FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	E-obchod s podporou logistiky rozvozu zboží
Student:	Tomáš Greger
Vedoucí:	Ing. Jiří Daněček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je navrhnout, implementovat a otestovat e-obchod s podporou logistiky rozvozu zboží.

1) Proveďte rešerši algoritmů pro vyhledávání cest v grafu a na základě rešerše vyberte nejvhodnější algoritmus pro účel řízení logistiky rozvozu zboží.
2) Implementujte ukázkový e-obchod se zvoleným grafovým algoritmem pomocí frameworku Spring Boot.
3) Prezentační část vytvořte v technologii Angular.
4) Funkčnost aplikace otestujte na vhodných datech.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 26. listopadu 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

E-obchod s podporou logistiky rozvozu zboží

Tomáš Greger

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Daněček

10. května 2018

Poděkování

Chtěl bych poděkovat především své rodině a přátelům, kteří mě během studia plně podporovali. Dále bych chtěl poděkovat vedoucímu své bakalářské práce Ing. Jiřímu Daněčkovi za spolupráci, rady a připomínky během tvorby práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

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

V Praze dne 10. května 2018 ……

Odkaz na tuto práci

Greger, Tomáš. *E-obchod s podporou logistiky rozvozu zboží.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem této práce je analyzovat, navrhnout, implementovat a otestovat webovou aplikaci ve formě e-obchodu. Teoretická část práce se zabývá rešerší grafových algoritmů vhodných pro řízení rozvozu zboží. V implementační části je kladen důraz na použití moderních technologií (Spring Boot, Angular) a moderních přístupů (HATEOAS), které jsou zde detailněji rozepsány. Aplikace je snadno rozšiřitelná vzhledem k rozdělení aplikace do jednotlivých modulů. Výsledkem práce je e-obchod s podporou logistiky rozvozu zboží, platby přes platební portál PayPal, práce s fakturami…Výsledná aplikace je dostupná na serveru Heroku.

Klíčová slova e-obchod, grafové algoritmy, hledání nejkratších cest, Dijkstrův algoritmus, Spring Boot, Angular

Abstract

The goal of this work is to analyze, design, implement and test web application in form of e-shop. Theoretic part contains research of graph algorithms suitable for goods distribution logistics. In implementing part is put emphasis on modern technologies (Spring Boot, Angular) and modern patterns (HATEOAS), which are here described. Application is easily extendable because of application division into several modules. Result of this work is e-shop with support of goods distribution logistics, payment via PayPal, work with invoices…Final application is available on Heroku server.

Keywords e-shop, graph algorithms, shortest path finding, Dijkstra's algorithm, Spring Boot, Angular

Obsah

Seznam obrázků

Úvod

Nakupování v internetových obchodech je dnes běžnou záležitostí. Dnešní internet si bez e-obchodů lze představit jen stěží. V době, kdy se čas posouvá v lidském žebříčku hodnot na přední místa, lidé odmítají trávit svůj čas v klasických kamenných obchodech. Na internetu se dnes dá sehnat nejrůznější spektrum zboží od potravin, přes oblečení či elektroniku. Pro nové podnikatele na internetu však není často jednoduché prorazit, jelikož internet je doslova přehlcený prodejci, kteří tak bojují o každého zákazníka. Objednáním zboží přes internet však nákup nekončí. Zboží je potřeba zákazníkovi doručit v co nejkratší době a za co nejméně peněz. V tomto spočívá jedna z konkurenčních výhod, které může nový podnikatel svým potencionálním zákazníkům nabídnout. Tato práce se věnuje tvorbě e-obchodu s podporou logistiky rozvozu zboží a obsahuje všechny fáze klasického softwarového procesu, od analýzy, návrhu, implementaci až po testování vzniklé aplikace.

Kapitola 1

Teoretická část

1.1 Zavedení pojmů

1.1.1 Graf

Graf definujeme jako uspořádanou dvojici *(V,E)*, kde V je neprázdná množina vrcholů a E je množina hran. Rozlišujeme dva typy grafů, orientované a neorientované. Tyto dva typy se od sebe liší definicí hrany. Hranu neorientovaného grafu definujeme jako neuspořádanou dvojici vrcholů *(u,v)* na rozdíl od orientovaného, kde je hrana definována jako orientovaná dvojice *(u,v)*, kde u se nazývá předchůdce a v následník. [1]

1.1.2 Ohodnocený graf

Graf, jehož hrany jsou ohodnoceny nějakou vahou, nazýváme ohodnocený. [2] Váha je udána nejčastěji v číselné podobě a může označovat, jako v našem případě, například čas, který strávíme při cestě z jednoho vrcholu do druhého nebo počet kilometrů mezi těmito vrcholy.

1.1.3 Cesta a nejkratší cesta

Cesta je taková posloupnost vrcholů a hran, kde se vrcholy neopakují. Nejkratší cestou mezi vrcholy u a v nazveme takovou cestu, jejíž součet hodnot všech hran je v rámci všech cest mezi u a v nejmenší. [3]

1.1.4 Souvislý graf

Graf je souvislý, pokud mezi každými dvěma vrcholy existuje cesta. [3]

1.2 Požadavky na algoritmus pro rozvoz zboží

Algoritmus musí umět spočítat optimální cesty mezi daným počátečním městem a cílovými městy pro dva případy:

- nejkratší
- nejrychlejší

Vzhledem k tomu, že algoritmus bude spouštěn maximálně v rámci jednotek za den, nesehrává v tomto případě rychlost algoritmu rozhodující roli. Algoritmus bude použit jen pro řízení rozvozu velkého zboží. Vzhledem k předpokládané četnosti nákupu takového zboží bude rozvoz realizován po jednom. Algoritmus tedy nebude řešit tzv. problém obchodního cestujícího.

1.3 Rešerše algoritmů pro hledání nejkratších cest

Pro problém hledání nejkratší cesty v grafu existuje několik algoritmů. Součástí rešerše budou tyto:

- Dijkstrův algoritmus
- Bellman-Fordův algoritmus
- Floyd-Warshallův algoritmus

V rámci rešerše těchto algoritmů předpokládejme jako vstup vždy souvislý graf.

1.3.1 Dijkstrův algoritmus

1.3.1.1 Popis

[4] Tento algoritmus, který v roce vymyslel v roce 1959 Edsger Dijkstra, slouží k hledání nejkratších cest v grafu s nezáporně ohodnocenými hranami. Algoritmus nedokáže najít pouze cestu mezi počátečním a koncovým vrcholem, ale i nejkratší cesty mezi počátečním a všemi ostatními vrcholy grafu.

1.3.1.2 Princip

1. V první fázi se všem vrcholům nastaví hodnota na ∞, do otevřených vrcholů přidáme vrchol počáteční a nastavíme mu hodnotu na 0.
2. Dalším krokem algoritmu je najít takový vrchol z množiny otevřených vrcholů, který má nejmenší hodnotu. Pro všechny hrany e, které vycházejí z vybraného vrcholu u, spočítáme hodnotu h, jako součet hodnoty vrcholu u a hodnoty hrany e. Pokud platí, že hodnota h je menší než

hodnota vrcholu v, nastavíme hodnotu vrcholu v na hodnotu h, nastavíme vrchol u jako předchůdce vrcholu v a zařadíme tento vrchol do otevřených uzlů.

3. Algoritmus pokračuje bodem 2, dokud není množina vrcholů prázdná.

1.3.1.3 Výstup

Výstupem algoritmu je graf, kde každý vrchol má definovanou hodnotu vzdálenosti od počátečního vrcholu a vrchol, který je jeho předchůdcem na nejkratší cestě z počátečního vrcholu. [4]

1.3.2 Složitost

Časová složitost Dijkstrova algoritmu při využití haldy k ukládání otevřených vrcholů je $\mathcal{O}((n+m)*\log(n))$, bez použití haldy $\mathcal{O}(n^2)$, kde n je počet vrcholů grafu a m je počet hran.

1.3.3 Bellman-Fordův algoritmus

1.3.3.1 Popis

[5] Na rozdíl od Dijkstrova algoritmu, tento algoritmus dokáže najít nejkratší cesty mezi počátečním a všemi ostatními vrcholy grafu s hranami, které mohou mít zápornou hodnotu. Jedinou podmínkou pro korektní výsledek nejkratších cest je neexistence záporného cyklu v grafu. Tento cyklus však algoritmus dokáže odhalit.

1.3.3.2 Princip

1. V první fázi se všem vrcholům nastaví hodnota na ∞, do otevřených vrcholů přidáme počáteční vrchol, nastavíme mu hodnotu na 0 a vložíme ho do fronty.

2. Odebereme vrchol u z fronty. Pro všechny hrany e, které vycházejí z vybraného vrcholu u, spočítáme hodnotu h, jako součet hodnoty vrcholu u a hodnoty hrany e. Pokud platí, že hodnota h je menší než hodnota vrcholu v, nastavíme hodnotu vrcholu v na hodnotu h, nastavíme vrchol u jako předchůdce vrcholu v a pokud není otevřený, zařadíme jej do fronty.

3. Algoritmus pokračuje bodem 2, dokud není fronta prázdná.

1.3.3.3 Výstup

Pokud graf neobsahuje záporné cykly, je výstup Bellman-Fordova algoritmu stejný jako u Dijkstrova algoritmu, jinak je existence záporného cyklu identifikována. [5]

1.3.4 Složitost

Časová složitost Bellman-Fordova algoritmu je $\mathcal{O}(n*m)$, kde n je počet vrcholů grafu a m je počet hran.

1.3.5 Floyd-Warshallův algoritmus

1.3.5.1 Popis

[6] Tento algoritmus je schopný nalézt nejkratší cestu mezi každou dvojicí vrcholů grafu, který neobsahuje záporné cykly.

1.3.5.2 Princip

„*Floyd-Warshallův má charakter postupného zpřesňování, při kterém rozšiřujeme množinu přípustných hodnot vnitřních uzlů nejkratších cest. Vnitřním uzlem cesty je každý uzel cesty s výjimkou krajních uzlů.*“ [7, s. 129]

1. Algoritmus nejdříve vytvoří matici délek (pokud mezi dvěma vrcholy *(i, j)* vede hrana, tak tato matice obsahuje na indexu *(i, j)* hodnotu této hrany, na diagonále se nacházejí 0 a na ostatních indexech ∞).

2. V každé iteraci algoritmus přepočítá hodnoty nejkratších cest v matici za použití neustále se zvětšující se množiny přípustných prostředníků. Po první iteraci bude matice vyjadřovat vzdálenost všech uzlů s možností využití jednoho prostředníka, po druhé iteraci vzdálenost při možném využití dvou prostředníků…

1.3.5.3 Výstup

Výstupem algoritmu jsou matice nejkratších cest a matice předchůdců. [6]

1.3.6 Složitost

Časová složitost Floyd-Warshallova algoritmu je $\mathcal{O}(n^3)$, kde n je počet vrcholů grafu.

1.3.7 Zvolený algoritmus

Úkolem vybraného algoritmu je nalézt nejkratší cesty z počátečního vrcholu do cílových. Hrany grafu, které v rámci aplikace reprezentují silniční síť, nemohou

být záporně ohodnocené (počet minut, počet kilometrů). V aplikaci jsem se rozhodl použít Dijkstrův algoritmus z těchto důvodů:

- Přímo řeší výše zmíněný problém.
- Ze všech tří zkoumaných algoritmů je pro daný problém nejvhodnější.
- Pracuje efektivně na grafu s nezáporně ohodnocenými hranami.

Kapitola 2

Analýza a návrh

2.1 Současný stav a řešení

V dnešní době není tvorba e-obchodu od základu příliš obvyklá. Na internetu lze nalézt velké množství firem, které nabízejí možnost založit si e-obchod na jejich vlastní platformě a nechat veškerou správu aplikace v jejich režii. Pro tvorbu většiny dnešních e-obchodů se používají především jazyky JavaScript, Python nebo PHP. [8]

Pokud chce zákazník svůj e-obchod přizpůsobit svým představám a dále rozvíjet, je výhodnější implementovat svůj vlastní. Volba programovacího jazyku a architektury aplikace tak zůstává v rukou zákazníka.

2.2 Funkční požadavky

1. Správa produktů, uživatelů a objednávek včetně editace.
2. Tvorba objednávek s různými typy platby a způsoby doručení.
3. Generování a aktualizace faktur při tvorbě objednávky nebo změně platebních/doručovacích údajů.
4. Integrace na platební portál PayPal.
5. Vkládání obrázků k produktům ve formátu (PNG, JPG, GIF) o definované maximální velikosti.
6. Přikládání a stahování příloh k produktům o definované maximální velikosti.
7. Stahování reportu pro rozvoz zboží ve formátu .xlsx (Microsoft Excel).

2.3 Nefunkční požadavky

1. **Zabezpečení**
   - Aplikace bude zabezpečena dvěma typy rolí (správce a zákazník).
   - Uživatel s rolí „správce“ bude moct spravovat produkty, objednávky a uživatelské účty. Dále mu bude umožněno stáhnout report pro rozvoz zboží.
   - Uživatel s rolí „zákazník“ bude moci objednávat zboží, upravovat svůj profil a prohlížet své objednávky.
   - Prohlížení dostupných produktů bude přístupné všem uživatelům.
2. **Rozšiřovatelnost a udržovatelnost**
   - Aplikace bude rozdělena do jednotlivých, logicky rozdělených, modulů.
   - Aplikace bude využívat principu HATEOAS, pokud to bude možné vzhledem k použitým technologiím.
3. **Dostupnost a výkonnost nasazené aplikace**
   - Žádné požadavky na dostupnost a výkonnost aplikace nasazené na serveru Heroku nelze garantovat. Součástí práce tak bude samostatně spustitelný JAR soubor.
4. **Podpora**
   - Aplikace bude určena pro desktopové počítače.
   - Aplikace bude zaručeně podporovaná webovým prohlížečem Google Chrome (verze 65.0.3325.181 a vyšší).

2.4 Architektura aplikace

Aplikace bude postavena na klasické třívrstvé architektuře. Jak již z názvu vyplývá, třívrstvá architektura rozděluje aplikace do tří základních vrstev – datová, aplikační a prezentační.

Aplikace bude rozdělena do těchto modulů:

- **application** – bude obsahovat závislosti na všechny ostatní moduly a samotnou spouštěcí třídu.
- **base-app** – bude obsahovat funkcionalitu společnou pro celou aplikaci (typicky veškeré konfigurace, zabezpečení, řešení výjimek, definici společných tříd).

- **db** – bude obsahovat databázové skripty pro vytvoření databáze a vložení iniciálních dat na server Heroku. [9]
- **services** – bude obsahovat jednotlivé moduly zajišťující funkcionalitu aplikace (**payment**, **shop**, **transport**)
- **ui** – bude obsahovat Angular aplikaci (zkompilovaná verze bude vložena do statického obsahu výsledné aplikace).

Výsledná architektura aplikace je zobrazena na obrázku 2.1 .

2.5 Diagram případů užití

Diagram případů užití (z anglického Use Case Diagram) je diagram, jehož účelem je popsat funkcionalitu systému v závislosti na roli aktéra (ang. Actor). Aktér je definován jako role, která komunikuje mezi jednotlivými případy užití. Diagram však nedefinuje, jak bude daná funkcionalita vyřešena technicky. V jednotlivých případech užití by se také měly objevit funkční požadavky. [10]

2.6 Návrh uživatelského rozhraní

Při návrhu uživatelského rozhraní jsem se soustředil především na jednoduchost a přehlednost obrazovek. Některé dnešní webové aplikace uživatele přímo zahlcují blikajícími tlačítky, různými reklamami nebo oznámeními, což je pro uživatele v mnohých případech velice matoucí, až nepříjemné.

2.6.1 Hlavička

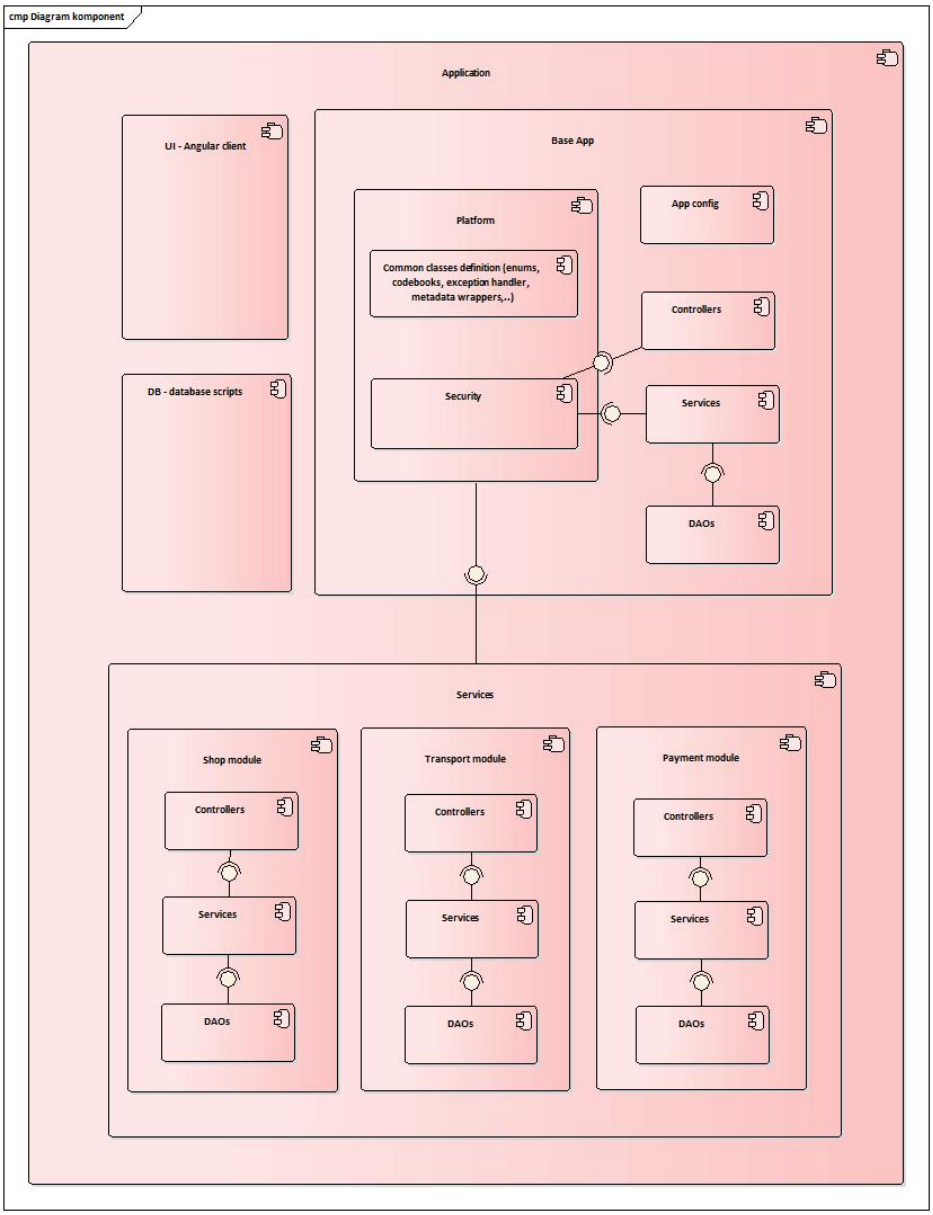
Každá obrazovka obsahuje společnou hlavičku (obrázek 2.3 a 2.4), která slouží k základní navigaci v rámci aplikace. Zobrazení tlačítek se řídí podle role přihlášeného uživatele.

2.6.2 Přehled

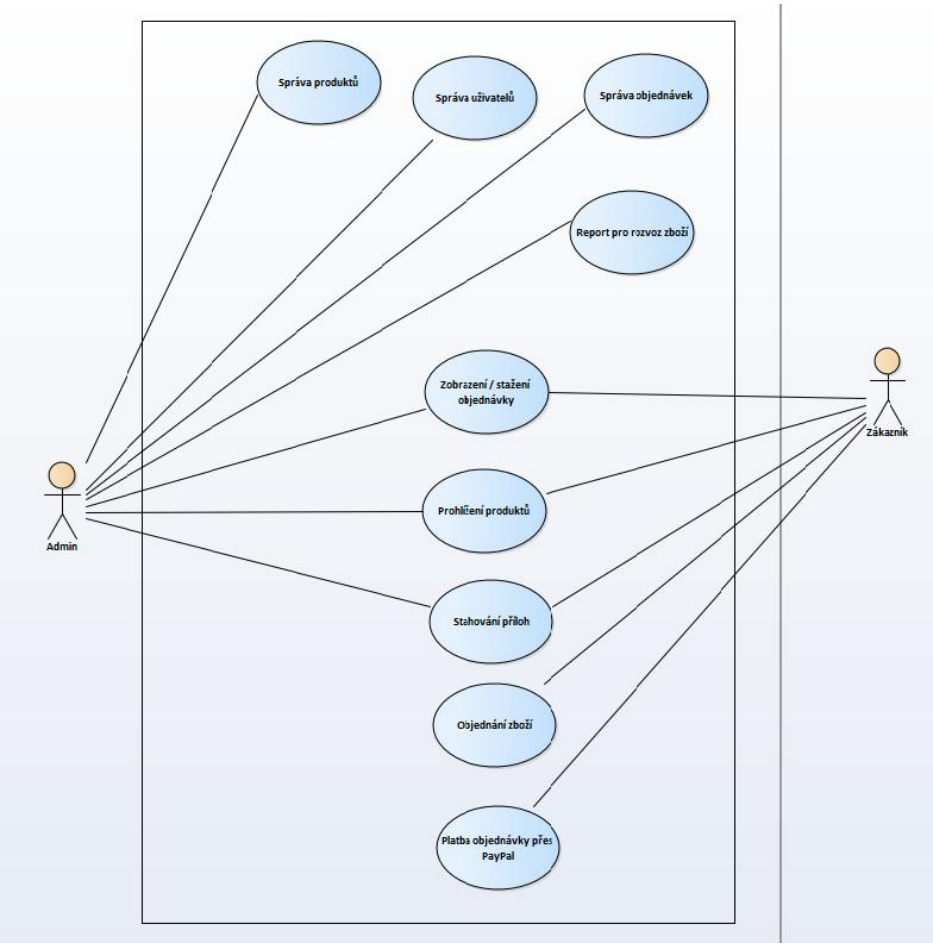
Obrazovky s přehledem (produktů, uživatelů, objednávek) je rozdělena do dvou sekcí. První sekce slouží pro vstupy a tlačítka pro filtrování výsledků a sekce druhá je určena pro samotné zobrazení vyhledaných výsledků.

2.6.3 Detail

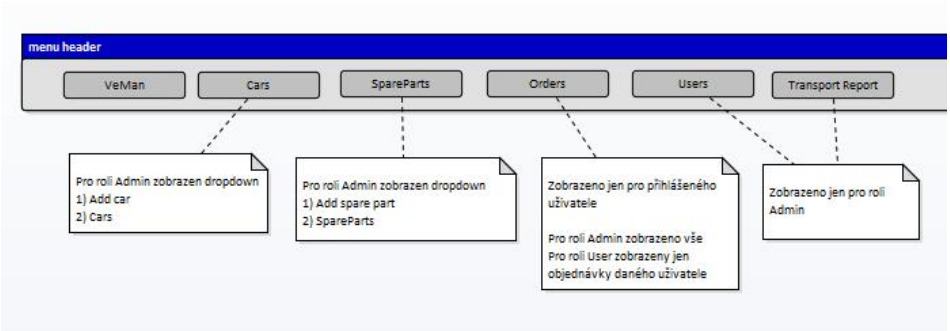
Další kategorií jsou obrazovky pro zobrazení detailu produktu, obrázek 2.6. Obrazovka mimo samotných údajů o produktu obsahuje i sekci obrázků a přiložených příloh.



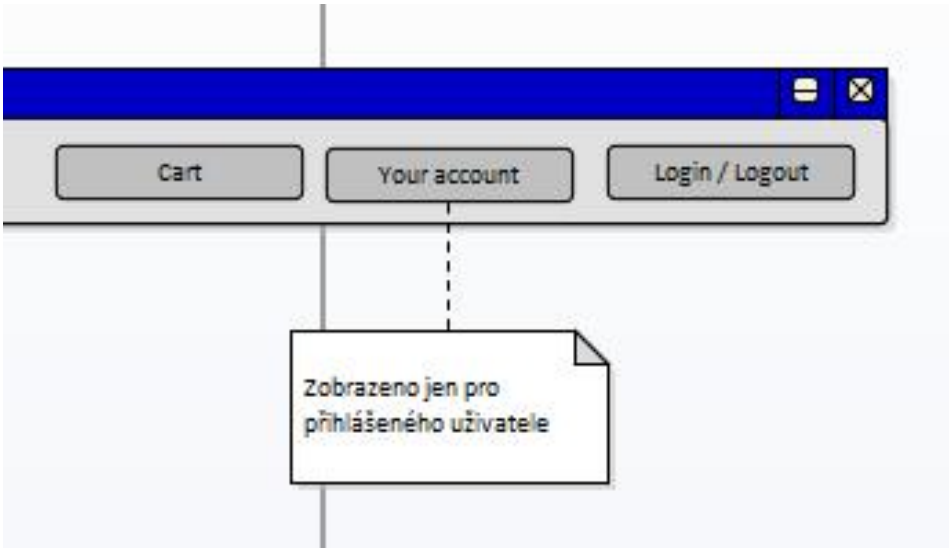
Obrázek 2.1: Diagram komponent



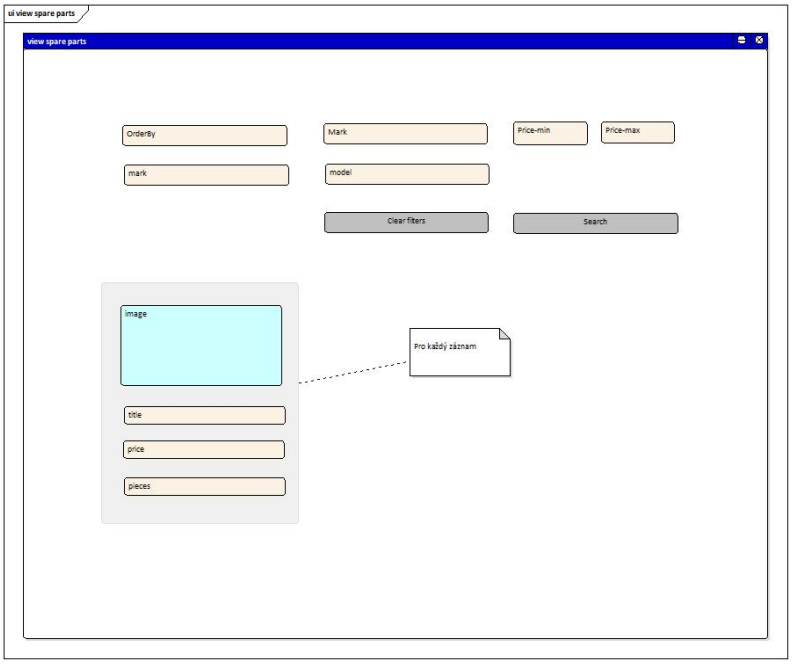
Obrázek 2.2: Diagram případů užití



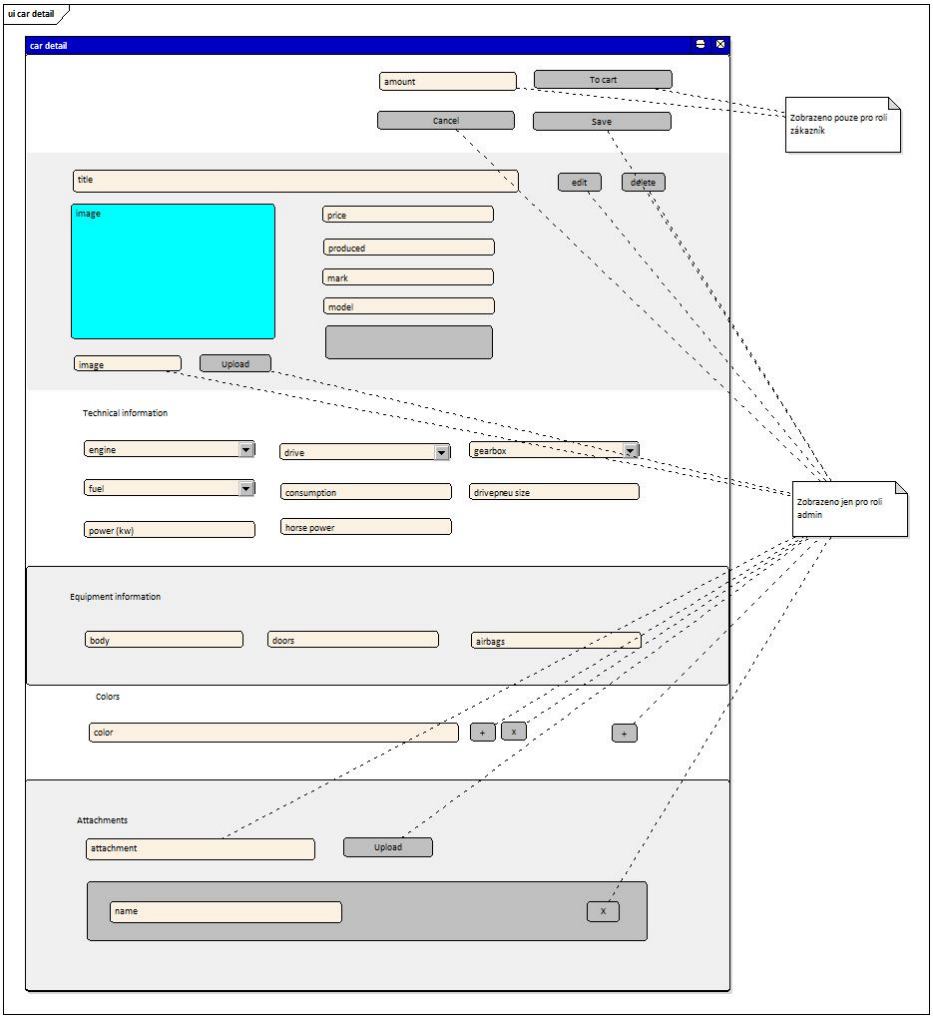
Obrázek 2.3: Uživatelské rozhraní – hlavička 1



Obrázek 2.4: Uživatelské rozhraní – hlavička 2



Obrázek 2.5: Uživatelské rozhraní – přehled produktů



Obrázek 2.6: Uživatelské rozhraní – detail produktu

Kapitola 3

Implementace

3.1 Volba programovacího jazyka a frameworku

V souladu se zadáním jsem si vybral pro backendovou část aplikace framework Spring Boot (verze 1.5.6) postavený na jazyku Java (verze 1.8) a pro frontendovou část Angular (verze 4.4.6).

3.1.1 Backendová část

3.1.1.1 Spring

[11] Spring je frameworkem pro tvorbu moderních enterprise aplikací v jazyku Java vyvíjený společností Pivotal. Strukturu samotného frameworku tvoří několik modulů se základním modulem Core. Spring poskytuje moduly pro různé služby jako jsou DataAccess/Integration, Web, AOP (Aspect Oriented Programming), Test a mnoho dalších.

3.1.1.2 Spring boot

Spring Boot framework ze Springu vychází a přináší především autokonfiguraci aplikace na základě připojených knihoven. Umožňuje programátorům soustředit se na logiku programu, na rozdíl od občas složité konfigurace samotné aplikace. Spring Boot mimo jiné přináší i vestavěný Tomcat server, což umožňuje spouštět aplikaci jako obyčejný JAR soubor, tedy odpadá potřeba nahrávat aplikaci na vývojový server. [12]

3.1.2 Frontendová část

3.1.2.1 Angular

[13] Angular je platforma pro tvorbu webových single-page aplikací. Umožňuje vyvíjet aplikace jak pro desktopové počítače, tak i pro mobilní zařízení.

Angular aplikace je nejčastěji psána v jazyce TypeScript, který je při sestavení aplikace kompilován do JavaScriptu.

3.2 Další použité vývojové nástroje

3.2.1 Maven

Maven je nástroj pro správu, řízení a automatizaci buildů aplikací od společnosti Apache. Základním elementem každého Maven projektu je Project Object Model (definováno v xml struktuře v souboru pom.xml), který popisuje projekt, jeho závislosti, proces sestavení aplikace a další. Artekfakty jsou Mavenem automaticky vyhledávány v definovaných repozitářích (globální, firemní,..) a instalovány do projektu. [14]

3.2.2 Node.js

Platforma pro vývoj desktopových i webových aplikací postavená na jazyku JavaScript. Samotné Node.js je použito při transformaci TypeScriptu do JavaScriptu, se kterým webové prohlížeče dokáží pracovat. [15] Node.js dále nabízí širokou škálu knihoven a nástroj Npm, který je s ním distribuován. [16]

3.2.3 Npm

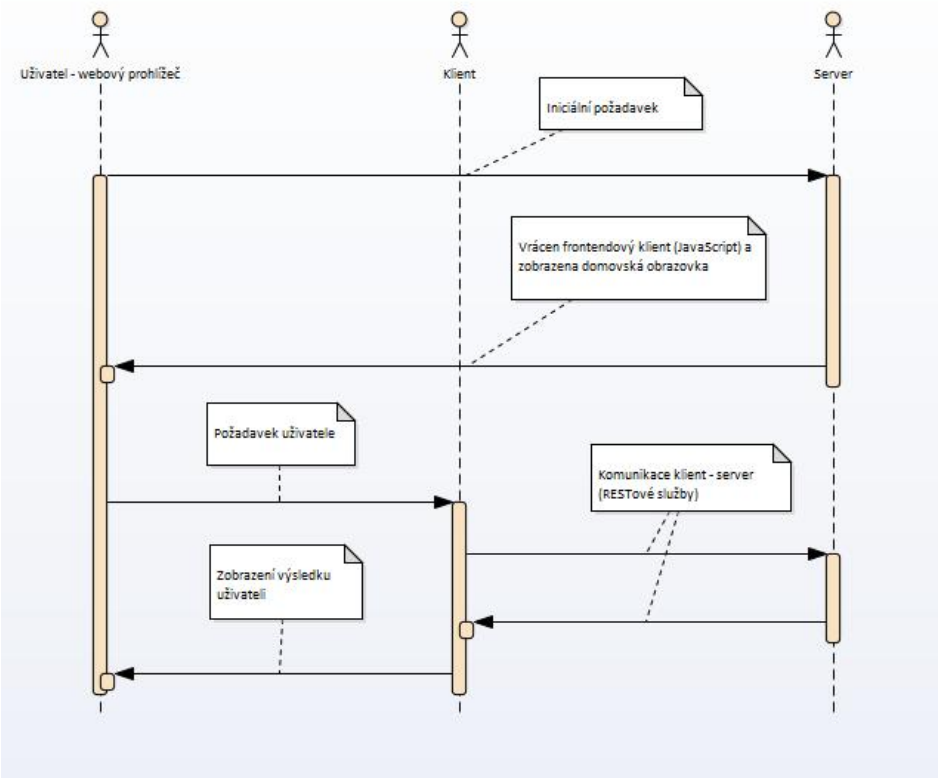
Npm je jedním z nejpoužívanějších správců JavaScriptových balíčků. Závislosti Angular aplikace jsou definovány v souboru package.json. Po spuštění příkazu npm install Npm zavede požadované balíčky do projektu. [17]

3.2.4 Git

Verzovací systém použitý pro správu zdrojových kódů, dokumentace i samotné práce. Umožňuje udržovat přehlednou historii celého projektu. [18]

3.3 Základní princip aplikace

Celá aplikace je rozdělena na backendový server a frontendového klienta. Při prvním požadavku na server je uživateli zaslána celá klientská část. Uživatel následně komunikuje pouze s klientskou částí, která získává data od serveru prostřednictvím RESTových služeb. Ze serverové části tedy chodí pouze data, definice HTML šablon jsou obsaženy přímo v klientské části. Pro přenos dat mezi klientem a serverem se používá především formát JSON. Schéma komunikace je zobrazena na obrázku 3.1.



Obrázek 3.1: Schéma komunikace v rámci aplikace

3.4 Zabezpečení

Zabezpečení je jednou z nejdůležitějších funkcionalit celé aplikace. Jelikož serverová část je Stateless (tj. neudržuje žádný stav), je nutné při každém požadavku na server ověřit totožnost přihlášeného uživatele a zajistit podle jeho rolí odpovídající přístup do aplikace. K tomuto přístupu můžeme použít zabezpečení pomocí cookie nebo tokenu. Vzhledem k tomu, že JWT (Json Web Token) na rozdíl od cookie dokáže zamezit CSRF útoku, jsem se rozhodl v aplikaci použít JWT.

3.4.1 Json Web Token

[19] JSON Web Token (JWT) je standard, jak bezpečně přenášet informaci mezi dvěma částmi jako JSON objekt. Těmto informacím můžeme důvěřovat a ověřit je, jelikož jsou digitálně podepsány.

Základní koncepty JWT:

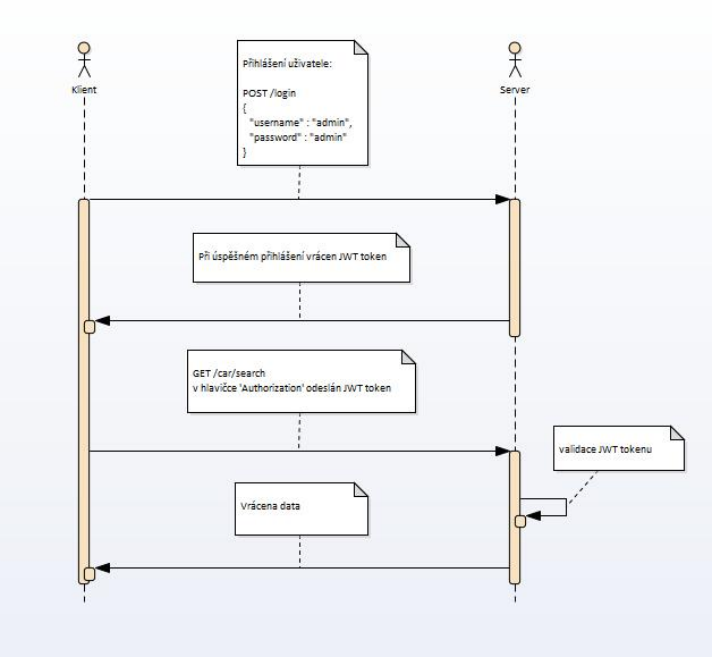
- Kompaktnost – kvůli své velikosti může být JWT poslán serverové části v URL, jako POST parametr nebo jako součást HTTP hlavičky.

- Soběstačnost – token obsahuje veškeré informace o přihlášeném uživateli a tím se vyhýbá následnému vícenásobnému přístupu do databáze.

Jelikož serverová část aplikace při práci s citlivými údaji vyžaduje přihlášení a přihlašovací token do každého požadavku na server vkládá pouze klientská část, útoku je tím zamezeno.

3.4.2 Princip zabezpečení

Zabezpečení pomocí JWT tokenu začíná přihlášením. Po úspěšném přihlášení serverová část vygeneruje token, který vloží do hlavičky HTTP odpovědi. Frontendová část aplikace si následně token uloží a při každém dalším požadavku na server je token vložen do hlavičky HTTP požadavku. Serverová část si z tokenu zjistí přihlášeného uživatele a jeho role a zajistí odpovídající přístup do aplikace. Obrázek 3.2 zobrazuje vzorovou komunikaci uživatele se serverovou částí.



Obrázek 3.2: Zabezpečení – Json Web Token

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable().authorizeRequests()
            .antMatchers(HttpMethod.GET, api_prefix +
                ApiLinks.URL_HOMEPAGE,
                    api_prefix + ApiLinks.URL_CAR_FILTER_INPUTS,
                        api_prefix + ApiLinks.URL_CAR_DETAIL,
                    api_prefix + ApiLinks.URL_CAR_SEARCH, api_prefix +
                        ApiLinks.URL_COLORS_FOR_PRODUCT,
                    api_prefix + ApiLinks.URL_SPAREPART_SEARCH,
                        api_prefix + ApiLinks.URL_SPAREPART_DETAIL,
                    api_prefix + ApiLinks.URL_SPAREPART_FILTER_INPUTS,
                    api_prefix + ApiLinks.URL_ATTACHMENTS_FOR_PRODUCT,
                        api_prefix + ApiLinks.URL_IMAGES_FOR_PRODUCT,
                    api_prefix + ApiLinks.URL_FILE, api_prefix +
                        ApiLinks.URL_CODEBOOK_CITIES)
                    .permitAll()
            .antMatchers(HttpMethod.POST, "/login",
                    api_prefix + ApiLinks.URL_CAR_FILTER,
                    api_prefix + ApiLinks.URL_SPAREPART_FILTER,
                    api_prefix + ApiLinks.URL_USER_NEW)
                    .permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilterBefore(new JWTLoginFilter("/login",
                authenticationManager(), userService),
                    UsernamePasswordAuthenticationFilter.class)
            .antMatcher("/api/**").addFilterBefore(new
                JWTAuthenticationFilter(),
            UsernamePasswordAuthenticationFilter.class);
}
```

Zdrojový kód 3.1: Zabezpečení přes URL

3.4.2.1 Zabezpečení na úrovni URL

Zabezpečení na úrovni URL dovoluje zabezpečit aplikaci nejen nutností být přihlášen, ale i pomocí rolí. Dovoluje také rozlišit zabezpečení i podle jednotlivých HTTP metod. V přiloženém zdrojovém kódu 3.1 můžeme vidět definici zabezpečení pomocí URL. Hlavně kvůli příliš veliké pravděpodobnosti potenciálních chyb se zde neberou v potaz role přihlášeného uživatele. Zabezpečení pomocí rolí se věnuje následující kapitola.

3.4.2.2 Zabezpečení pomocí rolí

Jak již bylo zmíněné výše, aplikace musí v rámci zabezpečení rozlišovat role přihlášeného uživatele. Zajisté není žádoucí, aby kdokoli mohl měnit cizí ob-

```
@Secured({Role.ROLE_ADMIN})
@Transactional
@Override
public void deleteUser(Integer userId) {
    logger.info(String.format("Delete user with id: %d", userId));
    userDAO.delete(userId);
}
```

Zdrojový kód 3.2: Zabezpečení přes anotace

jednávky nebo prohlížet cizí uživatelské účty. Pro zabezpečení pomocí rolí uživatelů je každá servisní metoda (pokud je to potřeba) označena Spring security anotací. Ve výpisu zdrojové kódu 3.2 můžeme vidět, že metoda označená touto anotací se provede pouze v případě, pokud přihlášený uživatel disponuje rolí „admin“.

3.4.3 Zabezpečení dat konkrétního uživatele

Ani použitím výše zmíněných zabezpečení nedosáhneme korektního chování. Příkladem může být funkcionalita aplikace sloužící k vyhledání zákazníka dle jeho identifikátoru. Tato metoda je zabezpečena rolí „zákazník“. To však nezabrání po přihlášení daného zákazníka zobrazit profily zákazníků jiných, a to jednoduchou změnou identifikátoru v URL.

Nejjednodušším řešením je zkontrolovat možnost přístupu k žádaným datům přímo v kódu metody, to však z hlediska čistoty kódu není ideální přístup. Z toho důvodu jsem jako řešení zvolil použití aspektově orientovaného programování.

Aspektově orientované programovaní si bere za cíl, aby každá třída obstarávala konkrétní funkčnost aplikace, tedy například nemíchala bussiness logiku se zabezpečením aplikace. Samotné aspektově orientované programovaní funguje na principu proxy tříd. [20]

Aspekt definovaný ve výpisu kódu 3.3 způsobí, že před každým zavoláním metody označené anotací „@CurrentUserOnly“ je zjištěno, zda je přihlášený uživatel oprávněn k zobrazení žádaných dat – jedná se o správce nebo o daného uživatele. Výsledná metoda s použitím definované anotace je zobrazena na výpisu kódu 3.4.

3.5 Princip HATEOAS

HATEOAS (zkratka z anglického Hypermedia as the Engine of Application State) je princip, který definuje, že kromě samotných dat dostáváme ze serverové části i informace o dalších možných krocích v rámci aplikace ve formě odkazů. HATEOAS je spjatý s architekturou RESTových služeb. [21]

```
@Aspect
@Component
public class ShopPermissionChecker {

    @Before(value = "@annotation(
           cz.gregetom.shop.platform.security.CurrentUserOnly)")
    public void checkUser(JoinPoint joinPoint) {
        final Integer userId = (Integer) joinPoint.getArgs()[0];
        logger.info(String.format("Check user with id %d",
            userAccessor.getCurrentId()));
        if (!userAccessor.hasAnyRole(Role.ROLE_ADMIN) &&
            !userAccessor.hasId(userId)) {
            throw new ForbiddenException();
        }
    }
}
```

Zdrojový kód 3.3: Zabezpečení dat konkrétního uživatele

```
@CurrentUserOnly
@Secured({Role.ROLE_USER})
@Transactional(readOnly = true, propagation = Propagation.SUPPORTS)
@Override
public Optional<User> findUserById(Integer userId) {
    logger.info(String.format("Find user with id: %d", userId));
    return Optional.ofNullable(userDAO.findById(userId));
}
```

Zdrojový kód 3.4: Zobrazení uživatele dle indentifikatoru

Tento princip umožňuje průchod v rámci celé aplikace pouze s jedinou známou URL (vstupní URL do aplikace). Pokud frontendová část potřebuje získat data ze serverové části, odkaz na tyto data je zaslán již při předchozím požadavku a není nutné držet odkazy na serverové API přímo v klientovi. Princip fungování je zobrazen na obrázku 3.3.

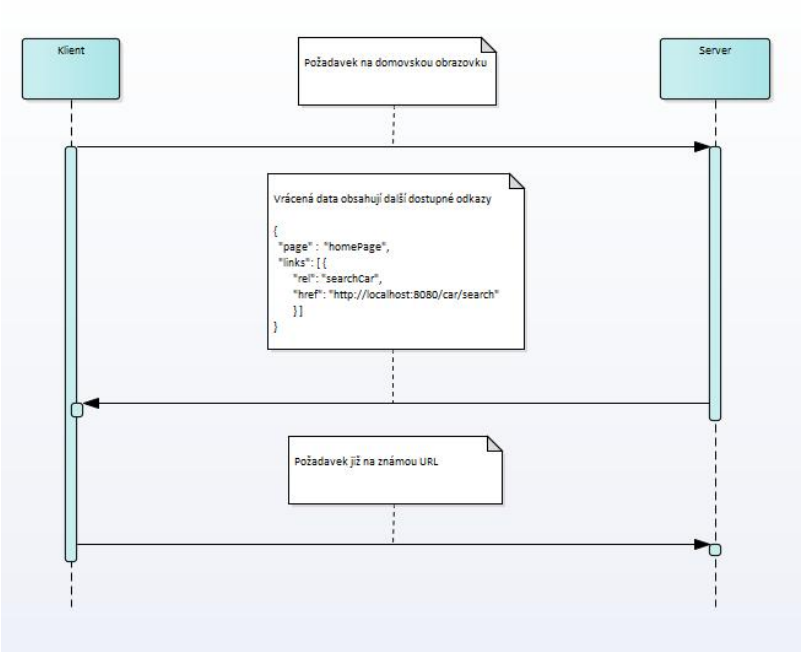
3.5.1 Datový model

Pro webové aplikace bývá zvykem rozlišovat databázové a frontendové entity. Zpravidla se na frontendovou část zasílají dodatečné informace, které se z hlediska udržovatelnosti kódu nemají vázat k databázové entitě. Frontendová entita obsahuje nejen databázovou entitu jako asosiaci, ale i dodatečné informace, například ty pro HATEOAS. Na obrázku 3.5 můžeme vidět příklad takové entity ve formátu JSON.

```
{
"id" : 1,
"mark": "Audi",

"links": [ {
    "rel": "self",
    "href": "http://localhost:8080/car/1"
    } ]
}
```

Zdrojový kód 3.5: HATEOAS datový model



Obrázek 3.3: HATEOAS – princip

3.5.2 Tvorba odkazů na serveru

Pro uchování odkazů je v aplikaci definována třída, ze které dědí všechny třídy určené pro datový kontrakt s frontendem. Definice této třídy je zobrazená na obrázku 3.6.

Pro samotnou tvorbu odkazů přináší Spring Boot podporu ve formě projektu *spring-boot-starter-hateoas*. V aplikaci jsou použity dva způsoby tvorby odkazů.

1. Prvním způsobem je využití přímého odkazu na metodu, která obsluhuje

specifickou URL. Použití je zobrazené na obrázku 3.7.

2. Pokud potřebujeme nastavit odkaz, jehož obsluhující metoda se nachází v jiném modulu aplikace, nemusí být tato metoda v kódu viditelná. V tuto chvíli je potřeba použít relativní adresu, na kterou je metoda namapována. Ukázka zdrojového kódu je zobrazena na obrázku 3.8.

Výsledné odkazy jsou závislé na kontextu běhu aplikace. Například při lokálním vývoji může mít odkaz vzniklý vykonáním zdrojového kódu 3.7 podobu `http://localhost:8080/api/shop/car/1`.

```
@Getter
@Setter
@NoArgsConstructor
public class MetaDataSupport implements Serializable {

    @JsonProperty("_links")
    private HashMap<String, Link> links;

    public void linkTo(Link link) {
        Assert.assertNotNull("Link must not be null", link);
        Assert.assertNotNull("Link-rel must not be null",
            link.getRel());
        this.getLinks().put(link.getRel(), link);
    }

    private HashMap<String, Link> getLinks() {
        if (links == null) links = new HashMap<>();
        return links;
    }

}
```

Zdrojový kód 3.6: Podpora pro HATEOAS

```
  model.linkTo(linkTo(methodOn(CarController.class)
      .detailCar(model.getCar().getProductId())).withSelfRel());
```

Zdrojový kód 3.7: Tvorba odkazů na serveru odkazem na metodu

```
    protected String expandURL(String template, Object... params) {
        return ServletUriComponentsBuilder.fromCurrentContextPath()
        .path(api_prefix + template).buildAndExpand(params).toString();
    }

    model.linkTo(new Link(expandURL(
        ApiLinks.URL_CAR_SEARCH), "findAllCars"));
```

Zdrojový kód 3.8: Tvorba odkazů na serveru pomocí relativní URL

```
export const RouterLinks = {
  /* Basic */
  URL_HOMEPAGE: 'homepage',
  URL_LOGIN: 'login',

  /* Car */
  URL_CAR_DETAIL: router_module_prefix.shop_prefix +
      '/car/:productId',
  URL_CAR_NEW: router_module_prefix.shop_prefix + '/car/new',
  URL_CAR_SEARCH: router_module_prefix.shop_prefix + '/car/search',
}

export const routes: Routes = [
  {path: '', redirectTo: RouterLinks.URL_HOMEPAGE, pathMatch: 'full'},

  /* Basic */
  {path: RouterLinks.URL_HOMEPAGE, component: HomePage}
  {path: RouterLinks.URL_LOGIN, component: LoginComponent},

  /* Car */
  {path: RouterLinks.URL_CAR_NEW, component: CreateCarComponent},
  {path: RouterLinks.URL_CAR_SEARCH, component: SearchCarComponent},
  {path: RouterLinks.URL_CAR_DETAIL, component: DetailCarComponent}
}
```

Zdrojový kód 3.9: Deklarace Angular routeru

3.5.3 HATEOAS a Angular router

3.5.3.1 Omezení

Standardní Angular router není principu HATEOAS přizpůsoben. Router přepíná obrazovky dle předem definovaných pravidel, viz. výpis zdrojového kódu 3.9. I když získáme od serveru odkazy na další možné kroky v aplikaci, po přepnutí obrazovky, tedy přechodu na novou komponentu, tyto data ztrácíme, jelikož komponenty jsou od sebe izolované. Problém je, jak na nové komponentě získat potřebné odkazy.

3.5.3.2 Řešení

Mé řešení spočívá ve schopnosti Angular routeru zjistit, na jaké URL se právě nachází. Při přechodu na novou obrazovku se pro získání odkazů na další kroky v aplikaci provedou tyto kroky:

1. Získání aktuální URL pomocí Angular Routeru
2. Připojení předpony „/api“ k získané URL

```
protected createApiLink(): string {
  return environment.api_prefix + this.currentUrl;
}

constructor(private http: HttpClient, private errorHandler:
    ErrorHandler, private pathRouter: Router) {
  super();

  this.currentUrl = this.pathRouter.url;
}

searchCars() {
  this.busy = this.http.get<SearchData<CarModel,
      SearchCarLinks>>(this.createApiLink())
    .catch(err => this.errorHandler.handleError(err, this))
    .subscribe(response => {
      this.items = response.data.items;
      this.links = response._links;
    });
}
```

Zdrojový kód 3.10: Získání dat při přechodu na novou obrazovku

3. Odeslání HTTP požadavku na server
4. Získání odkazů z HTTP odpovědi

Pro každou obrazovku, která komunikuje se serverovou částí, je tedy nutné, aby server poskytoval odkazy na URL vzniklé připojením předpony „/api“.

3.5.4 HATEOAS a uživatelské rozhraní

HATEOAS princip nemusí sloužit pouze k navigaci v rámci aplikace. Dle dostupných odkazů můžeme řídit i zobrazování elementů na uživatelském rozhraní. Zobrazování elementů se může řídit i dle rolí, ale při každé změně, kdy se má daný element zobrazit, musíme nejen upravit vkládání odkazu, ale i zobrazení na uživatelském rozhraní. Pokud element svážeme s příslušným odkazem, jak je vidět na výpisu zdrojového kódu 3.11, element se zobrazí pouze v případě, že ze serverové části přijde odkaz „delete“.

3.6 Profily

3.6.1 Maven profily

Při vývoji aplikací se prostředí od sebe často liší. Prostředí, ve kterém aplikaci vyvíjíme nebo testujeme se ve většině případů liší od prostředí produkčního.

```
<button *ngIf="model._links.delete && lock"
    class="pull-right btn btn-default"
    title="delete" (click)="openDeleteModal()">
    <span class="glyphicon glyphicon-remove"></span>
</button>
```

Zdrojový kód 3.11: Získání dat při přechodu na novou obrazovku

```
<profile>
    <id>heroku</id>
    <activation>
        <property>
            <name>!skipFirstProfile</name>
        </property>
    </activation>
    <properties>
        <maven.profile>heroku</maven.profile>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>9.4.1208</version>
        </dependency>
    </dependencies>
</profile>
```

Zdrojový kód 3.12: Maven profil

Může se nám stát, že aplikace v každém prostředí potřebuje jiné závislosti nebo jiné údaje pro spojení s databází. Některé závislosti nemusejí být v produkčním prostředí dostupné a aplikace by se kvůli absenci závislostí nepodařilo ani sestavit. V tomto případě se nabízí využít profily. V rámci Maven konfigurace můžeme každému profilu nadefinovat specifické závislosti, měnit proces sestavování aplikace a jiné.[22] Na přiloženém zdrojovém kódu 3.12 je definovaný profil „heroku“. Všimněme si vlastnosti „maven.profile“, která bude důležitá v následujících kapitolách.

3.6.2 Spring profily

Maven profily však nevyřeší všechny problémy spojené s databázovou komunikací. Použitý ORM nástroj MyBatis [23] má pro oba typy databáze rozdílné mapování, a tak je nutné načíst tu konfiguraci, která odpovídá danému prostředí.

Obecně je potřeba rozlišit konfiguraci pro každé prostředí. K tomutu lze

využít Spring profily, které umožňují konfiguraci projektu dle prostředí a definujeme je v souboru application.properties. [24]

```
spring.profiles.active = heroku
```

Zdrojový kód 3.13: Spring profil

3.6.3 Propojení Maven a Spring profilů

Definování každého Maven a Spring profilu zvlášť je přinejmenším nešikovné. Oba profily můžeme jednoduše svázat. K tomu využijeme vlastnost Maven profilu „maven.profile“ zmíněnou výše. Svázání obou profilů je vidět na výpisu zdrojového kódu 3.14.

```
spring.profiles.active = @maven.profile@
```

Zdrojový kód 3.14: Propojení Spring a Maven profilu

3.7 Propagace výjimek

Každá aplikace se může během svého běhu dostat do chybového stavu, ať už očekávaného (selhání validace) nebo neočekávaného (typicky nedostupné internetové připojení, nedostatek místa na disku…). Aplikace na tyto výjimečné stavy musí reagovat a podat uživateli srozumitelnou zprávu, jak v používání aplikace pokračovat. Jelikož klientská a serverová část spolu komunikují přes HTTP protokol, využil jsem HTTP statusy pro odlišení různých typů výjimek. Aplikace pracuje s těmito typy:

- 401 – Unauthorized
- 403 – Forbidden
- 404 – Not found
- 422 – Unprocessable entity
- ostatní

3.7.1 Serverové zpracování

3.7.1.1 422 – Unprocessable entity

Tato chybová zpráva je odeslána v případě selhání validace příchozích dat. Jedním z příkladů může být zamezení nahrání obrázku špatného formátu. Aplikace vyhodí výjimku, která je následně zpracována a je odeslán chybový kód společně s chybovou zprávou. Zpracování takové výjimky je vidět ve výpisu kódu 3.15.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(BackendException.class)
    public ResponseEntity<ErrorMessages>
        handleBackendException(BackendException e) {
        logger.info(String.format("Process %s", e.getClass()));

        final ErrorMessages errorMessages =
            ErrorMessagesUtil.decorateMessages(null, e.getMessages());
        return new ResponseEntity<>(errorMessages,
            HttpStatus.UNPROCESSABLE_ENTITY);
    }
}
```

Zdrojový kód 3.15: Odchycení výjimky na serveru

3.7.2 Klientské zpracování

Frontendový klient aplikace reaguje na odpovědi, které obsahují chybový stav pomocí metody, která je zobrazena ve výpisu zdrojového kódu 3.16. Reakce se liší podle statusu HTTP odpovědi na 2 případy:

- Pokud je chybový stav roven 422, dané komponentě jsou nastaveny chybové zprávy odeslané ze serveru. Pro zobrazení je na klientské části zřízena speciální komponenta.
- Všechny ostatní chybové statusy způsobí přepnutí na novou obrazovku. Typicky pokud uživatel není přihlášen a přihlášení je vyžadováno, je přesměrován na obrazovku k přihlášení.

3.8 Integrace na platební modul

Jako platební modul jsem zvolil PayPal kvůli jeho veliké rozšířenosti a přímočaré integraci. Implementace integrace na PayPal je umístěna v samostatném modulu payment, který se nachází v modulu services.

3.8.1 Účty

Pro úspěšnou integraci na PayPal je potřeba založit účet na webové adrese `https://developer.paypal.com/developer/accounts/`. Po založení účtu je nutné zaregistrovat aplikaci. Posledním krokem je vytvoření účtů pro nakupujícího a prodejce.

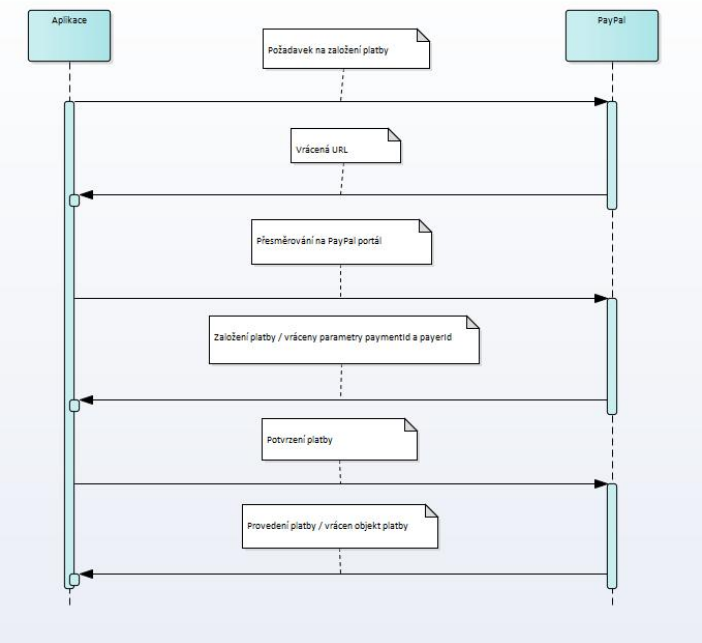
```
public handleError(error, errorWrapper) {
   switch (error.status) {
     case 401:
       this.pathRouter.navigateByUrl(expandForRouting(
              RouterLinks.URL_LOGIN));
       sessionStorage.removeItem('Authorization');
       return Observable.throw('401 Unauthorized');
     case 403:
       let token = sessionStorage.getItem('Authorization');
       if (token !== undefined && token !== null) {
         this.pathRouter.navigateByUrl(expandForRouting(
                RouterLinks.URL_ACCESS_DENIED));
         return Observable.throw('403 Forbidden');
       } else {
         this.pathRouter.navigateByUrl(
              expandForRouting(RouterLinks.URL_LOGIN));
         return Observable.throw('403 Forbidden');
       }
     case 404:
       this.pathRouter.navigateByUrl(
              expandForRouting(RouterLinks.URL_NOT_FOUND));
       return Observable.throw('404 Not found');
     case 422:
       try {
         errorWrapper.messages = JSON.parse(error.error);
       } catch (e) {
         errorWrapper.messages = error.error;
       }
       errorWrapper.growlMsgs =
           this.growlService.prepareMessage(GrowlSeverity.ERROR,
           'Something wrong',
       'Please read error messages');
       return Observable.throw('422 Unprocesable entity');
     default:
       this.pathRouter.navigateByUrl(
              expandForRouting(RouterLinks.URL_SERVER_ERROR));
       return Observable.throw('500 Server error');
   }
 }
```

Zdrojový kód 3.16: Odchycení výjimky na klientovi

3.8.2 Integrace

Pro úspěšnou integraci na PayPal je potřeba získat „client id“ a „client secret“. Jedná se o dvojici údajů, které identifikují účet v rámci PayPalu. Každá komunikace, probíhající přes REST API, tedy musí obsahovat tyto dva údaje. [25]

Na obrázku 3.4 můžeme vidět vzorovou komunikaci pro provedení platby.



Obrázek 3.4: Komunikace s portálem PayPal

Kapitola 4

Testování

Nedílnou součástí každého softwarového díla jsou testy. Pro minimalizaci chyb ve výsledné aplikaci jsem se využil nejen klasické unit testy, ale i testy komunikace s databází a testy integrační.

4.1 Testy komunikace s databází

Na rozdíl od unit testů, které ke svému běhu nepotřebují startovat samotnou aplikaci, databázové testy vyžadují spuštěnou aplikaci v takovém stavu, kdy je nastaveno spojení na databázi a veškeré beany [26] starající se o databázovou komunikaci jsou dostupné v aplikačním kontextu. Třída zobrazená ve výpisu kódu 4.1 slouží ke konfiguraci takového prostředí.

Avšak ne všechny výše uvedené anotace jsou pro vytvoření prostředí potřeba. Test databázové komunikace často vyžaduje data, která jsou v ní uložena. Vzhledem k tomu, že daná databáze může být využívána několika vývojáři, nemůžeme se spolehnout na existenci těchto dat. Pomocí anotace „@Sql“ můžeme do databáze vložit potřebná data. [27] Tyto data však nechceme v databázi po testu zanechat. Anotace „@Transactional“ zajistí odstranění dat po provedení testu. [27] Výsledný test si poté jednoduše vyžádá beanu [26] z apli-

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:config/spring-config.xml")
@Import({BaseAppConfig.class, ShopConfig.class})
@ActiveProfiles("test")
@Transactional
@Sql("classpath:test-insert-script.sql")
public class AbstractTest {
}
```

Zdrojový kód 4.1: Konfigurace pro testy databázové komunikace

kačního kontextu a zavolá operaci nad databází, viz. výpis zdrojového kódu 4.2.

```
public class UserDAOTest extends AbstractTest {

    @Autowired
    private UserDAO userDAO;
    @Autowired
    private RoleDAO roleDAO;

    @Test
    public void testInsert() {
        User user = new User();
        user.setName("Tester");
        user.setSurname("Tester");
        user.setEmail("tester@tester.com");
        user.setPassword("tester_password");
        user.setCity("Praha");
        user.setHouseNumber(5);
        user.setStreet("Ulice");
        user.setPhone("+420123456789");
        user.setPostCode(14000);
        userDAO.insert(user);
        User actual = userDAO.findById(user.getUserId());
        Assert.assertEquals(user.getEmail(), actual.getEmail());
    }
```

Zdrojový kód 4.2: Testy databázové komunikace

4.2 Servisní vrstva

4.2.1 Možná řešení

Pro unit testy servisních metod lze použít dva základní přístupy:

- Stub – Stub je třída implementující rozhraní tak, jak požaduje test (např. vrací jen vytvořený objekt na rozdíl od primární implementace, které provolává databázovou vrstvu). [28]
- Mock – Mock je „falešná třída“, které v testu definujeme chování. Výhodou mocku je možnost kontrolovat i zavolání, popř. jejich počet, konkrétní metody. [28]

4.2.2 Zvolené řešení

K servisním unit testům jsem se rozhodl využít druhý způsob – mockování. Na výpisu kódu 4.3 kódu můžeme vidět vzorový test.

```
@RunWith(EasyMockRunner.class)
public class UserServiceImplTest extends EasyMockSupport {

    private UserServiceImpl service = new UserServiceImpl();
    private UserDAO userDAOMock = createMock(UserDAO.class);
    private RoleDAO roleDAOMock = createMock(RoleDAO.class);

    @Before
    public void setUp() {
        service.userDAO = userDAOMock;
        service.roleDAO = roleDAOMock;
    }

@Test
public void testFindAllUsers() {
    final User user = new User();
    expect(userDAOMock.findAll())
    .andReturn(Collections.singletonList(user));
    replayAll();
    List<User> actual = service.findAllUsers();
    verifyAll();
    Assert.assertFalse(actual.isEmpty());
    Assert.assertEquals(user, actual.get(0));
}
```

Zdrojový kód 4.3: Servisní test

4.3 Integrační testy

Na rozdíl od jednotkových testů, které provádějí testy pouze v rámci jedné komponenty, integrační testy umožňují otestovat integraci více takových komponent. [29]

Integračními testy je pokryt pouze modul shop, jelikož moduly payment a transport využívají převážně externích knihoven.

4.3.1 Testovací scénáře pro modul shop

4.3.1.1 Založení zákazníka

- **Účel:** Kontrola založení zákazníka.
- **Popis:** Provedení založení zákazníka.
- **Očekávaný výsledek:** Po založení vznikne v databázi nový záznam v tabulce *uživatel* s atributy dle vyplněných hodnot a v tabulce *role* vznikne záznam s hodnotou „zákazník“ navázaný na daného uživatele.
- **Výsledek testu:** OK

4.3.1.2 Založení produktu a navázání dalších dat

- **Účel:** Kontrola založení produktu a navázání dalších dat.
- **Popis:** Provedení akcí založení produktu, nahrání obrázku/přílohy.
- **Očekávaný výsledek:** Po založení bude v databázi založen nový záznam se všemi atributy dle vyplněných hodnot. Při nahrání obrázku nebo přílohy vznikne, po úspěšné validaci na unikátní jméno souboru v rámci souborů navázaných na daný produkt, nový záznam v tabulce *soubor*.
- **Výsledek testu:** OK

4.3.1.3 Objednání zboží / založení objednávky

- **Účel:** Kontrola vytvoření objednávky s produkty z nákupního košíku a s navolenými dodatečnými údaji.
- **Popis:** Přidání zboží do košíku a založení objednávky.
- **Očekávaný výsledek:** Nalezení dostupných produktů a následné přidání některých z nich do nákupního košíku. Po odeslání nákupního košíku bude vrácena šablona objednávky s vybranými produkty. Po vyplnění dodatečných údajů a odeslání objednávky bude v databázi založena objednávka dle vyplněných údajů. Dále bude vytvořena faktura dle definice obsažené v dokumentaci, vznikne záznam v tabulce *doručovací adresa* a záznamy v tabulce reprezentující vztah produkt - objednávka. Počet dostupných kusů produktů, které byly objednány, se sníží o množství objednaných kusů.
- **Výsledek testu:** OK

4.3.1.4 Správa objednávek

- **Účel:** Kontrola správy objednávek.
- **Popis:** Provedení akcí vyhledání, editace a smazání objednávky.
- **Očekávaný výsledek:** Korektní nalezení dat po vyhledání objednávky dle identifikátoru. Po editaci budou záznamy odpovídající dané objednávce aktualizovány v databázi dle změněných hodnot (včetně přegenerování faktury). Smazání objednávky způsobí odstranění záznamu z tabulky *objednávka*. Dále budou odstraněny záznamy z tabulky *doručovací adresa* a z tabulky reprezentující vazbu produkt - objednávka navázané na danou objednávku.
- **Výsledek testu:** OK

Kapitola 5

Uživatelská příručka

5.1 Sestavení a spuštění dle prostředí

5.1.1 Lokální

Pro lokální sestavení a spuštění aplikace jsou potřeba splnit tyto kroky:

1. Nainstalovat JDK 8.
2. Nainstalovat Maven 3.5.0 nebo vyšší.
3. Vložit do lokálního Maven repozitáře driver pro Oracle databázi s následujícími koordináty:

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc7</artifactId>
    <version>10.1.0</version>
</dependency>
```

4. Spustit script „deploy-script-local.sh“, který je dostupný v kořenovém adresáři projektu. Tento skript automaticky nainstaluje vše potřebné k sestavení Angular klienta (Node.js, Npm), sestaví a spustí aplikaci. Po spuštění bude aplikace dostupná ve webovém prohlížeči na adrese `http://localhost:8080/`.

5.1.2 Heroku server

Pro nasazení na Heroku je potřeba splnit tyto kroky:

1. Založit účet na `https://dashboard.heroku.com`.
2. Ve vytvořeném účtu založit projekt.

3. Nainstalovat Heroku CLI dle návodu dostupného na adrese `https://devcenter.heroku.com/articles/heroku-cli`
4. Nastavit git remote se jménem heroku na hodnotu adresy git repozitáře založeného projektu.
5. Spustit skript „deploy-script-heroku.sh“, který je dostupný v kořenovém adresáři projektu (skript vkládá zdrojové kódy do git repozitáře serveru Heroku z větve, ve které se uživatel právě nachází).

Závěr

Cílem práce bylo vytvořit webovou aplikaci ve formě e-obchodu za použití všech fází klasického softwarového procesu, což bylo naplněno. Všechny funkční požadavky byly v aplikaci splněny. Stejně tak bylo splněno zadání z hlediska použitých technologií. V praktické části jsem se snažil použít moderní přístupy pro tvorbu webových aplikací ve světě Javy.

V práci byl kladen důraz na snadnou rozšiřitelnost aplikace, což je splněno vzhledem k rozdělení kódu do jednotlivých modulů. Přidání nového modulu do aplikace sebou může nést minimální zásahy do modulů base-app a application, zbytek aplikace zůstává nezměněn.

V aplikaci existuje mnoho možností pro rozvoj samotné bussiness logiky. Jedním z příkladů může být přidání dalších platebních portálů, jelikož v současném řešení je aplikace integrována pouze na portál PayPal. Dalším příkladem vylepšení může být zasílání emailů zákazníkům o průběhu objednávky.

Vylepšení nemusí být předmětem pouze logiky programu. V této době je stále více populární pracovat s platformou Docker pro vývoj aplikací v kontejnerech. [30] V neposlední řadě by pro usnadnění vývoje a správu projektu bylo vhodné zavést continuous integration [31] se všemi náležitostmi.

Výsledná práce, včetně analýzy a návrhu a zdrojových kódů, je nahrána na přiloženém CD. Samotná aplikace je dostupná na serveru Heroku na webové adrese `https://gregetom.herokuapp.com`.

Literatura

1. Graph definition. *Math Insight* [online] [cit. 2018-04-07]. Dostupné z: https://mathinsight.org/definition/graph.
2. Graph theory. *Analytic Tech* [online] [cit. 2018-04-07]. Dostupné z: http://www.analytictech.com/networks/graphtheory.htm.
3. Cesta a souvislost v grafu. *Teorie grafů* [online] [cit. 2018-04-11]. Dostupné z: http://teorie-grafu.cz/zakladni-pojmy/cesta-a-souvislost-grafu.php.
4. BONDY, J. A.; MURTY, U. S. R. Graph theory with applications. *Macmillan Publishers*. 1976, s. 17–20. ISBN 0-444-19451-7.
5. Bellman-Ford Algorithm. *Geeks For Geeks* [online] [cit. 2018-04-07]. Dostupné z: https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/.
6. Floyd-Warshallův algoritmus. *Algoritmy* [online] [cit. 2018-04-08]. Dostupné z: https://www.algoritmy.net/article/5207/Floyd-Warshalluv-algoritmus.
7. KOLÁŘ, Josef. Teoretická informatika. *Praha: Česká informatická společnost*. 2. vyd. 2000, s. 129. ISBN 80–900853–8–5.
8. 8 Best Programming Languages to Develop an Ecommerce Website in 2017. *Web ecommerce* [online] [cit. 2018-04-19]. Dostupné z: http://www.webecommercepros.com/best-programming-language-ecommerce-website-development-2017.
9. What is Heroku. *Heroku* [online] [cit. 2018-02-25]. Dostupné z: https://www.heroku.com/what.
10. Use Case Diagram. *IT network* [online] [cit. 2018-02-20]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-use-case-diagram.
11. Spring framework. *Spring* [online] [cit. 2018-03-26]. Dostupné z: https://spring.io/.

12. Spring Boot framework. *Projec Spring Boot* [online] [cit. 2018-03-26]. Dostupné z: https://projects.spring.io/spring-boot/.

13. Angular. *Angular docs* [online] [cit. 2018-03-26]. Dostupné z: https://angular.io/docs.

14. Maven. *Maven docs* [online] [cit. 2018-04-02]. Dostupné z: https://maven.apache.org/archetype/index.html.

15. Why does Angular need Node.js. *Quora* [online] [cit. 2018-04-02]. Dostupné z: https://www.quora.com/Why-does-Angular-2-need-Node-js.

16. Node.js. *Node.js docs* [online] [cit. 2018-04-02]. Dostupné z: https://www.infoworld.com/article/3210589/node-js/what-is-nodejs-javascript-runtime-explained.html.

17. Npm. *Npm docs* [online] [cit. 2018-04-02]. Dostupné z: https://docs.npmjs.com/getting-started/.

18. Git. *Git docs* [online] [cit. 2018-04-02]. Dostupné z: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics.

19. Json Web Token. *Auth0* [online] [cit. 2018-04-01]. Dostupné z: https://auth0.com/learn/json-web-tokens/.

20. AOP with Spring Framework. *Tutorials point* [online] [cit. 2018-04-08]. Dostupné z: https://www.tutorialspoint.com/spring/aop_with_spring.htm.

21. Hateoas. *Spring.io* [online] [cit. 2018-03-20]. Dostupné z: https://spring.io/understanding/HATEOAS.

22. Maven profiles. *Maven docs* [online] [cit. 2018-03-26]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-profiles.html.

23. MyBatis. *MyBatis docs* [online] [cit. 2018-04-08]. Dostupné z: http://www.mybatis.org/mybatis-3/.

24. Spring profiles. *Spring docs* [online] [cit. 2018-03-26]. Dostupné z: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html.

25. PayPal. *Developer PayPal* [online] [cit. 2018-04-07]. Dostupné z: https://developer.paypal.com/docs/integration/admin/manage-apps/.

26. Spring Bean Definition. *Tutorials point* [online] [cit. 2018-04-08]. Dostupné z: https://www.tutorialspoint.com/spring/spring_bean_definition.htm.

27. Spring testing. *Spring docs* [online] [cit. 2018-03-18]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html.

28. Unit Testing with Stubs and Mocks. *Spring.io* [online] [cit. 2018-03-18]. Dostupné z: `https://spring.io/blog/2007/01/15/unit-testing-with-stubs-and-mocks/`.

29. Integrační testování. *Testovani softwaru* [online] [cit. 2018-03-18]. Dostupné z: `http://testovanisoftwaru.cz/tag/integracni-testovani/`.

30. What is Docker? *Docker* [online] [cit. 2018-04-15]. Dostupné z: `https://www.docker.com/what-docker`.

31. Co je kontinuální integrace? *Microsoft Visual Studio* [online] [cit. 2018-04-15]. Dostupné z: `https://www.visualstudio.com/cs/learn/what-is-continuous-integration/`.

Dodatek A

Seznam použitých zkratek

JAR Java Archive

UI User interface

API Application programming interface

REST Representational State Transfer

HATEOAS Hypermedia as the Engine of Application State

XML Extensible Markup Language

CD Compact Disc

HTTP Hypertext Transfer Protocol

JWT Json Web Token

AOP Aspect Oriented Programming

JSON JavaScript Object Notation

I/O Input / Output

JDK Java Development Kit

Dodatek B

Obsah přiloženého CD

- `readme.pdf` ..... stručný popis obsahu CD
- `application` ..... adresář s aplikací
  - `src` ..... zdrojové kódy implementace
  - `application.jar` ..... spustitelná forma aplikace
- `doc` ..... adresář s dokumentací
- `text` ..... text práce
  - `src` ..... zdrojové kódy práce ve formátu LaTeX
  - `bakalářská práce.pdf` ..... text práce ve formátu PDF