
fi

open -a Docker

while ! docker ps >/dev/null 2>&1; do sleep 2; done

echo "== Stopping running docker containers..."
docker-compose down >/dev/null 2>&1
docker volume prune -f >/dev/null

osascript -e 'quit app "Docker"'

echo "== Resetting folder permissions..."
U=$(id -u)
G=$(id -g)
sudo chown -R "$U":"$G" .

echo "== Setting up nfs..."
LINE="/System/Volumes/Data -alldirs -mapall=$U:$G localhost"
FILE=/etc/exports
sudo cp /dev/null $FILE
grep -qF -- "$LINE" "$FILE" || sudo echo "$LINE" \
        | sudo tee -a $FILE > /dev/null

LINE="nfs.server.mount.require_resv_port = 0"
FILE=/etc/nfs.conf
grep -qF -- "$LINE" "$FILE" || sudo echo "$LINE" \
        | sudo tee -a $FILE > /dev/null

echo "== Restarting nfsd..."; sudo nfsd restart

echo "== Restarting docker..."; open -a Docker

while ! docker ps >/dev/null 2>&1; do sleep 2; done

echo "SUCCESS! Now go run your containers"
```

Ukázka kódu C.4: Shell skript pro vytvoření NFS serveru

```yaml
version: '3'
services:
    php:
        build: php-fpm
        ports:
            - '9000:9000'
        volumes:
            - nfsmount:/var/www/app
            - ./php-fpm/php.ini:/usr/local/etc/php/php.ini
            - ./php-fpm/xdebug.ini:/usr/local/etc/php/conf.d/xdebug.ini
        depends_on:
            - postgres_dev
            - postgres_test
    nginx:
        image: nginx:latest
        ports:
            - '8000:80'
        volumes:
            - nfsmount:/var/www/app
            - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
        depends_on:
            - php
    postgres_dev:
        image: postgres:alpine
        ports:
            - '5432:5432'
        environment:
            POSTGRES_USER: dev
            POSTGRES_PASSWORD: pass
            POSTGRES_DB: devdb
    postgres_test:
        image: postgres:alpine
        ports:
            - '5433:5432'
        environment:
            POSTGRES_USER: test
            POSTGRES_PASSWORD: pass
            POSTGRES_DB: testdb
volumes:
    nfsmount:
        driver: local
        driver_opts:
            type: nfs
            o: addr=host.docker.internal,rw,nolock,hard,nointr,nfsvers=3
            device: ':/System/Volumes/Data/${PWD}/..'
```

Ukázka kódu C.5: Konfigurační soubor Docker Compose pro spuštění Docker kontejnerů využívajících NFS pro propojení sdílených svazků

```yaml
image: projects.jagu.cz:8081/photographix/backend:latest

variables:
    APP_ENV: test
    POSTGRES_DB: testdb
    POSTGRES_USER: test
    POSTGRES_PASSWORD: pass

stages:
    - build
    - static tests
    - dynamic tests
    - deploy

download vendors:
    stage: build
    script:
        - composer install
        - php vendor/bin/codecept build
    artifacts:
        expire_in: 1 day
        paths:
            - vendor
            - var
            - tests/_support/_generated

lint:
    stage: static tests
    script:
        - php bin/console lint:yaml *.yml config/ src/ translations/ tests/ --ansi
        - php bin/console lint:twig templates/ --ansi
        - php bin/console lint:container --ansi
        - php vendor/bin/parallel-lint src tests/unit tests/api --colors

code style:
    stage: static tests
    script:
        - php vendor/bin/phpcs --standard=ruleset.xml --encoding=utf8
                    --extensions=php --colors -sp src tests/unit tests/api

static analysis:
    stage: static tests
    services:
        -   name: postgres:alpine
            alias: postgres_test
    script:
        - php vendor/bin/phpstan analyse -c phpstan.test.neon --level max src --ansi

# pokračování konfiguračního skriptu na další stránce ...
```

Ukázka kódu C.6: Konfigurační soubor pro kontinuální integraci a kontinuální dodávku pro nástroj GitLab CI/CD *backendové* části projektu (1. část)

```yaml
50 # ... pokračování konfiguračního skriptu z předchozí stránky
51
52 unit tests:
53     stage: dynamic tests
54     services:
55         -   name: postgres:alpine
56             alias: postgres_test
57     script:
58         - php vendor/bin/codecept run unit --ansi
59     artifacts:
60         name: unit tests
61         when: on_failure
62         expire_in: 1 day
63         paths:
64             - tests/_output
65
66 API tests:
67     stage: dynamic tests
68     services:
69         -   name: postgres:alpine
70             alias: postgres_test
71     script:
72         - php vendor/bin/codecept run api --ansi
73     artifacts:
74         name: API tests
75         when: on_failure
76         expire_in: 1 day
77         paths:
78             - tests/_output
79
80 code coverage:
81     stage: dynamic tests
82     services:
83         -   name: postgres:alpine
84             alias: postgres_test
85     variables:
86         PHP_ENABLE_XDEBUG: "true"
87     script:
88         - php vendor/bin/codecept run --coverage --silent --coverage-html --no-colors
89     artifacts:
90         name: API tests
91         expire_in: 1 day
92         paths:
93             - tests/_output/coverage
94
95 deploy to api.photographix.jagu.cz:
96     image: projects.jagu.cz:8081/infrastructure/containers-ci/deployer:latest
97     stage: deploy
98     before_script:
99         - eval $(ssh-agent -s)
100        - ssh-add <(echo "$SSH_PRIVATE_KEY")
101        - mkdir -p ~/.ssh
102        - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
103    script:
104        - cd .deployer
105        - dep deploy dev -vv
106    only:
107        - master
```

Ukázka kódu C.7: Konfigurační soubor pro kontinuální integraci a kontinuální dodávku pro nástroj GitLab CI/CD *backendové* části projektu (2. část)

```php
<?php declare(strict_types=1);

namespace Deployer;

require 'recipe/common.php';

// Project repository
set('repository', 'https://...@gitlab.jagu.cz/photographix/backend.git');
set('allow_anonymous_stats', false);

// Hosts
host('dev')->hostname('api.photographix.jagu.cz.internal')
  ->set('deploy_path', '~/site')->set('branch', getenv('CI_BUILD_REF_NAME'))
  ->user('web')->port(22)->addSshOption('StrictHostKeyChecking', 'no');

set('shared_dirs', [
  'var/log',
  'var/sessions',
  'var/storage'
]);
set('shared_files', [
  '.env',
]);
set('writable_dirs', ['var']);
set('migrations_config', '');

set('bin/console', function () {
  return parse('{{bin/php}} {{release_path}}/bin/console --no-interaction');
});

desc('Migrate database');
task('database:migrate', function () {
  $options = '--allow-no-migration';
  if (get('migrations_config') !== '') {
    $options = sprintf(
      '%s --configuration={{release_path}}/{{migrations_config}}',
      $options
    );
  }
  run(sprintf('{{bin/console}} doctrine:migrations:migrate %s', $options));
});

desc('Migrate database');
task('database:migrate', function () {
  run('{{bin/console}} doctrine:migrations:migrate');
});

desc('Clear cache');
task('deploy:cache:clear', function () {
  run('{{bin/console}} cache:clear --no-warmup');
});

desc('Warm up cache');
task('deploy:cache:warmup', function () {
  run('{{bin/console}} cache:warmup');
});

desc('Deploy project');
task('deploy', [
  'deploy:info', 'deploy:prepare', 'deploy:release', 'deploy:update_code',
  'deploy:shared', 'deploy:vendors', 'deploy:writable', 'deploy:cache:clear',
  'deploy:cache:warmup', 'database:migrate', 'deploy:symlink', 'cleanup',
]);
after('deploy', 'success');
```

Ukázka kódu C.8: Ukázka konfiguračního souboru nástroje *Deployer* využívaného k nasazení *backendové* části aplikace na cílové prostředí

Příloha D

Ukázka programátorské dokumentace

Entities and repositories

All entities can be created manually, but it is more comfortable to use Maker bundle. Due to custom project directory structure it is necessary to have `MAKER_NAMESPACE` environment variable set.

Let's describe basic usage with following example.

> Programmer wants to create new entity `Foo`. This entity belongs to the `Bar` *business group*. So we want to *generate* new entity `App\Model\Business\Bar\Entity\Foo` and repository `App\Model\Business\Bar\Repository\FooRepository` corresponding to the entity.

Maker bundle namespace

At first, we have to initialize `MAKER_NAMESPACE` environment like this:

```
export MAKER_NAMESPACE=\\App\\Model\\Business\\Bar
```

Now *Maker bundle* knows where to store generated entity.

Doctrine entities mapping

We have to tell Symfony, where new entities will be stored using following snippet to `config/packages/doctrine.yaml` config file.

```
doctrine:
    ...
    orm:
        ...
        mappings:
            ...
            Bar:
                type: annotation
                dir: '%kernel.project_dir%/src/Model/Business/Bar/Entity'
                prefix: 'App\Model\Business\Bar\Entity'
                is_bundle: false
```

Entity and repository generation

Let's generate entity with corresponding repository.

```
php bin/console make:entity
```

Name of the newly created entity will be `Foo`.

Follow instructions and add some attributes.

> If necessary to create an attribute referencing entity from another namespace, fully qualified name of the target entity is required.

Obrázek D.1: Úryvek programátorské dokumentace `README` *backendové* části projektu popisující postup vytváření databázové entity přes příkazovou řádku

Příloha E

Obsah přiloženého média

```
README.txt ........................... stručný popis obsahu média
src
  thesis ........................ zdrojové soubory textu práce ve formátu LaTeX
DP_Erben_Marek_2021.pdf ............... Text práce ve formátu PDF
```