FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Backend administrace e-shopu
Student:	Alois Kouba
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem této práce je ve spolupráci se studentem Tomášem Hojkem realizovat funkční backend webové aplikace administrace e-shopu. Tento backend musí nabízet vhodné rozhraní pro nově vznikající frontend, který souběžně vytváří Martin Dvořák a Jan Babák. Důraz je kladen na týmovou spolupráci a vhodnou metodiku vývoje.

Postupujte v těchto krocích:
1. Analyzujte současné řešení e-shopové administrace a předešlé práce věnující se vývoji daného projektu.
2. Společně s ostatními členy navrhněte vhodnou metodiku vývoje a spolupráce. Při návrhu přihlédněte k možný problémům týmové spolupráce.
3. Vhodným způsobem proveďte rozdělení potřebné práce a realizujte dílčí návrhy částí budoucího backendu.
4. Konzultujte návrh s frontendovou částí týmu.
5. Implementujte výsledný návrh.
6. Při realizaci nezapomeňte na důkladné testování, volte také vhodné testování při návrzích rozhraní pro frontendovou část projektu.
7. Zhodnoťte výsledné řešení, navrhněte úpravy do budoucna.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 21. ledna 2022 v Praze.

Bakalářská práce

BACKEND E-SHOPU

Alois Kouba

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Hunka
9. května 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

V první řadě bych chtěl poděkovat panu Ing. Jiřímu Hunkovi nejen za rady a čas, které mi v průběhu tvorby práce poskytl, ale také za možnost toto téma si zvolit. Dále bych chtěl poděkovat kolegům Bc. Tomášovi Hojkovi, Martinovi Dvořákovi a Janu Babákovi za spolupráci při implementaci. Dále bych chtěl poděkovat také Ing. Janu Matouškovi za pomoc při interpretaci funkčnosti původního řešení administrace.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2022 .....

Abstrakt

Tato práce se zabývá vývojem nového backendu systému administrace e-shopu. Tento vývoj probíhá v týmu se třemi dalšími kolegy. Nejprve se práce zabývá analýzou práce v týmu a vybráním vhodné metodiky vývoje. Následuje analýza již existujícího řešení administrace od společnosti Jagu s.r.o. Na základě této analýzy je vytvořen návrh API nového backendu, který je následně realizován v jazyce PHP. Výsledek je zároveň otestován pomocí automatických testů.

Klíčová slova eshop, API, backend, PHP, Symfony, tým

Abstract

This thesis deals with a complete development of a e-shop administration backend. This development is done in a team with another three colleagues. First, this thesis deals with the analysis of working in a team and choosing the correct development methodology. Next, the existing solution to e-shop administration from Jagu s.r.o. is analyzed. Based on this analysis, a suitable API design for the new project is created and subsequently implemented in PHP programming language. The result is also tested by automatic tests.

Keywords eshop, API, backend, PHP, Symfony, team

Seznam zkratek

API	Application Programming Interface
MVC	Model View Controller
JSON	JavaScript Object Notation
REST	Representational state transfer
XML	Extensible markup language
SQL	Structured query language
DBAL	Database Abstraction Layer
ORM	Object Relational Mapper
DQL	Doctrine query language
DTO	Data transfer object
HTTP	Hyper Text Transfer Protocol

Úvod

Internet je fenomén, který tu s námi v České republice je již od roku 1992. Ač byl původně omezený technologicky pomalým připojením i společenskými normami, v posledních letech jeho vývoj zrychluje. V některých oblastech internetové služby stále více vytlačují tradiční řešení. Jednou z takových oblastí je nakupování. Pro zákazníka je zkrátka pohodlnější nakoupit několika kliknutími na internetu, než cestovat do kamenného obchodu. Tato revoluce ale není jen tak – pro fungování tohoto modelu je potřeba vyvíjet kvalitní software, který zajistí možnost využívat moderní technologie i běžnému uživateli. Těchto e-shopů existuje nepřeberné množství – od velkých hráčů jako Alza.cz nebo Mall.cz, až po ty nejmenší internetové obchody. Pro dobré fungování těchto aplikací je potřeba mít možnost je efektivně spravovat.

Toto téma závěrečné práce jsem si zvolil, protože je velmi aktuální a prakticky zaměřené. Od práce si slibuji nabrání zkušeností s vývojem rozsáhlého webového rozhraní.

Na tuto práci navazují kolegové Jan Babák a Martin Dvořák, a to konkrétně souběžným vývojem frontendové aplikace administrace internetového obchodu. Dále na ní navazuje diplomová práce kolegy Bc. Tomáše Hojka, který se společně se mnou zabývá vývojem backendu pro tuto aplikaci.

Bakalářské práce je rozdělena do jednotlivých částí podle modelu vývoje softwaru, a to analýza, návrh, realizace a na konec testování. Vzhledem k velikosti aplikace se předpokládá iterativní vývoj a toto uspořádání tak není nutně chronologické.

Cíle

Jedním z řešení pro správu je systém administrace, dále stará administrace, pro e-shop www.stylka.cz a dalších e-shopů od společnosti Jagu s.r.o., dále Jagu. Vývoj tohoto systému probíhá od jeho prvního nasazení roku 2009 a vznikl tedy nápad vytvořit zcela nový systém.

Hlavním cílem této práce je vytvořit funkční webové rozhraní pro paralelně vyvíjenou webovou aplikaci administrace e-shopu.

Teoretická část se zaměřuje hlavně na analýzu staré administrace, která bude tímto nahrazena. Vzhledem k tomu, že vývoj probíhá v dvoučlenném týmu a stejná situace nastává u frontendové části, proběhne dále analýza metodiky vývoje a spolupráce. Zde bude nutné přihlédnout k možným problémům týmové spolupráce. Ve finále proběhne samotný návrh rozhraní. Vzhledem ke složitosti projektu budou návrhy probíhat iterativně.

Praktická část se bude zabývat především implementací navrženého řešení. Dalším požadavkem je důkladné testování. V průběhu práce jsou požadovány konzultace s frontendovou částí týmu.

Kapitola 1
Analýza

Analýza je nedílnou součástí vývoje software. Ačkoliv se může na první pohled zdát, že analýza je jen malým a dokonce zanedbatelným krůčkem ve vývoji softwaru, opak je pravdou. Analýza hluboce ovlivňuje všechny další kroky vývoje. Chybně provedené analýza může mít za následek vytvoření aplikace, která zcela nesplňuje vytyčené požadavky.

V této kapitole se budu nejprve zabývat analýzou práce v týmu a následně analýzou současného řešení administrace. Nakonec zpracuji seznam funkčních a nefunkčních požadavků, který bude využit dále při návrhu.

1.1 Analýza vývoje v týmu

Jak jsem zmínil v úvodu, na tuto práci navazují ve svých závěrečných pracech Tomáš Hojek, Martin Dvořák a Jan Babák. Společně budeme pracovat v týmu na dosažení našeho společného cíle – vytvoření funkčního nového systému administrace e-shopu. Je proto nutné se podívat, jak vlastně práce v týmu na nějakém programu probíhá, vyhodnotit možné problémy a rizika a pokusit se jim vyhnout.

Práce v týmu je velmi důležitým nástrojem pro vývoj jakékoliv rozsahově větší aplikace. Jako každý nástroj je ale potřeba i týmovou spolupráci správně použít. V opačném případě může být snadno spíše brzdou než pomocníkem. V této sekci nejprve definuji, co to tým vlastně je, následně se budu zabývat prostředky, které vývoj v týmu usnadňují, a nakonec se budu zabývat samotnou metodikou vývoje.

„*Tým = skupina spolupracujících lidí, kteří mají společné, časově ohraničené cíle a při plnění těchto cílů jsou na sobě závislí.*“ [1]

Zároveň podle [2] lze anglické slovo team volně přeložit jako „Společně všichni dosáhneme více“, a to podle čtyř klíčových slov:

- **T** Together
- **E** Everybody
- **A** Achieves
- **M** More

1.1.1 Problémy vývoje v týmu

Podle článku [3] až 60% týmů selhává ve smyslu nedosažení svého cíle. V následující pasáži některé z těchto obecných důvodů konkretizuji s ohledem na vývoj software a i konkrétně tohoto

projektu.

- **Nedostatek plánování**

  Plánování je při vývoji software kritické. I když pracuje člověk na projektu sám, rozvržení práce dopředu mnohdy ušetří čas. V našem případě se jedná dokonce o dva týmy. Zde je klíčové správně naplánovat práci týmů nejen jako takových, ale i v závislosti na sobě. Zároveň je systém velice rozsáhlý a je tedy vhodné si přesně určit cíle, co všechno chceme zpracovat.

- **Nedostatek prostředků**

  Prostředkem při vývoji software můžou být například peníze, čas nebo zkušenosti. V našem případě jsou největšími surovinami čas a zkušenosti. Toto se váže k prvnímu bodu ohledně plánování. Je potřeba správně odhadnout čas potřebný k vytvoření výsledného produktu právě s ohledem na dostupné prostředky. Pokud je prostředků nedostatek, může to mít za následek buď úplné selhání projektu, nebo přinejmenším jeho zdržení.

- **Nedostatek jasnosti**

  Článek popisuje nejasné vztahy mezi členy týmu, já bych k tomuto bodu ale zahrnul i komunikaci jako takovou. Ta by měla být jasná, účelná a efektivní. Bez efektivní komunikace dochází k plýtvání prostředků pro vývoj. Buď dochází k prosté ztrátě času, nebo v horším případě může dojít i k nedorozuměním při vývoji. Komunikace se dá rozdělit na několik druhů: [4]

  1. **Osobní**

     Nejúčinnější způsob výměny informací. Zapojuje kromě mluveného slova i neverbální komunikaci. Díky tomu umožňuje velmi efektivní řešení problémů. Protože je ale okamžitá, bez přípravy na schůzku se snadno opomenou některé části sdělení, které člověk chtěl probrat. Měla by tedy být podpořena i dalšími typy komunikace. Dalším problémem je neefektivita – aby se tým sešel, musí obětovat cenný čas cestě na schůzku. Toto není problém u klasického týmu na nějakém pracovišti, ale v případě školního projektu je to nepříjemná brzda.

  2. **Telefonická**

     Zde se nejedná pouze o telefonování, ale i o videohovory prostřednictvím konferenčních videohovorů. Tento způsob komunikace se v poslední době rychle rozvíjí kvůli pandemii koronaviru. Jeho výhody jsou jasné – velmi šetří čas díky možnosti se připojit z domova a zároveň zachovává největší výhody osobního kontaktu díky pozůstavší určité část neverbální komunikace. Současná technologie ale stále plně nevynahradí osobní kontakt a telefonická komunikace by tedy neměla být jediným nástrojem pro komunikaci v týmu.

  3. **Chat, Email**

     Také se dá klasifikovat jako asynchronní komunikace. Není tak nutné být k dispozici ve stejný moment jako druhá strana, stačí si poslat zprávu. Zároveň zde informace typicky zůstávají uložené a není problém je zpětně získat. Další výhodou je fakt, že člověk při psaní zprávy aktivně nad danou problematikou přemýšlí. Velmi často tak přijde na řešení při pokusu o popsání problému v textu. Jako poslední výhodu bych vypíchl kompletnost této komunikace. Zprávu je možno dopsat celou, je zde čas na přečtení si dané zprávy před odesláním a doplnění případných chybějících informací.

  4. **Web**

     Zahrnuje různá diskuzní fóra, zdroje na internetu a podobně. Jedním z nejznámějších takových stránek ve světě IT je web `www.stackoverflow.com`, který je diskuzním fórem, zabývajícím se problémy při programování. Z více formálních webových komunikačních kanálů bych zmínil specializované aplikace pro správu projektů. Ty umožňují efektivně rozdělovat práci členům týmu a jsou tak nedílnou součástí práce na projektu v týmu.

Obecně je zřejmé, že nejefektivnější kombinací pro komunikaci je mix výše zmíněných druhů komunikace. V našem týmu jsme používali všechny 4 způsoby komunikace – osobní schůzky, videokonference, chat i webové nástroje pro správu projektů. Konkrétními nástroji se budu zabývat v další podsekci.

1.1.2 Podpůrné nástroje pro vývoj v týmu

Je tedy jasné, že problémy v týmové spolupráci je potřeba nějak řešit. Konkrétně pro vývoj software existuje spoustu nástrojů, které toto usnadňují. Vzhledem k tomu, že backend e-shopu zpracováváme pro Jagu, byly nám doporučeny některé nástroje, které společnost již využívá. Těchto nástrojů je více a ne všechny jsou vhodné pro náš případ užití. Nástroji, které budeme využívat jsou Redmine, Gitlab a Slack. Posledním nástrojem, který jsme pro práci v týmu využívali byl Google Meet.

Rád bych ještě lehce zmínil nástroj Sentry [5]. Protože ho nebudeme využívat pro náš vývoj, tak ho nechci popisovat detailně, ale pro případný další vývoj již může být užitečný. Pokud dojde k reálnému nasazení projektu na server, Sentry může zajistit detailní analýzu chyb v tomto provozu.

1.1.2.1 Redmine

„*Redmine je flexibilní webová aplikace pro správu projektů. Je napsaný na frameworku Ruby on Rails, je cross-platformní a cross-databázový. Redmine je open source a vydaný pod GNU GPL v2.*“ [6]

Redmine umožňuje v rámci projektů vytvářet úkoly, důkladně je dokumentovat a zpracovávat. Velmi tím usnadňuje komunikaci mezi členy týmu. Není již nutné ztrácet čas mluvením o rozdělení úkolů mezi členy týmu, protože je vše jasně vidět na jednom místě.

Pro naše účely je nicméně mnohem důležitější z hlediska mezitýmové komunikace. Protože jsme v týmu pro backend jen dva, komunikace ohledně rozdělení úkolů není příliš náročná, za to komunikace s frontendovou částí týmu je klíčová. Redmine tedy předává informaci ohledně postupu vývoje podle otevřených i uzavřených úkolů.

Jak jsem již psal výše, Redmine je aplikace s otevřeným zdrojovým kódem. Toto se dá považovat za velkou výhodu. Zjednodušeně to znamená, že je možné aplikaci vzít a používat pouze za cenu provozu. Toto je výhoda pokud se Redmine rozhodnou používat lidé zabývající se vývojem software – rozšířit „svojí“ verzi Redmine není velkým problémem. Díky této vlastnosti existují i různé hostingové firmy, které Redmine poskytují a zajistí plynulý běh aplikace pro své zákazníky.

Jagu má svoji vlastní instanci Redmine, která je dostupná na konkrétní URL adrese, a kterou budeme pro vývoj používat.

1.1.2.2 Gitlab

„*Gitlab je DevOps platforma, dodaná jako jedna aplikace. Toto dělá Gitlab unikátním a vytváří přímočarý průběh práce na softwaru a umožní firmám opustit složité toolchainy.*“ [7]

„*DevOps platforma kombinuje možnost vyvíjet, zabezpečit a provozovat software v jediné aplikaci takže všichni – od manažera po vývojáře – můžou efektivněji pracovat společně.*“ [8]

Gitlab je tedy založený na systému Git. Hlavní funkcí Gitu je verzování projektů. Na rozdíl od mnoha podobných systémů umožňuje bezbolestnou práci více lidí v týmu, a to díky možnosti větvení. Není tedy potřeba zamykat přístup ke zdrojovému kódu když na něm jeden člen týmu pracuje – stačí si vytvořit novou větev a výsledek spojit. [9]

Hlavní důvody pro použití Gitlabu jsou dva:

1. Jagu Gitlab také používá a provozuje i svojí vlastní Gitlab doménu.

2. Gitlab se používá pro verzování v rámci FITu. Mám s ním již tedy jisté zkušenosti.

1.1.2.3 Slack

„*Slack je pracovní komunikační nástroj, jedno místo pro posílání zpráv, nástrojů a souborů.*“ [10]

Podobně jako Redmine a Gitlab, i Slack je používán Jagu, a to samo o sobě je dost dobrým důvodem tuto aplikaci používat. Slack ve své základní verzi je zdarma. Tuto verzi jsme používali, protože splňuje všechny požadavky námi kladené na aplikaci pro textovou komunikaci v týmu. Bezplatný Slack dává zákazníkům přístup k posledním 10000 zprávám, omezuje počet integrací na 10 a omezuje možnosti administrace systému. Toto se může zdát jako velká nevýhoda – relativně brzy nebudeme moci načítat naše týmové zprávy. Naštěstí jsme malým týmem, a toto omezení nas tedy reálně příliš neomezuje.

