FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Srovnání technologií pro implementaci backendu v Javě
Student:	Branislav Zlacký
Vedoucí:	Ing. Jiří Mlejnek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Seznamte se s existujícími frameworky pro vývoj backendu v jazyce Java a proveďte jejich srovnání. Na základě srovnání vyberte vhodné kandidáty, ve kterých proveďte implementaci jednoduché ukázkové aplikace, včetně automatických testů. Porovnejte výhody a nevýhody vybraných frameworků z pohledu náročnosti vlastního vývoje a jejich podpory pro testování.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 29. října 2021 v Praze.

Bakalárska práca

SROVNÁNÍ TECHNOLOGIÍ PRO IMPLEMENTACI BACKENDU V JAVĚ

Branislav Zlacký

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedúci: Ing. Jiří Mlejnek
8. mája 2022

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov kódu

Chcel by som sa poďakovať svojmu vedúcemu, Ing. Jiřímu Mlejnekovi, za pomoc a čas, ktorý mi venoval počas tvorby tejto práce. Taktiež by som sa rád poďakoval svojej rodine a kamarátom za podporu.

Vyhlásenie

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 8. mája 2022 .....

Abstrakt

Táto bakalárska práca sa zaoberá implementáciou backendu aplikácie, ktorá slúži ako jednoduchá sociálna sieť. Tento backend je implementovaný s využitím jazyku Java v štyroch rôznych technológiach Spring, Quarkus, Micronaut a AWS. Taktiež ponúka prihlásenie pomocou Google OAuth 2.0 a užívateľa udržiava prihláseného pomocou JWT. Keďže je to sociálna sieť, tak jej súčasťou sú obrázky, ktoré sú manažované pomocou služby Cloudinary. Okrem klasickej SQL databázy je využívaná NoSQL databáza menom Elasticsearch. Na synchronizáciu dát medzi týmito dvoma databázami sa využíva Logstash. Na monitorovanie a získavanie metrík z aplikácie sa používa Prometheus a Grafana. Pre lokálny vývoj AWS backendu je využívaná technológia Localstack a príkazy, ktoré ponúka AWS CLI. S jednoduchým spustením týchto backendov a spolu s nimi aj pomocných služieb a technológií pomáha Docker, ktorý vytvorí pre každú časť aplikácie kontajner. Tieto kontajnery sa pomocou Docker Compose môžu spustiť súčasne. Hlavným cieľom je tieto implementácie porovnať z pohľadu náročnosti vývoju a podpory pre testovanie.

Klíčová slova sociálna sieť, technológie na tvorbu backendu, porovnávanie, Java, Spring, Quarkus, Micronaut, AWS, Google OAuth 2.0, JWT, React, Cloudinary, Docker, Elasitcsearch, Logstash, Prometheus, Grafana, AWS CLI, Localstack

Abstract

This bachelor thesis covers implementation of a backend application that serves as a simple social network. This backend is implemented using Java language in four different technologies Spring, Quarkus, Micronaut and AWS. It also offers login using Google OAuth 2.0 and a user maintained logged in using JWT. Since it is a social network, it includes images that are managed using Cloudinary service. In addition to the classic SQL database is used a NoSQL database called Elasticsearch. Logstash is used to synchronize data between the two databases. Prometheus and Grafana are used to monitor and obtain metrics from the application. For the local development of AWS backend is used Localstack and commands offered by AWS CLI. With help of Docker, which for each part of the application creates container, it is easy to run these backends and with them launch the auxiliary services and technologies. These containers can be launched simultaneously using Docker Compose. The main goal is to compare these implementations in terms of the complexity of development and support for testing.

Keywords social network, backend technologies, comparison, Java, Spring, Quarkus, Micronaut, AWS, Google OAuth 2.0, JWT, React, Cloudinary, Docker, Elasitcsearch, Logstash, Prometheus, Grafana, AWS CLI, Localstack

Seznam zkratek

AWS	Amazon Web Services
JWT	JSON Web Token
JWS	JSON Web Signature
JWE	JSON Web Encryption
MAC	Message Authentication Code
SDK	Software Development Kit
CRUD	Create, Read, Update and Delete
ORM	Object-Relational Mapping
API	Application Programming Interface
JPA	Java Persistence API
REST	Representational State Transfer
CLI	Command Line Interface
GUI	Graphical User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
SQL	Structured Query Language
JPQL	Jakarta Persistence Query Language

Úvod

V tejto modernej dobe s pribúdaním času vzniká množstvo aplikácií na zjednodušenie života ľudí. Ich cieľom môže byť čokoľvek, napríklad zábava, efektivita v práci, udržanie domácnosti, komunikácia alebo množstvo ďalších. Naopak pre zjednodušenie života programátorov sa vytvára veľa knižníc a technológií na tvorbu aplikácií, aby nemuseli vytvárať to, čo už niekto za nich vytvoril. Vďaka tomu je programátor efektívnejší a istejší v tom, že používaná knižnica alebo technológia má menšiu chybovosť, ako keby to robil on sám. To je dôsledok toho, že za tým stojí komunita ľudí a určité časové obdobie, počas ktorého sa zlepšovala funkčnosť, efektívnosť a kvalita kódu.

V tejto práci si vymedzíme túto širokú oblasť na tvorbu backendu, čo je serverová časť webovej aplikácie alebo softwarového programu, ku ktorej užívateľ nemá prístup a nevidí ju. Konkrétne sa pozrieme na to, že ako to môžeme implementovať v jazyku Java s pomocou vybraných technológií. Objasníme si, čo sú tie technológie vlastne zač a popíšeme si rozdiely medzi nimi. To všetko si ukážeme pri tvorbe jednoduchej aplikácie, ktorá bude fungovať ako sociálna sieť.

Z pohľadu začinajúceho programátora, ktorý nemá prehľad v tejto oblasti, je niekedy ťažké si vybrať technólogiu, ktorá by ponúkala také funkčnosti, aké potrebuje. Niektoré technológie sú nové, s krátkou históriou a s malou komunitou ľudí, čo má za dôsledok to, že nie je najľahšie nájsť to, čo programátor potrebuje a existujú aj prípady, kedy to tá daná technológia nepodporuje. Preto som sa rozhodol vytvoriť túto prácu, aby som týmto ľuďom poskytol odrazový mostík, ktorý by im pomohol vo výbere.

Túto tému som si vybral z toho dôvodu, že sa sám v tomto považujem za začiatočníka a väčšina článkov sa k tomuto vyjadrí veľmi stručne a bez praktickej ukážky. Taktiež usudzujem, že dokážem najlepšie predať tieto informácie programátorom v podobnej situácií.

Na začiatku si detailnejšie popíšeme ciele práce a všeobecný popis technológie na tvorbu backendu. Ďalej sa pozrieme na to, čo spomínaná aplikácia má spĺňať a čo všetko v nej bude implementované. Následne si ukážeme ako by sa dala táto aplikácia realizovať a otestovať v rôznych technológiách. V neposlednom rade si vysvetlíme rozdiely v implementáciách a moju vlastnú preferenciu.

Kapitola 1
Cieľ práce

Cieľom teoretickej časti práce je všeobecný popis technológií na tvorbu backendu v Jave, na čo slúžia a vysvetlenie základných pojmov, ktoré sa s touto prácou spájajú a ktoré budú často spomínané v práci. Taktiež tam bude vysvetlený výber technológií, ktoré budú v práci navzájom porovnávané a detailný popis aplikácie, ktorá bude slúžiť ako príklad využiteľnosti týchto technológií. Popíšeme si aj pomocné služby a technológie, ktoré v aplikácií budeme používať.

Cieľom praktickej časti práce sú samotné implementácie aplikácie v technológiách Spring, Quarkus, Micronaut a AWS. Popis implementácie backendu si rozdelíme na dôležité celky, ktorých súčasťou je aj nasadenie a spustenie. Popíšeme si knižnice, ktoré nám k dokončeniu aplikácie jednotlivé technológie ponúkajú a dôvod ich výberu. Potom bude následovať otestovanie jenotlivých backendov, kde si povieme aké testy využijeme a ako ich napísať. Porovnanie implementácií backendov, z pohľadu náročnosti vlastného vývoju a podpory pre testovanie, bude popísané v závere jednotlivých kapitol v podobe zhrnutia, pretože tie najdôležitejšie rozdiely budú popísané v samotnej realizácii.

Prínosom tejto bakalárskej práce je jednoduchý teoretický a aj praktický prehľad technológií na tvorbu backendu v Jave. Budú tu ukazané možnosti implementácie bežných problémov pri tvorbe webovej aplikácie a tak sa ktokoľvek môže riešením inšpirovať.

Kapitola 2
Technológie na tvorbu backendu

V tejto kapitole si vysvetlíme, prečo je dobré využívať technológie na tvorbu backendu a nie implementovať všetko sám úplne od začiatku. Tiež si vysvetlíme výber technológií, ktorý bol spomenutý v cieli práce.

2.1 Prečo využívať technológie?

Technológie a knižnice, nie len na tvorbu backendu, uľahčujú programátorom implementáciu, pretože nemusia programovať programy úplne od začiatku. Využivajú triedy, nástroje a moduly, ktoré ich problemý riešia a sú väčšinou ľahko konfigurovateľné, aby dokázali presne to, čo vyžadujú. Technológie, ktoré sa využívajú na tvorbu backendu pozostávajú z nástrojov a modulov. Väčšinou serverová časť webových aplikácií potrebuje napríklad databázu na ukladanie dát, vystavenie REST API (REST je cesta ako jednoducho vytvoriť, čítať, editovať alebo mazať informácie zo serveru, detailnejšie je to popísané v [1]), zabezpečenie a mnoho ďalšieho a s tým všetkým nám technológie dokážu pomôcť. Využívanie toho čo ponúkajú je aj o mnoho spoľahlivejšie, pretože veľa chýb sa časom opravilo, na ktoré programátor môže naraziť v prípade, ak si vybral implementáciu na vlastnú päsť.

2.2 Výber technológií

Keďže budeme vytvárať backend webovej aplikácie v jazyku Java, tak si potrebujeme vybrať technológie, ktoré sú využitelné v tomto jazyku. Tatkiež je nutné, aby ponúkali riešenia na vystavenie REST API, perzistenciu, autentizáciu, jednoduché testovanie a podobne.

Existuje množstvo technológií na podporu tvorby backendu v Jave ako napríklad Spring, Quarkus, Micronaut, AWS, Dropwizard, Google Web Toolkit, Struts, Play, Grails, Wicket a veľa iných.

Vyžadujeme, aby vybrané technológie boli stále vyvíjané a ich rok zverejnenia nebudeme brať do úvahy. Vyberieme si dve technológie, ktoré sú jedny z najpoužívanejších a dve menej využívané. Využívanosť vybraných technológií budeme čerpať zo stránky https://stackshare.io.

Náš výber sa skladá z technológií Spring, Quarkus, Micronaut a AWS. Spring (3,5 tisíc „stacks") je moja prvá technológia, s ktorou som sa stretol na tvorbu backendu, čiže ma zaujíma, že ako obstojí v porovnaní s ostatnými. Taktiež má dlhoročnú históriu a je jedna z najpoužívanejších technológií v dnešnej dobe. Micronaut (141 „stacks") a Quarkus (200 „stacks") sú pomerne mladé technológie, ktoré sa častokrát objavia na vrchole rebríčku v tejto sfére. Zaujali ma aj z toho dôvodu, že sa používaním podobajú Springu a prial som si ich navzájom porovnať. Ako posledné máme AWS, ktoré ma zaujalo tým, že je to najrozšírenejšia cloud platforma a s ktorou sa pracuje inak ako s ostatnými technologiami zmienenými vyššie. Chcel som zistiť ako funguje a v čom všetkom sa líši od predošlých. Využívanosť AWS sa líši od konkrétnej služby, ale pohybuje sa to v tisícoch „stacks".

V prípade záujmu o informácie, ktoré nebudú súčasťou tejto práce, ich môžete nájsť na stránkach Springu (https://spring.io), Quarkusu (https://quarkus.io), Micronautu (https://micronaut.io) a AWS (https://aws.amazon.com).

Kapitola 3
Popis aplikácie

V tejto kapitole si spíšeme, čo všetko by implementovaná aplikácia mala spĺňať. Začneme tým, že si povieme o akú aplikáciu vlastne ide a v akej oblasti sa bude pohybovať. Následne si predstavíme doménový model a detailnejšie si popíšeme jednotlivé entity, ktoré budú reprezentovať údaje, s ktorými bude aplikácia pracovať. Popíšeme si čo je endpoint, akými endpointami by mala aplikácia disponovať a pomocou čoho ich zabecpečiť tak, aby k ním mal prístup len užívateľ s potrebnými právami. Na konci si stručne popíšeme pomocné služby a technológie, ktoré pri implementácií budeme využívať, aby sme ich mali na jednom mieste a aby sme mohli sa na ne spätne ohľadať.

3.1 Oblasť aplikácie

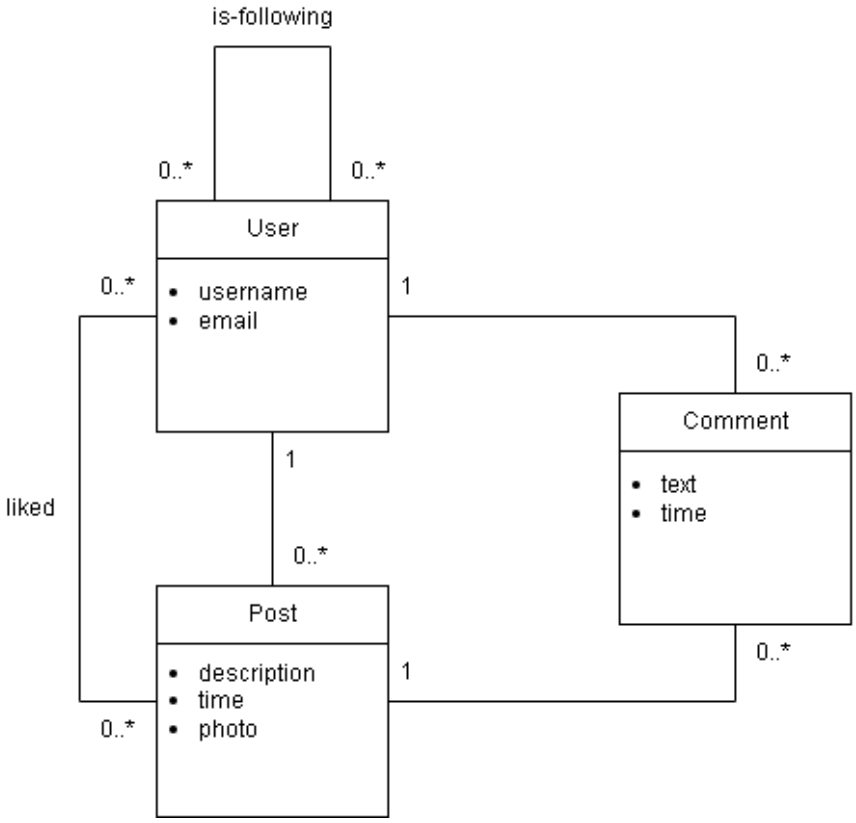
Aplikácia, ktorú budeme implementovať je inšpirovaná známou sociálnou sieťou Instagram. Čiže pôjde o aplikáciu, na ktorej si ľudia môžu udržovať kontakt medzi sebou v podobe zdieľania svojich zážitkov pomocou fotiek. Môžu si navzájom reagovať na svoje fotky a tak vedieť, čo sa okolo nich prebieha. Reagovať môžu pomocou komentárov a pre tých, ktorí to so slovami veľmi nevedia, bude dostupná možnosť označiť príspevok, že sa im páči. Na to, aby sa užívateľ mohol rýchlejšie dostať na profil iného užívateľa, o ktorého má záujem, tak bude dostupná možnosť sledovania. Pre užívateľov, ktorí by si chceli vyhľadať aj iných, ktorí nepatria do ich kruhu priateľov, bude poskytovaná možnosť hľadania podľa mena. Všetky potrebné dáta pre fungovanie aplikácie, budú ukladané v databáze.

3.2 Doménový model

V tejto podsekcií si predstavíme doménový model, ktorý sa nachádza na obrázku 3.1. Sú v ňom ukázané entity, s ktorými naša aplikácia bude pracovať. V tomto modeli sú ukázané aj všetky potrebné atribúty, vzťahy k ostnaným triedam a ich

kardinalita.

Obr. 3.1 Doménový model



3.2.1 Entita User

Táto entita reprezentuje registrovaného užívateľa. Tento užívateľ si môže vytvoriť ľubovoľný počet príspevkov a komentárov, kladne zareagovať na ktorýkoľvek príspevok a sledovať akýchkoľvek užívateľov. Atribúty sú zhrnuté v tabuľke 3.1.

Atribút	Popis
id	Unikátny identifikátor užívateľa
username	Meno užívateľa
email	Email užívateľa

Tabuľka 3.1 Entita User

3.2.2 Entita Post

Táto entita reprezentuje príspevok, ktorý si užívateľ vytvoril. Na príspevku nie je obmedzený počet komentárov a kladných reakcií užívateľov. Taktiež môžu existovať príspevky, ktoré nemajú ani jeden komentár a ani jednú kladnú reakciu. Súčasťou príspevku je popis, ktorý si užívateľ sám zvolí pri tvorbe a samotný obrázok, na ktorý má príspevok odkaz. Atribúty sú zhrnuté v tabuľke 3.2.

Atribút	Popis
id	Unikátny identifikátor príspevku
description	Popis príspevku
time	Čas vytvorenia príspevku
photo_url	Odkaz, kde sa obrázok nachádza
user_id	Identifikátor užívateľa, ktorému patrí tento príspevok

Tabuľka 3.2 Entita Post

3.2.3 Entita Follow

Táto entita reprezentuje sledovanie užívateľov. Musí mať vzťah s práve dvomi užívateľmi, kde jeden z nich je sledujúci a druhý je sledovaný. Atribúty sú zhrnuté v tabuľke 3.3.

Atribút	Popis
id	Unikátny identifikátor sledovania
user_id_who	Identifikátor užívateľa, ktorý začal sledovať užívateľa
user_id_whom	Identifikátor užívateľa, ktorý začal byť sledovaným

Tabuľka 3.3 Entita Follow

3.2.4 Entita Like

Táto entita reprezentuje kladnú reakciu užívateľa na príspevok. Vzťahuje sa presne k jednému uživateľovi a k jednému konkretnému príspevku. Atribúty sú zhrnuté v tabuľke 3.4.

Atribút	Popis
id	Unikátny identifikátor sledovania
user_id	Identifikátor užívateľa, ktorý túto kladnú reakciu vytvoril
post_id	Identifikátor príspevku, na ktorom sa reakcia ma prejaviť

Tabuľka 3.4 Entita Like

3.2.5 Entita Comment

Táto entita reprezentuje komentár užívateľa, ktorý sa vzťahuje k jedinému užívateľovi a k jedinému príspevku. Atribúty sú zhrnuté v tabuľke 3.5.

Atribút	Popis
id	Unikátny identifikátor sledovania
text	Text komentára
time	Čas vytvorenia komentára
user_id	Identifikátor užívateľa, ktorý vytvoril tento komentár
post_id	Identifikátor príspevku, na ktorom sa má zobraziť komentár

Tabuľka 3.5 Entita Comment

3.3 Endpointy aplikácie

Endpoint je webová adresa, na ktorú užívateľ môže poslať požiadavok, ktorý neskôr backend spracuje. V tomto backende budeme využívať protokol HTTP (protokol aplikačnej vrstvy na prenos hypermediálnych dokumentov, ako je HTML, detaijlenšie je to popísané v [2]) a REST API. V následujúcej podsekcií si ukážeme príklad endpointov pre prácu s entitou *User*.

3.3.1 Endpointy pre entitu User

Na tieto endpointy vypísané v tabuľke 3.6 môže užívateľ poslať požiadavok, aby dokázal pracovať s entitou User, ktorá je popísaná v tabuľke 3.1. Endpoint pre vytvorenie užívateľa nemusí byť vytvorený v prípade, ak sa užívateľ automaticky vytvorí po úspešnej autentizácii.

Absolútna cesta	HTTP metóda	Popis
/users	GET	Získanie všetkých užívateľov
/users	POST	Vytvorenie užívateľa
/users/{id}	GET	Získanie užívateľa
/users/{id}	DELETE	Vymazanie užívateľa
/users/{id}/editUsername	PUT	Zmena mena užívateľa
/users/byEmail?email	GET	Získanie užívateľa emailom
/users/search?prefix	GET	Získanie užívateľov, ktorí majú meno začínajúce sa zvoleným prefixom
/users/logged	GET	Získanie prihláseného užívateľa
/users/likedPost?post	GET	Získanie užívateľov, ktorým sa zvolený príspevok páči
/users/followers?user	GET	Získanie užívateľov, ktorí sledujú zvoleného užívateľa
/users/followers/count?user	GET	Získanie počtu užívateľov, ktorí sledujú zvoleného užívateľa
/users/following?user	GET	Získanie užívateľov, ktorí sú sledovaní zvoleným užívateľom
/users/following/count?user	GET	Získanie počtu všetkých užívateľov, ktorí sú sledovaní zvoleným užívateľom

Tabuľka 3.6 Endpointy pre entitu User

3.4 Zabezpečenie aplikácie

Aby užívateľ, ktorý nemá práva manipulovať s dátami, k ním nemal prístup, tak potrebujeme našu aplikáciu zabezpečiť. Existujé mnoho spôsobov ako to docieliť a najpoužívanejšie sú napríklad Basic (meno a heslo sú zakódované a zahrnuté v hlavičke), Bearer (zašifrovaný token, vytvorený serverom, ktorý užívateľ posiela v hlavičke) a OAuth 2.0 (protokol, ktorý poskytuje špecifické „authorization flows" pre aplikácie). [3] Na prihlásenie a registráciu budeme využívať kombináciu OAuth 2.0 a Bearer autentizácie. Konkrétne využijeme poskytovateľa Google a pre udržanie prihlásenia nam poslúži JWT, čo je Bearer token. V prípade, ak nepríhlasený užívateľ urobí požiadavok, tak budeme očakávať, že backend vráti kód 401 UNAUTHORIZED. V prípade, ak je príhlasený, ale nemá právo urobiť daný požiadavok, tak backend bude vracať kód 403 FORBIDDEN.

3.5 Pomocné služby a technológie

V tejto sekcií si popíšeme v skratke pomocné služby a technológie. Stručne budú popísané z toho dôvodu, že cieľom tejto práce nie je ich vysvetlenie, ale ich využitie a podpora na tvorbu backendu.

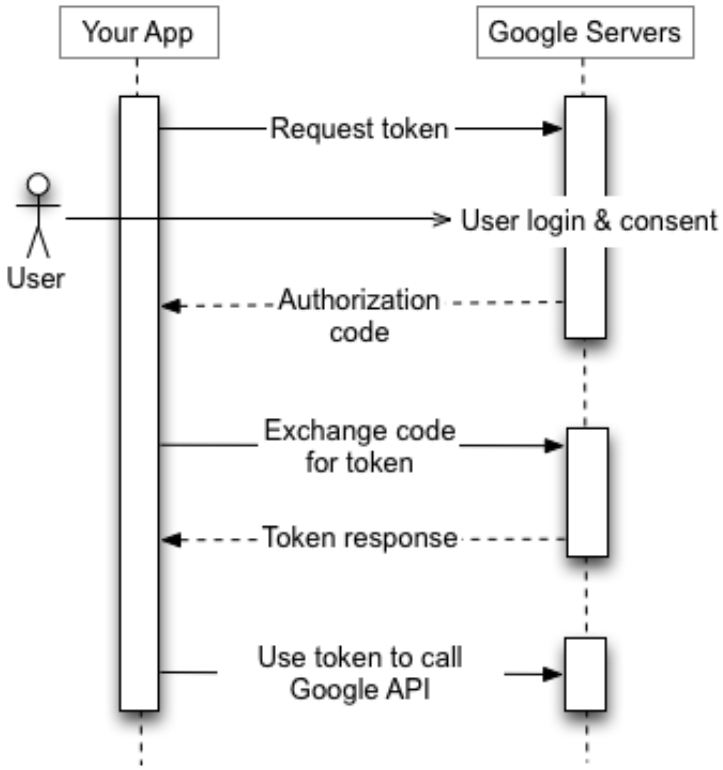
3.5.1 Zabezpečenie

V nasledujúcich podsekciách si popíšeme, čo budeme využívať na zabezpečenie našich backendov.

3.5.1.1 Google prihlásenie

Rozhrania Google API používajú na overenie a autorizáciu protokol OAuth 2.0. Google podporuje bežné scenáre OAuth 2.0, ako sú scenáre pre webový server, klienta, nainštalované aplikácie a aplikácie pre zariadenia s obmedzeným vstupom. [4] Túto možnosť prihlásenia sme si vybrali kvóli tomu, aby backend nemusel pracovať s prihlásovacími údajmi užívateľa. Priebeh autorizácie môžete vidieť na obrázku 3.2.

Obr. 3.2 Priebeh autorizácie Zdroj: [4]



3.5.1.2 JWT

Budeme žiadať, aby naša aplikácia bola bezstavová. To si ale žiada niečo, čo udrží užívateľa autentizovaného na viac požiadavkov ako len na jeden. Z toho dôvodu bol vybraný JWT. JWT je kompaktný, URL bezpečný spôsob na reprezentáciu kúskov informácií spájanými so subjektom, ktoré sa majú previesť medzi dvoma stranami. Informácie v JWT sú zakódované ako objekt JSON, ktorý sa používa ako prenášaný údaj JWS štruktúry alebo ako nešifrovaný text JWE štruktúry, ktorá umožňuje, aby boli informácie digitálne podpísané alebo integrita chránená pomocou MAC a/alebo šifrovaná. [5]

3.5.2 GUI - React

React umožňuje vytváranie interaktívnych používateľských rozhraní. Ponúka navrhnutie jednoduchého zobrazenia pre každý stav v aplikácii a React efektívne aktualizuje a vykreslí tie správne komponenty, keď sa údaje zmenia. [6] React sme si vybrali, pretože ako v definícií je napísané, tak aktualizuje komponenty pri zmene, kde komponenta môže byť čokoľvek. Taktiež React disponuje jednoduchou dokumentáciou, ktorá pomôže začiatočníkovi si rýchlo vytvoriť svojé prvé uživateľské rozhranie. Táto knižnica nám bude slúžiť na tvorbu frontendu, čo je viditeľna časť aplikácie. Tento frontend nám pomôže manuálne otestovať funkčnosť prihlásenia a aj iných endpointov backendu.

3.5.3 Práca s obrázkami - Cloudinary

Vo všetkých technológiách budeme na ukladanie fotiek využívať službu Cloudinary. Riešenie Programmable Media od Cloudinary nám umožňuje nahrávať obrázky a videá na cloud, transformovať, optimalizovať a doručovať ich do našej aplikácie prostredníctvom ľahko použiteľných rozhraní REST API. Cloudinary ešte k tomu ponúka Java SDK, ktoré obaľuje REST API a pridáva rôzne pomocné metódy na jednoduchšiu implementáciu. [7] Cloudinary sme si vybrali, pretože ako z definície plynie, tak poskytuje jednoduché riešenie na ukladanie obrázkov v jazyku Java a taktiež je ponúkaná zadarmo.

3.5.4 Nasadenie a spustenie - Docker

Docker je otvorená platforma pre vývoj, dodanie a spúšťanie aplikácií. Docker umožňuje oddeliť aplikácie od infraštruktúry, aby bolo možné rýchlo dodávať softvér. Poskytuje možnosť zabaliť a spustiť aplikáciu v slabo prepojenom izolovanom prostredí nazývanom kontajner. Izolácia a bezpečnosť umožňuje súčasne spúšťať viacero kontajnerov. Kontajnery obsahujú všetko potrebné na spustenie aplikácie, takže sa nemusí spoliehať na to, čo je aktuálne nainštalované. [8]

Túto službu využijeme pre spustenie aplikácie, databázi, frontendu, monitorovacích služieb a ďalších.

3.5.5 Rýchle vyhľadávanie užívateľov

V nasledujúcich podsekciách si ukážeme služby, ktoré využijeme na rýchle vyhľadávanie potrebných údajov v naších backendoch.

3.5.5.1 Elasticsearch

Elasticsearch je distribuovaný, RESTful vyhľadávací a analytický nástroj. Centrálne ukladá údaje na bleskurýchle vyhľadávanie, doladenú relevanciu a výkonné analýzy, ktoré sa dajú ľahko škálovať. [9] Túto službu využijeme pre rýchlejšie vyhľadávanie uživateľov, ktorým začína meno zvoleným prefixom v požiadavku.

3.5.5.2 Logstash

Na to aby Elasticsearch 3.5.5.1 bol naplnený dátami, s ktorými potrebujeme pracovať k rýchlejšiemu vyhľadávaniu užívateľov využijeme Logstash. Logstash je nástroj na zhromažďovanie údajov s otvoreným zdrojovým kódom s možnosťou „pipelining" v reálnom čase. Logstash dokáže dynamicky zjednotiť údaje z rôznych zdrojov a normalizovať údaje do cieľových umiestnení podľa výberu, čo je v tomto prípade Elasticsearch. [10] Príklad „pipeline", ktorá slúži na výber dát z databázy a posiela dáta do ELasticsearchu môžete vidieť vo výpise kódu 1. Detaily o tom ako sa sa „pipelines" vytvarajú si môže prečítať v [11].

3.5.6 Monitorovanie

V nasledujúcich podsekciách si ukážeme službu, ktorá nám pomôže s monitorovaním a zbieraním potrebných informácií. Ako druhú si ukážeme službu, ktorá nám tieto vyzbierané informácie dokáže vykresliť.

3.5.6.1 Prometheus

Na získavanie metrík našeho backendu využijeme Prometheus, kde metriky sú číselné merania, ktorých zmeny sa zaznamenávajú v priebehu času, napríklad čas spracovania požiadavku, použitá pamäť a podobne. Prometheus zhromažďuje a ukladá metriky ako údaje časových radov, teda informácie o metrikách sú uložené s časom, v ktorom boli zaznamenané. [12] Aby Prometheus dokázal zbierať metriky, potrebujeme ho správne nakonfigurovať a to si môžete preštudovať v [13].

3.5.6.2 Grafana

Na lepší prehľad získaných metrík pomocou Prometheus 3.5.6.1 použijeme službu Grafana. Grafana je „complete observability stack", ktorý umožňuje monitorovať a analyzovať metriky a logy. Umožňuje vyhľadávať, vizualizovať, upozorňovať a porozumieť údajom bez ohľadu na to, kde sú uložené. [14] Ako spustiť službu Grafana a vytvoriť „dashboard", ktorá vizualizuje metriky si môžete preštudovať v [15].

```
input {
    jdbc {
        jdbc_driver_library =>
        "/usr/share/jdbc/mysql-connector-java-8.0.21.jar"
        jdbc_driver_class => "com.mysql.jdbc.Driver"
        jdbc_connection_string => "jdbc:mysql://db:3306/instamini"
        jdbc_user => "zlackbra"
        jdbc_password => "password"
        schedule => "* * * * *"
        statement => "SELECT id_row AS id FROM User_log
                      WHERE activity = 'delete' AND time > :sql_last_value"
     }
}

filter {

}

output {
    elasticsearch {
        hosts => ["elasticsearch:9200"]
        action => delete
        document_id => "%{id}"
        index => "users"
    }
}
```

Výpis kódu 1 Príklad „pipeline" v Logstash

3.5.7 Vývoj AWS

V nasledujúcich podsekciách si ukážeme nástroj na manažovanie AWS služieb a technológiu, ktorá nám uľahčí lokálny vývoj AWS.

3.5.7.1 AWS CLI

AWS CLI je jednotný nástroj na správu služieb AWS. S jediným nástrojom na stiahnutie a konfiguráciu je možné ovládať viacero služieb AWS z príkazového riadku a automatizovať ich pomocou skriptov. [16] Tento nástroj budeme využívať s pomocou vyššie popísaného Dockeru.

3.5.7.2 Localstack

LocalStack poskytuje jednoducho použiteľnú testovaciu/mockovaciu technológiu pre vývoj cloudových aplikácií. Na lokálnom počítači spustí testovacie prostredie, ktoré poskytuje rovnaké funkcie a API ako skutočné cloudové prostredie AWS. [17] Kedže to nie je reálne AWS, tak niektoré príkazy spúšťané pomocou AWS CLI budú zjednodušené.

3.5.8 Analýza kódu - SonarQube

SonarQube je automatický nástroj na kontrolu kódu, ktorá spočíva v zisťovaní chýb, zraniteľností a „smells" kódu. [18] Tento nástroj využijeme na všetko, čo je v definícií popísané a vyvodíme z neho výsledky zo všetkých backendov, ktoré budú v tejto práci realizované, aby sme dokázali posúdiť, čo je nutné v kóde zlepšiť.

Služba/technologia	Backendové technológie
Google prihlásenie	Spring, Quarkus, Micronaut, AWS
JWT	Spring, Quarkus, Micronaut, AWS
React	Spring, Quarkus, Micronaut
Cloudinary	Spring, Quarkus, Micronaut, AWS
Docker	Spring, Quarkus, Micronaut, AWS
Elasticsearch	Spring, Quarkus, Micronaut, AWS
Logstash	Spring, Quarkus, Micronaut, AWS
Prometheus	Spring, Quarkus, Micronaut
Grafana	Spring, Quarkus, Micronaut
AWS CLI	AWS
Localstack	AWS
SonarQube	Spring, Quarkus, Micronaut, AWS

Tabuľka 3.7 Rozčlenenie využitia pomocných služieb a technológií

Kapitola 4
Realizácia

V tejto kapitole si ukážeme ako by sa backend našej aplikácie dal implementovať vo vybraných technológiách. Rozdelíme si to na sekcie, v ktorých sa zameriame na jednotlivé kúsky backendu a ďalej na podsekcie podľa technológií. Taktiež si zdôvodnime postup a iné možnosti riešenia.

4.1 Dáta

V tejto sekcií, si ukážeme ako budeme pristupovať k uloženým dátam v databáze. Samozrejme potrebujeme ich aj vytvárať a prípadne modifikovať. V Springu, Quarkuse a Micronaute vužívame Data Mapper Pattern. Data Mapper je vrstva softvéru, ktorá oddeľuje objekty v pamäti od databázy. [19]

4.1.1 Spring

Spring ponúka na jednoduchú implementáciu a konfiguráciu dátovej vrstvy Spring Data JPA. Spring Data JPA poskytuje podporu úložiska pre JPA. Uľahčuje vývoj aplikácií, ktoré potrebujú prístup k JPA zdrojom dát. [20] Súčasťou Spring Data JPA je jej predvolená implementácia Hibernate, ktorá ponúka ORM, ktoré slúži na namapovanie Java objektov na tabuľky databázy.

4.1.1.1 Entity

Ak chceme uviesť, že trieda je entita, tak potrebujeme danú triedu označiť anotáciou *@Entity*. Pomocou anotácie *@Table* môžme špecifikovať detaily tabuľky, na ktorú sa daná trieda namapuje.

Každá entita obsahuje atribúty, ktorých detaily môžme upresniť pomocou anotácie *@Column*. Taktiež musí obsahovať identifikátor, ktorý špecifikujeme pomocou anotácie *@Id* a je možné si zvoliť generačnú stratégiu pomocou *@GeneratedVa-*

lue. Každemú atribútu môžme zadať obmedzenia pomocou anotácií (*@NotBlank*, *@Size*, ...), aby sme dokázali chybnú entitu odhaliť.

Na mapovanie relácií používame anotácie *@OneToMany*, *@ManyToOne*, *@OneToOne*, *@ManyToMany* a *@JoinColumn*. Vo výpise kódu 2 môžme na riadkoch 23 a 24 vidieť príklad ako môžme vzťah dvoch entít zapísať v prípade, ak entita *Post* je vlastník. Na riadku 23 upresňujeme o aký vzťah ide a na riadku 24 špecifikujeme atribút pomocou ktorého ten vzťah vzniká. Ak *Post* nie je vlastníkom vzťahu ako môžme vidieť na riadkoch 27 a 30, tak špecifikujeme názov atribútu v anotácií *@OneToMany* pomocou ktorého je *Post* namapovaný k danému vzťahu. Môžme si taktiež všimnúť zvolený typ kaskády, ktorý hovorí, že v prípade ak *Post* bude vymazaný, tak sa odstránia aj entity, ktoré referujú na daný *Post*.

4.1.1.2 Repozitáre

Konkrétny repozitár vytvoríme ako rozhranie, ktoré rozširuje *JpaRepository*. *JpaRepository* je rozšírenie rozhrania *Repository* a disponuje funkčnosťami *CrudRepository* a *PagingAndSortingRepository*, čiže obsahuje implementované CRUD operácie, stránkovanie a zoraďovanie. [21] Pomocou anotácie *@Repository* dáme Springu vedieť o našej triede a že sa bude správať ako databázový repozitár. Ďalej *JpaRepository* očakáva typové parametre, kde prvý reprezentuje našu entitu a druhý typ identifikátora entity. *JpaRepository* sa takto postará o implementáciu základných operácií, ktoré sme zmienili vyššie. V prípade, ak by sme chceli dodať metódu, ktorú JpaRepository neponúka, tak Spring Data JPA ponúka dve možnosti.

Prvá z možností je pomocou názvu metódy, kde sa použijú podporované kľúčové slová a JPA to preloží do JPQL. Príklady si môžete pozrieť v dokumentácií [20].

Druhá možnosť je ukázaná vo výpise kódu 3 začínajúc na riadku 90 alebo 97. Ako môžme vidieť, tak je použitá anotácia *@Query*, do ktorej môžme napísať JPQL alebo natívny SQL požiadavok. [22]

4.1.2 Quarkus

Quarkus ponúka na implementáciu dátovej vrstvy Hibernate ORM. Je to štandardná implementácia JPA a ponúka celú šírku ORM. [23] V tomto prípade na implementáciu repozitárov je potrebné využívať triedu *EntityManager* a výtvarať požiadavky pomocou nej, čo je nepraktické. Z toho dôvodu využijeme Hibernate ORM Panache, ktorá ponúka rozhranie *PanacheRepositoryBase* obsahujúce CRUD operácie a veľa ďalších pomocných metód.

4.1.2.1 Entity

Definícia entít sa od Springu nelíši a pripomenúť si ho môžete v sekcií 4.1.1.1.

```
@Entity
@Table(name = "Post")
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @NotBlank
    @Size(max = MAX_LENGTH_DESCRIPTION_AND_TEXT)
    @Column(nullable = false)
    private String description;

    @NotNull
    @Size(max = MAX_LENGTH_PHOTO_URL)
    @Column(name = "photo_url", nullable = false)
    private String photoUrl;

    @NotNull
    @Column(columnDefinition = "TIMESTAMP", nullable = false)
    private LocalDateTime time;

    @NotNull
    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @OneToMany(mappedBy = "post", cascade = CascadeType.REMOVE)
    private List<Like> likes;

    @OneToMany(mappedBy = "post", cascade = CascadeType.REMOVE)
    private List<Comment> comments;
}
```

Výpis kódu 2 Entita Post v Springu

4.1.2.2 Repozitáre

Ako sme zmienili vyššie vytvoríme náš repozitár ako rozhranie, ktoré rozširuje *PanacheRepositoryBase*. Označíme naše rozhranie pomocou anotácie *@ApplicationScoped*, ktorá znamená, že pre aplikáciu sa používa jedna inštancia „beany" (objekt manažovaný technológiou) a zdieľa sa medzi všetkými injekčnými bodmi. Inštancia sa vytvorí „lazily", čiže po vyvolaní metódy nad „client proxy". [24]

```
1   @Repository
2   public interface PostRepository extends JpaRepository<Post, Integer> {
3       @Query(
4           value = "SELECT * FROM Post as p
5           WHERE p.user_id = :idUser ORDER BY p.time DESC",
6           nativeQuery = true
7       )
8       List<Post> findAllPostsOfUser(Integer idUser);
9
10      @Query(
11          value = "SELECT COUNT(*) FROM Post as p WHERE p.user_id = :idUser",
12          nativeQuery = true
13      )
14      Integer findCountOfAllPostsOfUser(Integer idUser);
15  }
```

Výpis kódu 3 Repozitár pre entitu Post v Springu

PanacheRepositoryBase vyžaduje dva typové parametre ako v prípade Springu, čiže prvý parameter reprezentuje našu entitu a druhý typ identifikátora entity. Na vytvorenie metódy s vlasným požiadavkom môžeme využiť metódy, ktoré nám ponúka *PanacheRepositoryBase* a ich použitie môžete vidieť vo výpise kódu 4 na riadkoch 5 a 10.

```
1   @ApplicationScoped
2   public class PostRepository implements
3       PanacheRepositoryBase<Post, Integer> {
4
5       public List<Post> findAllPostsOfUser(Integer idUser) {
6           return list("SELECT p FROM Post as p WHERE p.user.id = ?1
7           ORDER BY p.time DESC", idUser);
8       }
9
10      public Long findCountOfAllPostsOfUser(Integer idUser) {
11          return find("SELECT p FROM Post as p WHERE p.user.id = ?1
12          ORDER BY p.time DESC", idUser).count();
13      }
14  }
```

Výpis kódu 4 Repozitár pre entitu Post v Quarkuse

4.1.3 Micronaut

Micronaut ponúka na implementáciu dátovej vrstvy Hibernate JPA. Tak ako v Quarkuse aj v tomto prípade na implementáciu repozitárov je potrebné využívať triedu *EntityManager*, preto využijeme ďalší ponukaný modul s názvom Data Hibernate JPA, v ktorej využijeme rozhranie *CrudRepository* obsahujúca CRUD operácie a anotáciu *@Query* s rovnakou funkčnosťou ako v Springu.

4.1.3.1 Entity

Tak ako v Springu aj v Quarkuse, tak aj tu je definícia entít totožná, viď sekcia 4.1.1.1. Jediný rozdiel je ten, že Micronaut nepodporuje kaskádu, ktorá by v prípade vymazania entity, vymazala aj entity, ktoré na ňu referujú. Vymazanie týchto entít je potrebné robiť manuálne.

4.1.3.2 Repozitáre

Repozitár vytvoríme ako rozhranie rozširujúce *CrudRepository*. *CrudRepository* presne tak isto ako v prípade predošlých technológií vyžaduje dva typové parametre, kde prvý reprezentuje entitu a druhý typ identifikátora entity. Pomocou anotácie *@Repository* informujeme Micronaut, že ide o repozitár dát. Na vytvorenie vlastného požiadavku, ktorým rozšírované rozhranie nedisponuje použijeme vyššie zmienenú anotáciu *@Query*, do ktorej tak ako v Springu môžme napísať JPQL alebo natívny SQL požiadavok. [25]. Kód je takmer totožný s výpisom 3až na jeden rozdiel a tým je rozhranie, ktoré náš repozitár rozširuje.

4.1.4 AWS

AWS pristupuje k dátam ináč a na ich uloženie nám ponúka viacero možností. Na implementáciu backendu využijeme Amazon DynamoDB, pretože je to NoSQL databáza, ktorá ma vysoký výkon a taktiež je ponúkaná zadarmo. Ostatné možnosti si môžete preštudovať v [26]. AWS ponúka Amazon RDS, ktorá ponúka relačné databázy, aby to bolo, čo najpodobnejšie s predošlými implementáciami dátovej vrstvy. No táto služba narozdiel od Amazon DynamoDB nie je stále zadarmo a k použitiu Amazon DynamoDB v jazyku Java bolo nájdené väčšie množstvo dokumentácie.

4.1.4.1 Pripojenie k DynamoDB

Ak chceme mať prístup k databáze a robiť potrebné modifikácie, potrebujeme si inicializovať triedy, ktoré nám to sprístupnia. Na to potrebujeme poznať endpoint a región. Zvolený endpoint bude referovať na spustený Localstack 3.5.7.2. Príklad inicializácie môžete vidieť vo výpise kódu 5.

```
String serviceEndpoint = "http://localstack:4566";
String region = "eu-central-1";

AmazonDynamoDB amazonDynamoDB = AmazonDynamoDBClientBuilder
        .standard()
        .withEndpointConfiguration(
            new AwsClientBuilder.EndpointConfiguration(
            serviceEndpoint,
            region))
        .build();

DynamoDB dynamoDB = new DynamoDB(amazonDynamoDB);
```

Výpis kódu 5 Pripojenie k DynamoDB

4.1.4.2 Práca s DynamoDB klientom

DynamoDB a *AmazonDynamoDB* klienti ponúkajú množstvo metód, pomocou ktorých dokážeme pracovať s databázou. Dokumentáciu všetkého, čo novší klient *DynamoDB* dokáže si môžete preštudovať v [27]. Vo výpise kódu 6 je jednoduchý príklad na získanie všetkých údajov *Post* užívateľa pomocou identifikátora a názvu tabuľky. Na to potrebujeme vytvoriť objekt typu *ScanRequest*, v ktorom definujeme podmienky vyhľadávania a to takým spôsobom, že do metódy *withFilterExpression* zadáme podmienku a v nasledujúcich metódach definujeme jednotlivé parametre, aby nedošlo ku využitiu rezervovaných slov priamo v podmienke. Následne získame objekt typu *ScanResult*, v ktorom nájdeme všetky výsledky.

V prípade, ak chceme nájsť počet týchto údajov ako v príklade repozitárov v predošlých technológiách, tak pri tvorbe objektu *ScanRequest* pridáme metódu s argumentom následovne *withSelect(Select.COUNT)* a potom namiesto jednotlivých údajov v objekte *ScanResult* sa bude nachádzať ich počet.

4.1.5 Zhrnutie dátovej vrstvy

Spring a Micronaut sú praktický totožné a to aj z toho dôvodu, že Micronaut Data, modul ktorý sme využili, je inšpirovaný modulom Spring Data. Líši sa len v tom, že Micronaut Data nemá „runtime model", neprekláda definíciu metódy na databázový požiadavok, nepoužíva reflexiu alebo „runtime proxies" a kontroluje repozitár v kompilačnom kóde. [25] Taktiež Micronaut neponúka kaskádu v prípade vymazania entity narozdiel od Quarkusu a Springu.

Quarkus ponúka, tak ako Spring a Micronaut rozhranie, ktoré nám pomohlo s implementáciou repozitáru, jediný rozdiel je ten, že sa využívajú metódy k definícii požiadavkov a nie anotácie. Entity sme definovali rovnako ako v Springu

```
1   ScanRequest scanRequest = new ScanRequest()
2           .withTableName("Post")
3           .withFilterExpression("#u = :userId")
4           .withExpressionAttributeNames(of("#u", "userId"))
5           .withExpressionAttributeValues(of(
6                   ":userId", new AttributeValue("1")
7           ));
8
9   ScanResult scanResult = amazonDynamoDB.scan(scanRequest);
10  List<Post> posts = new ArrayList<>();
11  for (Map<String, AttributeValue> item : scanResult.getItems()) {
12      Post post = new Post(item.get("id").getS(),
13          item.get("description").getS(), item.get("photoUrl").getS(),
14          item.get("time").getS(), item.get("userId").getS());
15      posts.add(post);
16  }
```

Výpis kódu 6 Príklad dotazovania sa na DynamoDB

a v Quarkuse.

V AWS sme nepotrebovali definíciu entít, pretože sme nevyužívali namapovanie tried na tabuľky. Využilii sme NoSQL DynamoDB, ku ktorej sme mali prístup pomocou klienta, ktorý ponúka AWS. Žiadny dotatočný repozitár nebolo potrebné implementovať.

Implementácia dátovej vrstvy bola jednoduchá vo všetkých technológiách. Využívala sa len dokumentácia, v ktorej bolo všetko potrebné napísané a detailne vysvetlené.

4.2 Elasticsearch

V tejto sekcií si ukážeme akým spoôsobom je možné implementovať Elasticsearch 3.5.5.1. Tento nástroj využijeme na rýchle vyhľadávanie užívateľov podľa prefixu mena.

4.2.1 Spring, Quarkus, Micronaut

Všetky technológie v názve podsekcie majú modul, ktorý podporuje Elasticsearch a v skratke si vysvetlíme ako to používať. Použitie v týchto technológiách sa minimálne líši, čiže si ukážeme jeden príklad, ktorý vykonáva vyhľadávanie užívateľov podľa prefixu mena. V zjednodušenom výpise kódu 7 môžete vidieť ako sa používa v Quarkuse. Na riadku 1 definujeme index, čo je niečo ako databáza

v relačných databázach. [28] Premenná *client* na riadku 9 je typu *RestHighLevelClient*, ktorý sa injektuje v konštruktore a počas injektovania sa inicializuje pomocou údajov napísaných v konfiguračnom súbore. Na riadku 13 máme premennú *searchHits*, ktorá reprezentuje výsledok a dokážeme z neho získať všetkých nájdených užívateľov.

```
1   String usersIndex = "exampleUsersIndex";
2
3   QueryBuilder matchQueryBuilder = QueryBuilders.prefixQuery("username",
4       "examplePrefix").caseInsensitive(true);
5   SearchSourceBuilder sourceBuilder = newSearchSourceBuilder()
6       .query(matchQueryBuilder);
7   SearchRequest searchRequest = new SearchRequest(usersIndex);
8   searchRequest.source(sourceBuilder);
9   SearchResponse searchResponse;
10  try {
11      searchResponse = client.search(searchRequest,RequestOptions.DEFAULT);
12  } catch (IOException e) {
13      e.printStackTrace();
14  }
15  SearchHit[] searchHits = searchResponse.getHits().getHits();
```

Výpis kódu 7 Príklad použitia technológie Elasticsearch v Quarkuse

4.2.2 AWS

Z dôvodu nedostatku času Elasticsearch nie je v AWS implementovaný, ale ukážeme si ako by to bolo možné implementovať. AWS ponúkal službu Amazon Elasticsearch Service, ktorá bola premenovaná na Amazon OpenSearch Service, ktorá stále podporuje Elasticsearch. Čiže pre implementáciu rýchleho vyhľadávania by sme mohli použiť Amazon OpenSearch Service. OpenSearch je distribuovaný, „open source" vyhľadávací a analytický balík, ktorý sa používa na širokú škálu prípadov použitia ako je monitorovanie aplikácií v reálnom čase, analýza protokolov a vyhľadávanie na webových stránkach. OpenSearch poskytuje vysoko škálovateľný systém na poskytovanie rýchleho prístupu a odozvy na veľké objemy údajov. OpenSearch bolo pôvodne odvodené z Elasticsearch 7.10.2. [29] Pre prístup k dátam na vyhľadávanie stačí vytvoriť HTTP požiadavok v tvare 8, kde doména smeruje k našej OpenSearch službe a parameter q je náš požiadavok. Tento požiadavok prehľadáva všekty premenné a indexy pre výraz „house". Taktiež je možné využiť klienta, ktorý nám s poslaním požiadavku a so spracovaním odpovede pomôže. Práca s ním je popísaná v [30].

```
1    GET https://search-my-domain.eu-central-1.es.amazonaws.com/_search?q=house
```

Výpis kódu 8 Príklad požiadavku na prístup k dátam v Amazon OpenSearch Service

4.2.3 Zhrnutie Elasticsearch

Implementácia podpory Elasticsearch sa podobne ako pri dátovej vrstve v technológiách Spring, Quarkus a Micronaut nelíšila. Všetky ponúkajú rovnaké rozhrania a triedy na úspešné dotazovanie potrebných dát. V konfiguračnom súbore sa jedine líši syntax a umiestenie.

V AWS konfigurácia bola jednoduchšia, pretože sme si nemuseli sami získavať server Elasticsearch. Ten je uložený na AWS cloude a nám stačilo len vytvoriť doménu, poprípade index, ktorý smeroval priamo k naším dátam. V kóde stačilo vytvoriť požiadavok alebo využiť ponúkaného klienta.

4.3 REST API

V tejto sekcií bude popísané ako je možné implementovať REST API v jednotlivých technológiách. Príklad endpointov, ktoré by mala každá technológia vystavovať bola popísaná v sekcií 3.3.

4.3.1 Spring

Na implementáciu REST API v Springu využívame modul Spring Framework Web, ktorý nám ponúka anotácie na označenie metód, aby na ne boli namapované HTTP požiadavky. Triedu označíme anotáciou *@RestController*, aby ju Spring registroval ako „beanu" a ako triedu, ktorá vystavuje REST API. Na označenie triedných metód konkrétne využívame anotácie *@GetMapping*, *@PostMapping*, *@PutMapping* a *@DeleteMapping*, kde do ich zátvoriek môžeme definovať absolútnu cestu, typ konzumovaných dát, typ produkovaných dát, parametre požiadavku a podobne. Ďalej používame anotácie *@RequestParam* (namapovanie parametra uvedeného v URL), *@PathVariable* (namapovanie premennej uvedenej v URL) a *@RequestBody* (namapovanie tela požiadavku). [31] V prípade vytvorenia entity *Post*, je potrebné v tele požiadavku poslať obrázok. Naša príslušná metóda to musí vedieť namapovať a s tým nám pomôže anotácia *@ModelAttribute* a jej použitie si môžete pozrieť v [32].

V prípade nejakej chyby počas spracovania dokážeme vyhodiť výnimku pomocou triedy *ResponseStatusException*, ktorá prijíma statusový kód a správu. Táto výnimka sa neskôr preloží na odpoveď s uvedeným kódom a správa je zahrnutá len vtedy, ak je to povolené v konfiguračnom súbore.

V prípade POST a PUT metód je posielané s požiadavkom aj jej telo, ktoré musí dodržovať určité obmedzenia. Každé telo správy je v kóde reprezentované ako trieda, ktorej atribúty sú označené obmedzujúcimi anotáciami (*@NotBlank*, *@Size*, . . .) podobne ako pri kóde entity *Post* v 2. Pre validáciu použijeme anotáciu *@Value* pred parametrom v metóde, ktorý chceme validovať.

```
1   @RestController
2   public class PostController {
3       private final PostService postService;
4
5       @Autowired
6       public PostController(PostService postService) {
7           this.postService = postService;
8       }
9
10      @GetMapping("/posts/{id}")
11      public PostDTO byID(@PathVariable Integer id) {
12          Optional<PostDTO> optionalPostDTO;
13          optionalPostDTO = postService.findById(id);
14
15          if (optionalPostDTO.isPresent()) {
16              return optionalPostDTO.get();
17          }
18          else {
19              throw new ResponseStatusException(HttpStatus.NOT_FOUND,
20              "No such post.");
21          }
22      }
23  }
```

Výpis kódu 9 Ukážka implementácie REST API endpointu v Springu

4.3.2 Quarkus

Triedu, ktorá bude spravovať endpointy ku prístupu k našej entite označíme anotáciou *@ApplicationScoped*, ktorá bola vysvetlená v 4.1.2.2 a anotáciou *@Path*, kde do jej zátvorky definujeme základnú absolútnu cestu. Na označenie triedných metód konkrétne využívame anotácie *@GET*, *@POST*, *@PUT* a *@DELETE* pre špecifikáciu HTTP metódy, anotáciu *@Produces* pre špecifikáciu typu produktu, anotáciu *@Consumes* pre špecifikáciu typu konzumu a anotáciu *@Path*, kde do jej zátvorky môžme pridať dodatočnú cestu. Ďalej používame anotácie *@PathPa-*

ram (namapovanie parametra uvedeného v URL), *@QueryParam* (namapovanie premennej uvedenej v URL) a pri mapovaní tela požiadavku nemusíme nič explicitne uvádzať. Toto zmienené a ešte viac nám ponúka balíček javax.ws.rs. [33] V prípade vytvorenia entity *Post*, je potrebné v tele požiadavku poslať obrázok. Na namapovanie tela, ktoré obsahuje obrázok potrebujeme modul Quarkus Resteasy Multipart a použiť anotáciu *@MultipartForm* pred parametrom v metóde.

V prípade nejakej chyby počas spracovania dokážeme vyhodiť rôzne výnimky z balíčku javax.ws.rs, ktoré sa neskôr odchytia a preložia na odpoveď s odpovedajúcim kódom a s uvedenou správou, ktorú yadávame ako argument vyhadzovanej výnimke.

Validácia tela správy sa implementuje rovnako ako je popísané na konci podsekcie 4.3.1.

```
1   @Path("/posts")
2   @ApplicationScoped
3   public class PostController {
4
5       private final PostService postService;
6
7       @Inject
8       public PostController(PostService postService) {
9           this.postService = postService;
10      }
11
12      @Path("{id}")
13      @GET
14      @Produces(MediaType.APPLICATION_JSON)
15      public PostDTO byID(@PathParam("id") Integer id) {
16          Optional<PostDTO> optionalPostDTO;
17          optionalPostDTO = postService.findById(id);
18
19          if (optionalPostDTO.isPresent()) {
20              return optionalPostDTO.get();
21          }
22          else {
23              throw new javax.ws.rs.NotFoundException("No such post.");
24          }
25      }
26  }
```

Výpis kódu 10 Ukážka implementácie REST API endpointu v Quarkuse

4.3.3 Micronaut

Na implementáciu REST API v Micronaute využívame modul Micronaut Http, ktorý taktiež ponúka anotácie na označenie metód, aby na ne boli namapované HTTP požiadavky. Triedu označíme anotáciou *@Controller* a to jej zátvorky môžeme napísať základnú absolutnú cestu. Na označenie triedných metód využívame anotácie *@Get*, *@Post*, *@Put* a *@Delete*, kde do ich zátvoriek môžeme definovať dodatočnú absolútnu cestu, typ konzumovaných dát, typ produkovaných dát a podobne. Ďalej používame v metódach anotácie *@PathVariable* (namapovanie parametra uvedeného v URL), *@QueryValue* (namapovanie premennej uvedenej v URL) a *@Body* (namapovanie tela požiadavku). Na namapovanie obrázka, pri tvorbe entity *Post*, využívame triedu *CompletedFileUpload* a parameter reprezentujúci objekt na vytvorenie entity *Post* je rozdelený na všetky potrebné parametre.

V prípade nejakej chyby počas spracovania dokážeme vyhodiť výnimku pomocou triedy *HttpStatusException*, ktorá prijíma statusový kód a správu. Táto výnimka sa neskôr preloží na odpoveď s uvedeným kódom a so správou.

Validácia tela správy sa implementuje rovnako ako je popísané na konci podsekcie 4.3.1. Jedinú výnimku tvorí tvorba entity *Post* a to z toho dôvodu, že sú jednotlivé parametre metódy rozdelené. V tomto prípade vytvoríme v tejto jednej metóde jeden objekt, ktorý sa neskôr manuálne validuje pomocou triedy *Validator*.

4.3.4 AWS

REST API sa v prípade AWS nevytvára v kóde, ale vytvoríme ho pomocou AWS CLI 3.5.7.1 príkazu *aws apigateway create-rest-api* a pomocou služby Amazon API Gateway. Amazon API Gateway je plne spravovaná služba, ktorá vývojárom uľahčuje vytváranie, publikovanie, údržbu, monitorovanie a zabezpečenie API. Rozhrania API fungujú ako „predné dvere“ pre aplikácie na prístup k údajom, logike alebo funkciám z backendových služieb, čo v našom prípade je AWS Lambda. [34] AWS Lambda je bezserverová, udalosťami riadená služba, ktorá umožňuje spúšťať kód pre prakticky akýkoľvek typ aplikácie alebo backendovej služby bez poskytovania alebo správy serverov. [35]

Po jeho vytvorení ďalej vytvoríme zdroje pomocou AWS CLI príkazu *aws apigateway create-resource*, ktoré sa viažu na absolutnú cestu, následne pridáme k jednotlivým zdrojom HTTP metódy (príkaz *aws apigateway put-method*) a integráciu (príkaz *aws apigateway put-integration*), v ktorej zadefinujeme s akou AWS Lambda funkciou je zdroj a HTTP metóda spájaná. Na nasadenie REST API použijeme príkaz *aws apigateway create-deployment*. [36][37]

Validácia posielaných dát v AWS Lambda funkciách z dôvodu nedostatku času nebola implementována, ale popíšeme si ako by sme to mohli uskutočniť. Jedna z možností je manuálna kontrola v AWS Lambda funkciách. Kedže predošlé riešenie nie je veľmi praktické, tak AWS ponúka validátor požiadavku popísaný v [38]. V tejto dokumentácií nevyužívajú AWS CLI, ale AWS CLI ponúka ekviva-

lentné príkazy na dosiahnutie toho istého výsledku. Čiže pomocou AWS CLI by sme to mohli implementovať tak, že vytvoríme model pomocou príkazu *aws apigateway create-model* [39], kde popíšeme štruktúru tela, následne vytvoríme validátor požiadavku pomocou *aws apigateway create-request-validator* [40] a výsledné identifikátory modelu a validátora špecifikujeme ako argumenty príkazu na pridanie HTTP metódy ku zdroju. No v tejto možnosti nie je možnosť napríklad na kontrolu dĺžky textového údaju, čiže niektoré náležitosti je potrebné aj tak validovať manuálne.

4.3.5 Zhrnutie REST API

Implementácia REST API sa až na syntax veľmi nelíšila v technológiách Spring, Quarkus a Micronaut. Čo je aj logické, pretože pri REST API je nutné definovať základné veci akou je napríklad absolutná cesta endpointov. Všetky tri ponúkali potrebné anotácie, ako aj jednoduché vyvolanie vynímky, ktorú za nás tieto technológie preložili k odpovedajúcej odpovedi užívateľovi. Okrem syntaxu sa ešte líšili v spracovaní obrázku v tele správy a hlavne v Micronaute, kde sa mi nepodarilo nájsť spôsob ako obaliť obrázok spolu s ďalšími informáciami do triedy, pre jednoduchšiu validáciu.

Narozdiel od týchto troch technológií sa REST API v AWS neimplementovalo v kóde, ale pomocou AWS CLI príkazov. V kóde je to prehľadnejšie, všekty potrebné nastavenia endpointu a čo očakáva môžeme vidieť na jednom mieste, kde v AWS na rôzne validácie tiel požiadavkov, metódy a podobne, existuje príkaz, ktorý treba spustiť. AWS ponúka aj konzolu, kde to všetko sa dá vytvoriť, ale tak ako pri príkazoch, tak aj tu to nie je zhromáždené na jednom mieste a treba sa preklikávať medzi rôznymi nastaveniami. Ale samozrejme to závisí na preferencii programátora.

4.4 Autentizácia a autorizácia

V tejto sekcií si ukážeme ako zabezpečiť naše vytvorené backendy. Všeobecný popis toho, čo vyžadaujeme sme si popísali v 3.4. Pripomíname si, že na príhlásenie budeme využívať príhlásenie pomocou Google poskytovateľa 3.5.1.1 a na udržanie prihlásenia JWT 3.5.1.2.

4.4.1 Spring

S bezpečnosťou nám pomáha modul Spring Security. Spring Security poskytuje autentizáciu a autorizáciu pre Java aplikácie. Je to štandard pre zabezpečenie aplikácií na báze Spring. [41] Taktiež využijeme modul Spring Boot Starter OAuth2 Client na implementáciu Google prihlásenia 3.5.1.1 a knižnicu JSON Web Token Support For The JVM na implementáciu JWT 3.5.1.2.

```
1   public class GetPostByIdHandler implements
2       RequestHandler<APIGatewayProxyRequestEvent,
3           APIGatewayProxyResponseEvent> {
4
5       private final DynamoDB dynamoDB = InstanceModule.dynamoDB;
6
7       public APIGatewayProxyResponseEvent handleRequest(
8           final APIGatewayProxyRequestEvent event,
9           final Context context) {
10
11          Map<String, String> headers = new HashMap<>();
12          headers.put("Content-Type", "application/json");
13
14          APIGatewayProxyResponseEvent response =
15              new APIGatewayProxyResponseEvent()
16              .withHeaders(headers);
17          try {
18              Table table = dynamoDB.getTable(TableName.postTable);
19
20              Map<String, String> pathParameters = event.getPathParameters();
21
22              Item item = table
23                  .getItem(new PrimaryKey("id", pathParameters.get("id")));
24
25              if (item == null) {
26                  return response.withStatusCode(404);
27              }
28              return response.withStatusCode(200)
29                  .withBody(item.toJSONPretty());
30          } catch (final Exception e) {
31              return response.withStatusCode(500).withBody("{}");
32          }
33      }
34  }
```

Výpis kódu 11 Ukážka AWS Lambda funkcie

Ako prvé je potrebné nakonfigurovať OAuth2 Client v konfiguračnom súbore *application.yml*. Potrebujeme získať *client-id* a *client-secret*, ktoré si dokážeme vygenerovať podľa na adrese https://console.cloud.google.com. Taktiež je nutné špecifikovať, aké informácie budeme vyžadovať po prihlásení užívateľa. Výsledok je ukázaný vo výpise kódu 12.

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id:
              743755262714-l7ltj7o82enkqtgb9at53fnlcitfo83h
              .apps.googleusercontent.com
            client-secret: DmaRkmrDSz2cip9I5dRzIKvf
            scope:
              - email
              - profile
```

Výpis kódu 12 Konfigurácia OAuth2 Client v Springu

Ďalším krokom je vytvorenie triedy, ktorá dedí z triedy *WebSecurityConfigurerAdapter*. Táto vytvorená trieda nám slúži na konfiguráciu bezpečnosti, v ktorej popisujeme napríklad aké endpointy sú chránené, konfiguráciu OAuth2 prihlásenia a podobne.

Po týchto nastaveniach máme dostupné prihlásenie pomocou Google na endpointe */oauth2/authorization/google*. Pri vytvorení autorizačného požiadavku, ktorý bude smerovaný na Google poskytovateľa, za účelom vyplnenia prihlasovacích údajov užívateľom, si uložíme v triede *OAuth2AuthorizationRequestRepository* do „cookies" presmerovaciu URI, ak sa nachádza v premennej pri prihlasovacom požiadavku na náš backend, ktorá slúži na presmerovanie po úspešnom prihlásení. Po úspešnom prihlásení sa zavolá metóda *onAuthenticationSuccess* našej triedy *OAuth2LoginSuccessHandler*, v ktorej sa zo získaných informácií z Google poskytovateľa nájde registrovaný užívateľ v našej databáze. V prípade, ak takýto užívateľ neexistuje, tak ho v databáze vytvoríme.

Ako posledný krok úspešného príhlásenia je vytvorenie JWT pomocou emailu užívateľa a pomocou vyššie spomenutej knižnice. V kóde máme vytvorenú triedu *JwtUtil*, v ktorej sú implementované potrebné operácie s JWT. Vytvorený JWT vrátime užívateľovi ako premennú v URI, ktorú využívame na presmerovanie užívateľa.

Na kontrolu JWT pri požiadavkoch implementujeme *JwtRequestFilter*, v ktorom pomocou triedy *JwtUtil* validujeme token a získavame z neho email, ktorý bol použitý na vytvorenie JWT. Po získaní emailu sa nájde v databáze užívateľ spájaný s týmto emailom a využije sa pri definovaní kontextu bezpečnosti, aby sme k nemu mali prístup pri autorizácií.

Na to aby užívateľ nemohol zasahovať do údajov, ktoré patria niekomu inému,

je potrebná autorizácia. Tú implementujeme tak, že v metóde, ktorá spracuváva požiadavky posielané na konkrétny endpoint, pridáme ako parameter triedu *Principal*. Tento parameter bude injektovaný pri požiadavku a bude obsahovať prihláseného užívateľa, ktorého sme pridali do bezpečnostného kontextu. Teraz nám len zostáva skontrolovať, či užívateľ má právo manipulovať s údajmi, s ktorými sa daný endpoint spája. Príklad metódy na vymazanie užívateľa môžete vidieť vo výpise kódu 13. Úlohou triedy *AuthorizationUtil* je porovnanie identifikátorov dvoch užívateľov a v prípade, ak sa nezhodujú, tak je vyhodená výnimka *ResponseStatusException* s kódom 403.

```
1   @DeleteMapping("/users/{id}")
2   public void delete(@PathVariable Integer id, Principal principal) {
3       AuthorizationUtil.checkIfAuthorizedOrElseThrow(principal, id,
4               "Can't delete profile, which does not belong to you.");
5       try {
6           userService.deleteById(id);
7       } catch (NotFoundException e) {
8           throw new ResponseStatusException(HttpStatus.NOT_FOUND,
9           e.getMessage());
10      }
11  }
```

Výpis kódu 13 Autorizácia pri mazaní užívateľa v Springu

4.4.2 Quarkus

Quarkus ponúka na implementáciu autentizácie OpenID Connect. OpenID Connect je jednoduchá vrstva identity nad protokolom OAuth 2.0. Klientom umožňuje overiť identitu užívateľa na základe autentizácie vykonanej autorizačným serverom, ako aj získať základné informácie o užívateľovi. [42] Pre prípadné presmerovanie je potrebné si uložiť presmerovaciu URI napríklad v podobe „cookies". Na to sa potrebujeme dostať do autorizačného toku kódu, ale v dokumentácií a v návodoch Quarkusu sa mi spôsob na to nepodaril nájsť. Z toho dôvodu si proces autorizácie implementujeme sami pomocou knižníc ponúkaných spoločnosťou Google.

Na prístup ku Google poskytovateľovi, potrebujeme tak ako v Springu získať hodnoty *client-id* a *client-secret*, ktoré využijeme pri jednotlivých požiadavkoch. V celom procese, ktorý si popíšeme, nám pomáha implementovaná trieda *LoginService*, v ktorej sú všetky potrebné metódy.

Vystavíme endpoint */oauth2/authorization/google* ako pri tvorbe REST API v Quarkuse 4.3.2, na ktorom sa pri požiadavku vytvorí autorizačný požiadavok

a v ňom „cookie", ktorý obsahuje presmerovaciu URI, ktorá slúži na presmerovanie po úspešnom prihlásení. Súčasťou autorizačného požiadavku je ďalšia presmerovacia URI, ktorá slúži na presmerovanie od Google poskytovateľa po spracovaní tohoto požiadavku. Súčasťou tejto URI je absolútna cesta */login/oauth2/code/google*.

Aby sme presmerovanie od Google poskytovateľa dokázali spracovať, tak vystavíme ďalší endpoint */login/oauth2/code/google*, kde spracujeme odpoveď a získame potrebné informácie o užívateľovi, pomocou ktorých môžeme uživateľa registrovať a prihlásiť.

Ako posledný krok úspešného príhlásenia je vytvorenie JWT pomocou emailu užívateľa a pomocou knižnice JSON Web Token Support For The JVM a zaslať ho užívateľovi. V kóde máme vytvorenú triedu *JwtUtil*, v ktorej sú implementované potrebné operácie s JWT. Vytvorený JWT vrátime užívateľovi ako premennú v URI, ktorú využívame na presmerovanie užívateľa po úspešnom prihlásení, ktorú sme mali uloženú v „cookies".

Kedže sme sa rozhodli implementovať celý proces sami, tak potrebujeme triedu, ktorá manažuje prístup k endpointom. Z toho dôvodu si vytvoríme triedu *SecurityJwtInterceptor*, ktorá nám zároveň bude slúžiť na kontrolu JWT. Táto trieda filtruje požiadavky takým spôsobom, že ak metóda reprezentujúca endpoint nie je označená anotáciou *PermitAll*, tak je automaticky chránená a dostupná len prihláseným užívateľom. Na validáciu JWT využijeme triedu *JwtUtil* a získavame pomocou nej z JWT email. Po získaní emailu sa nájde v databáze užívateľ spájaný s týmto emailom a využije sa pri definovaní kontextu bezpečnosti, aby sme k nemu mali prístup pri autorizácií podobne ako v Springu.

Autorizáciu implementujeme tak, že v metóde, ktorá spracuváva požiadavky posielané na konkrétny endpoint, pridáme ako parameter triedu *SecurityContext* a pred tento parameter anotáciu *@Context*, ktorá slúži na injektovanie informácií. Zvyšok autorizácie je podobný implementácií v Springu, zmienená na konci podsekcie 4.4.1.

4.4.3 Micronaut

Na implementáciu autentizácie a autorizácie využíjeme modul Micronaut Security. Micronaut Security je plne vybavené a prispôsobiteľné bezpečnostné riešenie pre aplikáciu. [43] Konkrétne využijeme Miconaut Security OAuth2 a Micronaut Security JWT.

Je potrebné nakonfigurovať OAuth2 v konfiguračnom súbore *application.yml*. K tomu potrebujeme získať *client-id* a *client-secret*. Taktiež je nutné špecifikovať, aké informácie budeme vyžadovať po prihlásení užívateľa, autorizačnú URL poskytovateľa, URL na získanie tokenu od poskytovateľa, prihlásovaciu URI a URI, ktorá sa využije na presmerovanie od poskytovateľa po spracovaní autorizačného požiadavku. Výsledok je ukázaný vo výpise kódu 14.

Prístup k endpointom by mali mať len prihlasení užívatelia. To uskutočníme tak, že nad definiciou triedy, ktorá tieto endpointy vystavuje pridáme anotáciu

```
micronaut:
  security:
    oauth2:
      clients:
        google:
          client-id:
            743755262714-l7ltj7o82enkqtgb9at53fnlcitfo83h
            .apps.googleusercontent.com
          client-secret: DmaRkmrDSz2cip9I5dRzIKvf
          scopes:
            - email
            - profile
          authorization:
            url: https://accounts.google.com/o/oauth2/auth
          token:
            url: https://oauth2.googleapis.com/token
      login-uri: /oauth2/authorization/google
      callback-uri: /login/oauth2/code/google
```

Výpis kódu 14 Konfigurácia OAuth2 v Micronaute

@Secured(SecurityRule.IS_AUTHENTICATED).

Po týchto nastaveniach máme dostupné prihlásenie pomocou Google na endpointe */oauth2/authorization/google*. Na uloženie presmerovacej URI do „cookies", ktorá slúži na presmerovanie užívateľa po úspešnom prihlásení bolo obtiažne nájsť riešenie. Na uskutočnenie tohoto problému využijeme Proxy Pattern. Proxy Pattern umožňuje vytvoriť sprostredkovateľa, ktorý funguje ako rozhranie k inému zdroju, pričom zároveň skrýva zložitosť komponenty. [44] Čiže si vytvoríme triedu *CustomOAuthClient*, ktorá implementuje rozhranie *OauthClient* a obsahuje atribút typu *DefaultOauthClient*, čo je implementácia Micronaut Security. Ak sa zavolá naša implementácia pre získanie autorizačného požiadavku, tak to delegujeme na skutočnú implementáciu a k výsledku pridáme do „cookies" presmerovaciu URI. Po úspešnom prihlásení implementovaná trieda *GoogleAuthenticationMapper* v metóde *createAuthenticationResponse* získa informácie o prihlasovanom užívateľovi a vráti odpoveď autentizácie, ktorá sa ochytí v triede *CustomOAuthLoginHandler* v metóde *loginSuccess*. V tejto metóde sa registruje užívateľ, ak je to potrebné, vytvorí sa JWT pomocou triedy *JwtTokenGenerator*, ktorú nám Micronaut Security ponúka a vráti sa odpoveď s presmerovaním na URI, ktorá bola uložená v „cookies", spolu s JWT ako premennou.

Keďže používame JWT priamo z modulu Micronaut Security, tak jeho validáciu nemusíme implementovať, pretože Micronaut Security to urobí za nás.

Implementácia autorizácie v metódach je totožná s tou v Springu, ktorá je popísaná v závere podsekcie 4.4.1.

4.4.4 AWS

Z dôvodu nedostatku času, táto časť backendu v AWS nebola implementována, ale pozrieme sa na to, čo AWS ponúka.

Jednou z možností je vytvoriť REST API zdroj, ktorý by slúžil na prihlásenie a bol by prepojený s Lambda funkciou, v ktorej by sme robili požiadavky na Google poskytovateľa bez pomoci iných služieb, podobne ako v Quarkuse 4.4.2. Táto možnosť nie je nutná a to z dôvodu, že AWS ponúka službu AWS. Amazon Cognito poskytuje autentizáciu, autorizáciu a správu používateľov pre webové a mobilné aplikácie. Používatelia sa môžu prihlásiť priamo pomocou používateľského mena a hesla alebo prostredníctvom tretej strany, ako je Facebook, Amazon, Google alebo Apple. [45] Detailný popis vytvorenia autentizácie v Amazon Cognito môžete nájst v [46] a popis príkazov v [47].

Ako u predošlých technológií, tak aj tu je nutné získať *client-id* a *client-secret*. To, kde ich využijeme si ukážeme v nasledujúcich odstavcoch.

Vysvetlíme si celý proces konfigurácie tejto služby a pri jednotlivých krokoch si napíšeme AWS CLI príkazy, pomocou ktorých je možné danú vec vytvoriť.Začneme vytvorením „user pool", pomocou ktorého sa užívatelia budú môcť prihlásiť. Ten sa pomocou AWS CLI vytvorí pomocou príkazu *aws cognito-idp create-user-pool*. Pri vytváraní je nutné uviesť jeho meno.

Pokračujeme vytvorením poskytovateľa identity, čo je v našom prípade Google. Tu využijeme hodnoty *client-id*, *client-secret* a taktiež potrebujeme upresniť aký rozsah informácií užívateľa vyžadujeme (v našom prípade stačí email a profil). Pomocou AWS CLI príkazu *aws cognito-idp create-identity-provider* a uvedení všetkých zmienených argumentov vytvoríme poskytovateľa identity.

K „user pool" potrebujeme vytvoriť doménu, na ktorú užívateľ môže posielať požiadavok na prihlásenie. To dokážeme pomocou príkazu *aws cognito-idp create-user-pool-domain*, ktorý ako argumenty očakáva doménové meno a identifikátor „user pool".

Ako ďalšie potrebujeme vytvoriť „app client", v ktorom dokážeme priradiť poskytovateľa k „user pool". To dokážeme AWS CLI príkazom *aws cognito-idp create-user-pool-client*, ktorý ako argumenty očakáva meno vytvaraného klienta a identifikátor „user pool". Taktiež si v argumente môžeme zvoliť presmerovaciu URL, kde po úspešnej autentizácií bude užívateľ presmerovaný a jej súčasťou bude JWT ako premenná.

Na to, aby užívateľ bol pri prvom prihlásení vytvorený v DynamoDB použijeme Lambda „trigger", v ktorom môžeme pridať vlastnú logiku po tom, čo Amazon Cognito overí používateľa. [48] Cognito pošle Lambda funckii potrebné informácie na vytvorenie užívateľa. To dokážeme pomocou príkazu *aws cognito-idp update-user-pool*, kde musíme upresniť, v akom „user pool" to chceme nastaviť.

Následne, aby sme dokázali chrániť naše endpointy, je potrebné v API Gateway vytvoriť autorizátora. Vytvoríme ho pomocou príkazu *aws apigateway create-authorizer*, kde zadáme identifikátor REST API, naše zvolené meno, typ autorizátora (v našom prípade *COGNITO_USER_POOLS*) a zdroj identity, čo môže byť napríklad hlavička v požiadavku menom „Authorization". V tejto hlavičke budeme očakávať JWT, ktorý bude pri požiadavku kontrolovaný pomocou služby Cognito. Pri vytvaraní metód zdrojov v 4.3.4 už len stačí pridať autorizátora. [49][50]

Ako posledné nám zostáva implementácia autorizácie. Jedna z ponúkaných možností je získanie informácií užívateľa v Lambda funkcii z JWT, ktorý bol poslaný v hlavičke požiadavku a skontrolovanie, či tento užívateľ má právo meniť údaje, o ktoré požiadal. Druhá možnosť je, že pri vytváraní integrácie REST API zdroja s Lambda funkciou v 4.3.4, špecifikujeme atribúty, ktoré by sme potrebovali preniesť z „user pool" do Lambda funkcie a tie skontrolovať ako v prvej možnosti. [50]

4.4.5 Zhrnutie autentizácie a autorizácie

V Springu sa autentizácia a autorizácia implementovala pomerne jednoducho, všetky potrebné náležitosti sa dali nájsť v dokumentácií alebo v rôznych článkoch. Spring ponúkal rôzne rozhrania, ktoré sme si mohli ľubovoľne prispôsobiť ako aj nakonfigurovať celú bezpečnosť nášho backendu.

V Quarkuse nebol nájdený spôsob pomocou, ktorého by sa dala vytvoriť podobná implementácia ako v Springu. Síce ponúka modul OpenID Connect, no chýbalo tam možnosť vlastnej špecifikácie toho, čo sa presne kde a kedy má robiť. To sme potrebovali pri vytvorení užívateľa a aj uloženia presmerovacej URI do „cookies". Z toho dôvodu sa zvolila implementácia pomocou požiadavkov, ktoré sme si sami spracovávali.

V Micronaute bol podobný problém ako v Quarkuse, čiže problém nájsť časť kódu, ktorú by sme si mohli prispôsobiť. Nakoniec bol nájdený spôsob, kde pomocou anotácií a zdedenia správnych rozhraní sme mohli docieliť ukladanie užívateľa po registrácií ako aj uloženie presmerovacej URI. Implementácia JWT bola jednoduchšia ako v predošlých technológiách z toho dôvodu, že jedinú vec, o ktorú sa bolo potrebné postarať bolo vytvorenie JWT pri prihlásení a o zvyšok sa Micronaut postaral pomocou vlastnej implementácie.

Vzhľadom k tomu, že v predošlých technológiách bolo potrebné vytvoriť pomerne veľké množstvo kódu pri tejto implementácií, tak v AWS sme k tomu potrebovali len pár AWS CLI príkazov alebo pár preklikov v ich konzole. Tento prístup mi prišiel až prekvapivo jednoduchý, pretože programátorovi si v podstate stáči len zvoliť poskytovateľa, pridať autorizáciu k REST API a zvyšok sa Amazon Cognito postará.

4.5 Nasadenie a spustenie

V tejto sekcií si ukážeme ako spustiť vytvorené backendy spolu s ostatnými pomocnými službami a technológiami. S tým nám pomôže služba Docker popísaná v 3.5.4.

4.5.1 Spring, Quarkus, Micronaut

Nasadenie a spustenie týchto troch technológií v názve podsekcie a ich pomocných služieb/technológií sa nelíši. Pri ich spustení využijeme Docker, ktorý je dostupný na inštaláciu na adrese https://docs.docker.com/get-docker a Docker Compose, ktorý nám pomáha so spustením viacerých kontajnerov súčasne, dostupný na inštaláciu na adrese https://docs.docker.com/compose/install. Po inštalácii je potrebné byť v koreňovej zložke projektu a spustiť shell skript v tvare *bash ./pick-framework.sh name*, kde za *name* je potrebné napísať meno technológie, ktorú si prajeme spustiť (micronaut/spring/quarkus). V prípade, že už aplikácia bola raz spustená v inej technológii a chceme si ju zmeniť na inú, tak po napísaní predošlého príkazu je potrebné napísať príkaz *docker-compose build instamini*. Posledný príkaz, ktorý spustí náš backend a aj pomocné služby a technológie vrátane frontendu je *docker-compose up*. Po úspešnom spustení bude REST API dostupné na adrese http://localhost:8080.

Konfigurácia, ktorú využíva príkaz *docker-compose up* sa nachádza v koreňovej zložke v súbore *docker-compose.yml* a jej skrátenu verziu môžete vidieť vo výpise kódu 15. Každá služba/technológia uvedená v tomto konfiguračnom súbore má uvedený parameter *build*, v ktorom je cesta k *Dockerfile*, v ktorom sú napísané podrobnosti na spustenie danej služby/technológie. Ukážku Dockerfile, ktorý patrí Springu môžete vidiet vo výpise kódu 16. Ak máte záujem si o tom preštudovať viac, tak na adrese https://docs.docker.com je dostupná dokumentácia.

4.5.2 AWS

Ako sme si už spomínali, tak na lokálny vývoj používame Localstack 3.5.7.2. Na jeho spustenie využívame taktiež Docker a Docker Compose (adresy na inštaláciu sú spomenuté v 4.5.1). Je potrebné, aby sme mali nainštalovaný nástroj *jq*, kde návod na inštaláciu je dostupný na adrese https://stedolan.github.io/jq/download. Po inštalácii je potrebné byť v koreňovej zložke projektu. Na skompilovanie AWS Lambda funkcií je potrebné spustiť skript *bash ./scripts/code/lambdas_build_zip.sh*. Následne môžme spustiť samotný Localstack pomocou príkazu *docker-compose up localstack*. Pre prípravu backendu je potrebné vytvoriť tabuľky reprezentujúce entity, Lambda funkcie a REST API v Localstacku, čo dosiahneme spustením skriptu *bash ./scripts/aws/setup_aws.sh*, ktorý pracuje so spomenutým AWS CLI. Po úspešnom dokončení príkazu sú do konzoly vypísané URL, na ktoré sa môžu

```
version: "3.3"

services:
  db:
    container_name: instamini_mysql
    build: ./mysql
    restart: always
    ports:
      - "3307:3306"
    volumes:
      - instamini-db-v:/var/lib/mysql
    restart: on-failure

  instamini:
    container_name: instamini_spring
    build: ./instamini_spring
    ports:
      - "8080:8080"
    depends_on:
      - "db"
    restart: on-failure

volumes:
  instamini-db-v:
```

Výpis kódu 15 Skrátena verzia docker-compose.yml

```
FROM amazoncorretto:11

RUN mkdir /instamini_spring
WORKDIR /instamini_spring/
COPY . ./
RUN ./gradlew build

EXPOSE 8080

CMD java -jar /instamini_spring/build/libs/instamini_spring-0.0.1-SNAPSHOT.jar
```

Výpis kódu 16 Ukážka Dockerfile Springu

vytvárať požiadavky. V prípade implementácie autentizácie a autorizácie, príkazy popísané v podsekcií 4.4.4 by museli byť zahrnuté v skripte *setup_aws.sh*.

V poslednom spustenom skripte sa využívajú príkazy na tvorbu tabuliek *aws dynamodb create-table* a na tvorbu Lambda funkcií *aws lambda create-function*, kde ako argument uvádzame skompilovaný kód, ktorý bol výstupom príkazu na skompilovanie všetkých Lambda funckií. Konkrétny popis týchto príkazov aj veľa iných si môžete preštudovať v dokumentácií AWS CLI [37]. Súčasťou skriptu sú aj príkazy popísané v 4.3.4. Aj na spustenie AWS CLI využívame Docker Compose.

Tie isté príkazy, zahrnuté v skripte *setup_aws.sh*, môžeme použiť aj na skutočný AWS cloud, no je potrebné sa autorizovať. AWS ponúka aj konzolu s GUI, kde to všetko sa dá ekvivalentne vytvoriť.

4.5.3 Zhrnutie nasadenia a spustenia

Nasadenie a spustenie aplikácií v Springu, Quarkuse a Micronaute je totožné. To je docielené tým, že je to spúštané pomocou služby Docker a Docker Compose. Prispieva k tomu aj to, že sa v týchto technológiách využívajú rovnaké pomocné služby a technológie a tiež rovnaký zostavovací nástroj Gradle.

Narozdiel od týchto troch technológií je v AWS potrebné vytvoriť väčšinu vecí pomocou AWS CLI príkazov alebo AWS konzoly. Ako sme si ukázali, tak na to nám slúži skript, ktorý všetky príkazy spustí. Narozdiel od lokálneho vývoju, ktorý bol ukázaný v tejto práci, tak sa na nasadenie nevyužije Localstack, ale priamo AWS cloud. Výhoda AWS cloudu je tá, že pri nasadení je hňed k dispozícií URL, na ktorú môžme robiť požiadavky a nie len na lokálnej sieti.

V nasadení a spustení preferujem Spring, Quarkus a Micronaut, pretože aj pri takejto malej aplikácii sa v AWS nahromadilo veľké množstvo príkazov, pri ktorých môže človek urobiť malé chyby a častokrát ich nie je jednoduché nájsť.

4.6 Monitorovanie

V tejto sekcií si ukážeme ako môžme jednotlivé vytvorené backendy monitorovať. V nasledujúcich podsekciách sa bude objavovať služba Prometheus, ktorú sme si popísali v podsekcií 3.5.6.1.

4.6.1 Spring, Quarkus, Micronaut

Popis monitorovania backendov, vytvorenými pomocou týchto technológií som spojil do jedného, pretože konfigurácia je veľmi podobná. Stačí, že pridáme závislosť na ich špecifické implementácie služby Prometheus 3.5.6.1 a metriky sú možné nájsť na ich prevolenej absolútnej ceste. Kedže Spring bola prvá implementácia, tak sa pokusíme nastaviť absolútnu cestu v Quarkuse a Micronaute totožnú s tou v Springu (/actuator/prometheus).

V Quarkuse si stačí zvoliť absolútnu cestu v konfiguračnom súbore ako môžme vidieť vo výpise kódu 17.

```
quarkus.micrometer.export.prometheus.path=/actuator/prometheus
```

Výpis kódu 17 Zmena absolútnej cesty služby Prometheus v Quarkuse

V Micronaute je potrebné vytvoriť kontrolér, ktorý vráti metriky pomocou triedy *PrometheusMeterRegistry*. Príklad môžete vidieť vo výpise kódu 18..

```
@RequiresMetrics
@Controller("/actuator/prometheus")
public class PrometheusController {
    private final PrometheusMeterRegistry prometheusMeterRegistry;

    public PrometheusController(
        PrometheusMeterRegistry prometheusMeterRegistry) {
        this.prometheusMeterRegistry = prometheusMeterRegistry;
    }

    @Get
    @Produces("text/plain")
    public String metrics() {
        return prometheusMeterRegistry.scrape();
    }
}
```

Výpis kódu 18 Zmena absolútnej cesty služby Prometheus v Micronaute

4.6.2 AWS

Ak by sme chceli využívať službu Prometheus ako v predošlých technológiách, tak AWS ponúka Amazon Managed Service for Prometheus. Pretože túto službu je potrebné nakonfigurovať, tak pre uľahčenie implementácie si vyberieme službu Amazon CloudWatch, do ktorej sa automaticky posielajú metriky a logy z iných služieb. Amazon CloudWatch je služba monitorovania a správy, ktorá poskytuje údaje a použiteľné prehľady pre AWS, hybridné a lokálne aplikácie a zdroje infraštruktúry. [51] Táto služba je ľahko prístupná v AWS konzole a taktiež je dostupná pomocou AWS CLI.

4.6.3 Zhrnutie monitorovania

Monitorovanie v Springu, Quarkuse a Micronaute, je veľmi podobné a to z toho dôvodu, že sa využívaju rovnaké služby. No pri vybraní si vlastnej absolútnej cesty, kde by boli dostupné metriky, bol problém v Micronaute, kde sme si museli špecifikovať vlastný endpoint a zakázať ten predvolený. Samozrejme je to len pár riadkov kódu, čo vôbec problém nebol.

V AWS sa automaticky logy a metriky posielajú do služby Amazon CloudWatch. Táto možnosť mi zo všetkých štyroch technológií prišla najpríjemnešia na použitie a to z jednoduchého dôvodu, že som nemusel nič konfigurovať. Taktiež táto služba ponúka rôzne grafy a štatistiky ako v prípade Grafany, ktorá bola využitá v Springu, Quarkuse a Micronaute.

4.7 Dokumentácia a komunita

Na konci realizácie si spíšeme niečo o dokumentácii a komunite, pretože je to najhlavnejší zdroj informácií k práci s novou technológiou.

V prípade implementácie Springu, bolo všetko potrebné nájditeľné v rôznych dokumentáciách a návodoch, ktoré samotný Spring ponúkal. Dokumentácia je veľmi prehľadná a osobne mi veľmi vyhovovala. V prípade nejakého problému bolo možné nájsť rôzne fóra, kde sa stále našlo riešenie a to z toho dôvodu, že veľa backend vývojárov v Jave práve túto technológiu využívajú a aj preto, že Spring má dlhú históriu a to už od roku 2004. [52]

Quarkus na rozdiel od Springu je mladá technológia, kde prvé zverejnenie bolo v roku 2019. [53] Dokumentácia a návody dokázali pokryť základy, ale v prípade prispôsobenia funkčností, ktoré Quarkus ponúka, sa programátor musí väčšinu času vynájsť sám. Z dôvodu toho, že táto technológia je nová, tak medzi komunitou nie je až tak rozšírená a to znamená, že aj vyhľadávanie riešení špecifických problémov nie je najľahšie.

Micronaut podobne ako Quarkus je tiež mladá technológia z roku 2018. [54] No v tomto prípade dokumentácia a návody dostačovali na implementáciu nášho backendu. Aj v prípade konkrétnejších problémov bolo možné nájsť riešenie na rôznych fórach. No napriek tomu Spring mi v tomto stále vyhovoval viac.

AWS podobne ako Spring je v tejto oblasti dlhšie a to konkrétne od roku 2006. [55]. Dokumentácia a návody ma veľmi prekvapili, na implementáciu tohoto backendu nebolo potrebné hľadať riešenia na rôznych fórach. Na jediné téma som dokázal nájsť viacero riešení na daný problém pomocou AWS CLI príkazov ako aj s využitím AWS konzoly. AWS je najrozšírenejšia cloud platforma, čiže pri tomto výbere sa programátor nestratí.

4.8 Zhrnutie

V tejto sekcií si ukážeme tabuľku, ktorá zhrňuje všetky implementované časti takým spôsobom, že bude uvedená sekcia a k nej napísané technológie, ktoré boli z môjho pohľadu najjednoduchšie na implementáciu. Tabuľku možete vidieť v 4.1.

Sekcia	Technológie
Dáta	Spring, Quarkus, Micronaut
Elasticsearch	AWS
REST API	Spring, Quarkus, Micronaut
Autentizácia a autorizácia	AWS
Nasadenie a spustenie	Spring, Quarkus, Micronaut
Monitorovanie	AWS
Dokumentácia a komunita	Spring, AWS

Tabuľka 4.1 Preferencia technológií

Kapitola 5
Testovanie

V tejto kapitole si ukážeme akým spôsobom môžme jednotlivé backendy otestovať. Ukážeme si aké knižnice na testovanie jednotlivé technológie ponúkajú a ich použitie.

Na testovanie využijeme Unit testovanie, ktoré slúži na testovanie malého kusu kódu a integračné testovanie, ktoré testuje viacero komponent súčasne. Vo všetkých technológiach sa využije framework JUnit 5 a v Unit testoch sa využije Mockito. Mockito slúži na tvorbu falošných objektov, ktorým môžme zadefinovať ako sa správať v prípade, že je metóda tohoto objektu zavolaná a taktiež skontrolovať počet očakavaných zavolaní. Pri testoch využívame pomocné triedy na kontrolu očakavaných a skutočných dát, aby sme zredukovali opakovanie kódu.

V prípade Springu, Quarkusu a Micronautu využijeme H2 databázu, ktorá sa nám hodí na testovanie bez využitia produkčnej databázy. V technológií AWS z rovnakého dôvodu použijeme Testcontainers Dynalite, čo je klon Amazon DynamoDB.

5.1 Spring

V Springu využijeme na testovanie Spring Boot Starter Test, ktorý obsahuje aj zmienený JUnit a Spring Security Test. V integračných testoch využijeme triedu MockMvc, ktorá nám pomôže pri testovaní endpointov.

5.1.1 Unit testy

Triedu, obsahujúcu Unit testy, označíme *@RunWith(MockitoJUnitRunner.class)*, kde trieda *MockitoJUnitRunner* umožňuje používanie Mockito anotácií. [56] Príklad Unit testu môžete vidieť vo výpise kódu 19.

```
@Test
void findById() {
    Integer userId = 1;
    Integer postId = 2;
    User user = new User("user1", "user1", null, null, null, null, null);
    Post post = new Post("test", "test", localDateTime, user);
    ReflectionTestUtils.setField(user, "id", userId);
    ReflectionTestUtils.setField(post, "id", postId);

    Comment comment = new Comment("text", localDateTime, user, post);

    Integer commentId = 1;
    ReflectionTestUtils.setField(comment, "id", commentId);

    Mockito.when(commentRepository.findById(commentId))
        .thenReturn(Optional.of(comment));

    Optional<CommentDTO> commentDTOOptional =
    commentService.findById(commentId);

    if (commentDTOOptional.isPresent()) {
        CommentDTO commentDTO = commentDTOOptional.get();
        AssertCommentsUnitUtil.assertComments(comment, commentDTO,
            userService, postService);
    } else {
        throw new AssertionFailedError();
    }
    Mockito.verify(commentRepository,Mockito.times(1))
        .findById(commentId);
}
```

Výpis kódu 19 Unit test v Springu

5.1.2 Integračné testy

Triedu, obsahujúcu integračné testy, označíme anotáciami *@SpringBootTest* (vytváranie kontextu testovanej aplikácie), *@AutoConfigureMockMvc* (konfigurácia MockMvc) a *@Sql* (vkladanie testovacích dát do databázy pred testom a ich mazanie po teste). Implementujeme triedu *MockJwtAuthenticationUtil*, v ktorej zadefinujeme falošnému objektu type *JwtUtil* ako sa ma správať pri autentizácii. Atribút *commentsInDatabase* testovanej triedy obsahuje všetky komentáre uložené v databáze a na otestovanie práv manipulovania s dátami sú v triede dvaja užívatelia, kde jeden z nich reprezentuje prihláseného a druhý neprihláseného užívateľa. Príklad integračného testu môžete vidieť vo výpise kódu 20.

```
1     @Test
2     void all200() throws Exception {
3         mockJwtAuthenticationUtil.mockReturns();
4
5         ResultActions result = mockMvc.perform(get("/comments")
6                         .header("Authorization", "Bearer " +
7                         mockJwtAuthenticationUtil.getMockedToken()))
8                     .andExpect(status().isOk())
9                     .andExpect(content().contentType("application/json"));
10
11        for (int i = 0; i < commentsInDatabase.size(); i++) {
12            AssertCommentsIntegrationUtil.assertComments(result,
13                commentsInDatabase.get(i), i);
14        }
15
16        mockJwtAuthenticationUtil.verify(1);
17    }
```

Výpis kódu 20 Integračný test v Springu

5.2 Quarkus

V Quarkuse využijeme Quarkus JUnit 5 a v prípade integračných testoch využijeme REST Assured na testovanie endpointov, s ktorým sa pracuje podobne ako s MockMvc v Springu.

5.2.1 Unit testy

Unit testy sú v Quarkuse totožné s tými v Springu. Pre ich propmenutie môžete prejsť na podsekciu 5.1.1.

5.2.2 Integračné testy

Triedu, ktorá obsahuje integračné testy, označíme anotáciami *@QuarkusTest* (vytváranie kontextu aplikácie), *@QuarkusTestResource* (vybranie zdroja testovacích dát) a *@FlywayTest*. Anotácia *@FlywayTest* slúži na vkladanie a mazanie testovacích dát do databázy. [57] Toto rozšírenie si zvolíme z dôvodu toho, že Quarkus neponúka možnosť obnovenia databázy po každom teste. Podobne ako v Springu implementujeme triedu *MockJwtAuthenticationUtil*, definujeme atribút *commentsInDatabase*, ktorý obsahuje všetky komentáre uložené v databáze a taktiež definujeme dvoch užívateľov, kde jeden reprezentuje prihláseného užívateľa, na otestovanie prístupu k dátam. Príklad integračného testu môžete vidieť vo výpise kódu 21.

```
1   @Test
2   void all200() {
3       mockJwtAuthenticationUtil.mockReturns();
4       String result = RestAssured.given().header("Authorization",
5                   "Bearer " + mockJwtAuthenticationUtil.getMockedToken())
6               .when()
7                   .get("/comments")
8               .then().statusCode(200).contentType("application/json")
9                   .extract().body().asString();
10      for (int i = 0; i < commentsInDatabase.size(); i++) {
11          AssertCommentsIntegrationUtil.assertComments(result,
12              commentsInDatabase.get(i), i);
13      }
14      mockJwtAuthenticationUtil.verify(1);
15  }
```

Výpis kódu 21 Integračný test v Quarkuse

5.3 Micronaut

V Micronaute využijeme Micronaut Test JUnit 5 a v integračných testoch na otestovanie endpointov využijeme triedu HttpClient.

5.3.1 Unit testy

Unit testy sú v Micronaute totožné s tými v Springu alebo v Quarkuse. Pre ich propmenutie môžete prejsť na podsekciu 5.1.1.

5.3.2 Integračné testy

Triedu, ktorá obsahuje integračné testy, označíme anotáciami *@MicronautTest*, ktorá slúži na vytváranie kontextu aplikácie a *@FlywayTest*. Anotácia *@FlywayTest* slúži na vkladanie a mazanie testovacích dát do databázy. [57] Tak ako v Quarkuse, tak aj tu si toto rozšírenie zvolíme z dôvodu toho, že Micronaut neponúka možnosť obnovenia databázy po každom teste. Z toho dôvodu, že JWT nevalidujeme pomocou našej implementovanej triedy, ale Micronaut Security to robí za nás, tak priebeh autentizácie zostane nedotknutý. Keďže si definujeme dvoch užívateľov, kde jeden reprezentuje prihláseného užívateľa, tak pomocou jeho údajov si vytvoríme v triede *TestTokenGenerator* JWT. Tak ako v predošlých technológiách si potrebujeme definovať atribút *commentsInDatabase*, ktorý obsahuje všetky komentáre uložené v databáze, na kontrolu výstupov testov. Zjednodušený príklad integračného testu môžete vidieť vo výpise kódu 22.

```
1   @Test
2   void all200() {
3       HttpResponse<CommentDTO[]> result = client.toBlocking()
4       .exchange(HttpRequest.GET("").header("Authorization", "Bearer " +
5           loggedUserToken));
6       Assertions.assertTrue(result.getBody().isPresent());
7       CommentDTO[] resultBody = result.getBody().get();
8       Assertions.assertEquals(resultBody.length, commentsInDatabase.size());
9       int i = 0;
10      for (CommentDTO commentDTO : resultBody) {
11          AssertCommentsIntegrationUtil.assertComments(commentDTO,
12              commentsInDatabase.get(i++));
13      }
14  }
```

Výpis kódu 22 Integračný test v Micronaute

5.4 AWS

V AWS napíšeme len integračné testy, ktoré otestujú Lambda funkcie a jej komunikáciu s DynamoDB. Testovaciu triedu označíme *@Testcontainers*, aby sme mohli pracovať s Dynalite. Pred každým testom sa nahradi DynamoDB klient, ktorý Lambda funkcia využíva za jej klon Dynalite. Taktiež sa pred testami vytvoria testovacie záznamy v tejto databáze. Túto prípravu aj samotný test môžete vidieť vo výpise kódu 23.

V prípade testovania endpointov je dostupné manuálne testovanie v AWS konzole alebo ich dokážeme automatizovať pomocou služby AWS CodeBuild a AWS CodePipeline. AWS CodeBuild je služba, ktorá kompiluje zdrojový kód, spúšťa testy a vytvára softvérové balíky, ktoré sú pripravené na nasadenie. AWS CodePipeline je služba, ktorá pomáha automatizovať „pipelines" pre rýchle a spoľahlivé aktualizácie aplikácií a infraštruktúry. [58]

5.4.1 Zhrnutie testovania

Unit testovanie je totožné vo všetkých technológiách, pretože sa využíva framework JUnit 5 a taktiež Mockito. Ćiže podpora na Unit testovanie je vo všetkých štyroch technológiách totožná.

Integračné testy sú rozdielné a to konkrétne v knižniciach využívaných na volanie endpointov a databázou využívaných na testovanie. Spring, Quarkus a Micronaut na testovanie endpointov využívajú síce rôzne knižnice, ktoré sa pochopiteľne rozdielne používajú, ale výsledok je rovnaký. V prípade AWS na testovanie endpointov je potrebné nakonfigurovať AWS služby navyše a z toho dôvodu preferujem v tomto Spring, Quarkus a Micronaut.

Spring, Quarkus a Micronaut poskytujú databázu H2, ktorej použitie a konfigurácia sú totožné. V AWS sme použili Testcontainers Dynalite, čo nám ponúka pre integračné testovanie všetko, čo potrebujeme.

Všetky štyri technológie ponúkajú všetko, čo sme v rámci tejto práce potrebovali. Náročnosť písania testov bola jednoduchá vo všetkých technológiách, s výnimkou konfigurácie na testovanie endpointov v AWS, čo je ale z dôvodu, že REST API sa tvorí pomocou príkazov a nie v kóde ako v Springu, Quarkuse a Micronaute.

```
    @BeforeEach
    public void setUp() throws InterruptedException {
        DynamoDB dynamoDB = new DynamoDB(dynaliteContainer.getClient());
        InstanceModule.dynamoDB = dynamoDB;
        getCommentByIdHandler = new GetCommentByIdHandler();

        Table table = CreateTable.create(dynamoDB, TableName.commentTable);
        for (Comment comment : comments) {
            table.putItem(CreateItem.createCommentItem(comment));
        }
    }

    @Test
    void handleRequest() throws IOException {
        Map<String, String> pathParameters = new HashMap<>() {{
                put("id", comments.get(0).getId());
            }};
        APIGatewayProxyRequestEvent request =
        new APIGatewayProxyRequestEvent().withPathParameters(pathParameters);
        APIGatewayProxyResponseEvent response =
            getCommentByIdHandler.handleRequest(request, new TestContext());

        Assertions.assertEquals(200, response.getStatusCode());
        Comment commentResult =
        new ObjectMapper().readValue(response.getBody(), Comment.class);
        if (!AssertComments.commentsEquals(comments.get(0), commentResult)) {
            throw new AssertionFailedError();
        }
    }
```

Výpis kódu 23 Integračný test v AWS

Kapitola 6
Analýza kódu

V tejto kapitole si ukážeme výsledky z analýzy kódu. Rozdelíme si to podľa technológií, ktoré boli využité na tvorbu backendu.

Na analýzu kódu využijeme nástroj SonarQube popísaný v 3.5.8. To či kód statickou analýzou prešiel je dané nastavením „Quality Gate" a tie môžete vidieť v tabuľke 6.1.

Metrika	Operátor	Chyba
Pokrytie kódu	je menej ako	50 %
Duplicitné riadky	je viac ako	3 %
Hodnotenie udržiavateľnosti	je horšie ako	známka A
Hodnotenie spoľahlivosti	je horšie ako	známka A
Bezpečnostné hodnotenie	je horšie ako	známka A

Tabuľka 6.1 Nastavenia Quality Gate

6.1 Spring

Backend v Springu statickou analýzou kódu prešiel. Celkovo bolo prekontrolovaných 2,4 tisíc riadkov kódu. Našlo sa 0 „bugs", 2 zraniteľnosti bezpečnosti, predpokladaný čas na opravu „code smells" je 5 hodín, 52,1% pokrytie kódu a to z toho dôvodu, že boli testované len najdôležitejšie časti backendu. Duplikácia kódu je 0%. Spomínane zraniteľnosti bezpečnosti sa týkaju nevyužitia návratovej hodnoty vymazania súboru, ktorý je tvorený pri uploade.

6.2 Quarkus

Backend v Quarkuse statickou analýzou kódu prešiel. Celkovo bolo prekontrolovaných 2,4 tisíc riadkov kódu. Našlo sa 0 „bugs”, 2 zraniteľnosti bezpečnosti, predpokladaný čas na opravu „code smells” sú 3 hodiny, 50,6% pokrytie kódu a to z toho dôvodu, že boli testované len najdôležitejšie časti backendu. Duplikácia kódu je 0%. Spomínane zraniteľnosti bezpečnosti sa týkaju nevyužitia návratovej hodnoty vymazania súboru, ktorý je tvorený pri uploade.

6.3 Micronaut

Backend v Micronaute statickou analýzou kódu prešiel. Celkovo bolo prekontrolovaných 2,3 tisíc riadkov kódu. Našlo sa 0 „bugs”, 2 zraniteľnosti bezpečnosti, predpokladaný čas na opravu „code smells” je 5 hodín, 55,4% pokrytie kódu a to z toho dôvodu, že boli testované len najdôležitejšie časti backendu. Duplikácia kódu je 0%. Spomínane zraniteľnosti bezpečnosti sa týkaju nevyužitia návratovej hodnoty vymazania súboru, ktorý je tvorený pri uploade.

6.4 AWS

Backend v AWS statickou analýzou kódu prešiel. Celkovo bolo prekontrolovaných 2,6 tisíc riadkov kódu. Našlo sa 0 „bugs”, 8 zraniteľnosti bezpečnosti, predpokladaný čas na opravu „code smells” je 1 deň, 60,8% pokrytie kódu a to z toho dôvodu, že boli testované len najdôležitejšie časti backendu. Duplikácia kódu je 9,3% a tá by mohla byť vyriešená vytvorením knižníc, pretože každá Lambda funkcia je ako samostnatný projekt a v niektorých prípadoch majú podobné implementácie. Spomínane zraniteľnosti bezpečnosti sa týkaju chýbajúceho slovíčka „final” pri definícii niektorých atribútov.

	Spring	Quarkus	Micronaut	AWS
„Bugs”	0	0	0	0
Zraniteľnosti bezpečnosti	8	2	2	2
Oprava „code smells”	5 hodín	3 hodiny	5 hodín	1 deň
Pokrytie kódu	52,1%	50,6%	55,4%	60,8%
Duplikácia kódu	0%	0%	0%	9,3%

Tabuľka 6.2 Zhrnutie analýzy kódu

Záver

Cieľom tejto práce bol popis technológií na tvorbu backendu v Jave a popis aplikácie, ktorej serverovú časť sme neskôr implementovali.

V teoretickej časti sme si vysvetlili využitie technológií na tvorbu backendu v Jave a samotný výber technológií Spring, Quarkus, Micronaut a AWS. Ďalej sma sa pozreli na funkčnosti, ktorými by backend mal disponovať. Taktiež sme si stručne popísali pomocné služby a technológie, ktoré sme vo vybraných technológiách využili.

V praktickej časti následovala realizácia a otestovanie backendov vo vybraných technológiách zmienených v teoretickej časti. V rámci realizácie týchto backendov sme si ukázali možnú implementáciu, vysvetlili sme si použitý spôsob a popísali ďalšie možnosti, ktoré by bolo možné použiť. Na konci každej sekcii v realizácií sme si implementácie zhrnuli a navzájom ich porovnali.

V prípade rozšírenia tejto práce, by bolo ako prvé potrebné dokončiť časť neuskutočnenej implementácie v AWS. Konkrétne ide o implementáciu Elasticsearch 4.2.2, autentizácie, autorizácie 4.4.4 a validácie požiadavku 4.3.4. Táto implementácia nebola uskotočnená z dôvodu nedostatku času ako už bolo spomínane pri popise týchto sekcií. Taktiež by bolo možné pridať ďalšie technológie na tvorbu backendu, aby čitatelia tejto práce mali väčší prehľad v oblasti tvorenia backendu v Jave. Medzi implementáciami v technológiách sa opakuje časť logickej vrstvy, čo by sa dalo napraviť vytvorením knižnice, ktorú by si tieto technológie využívali. Zaujímavá časť, o ktorú by sa táto práca dala rozšíriť, je porovnanie technológií nie len z pohľadu využitia, ale aj z pohľadu ich vlastnej implementácie, efektivity kódu a rýchlosti.

Dodatok A
Zvyšný popis endpointov

A.1 Endpointy pre entitu Post

Na tieto endpointy vypísané v tabuľke A.1 môže užívateľ poslať požiadavku, aby dokázal pracovať s entitou Post, ktorá je popísaná v tabuľke 3.2.

Absolútna cesta	HTTP metóda	Popis
/posts	GET	Získanie všetkých príspevkov
/posts	POST	Vytvorenie príspevku
/posts/{id}	GET	Získanie príspevku
/posts/{id}	DELETE	Vymazanie príspevku
/posts/{id}/editDescription	PUT	Zmena popisu príspevku
/posts/ofUser?user	GET	Získanie všetkých príspevkov zvoleného užívateľa
/posts/count?user	GET	Získanie počtu všetkých príspevkov zvoleného užívateľa

Tabuľka A.1 Endpointy pre entitu Post

A.2 Endpointy pre entitu Follow

Na tieto endpointy vypísané v tabuľke A.2 môže užívateľ poslať požiadavku, aby dokázal pracovať s entitou Follow, ktorá je popísaná v tabuľke 3.3.

Absolútna cesta	HTTP metóda	Popis
/follows	GET	Získanie všetkých sledovania
/follows	POST	Vytvorenie sledovania
/follows/{id}	GET	Získanie sledovania
/follows/{id}	DELETE	Vymazanie sledovania
/follows/isFollowed/{userIdWhom}	GET	Získanie identifikátora sledovania, ak existuje, prihláseného užívateľa a zvoleného užívateľa

Tabuľka A.2 Endpointy pre entitu Follow

A.3 Endpointy pre entitu Like

Na tieto endpointy vypísané v tabuľke A.3 môže užívateľ poslať požiadavku, aby dokázal pracovať s entitou Like, ktorá je popísaná v tabuľke 3.4.

Absolútna cesta	HTTP metóda	Popis
/likes	GET	Získanie všetkých kladných reakcií
/likes	POST	Vytvorenie kladnej reakcie
/likes/{id}	GET	Získanie kladnej reakcie
/likes/{id}	DELETE	Vymazanie kladnej reakcie
/likes/onPost?post	GET	Získanie kladných reakcií na zvolenom príspevku
/likes/count?post	GET	Získanie počtu kladných reakcií na zvolenom príspevku
/likes/isLiked/{postId}	GET	Získanie identifikátora kladnej reakcie, ak existuje, pomocou prihláseného užívateľa a zvoleného príspevku

Tabuľka A.3 Endpointy pre entitu Like

A.4 Endpointy pre entitu Comment

Na tieto endpointy vypísané v tabuľke A.4 môže užívateľ poslať požiadavku, aby dokázal pracovať s entitou Comment, ktorá je popísaná v tabuľke 3.5.

Absolútna cesta	HTTP metóda	Popis
/comments	GET	Získanie všetkých komentárov
/comments	POST	Vytvorenie komentára
/comments/{id}	GET	Získanie komentára
/comments/{id}	DELETE	Vymazanie komentára
/comments/{id}/editText	PUT	Zmena textu komentára
/comments/onPost?post	GET	Získanie všetkých komentárov na zvolenom príspevku

Tabuľka A.4 Endpointy pre entitu Comment

Bibliografia

1. BALACHANDAR, Bogunuva Mohanram. *RESTful Java Web Services: A pragmatic guide to designing and building RESTful APIs using Java*. 3rd. Livery Place 35 Livery Street Birmingham B3 2PB: Packt, 2017. ISBN 9781788294041.
2. WONG, Clinton. *HTTP Pocket Reference*. Sebastopol, United States: O'Reilly Media, Inc, USA, 2000. ISBN 9781565928626.
3. *OAuth 2.0* [online]. 2022. Dostupné tiež z: `https://oauth.net/2`. [cit. 2022-05-02].
4. *Using OAuth 2.0 to Access Google APIs* [online]. 2021. Dostupné tiež z: `https://developers.google.com/identity/protocols/oauth2`. [cit. 2022-04-05].
5. JONES, ET AL. *JSON Web Token (JWT)* [online]. 2015. Dostupné tiež z: `https://datatracker.ietf.org/doc/html/rfc7519`. [cit. 2022-04-05].
6. *React* [online]. 2022. Dostupné tiež z: `https://reactjs.org`. [cit. 2022-04-21].
7. *Developer get started guide* [online]. 2022. Dostupné tiež z: `https://cloudinary.com/documentation/how_to_integrate_cloudinary`. [cit. 2022-04-08.
8. *Docker overview* [online]. 2022. Dostupné tiež z: `https://docs.docker.com/get-started/overview`. [cit. 2022-04-11].
9. *The heart of the free and open Elastic Stack* [online]. 2022. Dostupné tiež z: `https://www.elastic.co/elasticsearch/`. [cit. 2022-04-12].
10. *Logstash Introduction* [online]. 2022. Dostupné tiež z: `https://www.elastic.co/guide/en/logstash/current/introduction.html#introduction`. [cit. 2022-04-14].
11. *Logstash Reference* [online]. 2022. Dostupné tiež z: `https://www.elastic.co/guide/en/logstash/current/index.html`. [cit. 2022-04-14].
12. *What is Prometheus?* [Online]. 2022. Dostupné tiež z: `https://prometheus.io/docs/introduction/overview`. [cit. 2022-04-16].

13. *Getting started* [online]. 2022. Dostupné tiež z: https://prometheus.io/docs/prometheus/latest/getting_started. [cit. 2022-04-16].

14. *Introduction to Grafana* [online]. 2022. Dostupné tiež z: https://grafana.com/docs/grafana/latest/introduction. [cit. 2022-04-16].

15. *Getting started with Grafana* [online]. 2022. Dostupné tiež z: https://grafana.com/docs/grafana/latest/getting-started/getting-started. [cit. 2022-04-16].

16. *AWS Commant Line Interface* [online]. 2022. Dostupné tiež z: https://aws.amazon.com/cli. [cit. 2022-04-13].

17. *Localstack* [online]. 2022. Dostupné tiež z: https://localstack.cloud. [cit. 2022-04-13].

18. *SonarQube Documentation* [online]. 2022. Dostupné tiež z: https://docs.sonarqube.org/latest. [cit. 2022-04-26].

19. FOWLER, Martin. *Data Mapper* [online]. 2022. Dostupné tiež z: https://martinfowler.com/eaaCatalog/dataMapper.html. [cit. 2022-05-02].

20. GIERKE, Oliver; DARIMONT, Thomas; STROBL, Christoph; PALUCH, Mark; BRYANT, Jay. *Spring Data JPA - Reference Documentation* [online]. 2022. Dostupné tiež z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html. [cit. 2022-04-12].

21. GIERKE, Oliver; STROBL, Christoph; PALUCH, Mark; KRABBENBORG, Sander; WOUTERS, Jesse. *Interface JpaRepository* [online]. 2022. Dostupné tiež z: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html. [cit. 2022-04-12].

22. BAELDUNG. *Spring Data JPA @Query* [online]. 2021. Dostupné tiež z: https://www.baeldung.com/spring-data-jpa-query. [cit. 2022-04-12].

23. *Using Hibernate ORM and JPA* [online]. 2022. Dostupné tiež z: https://quarkus.io/guides/hibernate-orm. [cit. 2022-04-13].

24. *Introduction to contexts and dependency injection* [online]. 2022. Dostupné tiež z: https://quarkus.io/guides/cdi. [cit. 2022-04-13].

25. *Micronaut Data* [online]. 2022. Dostupné tiež z: https://micronaut-projects.github.io/micronaut-data/latest/guide. [cit. 2022-04-13].

26. *Build Modern Applications with Free Databases on AWS* [online]. 2022. Dostupné tiež z: https://aws.amazon.com/free/database. [cit. 2022-04-13].

27. *Java Code Examples* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/CodeSamples.Java.html. [cit. 2022-04-13].

28. *What is an Elasticsearch Index?* [Online]. 2022. Dostupné tiež z: https://www.elastic.co/blog/what-is-an-elasticsearch-index. [cit. 2022-04-14].

29. *What is OpenSearch?* [Online]. 2022. Dostupné tiež z: https://aws.amazon.com/opensearch-service/the-elk-stack/what-is-opensearch. [cit. 2022-04-16].

30. *Signing HTTP requests to Amazon OpenSearch Service* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/opensearch-service/latest/developerguide/request-signing.html. [cit. 2022-04-16].

31. *Building REST services with Spring* [online]. 2022. Dostupné tiež z: https://spring.io/guides/tutorials/rest. [cit. 2022-04-16].

32. KHAN, Uzma. *Multipart Request Handling in Spring* [online]. 2021. Dostupné tiež z: https://www.baeldung.com/sprint-boot-multipart-requests. [cit. 2022-04-16].

33. *Package javax.ws.rs* [online]. 2022. Dostupné tiež z: https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html. [cit. 2022-04-19].

34. *Amazon API Gateway* [online]. 2022. Dostupné tiež z: https://aws.amazon.com/api-gateway. [cit. 2022-04-19].

35. *AWS Lambda* [online]. 2022. Dostupné tiež z: https://aws.amazon.com/lambda. [cit. 2022-04-19].

36. *Set up an edge-optimized API using AWS CLI commands* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/apigateway/latest/developerguide/create-api-using-awscli.html. [cit. 2022-04-19].

37. *aws* [online]. 2022. Dostupné tiež z: https://awscli.amazonaws.com/v2/documentation/api/latest/reference/index.html. [cit. 2022-04-19].

38. *Enable request validation in API Gateway* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-request-validation.html. [cit. 2022-04-19].

39. *create-model* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/cli/latest/reference/apigateway/create-model.html. [cit. 2022-04-19].

40. *create-reuqest-validator* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/cli/latest/reference/apigateway/create-request-validator.html. [cit. 2022-04-19].

41. *Spring Security* [online]. 2022. Dostupné tiež z: https://spring.io/projects/spring-security. [cit. 2022-04-20].

42. *Welcome to OpenID Connect* [online]. 2022. Dostupné tiež z: https://openid.net/connect. [cit. 2022-04-20].

43. *Micronaut Security* [online]. 2022. Dostupné tiež z: https://micronaut-projects.github.io/micronaut-security/latest/guide. [cit. 2022-04-20].
44. BAELDUNG. *The Proxy Pattern in Java* [online]. 2019. Dostupné tiež z: https://www.baeldung.com/java-proxy-pattern. [cit. 2022-04-20].
45. *What is Amazon Cognito?* [Online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html. [cit. 2022-04-20].
46. *How do I set up Google as a federated identity provider in an Amazon Cognito user pool?* [Online]. 2021. Dostupné tiež z: https://aws.amazon.com/premiumsupport/knowledge-center/cognito-google-social-identity-provider. [cit. 2022-04-20].
47. *cognito-idp* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/cli/latest/reference/cognito-idp/index.html#cli-aws-cognito-idp. [cit. 2022-04-20].
48. *Post authentication Lambda trigger* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-lambda-post-authentication.html. [cit. 2022-04-20].
49. *apigateway* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/cli/latest/reference/apigateway/index.html#cli-aws-apigateway. [cit. 2022-04-20].
50. *Integrate a REST API with an Amazon Cognito user pool* [online]. 2022. Dostupné tiež z: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-enable-cognito-user-pool.html. [cit. 2022-04-20].
51. *Amazon CloudWatch Features* [online]. 2022. Dostupné tiež z: https://aws.amazon.com/cloudwatch/features. [cit. 2022-04-19].
52. RISBERG, Thomas. *Spring Framework 1.0 Final Released* [online]. 2004. Dostupné tiež z: https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released. [cit. 2022-04-27].
53. BERNARD, Emmanuel. *Announcing Quarkus 1.0* [online]. 2019. Dostupné tiež z: https://quarkus.io/blog/announcing-quarkus-1-0. [cit. 2022-04-27].
54. ROCHER, Greaeme. *Micronaut 1.0 GA Released* [online]. 2018. Dostupné tiež z: https://micronaut.io/2018/10/23/micronaut-1-0-ga-released. [cit. 2022-04-27].
55. *About AWS* [online]. 2022. Dostupné tiež z: https://aws.amazon.com/about-aws. [cit. 2022-04-27].

56. PARASCHIV, Eugen. *Getting Started with Mockito @Mock, @Spy, @Captor and @InjectMocks* [online]. 2021. Dostupné tiež z: `https://www.baeldung.com/mockito-annotations`. [cit. 2022-04-20].

57. CORTEZ, Roberto. *Flyway JUnit 5 Extensions* [online]. 2022. Dostupné tiež z: `https://github.com/radcortez/flyway-junit5-extensions`. [cit. 2022-04-20].

58. LAMADRID, Juan; SHARDHA, Kapil. *Automating your API testing with AWS CodeBuild, AWS CodePipeline, and Postman* [online]. 2020. Dostupné tiež z: `https://aws.amazon.com/blogs/devops/automating-your-api-testing-with-aws-codebuild-aws-codepipeline-and-postman`. [cit. 2022-04-26].

Obsah přiloženého média

- `src`
  - `impl` ..... zdrojové kódy implementácie
    - `aws-instamini-backend` ..... zdrojový kód v technológii AWS
    - `instagram_mini` ..... zdrojový kód v technológiách Spring, Quarkus a Micronaut
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF