UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

NÁVRH A IMPLEMENTACE WEBOVÉ APLIKACE PRO TVORBU ROZPOČTŮ

Jovkhar Issayev

Diplomová práce

2023

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jovkhar Issayev**
Osobní číslo:	**I20206**
Studijní program:	**N0613A140007 Informační technologie**
Téma práce:	**Návrh a implementace webové aplikace pro tvorbu rozpočtů**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

V teoretické části práce budou popsány současné trendy a nástroje v oblasti plánování rozpočtů a budou popsány výhody a nevýhody využívání webových aplikací. Dále pak budou popsány veškeré technologie, které budou pro realizace páce využity.
V rámci praktické části diplomové práce bude prvně třeba provést sběr a analýzu funkčních a nefunkčních požadavků. Po této části bude pozornost věnována návrhu vhodného databázového modelu a celkové koncepci webové aplikace včetně UI. V další kroku bude následovat vlastní implementaci webové aplikace pro tvorbu rozpočtu, a to s využitím frameworku JAVA Spring a technologie Vaadin. Systém musí být řešen i z pohledu různých uživatelských rolí a více uživatelů.
Výsledná webová apliakace bude nasazena na veřejný server v kontejneru Docker s přístupem přes doménu a na cloudový hosting heroku.

Rozsah pracovní zprávy:	**50-60**
Rozsah grafických prací:	
Forma zpracování diplomové práce:	**tištěná/elektronická**

Seznam doporučené literatury:

CARNELL, John. *Spring microservices in action: a multiplatform approach to building chatbots*. Shelter, Island, NY: Manning Publications Co., 2017. ISBN 16-172-9398-9.
CARNELL, John. *Beginning spring boot 2: applications and microservices with the spring framework*. New York, NY: Springer Science Business Media, 2017. ISBN 978-148-4229-309.

Vedoucí diplomové práce:	**Ing. Jan Fikejz, Ph.D.** Katedra softwarových technologií
Datum zadání diplomové práce:	**8. listopadu 2021**
Termín odevzdání diplomové práce:	**20. května 2022**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2021

Prohlašuji

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 18. 5. 2023 ……………………………………

Jovkhar Issayev

Poděkování

Děkuji vedoucímu diplomové práce Ing. Janu Fikejzovi, Ph.D. za odbornou pomoc a další cenné rady při zpracování diplomové práce. Rád bych poděkoval také své rodině a přátelům, kteří mě při vytváření této práce podpořili, a bez jejichž pomoci by nebylo možné práci dokončit.

Anotace

Diplomová práce se zaměřuje na návrh a vývoj webové aplikace pro tvorbu rozpočtů. V úvodu práce jsou popsány současné aplikace pro tvorbu rozpočtů. Dále byla provedena analýza požadavků a návrh implementace vlastní webové aplikace. V praktické části byla provedena implementace vlastního řešení pomocí programovacího jazyka Java, frameworku Spring a Vaadin. Docker balíček aplikace byl nasazen na cloudovou platformu Heroku a Doker hub.

Klíčová slova

Java, Spring Famework, Vaadin, PaaS, Docker, HTML, CSS, Javascript

Title

Develop a web application designated for budgeting using Spring/Java and Vaadin

Annotation

The diploma thesis focuses on the design and development of a web application for budgeting. The introduction describes the current applications for budgeting. An analysis of the requirements and a proposal for the implementation of the web application were also carried out. In the practical part, the application was implemented using the Java programming language, Spring and Vaadin framework. The Docker image has been deployed to Heroku cloud platform and Docker hub.

Keywords

Java, Spring Famework, Vaadin, PaaS, Docker, HTML, CSS, Javascript

Obsah

Seznam obrázků **10**

Seznam tabulek **11**

Seznam výpisů kódu **12**

Seznam zkratek **13**

Úvod **14**

1 Pohled na webové stránky a aplikace **15**

1.1 Rozdíl mezi webem a webovou aplikací 15

1.2 Historie webových stránek a aplikací 15

1.2.1 Stránky na straně serveru 15

1.2.2 AJAX 16

1.2.3 Single Page Application 16

1.2.4 Progressive Web Application 16

2 Analýza existujících aplikací pro tvorbu rozpočtu **17**

2.1 Free Procurement Project 17

2.2 Insio 17

2.3 Kissflow 17

2.4 Nabídky Plus 18

3 Použité technologie **19**

3.1 Java 19

3.2 Spring 19

3.2.1 Core Container 20

3.2.2 IoC 21

3.2.3 DI 22

3.2.4 Data Access/Integration 22

3.2.5 WEB 22

3.3 Spring Boot 22

3.4 Vaadin 22

3.4.1 Technologický základ Vaadin 23

3.4.2 Architektura Vaadin 24

3.4.3 Web components 25

3.4.4 Vaadin Components 26

3.4.5 Princip fungování Vaadin komponent 26

3.4.6 Navigace 27

3.4.7 Způsoby dopravy 27

3.4.8 Integrace Spring Boot a Vaadin 28

3.5 JUnit a @SpringBootTest 29

3.6 Docker ... 29
3.6.1 Základní pojmy ... 29
3.6.2 Práce s Docker ... 30
3.6.3 Vytvoření obrazu ... 30
3.6.4 Multi-stage docker build ... 31
3.7 Hibernate ... 31
3.8 Maven ... 32
3.9 Heroku ... 33
4 Návrh ... 34
4.1 Specifikace požadavků ... 34
4.2 Přehled uživatelů ... 35
4.3 Případy užití ... 36
4.4 Návrh designu ... 45
4.5 Doménový model ... 47
4.6 Výběr architektury ... 48
4.6.1 Serverless architektura ... 48
4.6.2 Událostmi řízená architektura ... 49
4.6.3 Mikroservisní architektura ... 50
4.6.4 Monolitická architektura ... 50
4.7 Databázový model ... 51
5 Implementace ... 54
5.1 Struktura projektu ... 54
5.1.1 Frontend ... 55
5.1.2 Client ... 55
5.1.3 Config ... 56
5.1.4 Converter ... 57
5.1.5 Error ... 57
5.1.6 Mapper ... 58
5.1.7 Model ... 59
5.1.8 Persistence ... 59
5.1.9 Security ... 62
5.1.10 Service ... 63
5.1.11 Utils ... 65
5.1.12 UI ... 65
5.1.13 Resources ... 74
5.2 Kontejnerizace a nasazení ... 75
5.2.1 Dockerfile ... 75
5.2.2 Docker-compose ... 77

5.2.3 Nasazení aplikace 78
6 Testování 79
6.1 Persistence 79
6.2 Service 80
6.3 Views 82
Závěr 85
Použitá literatura 86

Seznam obrázků

Obrázek 1: Moduly Spring Framework ... 19
Obrázek 2: Proces tvorby Spring Bean ... 20
Obrázek 3: Architektura Vaadin ... 25
Obrázek 4: Architektura komponenty Vaadin ... 27
Obrázek 5: Wireframe – Obrazovka položek uživatele ... 46
Obrázek 6: Wireframe – Obrazovka nastavení ... 46
Obrázek 7: Dialog – Nová položka ... 47
Obrázek 8: Doménový model ... 48
Obrázek 9: Datový model ... 53
Obrázek 10: Pohled – Položky uživatele ... 69
Obrázek 11: Dialog pro editaci položky ... 69
Obrázek 12: Stránka pro přihlášení uživatele do aplikace ... 71
Obrázek 13: Pohled SettingsView ... 72
Obrázek 14: Stránka pro zobrazení nabídek ... 73
Obrázek 15: Šablona nabídky ... 73
Obrázek 16: Stránka detailů nabídky ... 74
Obrázek 17: Dialogové okno OfferPatternForm pro editaci položek nabídky ... 74

Seznam tabulek

Tabulka 1: Funkční a nefunkční požadavky 35

Seznam výpisů kódu

Výpis kódu 1: Závislosti pro Vaadin Spring Boot 29
Výpis kódu 2: Procedura copy_patterns 53
Výpis kódu 3: CSS styly pro textové pole ceny nabídky 55
Výpis kódu 4: Zabezpečení aplikace pomocí Spring Security 56
Výpis kódu 5: Převodník doby montáže 57
Výpis kódu 6: Implementace vlastní obslužné rutiny chyb 58
Výpis kódu 7: Implementace vlastního posluchače nových seancí 58
Výpis kódu 8: Implementace třídy UserMapper 59
Výpis kódu 9: Implementace JPA repositáře 60
Výpis kódu 10: Implementace třídy AbstractEntity 61
Výpis kódu 11: Implementace třídy OfferEntity 61
Výpis kódu 12: Implementace Specification pro položky 62
Výpis kódu 13: Rozhraní CrudService 63
Výpis kódu 14: Rozhraní FilterableCrudService 64
Výpis kódu 15: Implementace třídy PatternService 65
Výpis kódu 16: Implementace abstraktní třídy pro vytvoření forem 68
Výpis kódu 17: Binder pro položky 70
Výpis kódu 18: Implementace událostí komponenty PatternForm 70
Výpis kódu 19: Nastavení posluchačů událostí komponenty PatternForm 70
Výpis kódu 20: Dockerfile 75
Výpis kódu 21: Docker-compose 77
Výpis kódu 22: Integrační test JPA repositáře FirmRepository 80
Výpis kódu 23: Integrační test služby FirmService 81
Výpis kódu 24: Jednotkový test komponenty PatternForm 83
Výpis kódu 25: Integrační test komponenty PatternsView 84

Seznam zkratek

AJAX	Asynchronous JavaScript and XML
AOP	Aspect Oriented Programming
API	Application Programming Interface
CLI	Command-line interface
CRUD	Create, Read, Update, Delete
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DI	Dependency Injection
DMS	Document Management Systems
DOM	Document Object Model
DPH	Daň z přidané hodnoty
FaaS	Function-as-a-Service
HTML	HyperText Markup Language
IČO	Identifikační číslo osoby
IoC	Inversion of Control
JDK	Java Development Kit
JDO	Java Data Objects
JPA	Java Persistence API
JRE	Java Runtime Environment
JS	JavaScript
JSON	JavaScript Object Notation
JSP	Java Server Pages
ORM	Object Relational Mapping
OTP	One-time password
PaaS	Platform as a service
PDF	Portable Document Format
POJO	Plain Old Java Object
POM	Project Object Model
PWA	Progressive Web Application
SPA	Single Page Application
SQL	Structured Query Language
UI	User Interface
WWW	World Wide Web
XML	Extensible Markup Language
XSD	XML Schema Definition

Úvod

Webové aplikace hrají důležitou roli v dnešním světě a zjednodušují práci lidem v různých oblastech. Jednou z takových oblastí je tvorba a kalkulace rozpočtu při provedení elektrikářských prací. Pro poslední zmíněné využití tato aplikace slouží.

Cílem této diplomové práce je navrhnout a provést implementaci webové aplikace pro tvorbu rozpočtů elektrikářských prací. Aplikace bude umožňovat uživateli vytvářet nabídky, včetně výpočtu nákladů, spravovat zákazníky, údaje o firmě, uživatele, jednotlivé položky, provádět import souboru s položkami a export hotových nabídek. Uživatelské rozhraní by mělo umožnovat komfortní práci a jednoduše, intuitivně se ovládat. Pomocí této aplikace firma bude mít možnost doplnit vlastní údaje z registru MF ČR za použití IČO přes služby ARES. Aplikace také bude podporovat více uživatelský režim, který umožní uživatelům upravovat a prohlížet nabídky pro zákazníky. Další výhodou aplikace budou výchozí elektrikářské aktivity, které jsou dostupné a přizpůsobitelné pro každou firmu.

Teoretická část práce je zaměřena na popis použitých technologií, historii vývoje a vlastností webových aplikací. Také v této části práce je provedena analýza již existujících řešení, která umožní lépe pochopit uživatelské potřeby a existující problémy těchto řešení s možností se vyhnout problémům při vlastní implementaci. V dalších kapitolách je provedena analýza a návrh vlastní aplikace.

Poslední kapitoly práce jsou zaměřeny na implementaci webové aplikace pomocí moderních frameworků Spring/Java a Vaadin. Aplikace bude obsahovat komponenty, šablony zobrazení a nástroje, které mohou firmám pomoci ušetřit čas při vytvoření nových nabídek. Při vývoji aplikace budou využity jen bezplatné a otevřené komponenty Vaadin.

Testování webové aplikace bude provedeno pomocí jednotkových a integračních testů. Také pro webovou aplikaci bude vytvořen Dockerfile, docker-compose a hotový balíček bude nasazen na cloudovou službu Heroku a Docker hub.

1 Pohled na webové stránky a aplikace

V dnešním informačním světě, kde se internet stal neoddělitelnou součástí života většiny globální populace, se webové stránky a aplikace používají stále více v různých oborech, jako je medicína, gastronomie, zemědělství atd. Navzdory tomu často uživatel netuší, pokud používá webovou aplikaci nebo stránku, jaký je mezi nimi rozdíl. Proto je úvod této kapitoly zaměřen na vysvětlení rozdílů těchto pojmů a také je popsána evoluce od webových stránek do webových aplikací.

1.1 Rozdíl mezi webem a webovou aplikací

K prohlížení webových stránek a aplikací slouží prohlížeč. Weby mohou prezentovat digitální obsah, obrázky, video a zvuk. Webové stránky jsou statické, což znamená, že obsah se dynamicky neaktualizuje. Většina stránek je vytvořena pomocí HTML, CSS a JavaScript. Webové stránky neumožňují uživatelům žádnou manipulaci s daty, na rozdíl od webových aplikací. [15] [16]

Webové aplikace na rozdíl od webových stránek obsahují interaktivní prvky. Gmail, Facebook, YouTube, Twitter atd. jsou webové aplikace, které jsou dynamické a interaktivní.

Webová aplikace je často připojena k databázi pro přístup k datům, která lze také ukládat a přistupovat k nim prostřednictvím přizpůsobeného rozhraní. Webové aplikace jsou obtížnější na vývoj a vyžadují zkušený tým softwarových vývojářů. [15] [16]

1.2 Historie webových stránek a aplikací

Vznik internetu byl nepochybně jedním z revolučních kroků v historii lidstva. První webové stránky byly pouze informativní a obsahovaly jen statický kontent, kde stránky byly propojeny pouze hypertextovými odkazy. Později vznikla možnost přidávat obrázky, videa a audio soubory na webové stránky, ale stále byly statické. V roce 1995 začal vývoj programovacího jazyka Javascript, který přidal webovým stránkám dynamiku. Javascript umožnil zobrazovat webové stránky s různými interaktivními prvky, včetně vektorové animace. [15] [16]

1.2.1 Stránky na straně serveru

Se vznikem programovacích technologií na straně serveru, jako je PHP, JSP, se načítání a editace dat začaly provádět přes databáze, kde jsou HTML stránky připraveny na základě některých šablon a tyto šablony jsou odeslány jako odpovědi. Problémem však je, že prakticky každá akce uživatele musí být zpracována serverem, který vrátí aktuální HTML

stránku, proto se stránka musí načíst znovu. Jako řešení problému aktualizace stránek vznikl AJAX, který umožňuje aktualizovat obsah asynchronně. [16]

1.2.2 AJAX

V roce 2005 vznikla technologie AJAX, která umožnila vytvářet asynchronní webové aplikace. S AJAXem mohou webové aplikace odesílat a získávat data ze serveru asynchronně (na pozadí), aniž by zasahovaly do zobrazení a chování stávající stránky. Oddělením vrstvy pro výměnu dat od prezentační vrstvy umožňuje AJAX webovým stránkám a webovým aplikacím dynamicky měnit obsah bez nutnosti znovu načítat celou stránku. Data se odesílají nebo přijímají ve formátu XML či JSON. [11] [16]

1.2.3 Single Page Application

V případě, že uživatel bude chtít načíst jinou stránku, tak bude muset čekat na odpověď od serveru. Proto zde přichází další koncept oddělení pohledu a dat, nebo jinými slovy oddělení front-endu a back-endu. [11]

Všechny šablony a logika front-endu jsou načteny a uloženy do cache paměti na front-endu při počátečním načtení. Front-end pomocí konceptu AJAX vykresluje a dostává požadovaná data prostřednictvím volání API. Navigace na libovolné stránky probíhá bez zpoždění, čeká se pouze na data, nikoli na zobrazení. Vývoj podobných front-endů umožňují frameworky a knihovny, jako je Vaadin, React, Angular, Vue nebo Ember. [11] [16]

1.2.4 Progressive Web Application

Asynchronnost akcí uživatele zajišťuje AJAX, doba načítání pohledů je řešena pomocí SPA. Zbývající část je datová část, kde musíme vždy provést načtení, abychom získali nejnovější data. Zde přichází progresivní webová aplikace (PWA), díky které uživatel cítí webovou aplikaci jako skutečnou. PWA je nový způsob poskytování úžasných uživatelských zážitků na webu, který aplikaci načte okamžitě, bez ohledu na stav sítě, a to i v režimu offline. Toho je dosaženo pomocí service worker proxy, který je umístěn mezi kódem aplikace a sítí. Service worker je specializovaný JavaScript skript, který funguje jako proxy mezi webovými prohlížeči a webovým serverem. Když aplikace poprvé dostane požadavek, service worker rozhodne, zda data uloží do cache paměti. Při dalším požadavku service worker rozhodne, zda obsluhovat odpověď uloženou v cache paměti, nebo poslat nový požadavek a poskytnout odpověď. [11] [15]

2 Analýza existujících aplikací pro tvorbu rozpočtu

Tato kapitola je zaměřena na analýzu a revizi stávajících softwarových produktů, které umožňují implementaci hlavních funkcí při tvorbě rozpočtu. Jsou popsány pozitivní a negativní aspekty analyzovaných programů.

2.1 Free Procurement Project

Free-Procurement Project je bezplatné řešení pro Windows systémy od kanadské společnosti SpendMap, které pomáhá malým a středním podnikům navrhovat nákupní objednávky pomocí vlastních formulářů, odesílat je a sledovat stav přijatých nebo zpožděných objednávek. Free-Procurement Project také umožňuje manažerům sledovat rozpočty a vytvářet výkazy výdajů, generovat nové objednávky pomocí šablon a sledovat historii položek. Modul řízení zásob automaticky sleduje a řídí stav zásob na více místech. Řešení je k dispozici zdarma a podpora je nabízena prostřednictvím e-mailu a telefonu. Pro větší podniky SpendMap také nabízí placenou webovou verzi. [17]

2.2 Insio

Mezi produkty, které poskytuje společnost Insio, patří Document Management System (DMS), který umožňuje uživateli vytvářet nákupní nabídky, zpracovávat faktury, smlouvy a cestovní příkazy. INSO DMS zjednodušuje uživatelům vytvoření nabídky a uživatel může při jejím vytvoření samostatně nastavit schvalovací procesy, oprávnění a přístupy, stavy a možnosti editace polí v jednotlivých krocích jsou plně konfigurovatelné. Systém také umožňuje ražení, filtrování objednávek. Další výhodou je možnost stáhnout nebo odeslat hotovou objednávku ve formátu PDF e-mailem například dodavateli. INSO DMS má také otevřené API, které umožňuje propojení se všemi ERP, skladovými a mzdovými systémy. Integrace s Google Translate dává možnost zákazníkům z celého světa používat tento systém. Insio poskytuje své produkty pro PC i pro mobilní zařízení. Zájemce o systém má možnost získat demo verzi na 30 dní zdarma. [18]

2.3 Kissflow

Kissflow je cloudové řešení nabízené společností OrangeScape Technologies, které automatizuje obchodní procesy a sleduje výkon. Je vhodný pro podnikání všech velikostí a odvětví. Produkt nabízí správu nákupních objednávek a schvalování platebních procesů. Kissflow pomáhá iniciovat požadavky na zpracování, zobrazit položky, které vyžadují akce uživatele, a schvalovat čekající úlohy. Podporuje řízení procesů lidských zdrojů (HR), včetně náboru zaměstnanců, správy pracovních výkazů, schvalování žádostí o dovolenou

a schvalování faktur. Kissflow se integruje s Google Apps a dalšími cloudovými aplikacemi, jako WebHooks a Zapier End Point, pomocí integračních rozhraní API. Kissflow je také dostupný na zařízeních Android a iOS pomocí nativních mobilních aplikací. Je k dispozici k zakoupení ročně nebo měsíčně. [19]

2.4 Nabídky Plus

Nabídky plus je jednoduchá aplikace pro tvorbu cenových nabídek s intuitivním ovládáním a přehledným zobrazením vypočteného zisku. Aplikace podporuje vytváření nabídek, objednávek a faktur. Také je zde podpora cizích měn, připojení dokumentů, komentářů a barevných informačních vlajek. Program obsahuje následující moduly:

- Nabídky – hlavní modul, umožňuje vytvářet nabídky a provádět jejich archivaci.
- Ceník – eviduje seznam výrobků, materiálu či služeb, které můžete vložit do nabídky.
- Adresář – obsahuje seznam všech adres a kontaktů. [20]

3 Použité technologie

V této kapitole jsou popsány technologie, které byly použity pro vývoj a nasazení webové aplikace.

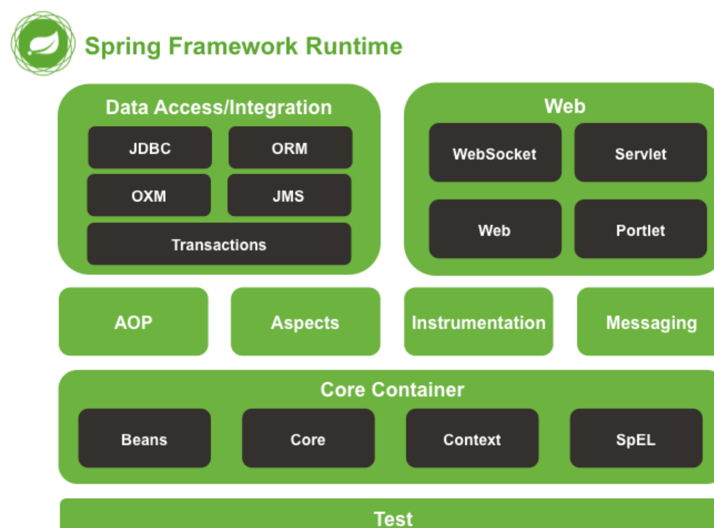
3.1 Java

Java je multiplatformní, objektově orientovaný programovací jazyk. Také patří mezi nejpoužívanější programovací jazyky. V této práci bylo použito prostředí Java Development Kit (JDK) 11 od společnosti Oracle. JDK slouží k vývoji softwaru v Javě a obsahuje nástroje potřebné k psaní programů, kompilaci a Java Runtime Environment (JRE) k jejich spouštění.

3.2 Spring

Spring framework slouží k vývoji aplikací v programovacím jazyce Java nebo Kotlin. Spring umožňuje vytvářet aplikace z „plain old Java objects“ (POJO) a nabízí mnoho rozšíření, která se používají pro vývoj nejrůznějších rozsáhlých aplikací na platformě Java EE. Také má obrovskou komunitu Java/Kotlin vývojářů, kteří přispívají k zavádění dalších rozšíření a vylepšování funkcí, jež Spring už má. [1] [2] [8]

Spring Framework se přibližně skládá ze 20 modulů, viz obrázek 1. Tyto moduly jsou seskupeny do Core Container, Data Access/Integration, Web, Aspect Oriented Programming (AOP), Instrumentation a Test. Základním modulem pro všechny aplikace ve Spring Framework je Core Container. Ostatní moduly každý vývojář přidává samostatně při vývoji aplikace. [4] [8]



Obrázek 1: Moduly Spring Framework

Zdroj: [8]

Dále jsou popsány moduly, které byly použity při vývoji webové aplikace v této práci.

3.2.1 Core Container

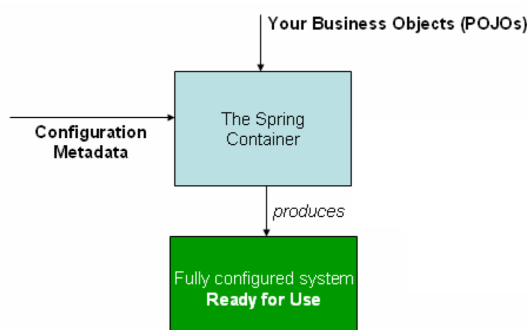
Core Container zahrnuje Beans, Core, Context a SpEL (jazyk výrazů). BeanFactory je zodpovědný za Beans, což je sofistikovaná implementace Factory Pattern. Modul Core poskytuje klíčové části frameworku, včetně vlastností „Inversion of Control“ (IoC) a „Dependency injection“ (DI). Context je postaven na Beans a Core a umožňuje přístup k libovolnému objektu, který je definován v nastavení. Klíčovým prvkem modulu Context je rozhraní ApplicationContext. Modul SpEL poskytuje výkonný výrazový jazyk pro manipulaci s objekty za běhu. [4] [8]

Spring Container

Klíčovou komponentou Spring Framework je Spring Container, který je zodpovědný za vytvoření, konfiguraci, správu a odstranění objektů. Spring Container používá Dependency Injection (DI) ke správě komponent, které tvoří aplikaci. Tyto komponenty se nazývají Spring Beans. Spring Container dostává instrukce, které objekty vytvořit a jak je nakonfigurovat prostřednictvím metadat. Metadata lze získat třemi následujícími způsoby:

- XML;
- Java anotace;
- Java kód. [4] [8]

Na obrázku 2 je znázorněn proces tvorby Spring Bean schematicky, kde třídy Java POJO vstupují do Spring Containeru, který na základě instrukcí přijatých prostřednictvím metadat vytvoří Spring Bean, který je připraven k použití v aplikaci.



Obrázek 2: Proces tvorby Spring Bean

Zdroj: [8]

Spring Framework obsahuje dva různé druhy kontejnerů:

- Spring BeanFactory Container;
- Spring ApplicationContext Container. [4] [8]

Spring ApplicationContext Container

ApplicationContext je složitější a pokročilejší Spring Container než BeanFactory. Stejně, jako BeanFactory, ApplicationContext načte Spring Beans, propojí je dohromady a nakonfiguruje určitým způsobem. Kromě toho má ApplicationContext další funkce: rozpoznávání textových zpráv z konfiguračních souborů a zobrazování událostí, které se v aplikaci vyskytují. Tento kontejner je definován rozhraním ApplicationContext a nejvíce se používají následující implementace:

- FileSystemXmlApplicationContext;
- ClassPathXmlApplicationContext;
- WebXmlApplicationContext. [2] [8]

Beany

Beany jsou objekty, jež jsou základem aplikace a jsou spravovány kontejnerem Spring IoC. Tyto objekty jsou vytvořeny pomocí konfiguračních metadat, která jsou specifikována v kontejneru. Každá definice beanu obsahuje konfigurační metadata, jež potřebuje řídící kontejner k získání informací ohledně vytvoření beanu, informace o životním cyklu beanu, závislosti beanu. Každý bean má svou oblast viditelnosti:

- Singleton – definuje jediný bean v kontejneru Spring IoC. Tato vlastnost se používá jako výchozí nastavení pro bean.
- Prototype – umožňuje mít jakýkoliv počet instancí beanu.
- Request – vytváří se jedna instance beanu pro každý HTTP požadavek.
- Session – vytváří se jedna instance beanu pro každou HTTP relaci.
- Global – vytváří se jedna instance beanu pro každou globální HTTP relaci. [1] [8]

3.2.2 IoC

Inversion of Control (IoC) je jedním z návrhových vzorů, který uvolňuje pevné vazby mezi objekty a deleguje některé naše povinnosti jiné komponentě. Ve Spring Framework IoC znamená, že se o vytvoření objektů bude starat ApplicationContext, což je reprezentace IoC kontejneru. Výhoda IoC je hlavně v tom, že vývojář nemusí vytvářet samostatně potřebné objekty v každé třídě. Jedním ze způsobů, který zajišťuje princip IoC, je DI. [4] [8]

3.2.3 DI

DI je návrhový vzor, který je specifičtější verzí IoC, kde jsou implementace předávány do objektu prostřednictvím konstruktorů/setterů/nastavení, na kterých bude objekt záviset, aby se choval správně. [4] [8]

3.2.4 Data Access/Integration

Kontejner Data Access/Integration se skládá z JDBC, ORM, OXM, JMS a modulu Transactions. JDBC poskytuje abstrakční vrstvu JDBC a eliminuje potřebu, aby vývojář ručně zapisoval opakující se kód spojený s připojením k databázi. ORM poskytuje integraci s oblíbenými ORM, jako je Hibernate atd. Důležitou součástí Spring Data je Spring Data JPA, která usnadňuje práci s databází. Spring Data JPA si klade za cíl výrazně zlepšit implementaci vrstev pro přístup k datům snížením úsilí na množství, které je skutečně potřeba. Vývojář poté musí jen provést implementaci rozhraní JpaRepository a napsat své vlastní vyhledávací metody. Spring zajistí implementaci automaticky. [4] [8]

3.2.5 WEB

Tato skupina se skládá z Web, Web-Servlet, Web-Struts a Web-Portlet. Tyto moduly poskytují podporu pro tvorbu webových aplikací. [4] [8]

3.3 Spring Boot

Spring Boot je nadstavbou nad Spring Framework a poskytuje všechny funkce jako Spring Framework, ale je jednodušší v nastavení a použití. Díky svým vlastnostem, jako je automatická konfigurace, Spring Boot nepotřebuje zbytečnou konfiguraci. Také Spring Boot nevyžaduje nasazení war souboru, ale přímo nasadí webovou aplikaci do Tomcat web serveru. Dále je možné nastavit i jiné webové servery, jako je WildFly nebo Jetty. Nejjednodušší způsob, jak vytvořit aplikaci pomocí Spring Boot, je použít šablonu aplikace na https://start.spring.io/, kde je také možné přidat požadované závislosti do projektu. Spring Boot poskytuje takzvané „starter“ závislosti, jež obsahují všechny základní moduly a autokonfigurace. Aplikace Spring Boot se spouští přes metodu main ve třídě, která je označena anotací @SpringBootApplication. Anotace @SpringBootApplication zapíná automatickou konfiguraci a skenování komponent. [2] [8] [10]

3.4 Vaadin

Vaadin je open-source framework pro vývoj webových aplikací v Javě/Kotlinu bez použití webových technologií na straně klienta, jako je HTML a JavaScript. Vaadin

umožňuje Java/Kotlin programátorům rychle vytvářet a udržovat různé webové aplikace, je kompatibilní s hlavními operačními systémy, webovými prohlížeči a webovými servery, ale na rozdíl od jiných webových frameworků má architekturu na straně serveru. [11]

Architektura na straně serveru umožňuje programátorům provozovat většinu byznys logiky na webovém serveru. Vývojáři mohou také zajistit, aby aplikace poskytovala interaktivní uživatelské prostředí spuštěním AJAX ve webových prohlížečích. Vaadin také zjednodušuje vývoj webových aplikací poskytováním UI designeru, řídícího panelu, nástrojů pro vizualizaci dat a add-on komponent. [11]

3.4.1 Technologický základ Vaadin

Vaadin je založen na různých technologiích, které jsou důležité pro správný běh frameworku. Nejdůležitější z nich jsou popsány níže.

HTML a Javasript

HTML je základním jazykem ve World Wide Web (WWW). Vaadin Framework používá HTML verzi 5, ale do značné míry skrývá použití HTML, což umožňuje se soustředit na struktuře komponent uživatelského rozhraní. Při vývoji na straně serveru je uživatelské rozhraní vyvinuto v Javě pomocí komponent uživatelského rozhraní a vykreslováno modulem na straně klienta jako HTML. Je také možné použití šablon HTML. [11]

JavaScript je na druhou stranu programovací jazyk, který umožňuje manipulaci se stránkami HTML prostřednictvím modelu Document Object Model (DOM) a zpracovává události od uživatele. Vaadin modul a jeho widgety na straně klienta fungují podobně, ačkoli je ve skutečnosti v Javě, která je kompilována do JavaScriptu pomocí Vaadin Client Compiler. [11]

CSS a Sass

Zatímco HTML definuje obsah a strukturu webové stránky, Cascading Style Sheet (CSS) je jazyk pro definování vizuálního stylu, jako jsou barvy, velikost textu, okraje atd. CSS je založen na sadě pravidel, kde pravidlo obsahuje selektor a blok deklarací. [11] [16]

Syntactically Awesome Stylesheets (Sass) je rozšíření jazyka CSS, které umožňuje použití proměnných, vnořování a mnoho dalších syntaktických funkcí, díky nimž je používání CSS jednodušší a přehlednější. Sass má dva alternativní formáty, jako je SCSS a starší syntaxi, která je stručnější. Kompilátor Vaadin Sass podporuje syntaxi SCSS. [11]

AJAX

Jak již bylo zmíněno výše, AJAX se používá pro asynchronní komunikaci. Ve výchozím nastavení Vaadin používá Ajax mezi prohlížečem a serverem. V případě potřeby obousměrné komunikace nebo nižší latence je možné použít WebSocket. Některé moduly na straně klienta komunikují se serverem pomocí volání RPC a serializací dat. [11]

GWT

Klientská strana Vaadin Framework je založena na GWT. Jeho účelem je umožnit vývoj webových uživatelských rozhraní pomocí Javy místo JavaScriptu. Moduly na straně klienta jsou vyvíjeny v Javě a kompilovány do JavaScriptu pomocí kompilátoru Vaadin, což je rozšíření kompilátoru GWT. [11]

Java Servlets

Servlet je třída, jež zpracovává požadavky a vrací odpovědi. Servlety jsou pod kontrolou Java web serveru (Servlet Container), který přijímá požadavky uživatele a následně jej předá cílovému servletu. Servlety mohou vracet HTML stránky, Java Servlet Pages (JSP) nebo jiný obsah. [4] [11]

Uživatelská rozhraní aplikací Vaadin na straně serveru běží jako servlety a jsou zabaleny do třídy servletů VaadinServlet. VaadinServlet je hlavní servlet, který zpracovává všechny příchozí požadavky do aplikace. Na počáteční požadavek vrátí stránku HTML a poté většinou odpovědi ve formátu JSON k synchronizaci widgetů. UI prvky na straně serveru jsou implementovány jako potomci třídy UI. [11]

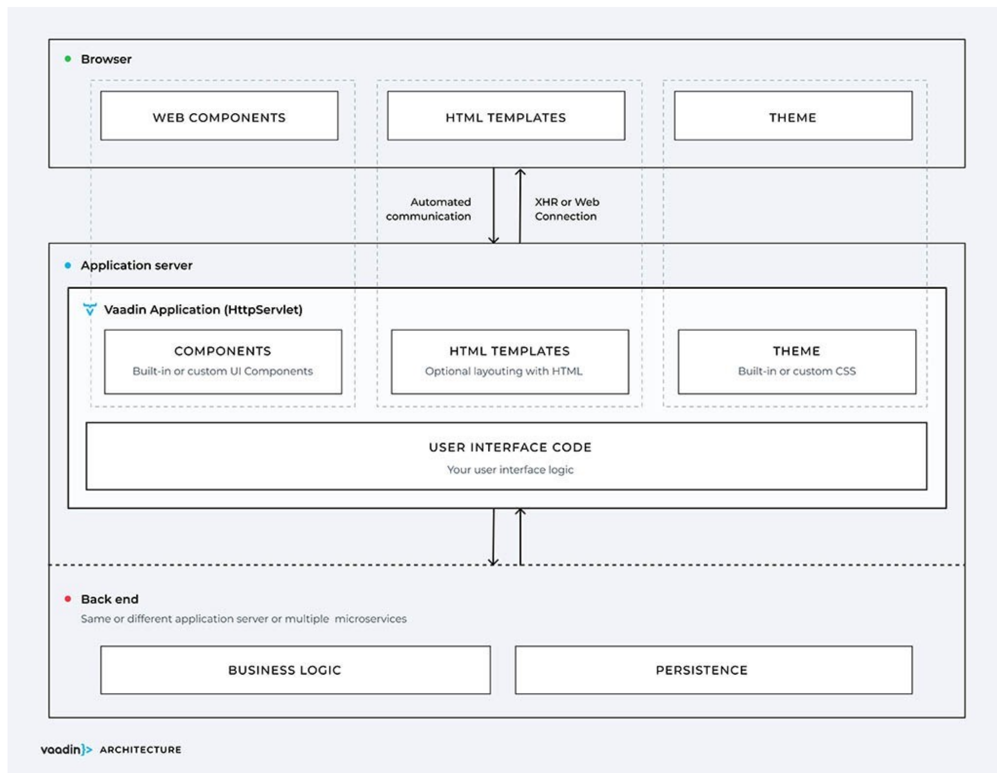
Klientský modul ve Vaadin se načítá do prohlížeče jako statické soubory JavaScriptu. Modul na straně klienta nebo sada widgetů z technického hlediska musí být umístěny pod cestou VAADIN/widgetsets ve webové aplikaci. Klientský modul je automaticky zkompilován tak, aby zahrnoval výchozí sadu widgetů a všechny nainstalované add-on komponenty. [11]

3.4.2 Architektura Vaadin

Ve Vaadinu jsou všechny prvky uživatelského rozhraní rozděleny do webových komponent. To dělá vývoj jednodušší, protože každý prvek je oddělen a izolován. Architektura Vaadin je znázorněna na obrázku 3 a zahrnuje následující vlastnosti:

- Typově bezpečné rozhraní Java UI Component API na straně serveru, které usnadňuje používání webových komponent.
- Automatickou obousměrnou komunikaci mezi serverem a prohlížečem.

- Obousměrnou datovou vazbu: v případě změn UI na straně klienta nebo serveru se změny automaticky projeví u druhé strany.



Obrázek 3: Architektura Vaadin

Zdroj: [11]

Vaadin také umožňuje přistupovat k rozhraním API prohlížeče, webovým komponentám a dokonce i k prvkům DOM přímo ze strany serveru Java. Není nutné rozumět tomu, jak funguje komunikace klient-server nebo webové komponenty. [11]

3.4.3 Web components

Web Components je kolekce webových standardů, která umožňuje vytvářet nové HTML tagy s vlastním názvem, opětovné použití a úplné zapouzdření stylů. Jedna z důležitých výhod webových komponent je v tom, ze jednou napsaný kód může být použit i v jiných částech aplikace. U tradičních značkovacích jazyků, jako HTML, to nebylo tak snadné a HTML kód byl často složitý a nepřehledný. Web Components si klade za cíl tyto problémy vyřešit – skládá se ze tří hlavních technologií, které lze společně použít k vytvoření všestranných vlastních prvků se zapouzdřenými funkcemi, které lze znovu použít. Přístup k implementaci webové komponenty:

- Vytvořit třídu, která bude obsahovat vlastnosti webové komponenty pomocí syntaxe ECMAScript 2015.
- Zaregistrovat nový vlastní prvek pomocí metody CustomElementRegistry.define() a předat mu název prvku, který má být definován, třídu nebo funkci, ve které je specifikována jeho funkčnost.
- V případě potřeby připojit stínový DOM k uživatelskému prvku pomocí metody Element.attachShadow().
- V případě potřeby definovat šablonu HTML pomocí <template> a <slot>. Použít běžné metody DOM pro klonování šablony a její připojení ke stínovému DOM.
- Použít svůj vlastní prvek kdekoliv na stránce, stejně jako jakýkoliv běžný prvek HTML. [11]

3.4.4 Vaadin Components

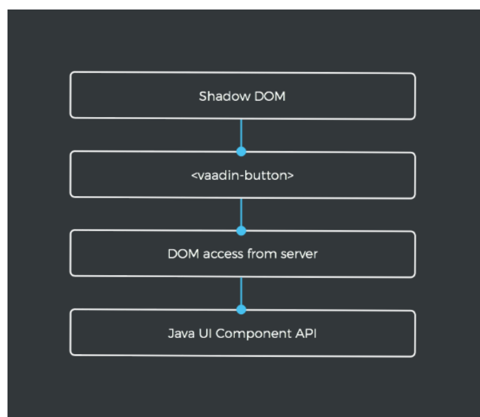
Jádrem Vaadinu jsou UI komponenty, které slouží k vytváření interaktivních webových aplikací. Vaadin poskytuje Java Web Components API, které umožňuje vytvářet uživatelská rozhraní extrémně produktivním způsobem. Není potřeba žádné HTML ani JavaScript znalosti a pouze dostatek zkušeností s CSS ke stylování a chování aplikace. [11] Java Web Components API obsahuje předpřipravené komponenty na straně serveru a nativní prvky HTML. Příklady komponent Vaadin:

- Grid – komponenta umožňuje zobrazovat tabulky a upravovat data v řádcích a sloupcích. Je vysoce flexibilní, snadno se používá a nabízí mnoho funkcí.
- TextField umožňuje uživateli vkládat a upravovat text. Podporovány jsou také předpony a přípony, například ikony.
- Form Layout – umožňuje vytvářet responzivní formuláře s více sloupci. [11]

3.4.5 Princip fungování Vaadin komponent

Architektura komponenty je znázorněna na obrázku 4 a Vaadin ovládá DOM na webu pomocí Java kódu. Strom DOM je reprezentován Java kódem na straně serveru. Všechny změny jsou automaticky synchronizovány se skutečným DOM stromem v prohlížeči. Strom DOM se skládá z instancí třídy Element, kde každá instance představuje prvek DOM v prohlížeči. Kořen stromu DOM na straně serveru je Element, který je instancí třídy UI. K němu je možné přistupovat pomocí metody ui.getElement. Tento element představuje tag <body>. Prvky na serveru jsou implementovány jako flyweight instance. To znamená, že není možné porovnávat prvky pomocí operátorů == a !=. Místo toho je nutné

použít metodu element.equals(otherElement) pro kontrolu, zda dvě instance odkazují na stejný prvek v DOM. [11]



Obrázek 4: Architektura komponenty Vaadin

Zdroj: [11]

Třída Component obaluje Element a poskytuje vyšší úroveň abstrakce. Element je možné získat pomocí metody Component.getElement. Prvek Component může obsahovat libovolný počet Elementů. Component může také obsahovat další Component. V hierarchii komponenty je možné procházet pomocí component.getParent pro navigaci nahoru a component.get-Children pro navigaci dolů. Hierarchie komponent je konstruována na základě hierarchie prvků. [11]

3.4.6 Navigace

Vaadin poskytuje třídu Router pro navigace ve webové aplikaci. K registraci navigace se používá anotace @Route. Je možné zadat cestu a volitelně i rodičovský pohled pro zobrazení komponenty pomocí hodnoty layout. Pomocí anotace @ParentLayout lze definovat rodičovský pohled pro komponenty v hierarchii směrování. [11]

3.4.7 Způsoby dopravy

Vaadin zajišťuje komunikaci mezi prohlížečem a serverem pomocí asynchronních AJAX požadavků, ale není to jediný způsob přenosu dat, který podporuje Vaadin. K dalším způsobům přenosu dat ve Vaadin patří WebSocket a HTTP Long Polling, které budou užitečné pro realtime aplikace. [11]

HTTP Long Polling

HTTP Polling je dalším krokem vpřed oproti klasickému mechanismu request/response – ačkoli existují různé verze dotazování, pouze Long Polling je použitelným způsobem pro aplikace v reálném čase, kde klient potřebuje vědět o nových

datech, jakmile jsou k dispozici. HTTP Long Polling je technika dotazování, kdy se server rozhodne ponechat připojení klienta otevřené tak dlouho, jak je to jen možné (obvykle až 20 sekund), přičemž odpověď doručí pouze po získání dat nebo po vypršení časového limitu. Hlavní výhoda Long Polling je v tom, že nová data jsou odeslána klientovi, jakmile jsou k dispozici. [11]

WebSockets

WebSockets umožňují serveru i klientovi odesílat zprávy kdykoli bez jakéhokoli vztahu k předchozímu požadavku. Téměř každý prohlížeč podporuje WebSockets.
Vlastnosti Websockets:

- Obousměrný protokol – klient/server může poslat zprávu druhé straně;
- Plně duplexní komunikace – klient a server mohou posílat zprávy nezávisle současně;
- Jediné připojení TCP – klient a server komunikují prostřednictvím stejného připojení TCP po celou dobu životního cyklu připojení WebSocket. [11]