Jak jsem psal výše, Slack splňuje všechny námi kladené požadavky na aplikaci pro textovou komunikaci. Co ale bezplatný Slack neumožňuje je možnost pořádat konferenční videohovory. Ty jsou velmi důležitým nástrojem. Kombinují výhody realtimové komunikace a možnost nebýt na schůzce fyzicky přítomen. Šetří tak tedy velmi čas.

1.1.2.4 Google Meet

„*Google zpřístupňuje videokonference na podnikové úrovni všem zájemcům. Každý, kdo má účet Google, teď může uspořádat online videokonferenci až pro 100 účastníků v délce až 60 minut.*“

„*Firmy, školy a další organizace mohou využít pokročilé funkce, včetně schůzek až pro 500 interních nebo externích účastníků a živých přenosů až pro 100 000 diváků v rámci domény.*“ [11]

Google Meet jsme zvolili hlavně kvůli jednoduchosti použití. K připojení stačí zařízení s připojením k internetu a nainstalovaným webovým prohlížečem – požadavek, který splní dnes již jakýkoliv počítač a nebo i chytré mobilní telefony. Google Meet sice má svou placenou verzi, ale vše, co potřebujeme umožňuje i bezplatná verze. Těmito vlastnostmi je možnost vytvořit videohovor pro více uživatelů a přenášet zvuk a obraz.

1.1.3 Metodika vývoje

Metodika vývoje je sada doporučení, která se zabývá otázkami, jak postupovat, do jakých fází rozdělit vývoj software, jaký je jejich výstup, kdo bude tyto výstupy vytvářet a v jakém rozsahu. Cílem je tedy efektivně vytvářet softwarové produkty. To může být obtížné, protože každý softwarový projekt je něčím specifický. Ať už je to velikostí nebo délkou trvání projektu, týmem který ho zpracovává, technologiemi, zákazníkem, atd... [12]

Skoro to tedy na první pohled vypadá, že když je každý projekt tak různorodý, tak se nemá smysl jimi zabývat. Opak je ale pravdou. Ačkoliv jsou projekty různorodé, nějaké společné části se najdou vždy. Když už ne přímo, tak alespoň myšlenky za různými postupy a metodami se nemění. Výhody: [12]

- Standardizace pracovních postupů
- Opakovatelnost
- Snadnější řízení projektu
  - Definice požadovaných artefaktů a jejich obsahu
  - Snadnější kontrola postupu v projektu
  - Sledování metrik
  - Snadnější odhadování

Samozřejmě nic není perfektní, a dá se tedy bavit i o nevýhodách používání nějaké metodiky vývoje: [12]

- Práce navíc
- Potřebný čas pro seznámení se s metodikou
- Přizpůsobení metodiky konkrétnímu projektu
- Některé metodiky jsou placené

Metodiky vývoje jsou různé. Každá metodika se zabývá nějakou částí vývoje software. Může se zabývat pouze pravidly na vysoké úrovni abstrakce, nebo i konkrétně a detailně zpracovávat určitý problém. Jakkoliv jsou metodiky různorodé, dá se je roztřídit podle základní klasifikace na metodiky klasické a metodiky agilní: [12]

1. **Klasické**

   Kladou velký důraz na přípravu před samotnou prací. Samotné implementaci předchází důkladná analýza, podrobné návrhy architektury a sběr a analýza požadavků. Až když je naprosto jasně vytyčen nejen cíl práce, ale i cesta jak ho dosáhnout, se začíná implementovat. Dalším zaměření je na tvorbu dokumentace.

   Jejich výhodou je jasný a metodický postup. Dále řeší širší spektrum problémů než agilní metodiky a obecně bývají propracovanější.

   Na druhou stranu ale můžou zvýšit zbytečný nárůst pracnosti a to neúměrně benefitům.

   Mezi klasické metodiky patří například „Unified Process“, „Modern Structured Analysis“ a „Rational Unified Process“. [12]

2. **Agilní**

   Agilní metodiky jsou v kontrastu s metodikami klasickými zaměřeny především na tvorbu výsledného produktu. Analýza a sběr požadavků před projektem je omezena na nutné minimum. Hlavním cílem je rychlá dodávka prototypu projektu, na který zákazník odpoví zpětnou vazbou. Zpětná vazba a komunikace se zákazníkem je tedy nedílnou součástí agilního vývoje software.

   Obecně je tedy výhodou rychlá doba dodání produktu. Nejlépe využitelné je tento způsob pro měnší projekty. Protože je komunikace nedílnou součástí agilního vývoje, je v neposlední řadě potřeba i spolupracující zákazník.

   Agilní metodiky obecně pokulhávají při práci ve velkých týmech, nebo pokud komunikace se zákazníkem (nebo i v týmu samotném) vázne.

   Příkladem agilní metodiky budiž „SCRUM“ nebo „Extrémní Programování“ [13]

   Při výběru metodiky vývoje je nutné zohlednit spoustu faktorů, jako například: [12]

- Velikost projektu
- Velikost a složení týmu
- Stabilita požadavků
- Zvyklosti a preference týmu a zákazníka

Zpracovávaný projekt je relativně komplikovaný. Zároveň je potřeba, aby zpracovával data pro více domén najednou. Ve skutečnosti je tak obsáhlý, že ani naše 4 bakalářské a diplomové práce neobsáhnou celou funkcionalitu staré administrace. Dalo by se tedy říct, že je vhodné zvolit klasickou metodiku. Opak je ale pravdou – vzhledem k velikosti našeho týmu by bylo velmi obtížné zpracovat komplexní analýzu staré administrace tak, aby se již návrh nemusel příliš pozměňovat. Vzhledem k tomu je pro nás tedy ideální zvolit agilní přístup.

Jak jsem zmínil výše, náš tým je relativně malý, což je signálem ke zvolení agilního přístupu.

Z hlediska stability požadavků je v našem případě vhodný opět agilní přístup. Důvod je prostý – protože je systém rozsáhlý a požadavky se analyzují iterativně, logicky se časem mění.

Z hlediska preference týmu se opět kloníme k agilnímu přístupu. Nakonec i sám zadavatel nám doporučil jít cestou spíše agilního přístupu.

Ačkoliv by se mohlo zdát, že bakalářská práce je ideálním projektem pro použití klasické metodiky pro její typické zaměření na rešerši a zpracování nějakého odborného tématu, v tomto případě tomu tak není. Požadavky kladené na tuto práci se vyvíjejí v čase, už jenom kvůli rozsáhlosti staré administrace a postupnosti zpracování. Výsledkem je tedy jednoznačné zvolení agilní metodiky vývoje. Není potřeba vybírat konkrétní metodiku, ale obecně se budeme zaměřovat spíše na co nejrychlejší vývoj software po částech. Je zde ale nutné vyzdvihnout jedno odchýlení od agilního přístupu. Ta spočívá ve tvorbě dokumentace. Podle [12] se tvorba dokumentace při agilním vývoji omezuje na minimum. Toto je ale nemyslitelné v případě vývoje backendu. Je životně důležité přesně definovat rozhraní, přes které frontend komunikuje s backendem. Tento požadavek je ještě umocněn faktem, že backend musí pracovat s již existující databází a některé požadavky, které by mohly přijít od frontendového týmu by nemusely být proveditelné, tudíž návrh by měl pocházet prvotně právě ze strany backendu.

1.1.4 Životní cyklus vývoje

Dále je třeba vybrat vhodný životní cyklus vývoje. Cykly dělíme na dvě hlavní kategorie: [12]

1. **Vodopád**

   Jedná se o vývoj takzvaně na jeden zátah. Nejdříve se provede sběr požadavků, analýza a návrh, následuje implementace a nakonec dodání a případná údržba. Tyto části na sebe navazují a mají pevně dané pořadí. Ačkoliv to reálná situace občas vyžaduje, teoretický model životního cyklu vývoje vodopád neumožňuje se při vývoji vracet k předchozímu kroku.

   Obecně je tedy vhodný spíše pro malé projekty, kde je rozsah řešené problematiky dobře zvládnutelný při zpracování najednou. Výhodou je přímočarost vývoje. Zákazník ma dobrý přehled o postupu projektu a odhadnout čas a pracnost projektu je jednodušší.

   Jak jsem zmínil výše, vodopád neumožňuje vracet se zpět, přičemž toto je v určitých situacích nevyhnutelné. Obecně se tedy vodopád nepoužívá moc často. [12]

2. **Iterativní**

   Iterativní vývoj probíhá, jak ostatně také název napovídá, postupně po menších částech. Každá z těchto částí, nebo-li iterací, se skládá z několika fází – sběr požadavků, analýza, návrh, implementace a testování. Jedná se tedy v podstatě o sekvenci několika menších „vodopádů“. Tento model životního cyklu vývoje je o poznání flexibilnější než vodopád, protože menší iterace umožňují rychlý sběr zpětné vazby od zákazníka a tedy i rychlejší změnu směru vývoje, pokud je to potřeba. Typicky je vhodný pro větší projekty, které jsou obtížné zpracovat najednou, a zpracování po částech je tehdy výhodnější.

   Jak jsem zmínil výše, velkou výhodou je flexibilita navíc oproti vodopádu. Další výhodou je jednodušší testování a hledání chyb, protože se v jeden moment zpracovává menší část kódu.

   Problémem může naopak být oproti vodopádu horší odhad pracnosti a sledování postupu zákazníkem. Toto je způsobeno předpokládanými změnami a úpravami požadavků za běhu.

V našem případě jde hlavně o čas, kterého můžeme snadno strávit prací mnohem více než bylo plánováno, ale v komerčním světě je toto ekvivalentem překročení plánovaného rozpočtu. [12]

Výběr životního cyklu vývoje je, podobně jako výběr metodiky vývoje, důležitou částí analýzy problémové domény. Hlavní věcí, kterou je potřeba zohlednit je rozhodnout, jestli se budou požadavky v průběhu vývoje projektu měnit, přibývat nebo ubývat. Pokud ne, je vhodné zvolit spíše vodopád. Důvod je prostý – obecně je téměř vždy lepší postupovat podle předem daného plánu. Pokud se požadavky nemění, je tento plán možné zkonstruovat hned na začátku vývoje a držet se jej. V našem případě ale jasně převážila potřeba požadavky za běhu měnit. Zpracovávaná doména je rozsáhlá a složitá, což velmi ztěžuje důkladnou analýzu. Tato analýza by tedy pravděpodobně trvala neúměrně dlouho a ve finále by stejně pravděpodobně přinesla potřebu změn v návrhu. Naší volbou je tedy ve finále iterativní způsob vývoje.

1.2 Současný stav

V současné době aplikace staré administrace funguje na systému OpenCart, který komunikuje s databází MySQL.

„*Opencart je open-source, online systém pro admistraci e-shopů. Je založený na PHP a používá MySQL databázi a HTML komponenty pro vytváření webových stránek nebo obchodů. Obchodníci můžou použít OpenCart pro vytvoření jejich online obchodu zadarmo, a OpenCart dodává veškeré potřebné nástroje pro interakci s prodávajícími, zobrazení produktů, přijmutí a zpracování objednávek a zařízení zákaznické podpory.*“ [14]

První nasazení tohoto systému se datuje k roku 2009. Od té doby systém prošel mnohými rozšířeními, které vylepšily jeho funkčnost a které reagovaly na požadavky klientů na další vývoj. Tyto změny byly implementovány postupně. Aplikace tedy funguje, ale jakékoliv rozšíření je prakticky neproveditelné kvůli složitosti kódu. `Opencart` je postavený na vícevrstvé architektuře, ale jedná se o monolitickou aplikaci. To, co vidí uživatel a backend je tedy součástí jedné aplikace. Tento přístup má své výhody, jako například: [15]

- Prvotní vývoj monolitické aplikace je relativně rychlý a jednoduchý k nasazení, a to hlavně pro menší aplikace.
- Jednoduše se testuje. Protože se aplikace nedělí na komponenty, testuje se celá aplikace a ne tyto komponenty separátně.

Nicméně, hlavně pro větší aplikace tento přístup může způsobit problémy, jako například: [15]

- Velké množství provázaného kódu. Ve výsledku může být aplikace obtížná na pochopení a změnu, a to hlavně pro nové členy vývojového týmu.
- Komplikované průběžné nasazování projektu. Při změně v jedné části aplikace se musí provést deploy celé aplikace znovu. Také může kvůli provázanosti kódu přestat fungovat i nedotčená část aplikace.
- Rozšíření aplikace může vyžadovat rozsáhlé změny ve zbytku kódu. Zároveň už jen analýza přidání nějaké funkcionality je velice obtížná kvůli vysoké provázanosti jednotlivých částí aplikace.
- Kvůli obtížné rozšiřitelnosti způsobuje dlouhodobé zamknutí se s daným balíčkem technologií. S monolitickou aplikací může být obtížné průběžně aktualizovat komponenty. Pokud o modernizaci takové aplikace může snadno vyžadovat vytvoření úplně nové aplikace, což je ostatně přesně to, co se děje v případě této administrace.

Podoba databáze odpovídá podobě aplikace. Také ona byla postupně rozšiřována a upravována podle požadavků na nové funkčnosti. Databáze svůj účel plní, ale v současné podobě určitě není ideálním příkladem databáze. Mezi zjištěné nedostatky patří například:

- Tabulky povětšinou neobsahují definované cizí klíče. Zde je potřeba zmínit, že tento problém nevznikl z důvodu neznalosti vývojářů – je to jednoduše artefakt použitého typu úložiště. Typ databáze, na kterém byl tehdejší systém `Opencart` vystavěn je nepodporuje. Toto způsobuje několik problémů:
  - V první řadě to znamená, že se nekontroluje na úrovni databáze konzistence dat. Na úrovni databáze lze tedy vložit záznamy, které se odkazují na neexistující entity. Zodpovědnost ohledně kontroly dat je tímto přenesena na aplikaci jako takovou, což nutně vnáší do databáze chyby, protože je velmi snadné pro programátora udělat chybu ve validaci ukládaných dat.
  - Podobný problém je u mazání, kde nelze použít kaskádovité mazání, které zajistí smazání všech souvisejících záznamů a udrží tak konzistenci databáze. V tomto případě se opět vše musí mazat ručně.
  - Zároveň, z hlediska databázové abstrakce to stěžuje práci s entitami v nějakém databázovém vztahu. Různé frameworky typicky abstrahují ID v databázi přímo do objektu reprezentujícího danou entitu v programovacím jazyce. Toto chování není bez cizích klíců dostupné.
- Databáze je velmi roztříštěná. Entity, které by mohly být sloučené jsou rozdělené. Příkladem budiž tabulka `oc_manufacturer` a `oc_manufacturer_description`. Druhá zmíněná tabulka obsahuje informace ohledně daného výrobce. Toto roztříštění je způsobeno předchozím vývojem aplikace, konkrétně lokalizací do různých jazyků. Přeložitelné části entit byly vyčleněny do svých tabulek, které jsou navíc identifikovány pomocí jazyka.
- Některé tabulky nejsou z různých důvodů využívané, to stejné platí pro některé atributy. Ve výsledku pouze znepřehledňují už tak nepřehlednou databázi.

Stará administrace je dále specifická tím, že se zabývá administrací více e-shopů najednou. Toto přidává velkou komplexitu do systému. Důvod je prostý – systém umožňuje přepisovat data jednotlivých entit pro různé domény nezávisle na sobě. Toto je řešeno pomocí tzn. „overridů“ – přepisů. Funkčností tohoto systému se budu zabývat později. Stará administrace je obecně rozsáhlá ale část, kterou zpracováváme se dá rozdělit na 4 hlavní logické části. Funkčnost těchto částí přiblížím v v podkapitole 1.3.1, zatím je pouze vyjmenuji:

- **Výrobci**
- **Kategorie**
- **Produkty**
- **Objednávky**

1.3 Požadavky

Nyní, po prvotním seznámení se se starou administrací je vhodné formulovat požadavky na nový software. Při vývoji jakékoliv aplikace je bezpodmínečně nutné přesně stanovit požadavky na daný projekt. Obecně je jejich účelem přesné vymezení funkčnosti vyvíjeného systému. Jedním rozdělením je rozdělení na funkční a nefunkční požadavky: [16]

funkční objasňují, jak systém pracuje a co musí umět. Příkladem můžou být následující požadavky:

- Systém bude umět vytvořit rezervaci.

- Systém bude umět upravit rezervaci.

nefunkční se zabývají obecnými charakteristikami systému. Příkladem budiž následující:

- Program bude napsán v jazyce PHP.

Pro úplnost bych chtěl ještě zmínit klasifikaci požadavků FURPS. Toto pokročilé rozdělení požadavků umožňuje lépe specifikovat oblast, kterou se daný požadavek zabývá. [16]

- F – Functionality – Zabývá se funkčností
- U – Usability – Zabývá se použitelností
- R – Reliability – Zabývá se spolehlivostí
- P – Performance – Zabývá se výkonností
- S – Supportability – Zabývá se rozšiřitelností, příp. udržitelností.

Při mém sběru požadavků budu primárně používat klasické rozdělení na funkční a nefunkční požadavky, ale ke každému požadavku připíši i rozdělení podle systému FURPS.

1.3.1 Funkční požadavky

- **FR1 Operace s daty – [F]**
  - Aplikace zabraňuje uložení špatných vstupů a udržuje konzistenci dat v databázi. Na tento bod je kladen důraz, neboť jak vyplynulo z analýzy databáze, v databázi nejsou přítomné základní validační mechanismy jako cizí klíče.
- **FR2 Funkčnost – [F]**
  - Aplikace kopíruje, případně v určitých místech rozšiřuje funkčnost předešlého systému OpenCart.
  - Protože je systém jako takový velice rozsáhlý, nebudeme ho zpracovávat celý. Část, kterou budeme zpracovávat v rámci této práce se bude zabývat hlavně 4 zmíněnými logickými celky. Jednotlivé požadavky na ně se dají shrnout do následujících bodů:
    - **FR 2.1 Výrobci – [F]**
      - Systém umožňuje vytvořit nového výrobce
      - Systém umožňuje upravit výrobce výchozí data výrobce
      - Systém umožňuje nastavit u výrobce různé hodnoty atributů na různých doménách, a to konkrétně pro atributy popis, meta popis, jméno výrobce, meta titulek, meta klíčová slova a SEO klíčové slovo[1]
      - Systém umožňuje získat data výrobce
      - Systém umožňuje výrobce smazat z databáze
      - Systém umožňuje výrobcům přiřazovat číslo pro řazení a následně je ve správném pořadí ukázat
    - **FR 2.2 Kategorie – [F]**
      - Systém umožňuje vytvořit novou kategorii
      - Systém umožňuje upravit výchozí data dané kategorie
      - Systém umožňuje nastavit u výrobce různé hodnoty atributů na různých doménách, a to konkrétně pro atributy popis, meta popis, jméno kategorie, meta titulek, meta klíčová slova a SEO klíčové slovo

[1] Jedná se o URL výrobce. www.stylka.cz/dany-vyrobce ukazuje na SEO klíč „dany-vyrobce“

- Systém umožňuje získat data kategorie
- Systém umožňuje kategorii smazat z databáze
- Systém umožňuje ke kategorii přiřadit parametry, které se následně nabízejí všem produktům v dané kategorii
- Systém umožňuje kategoriím přiřazovat číslo pro řazení a následně je ve správném pořadí ukázat
- Systém umožňuje přidávat ke kategoriím filtry, které se vážou na parametry a umožňují v e-shopu na základě těchto parametrů filtrovat produkty
- Kategorie existují v rámci stromové strutury, je tedy možné vytvářet a měnit stromy kategorií změnou rodičovské kategorie

* **FR 2.3 Produkty** – [**F**]
  - Systém umožňuje vytvořit nový produkt
  - Systém umožňuje upravit výchozí data daného produktu
  - Systém umožňuje nastavit u produktu různé hodnoty atributů na různých doménách, a to konkrétně pro atributy krátký popis, dlouhý popis, meta popis, jméno produktu, meta titulek, meta klíčová slova, SEO klíčové slovo, název propojeného produktu v systému Heuréky, klíč propojeného produktu v systému Heuréky a název kategorie daného produktu v rámci kategorií Google, Zboži a Heuréky
  - Systém umožňuje získat data produktu
  - Systém umožňuje produkt smazat z databáze
  - Systém umožňuje k produktu přiřadit a vyplnit hodnoty parametrů, které se k danému produktu vztahují
  - Systém umožňuje přidávat k produktům akce, což je v současnosti jediná používaná cesta pro změnu ceny na různých doménách a různým skupinám zákazníků.
  - Pro produkt lze vytvářet, mazat, upravovat a získávat varianty[2]
* **FR 2.4 Objednávky** – [**F**]
  - Byly podrobně analyzovány kolegou Bc. Tomášem Hojkem.

FR3 Rozšíření oproti předešlému systému

FR 3.1 Výchozí doména – [**F**]

Systém přidá možnost určit výchozí doménu pro výrobce, kategorie a produkty.

FR 3.2 Řazení – [**F**]

Systém opraví řazení výrobců a kategorií. Bude nejen přiřazovat pořadové číslo, ale v závislosti na tomto pořadovém čísle upraví i pořadové číslo zbytku výrobců a kategorií.

FR 3.3 Skupiny domén

Systém bude nově pro kategorie pracovat se skupinami domén. Tento systém se již ve starém systému dodržoval, avšak ne na úrovni aplikace, nýbrž na úrovni administrátorů. Skupiny domén jsou důležité pro stromy kategorií. Je potřeba zajistit, aby každá skupina domén měla svůj strom kategorií, a aby tyto stromy byly navzájem disjunktní

FR 3.4 Cizí měny u produktu – [**F**]

Systém bude nově umožňovat zadávat cenové hodnoty i v cizí finanční měně. Konkrétní finanční měna bude vždy nabízena podle nastavení konkrétní domény.

FR4 Testování – [**F**]

- Aplikace bude důkladně otestována

FR5 Autentizace a autorizace – [**F**]

[2] Např. různé velikosti bot

- Aplikace bude využívat autentizaci uživatele a následnou autorizaci k zabezpečení přístupu. Část API, které je v rámci této práce vyvíjeno, bude použito pro aplikaci administrace. Nebude tedy potřeba rozlišovat různé role uživatelů, jedinou možnou rolí bude admin. Zároveň je ale rozhraní potřeba zabezpečit, aby ho nikdo jiný než přihlášený admin nemohl využít.

1.3.2 Nefunkční požadavky

- **NR1 Programovací jazyk** – [S]
  - Aplikace bude napsána v programovacím jazyce PHP [17] s využitím frameworku Symfony [18]. Díky použití Symfony budeme k práci s databází používat framework Doctrine. [19]
- **NR2 Databáze** – [S]
  - Aplikace bude využívat již existující databázi MySQL.
- **NR3 Kompatibilita** – [S]
  - Aplikace bude kompatibilní s původním systémem administrace. Je nutné zajistit, aby nově přidané funkce nenarušily paralelní běh staré administrace.
- **NR4 Architektura** – [S]
  - Aplikace bude rozdělená na jednotlivé vrstvy podle vhodného návrhového vzoru
- **NR5 Frontendová část** – [U]
  - Bude paralelně zpracována kolegy Janem Babákem a Martinem Dvořákem.
- **NR6 Formát dat** – [U]
  - Aplikace pracuje s daty ve vhodném formátu pro komunikaci [20]
- **NR7 API** – [U]
  - Aplikace má API s vhodnou architekturou [21]
- **NR8 Business logika** – [R]
  - Business logika bude součástí backendu tam, kde to bude možné, a to hlavně z důvodu bezpečnosti
- **NR9 Dokumentace** – [U]
  - Rozhraní aplikace bude důkladně zdokumentováno.

Kapitola 2
Návrh

V této kapitole se budu zabývat návrhem nového systému pro systém administrace e-shopu. Nyní již máme seznam funkčních a nefunkčních požadavků a můžeme ho tedy aplikovat na návrh nové aplikace. Nakonec se také budu zabývat rozebráním použitých technologií.

2.1 Architektura

Architektury se týkají následující požadavky:

- **FR5 Autentizace a autorizace**

  Tento požadavek říká, že aplikace musí autentizovat uživatele a autorizovat jejich akce.

- **NR1 Programovací jazyk**

  Tento požadavek specifikuje jazyk PHP a framework Symfony.

- **NR2 Databáze**

  Tento požadavek říká, že databází je již existující MySQL

- **NR4 Architektura**

  Tento požadavek specifikuje, že má být využita vhodná architektura.

- **NR5 Frontendová část**

  Tento požadavek říká, že frontend bude samostatná komponenta.

- **NR8 Business logika**

  Tento požadavek říká, že business logika bude součástí backendu kde to bude možné.

Když shrneme tyto požadavky, výsledkem je zjištění, že se aplikace bude skládat ze 2 nebo 3 částí a to podle toho, zda bude část zabývající se autentizací a autorizací vyčleněna do vlastní komponenty. Jednou komponentou bude oddělený frontend, který není předmětem této práce a budu jej modelovat jako blackbox. Další jistou komponentou je samotné API. Toto je hlavním zaměřením této práce. Autentizaci a autorizaci ve své práci zpracovává Tomáš Hojek.

2.2 Backend administrace e-shopu

Úplně prvním krokem při návrhu API musí být zvolení architektury rozhraní. Těchto architektur je více, ale v dnešní době se nejčastěji používá architektonický styl REST. [22] Ten jsme zvolili po diskuzi v týmu a po validaci zadavatele. Toto rozhodnutí přímo řeší požadavek **NR7**.

Ruku v ruce s volbou architektury rozhraní přichází i volba formátu dat. Pro jeho jednoduchost a rozšířenost jsme zvolili formát dat JSON. Toto rozhodnutí řeší požadavek **NR6**. [20]

Komponenta, kterou se tato práce zabývá bude tedy jen jedna. Jedná se o samotný backend aplikace, který bude poskytovat rozhraní pro frontendovou část aplikace. V této kapitole se budu zabývat jejím návrhem, od architektury až po jednotlivé objekty abstrahující databázi. Tyto objekty jsou důležité, protože budou v pravděpodobně sloužit jako základ návrhu nové databáze v budoucnu. Samotnou specifikaci API si dovolím v textové práci vynechat, jeho kompletní dokumentaci bude možné nalézt na přiloženém médiu.

Ohledně API bych rád zmínil design některých endpointů. Výše jsme se rozhodli, že se bude jednat o REST API. V definici některých endpointů jsme se ale lehce odchýlili od čistého RESTu – konkrétně například u endpointu zabývajícím se parametry a filtry kategorií. Náš návrh obsahuje pouze metody GET a PUT. Důvod je prostý – uživatel data také v tomto případě zpracovává najednou. Z důvodu eliminace zbytečné business logiky na FE jsme se rozhodli výsledek zpracování uživatelem poslat celý najednou. Operace PUT tedy v tomto případě zajišťuje vytváření, mazání i upravování parametrů relevantních pro danou kategorii.

2.2.1 Architektura backendu administrace e-shopu

Jedním z hlavních přínosů nové aplikace má být rozdělení na více vrstev pro lepší rozšiřitelnost v budoucnu. Po konverzaci v týmu a konzulatci se zadavatelem jsme došli k závěru, že nejvhodnější bude napsat novou administraci jako třívrstvou. Konkrétně se jedná o architektonický vzor MVC, neboli Model View Controller. Aplikace je tedy rozdělena do 3 logických komponent: [23]

1. Model

   „*Tato úroveň se považuje za nejnižší v porovnání s vrstvami View a Controller. Primárně reprezentuje data uživateli a definuje ukládání datových objektů aplikace.*“ [23]

   V rámci Symfony jako takového se na této úrovni nachází framework Doctrine, který využívá tzv. Repository Pattern. Jedná se o abstrakci přístupu k databázi a v případě Symfony obsahuje předpřipravené metody pro vyhledávání dat. Také umožňuje vytvářet složitější dotazy, a to jak v čistém SQL, tak v abstrakci z frameworku Doctrine DQL, která získané objekty umí hned namapovat na entity vytvořené v PHP. [24]

   Ve většině případů není ukládání dat úplně přímočaré. Aplikace potřebuje nějaká pravidla, jak zpracovávat příchozí data. Jako příklad bych uvedl stav, kde databáze obsahuje nějaké dvě entity, které jsou na sobě závislé. Při návrhu rozhraní aplikace pak může být vhodné tyto data poslat najednou. Uložení dat pak musí probíhat podle určitých pravidel, které zajistí zpracování dat a správné uložení entit do databáze. Tyto pravidla se nazývají business logikou. [25]

   Pokud obsluha databáze vyžaduje nějakou business logiku, tato logika patří do „podvrstvy“ modelové vrstvy, která se nazývá Service. [26]

2. View

   „*Tato úroveň je hlavně spojena s uživatelským rozhraním a používá se k poskytnutí vizuální reprezentace MVC modelu. Jednodušeji řečeno, tato úroveň se zabývá zobrazováním výstupu uživateli.*“ [23]

   Tato vrstva bude zodpovědností frontendové části aplikace, zde se jí tedy nebudu hlouběji zabývat.

3. Controller

   „*Tato úroveň se zabývá správou request handleru, česky obsluhovače žádostí. Často se považuje za mozek MVC systému – spojení mezi uživatelem a systémem. Controller dokončuje cyklus získání uživatelského vstupu, konvertování daného vstupu na požadovaný výsledek a posílání tohoto výsledku do vrstvy view.*“

   Je tedy zřejmé, že Controller je nejkomplikovanější částí aplikace. Zajišťuje totiž všechno přijímání a odesílání dat. Naštěstí Symfony, tak jako každý další framework, složitou práci s požadavky klientů abstrahuje, a my musíme již jen reagovat na výsledky přijmutí těchto požadavků.

2.2.1.1 Model

Z analýzy problémové domény vyplývá, že toto bude nejsložitější část nové aplikace. Jak jsem zmínil výše, samotnou práci s databází zajišťuje v Symfony framework Doctrine, a práce s ním je uživatelsky přívětivá. Problémem je ale podoba databáze. Jak jsme se dozvěděli při analýze, databáze má několik vážných nedostatků, které ztěžují práci s ní:

- **Chybějící cizí klíče**

  Tento nedostatek přímo způsobí nabobtnání objemu business logiky ve vrstvě service. Za normálních okolností by cizí klíč poskytl dvě věci:

  1. Bylo by nemožné už na úrovni databáze uložit data, která nejsou konzistentní. Jednoduše řečeno, databáze by si sama hlídala, že odkazy mezi entitami jsou validní. Neumožňovala by smazání entit, na které je odkazováno a když už by to umožnila, smazala by i všechny odkazující entity.
  2. Framework Doctrine by identifikátory uložené v databázi uměl abstrahovat na objekty v programovacím jazyce. Získávat data z objektů, které jsou ve vztahu s danou entitou je pak tak jednoduché, jako zavolání příslušného getteru[1] dané entity.

- **Roztříštěnost databáze**

  Tento problém je problémem hlavně z důvodu neexistujících cizích klíčů. Umocňuje to problém zmíněný výše – kvůli neexistující abstrakci těchto vztahů v databázi je nutné ručně získávat entity postupně.

Ani jedna z těchto možností umožněných při přítomnosti cizích klíčů tedy bohužel není k dispozici. Znamená to tedy, že všechna data se musí důkladně zkontrolovat, aby v databázi nevznikaly nekonzistentní data. Zároveň nám databáze negarantuje, že data jsou konzistentní, a je potřeba se na tyto stavy připravit. Výsledkem je obrovské množství business logiky potřebné pro běh aplikace.

Pokud chceme frontendové aplikaci vrátit relevantní data k nějaké entitě, je potřeba tyto data poskládat z mnoha různých databázových tabulek. Takto získaná data se následně uloží v DTO – transportním objektu – který posíláme frontendové části aplikace. Vzniká tedy další mezivrstva na pomezí Controlleru a Modelu – tzv. DTO layer, neboli transportní vrstva. Tato vrstva se v mnoha aplikacích nenachází, ale v našem případě je nezbytná – je potřeba odstínit databázi od rozhraní.

Tato vrstva bude také v ideálním případě sloužit jako základ návrhu nové databáze, která se začne vyvíjet po dokončení kopírování všech funkcionalit staré administrace do nové aplikace.

[1] Jedná se o jednoduchou funkci, která pouze vrátí určitá data z daného objektu.

2.2.1.2 Controller

Řídící vrstva tedy potřebuje zajistit správné přijmutí dat a jejich odeslání ke zpracování do service vrstvy. Pro tento účel je Symfony vybaveno dobře – balíček Form [27] umožňuje vytvářet formuláře pro zpracování příchozích dat. Mohlo by se zdát, že takový formulář je spíše starostí frontendové části aplikace. To je pravda, nicméně balíček Form je univerzální a formulář jím vytvořený nemusí být vůbec zobrazen na webové stránce. Stačí ho aplikovat na příchozí data pro jejich validaci a transformaci z JSONu do – v našem případě – DTO.

Žádné další požadavky na tuto komponentu kladeny nejsou.

2.2.2 Návrh společných komponent backendu

Ačkoliv lze systém relativně dobře rozdělit na několik částí, stále mají tyto části nějaké stejné komponenty. Těmito „spojkami“ jsou objekty domén a SEO klíčových slov, systém přepisů dat pro různé domény a správa obrázků.

2.2.2.1 Doména

Doména je prvním takovou komponentou. Nastavení domény je komplikované, pro účely spojení se daty v systému jsou pro nás důležité dvě věci – název domény a jazyk s ní spojený.

Tabulka 2.1 Domain

název	typ	popis
name	string	Název dané domény
language	string	Jazyk, který má daná doména nastavený
languageCode	string	Zkratka jazyka, který má daná doména nastavený
languageId	integer	Identifikátor jazyka, který má daná doména nastavený
domainGroupId	integer	Identifikátor skupiny domén, do které skupina patří
geoZoneId	integer	Identifikátor státu, do které doména patří
currencyId	string	Identifikátor finanční měny, kterou se na doméně platí
isMainCategoryId	bool	Tento atribut říká, kam se ukládají hlavní kategorie produktu v této doméně. Možnosti jsou dvě, proto se jedná o boolovský atribut.

2.2.2.2 UrlSlug

Další takovou komponentou je objekt popisující SEO klíčová slova. V našem návrhu byl pozměněn název na UrlSlug.

Nyní je vhodný čas k popsání, jak SEO klíčová slova vlastně v aplikaci fungují. Data ohledně SEO klíčů se nachází v tabulce `oc_url_alias`. Tato data se dělí na dva typy –

1. Globální

   Tyto SEO klíče nemají vyplněnou doménu. Fungují tedy pro všechny domény, ve daný objekt je. Globální SEO klíč nesmí být shodný s žádným dalším globálním SEO klíčem ani s žádným dalším lokálním SEO klíčem.

2. Lokální

   Tyto SEO klíče jsou naopak platné pro doménu vyplněnou v atributu domain. Nemůžou kolidovat s žádným globálním SEO klíčem, ale můžou být shodné s jiným lokálním SEO klíčem pro jinou doménu.

Dalším specifikem SEO klíčů je, že se v tabulce ukládá historie. Pro tento účel slouží v databázi atribut moved, který ukazuje na ID objektu, který obsahuje nové klíčové slovo. V případě, že je moved roven nule je tento klíč aktivní. Neaktivní klíč se dá přepsat.

Tabulka 2.2 UrlSlug

název	typ	popis
id	integer	ID daného objektu SEO klíče
domain	string	Název domény, pro kterou toto SEO platí. Prázdný string znamená globální platnost.
keyword	string	Samotné klíčové slovo
active	boolean	Označuje, zda je tento klíč aktivní
query	string	Ukazuje na objekt, který s tímto SEO klíčem souvisí. [2]

2.2.2.3 Overridy

Poslední společnou částí jsou overridy. Tyto nebudeme posílat přímo jako nějaký objekt, nýbrž jejich data přímo propíšeme do ostatních poslaných objektů. Zároveň se v novém systému lehce mění jejich význam. Nejdříve bych ale chtěl vysvětlit důvod jejich existence a jejich funkčnost.

Overridy v tomto systému existují z prostého důvodu – administrace se zabývá spravováním vícedoménových e-shopů. Výsledkem je potřeba zobrazovat různá data na různých e-shopech – ať už jazykové mutace, nebo jiné odlišnosti. Nejjednodušší řešení by bylo mít pro každou doménu svojí vlastní entitu, toto je ale neefektivní z hlediska množství uložených dat. Jednotlivé atributy se tak přepisují pouze v případě, že je to nutné a společná data zůstávají neduplikovaná.

Jejich funkce je tedy přímočará – obsahují ukazatel na ID zdrojového objektu, odkaz na konkrétní sloupeček v databázi a samotnou hodnotu, která se použije pro přepis.

V novém systému se jejich význam mění z důvodu zavedení výchozí domény. Stará administrace nepracuje s pojmem výchozí domény, a vytváří tak overridy pro všechny domény bez rozdílu. Výchozí doména ale přináší nový pohled na věc – výchozí doména bude obsahovat základní data, tedy žádný override.

2.2.2.4 Obrázky

Stará administrace s obrázky pracuje pouze jako textový řetězec, který je cestou k danému souboru na serveru. Rozdělení aplikace na backend a frontend ale vyžaduje zavést nový systém práce se soubory, protože frontendová část aplikace již nebude mít jednoduše přístup k souborům uloženým na serveru. Tuto část problematiky zpracoval Tomáš Hojek, ale pro můj návrh je potřeba ukázat objekt, který bude nově posílán frontendu místo jednoduchého stringu.

2.2.3 Návrh výrobců

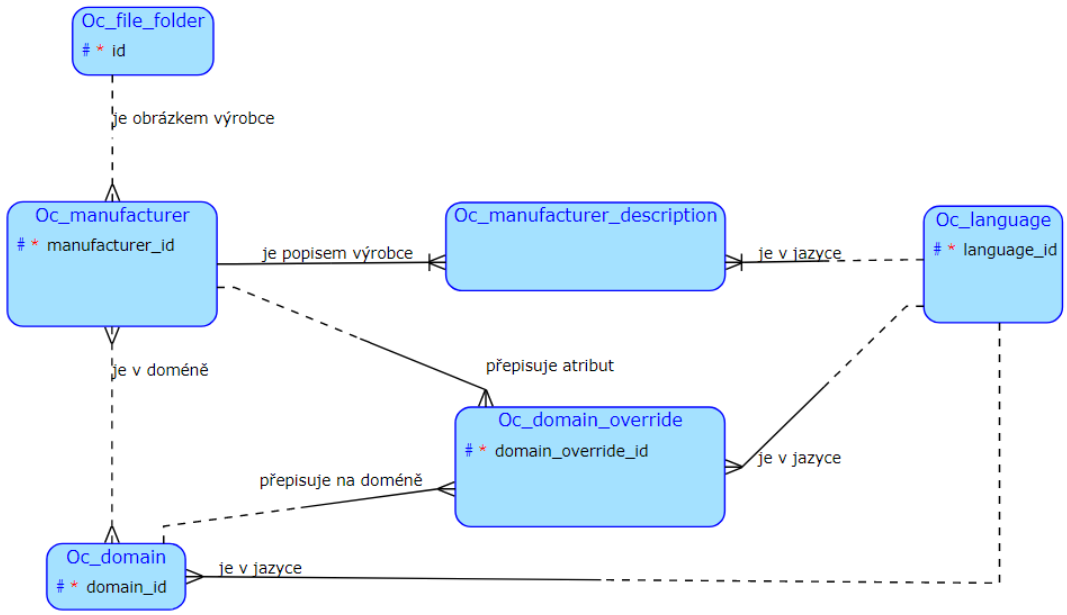
Výrobci jsou nejizolovanější částí databáze a jsou tak ideálním místem, kde začít. Částečný model databáze zabývající se výrobci je zachycen na následujícím obrázku 2.1. Pro jednoduchost jsou vynechány atributy, které nejsou primárními klíči.

Jak je z obrázku patrné, i relativně jednodušší část databáze není příliš uživatelsky přívětivá. Dále se tedy budu zabývat jednotlivými datovými objekty, které vzešly z této části databáze.

Tabulka 2.3 StoredFile

název	typ	popis
id	integer	Identifikátor souboru
name	integer	Název souboru
type	string	Typ souboru
src	string	Cesta pro získání daného souboru
size	string	Velikost souboru

Obrázek 2.1 Částečný model databáze výrobci



2.2.3.1 ManufacturerBasic

Zjednodušený model výrobce, který se vrací jako odpověď na požadavek GET všech výrobců.

Tabulka 2.4 ManufacturerBasic

název	typ	popis
id	integer	ID daného výrobce
name	string	Výchozí název daného výrobce
order	integer	Pořadí daného výrobce
domains	string array	Seznam domén, na kterých se výrobce nachází

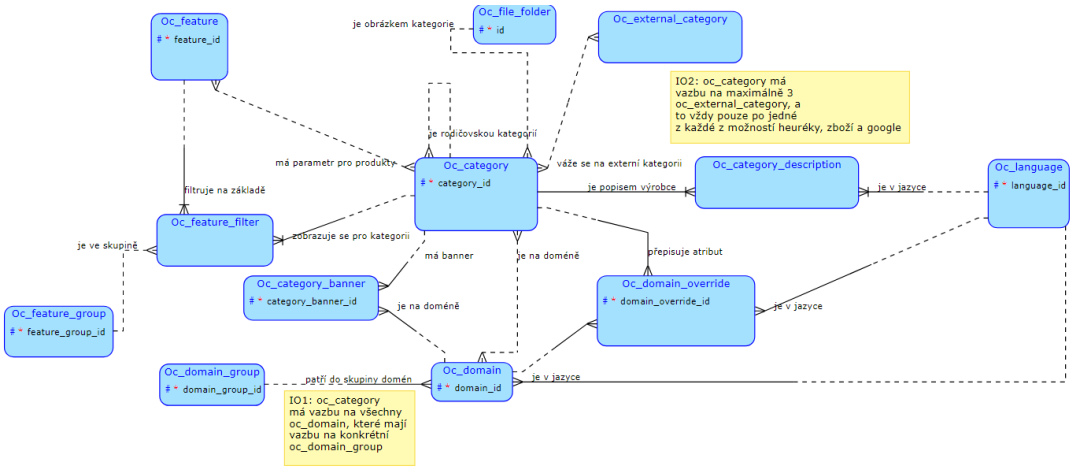
2.2.3.2 Manufacturer

Detail výrobce, který se vrací jako odpověď na požadavek GET konkrétního výrobce. Velmi podobným datovým objektem je ManufacturerCreate, který neobsahuje ID daného výrobce, otherDomainNames, order a posílá se s atributem imageId a bez atributu Image. Ten z důvodu podobnosti přeskočím. Atribut defaultDomain obsahuje informace ke konkrétní doméně, zbytek informací je společný pro všechny domény.

Tabulka 2.5 Manufacturer

název	typ	popis
id	integer	ID daného výrobce
defaultDomain	DomainManufacturer	Obsahuje data z výchozí domény
otherDomainNames	string array	Seznam domén, na kterých se výrobce nachází kromě výchozí domény
order	integer	Pořadí daného výrobce
imageId	integer	Identifikátor uloženého obrázku
image	StoredFile	Obrázek daného výrobce

Obrázek 2.2 Částečný model databáze kategorie



2.2.3.3 DomainManufacturer

Tento objekt obsahuje data, která jsou specifická pro danou doménu a lze je přepisovat pomocí overridů.

Tabulka 2.6 DomainManufacturer

název	typ	popis
domainName	string	Jméno dané domény
manufacturerName	string	Název daného výrobce v dané doméně
description	string	Popis výrobce
metaDescription	string	Meta popis daného výrobce
metaTitle	string	Meta titulek k danému výrobci
metaKeywords	string array	Klíčová slova související s daným výrobcem
seoKeyword	string	SEO klíčové slovo daného výrobce

2.2.4 Návrh kategorií

Kategorie jsou další zpracovávanou částí databáze. Oproti výrobcům je množina zpracovávaných tabulek větší. Částečný model databáze zachycující část zabývající se kategoriemi je zachycen na obrázku 2.2.

Stará administrace má u kategorií spoustu nedostatků. Kontrola konzistence dat probíhá na

úrovni administrátora, což v některých případech v minulosti vedlo až k pádu systému a nutnosti ručního zásahu do databáze v momentě, kdy administrátor zadal špatná data.

Kategorie neexistují jen jako samostatné entity, ale v rámci stromu kategorií. Tento vztah je potřeba v první řadě hlídat, aby nevznikly nekonečné smyčky kategorií. V druhé řadě je potřeba zajistit, aby tyto stromy zůstaly konzistentní. Pro dodržení konzistence stromů je potřeba pracovat se skupinami domén. Skupina domén je, jednoduše řečeno, množina domén, které čerpají ze stejného stromu kategorií. Je tedy nutné dodržet, aby každá kategorie byla ve vztahu s doménami z právě jedné skupiny domén a tento požadavek musí platit pro celý strom kategorií, jehož je daná kategorie součástí. Posledním problémem je zajistit, aby každá kategorií patřící do nějaké skupiny domén měla celou množinu domén dané skupiny. Toto se může zdát zřejmé, ale na úrovni databáze je tento vztah realizován jako pole textových řetězců názvů domén, oddělených speciálním oddělovacím znakem. Nedodržení tohoto požadavku by vedlo k nekonzistenci stromu kategorií na různých doménách z dané skupiny.

Základní struktura návrhu kategorií je velmi podobná návrhu Výrobců a myšlenky za jednotlivými objekty zůstává stejná. Odpadla zde ale možnost měnit pro kategorii skupinu domén. Důvod pro tento krok je prostý – jak jsme zjistili výše, pro konzistenci stromů je nutné, aby každá kategorie měla vztah se všemi doménami z dané skupiny domén.

Naopak je zde oproti výrobcům navíc požadavek pro získání externích kategorií. Tyto kategorie slouží pro napojení na ekvivalentní kategorie v systémech Heuréky, Google a Zboží. Ve staré administraci a databázi se s těmito kategoriemi pracuje pouze jako s textovými řetězci jejich názvů. Nová administrace přidává robustnější systém využívající unikátní identifikátory těchto kategorií. Tento systém je zpracováván Tomášem Hojkem a já budu pouze využívat jím vytvořené rozhraní.

2.2.4.1 CategoryBasic

Jedná se o objekt, který je vracem pro požadavek GET všech kategorií.

Tabulka 2.7 CategoryBasic

název	typ	popis
id	integer	Ientifikátor dané kategorie
name	string	Výchozí název dané kategorie
order	integer	Pořadí dané kategorie
domainGroupId	integer	Identifikátor skupiny domén, do které daná kategorie patří
parentCategoryId	integer	Identifikátor rodičovské kategorie ve stromě kategorií
domains	string array	Seznam domén, na kterých se kategorie nachází

2.2.4.2 Category

Detail kategorie, který se vrací jako odpověď na požadavek GET konkrétní kategorie. Velmi podobným datovým objektem je CategoryCreate, který obsahuje pouze atributy defaultDomain a domainGroupId. Ten z důvodu podobnosti přeskočím. Atribut defaultDomain obsahuje informace ke konkrétní doméně, zbytek informací je společný pro všechny domény.

2.2.4.3 DomainCategory

Tento objekt obsahuje data, která jsou specifická pro danou doménu kategorie. Podobnými mutacemi jsou DomainCategory, která se používá pro požadavky GET a obsahuje atribut image

Tabulka 2.8 Category

název	typ	popis
id	integer	ID dané kategorie
defaultDomain	DomainCategory	Obsahuje data z výchozí domény
otherDomainNames	string array	Seznam domén, na kterých se kategorie nachází kromě výchozí domény
order	integer	Pořadí dané kategorie
domainGroupId	integer	Identifikátor skupiny domén, do které daná kategorie patří

a DomainCategoryUpdate, která se používá pro požadavky POST a PUT a obsahuje atribut imageId.

Tabulka 2.9 DomainCategory

název	typ	popis
domainName	string	Jméno dané domény
categoryName	string	Název dané kategorie v dané doméně
description	string	Popis kategorie
metaDescription	string	Meta popis dané kategorie
metaTitle	string	Meta titulek k dané kategorii
metaKeywords	string array	Klíčová slova související s danou kategorií
seoKeyword	string	SEO klíčové slovo dané kategorie
parentCategoryId	integer	Identifikátor rodičovské kategorie ve stromě
heurekaCategoryId	integer	Identifikátor ekvivalentní kategorie v systému heuréka
googleCategoryId	integer	Identifikátor ekvivalentní kategorie v systému google
zboziCategoryId	integer	Identifikátor ekvivalentní kategorie v systému zboží
showManufacturerInName	integer	Určuje detaily při vypisování kategorie v e-shopu
active	integer	Určuje, zda je kategorie aktivní
image	StoredFile	Obrázek dané kategorie pro danou doménu
imageId	integer	Identifikátor obrázku pro danou kategorii

2.2.4.4 CategoryParameter

Parametry jsou určité atributy, které má každý produkt. Pro produkt se vyplňují konkrétními hodnotami. Pro kategorie se parametry vyplňují proto, aby byly poté efektivně nabízeny produktům v dané kategorii. Práce s parametry obnáší obměnu klasického modelu REST API, ve kterém ma každý objekt svoje metody CRUD (Create Read Update Delete). Důvod je prostý – práce s parametry z pohledu administrátora vypadá tak, že vyplní tolik kolik jich potřebuje a pošle je najednou. Výsledkem je rozhraní, které obsahuje pouze metody GET a PUT.

Metoda GET je přímočará, ale metoda PUT tím pádem zastává funkci POST, PUT i DELETE. Vytvoření nového parametru proběhne v případě, že není vyplněný atribut ID. Aktualizace existujícího parametru proběhne v případě, že atribut ID vyplněný je. Smazání proběhne v případě, že před posláním daného požadavku existoval vztah dané kategorie s nějakým parametrem, a jeho ID není obsaženo v novém požadavku. Je dobré zmínit, že proběhne pouze smazání

vazby tohoto parametru na kategorii, ne samotného parametru.

Tabulka 2.10 CategoryParameter

název	typ	popis
id	integer	Identifikátor daného parametru
name	string	Název daného parametru
type	string enum	Typ parametru
unit	string	Jednotka daného parametru (používá se pro typ unit)

2.2.4.5 CategoryFilter

Filtry jsou entity vázané na kategorie a parametry. Umožňují, jak již název napovídá, filtrování produktů při prohlížení e-shopu. Práce s nimi je stejná jako s parametry a rozhraní bude tedy opět obsahovat pouze metody GET a PUT, který se stará o vytváření, upravování i mazání.

Pro metodu PUT je práce se skupinami filtrů abstrahována – pokud chce uživatel vytvořit novou skupinu, stačí vyplnit pouze atribut groupName. Backend se poté postará, aby všechny nové skupiny se stejným jménem byly sjednoceny a bude tak vytvořena jen jedna skupina filtrů.

Tabulka 2.11 CategoryFilter

název	typ	popis
featureId	integer	Identifikátor souvisejícího parametru
groupId	integer	Identifikátor související skupiny filtrů pokud existuje
groupName	string	Název související skupiny filtrů pokud existuje
name	string	Název daného parametru
type	string enum	Typ parametru
unit	string	Jednotka daného parametru (používá se pro typ unit)

2.2.4.6 CategoryBanner

Každá kategorie může mít nějaké bannery. Tyto bannery se používají pro jednodušší navigaci uživatele po e-shopu. Protože jsou bannery vázané kromě kategorie i na konkrétní doménu, obsahují i metodu POST, která vytvoří bannery pro konkrétní doménu dané kategorie. Dále je již ale princip jejich zpracování podobný jako u parametrů a filtrů – posílá se jich celé pole a metoda PUT funguje analogicky k metodě PUT v rozhraní parametrů a filtrů.

2.2.5 Návrh produktů

Produkty jsou poslední částí databáze, kterou se tato práce zabývá. Jedná se o nejobsáhlejší část ze zpracovávaných částí. Částečný model zachycující zpracovávanou doménu je zachycen na obrázku 2.3. Pro přehlednost jsou některé tabulky vynechány.

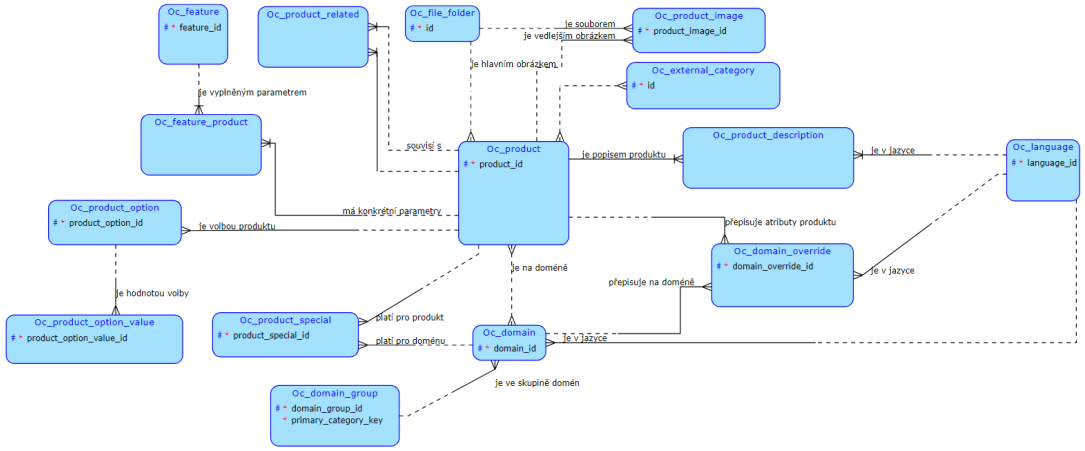
Práce s produkty jsou jádrem celé aplikace. Výrobci i kategorie jsou všechny o ničem bez produktů. Ačkoliv většina postupů je podobná, i produkty mají své unikátní zákeřnosti.

První takovou zákeřností je přiřazení kategorií. Každý produkt musí mít hlavní kategorii a k tomu může mít další, sekundární kategorie. Problém zde opět nastává kvůli faktu, že administrace spravuje více domén. Každý produkt může existovat na více doménách, které můžou mít

Tabulka 2.12 CategoryBanner

název	typ	popis
featureId	integer	Identifikátor souvisejícího parametru
groupId	integer	Identifikátor související skupiny filtrů pokud existuje
groupName	string	Název související skupiny filtrů pokud existuje
name	string	Název daného parametru
type	string enum	Typ parametru
unit	string	Jednotka daného parametru (používá se pro typ unit)

Obrázek 2.3 Částečný model databáze produkty



různé skupiny domén a tedy různé kategorie. Pro sekundární kategorie toto problémem není – stačí vyplnit vazební tabulku a e-shop zobrazí jen kategorie vhodné pro danou doménu. Hlavní kategorie ale není tak jednoduchá, protože každá doména potřebuje pro produkt jeho hlavní kategorii. Na úrovni databáze je toto řešeno dvěma sloupci v tabulce produktu – jeden pro každou skupinu domén. Každá skupina domén pak obsahuje informaci, který sloupec se může použít. Tím pádem se množina skupin domén dělí na dvě podskupiny, nazvěme je kategorické podskupiny. Každý produkt může obsahovat domény z právě dvou skupin domén, za podmínky že jsou obě tyto skupiny domén v různých kategorických podskupinách.

Další zákeřností je zamýšlená práce s finančními měnami. Stará administrace pracuje pouze s českými korunami. Pokud se poté má produkt zobrazit na doméně, která používá jinou měnu, cena se vypočte za běhu podle kurzu uloženého v tabulce `oc_currency`. V nové administraci vznikl požadavek na zadávání ceny ve finanční měně, kterou daná doména má. Je zřejmé, že kvůli zachování kompatibility se starou administrací nelze tento požadavek splnit na 100%. Přidání podpory měn by se totiž mělo řešit už na úrovni databáze.

Existuje ale možnost, jak toto do určité míry podporovat. Každá doména má totiž informaci o finanční měně na ní používané. Pokud zpracováváme příchozí data, lze pak hodnoty konvertovat pomocí tabulky `oc_currency`. Druhá část tohoto problému spočívá ve změně kurzu měn. Zadaná cena v jiné finanční měně než ve výchozích českých korunách by se měnila podle měnícího se kurzu vůči české koruně. Tady přichází na řadu automatický opravný skript, jehož kostra již existuje. V současné době tento skript upravuje kurz v tabulce `oc_currency`. Lze ho upravit, aby ve všech místech, kde je nějaká cena produktu pro jinou měnu než české koruny, vypočetl novou cenu podle změněného kurzu a původní ceny. Úpravu tohoto skriptu provede podle domluvy Ing. Jan

Matoušek, který má na starosti mj. správu staré administrace.

Obecně jsou pak produkty myšlenkově navrženy stejně, jako výrobci. Následuje již výčet jednotlivých objektů abstrahující část databáze, ve které jsou produkty. Výsledný návrh produktů ale dělá větší rozdíly mezi objekty odesílané na požadavky GET a objekty požadovanými při požadavcích POST a PUT. Tyto rozdíly jsou podobného charakteru jako obrázky u výrobců a kategorií, ale je jich víc. Dovolím si tedy varianty objektů využívané pro POST a PUT metody vynechat úplně.

2.2.5.1 ConnectedProduct

Pomocný objekt, který vrátí minimální potřebné informace pro identifikaci objektu v přehledu zobrazovaném frontendem.

Tabulka 2.13 ConnectedProduct

název	typ	popis
id	integer	Identifikátor produktu
productName	string	Název produktu
image	StoredFile	Hlavní obrázek produktu

2.2.5.2 ColorObject

Pomocný objekt, který vrátí informace o barvě

Tabulka 2.14 ColorObject

název	typ	popis
colorId	string	Identifikátor barvy, který je reprezentací barvy v RGB v hexadecimální soustavě
name	string	Název barvy

2.2.5.3 Product

Jedná se o objekt, který je prezentován při zobrazení seznamu všech produktů.

2.2.5.4 ProductDetail

Objekt vracený při požadavku GET na konkrétní produkt. Chtěl bych upozorni na nepěkné dělení na mainCategoryId a mainCategoryId2. Toto rozdělení je z důvodu výše zmíněné nutnosti mít až 2 hlavní kategorie. Zároveň to usnadňuje zpracování těchto kategorií pro frontend – nemusí je třídit a rovnou ví, k jakým doménám přiřadit jaké kategorie.

2.2.5.5 DomainProduct

Tento objekt obsahuje data specifická pro danou doménu. Tato data lze přepisovat pomocí overridů.

2.2.5.6 ProductParameter

Objekt, který obsahuje vyplněné parametry produktu. Na rozdíl od práce s parametry u kategorií zde již probíhá vyplnění konkrétní hodnotou. Z hodnot valueNumeric a valueTextual je vyplněná vždy právě jedna hodnota.

Tabulka 2.15 Product

název	typ	popis
id	integer	Identifikátor produktu
productName	string	Název produktu
manufacturerName	string	Název výrobce
mainCategoryName	string	Celý název hlavní kategorie
image	StoredFile	Hlavní obrázek produktu
ean	string	EAN kód
internalStock	integer	Počet kusů v interním skladu
externalStock	integer	Počet kusů v externím skladu
earnings	number	Odhadovaný zisk při prodeji jednoho produktu
viewsQuartal	integer	Shlédnutí produktu za poslední kvartál
soldQuartal	integer	Počet prodaných produktů za poslední kvartál
conversionQuartal	number	Poměr počtu zobrazení a prodaných kusů
toBuy	integer	Doporučený počet produktů k dokoupení
addedDate	string	Datum přidání produktu
modifiedDate	string	Poslední datum modifikace produktu
active	bool	Značí, zda se produkt nabízí v e-shopu
domains	string array	Všechny domény daného produktu

2.2.5.7 ProductSpecial

Stará administrace nabízela změnu ceny ve formě slevy a akce. Ukázalo se, že sleva se nevyužívala – nová administrace tak bude pracovat pouze s akcemi, nebo-li specials. Akce nastavují cenu pro konkrétní doménu a skupinu zákazníků.

2.2.5.8 ProductOption

Pro každý produkt můžou existovat související produkty, barevné varianty atp. V některých případech je ale vhodné zavést ještě jiný konstrukt, a to volby. Volba může být například velikost u bot nebo dvoubalení nějakého produktu. Každou volbu je nutné vytvořit pro všechny existující jazyky kvůli systému Opencart, na kterém funguje e-shop.

2.2.6 Testování

Jedním z požadavků je důkladné testování aplikace. Symfony pro tento účel integruje framework PHPUnit. Automatických testů z pohledu toho, co testují rozlišujeme několik druhů:

- Unit (jednotkové) testy[28]

  „*Jednotkový test zaručuje, že jednotlivé části zdrojového kódu (např. jedna třída, nebo specifická metoda v nějaké třídě) pracuje, jak bylo zamýšleno. Psaní unit testů v Symfony aplikaci je stejné jako psaní standartních PHPUnit jednotkových testů.*“

  U jednotkových testů je důležitá izolace – testuje se pouze daná krátká část kódu, a případné závislosti se nahrazují tzv. mockováním a předpokládá se, že fungují správně.

- Integrační testy [28]

  „*Integrační test otestuje větší část aplikace najednou v porovnání s jednotkovým testem (např. kombinaci tříd ze service vrstvy). Integrační testy můžou vyžadovat použití Symfony Kernelu*

Tabulka 2.16 ProductDetail

název	typ	popis
id	integer	Identifikátor produktu
defaultDomain	DomainProduct	Atributy specifické pro danou doménu
otherDomainNames	string	Všechny domény produktu kromě výchozí
mainCategoryId	integer	Identifikátor hlavní kategorie pro skupinu domén hlavní domény
otherCategoryIds	integer array	Zbytek kategorií pro skupinu domén hlavní domény
mainCategoryId2	integer	Identifikátor hlavní kategorie pro druhou skupinu domén
otherCategoryIds2	integer array	Zbytek kategorií pro druhou skupinu domén
stockStatusId	integer	Co se má zobrazit, pokud produkt není skladem
images	StoredFile array	Obrázky produktu, první obrázek je hlavním obrázkem
ean	string	EAN kód
width	number	Šířka produktu
length	number	délka produktu
height	number	Výška produktu
dimensionsUnitId	integer	Identifikátor jednotky rozměrů produktu
weight	number	Váha produktu
weightUnitId	integer	Identifikátor jednotky váhy produktu
internalStock	integer	Počet kusů v interním skladu
externalStock	integer	Počet kusů v externím skladu
reservedAmount	integer	Počet rezervovaných produktů
watchdogAmount	integer	Počet hlídacích psů na tomto produktu
freeShipping	bool	Určuje, zda se platí pro produkt poštovné
sendViaZasilkovna	bool	Určuje, zda lze produkt poslat Zásilkovnou
colors	ColorObject array	Barvy produktu
relatedProducts	ConnectedProduct array	Podobné produkty
relatedProductsMutual	bool	Zda je vazba podobnosti oboustranná
colorways	ConnectedProduct array	Barevné varianty produktu
colorwaysMutual	integer	Zda je vazba barevných variant oboustranná
gifts	ConnectedProduct array	Produkty, které se dávají jako dárky k tomuto produktu
availabilityDate	string	Datum dostupnosti produktu
warranty	integer	Délka záruky v měsících
active	bool	Značí, zda se produkt nabízí v e-shopu
manufacturerId	integer	Identifikátor výrobce produktu
model	string	Model produktu
taxClassId	integer	Identifikátor daňové třídy produktu
purchasePriceWithoutTax	number	Nákupní cena produktu

pro získání závislostí z kontejneru pro dependency injection.“ V Symfony aplikaci se pro integrační testy používá třída KernelTestCase, která rozšiřuje základní rozhraní od frameworku PHPUnit. Umožňuje zapínat kernel pro získání případných závislostí. Tento kernel se při každém testu restartuje, aby testy fungovaly nezávisle na sobě.

- Aplikační testy [28]

Tabulka 2.17 DomainProduct

název	typ	popis
domain	Domain	Daná doména
productName	string	Název daného produktu v dané doméně
description	string	Popis produktu
metaDescription	string	Meta popis dané produktu
intro	string	Krátký popis daného produktu
metaKeywords	string array	Klíčová slova související s daným produktu
seoKeyword	string	SEO klíčové slovo daného produktu
heurekaCategoryId	integer	Identifikátor kategorie produktu v systému heuréka
googleCategoryId	integer	Identifikátor kategorie produktu v systému google
zboziCategoryId	integer	Identifikátor kategorie produktu v systému zboží
heurekaName	string	Název produktu ve systému heuréka
heureka	string	Produkt v systému heuréka
sellingPriceWithoutTax	number	Prodejní cena daného produktu
finalPrice	number	Cena pro výchozí skupinu zákazníků pro danou doménu

Tabulka 2.18 ProductParameter

název	typ	popis
productId	integer	Identifikátor produktu
parameter	CategoryParameter	Parametr, který se má vyplnit
valueNumeric	number	Číselná hodnota parametru
valueTextual	string	Textová hodnota parametru

Tabulka 2.19 ProductSpecial

název	typ	popis
productId	integer	Identifikátor produktu
productSpecialId	integer	Identifikátor akce
customerGroupId	integer	Identifikátor cílové skupiny zákazníků
domain	string	Cílová doména
priority	integer	Priorita akce
price	number	Nová prodejní cena produktu
dateStart	string	Datum začátku akce
dateEnd	string	Datum konce akce

„*Aplikační testy testují integraci všech různých vrstev aplikace (od routingu po views).*“

Tyto testy v Symfony jsou opět realizovány rozšířením PHPUnit, a to konkrétně třídou WebTestCase. Testování již neprobíhá čistě s využitím PHP konstruktů, ale na pozadí běží reálný HTTP klient, který posílá reálné (programátorem definované) požadavky na testovací server a zkoumá výsledky.

Tato aplikace je otestována všemi z těchto typů testů. Většina testů je integračních – unit testy se zabývají převážně konverzemi dat na úrovni DTO objektů. Integrační testy se zabývají testováním tříd obsahujících business logiku. Tyto testy využívají balíčku DAMADoctrineTest-

Tabulka 2.20 ProductOption

název	typ	popis
productOptionId	integer	Identifikátor volby produktu
sortOrder	integer	Pořadí volby
languageVariants	LanguageVariant array	Jazykové mutace popisu dané volby
productOptionValues	ProductOptionValue array	Jednotlivé volby produktu (např. konkrétní velikosti bot)

Tabulka 2.21 ProductOptionValue

název	typ	popis
externalQuantity	integer	Počet kusů v externím skladu
internalQuantity	integer	Počet kusů v interním skladu
subtract	integer	Určuje, o kolik se sníží skladový počet při zakoupení této volby
prefix	string enum	Určuje, zda je rozdíl ceny oproti základnímu produktu záporný nebo kladný
price	number	Rozdíl ceny oproti základnímu produktu
sortOrder	integer	Pořadí hodnoty volby
ean	string	EAN kód
languageVariants	LanguageVariant array	Jazykové mutace popisu dané volby

Tabulka 2.22 LanguageVariant

název	typ	popis
languageId	integer	Identifikátor jazyka
optionName	string	Název v daném jazyce

Bundle, který využívá transakce pro eliminaci dopadů na testovací databázi. Toto je velmi užitečné hlavně při lokálním spouštění testů. Aplikační testy byly navrženy až při návrhu produktů, a to z důvodu přenosu validace přijímaných dat do Symfony formulářů. Validace kategorií a výrobců probíhá „ručně“ v service třídách, ale výsledek této validace je shodný s validací pomocí formulářů.

Z pohledu programátora se jedná o white-box testy, neboť kód píšeme já s kolegou Bc. Tomášem Hojkem.

Dalším způsobem testování aplikace je statická analýza. Na rozdíl od klasického testování není jejím cílem chování aplikace. Statická analýza prochází kód a hledá potenciálně chybné konstrukce. Pro náš projekt budeme využívat nástroj PHPStan [29].

2.3 Použité technologie

V této sekci představím hlavní použité technologie.

2.3.1 PHP

PHP (neboli PHP: Hypertext Preprocessor) je široce používaný, skriptovací programovací jazyk s otevřeným zdrojovým kódem, který je určen především k použití na serveru. Jazyk PHP je vyvíjen již od roku 1995 a nejnovější verzí je PHP 8.1. Tuto verzi budeme také používat pro vývoj. PHP je multiplatformním řešením a funguje na všech hlavních operačních systémech –

Windows, Linux i MacOS.

2.3.2 Symfony

Symfony je nejen framework pro vývoj webových aplikací s otevřeným zdrojovým kódem, ale také množina znovupoužitelných komponent. Vývoj Symfony probíhá od roku 2005. Framework se skládá jak z již existujících projektů s otevřeným zdrojovým kódem (Doctrine ORM, PHPUnit, ...), tak z vlastních komponent (Symfony YAML, Symfony Dependency Injector, ...). Framework staví na 3-vrstvé architektuře MVC. Mnoho komponent, které najdeme v Symfony, se dají najít ve velkém množství PHP aplikací, protože mají výbornou znovupoužitelnost. [18]

Náš projekt bude používat určitá rozšíření tohoto projektu. Asi nejvýznamnějším je balíček FOSRestBundle. Tento balíček se, jak již název napovídá, zaměřuje na Symfony aplikace poskytující webové rozhraní typu REST a usnadňuje v určitých místech práci. [30]

2.3.3 MySQL

Jedná se o relační databázový systém s otevřeným zdrojovým kódem. První vydání se datuje k roku 1995. Většinou se používá jako databázová komponenta u aplikací, ale je možné ji použít i samostatně. [31]

2.3.4 Doctrine

Doctrine je množina knihoven zabývající se ukládáním do databáze a mapováním objektů. Nejdůležitější projekty jsou DBAL a ORM. ORM je v tomto případě opravdu knihovnou pro PHP. Doctrine pracuje s vlastním dialektem SQL, nazvaném DQL (Doctrine query language). [19]

Doctrine je výchozí knihovnou zajišťující práci s databází pro Symfony.

2.3.5 Docker

„*Docker pomáhá vývojářům stavět kompaktní a přenosné software kontejnery, které zjednodušují vývoj, testování a nasazování softwarových aplikací.*“ [32]

Princip Dockeru je v podstatě velmi jednoduchý – jedná se o jakýsi virtuální počítač. Na rozdíl od klasických virtuálních počítačů ale všechny jeho podprocesy sdílejí systémové prostředky, díky čemuž je Docker mnohem výkonnější. Náš projekt používá zároveň nástroj Docker Compose, který slouží k snadnému běhu aplikací, které potřebují více Docker kontejnerů. [33]¨

2.3.6 nginx

Jedná se o HTTP server, reverzní proxy a zároveň IMAP/POP3 proxy server. nginx je zaměřený na výkon, stabilitu a nízké požadavky na běh. Vývoj probíhá od roku 2004. [34]

Důvodem použití nginxu a ne jiného řešení je fakt, že Jagu již nginx používá pro běh dalších svých projektů.

Kapitola 3
Implementace

Tato kapitola se bude zabývat implementací nového systému podle návrhu. Konkrétně budu popisovat jednotlivé použité technologie a knihovny na příkladech přímo ze zdrojového kódu aplikace. Strukturována bude tato kapitola podle jednotlivých logických částí projektu a chronologicky.

3.1 Implementace výrobců

První komponentou, kterou jsme začali implementovat, byli výrobci. Z hlediska dělby práce jsme na první iteraci výrobců s kolegou Bc. Tomášem Hojkem pracovali společně, a to jak na návrhu, tak na implementaci. Další potřebné změny již zaštiťoval Tomáš.

3.1.1 Vygenerování entit

Prvním krokem bylo vytvoření objektů reprezentujících entity. Protože jsme používali již existující databázi, podoba entit byla víceméně pevně daná. Symfony naštěstí nabízí nástroje, jak z již existující databáze vytvořit objekty reprezentující entity. Konkrétně se pro tento účel dá využít příkaz `php bin/console doctrine:mapping:import`. Po použití tohoto příkazu je potřeba ještě vytvořit repozitáře pro práci s entitami. Toto je opět možné vygenerovat pomocí příkazu `php bin/console make:entity --regenerate App`.

Klíčové jsou zde anotace – ty říkají frameworku Doctrine, na jaké sloupečky se jednotlivé atributy mapují a také napovídají vlastnosti těchto sloupečků.

Výpis kódu 3.1 Objekt Manufacturer

```
/**
 * @ORM\Table(name="oc_manufacturer",
 *     indexes={
 *         @ORM\Index(
 *               name="oc_manufacturer_domains_index",
 *               columns={"domains"})})
 * @ORM\Entity(
 *       repositoryClass=
 *           "App\Repository\Manufacturer\ManufacturerRepository")
 */
class Manufacturer implements OrderInterface
{
    /**
     * @var int
```

```
     * @ORM\Column(
     *      name="manufacturer_id", type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $manufacturerId;
    /**
     * @var string
     * @ORM\Column(
     *      name="name", type="string", length=64, nullable=false)
     */
    private $name = '';
   ...
```

3.1.2 Service vrstva

Jak jsme se dozvěděli dříve, service vrstva obsahuje hlavně business logiku. Z hlediska použitých technologií je zde zajímavé použití databázových transakcí. Databázová transakce je nějaká skupina příkazů, které převedou databázi z jednoho stavu do druhého, a to za dodržení čtyř vlastností „ACID“: [35]

1. **A** – Transakce je atomická, tzn. celá skupina příkazů se chová jako jeden celek
2. **C** – Počáteční i konečný stav databáze je konzistentní
3. **I** – Různé transakce jsou na sobě nezávislé
4. **D** – Výsledek úspěšně dokončené transakce je trvale uložen do databáze

Transakce jsou v našem případě nezbytné, protože pracujeme s daty z mnoha tabulek najednou v rámci jednoho požadavku. V případě selhání na konci dané transakce se všechna předtím uložená data zahodí kvůli udržení konzistence v databázi. Práci s transakcemi nám v Symfony abstrahuje objekt typu `EntityManagerInterface`, který se mj. stará o ukládání a mazání jednotlivých entit.

Výpis kódu 3.2 Ukázka použití transakcí

```
public function deleteManufacturerDomain(
    int $manufacturerId,
    string $domain): void
    {
        $manufacturer = $this->getManufacturerOrException(
            $manufacturerId);
        if ($manufacturer->getDefaultDomain() == $domain) {
            throw new ConflictHttpException('Unable to delete
                default domain!');
        }

        $conn = $this->entityManager->getConnection();
        $conn->transactional(function () use (
                $manufacturerId,
                $manufacturer,
                $domain
            ) {
            $this->domainService->removeDomainOverrides(
                $domain,
```

```
                OverrideConstants::MANUFACTURER_TABLE_NAME,
                $manufacturerId);
            $manufacturer->removeDomain($domain);
            $this->urlSlugService->deleteUrlSlug(
                UrlSlugTO::createQueryValueManufacturer($manufacturerId),
                $domain
            );
            $this->entityManager->flush();
        });
    }
```

3.1.3 DTO vrstva

Jak jsme se dozvěděli při analýze a návrhu, naše aplikace potřebuje abstrahovat databázi pomocí DTO vrstvy. Tato vrstva obsahuje z velké většiny pouze data a funkce pro manipulaci s danými daty. Z hlediska Symfony je zde nejzajímavější použití validačních pravidel. Tato pravidla se definují pomocí anotací a používají se při přijímání dat v Controlleru.

Na následující ukázce je vidět použití anotace `Valid`. Konkrétně tato anotace říká, že daný objekt musí také projít validací. V případě selhání vygeneruje Symfony chybu ve formuláři a tuto chybu my následně pošleme zpět. Validačních anotací je velké množství a lze si i napsat své vlastní validační anotace, což si ukážeme u produktů.

Výpis kódu 3.3 Ukázka validačních anotací v DTO

```
class ManufacturerTO
{
    ...
    /** The default domain information for this manufacturer.
     * @Assert\Valid()
     */
    private DomainManufacturerTO $defaultDomain;

    ...
```

3.1.4 Symfony formuláře

Jak jsme se dozvěděli při návrhu, Symfony pomocí balíčku Forms [27] umožňuje definovat pravidla pro přijímaná data. Pomocí formuláře lze snadno transformovat příchozí data ve formě JSON na objekty v PHP. Pro vytvoření formuláře stačí rozšířit třídu poskytnutou Symfony `AbstractType` a následně vydefinovat jednotlivá pole pro transformaci dat.

Vytvoření formuláře je, jak je vidět, relativně jednoduché. Formuláře lze do sebe zároveň vkládat.

Výpis kódu 3.4 Symfony formulář

```
class DomainManufacturerType extends AbstractType
{
    public function buildForm(
    FormBuilderInterface $builder,
    array $options): void
    {
        $builder->add('domainName', TextType::class);
        $builder->add('manufacturerName', TextType::class);
        $builder->add('description', TextType::class, [
```

```
            'empty_data' => ''
        ]);
        $builder->add('metaDescription', TextType::class, [
            'empty_data' => ''
        ]);
    ...
```

3.1.5 Implementace Controlleru

Podle našeho návrhu je Controller velmi tenkou vrstvou a jeho implementace by tedy neměla být problémem. Jediná práce Controlleru je transformovat případná příchozí data a zavolat příslušnou třídu typu Service, která se o zbytek postará. Aby Symfony umělo s naším Controllerem pracovat, je potřeba přidat anotace, které udávají cestu, při jejímž zavolání se použije příslušná metoda v Controlleru.

Zároveň Controller potřebuje instanci relevantní třídy typu Service. Správnou inicializaci konstruktoru zařídí Symfony pomocí Dependency Injection. [36] Ve zkratce, Symfony samo najde potřebnou závislost, inicializuje jí a následně vytvoří objekt typu Controller pomocí dané závislosti.

Výpis kódu 3.5 Ukázka metody Controlleru

```
class ManufacturerController extends AbstractController
{
    private ManufacturerService $manufacturerService;

    /**
     * ManufacturerController constructor.
     * @param ManufacturerService $manufacturerService
     */
    public function __construct(
    ManufacturerService $manufacturerService)
    {
        $this->manufacturerService = $manufacturerService;
    }
    ...
    /**
     * Creates a manufacturer resource.
     * @Rest\Post("/manufacturers")
     * @param Request $request
     * @return View
     * @throws Throwable
     */
    public function postManufacturer(Request $request): View
    {
        $manufacturerTO = new ManufacturerTO();
        $view = $this->createInputDataObject(
            ManufacturerType::class,
            $manufacturerTO,
            $request);
        if ($view !== null) {
            return $view;
        }

        try {
            $manufacturerId =
                $this->manufacturerService->addManufacturer(
```

```
                    $manufacturerTO);
        } catch (DataFormatException $ex) {
            return $this->processException($ex);
        }
        return View::create(
            ['id' => $manufacturerId],
            Response::HTTP_CREATED);
    }
    ...
```

Za zmínku zde ještě stojí zpracování formuláře. Toto zpracování je pořád stejné, vložili jsme ho tak do abstraktní třídy `AbstractController`. Zpracování formuláře je zcela v rukou Symfony – my mu pouze předáme objekt, do kterého se mají příchozí data transformovat, samotný formulář a požadavek od klienta, který obsahuje poslaná data.

Výpis kódu 3.6 Zpracování formuláře

```
class AbstractController extends AbstractFOSRestController
{
    /**
     * Convert request into custom TO object.
     * @param string $dataType
     * @param mixed $objectTO
     * @param Request $request
     * @return View|null returns View if there is error, otherwise null.
     */
    protected function createInputDataObject(
        string $dataType,
        mixed $objectTO,
        Request $request): ?View
    {
        $this->checkContentTypeJson($request);

        $form = $this->createForm($dataType, $objectTO);
        $form->submit($request->request->all());

        if(!$form->isValid()) {
            return $this->prettyOutput($objectTO, $form);
        }
        return null;
    }
...
```

3.2 Implementace kategorií

Kategorie jsem již implementoval z velké většiny vlastními silami. Tomáš Hojek ke kategoriím přidával jím naimplementovaného správce souborů pro správu obrázků kategorií a správu externích kategorií.

Co se týče technologií, nebyly zde použité v podstatě žádné nové technologie oproti výrobcům, a přeskočím tak popis implementace věcí, které jsem již zmínil u výrobců. Jedinou použitou technologií navíc je DQL, nebo-li Doctrine Query Language, který je sice použit i u výrobců, ale jeho použití tam je Tomášova práce, a proto se jím budu zabývat v rámci kategorií.. Dále rozepíšu ruční způsob získávání chyb.

3.2.1 DQL Doctrine Query Language

Doctrine v základu nabízí jednoduché metody pro nalezení požadovaných entit, přičemž bych poznamenal, že o mazání objektů se stará `EntityManager`:

- `find`– Nalezení objektu podle primárního klíče
- `findOneBy` – Nalezení jednoho objektu podle libovolného pole
- `findBy` – Nalezení všech objektů splňujících dané kritérium
- `findAll` – Nalezení všech objektů

Toto rozhraní je ve většině případů dostatečné. V některých případech ale nestačí. Zvláště v případě tohoto projektu je často potřeba udělat JOIN více tabulek a zároveň vracet více objektů najednou. Získávat je po jednom je velmi neefektivní a v některých případech ani nelze provést zamýšlené operace.

Doctrine má naštěstí k dispozici silnější nástroje pro práci s databází. DQL [37] je abstrakcí dotazovacího jazyka SQL pro framework Doctrine. Ačkoliv neobsahuje všechny funkce SQL – například funkce `UNION` není v DQL dostupná – tak pro drtivou většinu požadavků stačí. Výhodou DQL oproti obyčejnému SQL je v mapování entit z databáze na PHP objekty.

Výpis kódu 3.7 Ukázka použití DQL

```
class CategoryRepository extends ServiceEntityRepository
{
 ...
 public function getCategoriesSorted(
    int $categoryId,
    string $domainName): array
    {
        $qb = $this->createQueryBuilder('c');
        return $qb->select('c')
            ->andWhere('c.parentId = :categoryId')
            ->leftJoin(
                CategoryDescription::class,
                'd',
                Join::WITH,
                'd.categoryId = c.categoryId')
            ->andWhere('c.domains LIKE :domainName')
            ->setParameter('categoryId', $categoryId)
            ->setParameter('domainName', '%' . $domainName . '%')
            ->addOrderBy('c.sortOrder', 'DESC')
            ->addOrderBy('d.name', 'ASC')
            ->getQuery()
            ->getResult();
    }
...
```

3.2.2 Ruční získávání chyb

V úplně první verzi výrobců aplikace hlásila neexistující ID chybovou odpovědí s kódem 404 – Not Found. Takto se program choval i pro data přijatá ve formuláři. Při konzultaci s frontendovou částí týmu se ukázalo, že toto chování je nepřijatelné – Stejný chybový kód mohl přijít z mnoha různých zdrojů, a bylo tedy potřeba chyby vracet namapované na daný formulář.

Kategorie a výrobci obsahují první verzi tohoto mapování, a to ručně. Tento přístup je principově velmi jednoduchý. Pokud při zpracování některého atributu ve formuláři nastane chyba,

uložíme jednoduše krátký popis chyby do pole chyb. Na konci zpracování metody vyhodíme speciální výjimku, která tyto chyby dopraví zpět do Controlleru. Tam se odchytne a odešle se odpověď s kódem 400 – Bad Request.

Není třeba dodávat, že toto není hezké řešení. Lepší řešení si ukážeme u implementace Produktů.

Výpis kódu 3.8 Ukázka ručního získávání chyb

```
public function addCategory(CategoryTO $categoryTO): int
    {
        $errors = [];
        ...
        if (!$this->isDomainInGroup(
            $categoryTO->getDefaultDomain()->getDomainName(), $domains)
        ) {
            $errors['defaultDomain']['domainName'][] = 'NotFound';
        }
        ...
            if (!empty($errors)) {
                throw new DataFormatException($errors);
            }
    ...
```

3.3 Implementace produktů

Implementace produktů byla již čistě moje práce – Tomášem naimplementované externí kategorie a správce souborů jsem přidal v průběhu vývoje sám.

Z hlediska technologií bych u produktů rád demonstroval použití čistého SQL dotazu v rámci frameworku Doctrine, implementaci vlastních validátorů a použití stránkování spolu s parametry pro URL pro získávání produktů. Stránkování se u výrobců a kategorií neimplementovalo z různých důvodů – výrobců je malé množství a je pohodlnější stránky řešit přímo ve frontendové části aplikace, a kategorie jsou potřeba všechny kvůli sestavování stromů. Jinak je množina použitých technologií stejná, jako u kategorií.

3.3.1 Použití SQL v rámci Doctrine

Ačkoliv je DQL velmi silné kladivo, neobsahuje veškeré možnosti SQL. Zároveň byly některá data počítány ve staré administraci pomocí SQL a jejich replikace v DQL by byla obtížná. Rozhodl jsem se tedy v těchto místech použít přímý SQL dotaz.

V případech, kdy není potřeba získat celou entitu z databáze je tento přístup relativně jednoduchý, protože jediná ztráta oproti DQL je nemožnost automaticky namapovat entitu na PHP objekt.

Výpis kódu 3.9 Ukázka použití SQL v rámci Doctrine

```
public function getToBuyAndIncome(
    array $inactiveStockStatusIds,
    int $defaultCustomerGroupId,
    array $productIds): array
    {
      ...
      $sql = "SELECT GREATEST(
                            0,
                            IF(
                                ...
```

```
        $statement = $this->_em->getConnection()->prepare($sql);
        $statement->bindParam('customerGroupId', $defaultCustomerGroupId);
        $result = $statement->executeQuery()->fetchAllAssociative();
        return $result;
    }
```

3.3.2 Vlastní validátory

Jak jsem již předeslal dříve, u produktů jsem použil vlastní validátory formulářů. Hlavní výhoda je v přehlednosti – prakticky celá validační logika byla přesunuta do těchto validátorů. Další velká výhoda je v znovupoužitelnosti – takto napsaný validátor lze použít tam, kde je potřeba pomocí pouhé anotace.

V prvním kroku je potřeba vytvořit anotaci, která se bude následně uplatňovat v DTO vrstvě. Tato anotace je objekt rozšiřující třídu `Constraint`. Tato třída může obsahovat zprávu použitou při validaci a také nastavení, kde se tato anotace dá používat – místa k použití jsou jak celé třídy, tak pouze atributy.

Výpis kódu 3.10 Ukázka anotace

```
use Symfony\Component\Validator\Constraint;
/** @Annotation */
class ProductExists extends Constraint
{
    public string $message = 'This product doesn\'t exist!';
}
```

V dalším kroku je potřeba napsat samotnou validační logiku. V základu se třída s touto logikou jmenuje (pro tento případ) `ProductExistsValidator`, ale lze jí samozřejmě i přejmenovat a toto jméno specifikovat v `ProductExists`. Tento validátor potřebuje pouze metodu `validate`.

Výpis kódu 3.11 Ukázka validátoru existence produktu

```
class ProductExistsValidator extends ConstraintValidator
{
    private ProductRepository $repository;
    /** @param ProductRepository $repository */
    public function __construct(ProductRepository $repository)
    {
        $this->repository = $repository;
    }
    public function validate(mixed $value, Constraint $constraint): void
    {
        if (!is_integer($value) ||
            !($constraint instanceof ProductExists)) {
            return;
        }
        if ($this->repository->find($value) === null) {
            $this->context->buildViolation($constraint->message)
                ->addViolation();
       ...
```

Jak je vidět v ukázce výše, třída `ProductExistsValidator` využívá Dependency Injection. Její použití zde ovšem není automatické – je potřeba Symfony říct, že má pro inicializaci této třídy použít Dependency Injection. Toto lze snadno nastavit v konfiguračním souboru pro služby.

Výpis kódu 3.12 Ukázka registrace validátoru jako služby

```
services:
    ...
    validator.product:
        class: App\Entity\...\ProductExistsValidator
        tags:
            - { name: validator.constraint_validator, alias: the_alias }
       ...
```

Po tomto nastavení je již velmi jednoduché takový validátor použít jako anotaci. Na následující ukázce je vidět použití tohoto validátoru jak pro třídu, tak pro atribut.

Výpis kódu 3.13 Ukázka použití validátoru jako anotace

```
/**
 * @ProductAssert\ProductCreateValid
 */
class ProductCreateTO
{
    ...
    /**
     * @Assert\All({
     *     @Assert\Positive,
     *     @ProductAssert\ProductExists
     * })
     * @var int[]
     */
    private array $relatedProductIds;
    ...
```

3.3.3 Stránkování a parametry v URL

Parametry v URL jsou způsob, jak serveru předat další informace serveru pro danou cestu. Přidávají se na konec URL za symbol `?`, a více parametrů se odděluje symbolem `&`. Typicky se používají pro specifikaci kritérií pro filtrování dat. [38]

Získávání parametrů z URL umožňuje `ParamFetcherInterface`, který podle anotací získá parametry pro použití dále v aplikaci.

Výpis kódu 3.14 Ukázka použití ParamFetcherInterface

```
/**
     * @Rest\Get("/products")
     * @param ParamFetcherInterface $paramFetcher
     * @return View
     * @QueryParam(name="productName", strict=true, nullable=true)
     ...
     * @QueryParam(name="page",
     *          requirements="[\d+]+",
     *          strict=true, default=1,
     *          description="page")
     ...
     */
    public function getProductsFiltered(
        ParamFetcherInterface $paramFetcher): View
    {
        $params = $paramFetcher->all();
```

```
    ...
```

Poslední novou technologií pro produkty je stránkování. Pro tento účel jsem použil balíček `pagerfanta` [39], který zajistí veškerou těžkou práci.

Pro získání stránkované odpovědi stačí objektu `pagerfanta` předat vytvořený dotaz v DQL a informace ohledně počtu objektů na jednu stránku a požadované stránce. Poté stačí vytvořit objekt, který podle objektu `pagerfanta` nastaví své atributy a odešle se uživateli.

Výpis kódu 3.15 Ukázka použití objektu pagerfanta

```
public function findFiltered(array $params): Pagerfanta
        $qb = $this->createQueryBuilder('c')
            ...
        $pagerfanta = new Pagerfanta(
            new QueryAdapter($qb->getQuery(), false));
        try {
            $pagerfanta->setMaxPerPage(intval($params['limit']));
            $pagerfanta->setCurrentPage(intval($params['page']));
        } catch (NotValidCurrentPageException $e) {
            if (intval($params['page']) === 0) {
                $pagerfanta->setCurrentPage(1);
            } else {
                $pagerfanta->setCurrentPage($pagerfanta->getNbPages());
            }
        }
        return $pagerfanta;
    }
```

3.4 Souhrn

Výsledkem je funkční backend administrace e-shopu, který pokrývá většinu funkčnosti staré administrace. Zdrojové kódy implementace lze najít na Gitlabu poskytnutém Jagu. Práce na nové administraci ale není u konce.

Prvním námětem na práci do budoucna je refaktoring[1] částí, které byly navrženy a napsány jako první. Zde je možné zjednodušit business logiku v aplikaci. Pro část s výrobci a kategoriemi je také vhodné přesunout validační logiku do validátorů, podobně jako u produktů.

Dalším námětem je samotný vývoj této aplikace. Jak jsem zmínil výše, nová administrace stále nepokrývá celou funkčnost staré administrace. Pro reálné nasazení nové administrace je toto potřeba dotáhnout do konce. Naštěstí jsou ale námi zpracovávané části aplikace jádrem administrace, a další části by již měly být méně komplikované na vytvoření.

Nakonec bude také potřeba navrhnout novou databázi pro nahrazení databáze staré administrace. Základ pro tento návrh je již obsažen v nynější aplikaci nové administrace, je ale potřeba ho rozvinout a přenést do databáze.

[1] Přepsání kódu lépe při zachování funkčnosti

Kapitola 4
Testování

V této kapitole se budu zabývat testováním nové aplikace. Konkrétně popíši testy z hlediska použitých technologií a postupů. Nejprve se budu zabývat automatickými testy a následně také manuálního testování.

4.1 Automatické testování

Automatické testy jsou testy, které si programátor definuje ručně a následně je může opakovaně spouštět. Testy je tedy možné pouštět i ručně, ale typické je jejich automatické spouštění při odeslání změn v aplikaci na Gitlab nebo jiný verzovací systém. Konkrétně v případě Gitlabu se pro tento účel používá nástroj Gitlab CI, nebo-li Gitlab Continuous Integration.

Jak jsme se dozvěděli při návrhu, naše aplikace bude otestována různými testy – jednotkovými, integračními, aplikačními i statickou analýzou.

4.1.1 Statická analýza

Jedná se o nejzákladnější způsob testů. Jeho nasazení do projektu zajistil Tomáš Hojek. Výhodou statické analýzy je jednoduchost použití a fakt, že otestuje celý kód. Sice mnoho chyb odchytit nedokáže už z principu jejího fungování, ale přesto se jedná o velmi užitečný nástroj. V našem projektu pro tento typ testování používáme nástroj `PHPStan`. Tento nástroj se nastavuje pomocí souboru `phpstan.neon`:

Výpis kódu 4.1 Ukázka nastavení PHPStanu

```
includes:
  - vendor/phpstan/phpstan-symfony/extension.neon
  - vendor/phpstan/phpstan-symfony/rules.neon
  - vendor/phpstan/phpstan-doctrine/extension.neon
  - vendor/phpstan/phpstan-doctrine/rules.neon
  - vendor/phpstan/phpstan-phpunit/extension.neon
  - vendor/phpstan/phpstan-phpunit/rules.neon
parameters:
  level: max
  paths:
    - ./src
    - ./tests
  tmpDir: .tmp
```

```
  excludes_analyse:
  ignoreErrors:
    -
      message: '#Service ".*" is not registered in the container.#'
      path: /*/tests/*
  reportUnmatchedIgnoredErrors: false
  symfony:
    # containerXmlPath: var/cache/dev/srcDevDebugProjectContainer.xml
    # or with Symfony 4.2+
    # containerXmlPath: var/cache/dev/srcApp_KernelDevDebugContainer.xml
    # or with Symfony 5+
    containerXmlPath: var/cache/dev/App_KernelDevDebugContainer.xml
  doctrine:
    objectManagerLoader: tests/object-manager.php
```

4.1.2 Jednotkové testy

Jednotkové testy jsou nejjednodušším typem testů, protože se testuje jen velmi izolovaná část kódu. Jakékoliv závislosti v této části kódu jsou nahrazeny za konstrukty, které reagují podle předem daných, v testech definovaných pravidel. Tomuto se říká mockování. V praxi to ale znamená, že pro naše účely je jen velmi málo míst, kde se dají využít – drtivá většina částí aplikace potřebuje ke svému fungování databázi, již namockovat sice lze, ale testy by pak byly bezzubé.

4.1.3 Integrační testy

Integrační testy již testují „plnohodnotnou“ část aplikace. V praxi to znamená, že všechny potřebné závislosti nejsou mockovány a chovají se přesně tak, jak by se chovaly v produkčním prostředí.

Pozornému čtenáři jistě neuniklo, že tyto závislosti jsou v produkčním prostředí obsluhovány Symfony pomocí Dependency Injection. Naštěstí je tomu tak i při testech. Symfony zařídí správné vytvoření všech objektů a jejich závislostí. Pro zjednodušení testování nám Symfony poskytuje třídu `KernelTestCase`.

Postup použití testů je principem velmi jednoduchý – je potřeba vytvořit nějaká minimální testovací data, na takto vytvořená testovací data použít testované funkce a zkoumat výsledky.

Výpis kódu 4.2 Ukázka použití KernelTestCase

```
class CategoryServiceTest extends KernelTestCase
{
    ...
    protected function setUp(): void
    {
        parent::setUp();
        $container = static::getContainer();
        $this->categoryService = $container->get(
            CategoryService::class);
        $this->createTestData();
    }
    ...
```

Následně již stačí použít získané objekty pro testování.

Výpis kódu 4.3 Ukázka testování integračními testy

```
public function testGetAllCategories(): void
    {
        $categories = $this->categoryService->getAllCategories();
        $this->assertCount(5, $categories);
        ...
```

Výhodou tohoto typu testů je, že se vše odehrává přímo v PHP – není potřeba řešit žádné převody do JSON a podobně. Porovnávání výsledků je tedy naštěstí relativně jednoduché.

4.1.4 Aplikační testy

Tyto testy mají teoreticky otestovat celý postup zpracování požadavku – od jeho přijmutí serverem, přes jeho zpracování až po výslednou odpověď klientovi. Jejich zpracování je ale složitější, než integrační testy. Protože se v rámci aplikačních testů opravdu posílá plnohodnotný HTTP požadavek, je potřeba data pro zpracování dodat ve formátu JSON. Protože se ale stále pohybujeme v PHP, pracujeme s textovými řetězci. Porovnávat jednotlivé části řetězce s požadovaným výsledkem je pak relativně velmi pracné. Proto se použití aplikačních omezuje pouze na produkty a pouze na validační logiku ve formulářích.

Podobně jako pro integrační testy poskytuje Symfony rozhraní, které zjednoduší práci s HTTP požadavky. Název tohoto rozšíření je `WebTestCase`. Základní práce s ním je však velmi podobná `KernelTestCase` pro integrační testy. Navíc zde je obsluha HTTP požadavků – vytvoření virtuálního HTTP klienta a další podpůrné požadavky. Protože já aplikačními testy testuji pouze validaci, většinu těchto možností ale nevyužiji.

Výpis kódu 4.4 Použití WebTestCase

```
class ProductControllerTest extends WebTestCase
{
    ...
    public function setUp(): void
    {
        parent::setUp();
        $client = static::createClient();
        $this->client = $client;
        $kernel = $client->getKernel();
        /** @var EntityManagerInterface $entityManager */
        $entityManager = $kernel->getContainer()
            ->get('doctrine')
            ->getManager();
        $container = static::getContainer();
        $this->productCacheRepository =
            $container->get(ProductCacheRepository::class);
    ...
```

Následně je potřeba vytvořit požadavek, odeslat ho na server a zkoumat reakci serveru. Princip je ale identický jako u integračních testů výše.

Výpis kódu 4.5 Ukázka testování aplikačními testy

```
/** Tests the validation of product */
    public function testProductPostValidator(): void
    {
        // 1) Test if the existence validators check it properly
        $json = '
        {
          "defaultDomain": {
            "domainName": "www.neexistujicidomena.sk",
```

```
            "productName": "string",
            ...
        $this->client->request(
            'POST',
            '/api/products',
            [],
            [],
            [
                'CONTENT_TYPE' => 'application/json',
                'HTTP_X-Requested-With' => 'XMLHttpRequest'
            ],
            $json
        );
        $expectedResponse = json_encode('errors');
        $this->assertEquals(
            400,
            $this->client->getResponse()->getStatusCode());
        $this->assertEquals(
            $expectedResponse,
            json_encode($this->client->getResponse()->getContent()));
```

4.1.5 Sentry

Sentry jsem zmínil na začátku této práce s tím, že ho nebudeme používat. Pokud ale dojde k nasazení nové administrace na nějaký testovací server, Sentry je skvělým nástrojem pro testování. Jedná se v jednoduchosti o detailní logger provozu daného serveru, jeho používání tedy nekončí s koncem vývoje aplikace.

4.2 Manuální testování

Ačkoliv jsou automatizované testy skvělý nástroj, jsou relativně náročné na přípravu. Aby automatizované testy tstovaly každou relevantní část kódu, jejich objem by byl přímo obrovský. Zároveň vývoj probíhal „klasickou“ cestou – nejprve jsme implementovali aplikaci jako takovou, a až následně aplikovali testy. Na úplně prvotní testování aplikace jsme však používali primitivní techniku vytváření požadavků manuálně a jejich odesílání pomocí externího HTTP klienta, v mém konkrétním případě Postman.

4.2.1 Postman

Postman je API platforma pro budování a používání webových aplikačních rozhraní. Umožňuje jednoduše vytvářet, ukládat a i automaticky posílat HTTP požadavky na běžící server. Pro prvotní testování se jedná o nedocenitelný nástroj.

Závěr

Cílem této práce bylo vytvořit funkční backend části systému administrace e-shopu, který by funkčností kopíroval předchozí řešení od společnosti Jagu s.r.o., v určitých oblastech ho rozšiřoval o funkčnosti, které se ukázaly jako vhodné při používání staršího řešení a také bylo potřeba zajistit kompatibilitu obou aplikací při současném běhu. Toto bylo prováděno v týmu a bylo tedy při návrhu potřeba přihlédnout k možným problémům při týmové spolupráci. Celý vývoj probíhaj v iteracích.

Tento cíl se podařilo splnit. Konkrétně se tato práce nejprve zabývala analýzou práce v týmu. To zahrnovalo diskuzi o možných metodikách vývoje, jejich zhodnocení a výběr vhodné metodiky pro tento projekt. Také byly zhodnoceny možné problémy při práci na software v týmu. V rámci těchto diskuzí byly také okomentovány prostředky, které práci v týmu usnadňují. Další část analýzy se zabývala zhodnocením současného řešení systému administrace a jejích nedostatků. Na základě toho vznikl seznam funkčních a nefunkčních požadavků, který sloužil jako základ návrhu aplikace.

Navazující kapitola návrhu se zabývala v první fázi samotnou architekturou aplikace a dále také konkrétním návrhem jednotlivých komponent aplikace. V rámci návrhu se tato práce zabývala částí jedné komponenty – samotného backendu aplikace. V této části práce jsem se zabýval především návrhem DTO vrstvy, která má za cíl abstrahovat databázi a v ideálním případě být základem návrhu nové databáze v budoucnu. Dále jsem se zabýval návrhem samotného rozhraní pro frontendovou část aplikace.

Po dokončení návrhu jsem se věnoval samotné implementaci navržených struktur. V rámci této fáze jsem vytvořil všechny navržené komponenty. Tato fáze byla z celé práce nejnáročnější. Tato práce se v rámci implementační fáze zabývala především popisem použitých technologií a postupů.

Poslední kapitola této práce se zaměřuje na testování. V této kapitole se přibližují konkrétní postupy pro testování, které nabízí framework Symfony. V této části jsem se zabýval především ověřením funkčnosti výsledného řešení.

Výstupem práce je tedy systém administrace multidoménových e-shopů, napsaný v programovacím jazyce PHP při použití frameworku Symfony. Tento systém kopíruje funkčnost staršího systému administrace za použití stejné databáze a při zaručení kompatibility při paralelním běhu obou aplikací. Výsledek zatím nefunguje pro všechny funkcionality staré administrace, avšak pro její jádro ano – je možné administrovat výrobce, kategorie, produkty, objednávky a menší systémy s nimi přímo spojené.

Bibliografie

1. JANÍKOVÁ, Hana. *Týmová spolupráce* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: http://www.benepal.cz/files/project_4_file/TYMOVA-SPOLUPRACE.PDF.
2. KOLAJOVÁ, Lenka. *Týmová spolupráce*. Grada Publishing, 2006. ISBN 80-247-1764-6.
3. WITT, David. 60% of Work Teams Fail – Top 10 Reasons Why [online]. 2011 [cit. 2022-04-06]. Dostupné z: https://leaderchat.org/2011/11/03/60-of-work-teams-fail%E2%80%94top-10-reasons-why/.
4. MLEJNEK, Jiří. *Metodiky vývoje* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/506232/mod_resource/content/6/01.prednaska.pdf.
5. *Sentry* [online]. [B.r.] [cit. 2022-04-28]. Dostupné z: https://sentry.io/welcome/.
6. *Redmine* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://www.redmine.org/.
7. What is Gitlab? [Online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://about.gitlab.com/what-is-gitlab/.
8. DevOps platform [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://about.gitlab.com/topics/devops-platform/.
9. *About Git* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://git-scm.com/about.
10. WOODGATE, Rob. What Is Slack, and Why Do People Love It? [Online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://www.howtogeek.com/428046/what-is-slack-and-why-do-people-love-it/.
11. *Co je Google Meet?* [Online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://apps.google.com/meet/how-it-works/.
12. MLEJNEK, Jiří. *Metodiky vývoje* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/506277/mod_resource/content/2/11.prednaska.pdf.
13. MLEJNEK, Jiří. *Agilní přístup* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/437209/mod_resource/content/5/12.prednaska.pdf.
14. *Opencart* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://www.clickpost.ai/blog/opencart.
15. RICHARDSON, Chris. Pattern: Mobolithic Architecture [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://microservices.io/patterns/monolithic.html.

16. MLEJNEK, Jiří. *Analýza a sběr požadavků* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/506241/mod_resource/content/7/03.prednaska.pdf.
17. *What is PHP* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://www.php.net/manual/en/intro-whatis.php.
18. *What is Symfony* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://symfony.com/what-is-symfony.
19. *Doctrine* [online]. [B.r.] [cit. 2022-04-22]. Dostupné z: https://www.doctrine-project.org/.
20. *Introducing JSON* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://www.json.org/json-en.html.
21. *Representational State Transfer (REST)* [online]. [B.r.] [cit. 2022-04-06]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.
22. *Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC* [online]. [B.r.] [cit. 2022-04-29]. Dostupné z: https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/.
23. Benefit of using MVC [online]. 2021 [cit. 2022-04-06]. Dostupné z: https://www.geeksforgeeks.org/benefit-of-using-mvc/.
24. *Databases and the Doctrine ORM* [online]. [B.r.] [cit. 2022-04-21]. Dostupné z: https://symfony.com/doc/current/doctrine.html#querying-for-objects-the-repository.
25. *A Guide to Understanding Business Logic and Software* [online]. [B.r.] [cit. 2022-04-21]. Dostupné z: https://www.integrify.com/what-is-business-logic/.
26. NADEL, Ben. *A Better Understanding Of MVC (Model-View-Controller) Thanks To Steven Neiland* [online]. [B.r.] [cit. 2022-04-21]. Dostupné z: https://www.bennadel.com/blog/2379-a-better-understanding-of-mvc-model-view-controller-thanks-to-steven-neiland.htm.
27. *Forms* [online]. [B.r.] [cit. 2022-04-21]. Dostupné z: https://symfony.com/doc/current/forms.html.
28. *Testing* [online]. [B.r.] [cit. 2022-04-22]. Dostupné z: https://symfony.com/doc/current/testing.html.
29. *PHPStan* [online]. [B.r.] [cit. 2022-04-24]. Dostupné z: https://phpstan.org/.
30. *FOSRestBundle* [online]. [B.r.] [cit. 2022-04-22]. Dostupné z: https://github.com/FriendsOfSymfony/FOSRestBundle.
31. *About MySQL* [online]. [B.r.] [cit. 2022-04-22]. Dostupné z: https://www.mysql.com/about/.
32. CAREY, Scott. *What is Docker? The spark for the container revolution* [online]. [B.r.] [cit. 2022-04-24]. Dostupné z: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html.
33. *Overview of Docker Compose* [online]. [B.r.] [cit. 2022-04-24]. Dostupné z: https://docs.docker.com/compose/.
34. *Welcome to NGINX Wiki* [online]. [B.r.] [cit. 2022-04-22]. Dostupné z: https://www.nginx.com/resources/wiki/.
35. FÁBERA, Vít. *Transakce v databázových systémech* [online]. [B.r.] [cit. 2022-04-23]. Dostupné z: https://www.fd.cvut.cz/personal/xfabera/BIVS/DB1/prednasky/prednaska10/databaze_10.pdf.

36. *DependencyInjection* [online]. [B.r.] [cit. 2022-04-23]. Dostupné z: https://www.tutorialsteacher.com/ioc/dependency-injection.

37. *Doctrine Query Language* [online]. [B.r.] [cit. 2022-04-23]. Dostupné z: https://www.doctrine-project.org/projects/doctrine-orm/en/2.9/reference/dql-doctrine-query-language.html.

38. *What are URL parameters?* [Online]. [B.r.] [cit. 2022-04-23]. Dostupné z: https://www.botify.com/learn/basics/what-are-url-parameters.

39. *Pagerfanta Documentation* [online]. [B.r.] [cit. 2022-04-23]. Dostupné z: https://www.babdev.com/open-source/packages/pagerfanta/docs/3.x/intro.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `api` ..... Dokumentace API vytvořená v nástroji OpenApi
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF