VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Michal Kowalski

Bakalářská práce

Vedoucí práce: doc. Mgr. Miloš Kudělka, Ph.D.

Ostrava, 2022

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Kowalski**

Studijní program: B0613A140014 Informatika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Tieto Czech s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Mgr. Miloš Kudělka, Ph.D.**

Konzultant bakalářské práce: Mgr. Patrik Kolenovský

Datum zadání: 01.09.2021

Datum odevzdání: 30.04.2022

doc. Ing. Petr Gajdoš, Ph.D.	prof. Ing. Jan Platoš, Ph.D.
vedoucí katedry	*děkan fakulty*

Abstrakt

V této bakalářské práci popisuji průběh při absolvování individuální odborné praxe ve firmě Tieto Czech s.r.o., ve které jsem pracoval na pozici Junior Java Developer. Na úvod shrnu zadání projektu a použité technologie. V hlavní části práce popisuji postup při „fullstack“ vývoji aplikace pro správu poznámek, který sloužil primárně pro naučení se technologií, ve kterých firma pracuje. V závěru zmíním na čem právě ve firmě pracuju a shrnu získané zkušenosti v průběhu praxe.

Klíčová slova

typografie; LaTeX; diplomová práce, Tieto Czech s.r.o., PostgreSQL, Java, React, Spring, TypeScript

Abstract

In this Bachelor thesis, I describe the course of individual professional practise in the company Tieto Czech s.r.o., in which I worked in the position of Junior Java Developer. I will summarize the project assignment and the technology I used. In the main part of the thesis, I describe process of “fullstack” development of notes management application, which has served primarily for learning the technology in which the company is working. In the end, I mention what I am working on right now and summarize the experience gained during the practice.

Keywords

typography; LaTeX; master thesis, Tieto Czech s.r.o., PostgreSQL, Java, React, Spring, TypeScript

Poděkování

Rád bych na tomto místě poděkoval firmě Tieto Czech spol. s.r.o. za možnost absolvovat individuální odbornou praxi v rámci stáže. Dále bych rád poděkoval Miroslavovi Zemanovi a Vojtěchovi Pustowkovi za trpělivost při dohlížení nad mojí prací. A v neposlední řadě Matúšovi Mlíchovi a ostatním stážistům v týmu za pomoc v nesnázích.

Obsah

Seznam použitých zkratek a symbolů

NCP	–	Newcomers Project
MVCC	–	Multiversion Concurrency Control
REST	–	Representational State Transfer
API	–	Application Programming Interface
DTO	–	Data Tranfer Object
JSP	–	Java Server Pages
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
JPQL	–	Java Persistence Query Language
DOM	–	Document Object Model
JRE	–	Java Runtime Environment
IoC	–	Inversion of Control
UI	–	User Interface
UX	–	User Experience
SPA	–	Single Page Application
IDE	–	Integrated Development Environment

Seznam obrázků

Seznam výpisů zdrojového kódu

Kapitola 1

Úvod

V této bakalářské práci popisuju můj celý postup při vykonávání odborné praxe ve firmě Tieto Czech s.r.o. Stáž ve firmě jsem si našel sám, a poté jsem požádal o možnost spojit stáž s absolvováním individuální odborné praxe. Mým hlavním motivem bylo získání praktických zkušeností, seznámení se jak to funguje v realné firmě a poznání nových přátel. Do práce jsem nastupoval na pozici Junior Java Developer. Náplní mé práce mělo být vyvíjení backendu webové aplikace v Javě. Posléze jsem se dozvěděl, že se budu moci i naučit vyvíjet frontendovou část webové aplikace v Reactu, což byla pro mě tehdy „španělská vesnice“. Vývoj v reactu jsem si ale zamiloval a mám v plánu se mu více věnovat.

Jako první projekt, na kterém jsem začal pracovat, byl NCP (Newcomers project), o kterém zde budu primárně psát, a na kterém jsem pracoval po téměř celou dobu praxe. Jde o projekt, na kterém se nově příchozí stážisti učí full stack vývoj webové aplikace pro potencionálního zákazníka. Dostal jsem zadání, postup v krocích a zdroje odkud čerpat základní informace a mohl jsem začít.

V druhé kapitole představím firmu TietoEVRY. Ve třetí uvedu zadání od „zákazníka“ a použité technologie. V hlavní části se pak budu věnovat mému postupu při vývoji. Závěrem pak popíšu zkušenosti z přechodu z NCP na reálné projekty.

Kapitola 2

Představení firmy TietoEVRY

Tietoevry je největší severoevropský dodavatel IT služeb poskytující komplexní služby v oblasti IT pro soukromý i veřejný sektor a služby pro vývoj produktů v oblasti komunikací a moderních technologií. Společnost sídlí v Helsinkách. Zaměstnává přes 24 000 lidí a zákazníky má ve více než 90 zemích. S jejími akciemi se obchoduje na burze NASDAQ OMX v Helsinkách a Stockholmu.

Historie Tieto korporace sahá do roku 1968. Tehdy firma nesla název Tietotehdas Oy. V roce 1999, kdy se spojila finská společnost Tieto a švédská společnost Enator, došlo ke změně jména na TietoEnator a v roce 2010 opět na Tieto.

V roce 2019 došlo ke spojení se společností EVRY, která měla stejně jako Tieto významný podíl na skandinávských trzích v oblasti digiálních služeb. Tak vznikla nová společnost Tietoevry. Nyní má celkové příjmy v objemu tři miliardy eur [1].

V Českе republice záhájilo svoji činnost v roce 2004 v Ostravě s 54 zaměstnanci. Další pobočka vznikla v Brně. Dnes má v Česku přes 2600 zaměstnanců a počet stále roste.

2.1 Pracovní zařazení

Ve firmě jsem pracoval v kolektivu přibližně desíti lidí. Byl jsem součásti týmu, který se skládal převážně ze studentů na stáži, dále nás organizoval a zadával úkoly manažer a pokud jsme potřebovali radu, mohli jsme s problémem zajít za jedním ze dvou senior developerů. Jeden se zaměřením na backend a druhý na frontend.

Našim úkolem je vývoj a správa interních aplikací. Nepracujeme jen na jednom projektu, ale jsme rozprostřeni mezi vícero. Někteří pracuji sami, nicméně si každý může říct o pomoc.

Každý týden v pátek jsme měli všichni z týmu schůzku a shrnuli jsme, čeho jsme za ten týden dosáhli, sdíleli nabrané zkušenosti a radili se o dalším postupu.

Kapitola 3

Zadání projektu Správa poznámek

3.1 Požadavky zákazníka

Na začátku si musím projít požadavky „zákazníka“. Hlavní požadavek od zákazníka byl vytvořit webovou aplikaci pro uživatelé, do které by mohli zachytit inspiraci, když udeří. Tato aplikace by měla poskytnout nejrychlejší cestu k zapsání nápadů, myšlenek a seznamu úkolů bez ztráty soustředění. Zároveň bude možnost sdružovat poznámky ve skupinách a přidávat jim tagy. Součástí bude i možnost přihlášení. Další konkrétní specifikace aplikace od zákazníka jsou:

1. Jako uživatel
   (a) mohu vytvořit, zobrazit, upravit a smazat moje vlastní poznámky.
   (b) mohu vytvořit, zobrazit, upravit a smazat moje vlastní skupiny.
   (c) mohu přiřadit nebo odebrat poznámku do skupiny.
   (d) mohu přiřadit tag k poznámce.
   (e) mohu přiřadit tag ke skupině.
2. Jako admin
   (a) mohu vytvořit, zobrazit, upravit a smazat uživatele.

Ve výsledku věcí, na které musí vývojář myslet, je mnohem více, které ale běžný zákazník nevidí.

3.2 Použité technologie

Během stáže jsem se setkal s mnoha technologiemi. Většinu jsem neznal a musel se s ní naučit pracovat.

3.2.1 GitLab

GitLab je open source repozitář a platforma pro spolupráci na vývoji velkých DevOps a DevSecOps projektů. GitLab nabízí online úložiště kódu, možnost pro sledování chyb a CI/CD. Repositář umožňuje hostování různých vývojových řetězců a verzí, také umožňuje uživatelům zkontrolovat předchozí kód a vrátit se k němu v případě nepředvídaných problémů. GitLab poskytuje end-to-end DevOps funkce, a také funkce pro každou fázi životního cyklu vývoje softwaru. GitLab umožňuje vývojovým týmům automatizovat vytváření a testování jejich kódu. GitLab podporuje veřejné i soukromé vývojové větve a je pro jednotlivce zdarma. Naproti tomu někteří konkurenti, jako je GitHub, účtují poplatky za soukromá úložiště, zatímco jiní, jako je Bitbucket, účtují poplatky za další uživatelé nad rámec pěti povolených zdarma na soukromém repositáři [2].

3.2.2 PostgreSQL

PostgreSQL je výkonný, open source objektově relační databázový systém. Má více než patnáct let aktivního vývojové a osvědčenou architekturu, která mu vynesla silnou reputaci pro spolehlivost, integritu dat a správnost. PostgreSQL běží na všech hlavních operačních systémech včetně Linuxu, UNIX a Windows. Podporuje text, obrázky, zvuky a videa a zahrnuje programovací rozhraní pro C/C++, Java, Perl, Python, Ruby, Tcl a Open Database Connectivity [3].

PostgreSQL podporuje velkou část standardu SQL a nabízí mnoho moderních funkcí včetně následujících [3]:

1. složité SQL dotazy
2. SQL sub-select
3. cizí klíče
4. triggery
5. pohledy
6. transakce
7. MVCC
8. čtení replik
9. Hot Standby

Dále podporuje čtyři standardní procedurální jazyky PL/pgSQL, PL/Tcl, PL/Perl a PL/Python.

3.2.3 Java

Java je moderní objektově orientovaný programovací jazyk. Java byla vytvořena v Sun Microsystems, Inc., kde James Gosling vedl tým výzkumníků ve snaze vytvořit nový jazyk, který by elektronickým zařízením umožnil vzájemnou komunikaci.

Rozdíl mezi způsobem fungování Javy a jiných programovacích jazyků byl revoluční. Kód v jiných jazycích je nejprve překladačem přeložen do instrukcí pro konkrétní typ počítače. Kompilátor Java místo toho změní kód za Bytecode, který je pak interpretován softwarem zvaným Java Runtime Environment (JRE). JRE funguje jako virtuální počítač, který interpretuje Bytecode a překládá jej pro hostitelský počítač. Z tohoto důvodu může být Java kód napsán stejným způsobem pro mnoho platforem, což přispělo k jeho popularitě pro použití na internetu, kde mnoho různých typů počítačů může načíst stejnou webovou stránku [4].

3.2.4 Spring boot

Spring Boot je open source framework založený na Javě, který se používá k vytvoření mikro služby. Je vyvinut společností Pivotal Team a používá se k vytváření samostatných aplikací a aplikací připravených k produkci. Spring Boot poskytuje vývojářům dobrou platformu pro vývoj aplikace, kterou se může jednoduše spustit. Může se začít s minimálními konfiguracemi, bez potřeby celé konfigurace Spring [5].

Důležitou částí Spring bootu jsou tzv. Spring Bean. Je to object, který je vytvořen, spravován a odstraněn v Spring kontajneru.

Cíle Spring Bootu [5]:

1. Vyhnout se složité konfiguraci XML.
2. Vyvíjet produkčně připravené aplikace jednodušším způsobem.
3. Zkrátit dobu vývoje a spustit aplikaci samostatně.
4. Nabízí jednodušší způsob, jak začít s vývojem aplikací.

3.2.5 Postman

Postman je aplikace používaná pro testování API. Jedná se o HTTP klienta, který testuje HTTP požadavky s využitím grafického uživatelského rozhraní, jehož prostřednictvím získáváme různé typy odpovědí, které je potřeba následně validovat [6].

3.2.6 HTML

HyperText Markup Language je sada značkovacích symbolů nebo kódů vložených do souboru určeného k zobrazení na internetu. Označení říká webovým prohlížečům, jak zobrazit slova a obrázky na webové stránce [7].

Každý jednotlivý značkovací kód (který by spadal mezi znaky „<“ a „>“) se označuje jako element, ačkoli mnoho lidí jej také označuje jako tag. Některé prvky přicházejí v párech, které indikují, kdy má nějaký efekt zobrazení začít a kdy má skončit [7].

3.2.7 CSS

CSS se stará o vzhled a dojem z webové stránky. Pomocí CSS můžete ovládat barvu textu, styl textu, mezery mezi odstavci, velikost a uspořádání sloupců, jaké obrázky nebo barvy na pozadí se používají, návrhy rozvržení, variace zobrazení pro různá zařízení a velikosti obrazovky. Stejně jako řada dalších efektů [8].

3.2.8 JavaScript

JavaScript je programovací jazyk běžně používaný při vývoji webových aplikací. Původně byl vyvinut společností Netscape jako prostředek k přidávání dynamických a interaktivních prvků na webové stránky [9].

JavaScript funguje na straně klienta, což znamená, že zdrojový kód zpracovává webový prohlížeč klienta, nikoli webový server. To znamená, že funkce JavaScriptu mohou běžet po načtení webové stránky bez komunikace se serverem. Například funkce JavaScriptu může zkontrolovat webový formulář před jeho odesláním, aby se ujistil, že jsou vyplněna všechna požadovaná pole. Kód JavaScriptu může vygenerovat chybovou zprávu předtím, než jsou jakékoli informace skutečně přeneseny na server [9].

3.2.9 TypeScript

TypeScript je open-source programovací jazyk vytvořený a spravovaný firmou Microsoft. Jedná se o nadstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektově orientovaného programování (třídy, moduly, a další).

Samotný kód psaný v TypeScriptu se kompiluje do JavaScriptu. Jelikož je TypeScript nadstavbou nad JavaScriptem, je každý JavaScriptový kód automaticky validním TypeScript kódem.[10]

Rozšiřujícími vlastnostmi jsou [10]:

1. anotace typů a typová kontrola
2. třídy
3. rozhraní
4. výčtový typ
5. genericita
6. moduly

7. zkrácená syntaxe pro anonymní funkce
8. výchozí hodnoty parametrů funkcí

3.2.10 React

React je knihovna pro vývoj uživatelského rozhraní založená na JavaScriptu. Provozuje jej Facebook a open source komunita vývojářů. Přestože je React spíše knihovnou než jazykem, je široce používán ve vývoji webových aplikací. Knihovna se poprvé objevila v květnu 2013 a nyní je jednou z nejčastěji používaných frontendových knihoven pro vývoj webových aplikací [11].

Kapitola 4

Vývoj backendu

4.1 Co je to Backend

Backend se stará o zásadní logiku aplikace, webové stránky, softwaru nebo informačního systému. Samotný backend je neviditelným jádrem klientské aplikace, která z něj pouze čte data a pohybuje se v prostoru pomocí vystaveného API. Já jsem použil REST API [12].

4.2 REST API

REST API je velmi oblíbený druh API. Vyniká zejména svou jednoduchostí a čitelností. Je často využívané při vývoji mobilních i webových aplikací a tvorbě internetových stránek. API je zajištění komunikace mezi dvěma platformami, které si vzájemně vyměňují data. REST k tomuto obecnému mechanizmu dodává sadu doporučení a omezení, které když API dodržuje, tak se může nazývat REST API. Obsahuje dílčí endpointy, které jsou dostupné na definovaných URL adresách, které volá klientská aplikace přes HTTP protokol. Pro tělo požadavku se nejčastěji používá komunikační formát JSON [13].

4.3 Architektura

Typická pro Spring Boot projekt je třívrstvá architektura. Ta se skádá z následujících vrstev:

1. Controller
2. Servisní
3. Perzistentní

Obrázek 4.1: Třívrstvá architektura

4.3.1 Vrstva Controlleru

Třídy controlleru komunikují s klientem. Obdrží-li HTTP požadavek, zpracuje ho a dále ho pošle do servisní vrstvy. Klientovi pak vrací odpověd ve formátu JSON. V controlleru by měli být jen ty nejdůležitější věci jako například příjem vstupů, autentikaci a autorizaci, převod z DTO do modelu atd. Naopak by neměl řešit žádnou logiku [14].

4.3.2 Servisní Vrstva

V této vrstvě se provádí hlavní práce s daty aplikace a validace dat. Je volána třídami z controlleru a může volat další servisní třídy nebo repositář z perzistentní vrsvy [14].

4.3.3 Perzistentní Vrstva

Pro interakci s databází se využívá perzistentní vrstva. Její úkol je omezen na čtyři tzv. CRUD operace (vytvořit, číst, aktualizovat a smazat) [15].

4.4 Postup při vývoji

Při práci na projektu jsem měl k dispozici osnovu čemu se věnovat a v jakém pořadí. Po dokončení bloku osnovy jsem musel oslovit mého vedoucího pro kontrolu a až poté jsem mohl pokračovat dál.

4.4.1 Vývojové prostředí

Jako první věc jsem si musel vybrat vývojové prostředí. Ačkoli na vysoké škole moji profesoři, kteří mě vyučovali Javu, primárně používali Eclipse, byla mi doporučena IntelliJ IDEA a byla to změna k lepšímu.

4.4.2 Struktura projektu

Jako další postup bylo vytvoření struktury projektu. Pro správné fungování finalní aplikace není potřebná žádná specifická struktura, ale existuje doporučená tzv. „Best practices“, která pomůže v orientaci při práci a případným dalším programátorům, kteří po mě projekt převezmou. Na obrázku 4.2 je jedna z doporučených struktur.

Obrázek 4.2: Doporučená struktura projektu
[15]

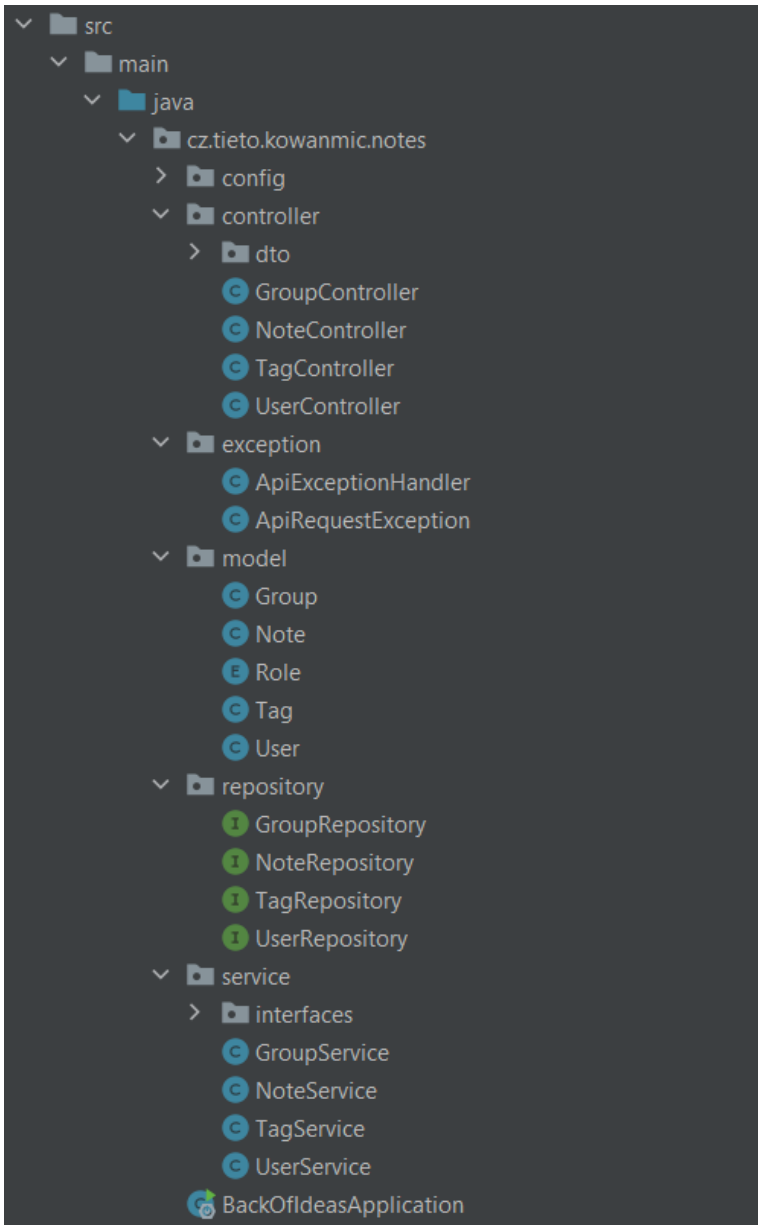
Další doproučená struktura projektu je rozdělení balíčků podle architektury. Na obrázku 4.3 je struktura mého projektu, která je rozdělená podle tohoto doporučení.

Pokud třída neobsahuje deklaraci balíčku, považuje se za součást „výchozího balíčku“. Také použití „výchozího balíčku“ se obecně nedoporučuje a je třeba se mu vyhnout. Může způsobit zvláštní problémy aplikacím Spring Boot, které používají anotace @ComponentScan, @EntityScan nebo @SpringBootApplication, protože se čte každá třída z každého jaru [15].

Nicméně je doporučeno, aby hlavní třída aplikace byla umístěna v kořenovém balíčku nad ostatními třídami. *@SpringBootApplication* anotace často umístěna nad hlavní třídou a implicitně definuje základní „hledací balíček“ pro určité položky [15].

Základní balíčky v projektu jsou:

1. config
2. model
3. repository
4. service
5. controller



Obrázek 4.3: Struktura mého projektu

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Table(name = "notes")
public class Note
```

Výpis kódu 4.1: Anotace třídy modelu Note

Poslední balíček exception, který jsem použil, není nutný. Uvnitř se nachazí mé vlastní nadefinované vyjímky.

4.4.3 Modely

Třídy v balíčku model reprezentují datový model a jejich atributy. Každá třída odpovída jednotlivé databázové tabulce se stejným názvem a atribut modelu odpovídá sloupci v odpovídající tabulce.

Moje modely jsou:

1. User
2. Tag
3. Note
4. Group

Nejdříve jsem si rozmyslel, jaké atributy budou jednotlivé modely obsahovat a jakého budou typu. Každý model navíc musí mít i ID, pro které jsem zvolil datový typ *Long*.

Lombok

Při tvorbě modelů mi velice pomohl Projekt Lombok. Projekt Lombok je knihovna založená na anotacích, která pomáha zredukovat psaný kód. Lombok nabízí různé anotace zaměřené na nahrazení kódu Java, o kterém je dobře známo, že je standardní, opakující se nebo únavný při psaní. Například pomocí Lomboku se můžu vyhnout psaní konstruktorů bez argumentů nebo psaní metod pro nastavování a získávání hodnot atributů pouhým přidáním několika anotací. Kouzlo se děje během kompilace, kdy knihovna vloží bajtkód představující požadovaný a standardní kód do souborů .class [16].

Anotace tříd

- @Entity - Pokud chci, aby se třída ukládala do databáze musím ji dát tuto anotaci, aby ji JPA rozeznalo, také musím zajistit, aby měla konstruktor bez argumentů a primární klíč [17].

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

@Column(
        name = "name",
        nullable = false,
        columnDefinition = "TEXT"
)
    private String name;

@Column(
        name = "text",
        columnDefinition = "TEXT"
)
private String text;

@Column(
        name = "expire",
        columnDefinition = "TIMESTAMP WITHOUT TIME ZONE"
)
private LocalDate expire;
```

Výpis kódu 4.2: Anotace atributů modelu Note

- @Table - Anotace, díky které můžu měnit název tabulky v databázi.
- Lombok anotace
  - @Data - Tato anotace pochází z Projekt Lombok a je to zkratka pro anotace *@ToString*, *@EqualsAndHashCode*, *@Getter* na všechny atributy, *@Setter* na všechny atributy, které nejsou označené jako final a jako poslední obsahuje *@RequiredArgsConstructor* [18].
  - @AllArgsConstructor - Vytvoří konstruktor se všemi atributy.
  - @NoArgsConstructor - Vytvoří konstruktor bez argumentů.

Anotace atributů

- @Id - Takto označený atribut se bude brát jako primární klíč.
- @GeneratedValue - Pro tento atribut bude automaticky generovaná hodnota, používá se pro primární klíč.
- @Column - Používá se pro nastavení vlastností odpovídajícího sloupce v databázi, při nespecifikování nastaví defaultní hodnoty.

```
logging.level.org.hibernate.type=trace
spring.datasource.hikari.connectionTimeout=20000
spring.datasource.hikari.maximumPoolSize=5
logging.pattern.console=%-5level %logger{36} - %msg%n
spring.datasource.url=jdbc:postgresql://127.0.0.1:5432/NotesDatabase
spring.datasource.username=****
spring.datasource.password=****
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=none
```

Výpis kódu 4.3: Nastevení v application.properties

- name - Nastavuje název sloupce.
- nullable - Říka, zdali je atribut povinný.
- columnDefinition - Definuje typ sloupce.
- unique - Definuje, jestli musí být hodnota atributu unikátní.

4.4.4 Databáze a vztahy mezi modely

Ve firmě se dlouhodobě používá PostgreSQL a tedy i já jsem dostal za úkol sprovoznit projekt na tomle databázovém systému. Jako první jsem si musel databázi nainstalovat a nastavit. Nastavení na straně Spring bootu se provadí v souboru *application.properties*, která se nachází ve složce *src/main/resources*.

4.4.4.1 application.properties

Tento soubor slouží k zápisu konfigurace vlastností související s aplikací. Také obsahuje různou konfiguraci, která je nutná ke spuštění aplikace v jiném prostředí, a každé prostředí bude mít jinou vlastnost, kterou definuje. Uvnitř souboru vlastností aplikace definujeme každý typ vlastnosti, jako je změna portu, konektivita databáze, připojení k serveru a mnoho dalších [19].

Já jsem nastavil tyto vlastnosti:

- spring.datasource.hikari - Implemetuje JDBC DataSource, která poskytuje recyklaci spojení (Connection pooling).
  - connectionTimeout - Tato vlastnost řídí maximální počet milisekund, po které bude klient čekat na připojení.
  - maximumPoolSize - Tato hodnota určuje maximální počet připojení k databázi.
- logging.level.org.hibernate.type=trace - Nastavuje úroveň logování na trace.
- logging.pattern.console - Nastavuje formát logu.

- nastavení databáze
  - spring.datasource.url=jdbc:postgresql://127.0.0.1:5432/NotesDatabase - Umístění databáze, kde *NotesDatabase* je její jméno.
  - spring.datasource.username a spring.datasource.password - Autentizace pro přístup do databáze.
  - spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect - Definuje SQL dialekt, aby generoval lepší SQL dotazy pro danou databázi.
  - spring.jpa.hibernate.ddl-auto=none - Specifikuje inicializaci databáze při startu. *None* se nejčastější používá v produkci. Během vývoje jsem používal *create-drop* pro opětovné restatování dat v databázi. Slovo *ddl-auto* odkazuje na automatické vytvoření databázových tabulek na základě modelů a jejich anotací.

4.4.4.2 Vztahy mezi modely

Nyní příchází na řadu implementovat vztahy mezi modely.

Z požadavků zákazníka 3.1 jsem vyvodil následující vztahy:

1. Uživatel
   (a) může vytvořit několik tagů.
   (b) může vytvořit několik skupin.
   (c) může vytvořit několik poznámek.
2. Tag
   (a) může patřit jednomu uživateli.
   (b) může být přiřazen několika skupinám.
   (c) může být přiřazen několika poznámkám.
3. Skupina
   (a) může patřit jednomu uživateli.
   (b) může mít několik poznámek.
   (c) může mít několik tagů.
4. Poznámka
   (a) může patřit jednomu uživateli.
   (b) může být přiřazena do jedné skupiny.

(c) může mít několik tagů.

Spring Boot JPA poskytuje tyto vztahy:

1. OneToOne - vztah jedna ku jedné
2. OneToMany - vztah jedna ku mnoha
3. ManyToOne - vztah mnoha ku jedné
4. ManyToMany - vztah mnoha ku mnoha

OneToOne V případě jedna ku jedné existuje primární klíč jedné tabulky jako cizí klíč v jiné tabulce. Tomuto neodpovíd žádný ze vztahů 4.4.4.2.

OneToMany a ManyToOne Zde je to poddobné jako v případě OneToOne. Primární klíč tabulky, který hraje roli One, existuje jako cizí klíč v přidružených tabulkách [20].

Tento vztah existuje mezi následujícími modely:

1. uživatel a skupina
2. uživatel a tag
3. uživatel a poznámka
4. skupina a poznámka

ManyToMany Ve scénáři ManyToMany je přidružení mezi dvěma entitami sledováno pomocí tabulky křížových odkazů. Toto je tabulka, která obsahuje oba primární klíče souvisejících entit jako cizí klíče [21].

Tento vztah existuje mezi následujícími modely:

1. tag a skupina
2. tag a poznámka

4.4.4.3 Implementace v projektu

Pro implementaci vztahů mezi entitami existují dva přístupy. Jedním z nich je jednosměrné mapování (Unidirectional Mapping). U takového mapování ví o vztahu pouze jedna strana. Směr odkazu pak určuje, kterým směrem se může dotazovat. V případě jednoho uživatele ke mnoha poznámkách, by se mohla poznámka dotazovat na uživatele, ale uživatel by se nemohl dotazovat na vlastněné poznámky. V druhém obousměrném mapování (Bidirectional Mapping) má každá entita vlastnost, která odkazuje na druhou entitu, se kterou je ve vztahu. Tedy o sobě ví navzájem. Mě bylo doporučeno použít obousměrné mapovaní [22].

```
@OneToMany(
        mappedBy = "user",
           orphanRemoval = true,
        cascade = {CascadeType.REMOVE},
        fetch = FetchType.LAZY
)
private Set<Note> notes = new HashSet<>();
```

Výpis kódu 4.4: Implementace OneToMany na straně uživatele

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(
        name = "user_id",
        nullable = false,
        referencedColumnName = "id",
)
private User user;
```

Výpis kódu 4.5: Implementace ManyToOne na straně poznámky

OneToMany Implementace OneToMany v jednom směru a ManyToOne ve směru opačném předvedu na vztahu mezi uživatelem a poznámkou. Pomocí anotace *@ManyToOne* specifikuji, který model bude vlastnit cizí klíč. To bude poznámka. *@JoinColumn* mi pomůže určit, který atribut v modelu uživatel bude použit pro připojení. Oproti *@ManyToOne* je *@JoinColumn* nepovinný [20]. V anotaci *@JoinColumn* také mohu specifikovat:

- name - Je název cizího klíče v tabulce.
- nullable - Říká, zdali může sloupec cizího klíče mít hodnotu *null*.
- referencedColumnName - Určuje název sloupce, na který ukazuje cizí klíč.

Jakmile jsem nadefinoval vlastnosti pro vlastníka vztahu, tak mi vziklo jednosměrné mapování. Pro obousměrné musím nastavit i referenční stranu vztahu a to pomocí anotace *@OneToMany*. I zde mohu podprobněji specifikovat vlastnosti:

- mappedBy - Je název atributu vlastníka vztahu.
- orphanRemoval - Pomocí boolean hodnoty nastavuji, jestli při smazání uživatele, se smažou i všechny přidružené poznámky.
- referencedColumnName - Definuje název sloupce, na který ukazuje cizí klíč.

```
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(
        name = "tag_notes",
        joinColumns = @JoinColumn(name = "tag_id"),
        inverseJoinColumns = @JoinColumn(name = "note_id")
)
private Set<Note> notes = new HashSet<>();
```

Výpis kódu 4.7: Implementace ManyToMany na straně tagu

- cascade - Vztahy entit často závisí na existenci jiné entity. Když provedeme nějakou akci na cílové entitě, stejná akce bude aplikována na přidruženou entitu. Cascade nám dovoluje definovat, jaké operace se přenesou. V příkladu 4.4 jsem použil *remove*, což znamená, že pokud bude smazán uživatel, smažou se i všechny jeho poznámky.

- fetch - Definuje načítání entit a entit ve vztahu. Typ *Lazy* načte související entity jen v případě jejího použití.

```
@ManyToMany(mappedBy = "notes", fetch = FetchType.LAZY)
private Set<Tag> tags = new HashSet<>();
```

Výpis kódu 4.6: Implementace ManyToMany na straně poznámky

ManyToMany Implementaci ManyToMany ukážu na vztahu mezi poznámkou a tagem 4.7. Vlastníkem přidružení je tag a to znamená, že asociaci definuje a entita poznámky na ni pouze odkazuje. I zde je vyžadována anotace, a to konkrétně *@ManyToMany*, na obous stranách asociace.[20] Pomocí *@JoinTable* definuju následující vlastnosti:

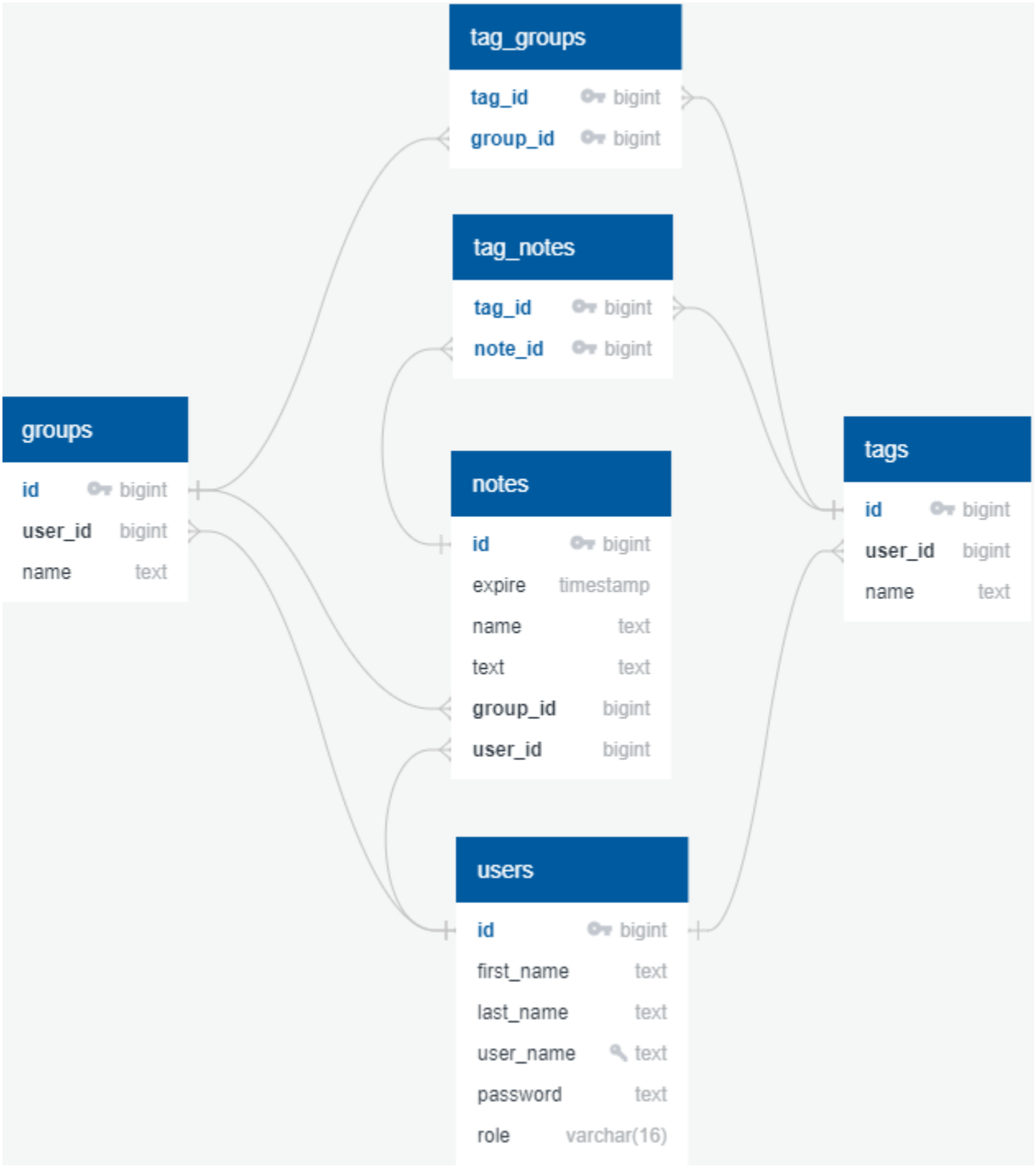
- name - Tímto atributem nastavím jméno tabulky.

- joinColumns - Připojení cizího klíče ke straně vlastníka.

- inverseJoinColumns - Připojení cizího klíče ke druhé straně.

4.4.4.4 Schéma databáze

Po přídání potřebných anotací v modelech a nastavení v *application.properties* se automaticky vytvoří tabulky v databázi. Výsledná schéma je vyzobrazena na obrázku 4.4



Obrázek 4.4: Schéma databáse

```java
@Repository
public interface NoteRepository extends PagingAndSortingRepository<Note, Long> {

    // @Query("SELECT n FROM Note n WHERE n.user.id = ?1")
    Page<Note> findAllByUserId(Long id, Pageable page);

    Optional<Note> findById(Long noteId);
}
```

Výpis kódu 4.8: Implementace ManyToMany na straně tagu

4.4.5 Repository

V dalším blocku jsem dostal za úkol implmentovat repositář, který je zodpovědný za provádění operací nad objektem, jako je například mazání, ukádání nebo vyhledávání podle kritérií.

Pro vytvoření repositáře je nutné oanotovat třidu pomocí *@Repository*. Tato anotace slouží k převedení vyjímek, které Spring nezná, na „Unchecked“ vyjímky, které rozezná. To pomáhá k identifikaci, kde problém nastal, jelikož všechny transakce se odehrávají v databázi. Repositář musí být rozhraní, které rozšiřuje jiné repositáře [23]. Já jsem použil *PagingAndSortingRepository*, která rozšiřuje *CrudRepository*, a která navíc poskytuje stránování a třídění. Ta bere jako typové argumenty model, nad kterým bude operace provádět a datový typ atributu ID.

Stránkování slouží ke zvýšení výkonu webových aplikací. Velké seznamy jsou rozděleny na stránky, které jsou uživateli zobrazeny a on mezi nimi může přepínat.[24]

Repositář poskytuje jen základní metody pro operace s objektem. V případě 4.8 potřebuju vyhledat všechny poznámky podle ID uživatele, jenomže repositář takový dotaz neposkytuje. Mám tedy dvě možnosti:

1. Získání dotazu odvozeného od názvu metody. Takový způsob ale není stoprocentní. Může se stát, že je zapotřebí složitější dotaz [25].

2. Pomocí anotace *@Query*, díky které se může napsat jakýkoli JPQL dotaz.

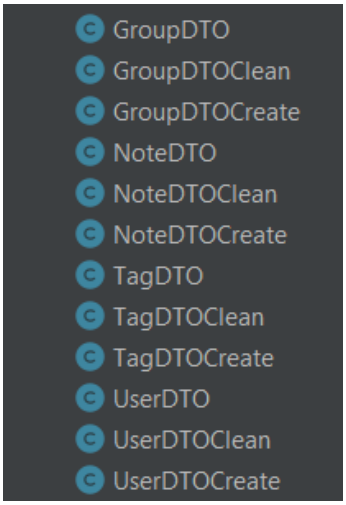
```java
@AllArgsConstructor
@NoArgsConstructor
@Data
public class UserDTOClean {

    private Long id;

    private String firstName;

    private String lastName;

    private String userName;

    private String role;
}
```

Výpis kódu 4.9: Implementace třídy UserDTOClean

4.4.6 Objekty pro přenos dat

Objekty pro přenos dat neboli DTO (Data Tranfer Object) slouží pro zajištění zabezpečení a integritu dat. Například pokud posílám instanci uživatele na fronedovou část, kde chci zobrazit pouze jméno a příjmení, je nežadoucí abych posílal i heslo nebo vlastněné poznámky. Proto definuji novou třídu *UserDTOClean* 4.9, která bude sloužit pro posílání základních informací.

Seznam všech DTO tříd 4.5.

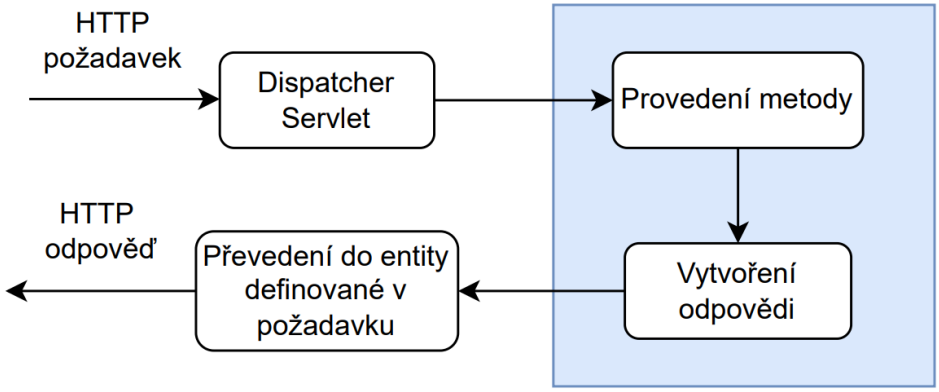


Obrázek 4.5: Seznam všech tříd DTO

4.4.7 Controller

Třídy Controlleru jsou zodpovědné za komunikaci s klientem. Zpracovávají příchozí REST API požadavky, připravují model, který pak následně vrátí jako odpověď.

Spring Boot REST API Workflow



Obrázek 4.6: Spring Boot REST API Workflow

Nejprve je požadavek přijat *Dispatcher Servlet*, který je zodpovědný za zpracování všech příchozích URI požadavků a mapuje je na odpovídající metody v *controlleru*. V diagramu 5.1 je popsán průběh požadavku. Modře označenou část má na starost programátor [26].

Třídy musí být oanotované pomocí *@Controller*, jinak Dispatcher Servlet nezahrne třídu controlleru do hledání po odpovídající metodě namapované na cestě pomocí *@RequestMapping* [26].

@RestController, který jsem použil v kódu 4.10 je kombinace *@Controller* a *@ResponseBody*. *@ResponseBody* po vrácení dat metodou serializuje odpověď do formátu JSON. Dále pomocí *@RequestMapping* definuju cestu, na kterou se má mapovat.

@GetMapping je zkrácený zapis pro *@RequestMapping(method = RequestMethod.GET)* a používá se k mapovaní GET požadavků na takto anotovanou metodu. Dohromady existuje anotace pro pět různých typů HTTP požavků [27]:

1. GetMapping - Používá se čtení nebo získání dat. V případě úspěchu vrací odpověd v JSON a stavový kód 200 (OK). V případě neúspěchu nejčastěji vrací 404 (NOT FOUND) nebo 400 (Bad Request).

2. PostMapping - Používá se pro vytváření nových dat. Při úspěchu vrací kód 201 (Created). Po neúspěchu vrací 404 (Not Found) nebo 409 (Conflict) pokud zdroj již existuje.

3. PutMapping - Nachází využití při aktualizaci. Posílá požadavek s tělem obsahující známy zdroj s pozměněnými daty. Po úspěšném provedení vrací 200 (OK) nebo 404 (Not Found) pokud zdroj pod hledaným ID neexistuje.

4. DeleteMapping - Maže zdroj specidikovaný v URI. Pokud smazání proběhlo s úspěchem, navrátí stavový kód 200 (OK), v opačném případě 404 (Not Found).

5. PatchMapping - Pouze obsahuje modifikovanou část zdroje a ne celou. Po úspěšné modifikaci vrací 200 (OK) nebo 404 (Not Found) pokud zdroj pod hledaným ID neexistuje.

@RequestBody převede object z těla HTTP požadavku do požadovaného modelu. Anotovaný model musí odpovídat JSON formátu, který dostaneme od klienta [28]. Zde využívám DTO modely, které pak pomocí *ModelMapper* mapuju do odpovídajícího doménového modelu a při vrácení požadavku zase naopak.

@PathVariable využívám k nastavení proměnné z URI požadavku do metody jako parametr. Název parametru musí odpovídat proměnné v cestě.

4.4.8 Business vrstva a její logika

Jako první přidam rozhraní pro všechny třídy v servisní vrstvě. Ta slouží ke čtyřem účelům [29]:

1. Vytvoření volného spojení mezi dvěma třídami. Pomocí rozhraní se třída, která závisí na servise, již nespoléhá na její implementaci. To umožní použít je nezávisle.

2. Při práci s více moduly, které na sobě závisí. Bez implementace rozhraní, budou všechny domény v rámci aplikace svázané dohromady a aplikace pak bude vysoce propojena.

3. Pro testovaní, kdy rozhraní poskytuje fiktivní implemetaci. Nicméně to už není potřeba, protože existují knihovny jako Mockito, které tenhle problém řeší.

4. Jako firemní standart, pro lepší srozumitelnost kódu, aby potenciální programátoři, kteří to po mě převezmou, se lehčeji zorientovali.

Nyní příchází na řadu vymyslet a napsat logiku aplikace a práci s daty. Hlavní kritéria, která musí aplikace splňovat jsou:

1. Tag je globálně viditelný a je doporučovaný ostatním uživatelům, nehledě na to, jestli daný tag uživatel vytvořil nebo ne. K tomuto využiju metodu z repositáře *findAll*.

2. Uživatel může editovat a mazat jen svoje tagy. Tedy ověřím, zdali přihlášený uživatel se shoduje s vlastníkem tagu.

3. Smazání uživatele smaže i všechny jeho poznámky, tagy a skupiny. Pro každý objekt svázaný s uživatelem zavolám metodu mazání.

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

private final UserService userService;
    private final ModelMapper modelMapper;

    public UserController(UserService userService, ModelMapper modelMapper) {
        this.userService = userService;
        this.modelMapper = modelMapper;
    }

    @GetMapping
    public Page<UserDTO> all(Pageable page) {
        return userService.findAll(page).map(this::convertToDto);
    }

    @PostMapping
    public UserDTOClean newUser(@RequestBody UserDTOCreate newUser) {
        return convertToDtoClean(userService.createUser(convertToEntityCreate(
            newUser)));
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteById(id);
    }
}
```

Výpis kódu 4.10: Implementace třídy UserController

4. Po smazání skupiny se odstraní i všechny tagy, které nejsou nikde jinde použité. Musím provést kontrolu, zdali má tag nějaké výskyty. V opačném případě ho zmažu.

5. Tak jako v předchozím bodě, i po smazání poznámky se smažou nepoužité tagy. Stejné jako v předchozím případě.

6. Smazání tagu, také odstraní veškerý výskyt tagu u skupin a poznámek. Při správném nastavení kaskády, se odstraní výskyty i v přidružených skupinách a poznámkých.

7. Smazání tagu, také odstraní veškerý výskyt tagu u skupin a poznámek. Stejné jako v přeschozím případě.

4.4.9 Testování

Nyní, po implementaci logiky aplikace, je na čase ji otestovat a zjistit, zdali vrací očekávané hodnoty.

Existuje několik důvodů, proč psát testy [30]:

1. Šetří čas při debugování.

2. Donutí mě přemýšlet o tom co píšete.

3. Částečně dokumentují kód.

4. Nutí mě psát kvalitní kod napoprvé.

5. Pomáhají chytat chyby na poslední chvíli před nasazením.

Existuje několik typů testovaní, které testují různé části aplikace. Já jsem použil následující typy:

1. Unit testy, které se zaměřují se na nejmenší jednotku návrhu softwaru. Testování se provádí pomocí vzorového vstupu a sledováním jeho odpovídajících výstupů [31].

2. Integrační testování je definováno jako typ testování, kde jsou softwarové moduly integrovány logicky a testovány jako skupina [31].

4.4.9.1 Unit testy

Pomocí Unit testů jsem jsem testoval servisní vrstvu, jenomže ta má závislost na perzistentní vrsvě. Nicméně abych mohl otestoval logiku servisní vrstvy, nepotřebuju znát implementaci a funkčnost perzistentní vrsvy. K tomuto účelu nám slouží mockování, které jsem implementoval pomocí frameworku Mockito.

Mockito je open source framework, který se používá pro Unit testování Java aplikací. Mockito se používá k simulování rozhraní, ke kterému se můžou přidat fiktivní funkce. Hlavním cílem použití

frameworku je zjednodušit vývoj testu mocknutím externích závislostí a jejich použitím v testovacím kódu [32].

Třídy s testy se nachází ve složce *test/java/notes*.

Jako prvním musím v kódu 4.11 pomocí Mockito mocknout závislosti. Ty poté nebudou vracet návratové hodnoty, ale stále na nich můžu testovat, zdali metody byly vyvolány. Za pomocí anotace *@BeforeEach* říkám, že se to má provést před každým testem.

Při testování se testuje vždy konkretní funkcionalita, například zdali byla metoda vůbec vyvolána nebo jestli vrátila očekávanou chybu. Testy by měli být napsány pro každou možnou verzi scénáře.

V metodě 4.11 testuju, zdali při tvorbě nového uživatele byly zadány všechny povinné parametry. Zde konkrétně parametr chybí a tedy očekávám, že metoda vrátí chybovou hlášku.

4.4.9.2 Integrační testy

Integrační testy jsem použil pro testování ukládání dat ze servisní vrstvy do databáze. Tyto testy by měli být oddělené od Unit testů, protože jsou časově náročné. Využívají totiž reálnou databázi k provedení.

Pro testování jsem použil novou instanci databáze H2. Pro připojení k ní ji musím prvně nastavit v *application-test.properties*.

Anotace *@SpringBootTest* je užitečná, když potřebujeme zavést celý IoC kontejner. Anotace funguje tak, že vytvoří ApplicationContext, který bude použit v testech [33].

V metodě updateUser kódu 4.12 testuju zdali se správně aktualizují informace o uživateli.

1. Nejdříve vytvořím novou instanci uživatele, který představuje získané data od klienta.
2. Vezmu z databáze uživatele se stejným ID.
3. Zavolám metodu updateUser, díky které by se měly zaměnit staré data za nové.
4. Pomocí *assertEquals* se můžu dotázat, zdali se data správně aktualizovala.

4.4.10 Autentikace a Autorizace

Poslední částí vývoje backendu je přidat autentikaci a autorizaci.

Autentikace je proces ověření identity uživatele. Uživatel tedy prokazuje, že je tím, za koho se vydává. Pro autentikaci do systému musí zadat uživatelské jméno a heslo.

Autorizace znamená povolení k nějakému úkonu nebo operaci. Nejčastěji používá role pro rozdělení pravomocí.

Byla mi poskytnutá šablona, využívající OAuth 2, kterou jsem měl do své aplikace implementovat.

```
class UserServiceTest {

    UserRepository repository;
    TagRepository tagRepository;
    IGroupService groupService;
    INoteService noteService;

    UserService userService;

    @BeforeEach
    void setUp() {
        this.repository = Mockito.mock(UserRepository.class);
        this.noteService = Mockito.mock(NoteService.class);
        this.groupService = Mockito.mock(GroupService.class);
        this.tagRepository = Mockito.mock(TagRepository.class);
        this.userService = new UserService(repository, tagRepository, groupService
            , noteService);
    }


  @Nested
    class CreateUser {
        @Test
        public void createUser_missingPar() {
            User user = new User(1L, "Michal", "Nemec", "nem");

            Mockito.when(repository.save(user))
                    .thenThrow(IllegalArgumentException.class);

            IllegalArgumentException thrown = Assertions.assertThrows(
                    IllegalArgumentException.class,
                    () -> userService.createUser(user)
            );


            Assertions.assertEquals("User with user name nem has forbidden data",
                thrown.getMessage());
        }
    }

}
```

Výpis kódu 4.11: Unit testování třídy UserService

```java
@SpringBootTest
public class UserServiceIntegrationTest {

    @Autowired
    UserService userService;

    @Test
    void updateUser() {
        User newUser = new User(1L, "Michal", "Alojs", "kowal", "1234");
        User oldUser = userService.findUserById(newUser.getId());

        User updatedUser = userService.updateUser(newUser, oldUser);

        Assertions.assertEquals(newUser.getLastName(), updatedUser.getLastName());
        Assertions.assertEquals(newUser.getFirstName(), updatedUser.getFirstName()
            );
        Assertions.assertEquals(newUser.getUserName(), updatedUser.getUserName());
    }
}
```

Výpis kódu 4.12: Integrační testování uživatele

4.4.10.1 OAuth 2

OAuth 2 je autorizační rámec pro umožnění sdílení zdrojů zabezpečeným způsobem prostřednictvím sekvence kroků, kdy vlastník zdroje povolí klientskou aplikaci k určitému chráněnému zdroji na omezenou dobu [34].

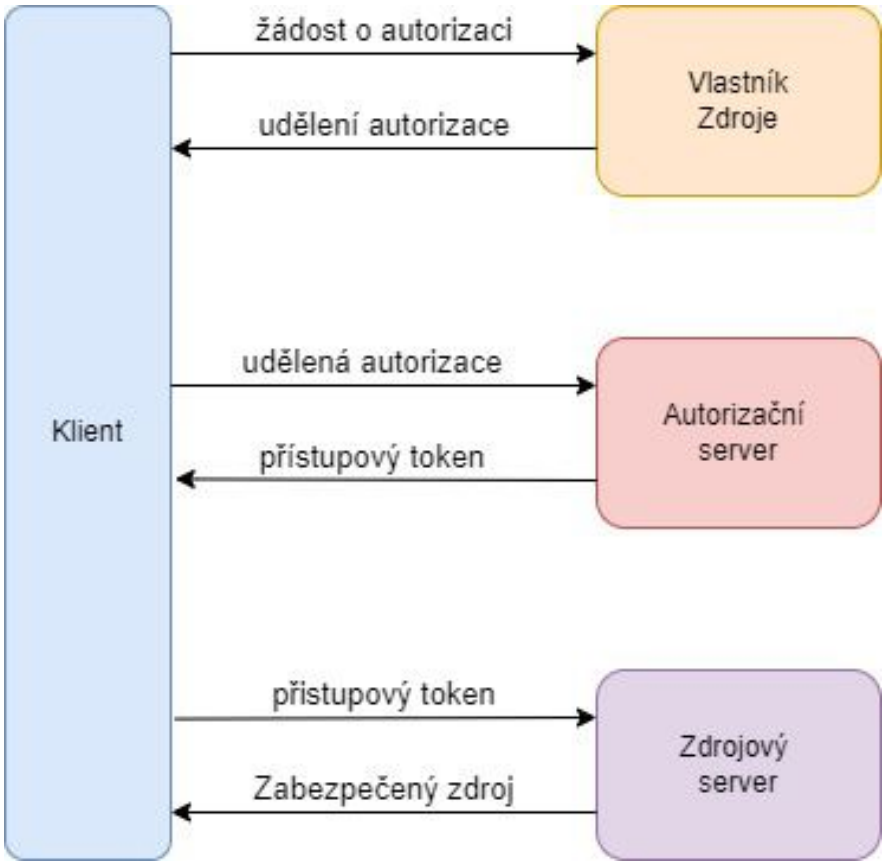
Hlavní role při autorizaci [34]:

1. Resourse owner (Vlastník zdroje) je uživatel, který vlastní chráněné prostředky na libovolném servru. Bez souhlasu vlastníka, nelze udělit žadnou autorizaci.

2. Client Application (Klientská aplikace) je aplikace třetí strany, v mém případě frontend, která je registrována na autorizačním servru a požaduje přístup k chráněným datům na zdrojovém servru jménem vlastníka zdroje.

3. Authorization Server (Autorizační server) je zodpovědný za poskytování autorizačních a přístupových tokenů klientovi jménem vlastníka zdroje. Autorizační server vydává přístupové tokeny jménem uživatele až poté, co byl uživatel nejprve ověřen.

4. Resource Server (Zdrojový server), která hostuje zabezpečené zdroje od vlastníka prostředků. Jakékoli chráněné prostředky na zdrojovém serveru jsou přístupné pouze vlastníkovi prostředku po ověření nebo jakékoli klientské aplikaci, které byl udělen přístup vlastníkem prostředku získáním přístupového tokenu vydaného prostřednictvím autorizačního serveru.



Obrázek 4.7: OAuth2 Workflow diagram
[34]

Kapitola 5

Vývoj frontendu

Po dokončení backedové části aplikace je na čase přejít na frontend. Primárním cílem téhle části bylo, abych se naučil v Reactu orientovat a nacházet chyby a ne vyvíjet celý frontend od základu.

5.1 Co je Frontend

Část webové aplikace nazývané také jako „client side“, která přímo interaguje s uživatelem. Zahrnuje to vše s čím přichází uživatel do styku, jako např. barvy textu, obrázky, tabulky či grafy, tlačítka, barvy a navigační menu. Základem dobrého frontendu je dobré UI (vzhled stránky) a UX (uživatelská zkušenost), tedy prostředí by mělo být intuitivní, uživatelsky přívětivé, nepřeplácané a příjemné na pohled [35].

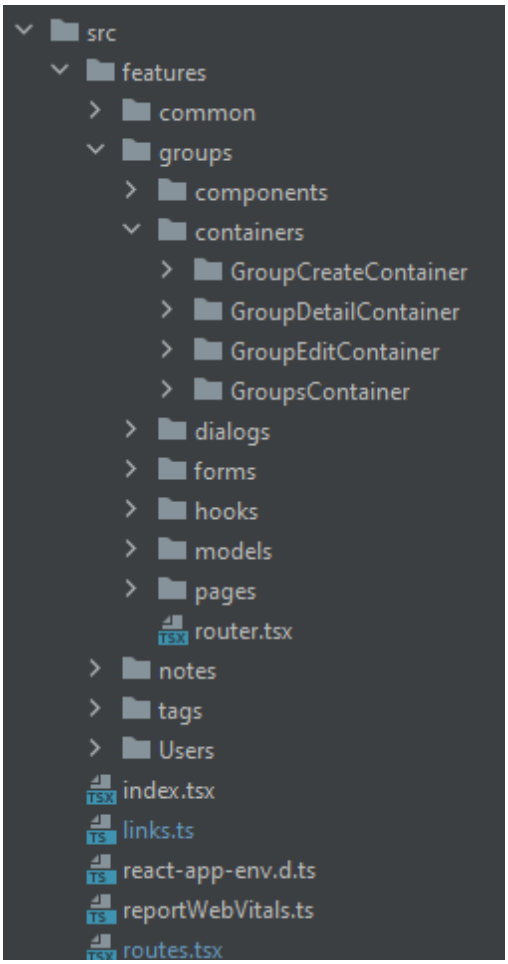
5.2 Nastavení prostředí

Jako úkol do začátku, jsem si nastavit prostředí. K vývoji frontendu jsem si vybral WebStorm, který má velice podobné rozhraní jako IntelliJ IDEA, které jsem použil pro vývoj backendu a pro studenty je zadarmo.

5.3 Rozložení komponent

První věc po nastavení prostředí, kterou jsem musel udělat bylo naklonovat šablonu Reactí aplikace z GitLabu. Ta sloužila jako odrazový můstek. V ní bylo základní nastavení a rozdělení do balíčků podle firemního standartu. Součástí původní šablony byl i příklad, který jsem mohl použít k inspiraci.

Mým úkolem bylo přidat stránky pro zobrazení, editaci a vytváření nových uživatelů, poznámek, tagů a skupin.



Obrázek 5.1: Struktura projektu

5.4 Použité techlologie

5.4.1 NPM

NPM je největší softwarový registr na světě a obsahuje přes 800 tisíc blíčků kódů. Nejčastěji ho používají open source vývojáři ke sdílení softwaru. Slouží také ke správě závislostí [36].

Všechny balíčky a závislosti jsou definovány v souboru s názvem *package.json*.

5.4.2 React

React je JavaScriptí knihovna pro tvorbu SPA (jedno stránková aplikace), díky kterému můžeme měnit kontent na stránce bez nutnosti znovunačtení. Při změně dat se znovu načtou jim odpovídající komponenty. Aplikace vyvíjena pomocí Reaktu je rozdělena do komponent. Každá komponenta je odpovědná za vykrelení části HTML a je znovu použitelná [37].

React využívá JSX, což je syntaktické rozšíření JavaScriptu. Používá se k vytváření reactích elementů, ty jsou poté renderovány do DOM.

```
const element = <h1> Prikladovy kod </h1>;
```

Výpis kódu 5.1: Přiklad rozšíření JSX

5.4.2.1 DOM

DOM je standardní logická reprezentace jakékoli webové stránky definovaná W3C. DOM je stromová struktura, která obsahuje ve svých uzlech všechny elementy a vlastnosti webové stránky. Před Reactem se po aktualizaci dat, musela vždy aktualizovat celá stránka, což spotřebovávalo hodně času. React ale poskytuje Virtualní DOM, což je kopie skutečné reprezentace DOM, do které se ukládají veškeré aktualizace provedené uživatelem a následně jsou všechny aktualizace přeneseny do původního DOM najednou [38].

5.4.3 Material UI

Material UI je open source frontend framework, který se používá pro design Reactích komponent. Šablona, kterou jsem použil pro inicializaci projektu používá starší verzi 4.

5.4.4 TypeScript

TypeScript je open source programovací jazyk, který je nástavbou nad JavaScriptem. Při vývoji v Reactu přidává tyto výhody [39]:

1. Přidává statické typování, díky kterému je kód i lépe čitelný.
2. Možnost vytvářet rozhraní.
3. Poskytuje podporu IDE, díky kterému mi prostředí lépe našeptává při psaní kódu.

Pokud by v příkladu 5.2 proměnná *data* nebyla datového typu Group, jak očekává komponenta *GroupDetailCard*, hodilo by mi to okamžitě chybu.

5.4.5 Router

React Router je základní knihovna pro směrování. Umožňuje navigaci mezi pohledy různých komponent v aplikaci React, umožňuje změnu adresy URL prohlížeče a udržuje uživatelské rozhraní synchronizované s její adresou [40].

V ukázce kódu 5.8 je směrovač mezi jednotlivými stránkami pro skupiny. Směrovač vrátí komponentu definovanou ve vstupu *component*, pokud vstup *path* odpovídá cestě.

```
const GroupDetailContainer: FunctionComponent = () => {
    ...
    return <GroupDetailCard group={data} />;
};

const GroupDetailCard: FunctionComponent<{ group: Group }> = ({ group }) => {
...
}
```

Výpis kódu 5.2: Příklad využití statického typování

```
const GroupsRouter: FunctionComponent = () => (
  <Route path={Links.groups}>
    <GroupsLayout>
      <Switch>
        <Route
          exact
          path={Links.groups}
          component={React.lazy(() => import("./pages/GroupsPage"))}
        />
        <Route
          exact
          path={Links.groupEdit}
          component={React.lazy(() => import("./pages/GroupEditPage"))}
        />
        <Route
          exact
          path={Links.groupCreate}
          component={React.lazy(() => import("./pages/GroupCreatePage"))}
        />
        <Route
          exact
          path={Links.group}
          component={React.lazy(() => import("./pages/GroupDetailPage"))}
        />
      </Switch>
    </GroupsLayout>
  </Route>
);
```

Výpis kódu 5.3: směrovač mezi stránkami

```
<Formik
      onSubmit={onSubmit}
      initialValues={GroupFactory(initialValues)}
      validationSchema={GroupFormSchema}
    >
    ...
<Formik>
```

Výpis kódu 5.4: Ukázka Formik

```
export interface Group {
  id: number;
  name: string;
  tags?: Partial<Tag>[];
}
```

Výpis kódu 5.5: Model skupiny

5.4.6 Formik

Vyváření formuláří může vyžadovat napsat velkou spoustu řádků kódu, podle požadavků na správu jednolivých polí. To mi pomohl vyřešit Formik. Formik je open source knihovna, která řeší tři klíčové problémy při vytváření formulářů [41]:

1. Jak se manipuluje se stavem formuláře.

2. Jak je zpracováno ověřování formulářů a chybové zprávy.

3. Jak se zpracovává odeslání formuláře.

Pokud edituji skupinu, potřebuju nastavit její původní hodnoty do polí formuláře. K tomu se mi hodí vstup *initialValues*. Pomocí *validationSchema* můžu definovat jak mají vstupy vypadat, jakého mají být typu a jestli jsou povinné nebo ne. K tomu mi poslouží Yup. *ValidationSchema* automaticky tranformuje validační chyby do pěkného objektu, se kterým mají vstupy stejné hodnoty klíčů [42].

5.4.7 Yup

Yup je JavaScriptový nástroj pro tvorbu schémat, které slouží pro parsování hodnot a jejich validaci [43].

V kódu 5.5 je nadefinovaný model skupiny, odpovídající modelu z backendové části.

Zde v 5.5 definuji jak má vypadat schméma. Ke každému atributu definuju typ, a jestli je *required* (povinný) nebo *optional* (nepovinný).

```
const GroupFormSchema: SchemaOf<Group> = object({
  id: number().required(),
  name: string().required(),
  tags: array()
    .of(
      object().shape({
        id: number(),
        name: string(),
      })
    )
    .optional(),
}).required();
```

Výpis kódu 5.6: Schméma skupiny pro validaci Formiku

5.4.8 Hooks

Hooks povolují používat stavy a další funkce Reaktu bez použití třídy. Jsou to funkce, které se „zavěsí“ na stav a životní cyklus funkční komponenty.

Pokud bych chtěl napsat funkční komponentu a poté ji přidat nějaký stav, musel bych ji nejprve převést do třídy. To již nemusím díky Hooks, které se dají použít uvnitř funkčních komponent.[44]

Hooks mají 2 pravidla [44]:

1. Nesmí se volat uvnitř smyček, podmínek nebo vnořených funkcí. Musí se vždy používat na nejvyšším stupni funkce. To zajišťuje, že Hooks budou volány vždy ve stejném pořadí při rendrování komponenty.

2. Hooks nelze volat z běžných JavaScriptových funkcí, pouze z funkčních komponent.

5.4.9 useQuery

Dotaz (Query) je deklarativní závislost na asynchronním zdroji dat, který je svázán s jedinečným klíčem. Dotaz lze použít s jakoukoli metodou založenou na příslibu (nejčastěji GET) k načtení dat ze serveru.

Příslib obaluje proměnnou, kterou můžu nebo nemusím v moment inicializace znát, ale poskytuje mi metodu, která se o ni postará hned po tom, co hodnotu získám [45].

Objekt, který nám useQuery vrátí, obsahuje několik důležitých stavů [46]:

- isLoading - Pokud je stav true, dotaz nemá žádné data a načítá je.

- isError - Pokud je stav true, dotaz narazil na problém.

- isSucces - Pokud je stav true, dotaz proběhl v pořádku a data jsou dostupná.

- error - Zde se nachází error, pokud je stav isError true.
- data - Pokud je stav isSucces true, zde se nachází data.

Ty poté můžu použít a například vykreslit stránku s chybovou hláškou, zobrazit grafický prvek znázorňující načítání nebo použít data z dotazu.

```
export const useGroupQuery = (id: string) =>
  useQuery(getGroupQueryKey(id), () =>
    axios
      .get(`http://localhost:8080/api/groups/${id}`)
      .then((response) => response.data as Group)
  );
```

Výpis kódu 5.7: Implementace useQuery

5.4.10 useMutation

Oproti useQuery se mutace používají k vytváření, aktualizaci nebo mazání dat na servru [47].

Stejně jako u useQuery 5.4.9 i zde useMutation vrací object se stejnými stavy.

```
const useGroupCreateCommand = () =>
  useMutation((group: Group) =>
    axios
      .post(`http://localhost:8080/api/groups`, group)
      .then((response) => response.data as Group)
  );
```

Výpis kódu 5.8: Implementace useMutation

5.4.11 axios

Axios je knihovna, která slouží k vytváření HTTP požadavků na backend a pomáhá získávat data, která předává v mém případě do stavu data od useQuery nebo useMutation. Hlavním účelem použití Axios je pro získání podpory pro zachycování požadavků a odpovědí, konverzi dat do formátu JSON a jejich transformaci. Axios je založen na slibech, což mi dává možnost využít asynchronní JavaScript [48].

Přiklad použí v kódu 5.4.9.

Kapitola 6

Na čem pracuju dál

Poté co jsem dokončil Newcomers projekt, mě již nasadili na reálné projekty.

Jako první mě přechodně přidali na nejmenovaný projekt pro recepční. Zde jsem pouze na frontendu upravoval vzhled stránky.

Další projekt, který jsem dostal během praxe na starost a stále na něm pracuji je frontendová část pro testování posílání emailu. Backedová část je napsaná na Javě 8 a mým úkolem je na základě její implementace navrhnout správné prostředí pro komunikaci s ní. Nyní na ni spolupracují s ještě jedním kolegou stážistou.

Kapitola 7

Závěr

Absolvovanou praxi hodnotím velmi kladně. Již od prváku na vysoké škole jsem si přál zapadnout do týmů vyvojářů a pracovat na něčem co má smysl a na něčem, co někdo doopravy využije a bude přínosné pro ostatní.

Během praxe jsem využil široké spektrum znalostí získané během studia na vysoké škole a to primárně z těhto předmětů:

1. Úvod do programování
2. Objektově orientované programování
3. Úvod do softwarového inženýrství
4. Databázové systémy I
5. Databázové systémy II
6. Vývoj informačních systémů
7. Programování v Java I
8. Programování v Java II
9. Uživatelská rozhraní
10. Vývoj internetových aplikací
11. Administrace databázových systémů

Na praxi bylo učení a studování nových techologií můj denní chléb. Naučil jsem se vývoj webových aplikací od backend po frontend. Naučil se používat nové techlogie jako například Spring Boot,

GitLab, Postman nebo React. Kromě technických znalostí mě praxe naučila i práci v týmu a komunikovat s kolegy. Dále mě praxe naučila jak to chodí ve velké firmě nebo jak probíhá vývoj aplikace od nápadu po nasazení do produkce.

Ve firmě jsem potkal soustu úžasných lidí a mám v plánu tam zůstat a získávat nové zkušenosti. Praxe mi i dala vizi do budoucna a směr, kterým se chci vydat. Ačkoli praxe jako taková skončila, ve firmě budu pokračovat pracovat dál a těším se, všechno se ještě díky této příležitosti naučím.

Literatura

1. *Tietoevry*. 2022-03-22. Dostupné také z: https://cs.wikipedia.org/wiki/Tietoevry.
2. *GitLab*. 2004-03-24. Dostupné také z: https://whatis.techtarget.com/definition/GitLab.
3. *PostgreSQL*. 2004-03-24. Dostupné také z: https://www.tutorialspoint.com/postgresql/postgresql_overview.htm.
4. *Java*. 2004-03-27. Dostupné také z: https://www.britannica.com/technology/Java-computer-programming-language.
5. *Spring Boot*. 2004-03-27. Dostupné také z: https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm.
6. *Postman*. 2004-04-15. Dostupné také z: https://www.encora.com/insights/what-is-postman-api-test.
7. *HTML*. 2004-04-10. Dostupné také z: https://www.investopedia.com/terms/h/html.asp.
8. *CSS*. 2004-04-10. Dostupné také z: https://www.tutorialspoint.com/css/what_is_css.htm.
9. *JavaScript*. 2004-03-27. Dostupné také z: https://techterms.com/definition/javascript.
10. *TypeScript*. 2004-04-10. Dostupné také z: https://cs.wikipedia.org/wiki/TypeScript.
11. *React*. 2004-03-27. Dostupné také z: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs.
12. *Backend developer*. 2004-04-15. Dostupné také z: https://www.ackee.cz/blog/glossary/backend-developer.
13. *REST API*. 2004-04-15. Dostupné také z: https://www.damidev.com/slovnik/rest-api.
14. *Spring Boot Project Architecture*. 2004-04-19. Dostupné také z: https://www.javaguides.net/2020/07/spring-boot-project-architecture.html.
15. *Developing with Spring Boot*. 2004-04-10. Dostupné také z: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.structuring-your-code.

16. *Lombok*. 2004-04-19. Dostupné také z: https://auth0.com/blog/a-complete-guide-to-lombok/.

17. *Defining JPA Entities*. 2004-04-24. Dostupné také z: https://www.baeldung.com/jpa-entities.

18. *@Data*. 2004-04-23. Dostupné také z: https://projectlombok.org/features/Data.

19. *Application Properties*. 2004-04-22. Dostupné také z: https://www.geeksforgeeks.org/spring-boot-application-properties/.

20. *Spring Boot JPA Relationship*. 2004-04-24. Dostupné také z: https://tenmilesquare.com/resources/software-development/spring-boot-jpa-relationship-quick-guide/.

21. *Many-To-Many Relationship*. 2004-04-23. Dostupné také z: https://www.baeldung.com/jpa-many-to-many.

22. *Direction in Entity Relationships*. 2004-04-24. Dostupné také z: https://docs.oracle.com/cd/E19798-01/821-1841/bnbqi/index.html.

23. *Spring Boot Repository*. 2004-04-28.

24. *Stránkování*. 2004-04-27. Dostupné také z: http://podpora.flexibee.eu/cs/articles/4722193-strankovani.

25. *JPA Repositories*. 2004-04-27. Dostupné také z: https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html.

26. *@Controller and @RestController Annotations in Spring Boot*. 2004-04-28.

27. *Using HTTP Methods for RESTful Services*. 2004-04-28.

28. *Spring's RequestBody boot*. 2004-04-27. Dostupné také z: https://www.baeldung.com/spring-request-response-body.

29. *Do I need an interface with Spring boot?* 2004-04-27. Dostupné také z: https://dimitr.im/spring-interface.

30. *5 reasons why testing code is great*. 2004-04-28.

31. *Types of Software Testing*. 2004-04-28.

32. *Mockito*. 2004-04-28.

33. *@SpringBootTest*. 2004-04-28.

34. *OAuth 2.0 Introduction*. 2004-04-29.

35. *Frontend*. 2004-04-29.

36. *NPM*. 2004-04-29.

37. *Why Use React*. 2004-04-29.

38. *DOM*. 2004-04-29.
39. *Using TypeScript with React*. 2004-04-29.
40. *Router*. 2004-04-29.
41. *Formik*. 2004-04-29.
42. *Validation*. 2004-04-28.
43. *Yup*. 2004-04-28.
44. *React Hooks*. 2004-04-28.
45. *What is a promise*. 2004-04-28.
46. *Queries*. 2004-04-28.
47. *Mutations*. 2004-04-29.
48. *React axios*. 2004-04-29.