3.4.8 Integrace Spring Boot a Vaadin

Vytvořit aplikaci pomocí Spring Boot a Vaadin je možné na webu vaadin.com/start nebo start.spring.io/, kde je také možnost přidávat nutné závislosti do projektu. Spring Boot, Vaadin poskytuje závislost „vaadin-spring-boot-starter“, která obsahuje všechny základní moduly a konfigurace. Doporučuje se také deklarovat vaadin-bom, pokud je potřeba mít další závislosti Vaadin. Pro produkční sestavení se doporučuje také přidat vaadin maven-plugin, který generuje optimalizované JavaScript balíčky, viz výpis kódu 1. [11]

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>${vaadin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>
        vaadin-spring-boot-starter
    </artifactId>
    <version>${vaadin.version}</version>
</dependency>
</dependencies>
<build>
<plugins>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>
```

```
            spring-boot-maven-plugin
        </artifactId>
    </plugin>
    <plugin>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-maven-plugin</artifactId>
        <version>${vaadin.version}</version>
        <executions>
            <execution>
                <goals>
                    <goal>prepare-frontend</goal>
                    <goal>build-frontend</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
```

Výpis kódu 1: Závislosti pro Vaadin Spring Boot

Spring DI funguje v komponentách s anotací @Route. Tyto komponenty jsou vytvořeny pomocí Spring jako beans a jsou přidány do ApplicationContext. [11]

3.5 JUnit a @SpringBootTest

JUnit framework umožňuje psát jednotkové testy a otestovat jednoduchou logiku komponent. Pro psaní integračních testů ve Spring Boot je potřeba použít Spring Boot test runner, kde anotace @SpringBootTest zajišťují, že je aplikace Spring Boot před spuštěním testů inicializována a umožňuje použít anotaci @Autowire. Pro psaní testů ve Spring Boot aplikaci stačí přidat do pom.xml spring-boot-starter-test závislost. [2] [8]

3.6 Docker

Docker je otevřená platforma pro vývoj, poskytování a provoz aplikací. Skládá se z nástroje příkazového řádku dockeru, který vyvolává službu se stejným názvem a vyžaduje přístup root. [12]

3.6.1 Základní pojmy

Container image – soubor, ve kterém je zabalena aplikace a její prostředí. Obsahuje souborový systém, který je k dispozici aplikaci a další metadata (jako příkazy, které se mají provést při spuštění kontejneru). Obrazy kontejnerů se skládají z vrstev (obvykle jedna vrstva – jedna instrukce). Různé obrazy mohou obsahovat stejné vrstvy, protože každá vrstva je postavena na jiném obrazu a dva různé obrazy mohou používat stejný nadřazený obraz jako základ. Obrazy jsou uloženy v Registry Server (registr) a verzovány pomocí tagů. [12]
Container je instancí obrazu kontejneru. Kontejner je běžící proces izolovaný od ostatních procesů na serveru a omezený vyhrazeným množstvím zdrojů. Kontejner uloží všechny vrstvy obrazu s přístupem pro čtení a vytvoří jeho spustitelnou vrstvu nahoře s přístupem pro zápis. [12]

Registry Server je úložiště, které ukládá obrazy. Například Docker Hub (docker.io) nebo RedHat Quay.io (quay.io). [12]

Container Engine je softwarová platforma pro balení, distribuci a spouštění aplikací, která stahuje obrazy a z pohledu uživatele spouští kontejnery (ve skutečnosti je Container Runtime zodpovědný za vytváření a provoz kontejnerů). Příklady: Docker nebo Podman. [12]

Container Runtime je softwarová komponenta pro vytváření a spouštění kontejnerů, například runc nebo crun. [12]

Host – server, na kterém běží Container Engine a běží kontejnery. [12]

Docker Daemon zpracovává příchozí Docker API požadavky na základě určitých podmínek. [12]

Docker Client – rozhraní, pomocí kterého uživatelé komunikují s Docker Daemon. Docker Daemon zpracovává příkazy zadané uživatelem, je schopen stáhnout obraz z registru, pokud jej ještě nemá, a vytvořit kontejner pomocí tohoto obrazu. V případě, že démon najde obraz ve svém vlastním úložišti, tak nebude obraz stažen z registru. [12]

3.6.2 Práce s Docker

Pro instalaci je potřeba postupovat podle oficiální dokumentace, která je uvedena na oficiálním webu docker.com. Kontejnery vyžadují funkce linuxového jádra, jako je namespaces a cgroups, takže fungují nativně pod Linuxem, téměř nativně v nejnovějších verzích Windows díky WSL2 (přes Docker Desktop nebo distribuci Linuxu). Doporučuje se používat v testovacím a zejména v produkčním provozu pouze Linux. [12]

3.6.3 Vytvoření obrazu

Vytváření a distribuce obrazů je jedním z hlavních úkolů Dockeru. Vytvořit obraz je možné jedním z následujících způsobů:

- Provést commit změn z kontejneru. Je nutné spustit kontejner interaktivně ze základního obrazu, provést změny a potvrdit výsledek do obrazu pomocí příkazu commit. V praxi je metoda vhodná pro malá rychlá vylepšení.
- Deklarativní popis přes Dockerfile. Je to nejpoužívanější způsob vytváření obrazů. Je nutné vytvořit Dockerfile s deklarativním popisem ve formátu yaml pomocí textového editoru a vytvořit obraz pomocí příkazu build. [12]

3.6.4 Multi-stage docker build

Docker multi-stage build je relativně nová vlastnost Docker, která je dostupná od verze Docker 17.05. Před verzí 17.05 se všechny procesy sestavování obrazu (stahování knihoven, kompilace, testování a balení) ukládaly do jednoho Dockerfile souboru, což mělo následující problémy:

- Dockerfile je rozsáhlý a špatně čitelný;
- velký počet úrovní obrazu;
- riziko úniku zdrojového kódu. [12]

Tyto problémy řeší multi-stage build, který umožňuje umístit více příkazů FROM do stejného souboru Dockerfile. Každá instrukce FROM může používat jiný základní obraz. Každý z příkazů FROM označuje začátek nové fáze procesu Docker build. Je možné také kopírovat build artefakty z jedné fáze do druhé a takto vyhodit to, co nepotřebujeme. Je to velmi užitečná funkce v situacích, kdy není potřeba, aby se závislosti sestavení zkopírovaly do konečného obrazu a pomáhá vytvářet obrazy, které jsou menší. Způsob použití multi-stage Docker je ukázán v praktické části práce. [12]

3.7 Hibernate

Hibernate je ORM nástroj, který umožňuje provádět mapování tabulky na POJO třídy, poskytuje jednoduché API pro čtení a zápis Java objektů z/do databáze. Také Hibernate pomocí fetching strategie zmenšuje počet přístupů k databázi. Hibernate podporuje všechny základní databáze, jako je Oracle, PostgreSQL, MySQL apod. Pro vytvoření dotazu je možné použít Hibernate Query Language (HQL) nebo Structured Query Language (SQL), kde rozdíl mezi HQL a SQL je v tom, že SQL pracuje s tabulkami v databázi a jejich sloupci, zatímco HQL pracuje s persistentními objekty a jejich poli (atributy tříd). [4] [9]

Hibernate podporuje API JDBC, JNDI, JTA. JDBC poskytuje nejjednodušší úroveň abstrakce funkčnosti pro relační databáze. JTA a JNDI zase umožňují Hibernate používat aplikační servery J2EE. Architektura Hibernate se skládá z následujících komponent:

- **Transaction –** je jednotkou práce s databází. V režimu spánku jsou transakce zpracovávány správcem transakcí.
- **Session Factory –** nejdůležitější a nejtěžší objekt (obvykle se vytváří při spuštění aplikace). Pro každou databázi je potřeba mít alespoň jednu SessionFactory, kdy každá je nakonfigurována v samostatném konfiguračním souboru.

- **Session –** používá se k získání fyzického připojení k databázi. Obvykle je relace vytvořena v případě potřeby a poté uzavřena. To je způsobeno tím, že jsou extrémně lehké. Vytváření, čtení, modifikace a mazání objektů probíhá prostřednictvím objektu Session.
- **Query –** tento objekt používá HQL nebo SQL ke čtení/zápisu dat z/do databáze. Instance dotazu se používá k navázání parametrů dotazu, omezení počtu výsledků, které budou vráceny a k provedení dotazu.
- **Configuration –** slouží k vytvoření objektu SessionFactory a konfiguruje Hibernate pomocí konfiguračního souboru XML, který vysvětluje, jak zacházet s objektem Session.
- **Criteria –** používá se k vytvoření a provedení objektově orientovaného dotazu k získání objektů. [4] [9]

Jeden ze způsobu konfigurace Hibernate jsou anotace, které hrají roli metadat. Mezi nejdůležitější anotace patří následující:

- **@Entity –** tato anotace říká Hibernate, že daná třída je entita bean. Taková třída musí mít výchozí konstruktor (prázdný konstruktor).
- **@Table –** říká Hibernate, na kterou tabulku má tuto třídu namapovat. Anotace @Table má různé atributy, pomocí kterých můžeme určit název tabulky, adresář, databázi a jedinečnost sloupce v databázové tabulce.
- **@Id –** určuje primární klíč třídy.
- **@GeneratedValue –** používá se ve spojení s anotací @Id a definuje parametry jako strategie a generátor.
- **@Transactional –** definuje rozsah jedné databázové transakce.
- **@EntityGraph –** umožňuje specifikovat atributy, které se mají zahrnout, když chceme načíst entitu a související asociace.
- **@Column –** definuje, do kterého sloupce v databázové tabulce patří konkrétní pole třídy. [4] [9]

3.8 Maven

Maven je nástroj pro řízení a sestavení projektů. Umožňuje vývojářům plně řídit životní cyklus projektu. Maven umožňuje automatizovat procesy spojené se sestavením, testováním, balením projektu atd. Struktura a obsah projektu Maven je specifikován ve speciálním xml souboru s názvem Project Object Model (POM), který je základním modulem celého projektu. Je vždy uložen v základním adresáři projektu a nazývá se

pom.xml. Soubor POM obsahuje informace o projektu a různé podrobnosti o konfiguraci, které používá Maven k sestavení projektu. [4] [8]

3.9 Heroku

Heroku je řešení Platform as Service (PaaS) vlastněné společností Salesforce pro vytváření, nasazování a správu cloudových služeb a aplikací. Heroku, původně zamýšlené pro použití s Ruby on Rails, nyní podporuje i další programovací jazyky jako Java, Python, Scala, PHP, Clojure a Node.js. Heroku také usnadňuje integraci s databází MySQL, která bude využita pro tuto práci. Klíčové termíny Heroku zahrnují:

- Heroku dynos: virtualizované kontejnery, které jsou „stavebními kameny“ webových aplikací Heroku, jež nabízí flexibilitu v reálném čase a automatické škálování.
- Add-ons Heroku: pluginové nástroje a služby, které rozšiřují funkčnost aplikace Heroku.
- Command Line Interface (CLI) je nástroj Heroku pro vytváření a spouštění aplikací Heroku přímo v počítačovém terminálu.
- Git: systém správy verzí pro ukládání a sledování změn zdrojového kódu v průběhu času. Uživatelé Heroku mohou použít CLI platformy k nasazení s Git. K nasazení aplikace je také možné využít Container Registry, tento způsob bude uveden v praktické části. [13]

4 Návrh

V této kapitole je provedena analýza požadavků, kterým musí aplikace odpovídat. Také při analýze a sběru požadavků budou specifikováni aktéři a případy užití aplikace.

4.1 Specifikace požadavků

Při návrhu softwarové aplikace je důležité specifikovat požadavky, které má software splňovat. Požadavky k softwaru se často rozdělují na funkční a nefunkční. Cílem funkčních požadavků je odpovědět na otázku, co má daná aplikace dělat. Nefunkční požadavky naopak neurčují chování systému, ale slouží k popisu jeho atributů, omezení. [3] [5]

V tabulce 1 jsou uvedeny funkční a nefunkční požadavky pro aplikaci tvorby rozpočtu.

ID	Popis	Typ požadavku	Priorita
R1	Systém musí poskytovat správu uživatelů	Funkční	Musí být
R2	Systém musí poskytovat správu nabídek	Funkční	Musí být
R3	Systém musí poskytovat správu zákazníků	Funkční	Musí být
R4	Systém musí poskytovat správu firem	Funkční	Musí být
R5	Systém musí poskytovat možnost přiřazování firmy uživateli	Funkční	Musí být
R6	Systém musí poskytovat správu položek	Funkční	Musí být
R7	Systém musí poskytovat správu vlastního účtu	Funkční	Musí být
R8	Systém musí podporovat autentizaci uživatelů	Funkční	Musí být
R9	Systém musí podporovat autorizaci uživatelů	Funkční	Musí být
R10	Systém musí podporovat export nabídek ve formátu PDF	Funkční	Musí být
R11	Systém musí podporovat import položek do systému	Funkční	Musí být
R12	Každý uživatel musí mít aspoň jednu roli	Funkční	Musí být
R13	Systém musí poskytovat správu rolí administrátorovi	Funkční	Musí být
R14	Systém musí poskytovat filtraci položek	Funkční	Musí být

R15	Systém musí poskytovat možnost nastavit hladinu DPH, slevu, cenu za kilometr, cenu za hodinu práce, prořez a pracovní dobu pro každou firmu	Funkční	Musí být
R16	Systém musí umět správně spočítat cenu nabídky, včetně DPH	Funkční	Musí být
R17	Systém musí poskytovat uživateli možnost obnovení hesla zasláním jednorázového hesla na e-mail uživatele	Funkční	Musí být
R18	Systém bude obsahovat unit a integrační testy pro testování základních funkcionalit	Nefunkční	Musí být
R19	Systém přihlásí uživatele do 5 sekund	Nefunkční	Musí být
R20	Pro kontejnerizaci aplikace bude sloužit Docker	Nefunkční	Musí být
R21	Obraz systému bude nasazen na Heroku	Nefunkční	Musí být
R22	ORM nástrojem bude sloužit Hibernate	Nefunkční	Měl by být
R23	Pro ukládání dat bude použita databáze MySQL	Nefunkční	Měl by být
R24	Pro front-end bude použit Vaadin framework	Nefunkční	Musí být
R25	Pro back-end bude použit Spring Boot	Nefunkční	Musí být
R26	Systém bude napsán v jazyce Java	Nefunkční	Musí být
R27	Komunikace bude zabezpečena pomocí protokolu TLS	Nefunkční	Měl by být

Tabulka 1: Funkční a nefunkční požadavky

4.2 Přehled uživatelů

Většina aplikací neumožňuje každému uživateli přistupovat ke všem datům aplikace a uživatel musí mít na to odpovídající oprávnění – roli. Autorizace podle rolí umožňuje omezit přístup ke zdrojům v závislosti na skupině, do které uživatel patří. V této aplikaci uživatel může mít následující role:

- ADMIN
- USER

ADMIN

Má přístup ke všem funkcím v aplikaci a může spravovat veškerá data. Aplikace musí mít alespoň jednoho administrátora. Pouze administrátor může provádět následující akce:

- spravovat uživatele;
- spravovat firmy;
- importovat položky do systému;
- spravovat role uživatelů;
- zablokovat nebo odblokovat uživatele;
- provádět veškeré akce, které jsou dostupné pro roli USER.

USER

Běžný uživatel může spravovat položky firmy, měnit nastavení firmy (DPH, sleva, pracovní doba, cena za kilometr, cena za práci, prořez kabelu). Uživatel také může provádět následující akce:

- spravovat položky;
- spravovat zákazníky;
- spravovat nabídky;
- změnit přihlašovací údaje;
- změnit osobní údaje.

4.3 Případy užití

Jedním z důležitých kroků při návrhu softwaru je správně definovat případy užití, které umožňují lépe specifikovat a popsat požadavky k systému. Případ užití je určitá posloupnost akcí ilustrujících chování systému. V softwarovém a systémovém inženýrství jsou případy užití popisy toho, jak se systém chová v reakci na externí požadavky. Případy užití popisují „kdo" a „co" může s daným systémem dělat a zaměřují se na to, jak dosáhnout určitého cíle nebo úkolu požadovaného uživatelem. Míra podrobnosti případů užití závisí především na složitosti a aktuální fázi vyvíjeného projektu. Scénáře se mohou stát složitějšími a během procesu vývoje procházet významnými změnami. [3]

Definovat případy užití je možné v textovém formátu nebo pomocí diagramů. Rozhodnutí o způsobu popisu případů užití leží na analytikovi. V této práci jsou případy užití popsány v textovém formátu z důvodu lepší přehlednosti. Níže jsou definovány případy užití pro danou aplikaci.

Případy užití

UC1: Přihlášení

Aktéři: User

Předpoklady: Žádné

Hlavní scénář:

Tento případ použití začíná, když se uživatel chce přihlásit do systému.

1. Systém vyžaduje, aby aktér zadal své jméno a heslo.
2. Aktér zadá své jméno a heslo.
3. Systém ověří zadané jméno a heslo a přihlásí aktéra do systému.

Alternativní scénáře:

Neplatné jméno / heslo

Pokud v základním postupu aktér zadá neplatné jméno nebo heslo, systém zobrazí chybovou zprávu. Aktér si může vybrat, zda se vrátí na začátek Základního toku, nebo zruší přihlášení, čímž případ použití končí.

Dodatečné podmínky:

Pokud byl případ užití úspěšný, je aktér přihlášen do systému. Pokud ne, stav systému se nezmění.

UC2: Zobrazení položek

Aktéři: User

Předpoklady: Žádné

Hlavní scénář:

Tento případ použití začíná, když chce uživatel zobrazit vlastní položky.

1. Aktér vybere v levém navigačním menu „Položky“.
2. Systém otevře stránku s položkami uživatele.
3. Pro každou položku systém zobrazí její název, popisek, cenu bez DPH a dobu montáže.

Dodatečné podmínky: Žádné

UC3: Odebrání položky

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží položky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel odstranit položku.

1. Aktér vybere položku, kterou chce odstranit.
2. Systém otevře dialogové okno s popisem položky.
3. Aktér zmáčkne tlačítko „Odstranit".
4. Systém odstraní položku.
5. Include (Zobrazení položek).

Dodatečné podmínky:

Položka je odstraněna ze seznamu

UC4: Hledání položky

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží položky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel najít určité položky.

1. Aktér zadá název položky do textového pole a zmáčkne tlačítko „Najít".
2. Systém najde položky, které odpovídají požadovaným kritériím.
3. Pokud systém najde požadované položky
    a. Include (Zobrazení položek).
4. Jinak
    a. Systém upozorní aktéra, že žádná položka neodpovídá požadovaným kritériím.

Dodatečné podmínky: Žádné

UC5: Přidání položky

Aktéři: User

Předpoklady:

Aktér prohlíží položky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel přidat další položku do systému.

5. Aktér zmáčkne tlačítko „Přidat položku“.
6. Systém otevře dialogové okno pro zadání nutných údajů.
7. Aktér vyplní název položky, popisek, dobu trvání a cenu bez DPH a zmáčkne tlačítko „Uložit“.
8. Systém ověří zadané údaje a přidá novou položku do systému.
9. Include (Zobrazení položek).

Alternativní scénáře:

Nesprávné údaje

Pokud v základním postupu aktér zadá nesprávné údaje a validace neproběhne úspěšně, systém upozorní aktéra na nevalidní údaje a zobrazí poznámky s radou pod každým polem. Aktér si může vybrat, zda údaje opraví, nebo zruší akci, čímž případ užití končí.

Zrušení akce

Pokud v základním postupu aktér zmáčkne tlačítko „Zrušit“, akce bude zrušena, čímž případ užití končí.

Dodatečné podmínky:

Pokud byl případ užití úspěšný, nová položka je úspěšně přidána do systému. Pokud ne, stav systému se nezmění.

UC6: Přidání zákazníka

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží zákazníky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel přidat zákazníka do systému.

1. Aktér zmáčkne tlačítko „Přidat zákazníka“.
2. Systém otevře dialogové okno pro zadání nutných údajů.
3. Aktér vyplní jméno zákazníka, e-mail a telefonní číslo, zmáčkne tlačítko „Uložit“.
4. Systém ověří zadané údaje a přidá nového zákazníka do systému.
5. Include (Zobrazení zákazníků).

Alternativní scénáře:

Nesprávné údaje

Pokud v základním postupu aktér zadá nesprávné údaje a validace neproběhne úspěšně, systém upozorní aktéra na nevalidní údaje a zobrazí poznámky s radou pod každým polem. Aktér si může vybrat, zda údaje opraví, nebo zruší akci, čímž případ užití končí.

Zrušení akce

Pokud v základním postupu aktér zmáčkne tlačítko „Zrušit“, akce bude zrušena, čímž případ užití končí.

Dodatečné podmínky:

Pokud byl případ užití úspěšný, nový zákazník je úspěšně přidán do systému. Pokud ne, stav systému se nezmění.

UC7: Odebrání zákazníka

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží zákazníky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel odstranit zákazníka.

1. Aktér vybere zákazníka, kterého chce odstranit.
2. Systém otevře dialogové okno s popisem zákazníka (jméno, e-mail a telefonní číslo).

3. Aktér zmáčkne tlačítko „Odstranit“.
4. Systém odstraní zákazníka.
5. Include (Zobrazení zákazníků).

Dodatečné podmínky:

Zákazník je odstraněn ze seznamu.

UC8: Hledání zákazníka

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží zákazníky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel najít určité zákazníky

1. Aktér zadá jméno nebo e-mail zákazníka do textového pole a zmáčkne tlačítko „Najít“.
2. Systém najde zákazníky, kteří odpovídají požadovaným kritériím.
3. Pokud systém najde požadované zákazníky
    a. Include (Zobrazení zákazníků).
4. Jinak
    a. Systém upozorní aktéra, že žádný zákazník neodpovídá požadovaným kritériím.

Dodatečné podmínky: Žádné

UC9: Zobrazení nabídek

Aktéři: User

Předpoklady: Žádné

Hlavní scénář:

Tento případ použití začíná, když chce uživatel zobrazit vlastní nabídky.

1. Aktér vybere v levém navigačním menu „Nabídky“.
2. Systém otevře stránku s nabídkami uživatele.

3. Pro každou nabídku systém zobrazí její celkovou cenu a dobu montáže, status, datum vytvoření, zákazníka a akce.

Dodatečné podmínky: Žádné

UC10: Zobrazení nabídek pro zákazníka

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží zákazníky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel zobrazit nabídky pro zákazníka.

1. Aktér vybere určitého zákazníka a v zmáčkne tlačítko „Detaily“.
2. Systém zobrazí okno s nabídkami pro vybraného zákazníka.

Dodatečné podmínky: Žádné

UC11: Přidání nabídky

Aktéři: User

Předpoklady:

Aktér prohlíží nabídky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel přidat další nabídku do systému.

1. Aktér zmáčkne tlačítko „Přidat nabídku“.
2. Systém otevře dialogové okno.
3. Aktér vyplní potřebné údaje o nákladech na dopravu, materiálových nákladech, pracnosti v hodinách a vybere hladinu DPH, zmáčkne tlačítko „Uložit“.
4. Systém ověří zadané údaje a přidá novou nabídku do systému.
5. Include (Zobrazení nabídek).

Alternativní scénáře:

Nesprávné údaje

Pokud v základním postupu aktér zadá nesprávné údaje a validace neproběhne úspěšně, systém upozorní aktéra na nevalidní údaje a zobrazí poznámky s radou pod každým polem. Aktér si může vybrat, zda údaje opraví, nebo zruší akci, čímž případ užití končí.

Zrušení akce

Pokud v základním postupu aktér zmáčkne tlačítko „Zrušit", akce bude zrušena, čímž případ užití končí.

Dodatečné podmínky:

Pokud byl případ užití úspěšný, nová nabídka je úspěšně přidána do systému. Pokud ne, stav systému se nezmění.

UC12: Odebrání nabídky

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží nabídky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel odstranit nabídku.

1. Aktér vybere nabídku, kterou chce odstranit.
2. Aktér klikne na tlačítko s ikonou koše.
3. Systém odstraní nabídku.
4. Include (Zobrazení nabídek).

Dodatečné podmínky:

Nabídka je odstraněna ze seznamu

UC13: Zobrazení nabídky

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží nabídky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel zobrazit nabídku.

1. Aktér vybere určitou nabídku a klikne na ni levým tlačítkem.
2. Systém otevře detaily nabídky.

Dodatečné podmínky: Žádné

UC14: Přidání položky do nabídky zákazníka

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží nabídky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel zobrazit nabídky pro zákazníka.

1. Aktér vybere určitou nabídku a klikne na ni levým tlačítkem.
2. Systém otevře detaily nabídky.
3. Uživatel vybere určitou položku ze seznamu.
4. Systém otevře dialogové okno, kde uživatel musí zadat počet položek a zmáčknout tlačítko „Uložit“.
5. Systém přidá položku do nabídky.
6. Include (Zobrazení nabídky).

Dodatečné podmínky: Žádné

UC15: Odebrání položky z nabídky zákazníka

Aktéři: User

Předpoklady:

Aktér je přihlášen do systému a prohlíží nabídky.

Hlavní scénář:

Tento případ použití začíná, když chce uživatel odstranit položku z nabídky.

1. Aktér vybere určitou nabídku a zmáčkne na ni levým tlačítkem.
2. Systém otevře detaily nabídky.

3. Uživatel vybere položku, kterou chce odstranit a zmáčkne pravé tlačítko myši.
4. Systém zobrazí menu, kde uživatel vybere možnost „Odstranit“.
5. Systém odstraní položku z nabídky.
6. Include (Zobrazení nabídky).

Dodatečné podmínky:

Položka je odstraněna ze seznamu.

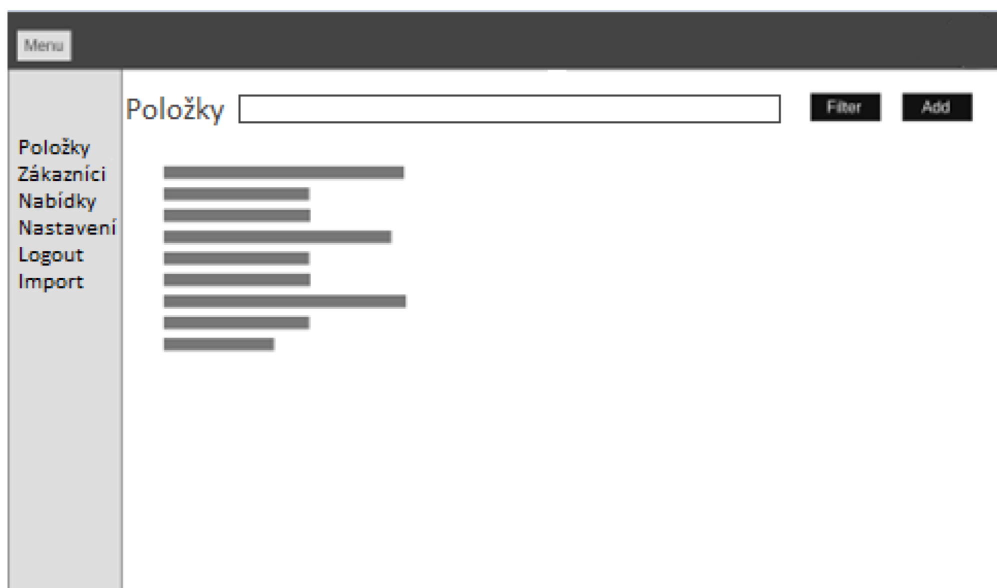
4.4 Návrh designu

Velmi často nízké udržení uživatelů je připisováno špatné použitelnosti produktu, proto je důležité přistupovat k návrhu aplikací co nejvážněji, zejména v raných fázích procesu vývoje. Návrh aplikace vyžaduje zvláštní pozornost. Design není pouze vizuální část, ale také struktura aplikace, založená na logice uživatelského chování, který musí poskytovat použitelnost a jednoduchost. Obvykle se při návrhu designu aplikace používají drátěné modely (wireframes), makety (mockups) a prototypy (prototypes). V této práci budou pro návrh designu sloužit drátěné modely, protože jsou jednoduché a pohodlné.

Drátěný model slouží k vytvoření základní kostry webu, která naznačuje jeho strukturu, umístění prvků a charakterizuje jednotlivé sekce stránky. Je to nákres toho, jak bude web vypadat. Drátěný model neřeší podobu prvků, jako jsou barvy, tvary či obrázky a ilustrace. Pro vytvoření drátěných modelů byl použit minimalistický online nástroj wireframe.cc, pomocí kterého je možné rychle vytvořit rozvržení stránky. Dále budou popsány drátěné modely pro aplikaci tvorby rozpočtu.

Obrazovka položek uživatele

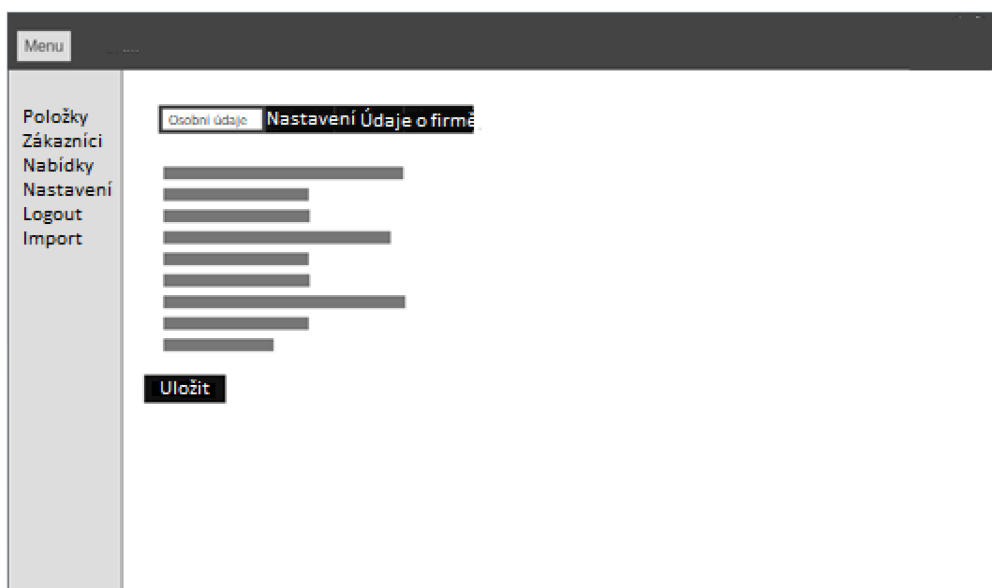
Po úspěšném přihlášení bude uživatel přesměrován na hlavní stránku s položky, viz obrázek 5, kde uživatel může spravovat vlastní položky, provádět filtraci v případě potřeby. Obrazovka je rozdělena na levé navigační menu, toolbar, záhlaví a sekci pro obsah stránky. Navigační menu poskytuje uživateli přepínání mezi stránkami položek, zákazníků a nabídek. Položka „Nastavení“ umožňuje uživateli spravovat vlastní účet a pokud je uživatel ADMIN, má možnost spravovat uživatele, role a firmy. Toolbar umožňuje uživateli provádět filtraci položek.



Obrázek 5: Wireframe – Obrazovka položek uživatele

Obrazovka nastavení

Obrazovka nastavení je znázorněna na obrázku 6 a je dostupná uživateli v navigačním menu pod názvem „Nastavení", kde uživatel může spravovat vlastní účet, změnit přihlašovací údaje nebo údaje o firmě. Pokud uživatel není administrátorem, tak se mu zobrazí jen panel osobní údaje, nastavení a údaje o firmě. Panel osobní údaje umožňuje uživateli změnit personální údaje jako jméno, příjmení, uživatelské jméno, e-mail a heslo. Panel nastavení slouží pro změnu hladiny DPH, slevy, pracovní doby, ceny dopravy a ceny práce za hodinu.



Obrázek 6: Wireframe – Obrazovka nastavení

Pokud je přihlášený uživatel zároveň administrátor, tak bude mít navíc přístup k panelům role, uživatelé a firmy. Panel role slouží pro správu rolí v systému, kde uživatel může odebrat existující role nebo přidat novou roli. Dalším panelem, který je dostupný pro administrátora, je uživatelé, který umožňuje spravovat uživatele systému. Panel firmy slouží ke spravování firem v systému.

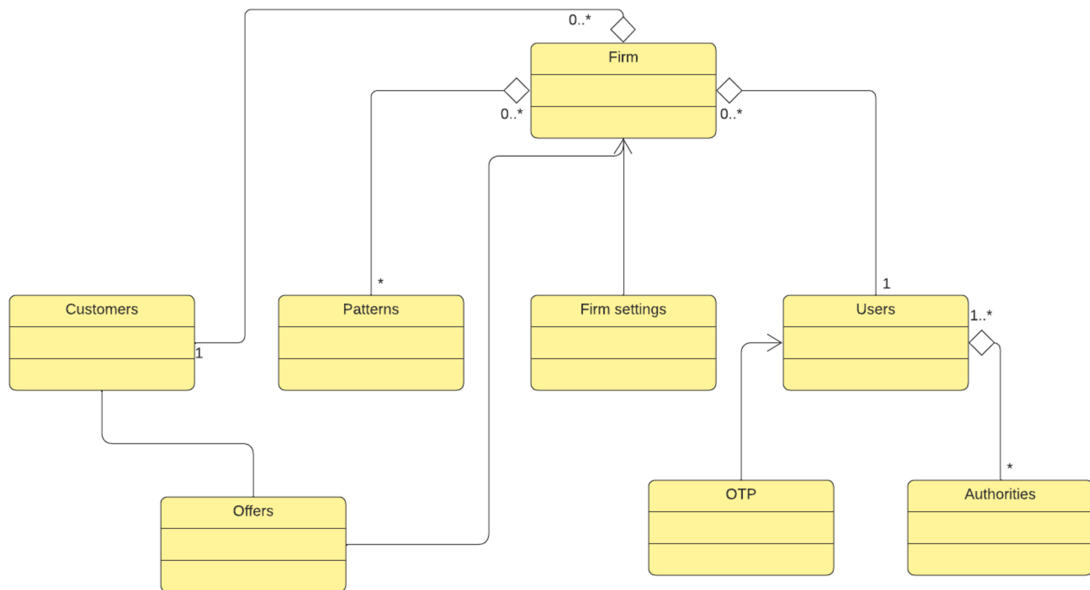
Dialog přihlášení a nová položka

Pro tvorbu nové položky slouží formulář, který je uveden na obrázku 7. Kromě textových polí pro zadání údajů formulář také obsahuje tlačítka uložit, odstranit a zrušit. Podobné formuláře se také používají pro správu nabídek, zákazníků, uživatelů, rolí a jiných údajů. Na stejném obrázku je také uveden formulář pro přihlášení uživatele do systému. V případě zapomenutého hesla formulář obsahuje možnost obnovit zapomenuté heslo. Systém provádí aktualizaci hesla pomocí jednorázového hesla a e-mailu.

Obrázek 7: Dialog – Nová položka

4.5 Doménový model

Pro vytvoření kvalitního systému nestačí rozumět jen obchodním procesům a potřebám zákazníka. Je důležité pochopit, jaká data by měl systém spravovat a k tomu je potřeba vědět, které objekty spadají do doménové oblasti navrženého systému a jaké logické vazby mezi nimi existují. K tomuto účelu se používají doménové modely. Cílem sestavení doménového modelu je získat grafické znázornění logické struktury zkoumané oblasti. Doménový model ilustruje entity a také jejich vzájemné vztahy. Vlastnosti objektů reálného světa jsou popsány pomocí atributů. Vztahy mezi entitami jsou znázorněny pomocí vazeb, kde poslední obvykle definují závislosti mezi entitami, nebo vliv jedné entity na druhou.



Obrázek 8: Doménový model

Na obrázku 8 je znázorněn doménový model pro danou aplikaci. Doménový model se skládá z osmi entit, kde každá entita má vlastní atributy a vztahy s dalšími entitami, jež jsou popsány v kapitole databázový model.

4.6 Výběr architektury

Softwarová architektura definuje strukturu systému a vysvětluje, jak systém funguje. Softwarová architektura poskytuje pevný základ, na kterém lze stavět software. Existuje mnoho architektonických vzorů a principů běžně používaných v moderních systémech. Často jsou označovány jako architektonické styly. Nesprávné architektonické rozhodnutí ovlivňuje kvalitu, výkon, udržovatelnost a celkový úspěch systému a dlouhodobé důsledky mohou ohrozit systém. Architektura softwarového systému je zřídka omezena na jeden architektonický styl. Místo toho kombinace stylů často tvoří kompletní systém. K často používaným architektonickým stylům patří následující:

- Serverless architektura;
- Událostmi řízená architektura;
- Mikroservisní architektura;
- Monolitická architektura.

4.6.1 Serverless architektura

Serverless architektura je přístup k návrhu softwaru, který umožňuje vývojářům vytvářet a provozovat aplikace a nestarat se o infrastrukturu. Vývojáři mohou psát

a nasazovat kód, zatímco poskytovatel cloudu poskytuje servery pro provoz jejich aplikací, databází a jiné služby. [14]

Jednou z nejpopulárnějších serverless architektur je Function as a Service (FaaS), kde vývojáři zapisují svůj kód jako sadu diskrétních funkcí. Každá funkce provede určitou úlohu, když je spuštěna událostí, jako je příchozí e-mail nebo požadavek HTTP. Když je funkce vyvolána, poskytovatel cloudu buď spustí funkci na běžícím serveru, nebo pokud aktuálně žádný server neběží, spustí nový server, aby funkci provedl. [14]
Výhody:

- Poskytovatelé cloudu účtují poplatky za volání, není nutné platit za servery nebo virtuální stroje.
- Instance funkcí jsou automaticky vytvářeny, nebo odstraňovány.
- Jednoduché nasazení kódu bez potřeby spravovat servery. [14]

Nevýhody

- Vyšší latence při startu funkce, v případě, že byla určitou dobu nečinná.
- Složitější testování, hlavně u integračních testů.
- Závislost na poskytovateli služeb. Pokud vznikne chyba hardwaru nebo výpadek datového centra, tak je nutné čekat na poskytovatele cloudu, který to opraví. [14]

4.6.2 Událostmi řízená architektura

Architektura řízená událostmi je způsob návrhu softwaru, který umožňuje komunikovat mezi oddělenými službami pomocí událostí a jednat podle nich v reálném čase. Tento vzor nahrazuje tradiční architekturu požadavek/odpověď, kde každá služba musí čekat na zpracování požadavku, než bude schopna pokračovat. Architektura řízená událostmi je často označována jako asynchronní komunikace. To znamená, že odesílatel a příjemce nemusí čekat a mohou pokračovat na dalších úkolech. Událostmi řízená architektura se zpravidla skládá ze tří komponent: producent, spotřebitel a zprostředkovatel. Producenti produkují události, které odesílají do zprostředkovatele. Spotřebitelé poté tyto události zpracovávají. Roli zprostředkovatele často hraje Kafka nebo RabbitMQ. [21]
Výhody

- Odstranění přímé vazby mezi API producenta a spotřebiteli.
- Lepší rozšiřitelnost a odolnost.
- Výpočetní a síťové zdroje se využívají optimálnějším způsobem. [21]

Nevýhody

- Vyžaduje další náklady a dobrou infrastrukturu.
- Náročnější ladění a monitoring. [21]

4.6.3 Mikroservisní architektura

Architektura mikroslužeb je způsob organizace architektury systému, který je založen na sadě nezávislých služeb. Tyto služby mají svou vlastní obchodní logiku a databázi se specifickým účelem. Aktualizace, testování, nasazení a škálování se provádí v rámci každé služby. Mikroslužby nesnižují složitost, ale zviditelňují a zlepšují zvládnutí jakékoli složitosti tím, že rozdělují řešení problémů na menší časti, kde každá služba řeší vlastní problematiku. [6] [7] [21]
Výhody

- Rozsah mikroslužeb je menší než u monolitických aplikací.
- Technologická flexibilita. Díky architektuře mikroslužeb je možné vybrat nástroje podle svých preferencí.
- Vysoká spolehlivost. Nasazení změn konkrétní služby nezpůsobí pád celého systému.
- Flexibilní škálování. Když mikroslužba dosáhne limitu zatížení, je možné nasadit nové instance této služby na clusteru a snížit zatížení. [6] [7]

Nevýhody

- Složitější ladění.
- Růst vývojového procesu. Více služeb vzniká na různých místech a pokud není roztahování řádně kontrolováno, zpomaluje se vývoj a snižuje se provozní efektivita.
- Vyšší požadavky na hardware.
- Složitější konfigurace a nasazení služeb. [6] [7]

4.6.4 Monolitická architektura

Monolitická architektura patří k tradičním softwarovým modelům. Představuje jedinou jednotku, která funguje autonomně a nezávisle na jiných aplikacích. Monolitická aplikace v sobě obsahuje veškerou byznys logiku, uživatelské rozhraní a serverovou část aplikace. Monolitická aplikace se může skládat z jedné až n vrstev. Všechny části softwaru jsou sjednoceny a všechny jeho funkce jsou spravovány na jednom místě. Monolity jsou skvělé pro použití v raných fázích projektů, aby usnadnily nasazení a nevyžadovaly příliš mnoho úsilí ke správě kódu. [6] [7] [21]

Výhody

- Lepší výkonnost při správném postupu vývoje aplikace. Monolitická aplikace obsahuje celou logiku a je nezávislá na ostatních komponentách, kde mikroservisní služby jsou opačným případem a jsou závislé na jiných API.
- Jednodušší ladění, protože veškerý kód je na jednom místě, což usnadňuje spouštění dotazů a hledání problémů.
- Jednoduché nasazení. Použití jednoho spustitelného souboru.
- Jednodušší testování. Monolitická aplikace je jediná centralizovaná jednotka, takže testování lze provést rychleji než u distribuované aplikace. [6] [7]

Nevýhody

- Snížená rychlost vývoje. Velká monolitická aplikace komplikuje a zpomaluje vývoj.
- Škálovatelnost. Není možné škálovat jednotlivé komponenty.
- Spolehlivost. Chyba v jednom modulu může ovlivnit dostupnost celé aplikace.
- Nasazení. V případě malé změny je potřeba znovu nasadit celou monolitickou aplikaci. [6] [7]

Pro tuto práci bylo rozhodnuto zvolit monolitickou architekturu, protože aplikace zatím nemá velký rozsah a výběr jiné architektury by zbytečně zkomplikoval projekt již na začátku. Důraz je však kladen na to, aby se aplikace skládala z nezávislých modulů, které v budoucnu v případě potřeby bude možné rozdělit na nezávislé služby.

4.7 Databázový model

Důležitým krokem při vývoji aplikace je správný návrh databázového modelu. Tvorba relačního modelu databáze se prováděla na základě analýzy funkčních a nefunkčních požadavků, doménového modelu a případů užití. Databázový model je znázorněn na obrázku 9. Níže jsou popsány databázové objekty, které byly použity v této práci.

Tabulky

Firms: Pro ukládání údajů o firmě slouží tabulka firms. Každá entita typu firma musí mít název, telefonní číslo, identifikační číslo osoby (CIN), daňové identifikační číslo (VATIN), e-mail, stát, město, ulici a poštovní směrovací číslo (ZIP). Sloupec copy_default_patterns má výchozí hodnotu nastavenou na false a po spuštění databázové procedury copy_patterns proběhne kopírování výchozích položek v tabulce položky, potom

hodnota bude změněna na true. Tabulka firmy má vztah s tabulkou firm_settings typu 1:1. Vztah typu 1:N tabulka firms má s tabulkami customers, users, patterns a také N:M s tabulkou patterns přes vazební tabulku firm_pattern.

Firm_Settings: Pro ukládání údajů o nastavení firmy slouží tabulka firm_settings. Každá firma musí mít nastavení, které se skládá z ceny práce za hodinu a dojezd, DPH, pracovní doby, slevy a prořezu. Sloupce id je zároveň primárním a cizím klíčem, který odkazuje na určitou firmu.

Users: Tabulka users obsahuje informaci o uživatelích systému. Každý uživatel musí mít další atributy: jméno, příjmení, heslo, uživatelské jméno, e-mail a čas vytvoření uživatele. Tabulka firms má vztah 1:N s tabulkou uživatele, kde cizí klíč firm_id určuje, do jaké firmy uživatel patří. Další vztah N:M tabulka users má s authorities přes vztahovou tabulku user_authority, která obsahuje cizí klíče user_id na tabulku users a authority_id na tabulku authorities. Pro ukládání jednorázového hesla slouží tabulka otp, která má vztah s tabulkou users typu 1:1.

Patterns: Tato tabulka slouží k ukládání položek. Každá položka musí mít název, popisek, dobu montáže a cenu bez DPH. Tabulka má vztah 1:N s tabulkou firms. Cizí klíč firm_id obsahuje odkaz na firmu, které položka patří. Položky, které jsou dostupné pro všechny firmy, obsahuji null ve sloupci firm_id. Při vytvoření nové firmy se všechny výchozí položky zkopírují tak, že se vytvoří nový záznam ve vazební tabulce firm_patterns pro každou výchozí položku. Uživatel poté může vytvořit specifickou položku pro vlastní účely nebo změnit existující a frm_id pak bude odkazovat na daného uživatele. Tabulka položky má také vztah s tabulkou offers přes vazební tabulku offer_pattern, kde tabulka offer_pattern obsahuje cizí klíč offer_id a pattern_id, což umožňuje určit, jaké položky obsahuje každá nabídka.

Customers: Pro ukládání údajů o zákazníkovi slouží tabulka customers. K atributům zákazníka patří jméno, email, telefonní číslo a poznámka k zákazníkovi. Cizí klíč firm_id obsahuje odkaz na firmu, se kterou zákazník spolupracuje.

Authorities: Tabulka authorities obsahuje role uživatelů a má vztah s tabulkou users přes vazební tabulku user_authority, kde user_id je cizí klíč na tabulku users a authority_id na tabulku authorities.

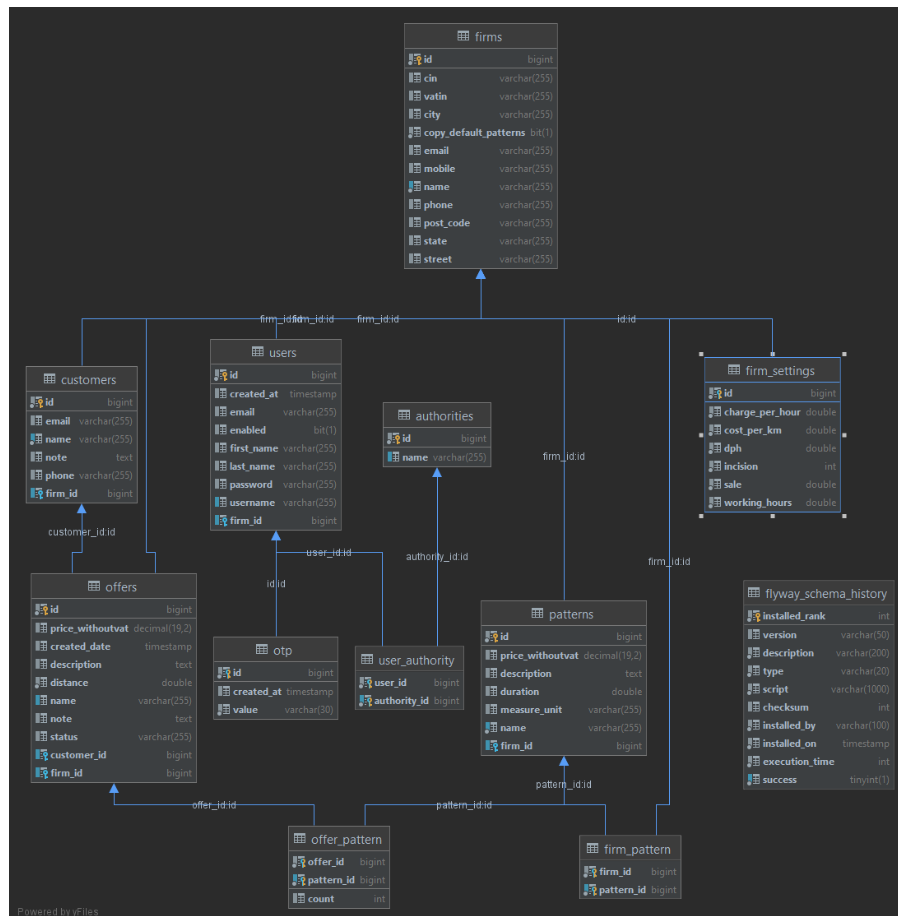
OTP: Jednorázové heslo, jež obsahuje tato tabulka, slouží k obnovení zapomenutého hesla. Povinné sloupce v této tabulce jsou hodnota a čas vytvoření OTP. Sloupec id zajišťuje vztah 1:1 s tabulkou users a hraje roli primárního, cizího klíče.

Offers: V této tabulce se ukládají údaje o nabídkách pro zákazníka. Mezi atributy nabídky patří cena bez DPH, popisek, vzdálenost, název, čas vytvoření nabídky, její status a poznámka. Cizí klíč customer_id určuje, jakému zákazníkovi patří nabídka. Cizí klíč firm_id obsahuje odkaz na firmu, která nabídku vytvořila a zpracovává.

Firm_Pattern: Tato tabulka je vazební mezi patterns a offers. Pomocí cizích klíčů offer_id a pattern_id určuje položky nabídky.

Offer_Pattern: Tato tabulka je vazební mezi patterns a offers. Pomocí cizích klíčů offer_id a pattern_id určuje položky nabídky.

User_Authority: Tabulka user_authority, stejně jako tabulka offer_patterns, je vazební tabulkou pro users a authorities, která určuje role uživatele v systému pomocí cizích klíčů user_id a authority_id.



Obrázek 9: Datový model

Procedury

Copy_Patterns: Procedura copy_patterns slouží ke kopírování výchozích položek z tabulky patterns, viz výpis kódu 2. Spouští se vždy při vytvoření nové firmy. Procedura vybírá všechny položky, které mají hodnotu null ve sloupci firm_id a vloží výsledek do vazební tabulky firm_pattern.

```
Create procedure copy_patterns(IN firmId bigint)
begin
    insert ignore into firm_pattern(firm_id, pattern_id) select firmId, id from
patterns where firm_id is null;
end;
```

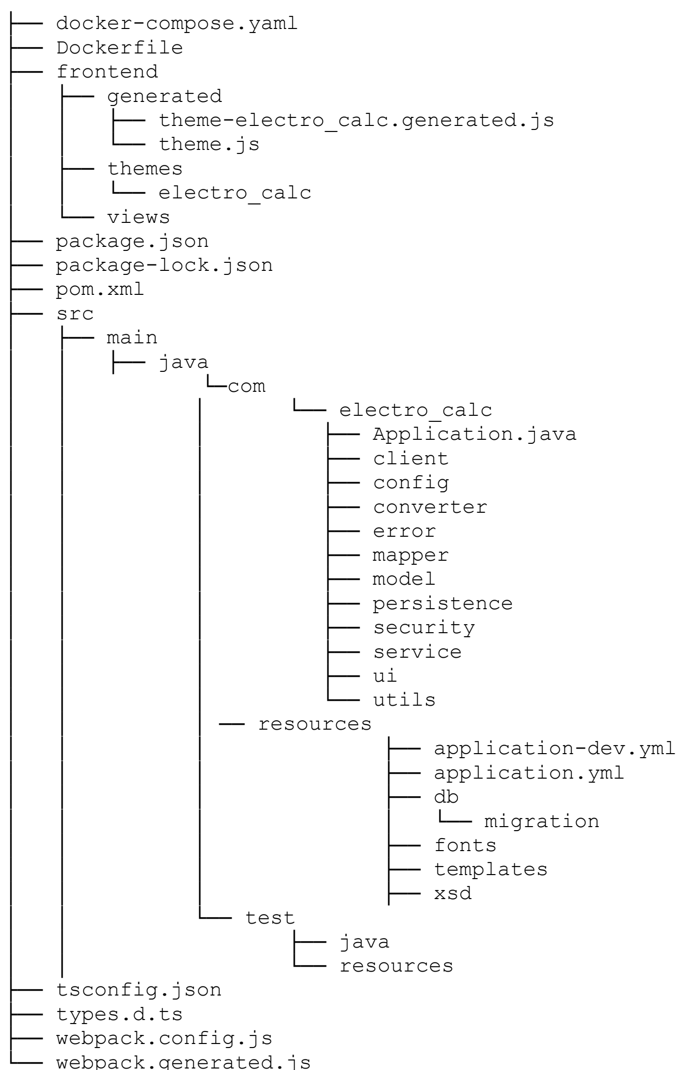
Výpis kódu 2: Procedura copy_patterns

5 Implementace

V této kapitole je popsán proces vývoje webové aplikace. V úvodu je vysvětlena struktura projektu. Dále je rozebrán každý modul, včetně back-endové a front-endové části aplikace. Také jsou popsány některé konfigurační soubory, proces kontejnerizace, nasazení aplikace a způsoby testování.

5.1 Struktura projektu

Struktura projektu vypadá takto:

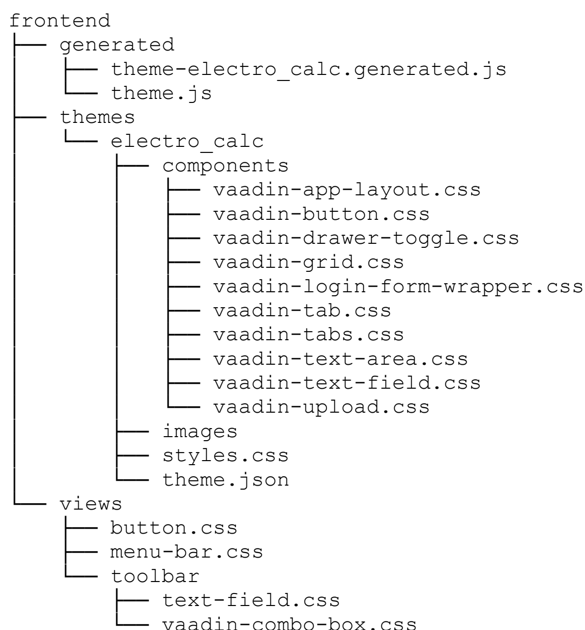


```
├── docker-compose.yaml
├── Dockerfile
├── frontend
│   ├── generated
│   │   ├── theme-electro_calc.generated.js
│   │   └── theme.js
│   ├── themes
│   │   └── electro_calc
│   └── views
├── package.json
├── package-lock.json
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │      └─com
│   │   │          └── electro_calc
│   │   │              ├── Application.java
│   │   │              ├── client
│   │   │              ├── config
│   │   │              ├── converter
│   │   │              ├── error
│   │   │              ├── mapper
│   │   │              ├── model
│   │   │              ├── persistence
│   │   │              ├── security
│   │   │              ├── service
│   │   │              ├── ui
│   │   │              └── utils
│   │   │  ── resources
│   │   │                   ├── application-dev.yml
│   │   │                   ├── application.yml
│   │   │                   ├── db
│   │   │                   │   └── migration
│   │   │                   ├── fonts
│   │   │                   ├── templates
│   │   │                   ├── xsd
│   │   └── test
│   │           ├── java
│   │           └── resources
├── tsconfig.json
├── types.d.ts
├── webpack.config.js
└── webpack.generated.js
```

Projekt byl rozdělen na front-endovou a back-endovou část. Back-endová část se nachází ve složce electro_calc a komponenty front-endu v adresáři ui. Většina konfiguračních souborů je umístěna ve složce resources a v kořenovém adresáři projektu. V dalších kapitolách je obsah každého adresáře popsán podrobněji.

5.1.1 Frontend

Adresář fronted má následující strukturu:



```
frontend
├── generated
│   ├── theme-electro_calc.generated.js
│   └── theme.js
├── themes
│   └── electro_calc
│       ├── components
│       │   ├── vaadin-app-layout.css
│       │   ├── vaadin-button.css
│       │   ├── vaadin-drawer-toggle.css
│       │   ├── vaadin-grid.css
│       │   ├── vaadin-login-form-wrapper.css
│       │   ├── vaadin-tab.css
│       │   ├── vaadin-tabs.css
│       │   ├── vaadin-text-area.css
│       │   ├── vaadin-text-field.css
│       │   └── vaadin-upload.css
│       ├── images
│       ├── styles.css
│       └── theme.json
└── views
    ├── button.css
    ├── menu-bar.css
    └── toolbar
        ├── text-field.css
        └── vaadin-combo-box.css
```

Ve složce themes se nachází CSS soubor style.css, který platí pro celou aplikaci. Složka components obsahuje styly jen pro určité komponenty Vaadin. Adresář view obsahuje CSS styly jen pro jednotlivé komponenty, na rozdíl od složky components, kde jsou definovány styly komponent pro celou aplikaci. Například soubor toolbar/text-field.css definuje styly pro textové pole ceny nabídky s a bez DPH, viz výpis kódu 3.

```css
host([theme="without-vat"]) [part="input-field"] {
    color: white;
    background-color: green;
}
:host([theme="with-vat"]) [part="input-field"] {
    color: white;
    background-color: black;
}
```

Výpis kódu 3: CSS styly pro textové pole ceny nabídky

5.1.2 Client

```
client
├── FeignConfig.java
└── MFCRInfoClient.java
```

Adresář client obsahuje třídu FeignConfig pro konfiguraci deklarativního HTTP klienta Feign, kde anotace @EnableFeignClients umožňuje skenování komponent, které jsou označeny jako @FeignClient. Feign klient MFCRInfoClient slouží ke komunikaci s ARES API, které dává možnost vyhledat informace o ekonomických subjektech v České republice. Klient obsahuje metodu getAresResponses, kde povinným parametrem metody je IČO. ARES API vrací výsledky v XML formátu, proto byl pro serializaci a deserializaci použit JAXB plugin. JAXB také umožňuje generovat Java třídy z XSD schématu. Metoda

getAresResponses vrací objekt typu AresOdpovedi, Tato Java třída byla vygenerována pomocí JAXB pluginu na základě XSD schématu, které poskytuje ARES. Vrácený výsledek se poté používá k vyplnění základních údajů o firmě jako název, adresa atd.

5.1.3 Config

Složka config obsahuje konfigurační soubory a má následující obsah:

```
config
├── CustomRequestCache.java
|── SecurityConfig.java
|── EmbeddedCacheConfig.java
```

Pro nastavení autentizace a autorizace uživatelů pomocí Spring Security slouží třída SecurityConfig a je potomkem třídy WebSecurityConfigurerAdapter, která umožňuje přizpůsobit WebSecurity i HttpSecurity konfiguraci pro danou aplikaci. Příklad nastavení Spring Security je uveden na výpisu kódu 4.

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    private static final String LOGIN_PROCESSING_URL = "/login";
    private static final String LOGIN_FAILURE_URL    = "/login?error";
    private static final String LOGIN_URL            = "/login";
    private static final String LOGOUT_SUCCESS_URL   = "/login";
    private final CustomUserDetailsService userDetailsService;

    public SecurityConfig(CustomUserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .requestCache()
                .requestCache(new CustomRequestCache())
                .and().authorizeRequests()
                .requestMatchers(SecurityUtils::isFrameworkInternalRequest)
                .permitAll()
                .anyRequest().authenticated()
                .and().formLogin()
                .loginPage(LOGIN_URL).permitAll()
                .loginProcessingUrl(LOGIN_PROCESSING_URL)
                .failureUrl(LOGIN_FAILURE_URL)
              .and().logout().logoutSuccessUrl(LOGOUT_SUCCESS_URL);
    }
…
```

Výpis kódu 4: Zabezpečení aplikace pomocí Spring Security

Výše uvedená konfigurace dělá pro aplikaci následující:

- Vypíná CSRF zabezpečení, protože Vaadin zpracovává CSRF interně;
- Zaregistruje CustomRequestCache, která přesměruje uživatele po úspěšném přihlášení na požadovanou stránku;
- Omezí přístup k aplikaci;
- Povolí všechny interní požadavky Vaadin;
- Povolí všechny požadavky od přihlášených uživatelů;

- Povolí ověření pomocí ověřování založeného na formuláři;
- Nastaví URL po úspěšném odhlášení.

Třída CustomRequestCache ukládá neověřené požadavky proto, aby přesměrovala uživatele na požadovanou stránku po úspěšném přihlášení. Dalším souborem je třída EmbeddedCacheConfig, která zapíná funkci ukládání do cache přidáním anotace @EnableCaching a registruje správce cache ConcurrentMapCacheManager s názvem cache firms, která bude sloužit k ukládání firem.

5.1.4 Converter

Obsah složky converter je následující:

```
converter
├── DurationToStringConverter.java
├── PriceConverter.java
└── StringToLongConverter.java
```

Cílem těchto tříd je převod hodnot z jednoho datového typu do jiného. Pro tento účel Spring poskytuje přednastavené různé převodníky. Pro vytvoření vlastního převodníku stačí provést implementaci rozhraní Converter. Příklad převodníku, který převede dobu montáže položky z minut na hodiny a minuty, je uveden na výpisu kódu 5.

```
@Component
public class DurationToStringConverter implements Converter<Double, String> {
    @Override
    public String convert(Double duration) {
        String tmp = "";
        int hours = (int) (duration / 60);
        int minutes = (int) (duration % 60);
        if (hours > 0)
            tmp += hours + " h ";
        tmp += minutes + " min";
        return tmp;
    }
}
```

Výpis kódu 5: Převodník doby montáže

Převodník PriceConverter slouží k formátování měny na základě hodnoty Locale.

5.1.5 Error

Složka error obsahuje třídy pro handlování chyb, které mohou vzniknout při routování nebo za běhu aplikace. Obsah složky je následující:

```
error
├── AccessDeniedExceptionHandler.java
├── AccessDeniedException.java
└── CustomErrorHandler.java
```

Třída CustomErrorHandler slouží k handlování chyb, které se mohou objevit za běhu aplikace. Pro vytvoření vlastního handleru chyb je potřeba provést implementaci rozhraní

ErrorHandler a poté pomocí této třídy přepsat výchozí obslužnou rutinu chyb. Příklad vlastní obslužné rutiny chyb je uveden na výpisu kódu 6.

```
public class CustomErrorHandler implements ErrorHandler {

    private static final Logger logger =
LoggerFactory.getLogger(CustomErrorHandler.class);

    @Override
    public void error(ErrorEvent errorEvent) {
        logger.error("Something wrong happened",errorEvent.getThrowable());
        if (UI.getCurrent() != null) {
            UI.getCurrent().access(() ->
NotificationService.error(errorEvent.getThrowable().getMessage()));
        }
    }
}
```

Výpis kódu 6: Implementace vlastní obslužné rutiny chyb

Přepsat výchozí obslužnou rutinu chyb je možné způsobem, který je uveden na výpisu kódu 7.

```
@SpringComponent
public class ServiceListener implements VaadinServiceInitListener {

    @Override
    public void serviceInit(ServiceInitEvent event) {
        event.getSource().addSessionInitListener(initEvent ->
initEvent.getSession().setErrorHandler(new CustomErrorHandler()));
    }
}
```

Výpis kódu 7: Implementace vlastního posluchače nových seancí

Třída AccessDeniedExceptionHandler se aktivuje pro neošetřené výjimky AccessDeniedException vyvolané během navigace, aby uživateli zobrazila chybovou hlášku. Navigace při vzniku chyby je vyřešena na základě typu vyvolané výjimky. V tomto případě uživatel uvidí chybovou hlášku, kterou přidává metoda setErrorParameter jen v případě, ze aplikace hodí AccessDeniedException a výjimka nebude ošetřena. Podobnou třídu je možné vytvořit i pro jiné druhy výjimek, které musí také implementovat rozhraní HasErrorParameter<T extends Exception> a při spuštění aplikace budou shromážděny všechny třídy implementující toto rozhraní.

Další třídou v tomto balíčku je AccessDeniedException, což je vlastní výjimka, která je potomkem Exception a používá se v případě, kdy uživatel přistupuje ke zdrojům, kde nemá dostatečné oprávnění.

5.1.6 Mapper

Složka mapper obsahuje třídy, jež slouží k mapování objektů. Struktura složky:

```
mapper
├── PatternMapper.java
└── UserMapper.java
```

Příklad mapování entity UserEntity do User, která je součástí Spring Security a používá se při autentizaci uživatele v aplikaci, je uveden na výpisu kódu 8.

```
public class UserMapper {
    private UserMapper() {
    }

    public static User convertToUser(UserEntity source) {
        return new User(
                source.getUsername(),
                source.getPassword(),
                true,
                true,
                true,
                source.getEnabled(),
                source.getAuthorityEntities().stream()
                        .map(it -> new
SimpleGrantedAuthority(it.getName())).collect(Collectors.toList()));
    }
}
```

Výpis kódu 8: Implementace třídy UserMapper

Po zadání přihlašovacích údajů proběhne pokus o nalezení daného uživatele a v případě úspěchu se entita user mapuje do TO třídy user. Objekt user poté slouží k vytvoření Authentication.

5.1.7 Model

Složka model obsahuje výčtové typy a data transfer objekty (DTO). Obsah složky je znázorněn níže. DTO UpdatePassword slouží k obnovení hesla uživatele. Výčtový typ OfferStatus obsahuje možné statusy nabídky.

```
Model
├── dto
│   └── UpdatePassword.java
└── enums
    └── OfferStatus.java
```

5.1.8 Persistence

Adresář persistence obsahuje třídy pro práci s vrstvou persistence. Složka je rozdělena do podsložek repository, predicate a entity s následujícím obsahem:

```
persistence
├── entity
│   ├── AbstractEntity.java
│   ├── AuthorityEntity.java
│   ├── CustomerEntity.java
│   ├── FirmEntity.java
│   ├── FirmSettingsEntity.java
│   ├── OfferEntity.java
│   ├── OfferPattern.java
│   ├── OfferPatternKey.java
│   ├── OneTimePasswordEntity.java
│   ├── PatternEntity.java
│   ├── PatternType.java
│   ├── UserEntity.java
│   └── VATEntity.java
├── predicate
│   ├── CustomerSpecification.java
│   ├── OfferSpecification.java
│
```

```
│   └── PatternSpecification.java
└── repository
    ├── AuthorityRepository.java
    ├── CustomerRepository.java
    ├── FirmRepository.java
    ├── FirmSettingsRepository.java
    ├── OfferPatternRepository.java
    ├── OfferRepository.java
    ├── PatternRepository.java
    └── UserRepository.java
```

Adresář repository obsahuje Spring Data JPA repositáře, které slouží k provedení CRUD operací, třídění a stránkování dat. Níže na výpisu kódu 9 je uveden příklad implementace repositáře pro práci s položkami.

```
public interface PatternRepository extends JpaRepository<PatternEntity, Long>,
JpaSpecificationExecutor<PatternEntity> {

   @Query("FROM PatternEntity p WHERE p.firmEntity is null and p.name =?1")
    PatternEntity findByName(String name);

   @Query("SELECT COUNT(p) FROM PatternEntity p WHERE p.firmEntity.id = ?1 OR
p.firmEntity.id is null")
    int count(Long firmId);

    @EntityGraph(attributePaths = {"offerPatterns", "firmEntities"})
    PatternEntity findFullPatternById(Long id);

    PatternEntity findPatternById(Long id);

    @Override
    long count(Specification<PatternEntity> specification);

    @Override
    @EntityGraph(attributePaths = {"firmEntity"})
    PatternEntity getOne(Long id);
}
```

Výpis kódu 9: Implementace JPA repositáře

Pro vytvoření repositáře je potřeba provést implementaci rozhraní JpaRepository. Spring Data JPA má své vlastní pojmenování metod a podle těchto konvencí je vytvořena metoda findPatternById. Další možnost vytvoření metody je findByName, kde vlastní dotaz tvoří anotace @Query a jazyk JPQL. Pro specifikaci atributů, které se mají zahrnout při načítání entity, je použita anotace @EntityGraph. @EntityGraph anotace slouží hlavně k řešení problému N+1, kde kromě 1 dotazu se provádí N dalších pro načtení asociací.

Entity jsou definovány ve složce entity. Jak již bylo zmíněno v teoretické části, entity se definují pomocí anotace @Entity. Většina entit jsou potomci entity AbstractEntity, která obsahuje pro všechny entity společný atribut id a je označena jako @MappedSuperClass. Implementace třídy AbstractEntity je uvedena na výpisu kódu 10.

```
@MappedSuperclass
public abstract class AbstractEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    public Long getId() {
        return id;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        AbstractEntity that = (AbstractEntity) o;
        return Objects.equals(id, that.id);
    }
}
```

Výpis kódu 10: Implementace třídy AbstractEntity

Entity, u kterých má smysl počítání ceny a DPH, zdědí od třídy VATEntity atributy s anotací @Transient. Hodnoty většiny těchto atributů se počítají za běhu aplikace. Níže na výpisu kódu 11 je uveden příklad třídy OfferEntity, která je potomkem VATEntity.

```
@Entity
@Table(name = "offers")
public class OfferEntity extends VATEntity {

    private String name;
    @Column(columnDefinition = "TEXT")
    private String description;
    private double distance = 0;
    @Column(columnDefinition = "TEXT")
    private String note;
    @Enumerated(EnumType.STRING)
    private OfferStatus status = OfferStatus.NONE;
    private LocalDate createdDate = LocalDate.now();
    @Transient
    private BigDecimal transportationCost = BigDecimal.ZERO;
    @Transient
    private BigDecimal materialsCost = BigDecimal.ZERO;
    @Transient
    private Double workDuration = 0.0;
    @Transient
    private BigDecimal workCost = BigDecimal.ZERO;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "customer_id")
    private CustomerEntity customerEntity;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "firm_id")
    private FirmEntity firmEntity;
    @OneToMany(
            mappedBy = "offerEntity",
            cascade = CascadeType.ALL,
            orphanRemoval = true
    )
    private Set<OfferPattern> offerPatterns;

// getters/setters
```

Výpis kódu 11: Implementace třídy OfferEntity

Další zajímavou složkou je predicate, která obsahuje Spring Data Specification pro filtrování entit na základě vstupních parametrů. Parametry firmId a name pro filtrování položek se předávají přes konstruktor. Dále se za použití criteria query vytváří SQL dotaz. Níže na výpisu kódu 12 je uveden příklad implementace Specification pro položky.

```
public class PatternSpecification implements Specification<PatternEntity> {
    private final Long firmId;
    private final String name;

    public PatternSpecification(Long firmId, String name) {
        this.firmId = firmId;
```

```
        this.name = name;
    }

    @Override
    public Predicate toPredicate(Root<PatternEntity> root,CriteriaQuery<?>
criteriaQuery, CriteriaBuilder criteriaBuilder) {
        final ArrayList<Predicate> predicates = new ArrayList<>();
        if (name != null && !name.isBlank()) {
          predicates.add(criteriaBuilder.like(root.get("name"), "%" + name + "%"));
        }
        Predicate firmEntities = criteriaBuilder.equal(
criteriaBuilder.treat(root.get("firmEntity"),
 FirmEntity.class).get("id"),
 firmId);
        Join<PatternEntity, FirmEntity> defaultEntities = root.join("firmEntities",
JoinType.LEFT);
        Predicate defaultPatternFirmPredicate = criteriaBuilder.equal(
defaultEntities.get("id"),firmId);
        predicates.add(criteriaBuilder.or(firmEntities,
defaultPatternFirmPredicate));

        criteriaQuery.orderBy(criteriaBuilder.desc(root.get("id")));
        return criteriaBuilder.and(predicates.toArray(Predicate[]::new));
    }
}
```

Výpis kódu 12: Implementace Specification pro položky

Zpočátku se přidává predikát názvu položky, pokud je parametr name validní. Dále se přidává predikát identifikátoru firmy, který umožní vrátit jen položky z tabulky patterns, které budou odpovídat parametru firmId. Pro výběr výchozích položek z vazební tabulky firm_patterns dle firmId slouží klauzule left join. Seřazení položek probíhá na základě identifikátoru v sestupném pořadí.

5.1.9 Security

Další složkou je security, která slouží k zabezpečení aplikace. Její struktura je uvedena níže.

```
security
|— ConfigureUIServiceInitListener.java
|— CustomUserDetailsService.java
|— SecurityUtils.java
```

Třída CustomUserDetailsService bývá používána k načítání dat souvisejících s uživatelem. Má jednu metodu loadUserByUsername(), která slouží k vyhledávání uživatele v databázi a používá ji DaoAuthenticationProvider k načtení podrobností o uživateli během ověřování. Třída SecurityUtils obsahuje následující užitkové metody:

- isFrameworkInternalRequest() – určuje, zda je požadavek interní pro Vaadin;
- isUserLoggedIn() – zkontroluje, zda je aktuální uživatel přihlášen;
- isAccessGranted – zkontroluje, zda aktuální uživatel má nutné oprávnění ke zdrojům.

Třída ConfigureUIServiceInitListener propojuje Spring Security se systémem Vaadin, což je nutný krok, protože aplikace Vaadin jsou jednostránkové aplikace

a nespouštějí úplnou aktualizaci prohlížeče, když uživatel přechází mezi zobrazeními. Metoda serviceInit poslouchá inicializaci interní kořenové komponenty UI ve Vaadinu a zatím před každým přechodem mezi pohledy přidává posluchač, kde se provolává metoda authenticateNavigation. V metodě authenticateNavigation je zajištěno přesměrování všech požadavků na přihlášení, pokud uživatel není ověřen.

5.1.10 Service

Tento adresář obsahuje komponenty, které jsou označeny anotací @Service. Obsah složky je následující:

```
service
├── AuthorityService.java
├── AuthService.java
├── CrudService.java
├── CustomerService.java
├── EmailNotificationService.java
├── FilterableCrudService.java
├── FinancialService.java
├── FirmService.java
├── FirmSettingsService.java
├── ImportService.java
├── OfferService.java
├── PatternService.java
├── PdfGenerateService.java
└── UserService.java
```

Třídy typu service se používají ke zpracování byznys logiky aplikace a také k přístupu do vrstvy persistence. Služby, jež pracují s vrstvou persistence, implementují rozhraní CrudService, které je uvedeno na výpisu kódu 13.

```
public interface CrudService<T> {

    JpaRepository<T, Long> getRepository();

    default T save(T entity) {
        return getRepository().saveAndFlush(entity);
    }

    default void delete(T entity) {
        if (entity == null) {
            throw new EntityNotFoundException();
        }
        getRepository().delete(entity);
    }

    default void deleteById(long id) {
        delete(load(id));
    }

    default long count() {
        return getRepository().count();
    }

    default T load(long id) {
        return getRepository().findById(id).orElseThrow(() -> new
EntityNotFoundException("Entita " + id + " neexistuje"));
    }

    default List<T> loadAll() {
        return getRepository().findAll();
    }
}
```

Výpis kódu 13: Rozhraní CrudService

Rozhraní CrudService je generická třída, která poskytuje základní operace nad entitami. Typový parametr se nahrazuje konkrétní entitou. Pro filtraci entit na základě Specification slouží rozhraní FilterableCrudService, které také implementuje CrudService. Rozhraní FilterableCrudService může také sloužit k paginaci výsledků a jej obsah je uveden na výpisu kódu 14.

```
public interface FilterableCrudService<T e‟xtends AbstractEntity> extends
CrudService<T> {

    long countAnyMatching(Specification<T> specification);

    Set<T> filter(Specification<T> specifications);

    Set<T> filter(Specification<T> specifications, int offset, int size);
}
```

Výpis kódu 14: Rozhraní FilterableCrudService

Příkladem třídy implementující FilterableCrudService interface je PatternService, viz výpis kódu 15. Třída PatternService obsahuje repositář PatternRepository pro přístup k databázi. Implementace této třídy je uvedena níže, kde metoda save slouží k ukládání položky firmy. Před uložením položky probíhá její kontrola, jestli je výchozí nebo vlastní. V případě, že položku vlastní firma, tak proběhne její uložení do databáze. Pokud je položka výchozí a určitá její hodnota byla změněna, tak se vytvoří kopie této položky, která poté bude uložena do tabulky patterns jako vlastní položka firmy. Stejná výchozí položka bude odstraněna z vazební tabulky firm_patterns. Metoda delete slouží k odstranění položky, kde se nejdříve také zkontroluje stav položky a položka bude odstraněna z vazební tabulky firm_patterns, pokud je výchozí. Jinak se položka považuje za vlastní a bude odstraněna z tabulky patterns.

```
@Service
public class PatternService implements FilterableCrudService<PatternEntity> {
    private final PatternRepository patternRepository;
    private final FirmService firmService;

    public PatternService(PatternRepository patternRepository,
 FirmService firmService) {
        this.patternRepository = patternRepository;
        this.firmService = firmService;
    }

    @Override
    public JpaRepository<PatternEntity, Long> getRepository() {
        return patternRepository;
    }

    @Override
    @Transactional
    @Async
    public PatternEntity save(PatternEntity entity) {
        if (PatternEntity.isDefaultPattern(entity)) {
            var pattern = patternRepository.findPatternById(entity.getId());
            if (!pattern.equals(entity)) {
                var firm = firmService.getFirmWithDefaultPatterns();
                firm.removeDefaultPattern(pattern);
                entity = PatternMapper.convertToEntity(entity, firm);
                entity.getFirmEntity().addPattern(entity);
                if (!pattern.getOfferPatterns().isEmpty()) {
                    var savedPattern = FilterableCrudService.super.save(entity);
```

```
                    pattern.getOfferPatterns().forEach(it ->
it.getId().setPatternId(savedPattern.getId()));
                }
            }
        }
        return FilterableCrudService.super.save(entity);
    }

    @Override
    @Transactional
    public void delete(PatternEntity entity) {
        var pattern = patternRepository.findFullPatternById(entity.getId());
        if (PatternEntity.isDefaultPattern(pattern)) {
            var firm = firmService.getFirmWithDefaultPatterns();
            firm.removeDefaultPattern(pattern);
            firmService.save(firm);
        } else {
            var firm = firmService.getFirmWithCustomPatterns();
            firm.removePattern(pattern);
            FilterableCrudService.super.delete(pattern);
        }
    }
...
}
```
Výpis kódu 15: Implementace třídy PatternService

Pro vytvoření šablony nabídky slouží třída PdfGenerateService, která generuje šablonu pomocí Thymeleaf a potřebná data se předávají pomocí kontextu. Podobný způsob generování šablony se využívá ve třídě EmailNotificationService, která slouží k odesílání e-mailu s OTP uživateli pomocí knihovny JavaMail. HTML kostry těchto šablon jsou uloženy ve složce template.

5.1.11 Utils

Adresář utils obsahuje následující třídy:

```
utils
├── FormattingUtils.java
├── ServiceListener.java
└── UIUtils.java
```

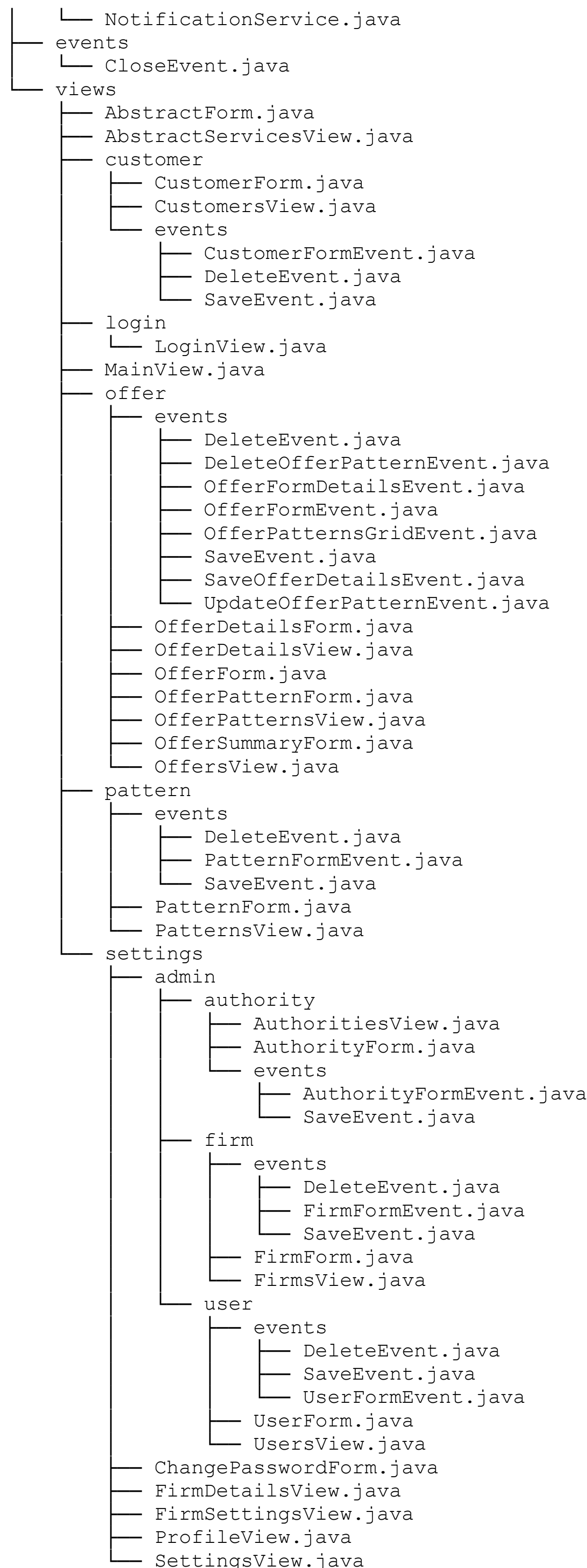
Třída FormattingUtils utils slouží k formátování textu a obsahuje metodu formatAsCurrency, která zajišťuje formátování ceny a přidává označení měny na základě hodnoty Locale.

Třída UIUtils obsahuje metodu createLabel, která vytváří komponentu Label na základě vstupních atributu value a color. Tato metoda se používá u nabídek pro vytvoření barevného textu. Třída ServiceListener implementuje rozhraní VaadinServiceInitListener a přepisuje metodu seviceInit, ve které se přidává vlastní obslužná rutina chyb, které mohou vzniknout za běhu aplikace.

5.1.12 UI

Složka ui obsahuje pohledy uživatelského rozhraní a má následující strukturu:

```
ui
├── components
```



```
│   └── NotificationService.java
├── events
│   └── CloseEvent.java
└── views
    ├── AbstractForm.java
    ├── AbstractServicesView.java
    ├── customer
    │   ├── CustomerForm.java
    │   ├── CustomersView.java
    │   └── events
    │       ├── CustomerFormEvent.java
    │       ├── DeleteEvent.java
    │       └── SaveEvent.java
    ├── login
    │   └── LoginView.java
    ├── MainView.java
    ├── offer
    │   ├── events
    │   │   ├── DeleteEvent.java
    │   │   ├── DeleteOfferPatternEvent.java
    │   │   ├── OfferFormDetailsEvent.java
    │   │   ├── OfferFormEvent.java
    │   │   ├── OfferPatternsGridEvent.java
    │   │   ├── SaveEvent.java
    │   │   ├── SaveOfferDetailsEvent.java
    │   │   └── UpdateOfferPatternEvent.java
    │   ├── OfferDetailsForm.java
    │   ├── OfferDetailsView.java
    │   ├── OfferForm.java
    │   ├── OfferPatternForm.java
    │   ├── OfferPatternsView.java
    │   ├── OfferSummaryForm.java
    │   └── OffersView.java
    ├── pattern
    │   ├── events
    │   │   ├── DeleteEvent.java
    │   │   ├── PatternFormEvent.java
    │   │   └── SaveEvent.java
    │   ├── PatternForm.java
    │   └── PatternsView.java
    └── settings
        ├── admin
        │   ├── authority
        │   │   ├── AuthoritiesView.java
        │   │   ├── AuthorityForm.java
        │   │   └── events
        │   │       ├── AuthorityFormEvent.java
        │   │       └── SaveEvent.java
        │   ├── firm
        │   │   ├── events
        │   │   │   ├── DeleteEvent.java
        │   │   │   ├── FirmFormEvent.java
        │   │   │   └── SaveEvent.java
        │   │   ├── FirmForm.java
        │   │   └── FirmsView.java
        │   └── user
        │       ├── events
        │       │   ├── DeleteEvent.java
        │       │   ├── SaveEvent.java
        │       │   └── UserFormEvent.java
        │       ├── UserForm.java
        │       └── UsersView.java
        ├── ChangePasswordForm.java
        ├── FirmDetailsView.java
        ├── FirmSettingsView.java
        ├── ProfileView.java
        └── SettingsView.java
```

Podsložka components obsahuje komponentu pro zobrazení upozornění v aplikaci. V podsložce events se nachází obecná událost CloseEvent. Události, jež jsou specifické pro každou stránku, jsou uloženy v podsložce events dané stránky. Nejrozsáhlejší podsložkou je views, která obsahuje formy a všechny stránky aplikace.

Hlavní stránkou je MainView, která je potomkem AppLayout a vytváří obecné UI komponenty, jako je záhlaví a navigační menu. Pro vytvoření záhlaví se volí metoda createHeader, kde se přidává H2 komponenta s názvem aplikace a ikona do komponenty HorizontalLayout. Metoda createDrawer slouží k vytvoření levého navigačního menu pomocí komponenty RouterLink, která odkazuje na konkrétní stránku a přesměruje na ni uživatele po kliknutí. Další metoda createUploadButton vytváří tlačítko pro import csv souboru s položkami.

K vytvoření forem a pohledů slouží generické komponenty AbstractForm a AbstractServicesView. Tyto generické třídy obsahují tlačítka, mřížky, bindery a další komponenty, které jsou nutné pro vytvoření formy nebo uživatelského rozhraní. Příklad implementace AbstractForm je uveden na výpisu kódu 16.

```
public abstract class AbstractForm<E extends AbstractEntity> extends Div {
    protected final Binder<E> binder;
    protected final Dialog dialog;
    protected final H2 headline;
    protected final Button saveButton;
    protected final Button cancelButton;
    protected final Button deleteButton;
    private E entity;

    public AbstractForm(Binder<E> validator) {
        this.headline = new H2();
        this.dialog = new Dialog();
        this.binder = validator;
        this.saveButton = new Button("Uložit");
        this.deleteButton = new Button("Odstranit");
        this.cancelButton = new Button("Zrušit");
        this.saveButton.addClickShortcut(Key.KEY_S, KeyModifier.ALT);
        this.cancelButton.addClickShortcut(Key.KEY_C, KeyModifier.ALT);
        this.deleteButton.addClickShortcut(Key.DELETE);
    }

    protected abstract void setBinder();

    protected abstract Component createDialogLayout();

    public void open(String title) {
        headline.setText(title);
        if (getEntity() == null || getEntity().getId() == null)
            getDeleteButton().setVisible(false);
        else
            getDeleteButton().setVisible(true);
        dialog.open();
    }
    public void close() {
        dialog.close();
    }

    public void setEntity(E entity) {
        this.entity = entity;
        binder.readBean(entity);
    }
    protected HorizontalLayout createButtonsLayout() {
        return null;
    }
    protected abstract void validateAndSave();

    public Dialog getDialog() {
        return dialog;
    }
    public E getEntity() {
        return entity;
    }
    public Button getSaveButton() {
```

```
        return saveButton;
    }
    public Button getCancelButton() {
        return cancelButton;
    }
    public Button getDeleteButton() {
        return deleteButton;
    }
    public H2 getHeadline() {
        return headline;
    }
    public <T extends ComponentEvent<?>> Registration addListener(Class<T>
eventType,

ComponentEventListener<T> listener) {
        return getEventBus().addListener(eventType, listener);
    }
```

Výpis kódu 16: Implementace abstraktní třídy pro vytvoření forem

Pro zobrazení seznamu položek slouží třída PatternsView, viz obrázek 10. V tomto rozhraní jsou uvedeny všechny položky firmy. Uživatel může vyhledávat, přidávat, upravovat a odstraňovat položky.

Pohled obsahuje záhlaví, panel nástrojů, navigační menu a list s položkami. Pohled PatternsView má následující vlastnosti:

- Je potomkem komponenty AbstractServicesView.
- Komponenta Grid je parametrizována entitou PatternEntity.
- Konfigurace Grid se provádí v metodě configureGrid pro lepší čitelnost.
- Vytvoření a konfigurace komponenty panel nástrojů se provádí v metodě getToolBar.
- Metoda add přidává panel nástrojů a mřížku do VerticalLayout.
- Metoda configureEvents nastavuje posluchač událostí SaveEvent, DeleteEvent a CloseEvent.
- Aktualizace seznamu položek se provádí pomocí metody updateList.

Pro odstranění, přidávání a editaci položek pohled obsahuje formulář PatternForm. Formulář je znázorněn na obrázku 11 a obsahuje následující prvky:

- TextField pro název a popisek položky;
- NumberField pro dobu trvání a cenu bez DPH;
- Button pro odstranění, uložení a zrušení.

Obrázek 10: Pohled – Položky uživatele

Obrázek 11: Dialog pro editaci položky

PatternForm je potomkem abstraktní třídy AbstractForm. Pro vytvoření formuláře byl použit binder, který sváže pole ve formuláři a atributy datových objektů podle názvu. Binder také podporuje pokročilé rozhraní, kde je možné konfigurovat převody dat a další pravidla ověřování. Pro tuto aplikaci byla tato možnost využita a každý formulář má nastavený vlastní validátor. Příklad nastavení binderu je uveden na výpisu kódu 17.

Vaadin má také systém pro zpracování událostí komponent nebo další možností zpracování událostí mezi komponenty je knihovna EventBus Guava od společnosti Google. Pro tento projekt byla vybrána výchozí možnost Vaadinu. Příklad definování nových událostí znázorňuje výpis kódu 18.

```
    public PatternForm() {
        super(new BeanValidationBinder<>(PatternEntity.class));
        setBinder();
        dialog.add(getHeadline(), createDialogLayout(), createButtonsLayout());}

    @Override
    protected void setBinder() {
        binder.forField(nameField).asRequired("Nazev je povinny")
                .withValidator(
                        name -> name.length() >= 3,
                        "Název musí obsahovat alespoň 3 znaků")
                .bind(PatternEntity::getName, PatternEntity::setName);
        binder.forField(descriptionField)
                .bind(PatternEntity::getDescription, PatternEntity::setDescription);
        binder.forField(durationField).asRequired("Delka práce je povinná")
                .bind(PatternEntity::getDuration, PatternEntity::setDuration);
        binder.forField(priceWithoutVatField).asRequired("Cena je povinná")
                .bind(PatternEntity::getPriceWithoutVAT,
PatternEntity::setPriceWithoutVAT);}
...
```

Výpis kódu 17: Binder pro položky

```
Public static abstract class PatternFormEvent extends ComponentEvent<PatternForm> {
    private PatternEntity item;

    protected PatternFormEvent(PatternForm source, PatternEntity item) {
        super(source, false);
        this.item = item;
    }

    public PatternEntity getItem() {
        return item;
    }
}

public static class SaveEvent extends PatternFormEvent {
    SaveEvent(PatternForm source, PatternEntity contact) {
        super(source, contact);
    }
}

public static class DeleteEvent extends PatternFormEvent {
    DeleteEvent(PatternForm source, PatternEntity contact) {
        super(source, contact);
    }
}
public <T extends ComponentEvent<?>> Registration addListener(Class<T> eventType,

ComponentEventListener<T> listener) {
    return getEventBus().addListener(eventType, listener);
}
```

Výpis kódu 18: Implementace událostí komponenty PatternForm

Pro PatternForm byly definovány události SaveEvent, DeleteEvent a CloseEvent. Metoda addListener používá sběrnici událostí Vaadin k registraci vlastních typů událostí.

Další kus kódu ukazuje příklad nastavení posluchačů událostí při zmáčknutí tlačítek save, delete a cancel, kde tlačítko save vyvolá metodu validateAndSave. Tlačítko delete vyvolá událost odstranění a předá aktivní položku. Tlačítko cancel vyvolá událost uzavření.

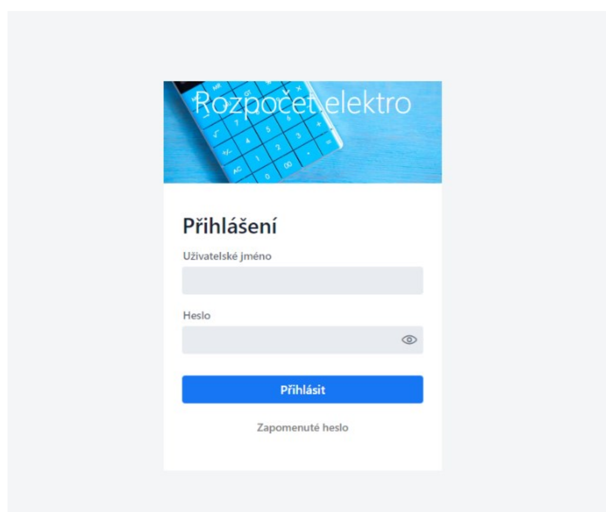
```
private void configureEvents() {
    patternForm.addListener(PatternForm.SaveEvent.class, this::save);
    patternForm.addListener(PatternForm.CloseEvent.class, e -> closeEditor());
    patternForm.addListener(PatternForm.DeleteEvent.class, this::delete);
}
```

Výpis kódu 19: Nastavení posluchačů událostí komponenty PatternForm

Posluchače pro tyto typy událostí nastavuje komponenta PatternsView při inicializaci pomocí metody configureEvents. Příklad nastavení události pro PatternForm je uveden na výpisu kódu 19.
Pohled a forma pro manipulaci se zákazníky mají podobnou strukturu jako položky a jsou umístěny ve složce customer.

Další důležitou stránkou je LoginView (viz obrázek 12), která slouží k autentizaci uživatele a má následující vlastnosti:

- Je potomkem komponenty Div;
- Pro zachycení uživatelského jména a hesla se používá komponenta LoginForm;
- Pro odeslání přihlašovacích údajů do Spring Security, akce LoginForm je nastavena na hodnotu „login";
- LoginView implementuje rozhraní BeforeEnterObserver a pokud se pokus o přihlášení nezdaří, uživatel bude informován o chybě při ověřování;
- Pro obnovení hesla pomocí OTP slouží tlačítko zapomenuté heslo.



Obrázek 12: Stránka pro přihlášení uživatele do aplikace

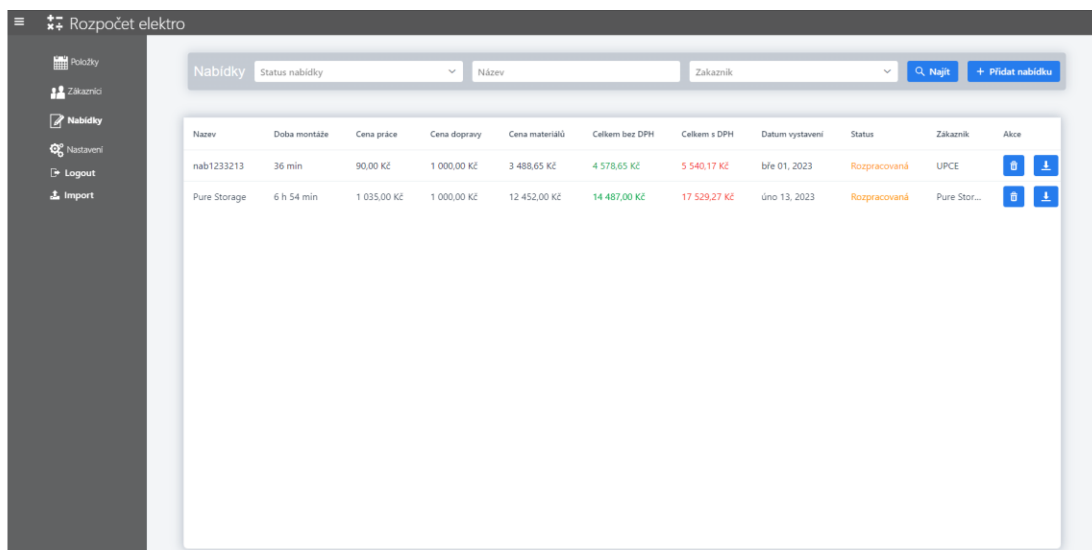
Stránka SettingsView ve složce settings je vytvořena pomocí komponenty Tab a její grafický design je uveden na obrázku 13. Celkem SettingsView obsahuje šest panelů jako ProfileView, FirmSettingsView, FirmDetails, UsersView, AuthoritiesView, FirmsView, kde každý uživatel vidí panely v závislosti na roli v aplikaci. Přístup k posledním třem stránkám má pouze uživatel s rolí admin. Účel a struktura každého panelu je následující:

- ProfileView slouží k editaci osobních údajů a změně hesla.

- FirmSettingsView umožňuje uživateli editovat pracovní dobu, cenu za dopravu, hladinu DPH, výše slevy, cenu za práci a prořez.
- Stránka FirmDetails slouží k editaci základních údajů o firmě a umí vyhledávat a automaticky doplňovat informace o ekonomických subjektech na základě IČO pomocí služeb ARES. Při provolání služeb ARES do URL adresy se přidává parametr IČO.
- Stránka AuthoritiesView a UsersView slouží ke správě rolí a uživatelů.
- Pro správu firem slouží stránka FirmsView. Při přidání nové firmy je možné také použít ARES služby pro vyhledávání informací o ekonomických subjektech v registru MF ČR na základě IČO.

Obrázek 13: Pohled SettingsView

Pro zobrazení nabídek slouží pohled OffersView a pro ilustraci je uveden na obrázku 14. Na stránce jsou umístěny základní informace o nabídce jako název, cena dopravy, cena montáže, status nabídky atd. Panel nástrojů umožňuje uživateli přidávat, vyhledávat nabídky dle názvu, statusu nebo jména zákazníka. Již hotovou nabídku je možné stáhnout kliknutím na tlačítko stáhnout ve sloupci Akce. Šablona hotové nabídky je znázorněna na obrázku 15.



Obrázek 14: Stránka pro zobrazení nabídek

Datum vytvoření: 2023-03-01

Odběratel:
UPCE
Email: upce@upce.cz
Tel.:

Dodavatel
AddAI.Life s.r.o.
Email: demor@upce.cz
Tel.:
Adresa: Rybná 716/24, 11000 Praha, Česká republika
IČO: 08740666
DIČ:

Nabídka elektrikáských prací

№	Název	Množství	Cena (Kč)	Popis	Doba montáže (min)
1	zvonek elektrický vnitřní.	1	220.00		27.0
2	zvonkové tlačítko vč.zapojení.	1	3113.00		9.0

Cena montáže:	90,00 Kč
Cena materiálu:	3 488,65 Kč
Cena dopravy:	1 000,00 Kč
DPH:	21.0 %
Sleva:	0.0 %
Celkem k platbe bez DPH	**4 578,65 Kč**
Celkem k platbe	**5 540,17 Kč**

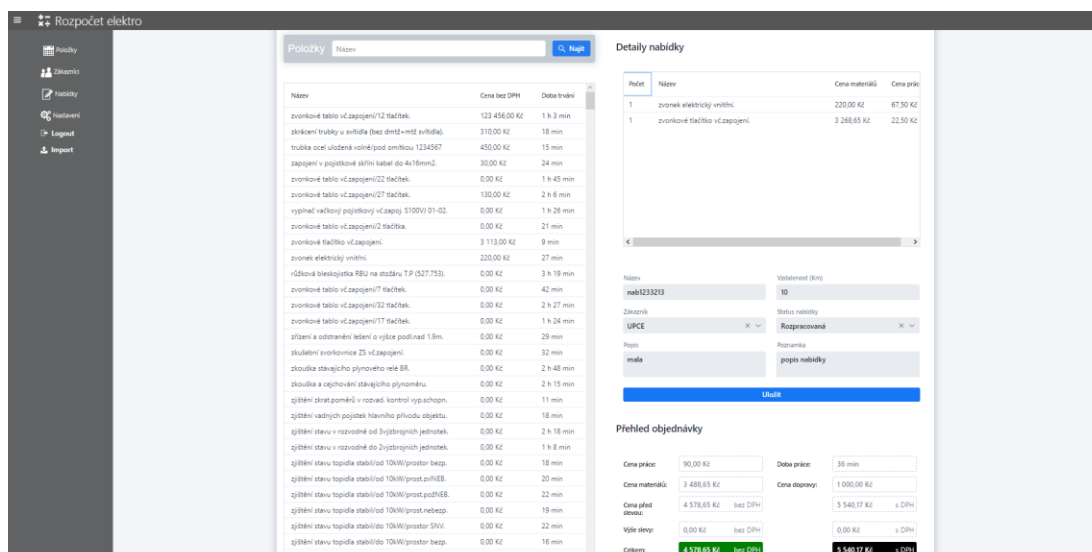
Razitko a podpis: ______________________

Obrázek 15: Šablona nabídky

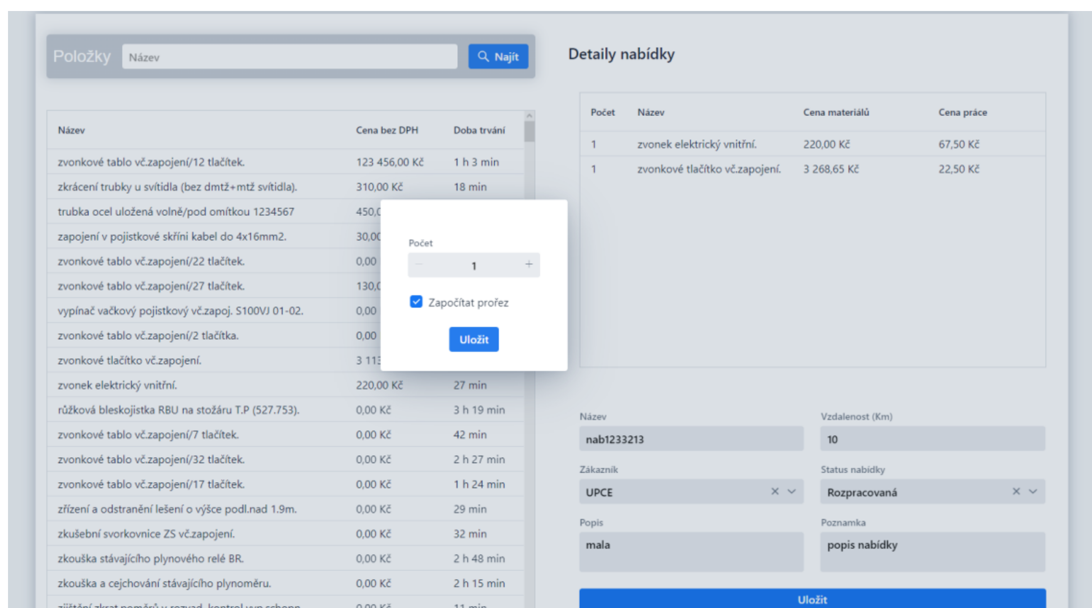
Pro zobrazení detailů nabídky slouží komponenta OfferDetailsView (viz obrázek 16), kde se v levé části nachází stránka s položkami firmy, které uživatel může přidat do nabídky.

Při přidání položky může uživatel nastavit počet položek v nabídce pomocí dialogového okna OfferPatternForm, které je znázorněno na obrázku 17. Dialog OfferPAttern také umožňuje uživateli započítat do ceny prořez. Levá část stránky se skládá z formy OfferPatternForm, která zobrazuje již přidané položky do nabídky. Přidané položky je možné odstranit nebo editovat pravým tlačítkem myši. Komponenta OfferDetailsForm

umožňuje editovat základní údaje o firmě jako název, vzdálenost, zákazník atd. Komponenta OfferSummaryForm slouží ke zobrazení souhrnu, kde je spočítána celková cena nabídky s a bez DPH, výše slevy a další hodnoty.



Obrázek 16: Stránka detailů nabídky



Obrázek 17: Dialogové okno OfferPatternForm pro editaci položek nabídky

5.1.13 Resources

Resources jsou data jako obrázky, text, konfigurační soubory, ke kterým program potřebuje přistupovat způsobem, jenž je nezávislý na umístění programového kódu. Struktura této složky je následující:

```
resources
├── application-dev.yml
├── application.yml
├── db
│   └── migration
│       ├── V001__create_firm.sql
│       ├── V002__create_user.sql
│       ├── V003__create_patterns.sql
│       ├── V004__create_customers.sql
│       ├── V005__create_offers.sql
│       ├── V006__insert_init_data.sql
│       └── V007__create_copy_patterns.sql
├── fonts
│   └── Tahoma.ttf
├── templates
│   ├── notification.html
│   └── offer.html
└── xsd
    └── AresAnswer.xsd
```

Ve složce db/migration se nachází Flyway skripty, které zajišťují jednoduchou migraci databáze. HTML šablony jsou umístěny ve složce templates, kde offer.html slouží k vytvoření šablony nabídky a notification.html jako e-mailová šablona pro resetování hesla. Pro deserializaci odpovědi od ARES API slouží Java třída AresOdpovedi, která byla vygenerována pomocí JAXB pluginu na základě XSD schématu AresAnswer ve složce xsd.

5.2 Kontejnerizace a nasazení

Kontejnerizace softwaru umožňuje nasadit jej do různých prostředí s malými nebo žádnými úpravami. V této práci ke kontejnerizaci aplikace slouží Docker. Vytvořený obraz aplikace pomocí Dockerfile je možné nasadit na Heroku, Openshift a jiné cloudové služby.

5.2.1 Dockerfile

Pro vytvoření obrazu aplikace slouží Dockerfile, viz výpis kódu 20.

```
1.FROM maven:3-openjdk-11-slim as build
2.RUN curl -L https://deb.nodesource.com/setup_14.x | bash -
3.RUN apt-get update -qq && apt-get install -qq --no-install-recommends nodejs

4.RUN useradd -m myuser
5.WORKDIR /usr/src/app/
6.RUN chown myuser:myuser /usr/src/app/
7.USER myuser

8.COPY --chown=myuser pom.xml ./
9.RUN mvn dependency:go-offline -Pproduction

10.COPY --chown=myuser:myuser src src
11.COPY --chown=myuser:myuser frontend frontend
12.COPY --chown=myuser:myuser package.json ./
13.COPY --chown=myuser:myuser package-lock.json* pnpm-lock.yaml* webpack.config.js*
./
14.RUN mvn clean package -DskipTests -Pproduction

15.FROM openjdk:11-jdk-slim
16.COPY --from=build /usr/src/app/target/*.jar /usr/app/app.jar
17.RUN useradd -m myuser
18.USER myuser
19.CMD java  -Dserver.port=$PORT  $JAVA_OPTS -jar /usr/app/app.jar
```

Výpis kódu 20: Dockerfile

O výše uvedeném souboru je třeba poznamenat následující body:

- Řádek musí začínat klíčovým slovem FROM a říká dockeru, ze kterého základního obrazu je potřeba obraz založit. V tomto případě se obraz vytváří z maven pro JDK 11. Za instrukci FROM je přidán parametr AS, který může pojmenovat fázi pro snadné použití v následujících fázích.
- Další dva řádky slouží k instalaci Node.js, který Vaadin používá ve vývojovém režimu (development mode) ke spuštění vývojového serveru Webpack a správce balíčků Node.js (npm) a balíček runner (npx) k načítání, instalaci a spouštění front-endových balíčků.
- Kontejnery Dockeru ve výchozím nastavení běží s oprávněním root a stejně tak i aplikace, která běží uvnitř kontejneru, což je velký problém z hlediska zabezpečení, protože hackeři mohou získat root přístup k hostiteli Docker prolomením aplikace běžící uvnitř kontejneru. Proto se doporučuje spouštět Docker bez root oprávnění a příkaz na řádku 4 přidává uživatele s non-root oprávněním.
- Další příkaz WORKDIR se používá k definování pracovního adresáře kontejneru Docker v daném okamžiku. Pro tento případ je pracovním adresářem /usr/src/app.
- Následující dva příkazy slouží ke změně vlastníka pracovního adresáře a změně uživatele, který byl přidán v předchozím kroku.
- Příkazy COPY a RUN na řádku 9 a 10 zkopíruje pom.xml a předem načte závislosti, aby opakované sestavení mohlo pokračovat od dalšího kroku se stávajícími závislostmi.
- Další příkazy COPY zkopírují všechny potřebné soubory projektu do složek.
- Příkaz na řádku 14 sestaví produkční balíček za předpokladu, že jsme verzi ověřili dříve, takže není potřeba znovu spouštět testy.
- Od řádku 15 je část, která se používá ke spuštění aplikace. Jako základní obraz slouží JDK slim, který obsahuje pouze minimální balíčky potřebné ke spuštění Javy. Parametr –from se přidává za instrukci COPY, který se používá k určení, na které výsledky předchozí fáze se má odkazovat.
- Poslední příkaz CMD v souboru určuje instrukci, která se má provést při spuštění kontejneru Docker. Pro úspěšné nasazení na Heroku byly při spuštění jar souboru přidány proměnné prostředí PORT pro nastavení portu aplikace a JAVA_OPTS pro konfiguraci JVM.

Vytvořený obraz aplikace se poté používá k nasazení na Heroku nebo jiné cloudové platformy. Obraz také zjednodušuje proces spuštění aplikace na lokálně. Ke spuštění

aplikace je potřeba kromě obrazu aplikace mít připravenou MySQL databázi. Jedním z nástrojů pro spuštění více služeb je docker-compose.

5.2.2 Docker-compose

Pro spuštění aplikace a všech služeb, na kterých je tato aplikace závislá, slouží docker-compose.yml soubor, který je uveden na výpisu kódu 21.

```
version: 3.9
services:
  db:
    image: mysql
    environment:
      MYSQL_USER: upce
      MYSQL_PASSWORD: upce
      MYSQL_ROOT_PASSWORD: upce
      MYSQL_DATABASE: upce
    volumes:
      - ./mysql:/var/lib/mysql
    ports:
      - "3308:3308"
    healthcheck:
      test: "/usr/bin/mysql --user=root --password=upce --execute \"SHOW
DATABASES;\""
      interval: 1s
      timeout: 20s
      retries: 10
  electro_calc:
    image: electro_calc
    build:
      context: .
    environment:
      SPRING_DATASOURCE_URL: "jdbc:mysql://db:3308/upce"
      SPRING_DATASOURCE_USERNAME: upce
      SPRING_DATASOURCE_PASSWORD: upce
    ports:
      - "8083:8080"
    depends_on:
      - db:
            condition: service_healthy
```

Výpis kódu 21: Docker-compose

Poznámky k výše uvedenému docker-compose souboru:

- Aplikace se jmenuje electro_calc. Je to druhá ze dvou služeb.
- Obraz aplikace má název electro_calc. Docker vytvoří tento obrázek z Dockerfile v aktuálním adresáři.

  Aplikace používá jako zdroj dat MySQL a název databáze, uživatelské jméno a heslo jsou přidány pomocí proměnných prostředí.
- Kontejner aplikace je závislý na službě db a proto startuje po této službě jen v případě, že bude splněna podmínka service_healthy. Služba db má sekci healthcheck pro kontrolu stavu služby.

5.2.3 Nasazení aplikace

Při nasazení aplikace na cloudovou platformu Heroku je potřeba udělat následující nutné kroky:

- Mít účet na Heroku;
- Mít vytvořenou aplikaci na Heroku přes Heroku CLI nebo GUI;
- Docker;
- Heroku CLI.

Před spuštěním aplikace je také nutné vytvořit instanci MySQL databáze a použít poskytnuté údaje pro napojení aplikace k databázi. Nutné proměnné prostředí pro aplikaci:

- SPRING_DATASOURCE_URL;
- SPRING_DATASOURCE_USERNAME;
- SPRING_DATASOURCE_PASSWORD;
- JAVA_OPTS(-Xmx) – určuje maximální velikost heap.

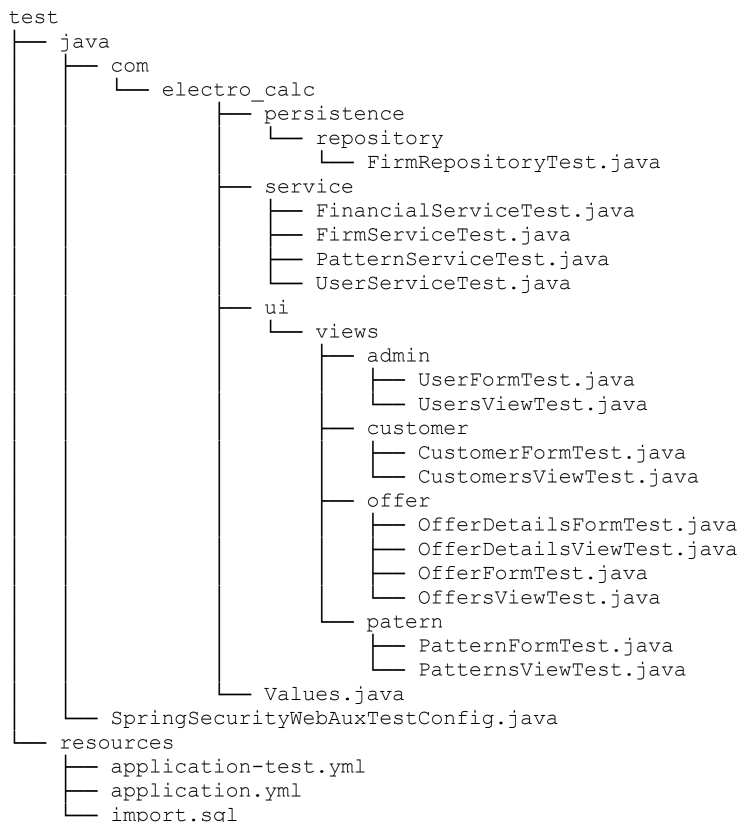
Postup nasazení je následující:

- Přihlásit se ke svému účtu Heroku pomocí příkazu `heroku login` a podle pokynů vytvořit nový veřejný klíč SSH.
- Přihlásit se do registru kontejnerů příkazem `heroku container:login`.
- Provést push obrazu aplikace do registru kontejnerů pomocí příkazu `heroku container:push electro_calc –app electro_calc`.
- Provést vydání aplikace příkazem `heroku container:release electro_calc`.

V případě potřeby je možné obraz aplikace nahrát i do jiného registru, jako je Docker hub. Obraz této aplikace byl také nahrán do registru Docker hub, repositář st55409/electro_calc.

6 Testování

Testování je důležitý proces při psaní softwaru. Běžným osvědčeným postupem je testovat co nejméně kódu v jednom testu. Tímto způsobem, když se něco pokazí, selžou pouze relevantní testy. Existují různé metodiky vývoje softwaru, které jsou založeny na Test Driven Development, kde se nejdříve píšou testy a až poté probíhá implementace kódu. V této práci byl použit klasický způsob psaní testu po implementaci funkcionality v kódu. Pro testování uživatelského rozhraní byly napsány jednotkové a integrační testy. Všechny testy se nacházejí ve složce test. Struktura složky je následující:



```
test
├── java
│   ├── com
│   │   └── electro_calc
│   │           ├── persistence
│   │           │   └── repository
│   │           │       └── FirmRepositoryTest.java
│   │           ├── service
│   │           │   ├── FinancialServiceTest.java
│   │           │   ├── FirmServiceTest.java
│   │           │   ├── PatternServiceTest.java
│   │           │   └── UserServiceTest.java
│   │           ├── ui
│   │           │   └── views
│   │           │       ├── admin
│   │           │       │   ├── UserFormTest.java
│   │           │       │   └── UsersViewTest.java
│   │           │       ├── customer
│   │           │       │   ├── CustomerFormTest.java
│   │           │       │   └── CustomersViewTest.java
│   │           │       ├── offer
│   │           │       │   ├── OfferDetailsFormTest.java
│   │           │       │   ├── OfferDetailsViewTest.java
│   │           │       │   ├── OfferFormTest.java
│   │           │       │   └── OffersViewTest.java
│   │           │       └── patern
│   │           │           ├── PatternFormTest.java
│   │           │           └── PatternsViewTest.java
│   │           └── Values.java
│   └── SpringSecurityWebAuxTestConfig.java
└── resources
    ├── application-test.yml
    ├── application.yml
    └── import.sql
```

Pro spuštění integračních testů je potřeba mít spuštěnou databázi MySQL.

6.1 Persistence

Složka persistence obsahuje testy pro vrstvu persistence. Její struktura je následující:

```
├── persistence
│   └── repository
│       └── FirmRepositoryTest.java
```

Jelikož většina tříd vrstvy persistence jsou velmi jednoduché, jejich testování je nutné jen v případě, kdy se provádí složité SQL dotazy. Příkladem podobného testu je třída FirmRepositoryTest, viz výpis kódu 22.

```
@DataJpaTest(excludeAutoConfiguration = EmbeddedDatabase.class)
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class FirmRepositoryTest {

    @Autowired
    private FirmRepository firmRepository;

    @Test
    void shouldReturnFirmDefaultPatterns() {
        final long firmId = 1L;

        var result = firmRepository.findWithDefaultPatternsById(firmId);

        assertTrue(result.getDefaultPatterns().size() > 0);
        assertTrue(result.getDefaultPatterns().stream().allMatch(it ->
it.getFirmEntity() == null));
    }

    @Test
    void shouldReturnFirmWithCustomPatterns() {
        final long firmId = 1L;

        var result = firmRepository.findWithCustomPatternsById(firmId);

        assertTrue(result.getDefaultPatterns().size() > 0);
        assertTrue(result.getPatterns().stream().allMatch(it ->
it.getFirmEntity().getId() == firmId));
    }
}
```

Výpis kódu 22: Integrační test JPA repositáře FirmRepository

Pro spuštění integračních testů pro vrstvu persistence slouží anotace @DataJpaTest. Ve výchozím nastavení se rollback transakce provádí automaticky po spuštění testovací metody. Anotace @AutoConfigureTestDatabase umožňuje nahradit výchozí H2 databázi a pro tento případ je to hodnota NONE. Metoda shouldReturnFirmDefaultPatterns slouží k ověření, že firmRepository vrátí entitu firma s výchozími položkami, které se ukládají ve vazební tabulce firm_pattern. Další metoda shouldReturnFirmCustomPatterns testuje, že bude vracena firma s vlastními položkami.

6.2 Service

```
service
├── FinancialServiceTest.java
├── FirmServiceTest.java
├── PatternServiceTest.java
└── UserServiceTest.java
```

Složka service obsahuje integrační testy a slouží k testování komunikace mezi vrstvou persistence a vrstvou obsahující byznys logiku. Struktura složky je uvedena výše. Příklad je uveden na výpisu kódu 23.

```
@SpringBootTest
@Transactional
class FirmServiceTest {

    @Autowired
    private FirmService firmService;

    @Autowired
    private PatternService patternService;
```

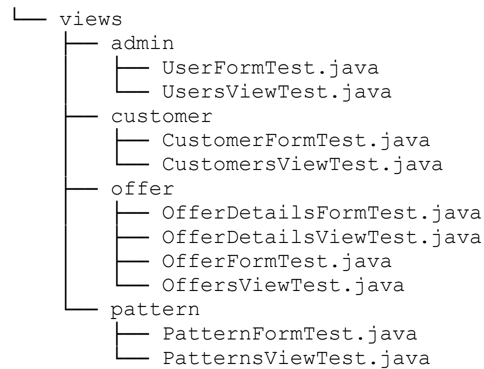
```
    @Test
    @WithMockUser(username = TESTER)
    void shouldReturnFirmDefaultPatterns() {
        var firm = firmService.getFirmWithDefaultPatterns();

        assertEquals(4, firm.getDefaultPatterns().size());
    }

    @Test
    @WithMockUser(username = TESTER)
    void shouldRemoveFirmDefaultPattern() {
        var firm = firmService.getFirmWithDefaultPatterns();
        var pattern = patternService.load(5083);

        assertEquals(4, firm.getDefaultPatterns().size());
        assertTrue(firm.getDefaultPatterns().contains(pattern));
        firmService.deleteFirmWithDefaultPattern(pattern);

        assertEquals(3, firm.getDefaultPatterns().size());
        assertFalse(firm.getDefaultPatterns().contains(pattern));
        pattern = patternService.load(5083);

        assertNotNull(pattern);
    }

    @Test
    @WithMockUser(username = TESTER)
    void shouldReturnFirmCustomPatterns() {
        var patterns =   firmService.getFirmWithCustomPatterns().getPatterns();

        assertEquals(7, patterns.size());
    }

    @Test
    @WithMockUser(username = TESTER)
    void shouldRemoveFirmCustomerPattern() {
        var pattern = patternService.load(5097);
        firmService.deleteFirmWithCustomPattern(pattern);
        var patterns = firmService.getFirmWithCustomPatterns().getPatterns();

        assertEquals(6, patterns.size());
        assertThrows(EntityNotFoundException.class, () -> {
            patternService.load(5097);
        });
    }

    @Test
    @WithMockUser(username = TESTER)
    void shouldCopyDefaultPatterns() {
        var pattern = patternService.load(5196);
        var firm = firmService.load(1);

        assertFalse(firm.getPatterns().contains(pattern));
        firmService.copyDefaultPatterns(firm.getId());

        assertTrue(firm.getPatterns().contains(pattern));
    }
}
```

Výpis kódu 23: Integrační test služby FirmService

Třída FirmServiceTest slouží k testování správného chování aplikace při manipulaci s položkami. Například metoda shouldRemoveFirmDefaultPattern testuje případ, kdy při odstranění výchozí položky s id 5083 z vazební tabulky firm_pattern položka zůstane v tabulce patterns, protože je výchozí. Při spuštění integračních testů je potřeba, aby uživatel byl autentizován v systému a k tomuto účelu slouží anotace @WithMockUser, která vytvoří objekt typu UsernamePasswordAuthenticationToken a přidá jej do SecurityContext. Data pro integrační testy se nachází v souboru import.sql, které se přidávají při vytvoření

aplikačního kontextu. Je důležité provádět rollback všech změn provedených v databázi po ukončení testu, proto byla nad třídou přidána anotace @Transactional, která zruší veškeré změny v databázi po ukončení testu.

6.3 Views



V této složce se nachází jednotkové a integrační testy k ověření správného chování UI komponent. Pro ověření, že se pole formulářů vyplňují správně při provedení různých akcí nad určitým beanem, byly napsány jednotkové testy. Příklad takového testu je uveden na výpisu kódu 24.

```
class PatternFormTest {
    private final String name = "PATTERN";
    private final String description = "PATTERN_DESCRIPTION";
    private final Double duration = 0.00;
    private final BigDecimal priceWithoutVAT = BigDecimal.valueOf(100);

    @Test
    void formFieldsPopulated() {
        PatternForm patternForm = new PatternForm();
        PatternEntity patternEntity = getPattern();
        patternForm.setEntity(patternEntity);

        assertEquals(name, patternForm.getNameField().getValue());
        assertEquals(description, patternForm.getDescriptionField().getValue());
        assertEquals(duration, patternForm.getDurationField().getValue());
        assertEquals(priceWithoutVAT,
patternForm.getPriceWithoutVatField().getValue());
    }

    @Test
    void saveEventHasCorrectValues() {
        PatternForm patternForm = new PatternForm();
        PatternEntity pattern = new PatternEntity();
        patternForm.setEntity(pattern);
        patternForm.getNameField().setValue(name);
        patternForm.getDescriptionField().setValue(description);
        patternForm.getDurationField().setValue(duration);
        patternForm.getPriceWithoutVatField().setValue(priceWithoutVAT);

        AtomicReference<PatternEntity> savedContactRef = new
AtomicReference<>(null);
        patternForm.addListener(SaveEvent.class, e -> {
            savedContactRef.set(e.getItem());
        });
        patternForm.getSaveButton().click();

        PatternEntity savedPattern = savedContactRef.get();

        assertEquals(name, savedPattern.getName());
        assertEquals(description, savedPattern.getDescription());
        assertEquals(duration, savedPattern.getDuration());
```

```
        assertEquals(priceWithoutVAT, savedPattern.getPriceWithoutVAT());
    }
...
```

Výpis kódu 24: Jednotkový test komponenty PatternForm

Třída PatternFormTest slouží k ověření správného chování formy pro položky. Metoda formFieldsPopulated ověřuje, že jsou pole správně vyplněna. Nejprve probíhá inicializace formuláře a poté nastavení položky. Pro kontrolu výsledku slouží metoda JUnit assertEquals. Metoda saveEventHasCorrectValues má podobnou logiku, kde se nejdříve inicializuje prázdný formulář. Poté probíhá vyplnění hodnot do formuláře. Pro zachycení uložené položky po zmáčknutí tlačítka uložit slouží třída AtomicReference. Pro kontrolu vyplněných údajů slouží JUnit metoda assertEquals.

Pro testování UI tříd, které používají anotaci @Autowire, databázi nebo jakoukoli jinou funkci poskytovanou Spring Boot, jsou použity integrační testy. Příkladem podobného testu je PatternsViewTest. Je uveden na výpisu kódu 25.

```
@SpringBootTest
@Transactional
class PatternsViewTest {
    private final UI ui = new UI();
    @Autowired
    private PatternService patternService;
    private PatternsView patternsView;
    @Autowired
    private UserService userService;

    @BeforeEach
    void init() {
        UI.setCurrent(ui);
        this.patternsView = new PatternsView(patternService, userService, null);
    }

    @Test
    @WithMockUser(username = TESTER)
    void formShownWhenPatternSelected() {
        Grid<PatternEntity> grid = patternsView.getItems();
        PatternEntity patternEntity = getFirstPattern(grid);

        PatternForm form = patternsView.getPatternForm();

        assertFalse(patternsView.getPatternForm().getDialog().isOpened());
        grid.asSingleSelect().setValue(patternEntity);

        assertTrue(patternsView.getPatternForm().getDialog().isOpened());
        assertEquals(patternEntity.getName(), form.getNameField().getValue());
    }

    @Test
    @WithMockUser(username = TESTER)
    void formShownWhenAddPatternClicked() {
        PatternForm form = patternsView.getPatternForm();

        assertFalse(patternsView.getPatternForm().getDialog().isOpened());
        patternsView.getAddButton().click();

        assertTrue(patternsView.getPatternForm().getDialog().isOpened());
        assertNotNull(form.getEntity());
        assertEquals("", form.getNameField().getValue());
        assertEquals("", form.getDescriptionField().getValue());
        assertEquals(0, form.getDurationField().getValue());
        assertEquals(BigDecimal.ZERO, form.getPriceWithoutVatField().getValue());
```

```
    }
...
```

Výpis kódu 25: Integrační test komponenty PatternsView

Metoda formShownWhenPatternSelected slouží ke kontrole, kde uživatel vybere položku a proběhne ověření, že se zobrazí formulář s vybranou položkou. Podobnou logiku má metoda formShownWhenAddPatternClicked, kde se ověří, že při kliknutí na tlačítko přidat položku se otevře prázdný formulář. Před spuštěním každého testu je potřeba provést inicializaci a nastavení aktuální stránky pomocí metody UI.setCurrent.

Závěr

Cílem této práce bylo provést návrh a implementaci webové aplikace pro tvorbu rozpočtu. Aplikace by měla podporovat přihlášení více uživatelů, kdy je přístup ke zdrojům zabezpečen pomocí rolí. Také by aplikace měla umožňovat uživateli aktualizovat zapomenuté heslo pomocí jednorázového hesla, importovat a spravovat položky, nabídky, zákazníky a uživatele. Zároveň by aplikace měla podporovat stažení hotové nabídky v PDF formátu. Aplikace by měla poskytovat UI komponenty, jež umožní uživateli jednoduché a komfortní ovládání.

Nejdříve byla provedena analýza již existujících řešení, na základě čehož vznikl přehled o tom, jak by se výsledná aplikace měla chovat. Také při této analýze vznikly první návrhy uživatelského rozhraní. Dále byly sepsány funkční a nefunkční požadavky a případy užití, což umožnilo lépe pochopit účel aplikace a stanovit hranice funkcionalit, které má poskytovat uživateli. Poté byl na základě sepsaných požadavků vytvořen doménový model aplikace, který dává první pohled na entity v systému a vztahy mezi nimi. Po dokončení sestavení doménového modelu byl vytvořen databázový model. Následně byla vybrána architektura aplikace, což je v tomto případě monolit. Proběhlo rozdělení aplikace na UI, Service a Data access vrstvy, kde UI komponenty komunikují s vrstvou služeb, která odděluje vrstvu uživatelského rozhraní od vrstvy byznys logiky. Vrstva Service je pak kontaktním bodem pro vrstvu UI a Data access, která komunikuje s databází. Také byla do aplikace přidána možnost komunikace s ARES API pro získání informace o ekonomickém subjektu na základě jeho IČO.

Při vývoji aplikace byl použit Spring a Vaadin framework. Zvoleným programovacím jazykem byla Java. Také byly napsány unit a integrační testy, které by měly kontrolovat funkčnost aplikace, zjednodušit její udržování a další vývoj. Pro kontejnerizaci, jednoduché rozběhnutí a nasazení aplikace na různé cloudové platformy byl vytvořen Dockerfile a docker-compose. Obraz aplikace byl nasazen na Heroku a také přidán do registru Docker hub. Aplikaci je možné dále rozvíjet a přidávat další funkcionality nebo zlepšovat již existující.

Použitá literatura

1. CARNELL, John. *Spring microservices in action: a multiplatform approach to building chatbots*. Shelter, Island, NY: Manning Publications Co., 2017. ISBN 16-172-9398-9.
2. CARNELL, John. *Beginning spring boot 2: applications and microservices with the spring framework*. New York, NY: Springer Science Business Media, 2017. ISBN 978-148-4229-309.
3. ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací.* Computer Press. Brno: Computer Press, 2011. ISBN 978-80-251-1503-9.
4. WALLS, Craig. *Spring in Action*. 4rd ed. New York: Manning Publications Co, 2019. ISBN 9781617294945.
5. MARTINŮ, Jiří a Petr ČERMÁK. Metodiky vývoje software [online]. [cit. 2019-08-21]. Dostupné z: http://mvso.cz/wp-content/uploads/2018/02/Metodiky-v%C3%BDvojesoftware-studijn%C3%AD-text.pdf
6. KHARENKO, Anton. Monolithic vs. Microservices Architecture [online]. [cit. 2019-08-21]. Dostupné z: https://articles.microservices.com/monolithic-vs-microservicesarchitecture-5c4848858f59
7. HARRIS, CHANDLER. Microservices vs. monolithic architecture. *Atlassian* [online]. [cit. 2022-07-20]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith
8. *Spring Framework* [online]. 2022 [cit. 2022-07-20]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/reference/html/index.html
9. *Hibernate* [online]. [cit. 2022-07-20]. Dostupné z: https://hibernate.org/orm/documentation/6.1/
10. *Spring Security* [online]. 2022 [cit. 2022-07-20]. Dostupné z: https://docs.spring.io/spring-security/reference/index.html
11. *Vaadin* [online]. 2022 [cit. 2022-07-20]. Dostupné z: https://vaadin.com/docs/v14/
12. *Docker* [online]. 2022 [cit. 2022-07-20]. Dostupné z: https://docs.docker.com/
13. *Heroku* [online]. 2022 [cit. 2022-07-20]. Dostupné z: https://devcenter.heroku.com/categories/reference
14. *Serverless Architecture Overview* [online]. [cit. 2022-07-20]. Dostupné z: https://www.datadoghq.com/knowledge-center/serverless-architecture/
15. ALERT, Talor. Building Progressive Web Apps [online]. Sebastopol: O'Reilly Media, 2017 [cit. 2023-03-28]. ISBN 9781491961650. Dostupné z: https://www.oreilly.com/library/view/building-progressive-web/9781491961643/
16. MIKOWSKI, MICHAEL a JOSH POWELL. *Single Page Web Applications* [online]. Shelter Island: Manning Publications Co., 2014 [cit. 2023-03-28]. ISBN 9781617290756. Dostupné z: http://bedford-computing.co.uk/learning/wp-content/uploads/2016/07/Single-Page-Web-Application-1.pdf
17. *Free purhcase order software* [online]. [cit. 2023-03-28]. Dostupné z: https://free-procurement.com
18. *INSIO DMS* [online]. [cit. 2023-03-28]. Dostupné z: https://insio.cz/aplikace/dms
19. *Kissflow* [online]. [cit. 2023-03-28]. Dostupné z: https://kissflow.com/
20. *Nabidky plus* [online]. [cit. 2023-03-28]. Dostupné z: https://nabidkyplus.cz/

21. RICHARDS, Mark. *Software Architecture Patterns* [online]. 2. vyd. O'Reilly Media, 2022 [cit. 2023-02-20]. ISBN 978-1-098-13427-3. Dostupné z: https://www.oreilly.com/library/view/software-architecture-patterns/9781098134280/