Usporedba Java i Kotlin mikroservisne arhitekture bazirane na Spring boot frameworku

Milošević, Marija

Sveučilište J.J. Strossmayera u Osijeku
Filozofski fakultet Osijek
Dvopredmetni diplomski studij informatologije i informacijskih tehnologija

---

Marija Milošević

Usporedba Java i Kotlin mikroservisne arhitekture bazirane na Spring boot frameworku

Diplomski rad

Mentor: doc. dr. sc. Tomislav Jakopec

Osijek, 2023.

Sveučilište J.J. Strossmayera u Osijeku
Filozofski fakultet Osijek
Odsjek za informacijske znanosti
Dvopredmetni diplomski studij informatologije i informacijskih tehnologija

---

Marija Milošević

Usporedba Java i Kotlin mikroservisne arhitekture bazirane na Spring boot frameworku

Diplomski rad
Društvene znanosti, informacijske i komunikacijske znanosti, informacijsko i programsko inženjerstvo

Mentor: doc. dr. sc. Tomislav Jakopec

Osijek, 2023.

IZJAVA

Izjavljujem s punom materijalnom i moralnom odgovornošću da sam ovaj rad samostalno napisala te da u njemu nema kopiranih ili prepisanih dijelova teksta tuđih radova, a da nisu označeni kao citati s navođenjem izvora odakle su preneseni.

Svojim vlastoručnim potpisom potvrđujem da sam suglasna da Filozofski fakultet u Osijeku trajno pohrani i javno objavi ovaj moj rad u internetskoj bazi završnih i diplomskih radova knjižnice Filozofskog fakulteta u Osijeku, knjižnice Sveučilišta Josipa Jurja Strossmayera u Osijeku i Nacionalne i sveučilišne knjižnice u Zagrebu.

U Osijeku, 4.10.2023.

Marija Milošević, 0122228714

ZAHVALA

Zahvaljujem svima koji su mi nesebično pomogli prilikom izrade ovog diplomskog rada, posebice komentoru Damiru Tibljašu koji je uvijek bio na raspolaganju za rješavanje pogrešaka u programskom kodu, kao i detaljnijih objašnjenja pojedinih principa. Svojom stručnošću i izvrsnim poznavanjem Java i Kotlin programskih jezika neprestano je upućivao na primjere dobrih praksi i relevantne izvore kojima je osigurao kvalitetu ovoga rada.

SAŽETAK

Svrha ovoga rada je opisati sličnosti i razlike Java i Kotlin programskog jezika kreiranjem aplikacije mikroservisne arhitekture temeljene na Spring Boot frameworku. Tema rada definirana je prateći trendove informacijskih tehnologija prema kojima sve veću popularnost dobivaju razvojni okviri poput Springa i Spring Boota, objektno-orijentirani jezici kao što su Java i njemu alternativni programski jezik Kotlin koji se koristi za izradu mobilnih aplikacija te jedan od većih trendova i prekretnica u tehnologiji: mikroservisi. Kako bi se ostvarila svrha i tema rada, kreirane su dvije aplikacije iste programske logike, s razlikom što je jedna kreirana u Java programskom jeziku, a druga u Kotlin programskom jeziku. Obje aplikacije kreirane su prema principima mikroservisne arhitekture te se sastoje od jednog mikroservisa koji predstavlja logiku rada aplikacije, drugog mikroservisa koji upravlja komunikacijom između mikroservisa te trećeg mikroservisa koji služi za otkrivanje pokrenutih mikroservisa. Kao razvojni okvir za kreiranje aplikacija, odabran je Spring Boot koji se, u trenutku pisanja ovoga rada, nalazi među najpopularnijim razvojnim okvirima za izradu mikroservisnih aplikacija, a svojim anotacijama i brojnim zavisnostima olakšava i ubrzava kreiranje aplikacija. Navedeni mikroservisi osiguravaju slanje zahtjeva preko REST API-ja te je cilj kreirati aplikaciju koja će osigurati uspješno registriranje korisnika, upravljanje podacima o korisnicima i porukama te osigurati slanje SMS poruke preko API-ja. API za komuniciranje javno je dostupan registriranim korisnicima na Infobipovoj web stranici te je upotrijebljen za potrebe ovoga rada. Nakon kreiranja svih navedenih mikroservisa, osigurana je njihova međusobna komunikacija. Radom se opisuju koraci u izradi aplikacije u svakom pojedinom programskom jeziku, razlozi korištenja određenih dijelova programskog koda te načini provjere rada određenih dijelova. Nakon uspješnog kreiranja aplikacije u oba programska jezika, uspoređene su sličnosti i razlike navedenih programskih jezika.

Ključne riječi: mikroservisna arhitektura, mikroservisna aplikacija, Spring Boot framework, Java, Kotlin

SADRŽAJ

1. Uvod

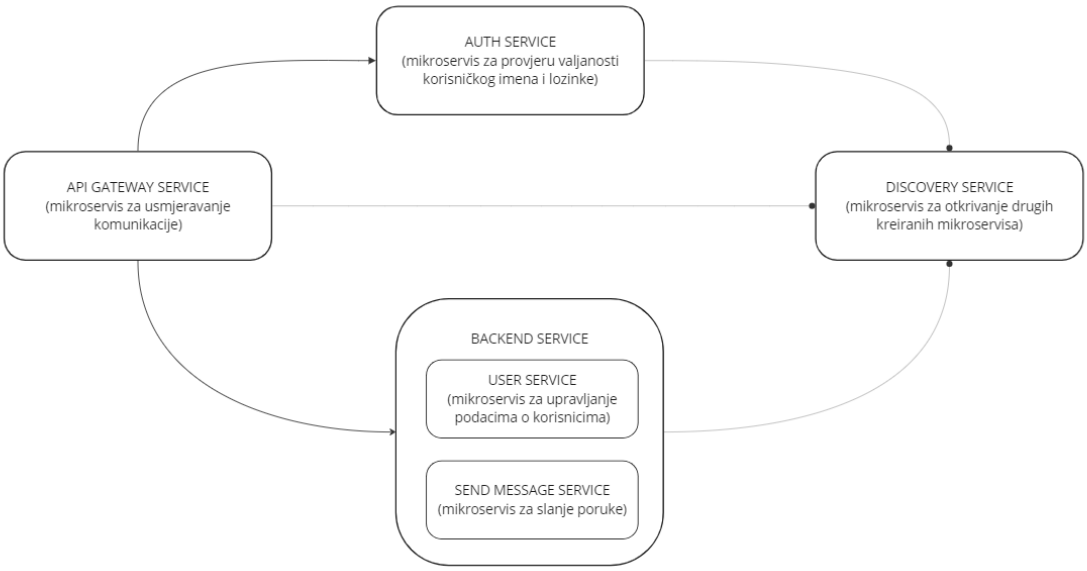
Razvojem tehnologija neprestano se mijenjaju trendovi i popularnosti pojedinih elemenata te tehnologije. Trenutnu popularnost uživaju tehnologije kao što su umjetna inteligencija i strojno učenje, internet stvari, progresivne mrežne aplikacije, API-ji (engl. *Application Programming Interface*) i mikroservisi, ali i brojne druge tehnologije.[1] U budućnosti, trenutne trendove zasigurno će zamijeniti ili pak poboljšati neke druge tehnologije i alati, ali za trenutni opstanak na tržištu neupitno je pratiti trendove i primjenjivati ih. Tako je zadatak ovog diplomskog rada koristeći neke od alata i elemenata trendova kreirati aplikaciju mikroservisne arhitekture koristeći javno dostupan API dostupan na mrežnoj stranici tvrtke Infobip za slanje SMS (engl. *Short Message Service*) poruke. Cilj je rada izraditi i prikazati rad *backend* dijela aplikacije u Java i Kotlin programskom jeziku. U ovom je radu naglasak na *backend* dijelu koji se može opisati kao programska logika, odnosno sve ono što korisnik ne vidi, a ima važnu ulogu za funkcioniranje mrežnih stranica ili aplikacija. Rad je podijeljen na nekoliko poglavlja u kojima se nastoji opravdati svrha rada. Tako poglavlje koje slijedi nakon uvodnog dijela se nadovezuje na uvodni dio tako što na teorijskoj razini opisuje što su to razvojni okviri, zašto su se oni pojavili te zašto je za ovaj rad odabran Spring Boot i koje su njegove karakteristike. Također, opisat će se teorijske postavke mikroservisa te razlog učestalog korištenja REST (engl. *REpresentational State Transfer*) principa s arhitekturom mikroservisa. Popularni model koji opisuje elemente REST pristupa kreirao je Leonard Richardson te se prema njemu i naziva *Richardson Maturity Model*. Model je podijeljen na nekoliko razina, počevši od razine 0 pa sve do razine označene brojem 3.[2] Nulta razina predstavlja korištenje HTTP-a (engl. *HyperText Transfer Protocol*) kao transportnog sustava za udaljene interakcije bez korištenja mehanizma weba, odnosno za vlastite interne mehanizme interakcije. Prva razina predstavlja poziv metode objekta preko argumenta, dok druga razina obuhvaća korištenje HTTP metoda kao što su GET, POST, PUT i DELETE. Zadnja razina, razina 3, sadrži kontrole poznatije pod kraticom HATEOAS (engl. *Hypermedia As Transfer Engine Of Application State*) koje upućuju na daljnje korake,

[1] Usp. Programming Trends Worth Watching in 2023, 2023. URL: https://blog.gitnux.com/programming-trends/ (2023-07-17)
[2] Usp. Fowler, M. Richardson Maturity Model, 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (2023-07-17)

odnosno mijenja URI (engl. *Uniform Resource Identifier*) shemu.[3] U ovome radu, nastojalo se pratiti navedeni model i primjere dobre prakse prilikom kreiranja REST API sustava.

Sljedeće poglavlje započinje kratkim opisom Java programskog jezika, nakon čega slijedi potpoglavlje opisa osiguravanja radne okoline za razvoj Java i Kotlin aplikacije. Tako je opisan način kreiranja Spring Boot projekta pomoću Spring Initializr alata integriranog u IntelliJ IDEA razvojno okruženje te su definirane zadane anotacije i datoteke unutar kreiranog projekta, kao i dodavanje baze podataka putem Docker alata za kontejnerizaciju (prema engl. *container*) programske podrške. U sljedećem poglavlju opisani su načini korištenja REST API standarda kroz kreirane projekte u Java programskom jeziku, odnosno mikroservise te sve zavisnosti koje je potrebno dodati i dijelovi koje je potrebno promijeniti kako bi poslani zahtjevi rezultirali željenim odgovorom, bilo za slanje podataka o korisniku ili o porukama. U potpoglavlju kreiranja mikroservisa za slanje poruke, objašnjen je postupak povezivanja s javno dostupnim API-jem globalne telekomunikacijske tvrtke Infobip, dok je u zadnjem potpoglavlju objašnjen postupak kreiranja mikroservisa koji omogućuje prvotnu registraciju za slanje prethodno navedenih zahtjeva preko API-ja, mikroservisa koji služi za komunikaciju i mikroservisa za otkrivanje svih povezanih mikroservisa. Zatim slijedi poglavlje broj 5. *Razvoj mikroservisne arhitekture u Kotlin programskom jeziku pomoću Spring Boot frameworka* koje započinje kratkim opisom Kotlin programskog jezika i njegovom upotrebom u suvremenom kontekstu, dok su potpoglavlja organizirana na isti način kao u prethodnom poglavlju gdje je cilj bio kreirati Java aplikaciju mikroservisne arhitekture. Iz tog razloga sadržaj poglavlja slijedi definirani sadržaj 4. poglavlja. Dijagram kreiranih mikroservisa vidljiv je na Slici 1.

[3] Usp. Isto.



Slika 1. Dijagram kreirane mikroservisne arhitekture

Zadnje poglavlje odnosi se na usporedbu dviju kreiranih aplikacija, odnosno sličnosti i različitosti između Java i Kotlin programskih jezika na temelju kreiranih mikroservisnih aplikacija čime je zaključen ovaj rad.

2. Spring Boot framework i mikroservisi

Kao što je navedeno u uvodnom dijelu, svrha rada je prikazati razvoj mikroservisne aplikacije u Java i Kotlin programskim jezicima korištenjem Spring Boot razvojnog okvira ili razvojne cjeline ili engl. *framework* (dalje u tekstu razvojni okvir). Razvojni okvir koristi se prilikom razvoja aplikacije za pojednostavljivanje razvojnog procesa aplikacije. S obzirom da se izrada aplikacije može podijeliti na *frontend* i *backend* dio, odnosno na dio koji je vidljiv korisnicima i na onaj koji obavlja funkcionalnosti u pozadini, na isti se način mogu podijeliti i razvojni okviri. *Frontend* razvojni okviri usmjereni su na HTML (engl. *HyperText Markup Language*) označiteljski jezik, CSS (engl. *Cascading Style Sheets*) stilski jezik te JavaScript programski jezik ili JavaScript radne okoline, dok su razvojni okviri iz područja *backenda* najčešće usmjereni na programske jezike poput Ruby, Python, PHP-a, Jave, JavaScripta ili C#.[4] Od pojave prvog razvojnog okvira do danas, stručnjaci iz područja informacijskih tehnologija navode njihove brojne prednosti koje, među ostalima, uključuju uštedu vremena, sigurniji kod

[4] Usp. Top Web Development Frameworks (Frontend & Backend), 2022. URL: https://www.emizentech.com/blog/web-development-frameworks.html (2023-06-28)

bez nepotrebnih ponavljanja te jednostavnije testiranje aplikacija.[5] Posljedica više prednosti nego nedostataka razvojnih okvira dovela je do čitavog niza raznih razvojnih okvira, a osim onih koji pripadaju *backendu* i *frontendu*, razlikujemo još i razvojne okvire za mrežne aplikacije kao što su Django i Ruby on Rails, za razvoj mobilnih aplikacija kao što su Flutter i React Native, za podatkovnu znanost gdje se izdvajaju PyTorch i TensorFlow te razvojni okviri za upravljanje sadržajem gdje veliku popularnost ima WordPress.[6] Također, postoje popularniji i oni manje popularni razvojni okviri, a među popularnim *frontend* razvojnim okvirima izdvajaju se AngularJS, ReactJS te Vue, dok se među popularnijim *backend* razvojnim okvirima izdvajaju oni koji su razvijeni za potrebe Pythona. Od ostalih razvojnih okvira kreiranih za potrebe ostalih programskih jezika izdvaja se Spring, razvojni okvir kreiran za Java programski jezik, čija je ekstenzija Spring Boot korisna za kreiranje aplikacija bez ili s minimalno konfiguracije, a upravo se na tom radnom okviru temelji ovaj rad.[7]

2.1. Korištenje Spring Boot frameworka u Javi i Kotlinu

Spring Boot je razvojni okvir utemeljen na Java programskom jeziku koji se razvio iz Spring razvojnog okvira kako bi se osigurala jednostavnija izrada aplikacija. Iz tog se razloga Spring Boot smatra ekstenzijom Spring razvojnog okvira koja koristi konfiguraciju putem anotacije i application.properties ili application.yml datoteke kojom se nastoji smanjiti uporaba XML-a (engl. *eXtensible Markup Language*) redovito korištenog u tradicionalnim Spring aplikacijama.[8] Ono po čemu su Spring i Spring Boot slični su postojanje inverzije kontrole (engl. *Inversion of Control* ili kratica IoC) te injekcija ovisnosti (engl. *Dependency Injection* ili kratica DI). Inverzija kontrole je princip korišten u programskom inženjerstvu koji prenosi kontrolu nad objektima ili dijelovima programa u spremnik (engl. *container*) ili razvojni okvir (engl. *framework*).[9] Pojednostavljeno, inverzija kontrole predstavlja način kreiranja i povezivanja komponenti u aplikaciji, no umjesto da to radi programer, u Spring Bootu to radi razvojni okvir. Na taj je način osigurana veća modularnost programa, lakši prijelaz između

[5] Usp. Mitchell, B. What is a Framework and Why It's Useful, 2023. URL: https://www.codingdojo.com/blog/what-is-a-framework (2023-06-28)
[6] Usp. Isto.
[7] Usp. Spring vs Spring Boot: An in-depth Comparison, 2023. URL: https://www.turing.com/kb/spring-vs-spring-boots-best-web-apps (2023-06-28)
[8] Usp. Isto.
[9] Usp. Crusoveanu, L. Intro to Inversion of Control and Dependency Injection with Spring, 2023. URL: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring (2023-07-03)

implementacija, jednostavnije testiranje i slično.[10] Inverzija kontrole može se postići na nekoliko načina, koristeći određene uzorke ili injekciju ovisnosti. U injekciji ovisnosti, kontrola koja prolazi inverziju je postavljena kao ovisnost objekta, a injekciju ovisnosti Spring Boot također odrađuje automatski.[11] Pojednostavljeno rečeno, injekcija ovisnosti u Spring Boot razvojnom okviru predstavlja osiguravanje svih potrebnih komponenti tako da razvojni okvir sam prepoznaje što mu je potrebno te automatski osigurava što mu je potrebno. Postoje i određene razlike Springa i Spring Boota, a one se ogledaju u tome što je Spring razvojni okvir stvoren za rješavanje složenosti razvoj poslovnih aplikacija, dok je Spring Boot najpopularniji razvojni okvir za izradu mikroservisnih aplikacija. Iako je Spring kreiran za korištenje s Java programskim jezikom, odnosno smatra se jednim od popularnijih Java razvojnih okvira, danas on pruža podršku i za druge programske jezike, ali samo one koji su interoperabilni s Javom. Tako se Spring i Spring Boot često koriste s Kotlinom, a moguća je i uporaba s Apache Groovy objektno orijentiranim programskim jezikom ili pak nekim od dinamičkih programskih jezika kao što je JRuby implementacija Ruby programskog jezika na Java virtualnom stroju (engl. *Java Virtual Machine*).[12] U ovom će se radu prikazati rad Spring Boot razvojnog okvira u dva programska jezika: Javi i Kotlinu.

Spring Boot razvojni okvir olakšava izradu aplikacija te je od samih početaka prihvaćen u zajednici programera. Njegov način rada temelji se na pregledavanju programskog koda i izvršavanju napisanog koda na temelju anotacije.[13] Anotacije koje u tom slučaju pregledava su *@RestController* i *@RequestMapping*, a cijeli taj postupak omogućen je zbog postojanja ugrađenog Tomcat servera unutar razvojnog okvira. Anotacija koja predstavlja i upućuje na to da se aplikacija izvodi pomoću Spring Boot razvojnog okvira je *@SpringBootApplication* i ona se nalazi u *main* klasi programskog koda čime se inicijalizira Spring aplikacija. Od ostalih značajki Spring Boota mogu se izdvojiti klasa *SpringApplicationBuilder* koja olakšava kreiranje API-ja, mogućnost pristupa bilo kojem argumentu aplikacije, dodavanje vlastite konfiguracije, konfiguriranje i upravljanje potrebnim zavisnostima i slično.[14] Spring Boot aplikacija može se kreirati na više načina:

[10] Usp. Isto.
[11] Usp. Isto.
[12] Usp. Spring Framework Documenation: Language Support,2020. URL: https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/languages.html (2023-07-04)
[13] Usp. Gutierrez, F. Pro Spring Boot. New Mexico: Apress, 2016. Str. 4.
[14] Usp. Isto, str. 6-7.

1. pomoću Spring Boot Command Line Interface (CLI) koji omogućava korištenje Groovy i Java aplikacija pomoću integriranog razvojnog okruženja (engl. *Integrated Development Environment* ili kraće IDE)
2. pomoću Spring Initializr (dostupan na poveznici: https://start.spring.io/ putem weba ili preko cURL-a, engl. *Client for Uniform Resource Locator*) koji također ima i svoju verziju implementiranu u IntelliJ IDEA integrirano razvojno okruženje
3. ili korištenjem Spring Tool Suite razvojnog okruženja.[15]

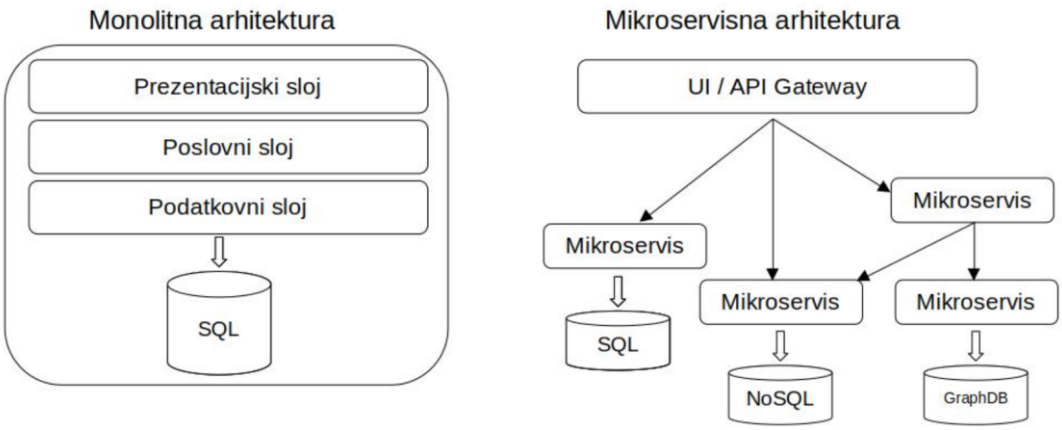
Također, osim navedenog moguće je i kreiranje aplikacije pomoću Visual Studio Code uređivača (engl. *code editor*), no u tom je slučaju potrebno instalirati nekoliko ekstenzija, kao što su *Coding Pack for Java* ili *Java Extension Pack* te *Spring Boot Extension Pack*.[16] Neovisno o načinu na koji se kreira i koristi aplikacija, odnosno projekt koji želimo kreirati, potrebno je osigurati određeno razvojno okruženje ili uređivač te neovisno kreiramo li Java ili Kotlin aplikaciju, u slučajevima korištenja Visual Studio Code uređivača, Spring Tool Suite ili IntelliJ IDEA razvojnog okruženja potreban je JDK, odnosno engl. *Java Development Kit* koji omogućava kompajliranje (prema engl. *compile*) i pokretanje Java aplikacija.[17] Za potrebe ovoga rada i prikaza aplikacije u Java i Kotlin programskom jeziku koristit će se integrirani Spring Initalizr u IntelliJ IDEA razvojnom okruženju, a kao alat za automatizaciju izrade (engl. *build automation tool*) koji omogućavaju stvaranje izvršnih aplikacija iz izvornog koda koristit će se Maven. Maven alat kreirala je Apache grupa i specijaliziran je za projekte kreirane u Java programskom jeziku, a za strukturu projekta koristi XML.[18]

2.2. Mikroservisna arhitektura

Mikroservisna arhitektura koristi se za razdvajanje funkcionalnosti aplikacije u više dijelova, odnosno na više servisa. Takvo razdvajanje osigurava određene prednosti kao što su jednostavnost, poboljšana komunikacija, fleksibilnost, skalabilnost te učinkovitost u razvoju aplikacija.[19] Također, u takvoj arhitekturi svaki pojedini servis pokreće određeni proces,

[15] Usp. Isto, str. 9-24.
[16] Usp. Building an Application with Spring Boot, 2023. URL: https://spring.io/guides/gs/spring-boot/ (2023-07-04)
[17] Usp. Isto.
[18] Usp. Gradle Vs Maven: What's The Difference?, 2023. URL: https://www.interviewbit.com/blog/gradle-vs-maven/ (2023-04-07)
[19] Usp. Foote, K. D. A Brief History of Microservices, 2021. URL: https://www.dataversity.net/a-brief-history-of-microservices/ (2023-07-04)

odnosno postoji više nezavisnih aplikacija koje se mogu zasebno pokrenuti, neovisno o postojanju drugih aplikacija tj. servisa. Razvili su se iz postojeće monolitne arhitekture kao odgovor na nedostatke, kao što su to da ukoliko jedna komponenta ne radi, to znači da niti cijeli sustav neće raditi ili pak ukoliko je potrebno ažurirati jednu komponentu, neizostavno je ažuriranje cijelog sustava.[20] Za razliku od prijašnjih monolitnih aplikacija, svaka mikroservisna aplikacija ima vlastite slojeve kao što su prezentacijski, poslovni i podatkovni, dok u monolitnoj arhitekturi čine jedinstvenu strukturu (Slika 2.).[21]



Slika 2. Razlika monolitne i mikroservisne arhitekture[22]

S obzirom na sve učestalije korištenje mikroservisne arhitekture u raznim projektima, neizbježno je unapređivanje postojećih razvojnih okvira te kreiranje novih. Već spomenuti razvojni okvir koji je vrlo popularan prilikom kreiranja mikroservisnih aplikacija je Spring Boot, a još neki koji se koriste uz Java programski jezik su Jersey koji je popularan za kreiranje RESTful servisa te Swagger koji olakšava dokumentiranje API-ja.[23] Osim kreiranja mikroservisnih aplikacija pomoću određenog razvojnog okvira krećući od početka, postoji i način kreiranja mikroservisa tako što se prvotna monolitna aplikacija podijeli na nekoliko mikroservisnih aplikacija. Ideja takvog pristupa je olakšati cijeli proces kreiranja i održavanja

[20] Usp. Kappagantula, S. Microservices Tutorial: Learn all about Microservices with Example, 2023. URL: https://www.edureka.co/blog/microservices-tutorial-with-example (2023-07-04)
[21] Usp. Rajesh, R. V. Spring Microservices: Build scalable microservices with Spring, Docker and Mesos. Birmingham: Packt Publishing, 2016. Str. 7.
[22] Usp. Gutierrez, F. Nav. dj. Str. 309.
[23] Usp. What are Microservices? Code Examples, Best Practices, Tutorials and More, 2019. URL: https://stackify.com/what-are-microservices/ (2023-07-04)

takve aplikacije, a neki programeri smatraju da je takav pristup primjer dobre prakse za kreiranje mikroservisnih aplikacija.[24]

Kreiranjem mikroservisa osigurava se rad zasebnih aplikacija među kojima je potrebno ostvariti komunikaciju. Komunikacija mikroservisne arhitekture može se provesti sinkronim ili asinkronim putem.[25] Sinkrona komunikacija podrazumijeva čekanje servisa na odgovor nakon što se pošalje zahtjev, a asinkrona komunikacija najčešće podrazumijeva sustav poruka kao što je RabbitMQ ili Apache Kafka. Ona se odvija tako što klijent dobiva poruku od posrednika poruke (engl. *message broker*) koji dobiva poruku od servisa, a glavna značajka je da pošiljatelj ne treba čekati odgovor.[26] Sinkrona komunikacija najčešće se odvija preko HTTP ili REST servisa koji kao odgovor prikazuju podatke u XML ili JSON (engl. *JavaScript Object Notation*) formatu. Koristeći takav način komunikacije servisima su dostupne *GET*, *POST*, *PATCH*, *PUT* ili *DELETE* metode, a kao mogući REST klijent za komunikaciju dostupno je sučelje Springa nazvano *WebClient* koje se koristi za slanje mrežnih zahtjeva.[27] Drugi oblik komunikacije, asinkroni oblik komunikacije, najčešće se provodi preko JMS (engl. *Java Message Service*) implementacije ili putem protokola AMQP (engl. *Advanced Message Queuing Protocol*) te se koristi kada odgovor nije potreban u trenutku slanja poruke.[28] U takvoj vrsti komunikacije posrednik poruke je odgovoran za poslanu poruku, odnosno osigurava njezin dolazak do ciljanog odredišta.

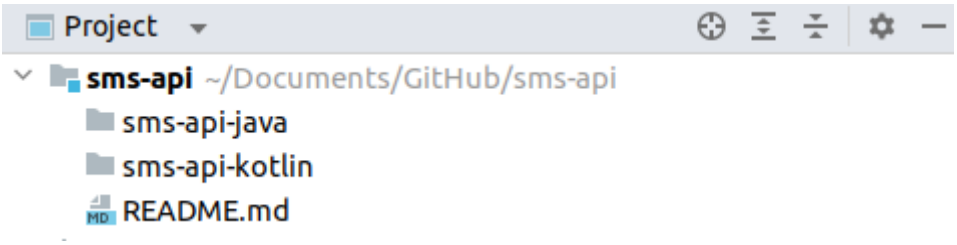
Svaka mikroservisna arhitektura se sastoji od nekoliko sastavnih dijelova koje čine klijenti (engl. *Clients*), pružatelji identiteta (engl. *Identity Providers*), API pristupnici (engl. *API Gateway*), statički sadržaj (engl. *Static Content*), upravljanje (engl. *Management*), otkrivanje servisa (engl. *Service Discovery*), mreža za dostavljanje podataka (engl. *Content Delivery Networks*) i udaljeni servisi (engl. *Remote Service*).[29] Svaki dio ima svoju ulogu kada aplikaciji pristupe različiti klijenti koji koriste različite uređaje. Tako pružatelji identiteta obavljaju provjeru vjerodostojnosti (engl. *authentication*) klijenta i dodjeljuju tokene nakon što zaprime zahtjev određenog klijenta koji zatim odlazi do API pristupnika koji upravljaju zahtjevima klijenata tako što ga šalju do određenog mikroservisa, dok statički sadržaj

[24] Usp. Behler, M. Java Microservices: A Practical Guide, 2020. URL: https://www.marcobehler.com/guides/java-microservices-a-practical-guide (2023-07-04)
[25] Usp. Isto.
[26] Usp. Atlasik, K. How to communicate Java microservices?, 2021. URL: https://softwaremill.com/how-to-communicate-java-microservices/ (2023-07-04)
[27] Usp. Spring 5 WebClient, 2022. URL: https://www.baeldung.com/spring-5-webclient (2023-07-04)
[28] Usp. Behler, M. Nav. dj.
[29] Usp. Kappagantula, S. Nav. dj.

predstavlja sadržaj cijelog sustava. Uloga upravljanja je identificiranje neuspjeha, a uloga otkrivanja servisa je, kako i sama riječ kaže, otkrivanje puteva za komunikaciju između mikroservisa. Mreža za dostavljanje podataka predstavlja distribuiranu mrežu proxy servera i njihovih podatkovnih centara, a udaljeni servisi omogućuju udaljeni pristup informacijama pohranjenim na tehnološkim uređajima.[30]

3. Postavljanje okoline za razvoj aplikacije

S ciljem kreiranja bilo koje aplikacije i korištenja određenih alata te programskih jezika nužno je prvotno postaviti radnu okolinu kako bi se osigurao neometani rad aplikacije. Kao prvi korak kreiran je direktorij na računalu u kojem će biti smješteni projekti za Java i Kotlin aplikaciju, a unutar tog direktorija će se nalaziti dodatna dva direktorija kako bi se odvojile Java i Kotlin aplikacije. Unutar svakog pojedinog direktorija nalazit će se projekti za svaki programski jezik, a svaki projekt će predstavljati jedan mikroservis. Za cijeli postupak pisanja koda koristit će se IntelliJ IDEA Ultimate Edition razvojno okruženje koje je besplatno dostupno za studente preko GitHub studentskog paketa.[31] U odabranom razvojnom okruženju otvoren je prethodno kreirani direktorij (Slika 3.) što označava početak kreiranja projekata, odnosno mikroservisne arhitekture aplikacije.

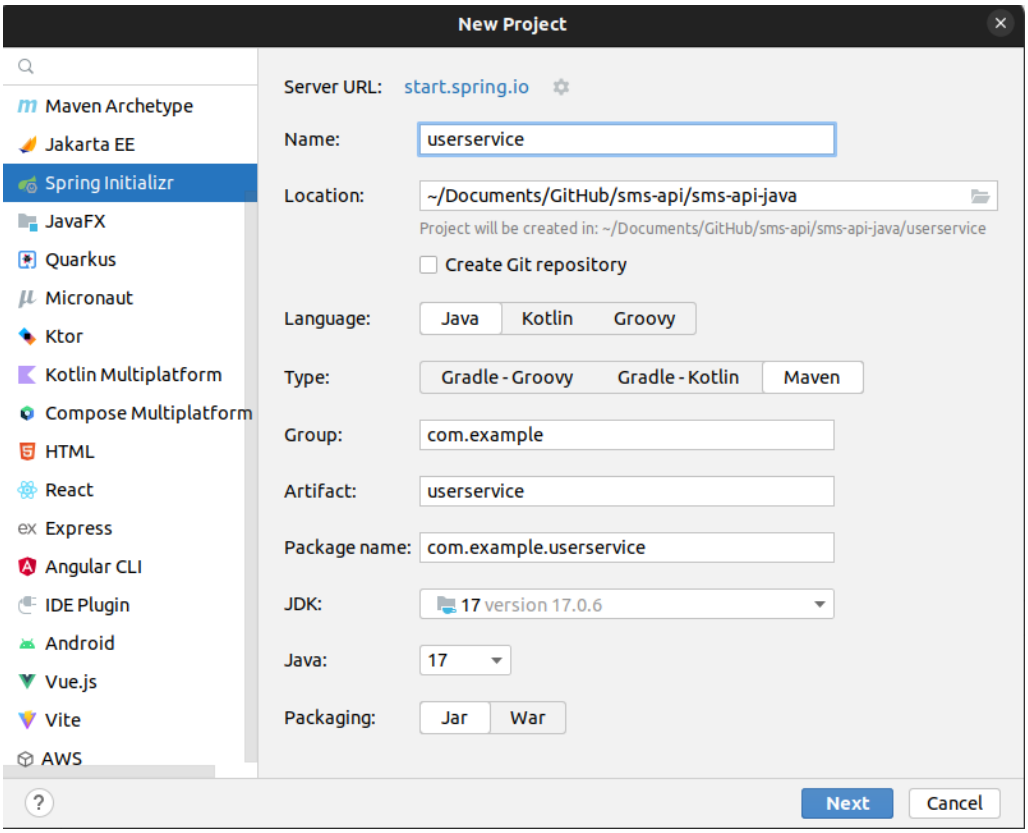


Slika 3. Struktura direktorija

Na slici je vidljivo kako su za potrebe ovog projekta kreirana dva direktorija te je dodana datoteka README.md jer se cijeli direktorij, u obliku repozitorija, nalazi na GitHubu na poveznici: https://github.com/marmilosev/sms-api. Sljedeći korak je kreirati projekt, odnosno mikroservis u Java programskom jeziku koji će služiti za upravljanje podacima o korisnicima te je iz tog razloga nazvan *user service*. Projekt je kreiran tako što se u IntelliJ IDEA razvojnom okruženju odabrala opcija za stvaranje novog projekta, nakon čega se otvori novi

[30] Usp. Isto.
[31] Usp. GitHub Student Developer Pack, 2023. URL: https://education.github.com/pack (2023-07-05)

prozor u kojemu se biraju opcije, a u tim opcijama je odabran Spring Initializr, prethodno spomenuti način kreiranja Spring Boot projekta (Slika 4.).



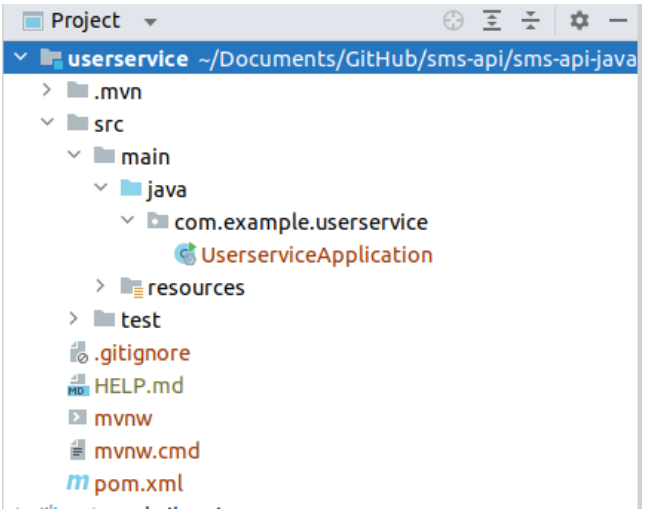
Slika 4. Kreiranje projekta *userservice* kroz IntelliJ IDEA pomoću alata Spring Initializr

Na priloženoj slici vidljivo je sve što je potrebno odabrati prilikom kreiranja projekta. Prvo je potrebno odabrati ime pod kojim će projekt biti poznat, a zatim odabrati željenu lokaciju spremanja. Od tehničkih karakteristika potrebno je odabrati programski jezik koji će se koristiti uz Spring Boot razvojni okvir, alat za automatizaciju izrade (Gradle ili Maven) te nakon odabira Java programskog jezika za razvoj aplikacije potrebno je odabrati verziju Jave te ekstenziju paketa. Osim imena, također je potrebno odabrati naziv grupe (engl. *Group*) koji predstavlja jedinstveno ime korijenskog paketa i identificirano je oznakom `<groupId>` u Apache Maven xml datoteci, a uobičajeno predstavlja naziv organizacije ili pojedinca koji kreiraju projekt te naziv predmeta (engl. *Artifact*) koji predstavlja jedinstveni naziv projekta.[32] Na slici je također vidljivo sve što je odabrano za mikroservis čija je uloga upravljati podacima o korisniku. Nakon toga potrebno je odabrati zavisnosti koje želimo implementirati u naš kod, a odabrane su *Spring Data JPA* (engl. *Java Persistence API*) koja osigurava

[32] Usp. Nicoll, S.; Syer, D. Spring Initializr Reference Guide. URL: https://docs.spring.io/initializr/docs/0.4.x/reference/htmlsingle/ (2023-07-05)

podatke za pohranu u bazi koristeći Spring Data i Hibernate, *Lombok* koji reducira kod te *Spring Web* koji olakšava kreiranje mrežnih aplikacije koristeći Spring MVC (engl. *Model View Controller*) te ima ugrađeni Apache Tomcat server pomoću kojega se aplikacija može pokrenuti. Zavisnosti se uvijek mogu naknadno dodati, što će i ovdje biti učinjeno, a za sada s onim zavisnostima koje su odabrane može se kreirati projekt.

Kreirani projekt ima strukturu kao što je prikazano na Slici 5. Spring Boot automatski kreira klasu u korijenskom paketu u kojem se nalazi *main* metoda i anotacija *@SpringBootApplication*. *Main* je metoda koju Java interpreter poziva u trenutku pokretanja Java programa te se program izvodi sve dok interpreter ne dođe do kraja te metode.[33] Navedena anotacija označava kako je kreirana aplikacija temeljena na Spring Boot razvojnom okviru te ukoliko odaberemo detaljnije pregledavanje navedene anotacije (tipka *Ctrl* na tipkovnici i lijevi klik miša na anotaciju) otvara se klasa naziva *SpringBootApplication* koja sadrži nekoliko dodatnih anotacija. Među tim anotacijama su *@SpringBootConfiguration* koja sadrži definicije Spring konfiguracije, *@EnableAutoConfiguration* za automatsku konfiguraciju aplikacije te *@ComponentScan* za pregledavanje aplikacije tako da pregled započne iz paketa s anotiranom klasom.[34]



Slika 5. Struktura projekta nakon odabira svih potrebnih opcija u Spring Initializr

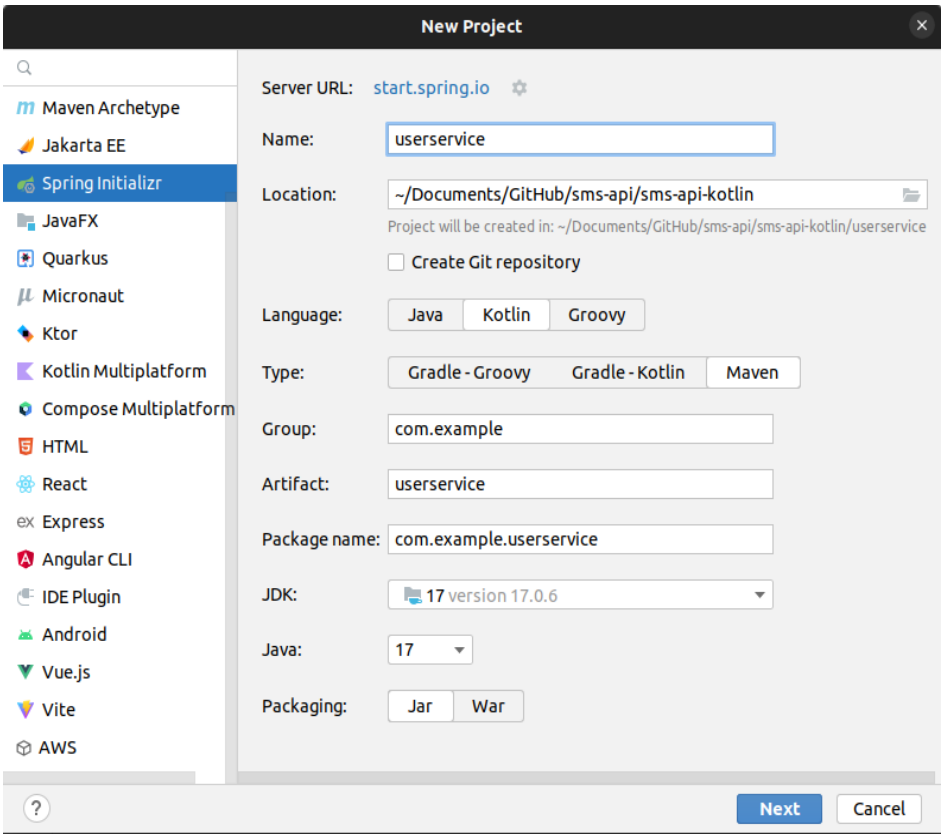
U direktoriju naziva *resources* nalazi se datoteka naziva application.properties i ona predstavlja datoteku u koju se dodatno mogu napisati potrebne konfiguracije, npr. opcije o bazi podataka koja se planira koristiti, promijeniti zadani port na kojemu se aplikacija pokreće

[33] Usp. Flanagan, D. Nav. dj. Str. 15.
[34] Usp. Antonov, A. Spring Boot Cookbook. Birmingham: Packt Publishing, 2015. Str. 6.

i slično. Iako Spring Boot automatski kreira datoteku naziva application.properties, moguće je koristiti application.yml datoteku, odnosno YAML (engl. YAML Ain't Markup Language) jezik za serijalizaciju podataka.[35] Za potrebe ovog rada, prikazat će se struktura i application.properties i application.yml datoteke tako što će se u nekim projektima napraviti datoteka s ekstenzijom .properties, a u nekoliko projekata će ona biti .yml. Također, postoje brojni programi i ekstenzije unutar razvojnih okruženja koji pretvaraju sadržaj .yml ili .yaml datoteka u sadržaj pogodan za .properties datoteku i obrnuto. Sljedeći direktorij kreiran u sklopu Spring Boot projekta je test direktorij kojemu je, kako i samo ime kaže, uloga pisanje testova. S obzirom da u ovom radu nije naglasak na pisanju testova, za potrebe ovoga rada taj direktorij će se preskočiti i neće se koristiti. Zadnja datoteka kojoj će se u ovome radu pridodati značaj je pom.xml datoteka u kojoj su definirane sve deklaracije zavisnosti koje koriste alati za automatizaciju izrade, u ovom slučaju Maven.[36] Deklaracije zavisnosti mogu se dodavati iz Maven ili Mvn repozitorija, dostupnog na poveznici: https://mvnrepository.com/ u kojemu se osim deklaracije za Maven mogu pronaći deklaracije za Gradle, Ivy, Grape i druge.

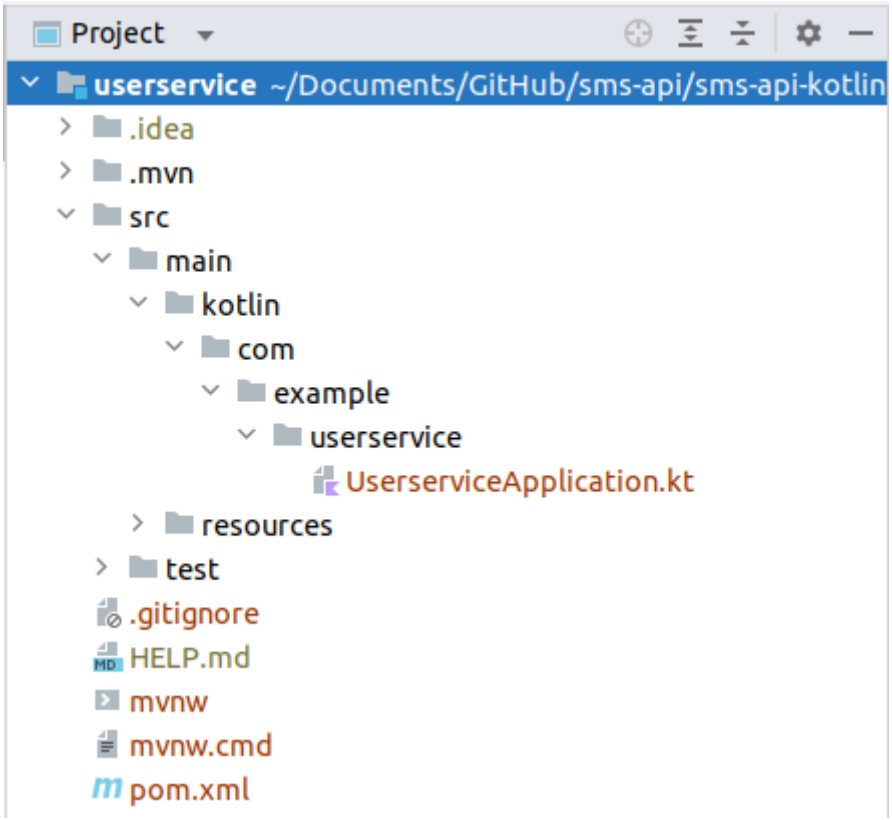
Za potrebe kreiranja aplikacije u Kotlin programskom jeziku također se koristi IntelliJ IDEA razvojno okruženje te Spring Initializr implementiran u navedenom okruženju. Za razliku od prethodnog slučaja, odabran je drugi spomenuti direktorij za pohranu, za programski jezik odabran je Kotlin, dok su ostale karakteristike iste kao i za aplikaciju u Java programskom jeziku (Slika 7.). Iako je Gradle alat za automatizaciju izrade češći u uporabi u Kotlin programskom jeziku od Mavena jer se koristi kao zadani alat kod izrade Android aplikacija, u ovom primjeru odabran je Maven zbog sličnosti s prvim kreiranim projektom u Javi.[37]

[35] Usp. Siva Prasad Reddy, K. Beginning Spring Boot 2: Applications and Microservices with the Spring Framework. Hyderabad: Apress, 2017. Str. 49.
[36] Usp. Antonov, A. Nav. dj., str. 5.
[37] Usp. Roy, A. S.; Karanpuria R. Kotlin Programming Cookbook. Birmingham: Packt Publiching, 2018. Str. 23.



Slika 7. Kreiranje projekta *userservice* kroz IntelliJIDEA pomoću alata Spring Initializr

Zavisnosti su odabrane iste kao u prvom primjeru, a to su *Spring Data JPA*, *Lombok* te *Spring Web*. Klikom na gumb *create* kreira se željeni projekt strukture slične onoj u prethodnom primjeru. Također postoji klasa koja sadrži anotaciju *@SpringBootApplication* te se sastoji od *main* metode koja pokreće ovaj servis. Jedina razlika koja je ovdje najvažnija je što se klase projekta neće nalaziti u direktoriju java, nego u direktoriju naziva kotlin te klase imaju ekstenziju .kt jer su kreirane u Kotlin programskom jeziku. Cjelokupna struktura za sada kreiranog projekta vidljiva je na Slici 8.



Slika 8. Struktura projekta nakon odabira svih potrebnih opcija u Spring Initializr

Zadnji korak u postavljanju okoline kako bi mikroservisi funkcionirali je postavljanje baze podataka. Baza podataka može se lokalno instalirati te lokalno koristiti, ali za potrebe ovoga rada, odabran je način korištenja baze podataka preko Dockera. Docker je alat koji omogućava lako kreiranje, razvijanje i pokretanje aplikacija putem spremnika. Spremnici predstavljaju pakete unutar kojih se nalaze aplikacije zajedno sa zavisnostima i bibliotekama čime olakšavaju daljnje korištenje i produkciju aplikacija.[38] Koraci koje je potrebno slijediti kako bi se osigurala baza podataka preko Dockera su sljedeći: sa službene stranice potrebno je preuzeti (engl. *download*) Docker za operativni sustav koji se koristi, preuzeti s Dockerovog repozitorija (zvanog Docker Hub) sliku baze podataka koju želimo te je potrebno kreirati bazu podataka. Za potrebe ovoga rada, odabrana je baza PostgreSQL koja se preuzima s Docker Huba naredbom kroz terminal

```
docker pull postgres
```

zatim se kreira baza podataka naredbom

[38] Usp. Nuwanthilaka, I. Docker: Zero to Hero (with SpringBoot + Postgres), 2018. URL: https://isurunuwanthilaka.medium.com/docker-zero-to-hero-with-springboot-postgres-e0b8c3a4dccb (2023-07-05)

```
docker run -p 5432:5432 --name postgres_db -e
POSTGRES_PASSWORD=marija123 -d postgres
```

Nakon toga naredbom

```
docker ps -a
```

mogu se izlistati sve slike pohranjene na računalu te naredbom

```
docker start postgres_db
```

pokreće se ona slika koja nam je potrebna (Slika 9.).[39] Važno je napomenuti kako se pojedini dijelovi u prethodno navedenim naredbama mogu promijeniti kao što su mapiranje portova koje je potrebno kako bi port bio vidljiv alatima za upravljanje bazama podataka, ime spremnika (u ovom slučaju postgres_db) te vrijednost lozinke (u ovom slučaju marija123).

```
marija@marija:~$ docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
faef57eae888: Pull complete
a33c10a72186: Pull complete
d662a43776d2: Pull complete
a3ba86413420: Pull complete
a627f37e9916: Pull complete
424bade69494: Pull complete
dd8d4fcd466b: Pull complete
03d0efeea592: Pull complete
4f27e1518a67: Pull complete
0c8ac8b8eb90: Pull complete
c08e79653ad2: Pull complete
d5724e8c22af: Pull complete
3db4aa0d2013: Pull complete
Digest: sha256:362a63cb1e864195ea2bc29b5066bdb222bc9a4461bfaff2418f63a06e56bce0
Status: Downloaded newer image for postgres:latest
docker.io/library/postgres:latest
marija@marija:~$ docker run -p 5432:5432 --name postgres_db -e POSTGRES_PASSWORD=marija123 -d postgres
1a3f56c78c591099efbac4439082fff23383999ba8c63ac484b11c6d3d94e254
marija@marija:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED         STATUS         PORTS                                       NAMES
1a3f56c78c59   postgres   "docker-entrypoint.s…"   7 seconds ago   Up 5 seconds   0.0.0.0:5432->5432/tcp, :::5432->5432/tcp   postgres_db
marija@marija:~$ docker start postgres_db
postgres_db
marija@marija:~$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          PORTS                                       NAMES
1a3f56c78c59   postgres   "docker-entrypoint.s…"   44 seconds ago   Up 42 seconds   0.0.0.0:5432->5432/tcp, :::5432->5432/tcp   postgres_db
```

Slika 9. Pokretanje baze podataka preko Dockera

Nakon uspješnog kreiranja baze podataka, istu je moguće otvoriti u DBeaver alatu ili preko IntelliJ razvojnog okruženja gdje je potrebno napraviti konekciju s PostgreSQL bazom podataka, nakon čega je baza spremna za korištenje.

4. Razvoj mikroservisne arhitekture u Java programskom jeziku pomoću Spring Boot frameworka

Java programski jezik pojavio se krajem 1995. godine te je vrlo brzo postao popularan programski jezik koji se koristi i danas. Kreirao ga je James Gosling u tvrtki Sun Microsystems.[40] Svoju popularnost stekao je idejom kreiranja programskog jezika koji će

[39] Kreirano prema Nuwanthilaka, I. Nav. dj.

[40] Usp. Java (programming language). URL: https://hmong.ru/wiki/Java_language (2023-07-05)

odgovoriti na potrebe tadašnjeg doba. Java se često opisuje kao jednostavan, objektno-orijentirani, interpretirani, dinamički jezik.[41] S obzirom na činjenicu kako je nastao ukidanjem "složenijih" značajki drugih programskih jezika poput C i C++ u cilju kreiranja jednostavnijeg programskog jezika, povećala se i popularnost Java programskog jezika koja ostaje do danas. Za svaki programski jezik, pa tako i za Javu, tijekom vremena su se pojavili razvojni okviri, a kako je već spomenuto, u ovom dijelu rada opisat će se postupak kreiranja nekoliko mikroservisa u Java programskom jeziku pomoću Spring Boot razvojnog okvira.

4.1. Kreiranje mikroservisa za upravljanje podacima o korisnicima

Nakon prethodno postavljene okoline, razvija se željena aplikacija. S obzirom da je ideja kreirati mikroservisnu aplikaciju koja će slati SMS poruke, prvo je potrebno kreirati mikroservis koji će služiti za upravljanje korisničkim podacima. Ideja je aplikacije kreirati interakciju putem REST API standarda. API predstavlja način na koji aplikacije komuniciraju međusobno te se ona najčešće provodi putem HTTP metoda (GET, POST, PUT, PATCH i DELETE). HTTP GET koristi se za dohvaćanje podataka, POST za kreiranje podataka, PUT za promjenu postojećih podataka, PATCH za stvaranje djelomične promjene i DELETE za brisanje.[42] REST API predstavlja sučelje povrh HTTP-a te omogućava lakšu uporabu API-ja, a u Spring Bootu ono se može iskoristiti pomoću *Spring MVC*-a.[43] Primjer u ovom radu prikazat će rad HTTP GET, POST, PUT i DELETE metoda na određenim endpointovima koji predstavljaju URI-je koji pružaju funkcionalnost API-jima. Za njihovo testiranje koristit će se alat Postman, ali prvo se treba kreirati programski kod koji će se sastojati od modela, komponente za procesuiranje i koordinaciju zahtjeva (engl. *controller*) i ostalih potrebnih dijelova, a započinjemo s modelom. Model se sastoji od jedne klase nazvane *User* koja se sastoji od svojstava koje svaki korisnik treba imati (jedinstvenog id-a, imena, prezimena, korisničkog imena, e-maila, lozinke i broja mobitela), jednog praznog konstruktora i jednog konstruktora koji prima argumente svih definiranih svojstava te od nekoliko anotacija kao što su *@Entity* što označava da se klasa smatra JPA entitetom i može biti pohranjena u JPA repozitorij, *@Table(name = "users")* što označava da će klasa biti tablica u bazi podataka imena "users", *@Getter* generira sve pristupnike (engl. *getters*) za svojstva te *@Setter* generira sve promijenitelje (engl. *setters*) za svojstva. Kod svojstva također imamo dvije

[41] Usp. Flanagan, D. Java in a Nutshell. 2. izd. Cambridge… [et al.]: O' Reilly, 1997. Str. 3.
[42] Usp. Gupta, L. HTTP Methods, 2021. URL: https://restfulapi.net/http-methods/ (2023-07-05)
[43] Usp. Macero, M. Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, RIbbon, Zuul and Cucumber. New York: Apress, 2017. Str. 41-42.

anotacije, a to su *@Id* i *@GeneratedValue* koje upućuju kako je to svojstvo primarni ključ te da se automatski generira.[44] Osim *@Getter* i *@Setter* anotacije koje pripadaju *Lombok* zavisnosti, u modelu je dodana još jedna anotacija iz iste zavisnost: *@Builder*. Ta je anotacija zadužena za kreiranje uzorka dizajna kompleksnih objekata, a cilj joj je podijeliti proces instanciranja koristeći druge objekte za konstruiranje jednog objekta.[45] Klasa naziva *User* izgleda ovako:

```
@Entity
@Table(name = "users")
@Builder
@Getter
@Setter
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long idUser;
    private String firstName;
    private String lastName;
    private String username;
    private String mail;
    private String password;
    private String number;

    public User() {
    }

    public User(long idUser, String firstName, String lastName,
String username, String mail, String password, String number) {
                this.idUser = idUser;
        this.firstName = firstName;
        this.lastName = lastName;
        this.username = username;
        this.mail = mail;
        this.password = password;
        this.number = number;
    }
}
```

[44] Usp. Macero, M. Nav. dj., str. 73.

[45] Usp. Introduction to Creational design Patterns, 2022. URL: https://www.baeldung.com/creational-design-patterns#builder (2023-07-06)

Zatim se kreirao repozitorij (engl. *repository*) paket koji sadrži sučelje, a sučelje (engl. *interface*) nasljeđuje *JpaRepository*. *JpaRepository* je sučelje koje omogućava repozitoriju Spring Data implementiranje CRUD (engl. *Create, Read, Update, Delete*) procesa i dodatnih funkcija koje ga razlikuju od *CrudRepository* sučelja koje osigurava samo CRUD operacije.[46] Iako postoji više takvih repozitorija koji se mogu naslijediti, među popularnijima su, uz spomenuta dva (iako *JpaRepository* sadrži *CrudRepository* uz dodatne funkcije), još jedan poznatiji po nazivu *PagingAndSortingRepository* koji je zapravo ekstenzija *CrudRepository* sučelja, a predstavlja dodatne funkcionalnosti kao što su sortiranje i paginacija.[47] Također, u repozitorij klasu dodana je i anotacija *@Query* koja omogućava korištenje JPQL (engl. *Jakarta Persistence Query Language*) što će biti potrebno kod mikroservisa koji će provjeravati validaciju korisnika. Primjer korištenja JPQL-a je prikazan sljedećom metodom:

```
@Query("select u from User u where u.username=:username")
User findByUsername(@Param("username") String username);
```

Nakon modela i repozitorija, slijedi izrada servisa (engl. *service*) i kontrolnog (engl. *controller*) sloja. Servisni dio aplikacije implementira poslovnu logiku, dok je kontrolni dio zadužen za procesuiranje i koordinaciju zahtjeva koje usmjeravaju prema određenim metodama te ih vraćaju kao odgovore korisnicima. Spring Boot anotacijom označava koji je dio aplikacije za što zadužen. Tako je za repozitorij anotacija *@Repository*, za servis *@Service*, a za kontroler *@Controller*. S obzirom da *@Controller* anotacija kreira kontroler koji vraća izgled (engl. *view*) koji ovdje ne postoji, za potrebe ove aplikacije koristi se *@RestController* anotacija koja je kreirana s ciljem vraćanja podataka u JSON ili XML formatu.[48] Servisni dio aplikacije, sljedeći primjere dobre prakse, sastoji se od sučelja i klase. Klasa implementira sučelje i sve njegove metode čime se olakšava mijenjanje servisa u budućnosti, odnosno implementacija više sučelja ukoliko je to u budućnosti potrebno. Tako sučelje sadrži opisane metode, dok klasa sadrži anotaciju *@Service* i implementaciju metoda. Metode koje su opisane u servisu su metoda za dohvaćanje svih korisnika, dohvaćanje jednog korisnika prema jedinstvenom ključu (id), kreiranje novog korisnika, ažuriranje postojećeg korisnika i brisanje. Metoda koja dohvaća korisnika prema jedinstvenom ključu u klasi *UserServiceImpl* izgleda ovako:

---

[46] Usp. Gutierrez, F. Nav. dj. Str. 33.

[47] Usp. Antonov, A. Nav. dj., str. 23.

[48] Usp. Difference between @Controller and @RestController in Spring Boot and Spring MVC?, 2020. URL: https://medium.com/javarevisited/difference-between-controller-and-restcontroller-in-spring-boot-and-spring-mvc-216578ad445f (2023-07-05)

```
@Override
public User getUserById(long id) {
    return userRepository.findById(id).get();
}
```

Također, kod kreiranja i mijenjanja podataka, kako lozinka korisnika ne bi bila javno vidljiva u bazi, odabran je mehanizam hashiranja (engl. *hashing*) lozinke koji enkodira lozinku prema određenom algoritmu kako bi lozinku pretvorili u nečitljiv niz znakova.[49] U ovom slučaju korišten je Argon2i mehanizam, a sve što je potrebno kako bi se lozinka hashirala je dodati zavisnost za Argon2 u pom.xml datoteku te se on može iskoristiti na sljedeći način

```
String hashPassword = argon2.hash(2, 1024, 4, user.getPassword());
```

nakon čega se za vrijednost lozinke koristi varijabla *hashPassword*. Kako bi se u metodi za brisanje osiguralo hvatanje iznimki ukoliko korisnik koji se želi obrisati ne postoji, dodana je datoteka koja hvata iznimku te ispisuje problem nepostojanja korisnika. Zatim se kreira paket za rad sa zahtjevima, nazvan *controller*. U njemu se nalazi klasa koja sadrži anotacije *@RestController* i *@RequestMapping* te sadrži metode kao u servisnom dijelu koje označava posebnim anotacijama kako bi se omogućilo korištenje REST metoda na određenim *endpointovima*. Primjer GET metode za dohvaćanje korisnika prema jedinstvenom ključu u *controller* klasi je

```
@GetMapping("/{id}")
public ResponseEntity<UserDto> getUserById(@PathVariable("id") int id){
        User user = userService.getUserById(id);
        UserDto userResponse = modelMapper.map(user, UserDto.class);
        return ResponseEntity.ok().body(userResponse);
}
```

S obzirom da ova aplikacija nema vidljivi dio za korisnike, ideja je bila prilikom slanja određenih REST metoda dobiti poruku i status poslanog zahtjeva te je iz tog razloga uvedena posebna klasa koja prilikom dobivanja odgovora upućuje na API dokumentaciju. Za API dokumentaciju kloniran je GitHub repozitorij dostupan na poveznici: https://github.com/floriannicolas/API-Documentation-HTML-Template te su se HTML i CSS datoteke proizvoljno mijenjale, odnosno dodana je datoteka za svaki *endpoint*. Prikaz izgleda HTML stranice s API dokumentacijom dostupan je na Slici 10. te na poveznici: https://oziz.ffos.hr/nastava20192020/mmilosevic_19/DiplomskiRad/HTML-API-Docs/. Također, u

[49] Usp. Walls, Craig. Spring in Action. New York: Manning Publications, 2022. Str. 117.

datoteku pom.xml dodane su dvije dodatne zavisnosti: *model mapper* i *spring boot starter validation*. *Model mapper* je biblioteka koja služi za automatsko mapiranje objekta jer se u aplikaciji odlučilo za korištenje DTO (engl. *Data Transfer Objects*) klasa. DTO klase čine rad s REST API-jima sigurnijima jer pružaju pristup podacima bez izlaganja JPA entiteta.[50] Kako bi *model mapper* radio potrebno ga je u datoteci s *main* metodom označiti kao *@Bean*. Time ga definiramo kao objekt kojim upravlja spremnik inverzije kontrole (IoC) te se koristi u slučajevima kada se ne može provesti automatska konfiguracija kao u slučajevima korištenja anotacije *@Component*, *@Service*, *@Repository* i *@Controller*.[51]

Slika 10. Izgled mrežne stranice za API dokumentaciju

Kao posljednji korak za potrebe DTO klase dodana je zavisnost *spring boot starter validation* kako bi se mogle dodati anotacije *@NotNull* koja sprječava nepoznavanje vrijednosti, odnosno vrijednost *null* i *@NotBlank* koja sprječava nedostatak vrijednosti tako da prvo zanemari sve razmake (engl. *trimmed length*) pa tek onda provjerava postoji li vrijednost.[52] Također, ukoliko se ne koristi Spring Boot ili Spring razvojni okvir umjesto korištene zavisnosti za provjeru vrijednosti, moguće je koristiti zavisnost *hibernate validator* koja je uključena u *spring boot starter validation*. U zagradama nakon *@NotNull* i *@NotBlank* nalaze se poruke koje će se ispisati, a u ovoj aplikaciji upućuju na datoteku koja se nalazi u

[50] Usp. Fadatare, R. Spring Boot DTO Example - Entity To DTO Conversion, 2021. URL: https://www.javaguides.net/2021/02/spring-boot-dto-example-entity-to-dto.html (2023-07-06)
[51] Usp. Webb, P. … [et al.]. Spring Boot Reference Documentation. Str. 35. URL: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/pdf/spring-boot-reference.pdf (2023-07-06)
[52] Usp. Ugarte, A. Difference Between @NotNull, @NotEmpty, and @NotBlank Constraints in Bean Validation, 2023. URL: https://www.baeldung.com/java-bean-validation-not-null-empty-blank (2023-07-06)

resources te su u toj datoteci napisane sve poruke koje treba ispisati ukoliko jedan od vrijednosti podataka nedostaje. Primjer ograničavanja nepostojanja podataka je sljedeći:

```
@NotBlank(message = "validation.firstNameMandatory")
@NotNull(message = "validation.firstNameMandatory")
private String firstName;
```

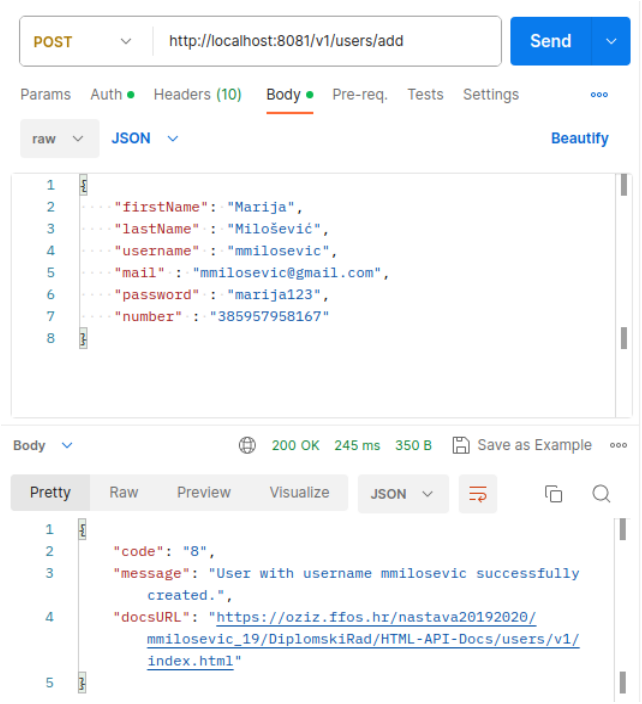
Dok u application.properties datoteku treba postaviti konfiguraciju za upućivanje na drugu datoteku:

```
spring.messages.basename=lang/messages
```

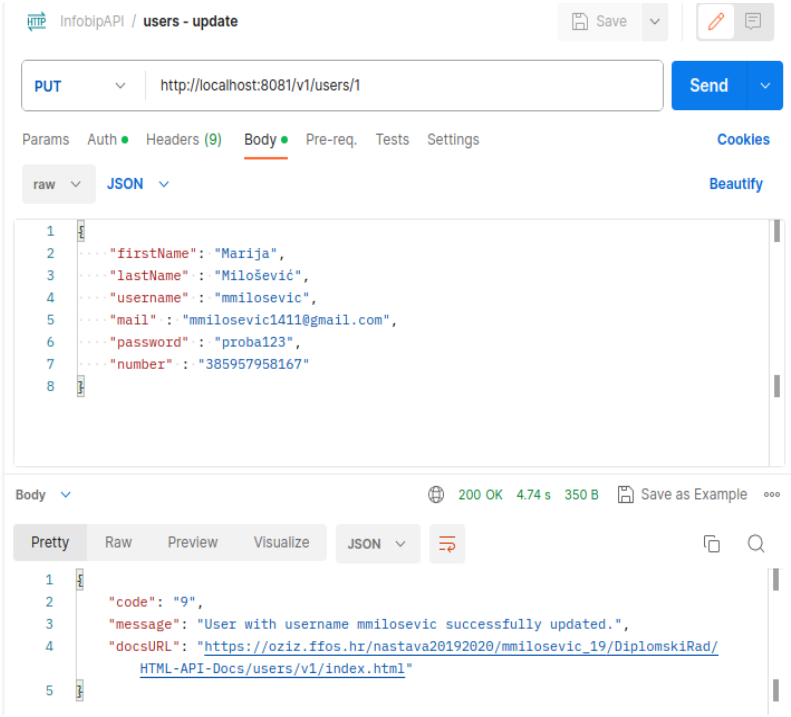
Kako je cjelokupni kod gotov za ovaj dio mikroservisne arhitekture, ostaje još povezati bazu podataka kako bi se spremali podaci. Za bazu podataka koristi se PostgreSQL preko Docker alata zbog čega je potrebno prvo dodati zavisnost za PostgreSQL i u application.properties datoteku napisati upute aplikaciji kako će se i na koju bazu spojiti. Konfiguracija izgleda ovako:

```
spring.jpa.hibernate.ddl-auto=update
spring.sql.init.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/smsApi_
db
spring.datasource.username=postgres
spring.datasource.password=marija123
spring.jpa.show-sql= true
```

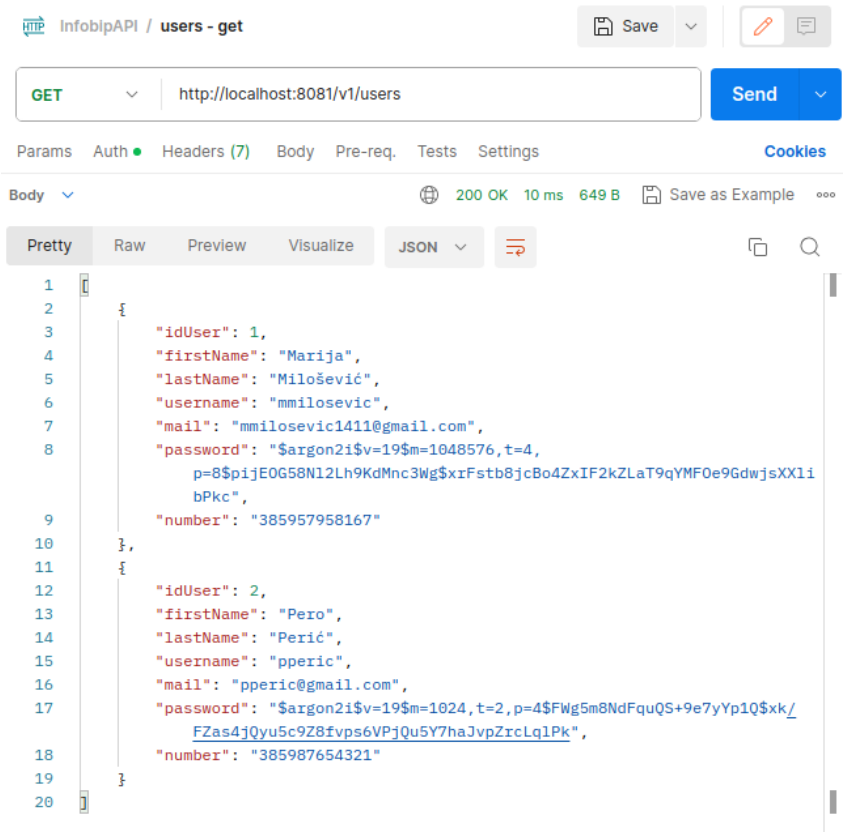
Zatim se kreirani *endpointovi* definirani u kontroleru mogu testirati preko Postman alata. *Endpointovi* su se odnosili na POST metodu (Slika 11.), PUT (Slika 12.), GET (Slika 13. i 14.) i DELETE metodu (Slika 15.). Također izgled baze podataka preko DBeaver alata vidljiv je na Slikama 16. i 17. Cjelokupni programski kod mikroservisa koji služi za upravljanje podataka o korisnicima dostupan je na GitHub repozitoriju na sljedećoj poveznici: https://github.com/marmilosev/sms-api/tree/main/sms-api-java/userservice.



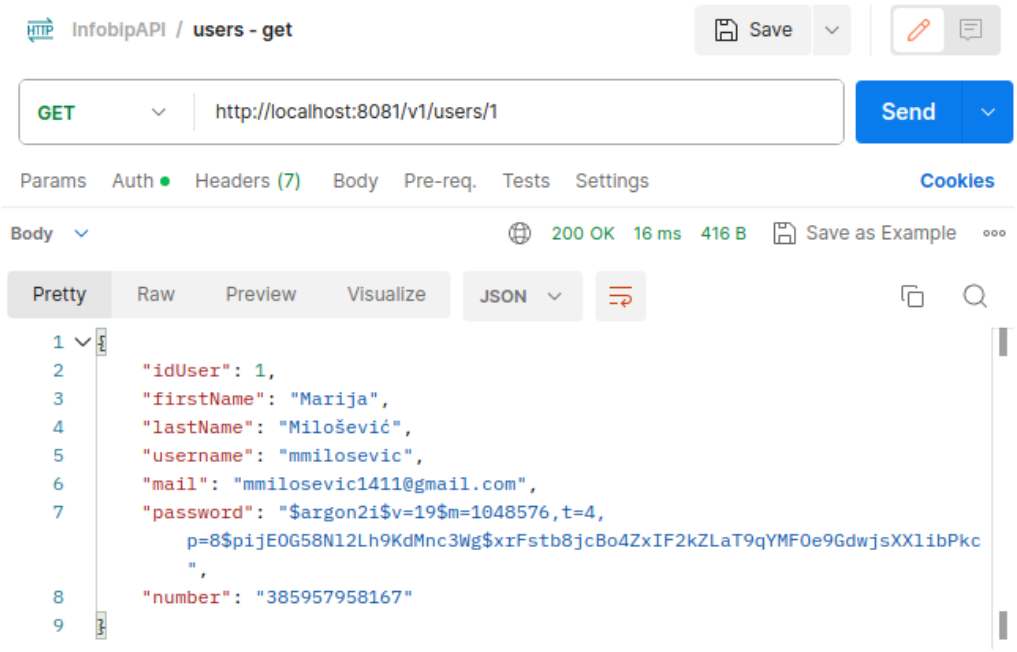
Slika 11. POST zahtjev za kreiranjem korisnika te ispis odgovora i statusa zahtjeva



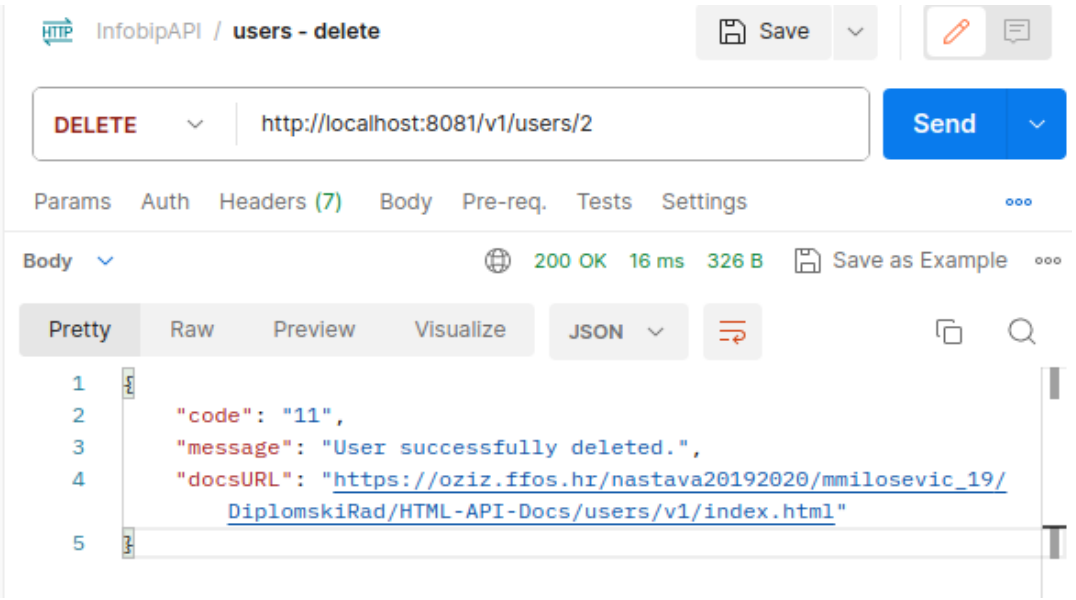
Slika 12. PUT zahtjev za mijenjanje mail-a korisnika te ispis odgovora i statusa zahtjeva



Slika 13. GET zahtjev za ispis svih korisnika te ispis odgovora i statusa zahtjeva



Slika 14. GET zahtjev za ispis korisnika s id-jem 1 te ispis odgovora i statusa zahtjeva



Slika 15. DELETE zahtjev za brisanjem korisnika s id-jem 2 te ispis odgovora i statusa zahtjeva

	id_user	first_name	last_name	mail	number	password	username
1	1	Marija	Milošević	mmilosevic1411@gmail.com	385957958167	$argon2i$v=19$m=10485	mmilosevic
2	2	Pero	Perić	pperic@gmail.com	385987654321	$argon2i$v=19$m=1024,	pperic

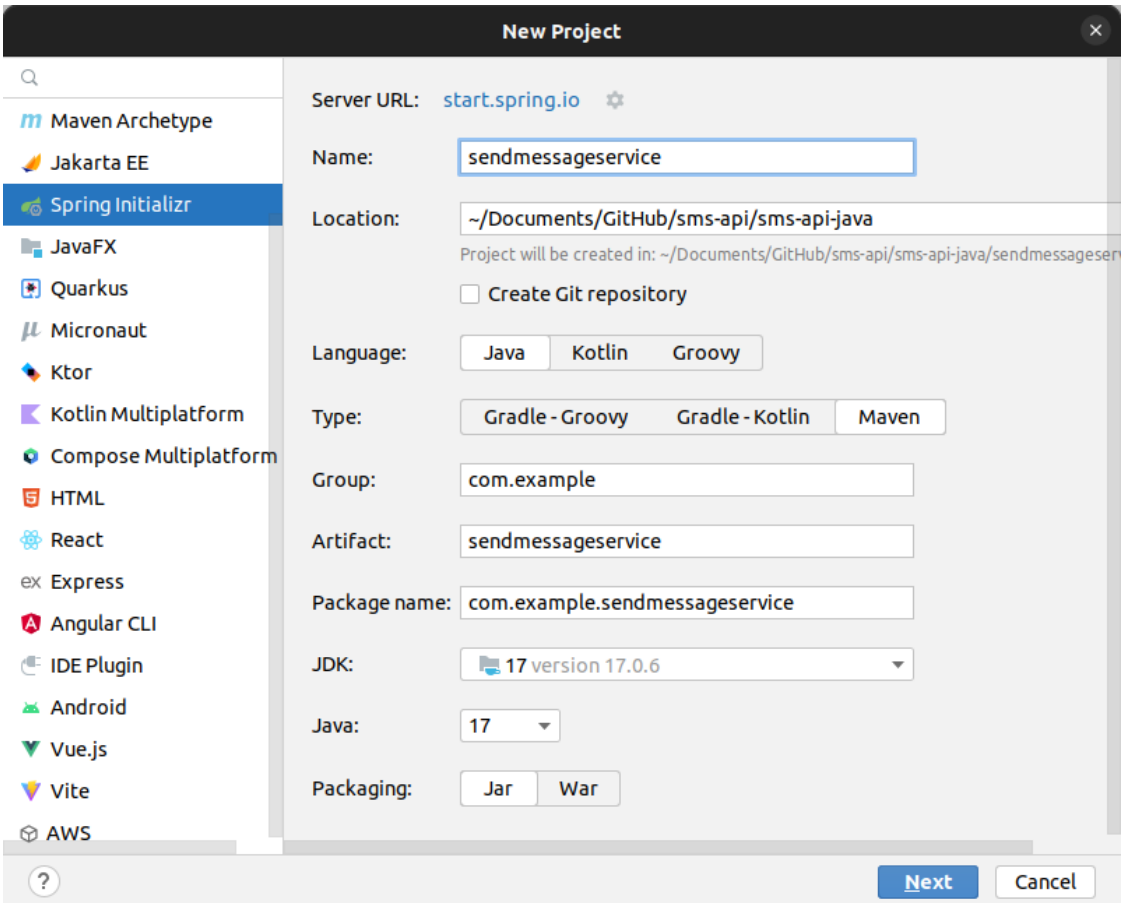
Slika 16. Baza podataka prije izvršavanja DELETE zahtjeva

	id_user	first_name	last_name	mail	number	password	username
1	1	Marija	Milošević	mmilosevic1411@gmail.com	385957958167	$argon2i$v=19$m=10485	mmilosevic

Slika 17. Baza podataka poslije izvršavanja DELETE zahtjeva

4.2. Kreiranje mikroservisa za slanje poruke

Ideja sljedeće kreiranog mikroservisa je kreirati projekt koji će slati poruke preko Infobipovog API-ja za slanje SMS poruke. Prilikom kreiranja projekta, odabrane su iste tehničke karakteristike kao i u prethodnom mikroservisu, uz razliku u imenu (Slika 18.), dok su za zavisnosti također odabrane iste tri zavisnosti kao u prvom primjeru.



Slika 18. Kreiranje projekta *sendmessageservice* kroz IntelliJ IDEA pomoću alata Spring Initializr

Kreirani projekt ima istu strukturu kao i prethodni te je početak kreiranja aplikacije isti, odnosno kreiraju se paketi za model, repozitorij, servis i kontroler, ali u ovom mikroservisu će se oni orijentirati na poruke kako bi se na kraju odradilo uspješno slanje SMS poruke. Svi navedeni paketi imat će sličnu strukturu kao i u prvom mikroservisu zajedno s istim anotacijama te će koristiti istu bazu podataka. Prvo se unutar paketa model kreira klasa koja predstavlja svojstva poruke, a jedina je ovdje razlika u odnosu na onu iz prvog mikroservisa ta što postoji anotacija *@ManyToOne* koja označava kako poruka može imati jednog korisnika, odnosno pošiljatelja, ali da korisnik može poslati više poruka.[53] Također je korištena još jedna dodatna anotacija, *@JoinColumn* koja označava reference na druge entitete.[54] S obzirom da se *Message* klasa povezuje s *User* klasom u model paketu moraju postojati obje klase, a također neizostavna je postojanost takvih klasa za DTO. Klasa naziva *Message* definirana je na sljedeći način:

[53] Usp. Walls, Craig. Nav. dj., str. 137.
[54] Usp. Macero, M. Nav. dj., str. 37.

```
@Entity
@Table(name = "messages")
@Getter
@Setter
public class Message {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long idMessage;
    @ManyToOne(cascade = {CascadeType.MERGE})
    @JoinColumn
    private User user;
    private String number;
    private Date dateTime;
    private String messageText;

    public Message() {
    }
}
```

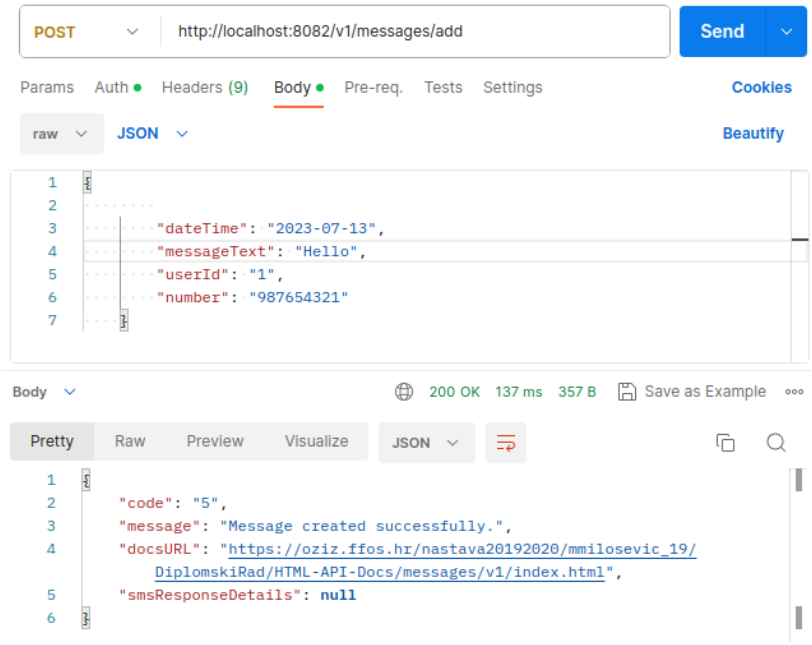
I u ovom slučaju kreirala se posebna klasa za prikaz određenih poruka prilikom obrade POST, PUT, GET i DELETE zahtjeva, a metode koje se nalaze u servis i kontroler paketima su iste kao i u mikroservisu za upravljanje korisničkim podacima. Iako je u ovom dijelu projekta naglasak na slanju poruke preko API-ja koji je kreirala tvrtka Infobip, kontroler se podijelio u dvije klase, jedna koja omogućava REST metode, dok je drugi kreiran za navedenu potrebu slanja poruke. Tako za potrebe prvog kontrolera kojemu je cilj odraditi POST, PUT, GET i DELETE zahtjeve, također su se kao i u prvom primjeru, dodale zavisnosti za *model mapper i spring boot starter validation*. Najznačajnija razlika u ovom mikroservisu je dodavanje zavisnosti *spring boot starter webflux* koji omogućava rad *WebClienta*. Na taj je način osmišljena komunikacija između mikroservisa, odnosno na taj je način omogućeno pozivanje udaljenih REST servisa, u ovom slučaju onih koji se nalaze u prvom mikroservisu naziva *user service*. *WebClient* zamjenjuje u prošlosti korišten *RestTemplate* te pruža više funkcionalnosti korištenjem *Buildera*.[55] Tako je i za potrebe ovoga rada upotrijebljen na sljedeći način:

```
@Autowired
private WebClient.Builder webClientBuilder;
```

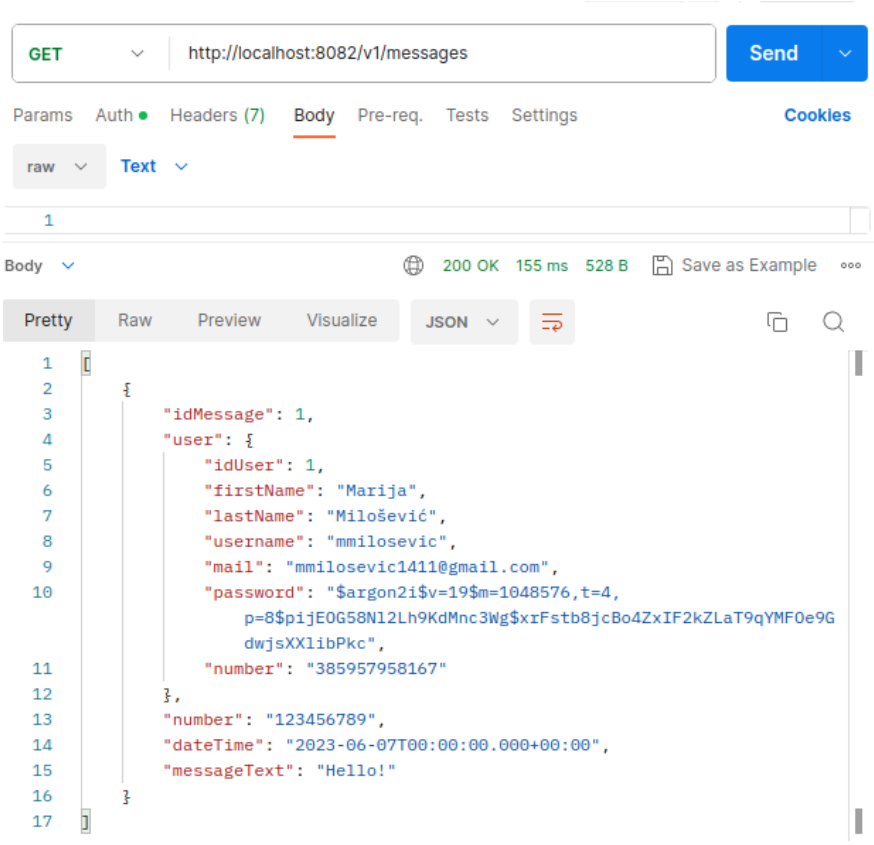
čime je osigurana komunikacija između dva mikroservisa. Na Slikama 19. i 20. prikazane su POST i GET metode na ovaj način, dok PUT i DELETE nisu prikazane jer su slične

[55] Usp. Webb, P. … [et al.]. Nav. dj., str. 431.

prijašnje prikazanim metodama. Također, važno je napomenuti kako je u oba projekta promijenjen zadani port u application.properties.



Slika 19. POST zahtjev za kreiranje poruke te ispis odgovora i statusa zahtjeva



Slika 20. GET zahtjev za ispis svih poruka te ispis odgovora i statusa zahtjeva

Za potrebe slanja SMS poruke, koristi se Infobipov API Java klijent, a prije toga potrebno je registrirati se na stranicu Infobipa. Svaki registrirani korisnik tako dobije *API Base URL* (engl. *Uniform Resource Locator*) te jedinstveni *API Key* koji je potreban za slanje SMS poruka. Osim SMS poruka, Infobip osigurava brojne druge API-je kao što su API za slanje MMS-ova, WhatsApp poruka, Viber poruka i slično. S obzirom da se ovdje testiralo SMS poruke te se koristio besplatni probni period, rad je bio ograničen na sto poruka i slanje samo s jednog registriranog broja mobilnog uređaja. Koristeći dokumentaciju dostupnu na Infobipovoj stranici, za svaki se API mogu pregledati dostupni HTTP zahtjevi koje je moguće odraditi za određeni API te se za svaki zahtjev mogu pogledati podaci koji se mogu poslati te primjeri kodova u različitim programskim jezicima. Registriranjem na stranicu, otvara se korisnički profil gdje se registrira određeni broj mobilnog uređaja, prati potrošnja poruka ili mijenja *API Key* ukoliko je on istekao ili se javno podijelio. Prvi korak u osposobljavanju projekta za slanje poruka je dodavanje zavisnosti pod nazivom *infobip api java client* čime klasa *ApiClient* postaje dostupna za korištenje, a ona omogućuje uvrštavanje dobivenog *API Keya* i *API Base URL-a* u aplikaciju. *API Key* i *API Base URL* u ovom se slučaju, prateći primjere dobre prakse, koriste kao konstante definirane u application.properties te su kao konstante korištene u dijelu koda koji procesuira zahtjeve, odnosno u kontroleru na sljedeći način:

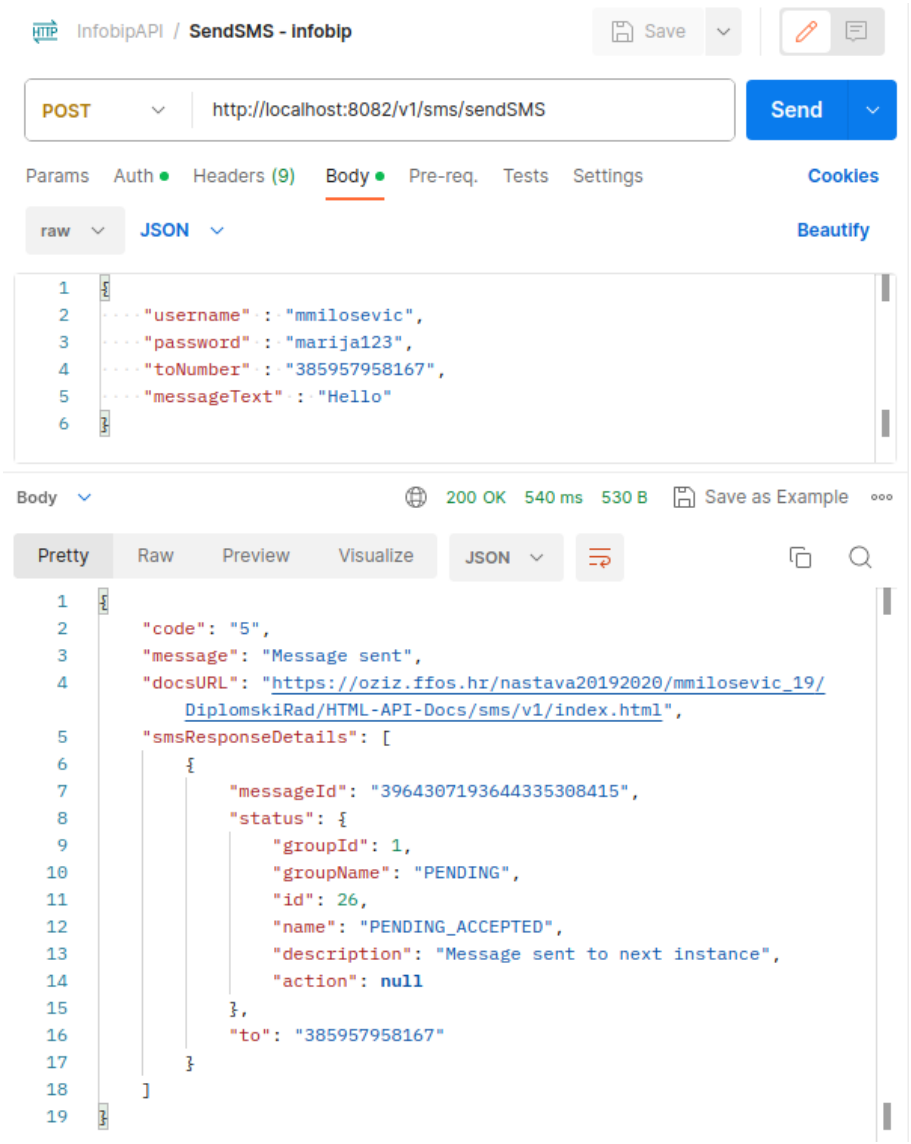
```
@Value("${infobip.apiKey}")
private String apiKey;
@Value("${infobip.baseUrl}")
private String baseUrl;
@PostConstruct
public void init() {
   this.apiClient = ApiClient.forApiKey(ApiKey.from(apiKey))
           .withBaseUrl(BaseUrl.from(baseUrl))
           .build();
}
```

Dodavanjem navedene zavisnosti, osim *ApiClient* klase, dostupnim postaju i niz drugih klasa te metoda koje su potrebne za slanje SMS poruke. Ukratko opisano, klasa *SmsController* koja služi za slanje poruka sastoji se od metode označene anotacijom *@PostMapping* koja upućuje na endpoint preko kojega se odrađuje POST zahtjev te anotacijom *@ResponseBody* koja serijalizira vraćenu vrijednost kao HTTP odgovor.[56] Dio koda za slanje poruke vidljiv je u nastavku:

[56] Usp. Webb, P. … [et al.]. Nav. dj., str. 612.

```
SmsTextualMessage smsMessage = new SmsTextualMessage()
        .from(userResponse.getFirstName() + " " +
                    userResponse.getLastName())
        .addDestinationsItem(new
              SmsDestination().to(smsRequest.getToNumber()))
        .text(smsRequest.getMessageText());
```

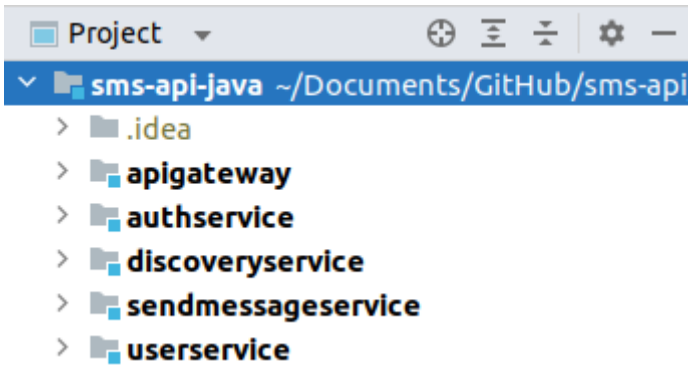
Metoda kreira niz novih instanci koje služe za korištenje vrijednosti instanciranih klasa te dobivanja odgovora od POST poslanog zahtjeva koji je vidljiv na Slika 21. Cjelokupni kod dostupan je na: https://github.com/marmilosev/sms-api/tree/main/sms-api-java/sendmessageservice.



Slika 21. Uspješno poslana poruka

4.3. Kreiranje mikroservisa za provjeru valjanosti korisničkog imena i lozinke, za usmjeravanje komunikacije i otkrivanje svih kreiranih mikroservisa

Ostale mikroservise koje je potrebno kreirati su mikroservis za provjeru valjanosti korisničkog imena i lozinke te mikroservis za otkrivanje svih mikroservisa u projektu, kao i mikroservis koji će usmjeravati komunikaciju s jednog mikroservisa na drugi, poznatiji kao *API Gateway*. Prvi korak je kreirati dodatna tri projekta, odnosno mikroservisa, u postojeći direktorij. Struktura direktorija sada izgleda kao na Slici 22.



Slika 22. Struktura direktorija sa svim kreiranim mikroservisima u Java programskom jeziku

Kreirani mikroservis pod nazivom *user service* zadužen je za upravljanje podacima o korisnicima, dok je kreirani mikroservis naziva *send message service* zadužen za slanje SMS poruke. Kao što je već navedeno, mikroservis *auth service* služi za provjeru valjanosti korisničkih podataka, dok su preostala dva mikroservisa namijenjeni za komunikaciju i otkrivanje svih mikroservisa. Prvi korak je definirati uloge dodjeljujući svakom mikroservisu zavisnost radi li se o Eureka klijentu ili Eureka serveru. Eureka alat za otkrivanje servisa (engl. *Service Discovery Client*) je Neflixov alat integriran u Spring Cloud koji prepoznaje servise unutar projekta na temelju zavisnosti *spring cloud starter netflix eureka server* i *spring cloud starter netflix eureka client*.[57] Sljedeći korak je unutar konfiguracije svakog mikroservisa definirati zadani URL na kojem se nalazi Eureka server te definirati ime svakog mikroservisa. Navedeni dio konfiguracije izgleda ovako:

```
eureka.client.registerWithEureka=true
eureka.client.fetchRegistry=true
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
spring.application.name=user-management-service
```

[57] Usp. Macero, M. Nav. dj., str. 203.

gdje se mijenja spring.application.name u odnosu na naziv mikroservisa za koji definiramo konfiguraciju, dok u mikroservisu čija je uloga otkriti druge mikroservise konfiguracija izgleda drugačije sintaksom jer se koristi .yml datoteka:

```
spring:
 application:
 name: discovery-service

server.port: 8761
eureka:
 instance:
   preferIpAddress: true
 renewalPercentThreshold: 0.85
 client:
   registerWithEureka: false
   fetchRegistry: false
```

Zatim je potrebno definirati kod za mikroservis koji će služiti kao provjera vjerodostojnosti korisnika, ovdje nazvanim *auth service*. Prvo se kreira DTO klasa za korisnika koja će sadržavati korisničko ime i lozinku, a na temelju korisničkog imena i lozinke ideja je da servis, odnosno kontroler, kao rezultat prikaže JWT (engl. *JSON Web Token*) token. JWT token predstavlja standard za sigurni prijenos informacija u obliku JSON objekta, a mogu biti kreirani pomoću engl. *secret* koristeći HMAC (engl. *Hash-based Message Authentication Code*) algoritam ili preko javnog i privatnog ključa koristeći RSA (engl. *Rivest-Shamir-Adleman* nazvan prema tvorcima) ili ECDSA (engl. *Elliptic Curve Digital Signature Algorithm*).[58] Token se sastoji od zaglavlja (engl. *header*) koji se sastoji od dva dijela: vrste tokena (JWT) i algoritma (HMAC ili RSA), zatim se sastoji od tvrdnji (engl. *payload*) o entitetu, a one mogu biti registrirane, javne i privatne tvrdnje. I kao zadnji dio tokena čini potpis koji se sastoji od enkodiranog zaglavlja, enkodiranih tvrdnji i algoritma.[59] Kako bi se JWT token iskoristio u Spring Boot aplikaciji potrebno je dodati njegovu Java implementaciju kroz zavisnost. Osim DTO klase, kreirana je još kontroler klasa i paket za upravljanje tokenima gdje postoji sučelje i klasa koja implementira to sučelje. U sučelju su definirane dvije metode za generiranje tokena i provjeru valjanosti generiranog tokena čija je logika definirana u implementaciji:

[58] Usp. Mool, S. Json Web Token (JWT), 2019. URL: https://medium.com/@mool.smreeti/json-web-token-jwt-2ba5d032685e (2023-07-07)
[59] Usp. Isto.

```
public interface TokenManager {
    String generateToken(UserDto userDto);
    void validateToken(String token);
}
```

Kontroler klasa također ima metode koje se pozivaju POST metodom te se izvode na temelju prethodne dvije metode iz sučelja. Tako upisivanjem korisničkog imena i lozinke dobivamo generirani JWT token (Slika 23.) koji se drugom metodom može provjeriti. Metoda za vraćanje JWT tokena prikazana je u nastavku:

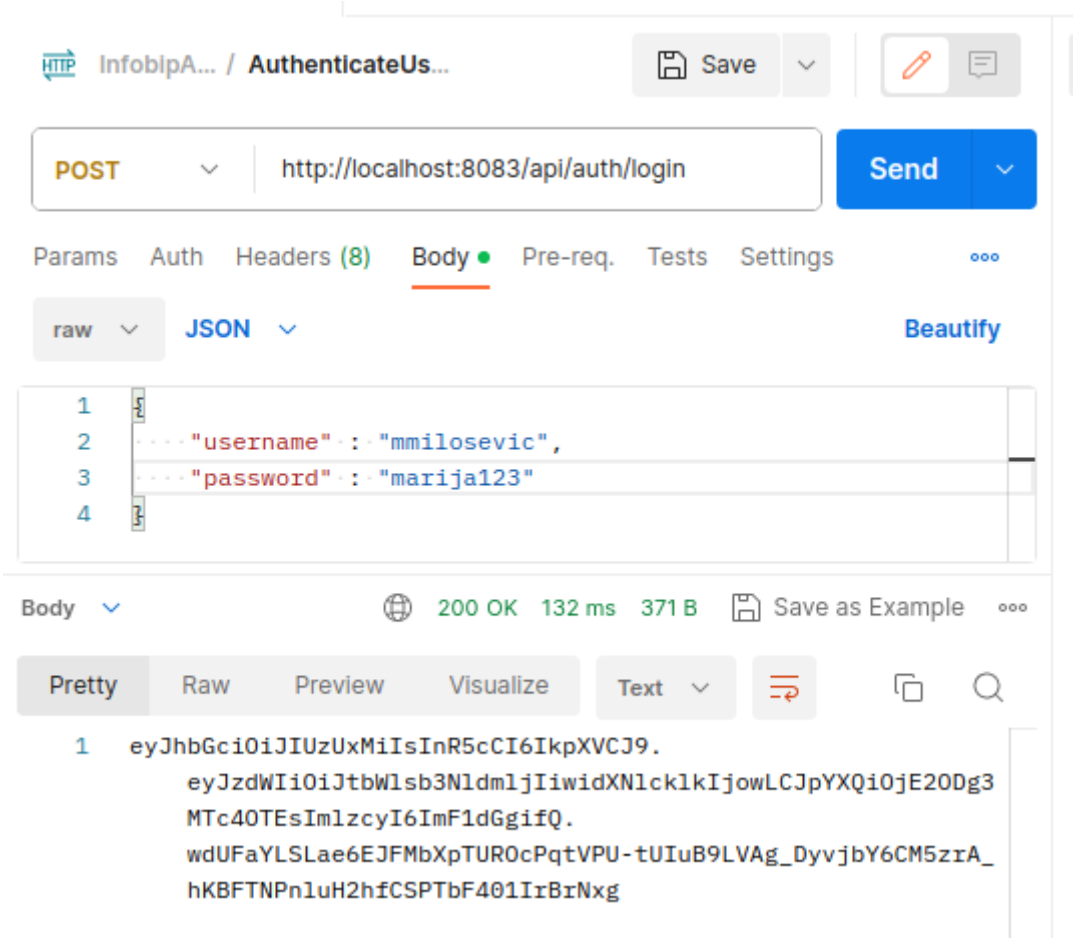
```
@PostMapping("login")
      public ResponseEntity<String> authenticate(@RequestBody
UserDto                  userDto) {
       logger.info("Received validate request.");
            return new

       ResponseEntity<>(tokenManager.generateToken(userDto),
                        HttpStatus.OK);
}
```

Također je u ovom mikroservisu potrebno osigurati registraciju korisnika zbog čega se u kontroler klasi kreirala nova POST metoda za registraciju korisnika, uz određene dodatne klase koje su kreirane kako bi navedena metoda funkcionirala. Prilikom pokretanja mikroservisa za provjeru vjerodostojnosti potrebno je prvotno pokrenuti mikroservis za otkrivanje drugih servisa. No, prije toga potrebno je kod svih mikroservisa u datoteku gdje se nalazi *main* metoda, ispod anotacije *@SpringBootApplication* dodati anotaciju *@EnableEurekaServer* za *discovery service* koji aktivira agenta za otkrivanje, dok u svim drugima je potrebno dodati *@EnableDiscoveryClient*.[60] Također je važno napomenuti kako je u pom.xml datoteci *discovery service* potrebno zavisnosti napraviti kompatibilnima. Moguća je pojava greške zbog koje se mikroservis neće pokrenuti zbog toga što dolazi do nepodudaranja *Spring Cloud* i *Spring Boot* zavisnosti zbog čega je u ovom primjeru zavisnost *spring cloud starter* postavljena na verziju 4.0.2, dok je *spring cloud starter eureka server* postavljena na verziju 1.4.7. RELEASE. Cijela matrica kompatibilnosti dostupna je na Spring Cloud mrežnoj stranici.[61]

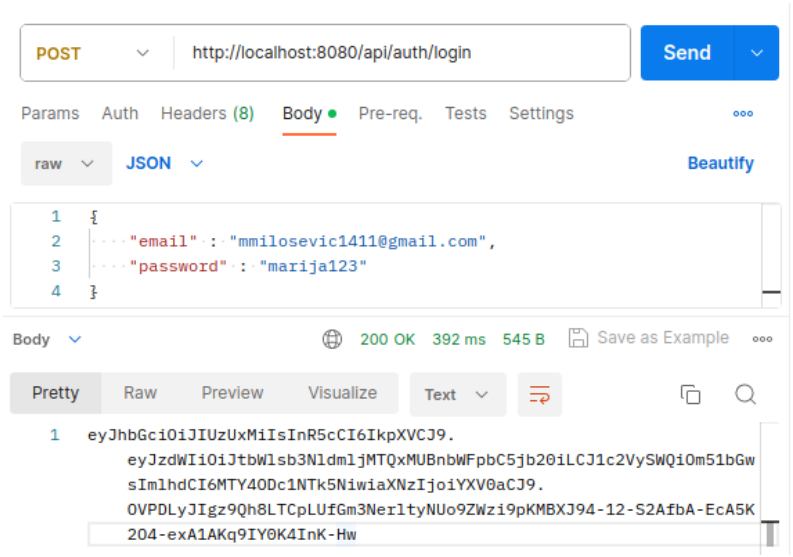
[60] Usp. Macero, M. Nav. dj., str. 232.

[61] Usp. Spring Cloud, 2022. URL: https://spring.io/projects/spring-cloud (2023-07-07)

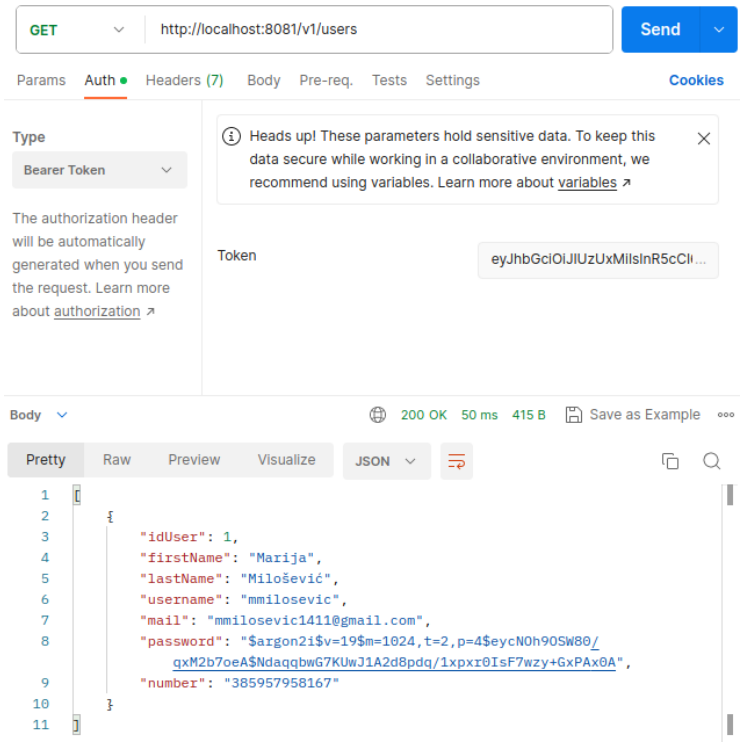


Slika 23. Generirani JWT token na temelju korisničkog imena i lozinke

Daljnji postupak temelji se na osposobljavanju *gateway* mikroservisa. Nakon dodavanja potrebne anotacije u datoteku s *main* metodom i odgovarajuće konfiguracije, u application.properties datoteku dodaje se dodatna konfiguracija koja definira kojim se *endpointovima* može pristupiti bez i s JWT tokenima, odnosno za pristup kojim metodama korisnik mora biti prvotno prijavljen u sustav. Upravo se *api gateway* mikroservis najviše razlikuje od ostalih po svojim zavisnostima jer je potrebno dodati *spring cloud starter gateway* te nekoliko zavisnosti koje su potrebne za sigurnost (engl. *security*) koda zbog korištenja JWT tokena. Ideja je *gatewaya* biti mikroservis preko kojega se upućuju svi zahtjevi. Oni koji se mogu izvršiti bez JWT tokena su dopušteni upisujući *endpoint*, dok oni koji nisu rezultiraju statusom 401 koji označava neautoriziranog (engl. *unauthorized*) korisnika. Primjer zahtjeva koji se može izvršiti bez autorizacije prikazan na Slici 24., dok je na Slici 25. prikazano slanje zahtjeva pomoću JWT tokena.

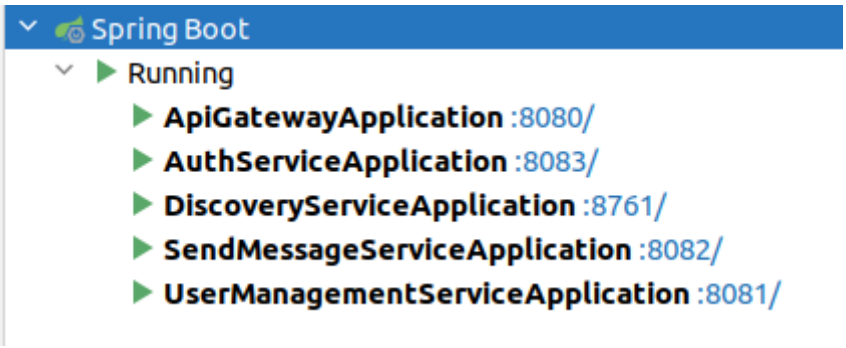


Slika 24. Prikaz dobivanja JWT tokena preko *api gateway* mikroservisa



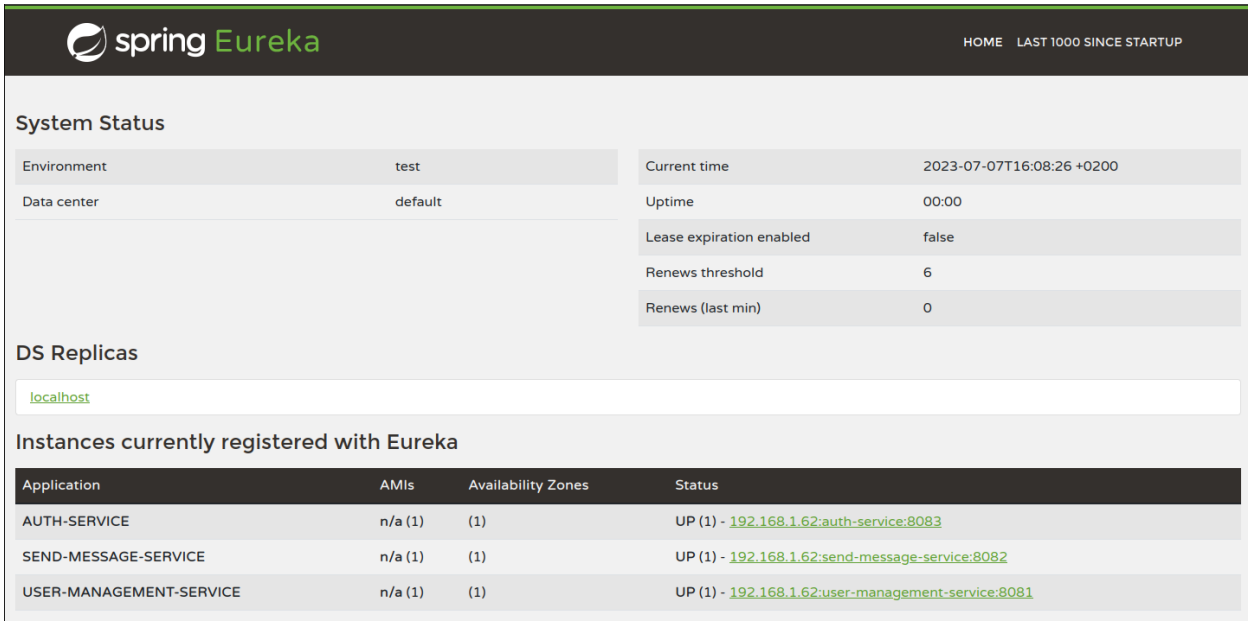
Slika 25. GET metoda koja vraća podatke pomoću JWT tokena

Kao što je vidljivo na Slici 25., zahtjevi više ne idu preko porta svakog zasebnog servisa, već preko *gatewaya* koji prepoznaje svaki drugi mikroservis te obavlja poslane zahtjeve. Struktura pokrenute mikroservisne arhitekture vidljiva je na Slici 26.



Slika 26. Pokrenuti mikroservisi

Također, provjera pronalazi li Eureka server sve druge mikroservise moguća je upisivanjem u tražilicu internetskog preglednika localhost: i broj porta na kojemu je Eureka pokrenuta, u ovom slučaju localhost:8761 gdje se otvara stranica na kojoj se vide svi mikroservisi označeni kao instance (Slika 27.).



Slika 27. Prikaz stranice u internetskom pregledniku kada je Eureka server pokrenut i pronalazi sve mikroservise

5. Razvoj mikroservisne arhitekture u Kotlin programskom jeziku pomoću Spring Boot frameworka

Programski jezik Kotlin razvila je kompanija JetBrains 2010. godine koja je većinu svojih proizvoda kreirala u Java programskom jeziku te su htjeli kreirati "poboljšanu" verziju Jave. Kotlin je dizajniran za rad na svakoj platformi, posebno na Android, JVM (engl. *Java Virtual*

Machine), JavaScript i Native.[62] Iako se u široj javnosti pojavio tek šest godina nakon njegove izrade, trenutno broji više od pet milijuna korisnika čemu doprinosi informacija da je 2019. godine proglašen dominantnim jezikom za razvijanje Android aplikacija.[63] Kao vodeću prednost Kotlin programskog jezika, brojni programeri i stručnjaci iz istog područja, navode njegovu interoperabilnost s Java programskim jezikom čime se postiže neometana mogućnost korištenja Java razvojnih okvira i biblioteka. Primjer korištenja Spirng Boot Java razvojnog okvira za kreiranje mikroservisne arhitekture u Kotlin programskom jeziku prikazat će se u nastavku.

5.1. Kreiranje mikroservisa za upravljanje podacima o korisnicima

S obzirom da je prethodno postavljena cjelokupna radna okolina za razvoj aplikacije, rad se nastavlja u direktoriju kreiranom za Kotlin programski jezik i kreiranom *user service* projektu. Iz istog razloga nije bilo potrebno kreirati nove postavke za potrebe baze podataka, već su upotrijebljene postojeće, jedino se za potrebe spremanja podataka obrisala postojeća baza podataka i kreirala nova istog imena. Takva baza spremna je za korištenje te se nastavlja s razvojem mikroservisa za upravljanje podacima o korisnicima, odnosno *user service* projektom ili mikroservisom. Kao i u prethodnom poglavlju, cilj je aplikacije isti te se iz tog razloga na isti način kreira aplikacija u Kotlin programskom jeziku. Prva dva mikroservisa bazirat će se na radu s REST API-jima putem kojih se mogu poslati HTTP zahtjevi (GET, POST, PUT i DELETE) kako bi se izvršila određena metoda definirana određenim zahtjevom u kontroler klasi. Paketi i klase se kreiraju na isti način te se prvo kreira paket model s klasom naziva *User*. Prilikom kreiranja klase IntelliJ IDEA razvojno okruženje pruža mogućnost kreiranja Kotlin klase i Kotlin datoteke. Razlika je u tome što klasa sadrži samo jednu klasu, dok datoteka može sadržavati dvije ili više klasa.[64] S obzirom na korištenje jedne klase po jednoj datoteci, koristit će se opcija kreiranja Kotlin klase. Tako prva kreirana klasa za opis svojstva (engl. *property*) korisnika izgleda:

```
@Entity
@Table(name = "users")
class User (
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

[62] Usp. What exactly is Kotlin?, 2022. URL: https://codeop.tech/what-exactly-is-kotlin/ (2023-07-09)
[63] Usp. Isto.
[64] Usp. Moskala, M.; Wojda, I. Android Development with Kotlin. Birmingham: Packt Publishing, 2017. Str. 23.

```
    var idUser: Long,
    var firstName: String,
    var lastName: String,
    var username: String,
    var mail: String,
    var password: String,
    var number: String
)
```

Sličnost je vidljiva u korištenju anotacija jer se koriste iste zavisnosti, osim anotacija *@Getter* i *@Setter* čiji je razlog nedostajanja objašnjen u sljedećem poglavlju. Također, kod definiranja svojstava potrebna je ključna riječ var (ili val) ovisno o tome želi li se definirati promjenjiva ili nepromjenjiva vrijednosti. Ono što se može primijetiti na danom primjeru je to što iza definiranog imena klase ne slijede vitičaste zagrade, nego oble. Kada su svojstva definirana u zaglavlju klase, to se naziva primarni konstruktor te je na taj način definiran parametar primarnog konstruktora i tako parametar konstruktora postaje svojstvo.[65] Sljedeće se kreira repozitorij sučelje koji kao i u prvom primjeru nasljeđuje *JpaRepository* sučelje za lakše korištenje operacija koje će se izvesti na poziv HTTP metoda. Repozitorij sučelje je označeno anotacijom *@Repository*, dok je metoda unutar repozitorija označena *@Query* anotacijom te je kreirana na isti način kao i prethodna aplikacija u Java programskom jeziku uz razliku u sintaksi zbog korištenja drugog programskog jezika:

```
@Query("select u from User u where u.username=:username")
fun findByUsername(@Param("username") username: String): User
```

Na isti način kreiran je i servis paket koji implementira poslovnu logiku i sadrži anotaciju *@Service*. Navedeni paket poslovne logike sadrži sučelje s definiranim metodama i klasu koja implementira to sučelje kako bi definirala postupke koje metode provode. Primjer metode za dohvaćanje korisnika prema jedinstvenom ključu prikazana je u nastavku:

```
override fun getUserById(id: Long): User = userRepository.
        findByIdOrNull(id) ? : throw
    ResponseStatusException(HttpStatus.NOT_FOUND)
```

Prikazana metoda naslijeđena je iz sučelja što označava *override* ključna riječ. Nakon nje slijedi *fun* što označava metodu, odnosno funkciju prema engl. *function*, a zatim slijedi naziv metode s parametrom i tipom podatka koji metoda vraća. Sve ostalo nakon znaka jednakosti predstavlja tijelo metode koje prvo pronalazi određenu metodu u repozitoriju koja je zadana u *JpaRepository* te će vratiti ili *User* ili *null* vrijednost. Zadnji dio nakon druge dvotočke

[65] Usp. Hagos, T. Learn Android Studio 3 with Kotlin: Efficient Android App Development. Manila: Apress, 2018. Str. 94

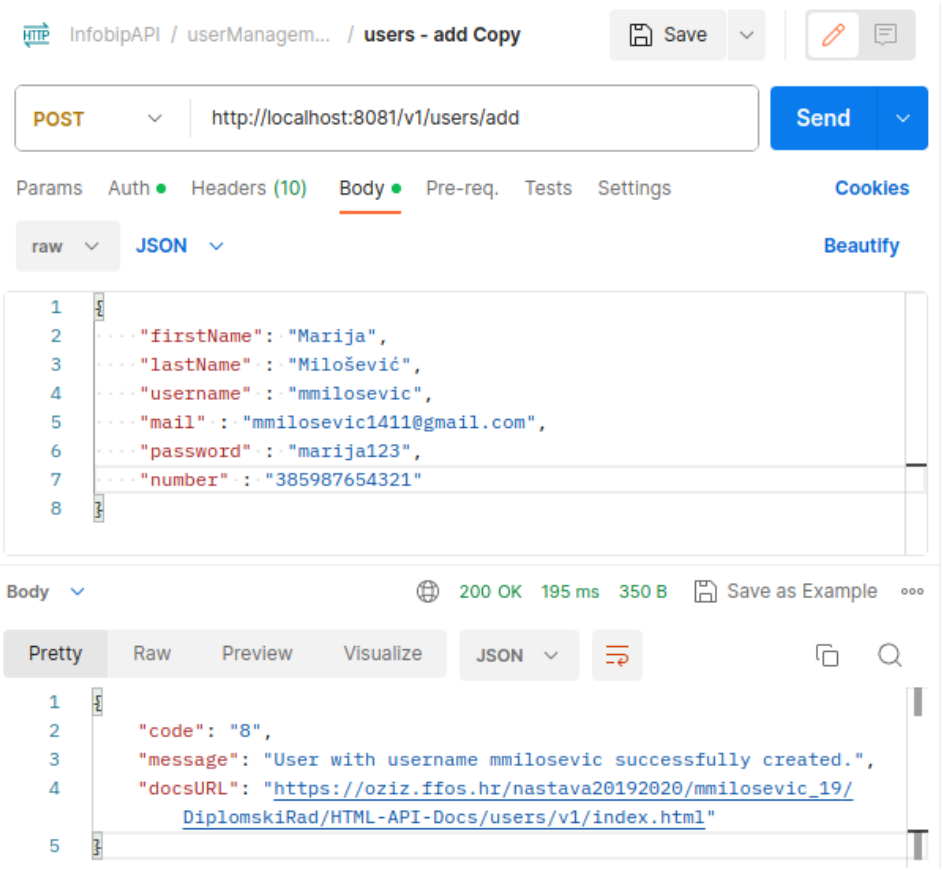
definira HTTP status koji je potrebno prikazati ukoliko *User* s određenim *id-j*em ne postoji. Metode koje su opisane su identične onima u prethodnoj aplikaciji te se također za hashiranje lozinke koristi Argon2i mehanizam:

```
        val argon2 : Argon2 = Argon2Factory.create
(Argon2Factory.                    Argon2Types.ARGON2i)
var hashPassword = argon2.hash(2, 1024, 4, user.password)
```

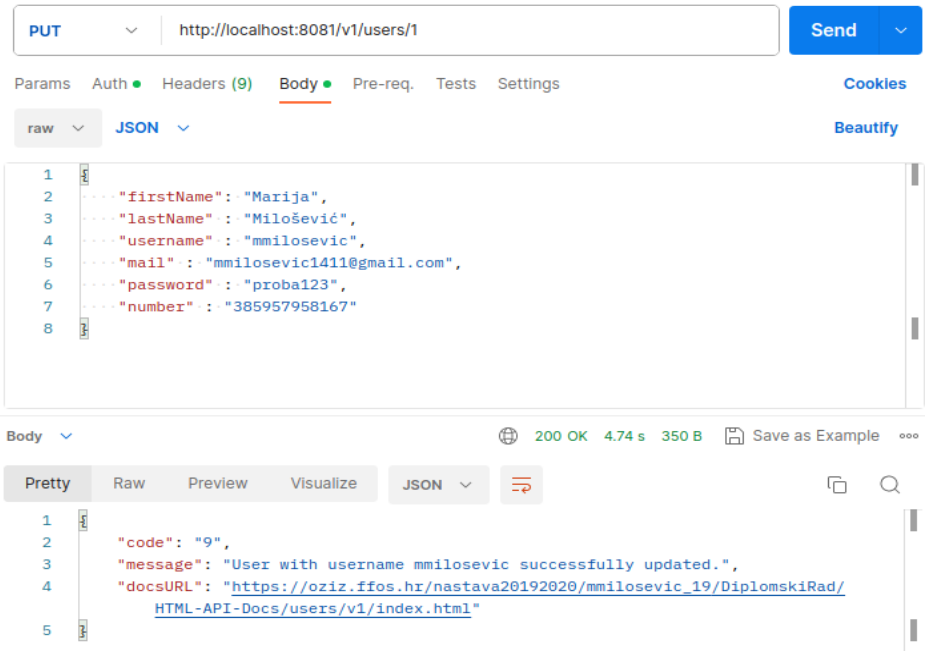
Kontroler klasa sadrži iste endpointove kao i aplikacija u Javi te također definira *endpointove* pomoću kojih se pristupa određenim metodama, a povezana je sa servisnom klasom. U nastavku, kao i u prethodnom primjeru s Java programskim kodom, prikazana je metoda za dohvaćanje korisničkih podataka korisnika s određenim *id*-em.

```
@GetMapping("/{id}")
fun getUser(@PathVariable id: Long): ResponseEntity<UserDto>{
      var user = userServiceImpl.getUserById(id)
      var userDto = user.toUserDto()
      return ResponseEntity.ok().body(userDto)
}
```

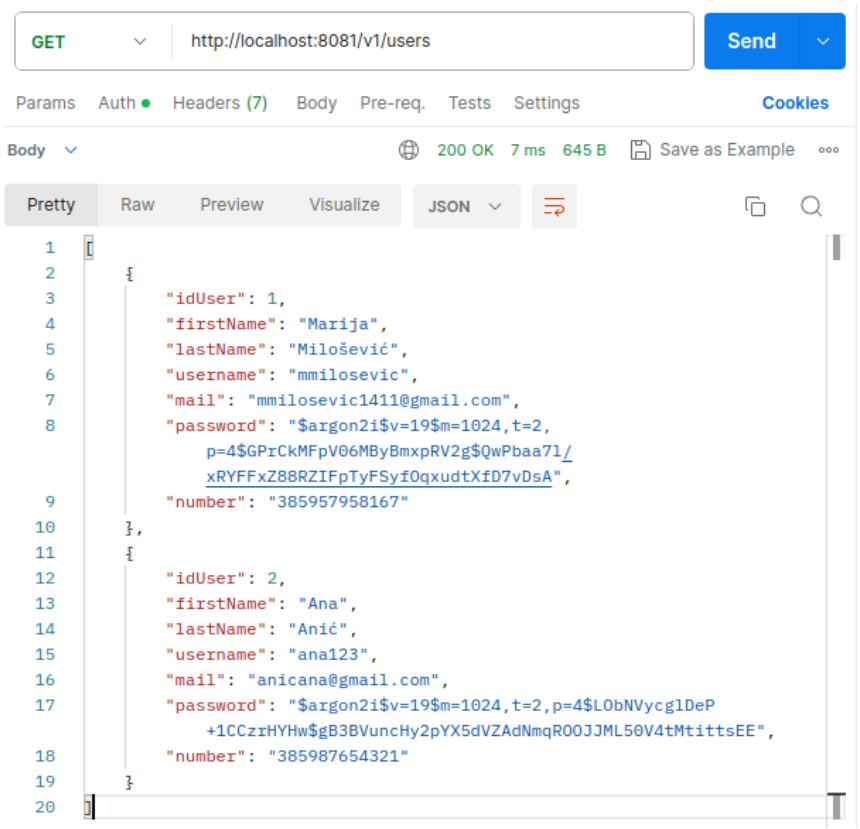
Za potrebe API dokumentacije kreirana je dodatna klasa s određenim svojstvima koja prikazuju potvrdnu poruku ili poruku pogreške, a svaka poruka je popraćena poveznicom na mrežnu stranicu s detaljnim objašnjenjem. Također za tu je potrebu iskorištena ista mrežna stranica kao i u četvrtom poglavlju ovoga rada. Zatim se kreirala DTO klasa koja ovaj puta nije povezana sa zavisnosti *model mapper* te je korištena metoda pretvaranja obične klase u DTO klasu. Također zbog načina rada Kotlina, nije bilo potrebe dodavati zavisnost *spring boot starter validation* i anotacija *@NotNull* i *@NotBlank* jer ukoliko se ne postavi stroga nulta sigurnost (engl. *strict null safety*) program automatski provjerava te javlja grešku ukoliko je vrijednost određenog podatka *null*. Zadnji korak je povezivanje s bazom podataka u application.properties datoteci koja izgleda identično kao i u Java programskom jeziku, promjena porta na kojemu će aplikacija raditi te testiranje REST API metoda. Na sljedećim slikama dostupan je prikaz poslanih zahtjeva, POST zahtjev (Slika 28.), PUT zahtjev (Slika 29.), GET zahtjevi (Slika 30. i 31.) te DELETE (Slika 32.). Cjelokupni kod prvog mikroservisa dostupan je na poveznici: https://github.com/marmilosev/sms-api/tree/main/sms-api-kotlin/userservice.



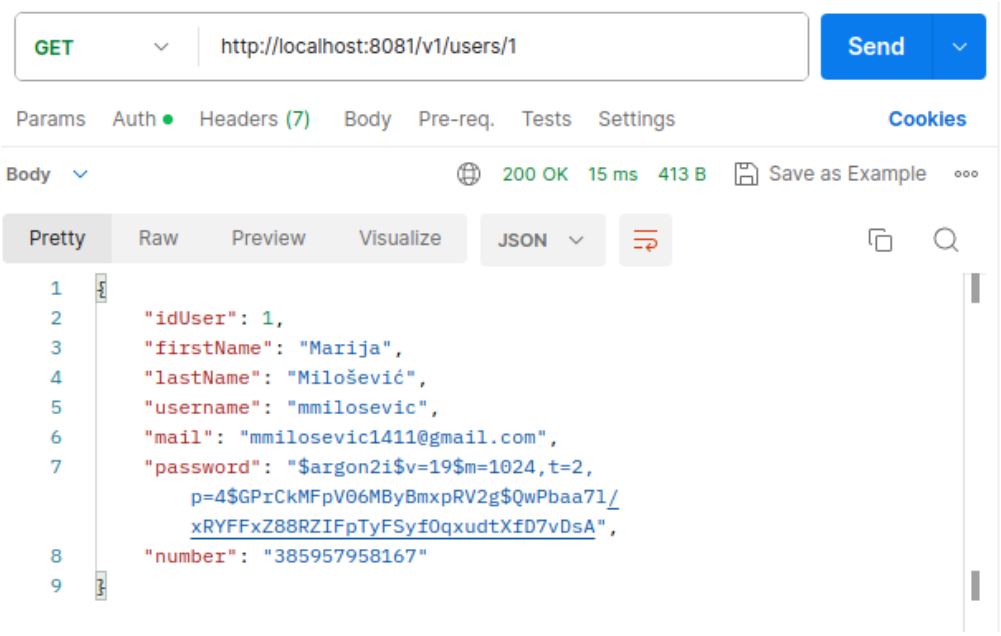
Slika 28. POST zahtjev za kreiranjem korisnika te ispis odgovora i statusa zahtjeva



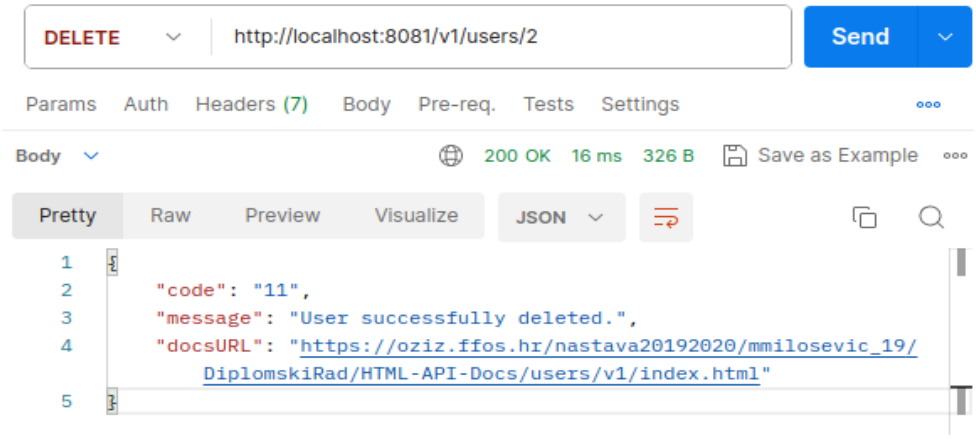
Slika 29. PUT zahtjev za mijenjanjem broja mobitela korisnika te ispis odgovora i statusa zahtjeva



Slika 30. GET zahtjev za ispis svih korisnika te ispis odgovora i statusa zahtjeva



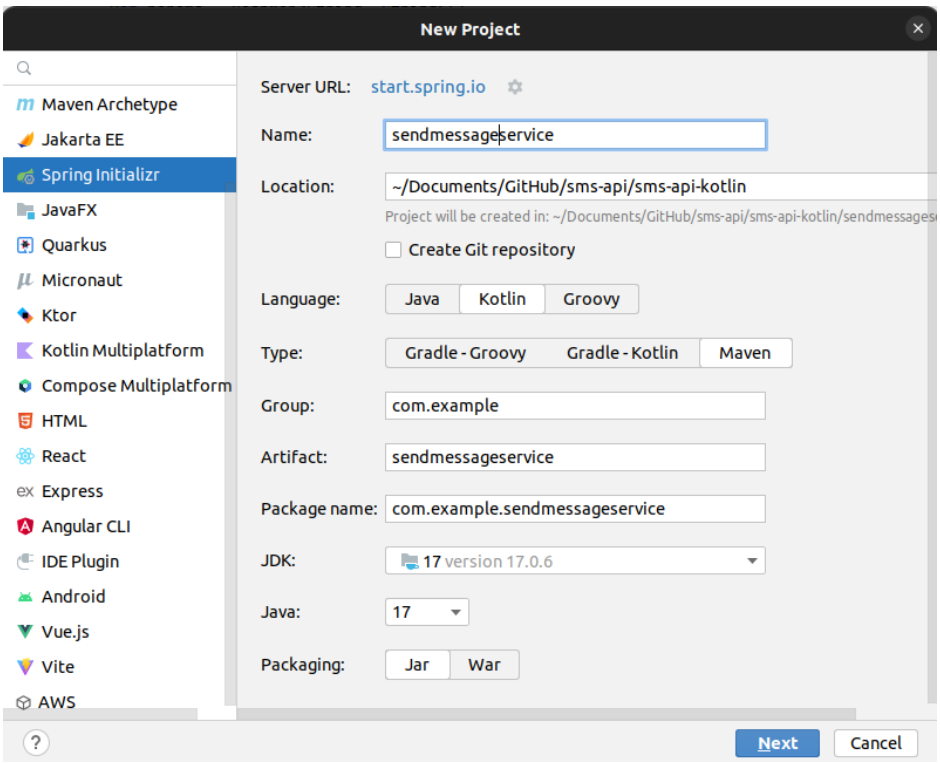
Slika 31. GET zahtjev za ispis korisnika s id-jem 1 te ispis odgovora i statusa zahtjeva



Slika 32. DELETE zahtjev za brisanjem korisnika s id-jem 2 te ispis odgovora i statusa zahtjeva

5.2. Kreiranje mikroservisa za slanje poruke

U ovom će se poglavlju kreirati mikroservis koji služi za slanje poruka pomoću Infobipovog SMS API-ja. Prvi korak je kreiranje novog projekta u direktoriju gdje se nalazi projekt opisan u prethodnom potpoglavlju, a kreirani projekt ima iste zavisnosti i tehničke karakteristike kao i prethodno kreirani projekt (Slika 33.).



Slika 33. Kreiranje projekta *sendmessageservice* kroz IntelliJ IDEA pomoću alata Spring Initializr

Kao i prethodno kreirani projekt u Kotlinu, i ovaj će slijediti strukturu i funkcionalnost prvotno kreiranog projekta u Java programskom jeziku. Tako se prvo kreira paket naziva model koji sadrži klase koje će opisati svojstva poruke i korisnika. Klasa korisnik je potrebna zbog anotacije *@ManyToOne* koja se koristi u klasi za poruku. Obje klase definirane su anotacijama *@Entity*, *@Table* te kod svojstva koje definira primarni ključ uključena je anotacija *@Id* i *@GeneratedValue*. *Message* klasa izgleda ovako:

```
@Entity
@Table(name = "messages")
class Message (
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var idMessage: Long? = null,
    var number: String? = null,
    var dateTime: Date? = null,
    var messageText: String? = null,
    @ManyToOne
    var userId: User? = null
)
```

Na sličan način kreirane su i DTO klase, uz razliku što ne postoji navedena anotacija, već je id korisnika koji se povezuje uz poslanu poruku označen kao *Long* tip podataka jer je uvijek spremljen kao broj. Također svojstva unutar DTO klase definirana su kao primarni konstruktor te su im vrijednosti postavljene na inicijalnu vrijednost *null* i osigurana je dozvola nepoznavanja tipa podataka stavljanjem znaka upitnik nakon tipa podataka.[66] DTO klase su označene kao klase s podacima što je omogućeno dodavanjem ključne riječi *data* ispred riječi *class* koja označava klasu. Time je označeno kako DTO klase sadrže samo podatke i nikakve druge funkcionalnosti.[67] Sljedeće je kreiran repozitorij koji ima funkciju nasljeđivanja *JpaRepository* i svih metoda koje je moguće iskoristiti iz tog repozitorija, odnosno sučelja. Servis paket sadrži pet metoda koje sadrže logiku kreiranja, izlistanja, brisanja i ažuriranja podataka o porukama i samih poruka koje su osigurane implementacijom repozitorija. Navedene metode su *saveMessage()*, *getAllMessages()*, *getMessageById()*, *deleteMessage()* te *updateMessage()*. Kontroler dio podijeljen je na dva dijela. Prvi dio, kao i u ostalim slučajevima, definira *endpoint*e preko kojih se izvršavaju CRUD operacije na temelju metoda definiranih u servis klasama. Kao i u ostalim primjerima, za potrebe kontrolera kreirana je

[66] Usp. Roy, A. S.; Karanpuria R. Nav. dj., str. 73.
[67] Usp. Hagos, T. Nav. dj., str. 96-99.

dodatna klasa koja sadrži poruke potvrde ili pogreške prilikom izvršavanja određenih upita. Također u ovom je primjeru dodana zavisnost *model mapper* i *spring boot starter validation* kako bi se osigurala komunikacija s prvim mikroservisom. Drugi dio kontrolera zadužen je za slanje poruka koje se temelji na Infobipovom API ključu i URL-u. Obje navedene vrijednosti definirane su kao konstante u application.properties datoteci te na taj način definirane u kontroler klasi. Navedene vrijednosti *API Keya* i *API Base URL-a* te način njihovog dobivanja opisani su u prethodnom poglavlju, a u ovom su poglavlju oni iskorišteni tako što su iste vrijednosti iskorištene na isti način:

```
@Value("\${infobip.apiKey}")
lateinit var apiKey: String
@Value("\${infobip.baseUrl}")
lateinit var baseUrl: String
@PostConstruct
fun init() {
    apiClient = ApiClient.forApiKey(ApiKey.from(apiKey))
        .withBaseUrl(BaseUrl.from(baseUrl))
        .build()
}
```

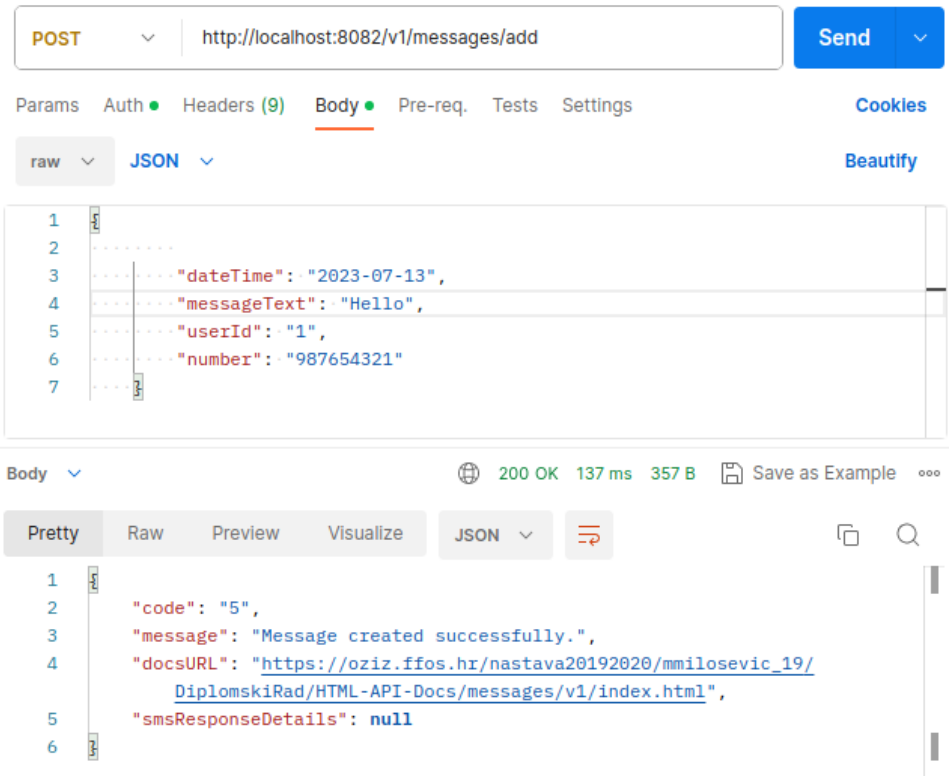
U definiranju konstanti koristimo ključnu riječ *lateinit* koja označava ono što bi se doslovno moglo prevesti kao kasna inicijalizacija. To znači da je spriječena inicijalizacija svojstva u vrijeme konstruiranja objekta njegova klase.[68] Ostatak koda sadrži istu logiku kao i kontroler opisan u potpoglavlju *4.2. Kreiranje mikroservisa za slanje poruke* koji se odnosi na mikroservisnu arhitekturu u Java programskom jeziku te poput klase u Javi, i kontroler klasa u Kotlinu sadrži anotacije *@RequestMapping* i *@PostMapping* koje upućuju da je za slanje upita potrebno definirati *endpoint* koji glasi *v1/sms/sendSMS*. Također, iako je dio programskog koda namijenjen slanju poruka u Java programskom jeziku kreiran na temelju dokumentacije za korištenje SMS poruka, ovdje je u potpunosti kreiran na temelju dijela programa u Javi zbog nedostatka dokumentacije u Kotlin programskom jeziku. Isti dio koda koji je prikazan u Java programskom jeziku, nalazi se u nastavku te je napisan u Kotlin programskom jeziku:

```
var smsMessage = SmsTextualMessage()
      .from("${userResponse.firstName}
            ${userResponse.lastName}")
      .addDestinationsItem(SmsDestination().to(
```

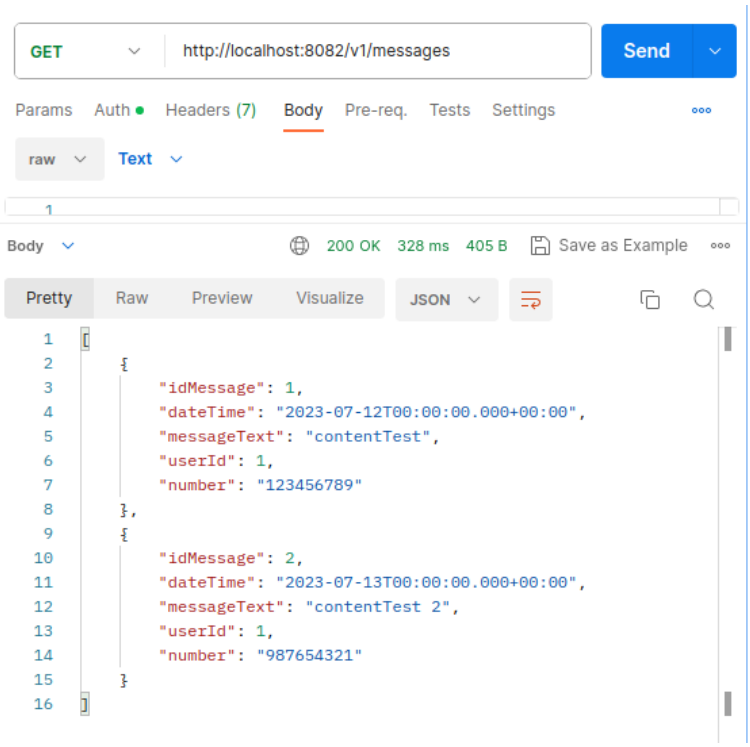
[68] Usp. Chhodde, R. Initializing lazy and lateinit variables in Kotlin, 2022. URL: https://blog.logrocket.com/initializing-lazy-lateinit-variables-kotlin/ (2023-07-12)

```
                smsRequest.toNumber))
        .text(smsRequest.messageText)
```

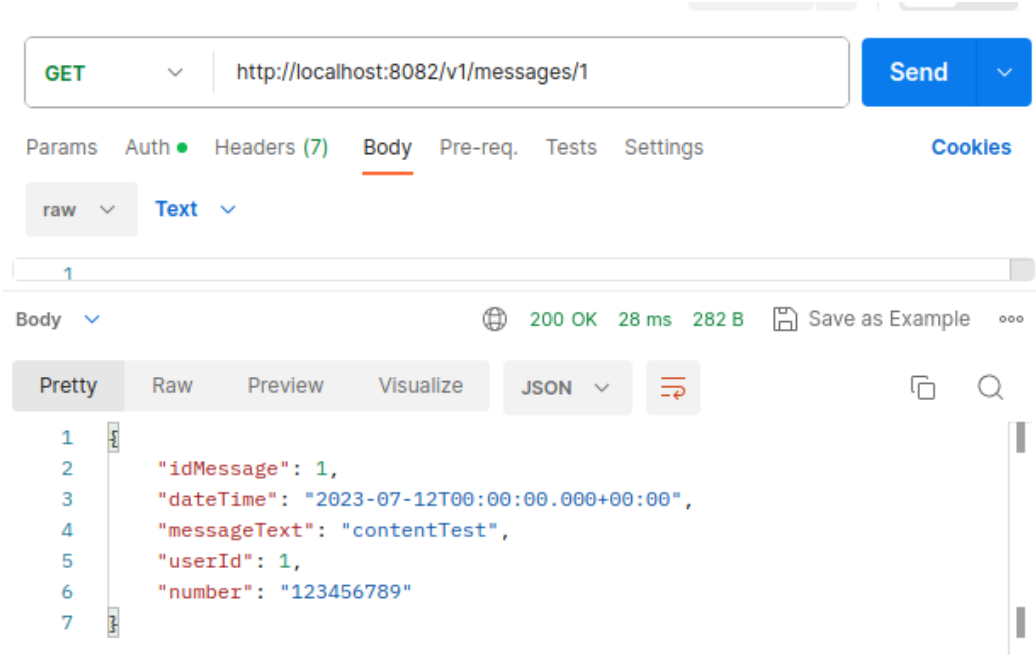
U nastavku je dostupan pregled svih poslanih zahtjeva na temelju oba kontrolera, kontrolera zaduženog za CRUD operacije nad entitetom poruka (Slika 34. – 38.) i kontrolera za slanje poruka (Slika 39.). Mikroservis za slanje poruka, s iznimkom application.properties datoteke (zbog privatnog Infobipovog API Key-a) dostupan je na: https://github.com/marmilosev/sms-api/tree/main/sms-api-kotlin/sendmessageservice.



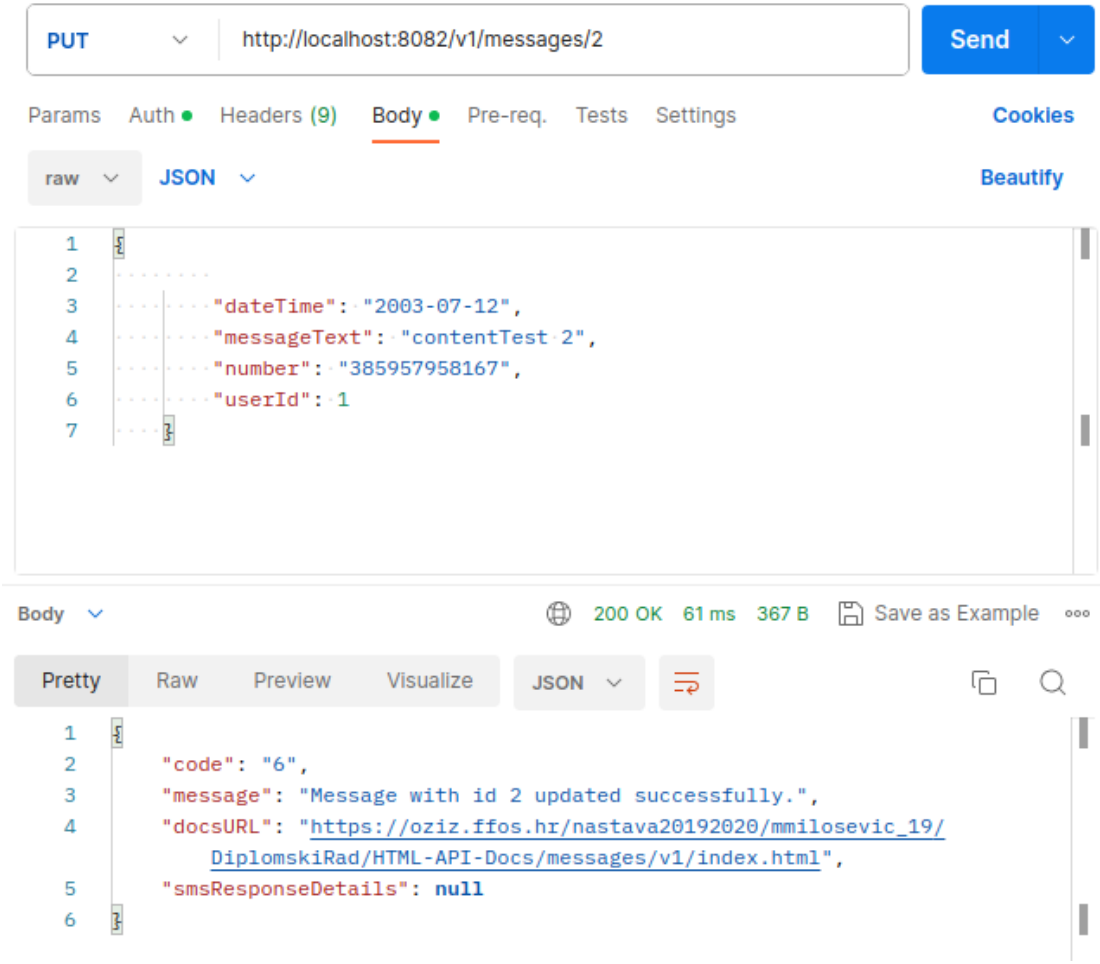
Slika 34. POST zahtjev za kreiranje poruke te ispis odgovora i statusa zahtjeva



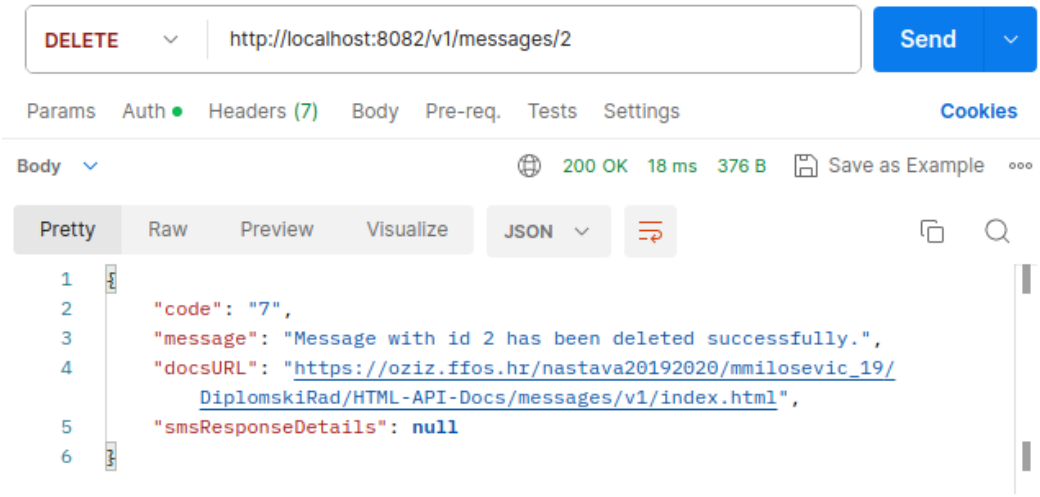
Slika 35. GET zahtjev za ispis svih poruka te ispis odgovora i statusa zahtjeva



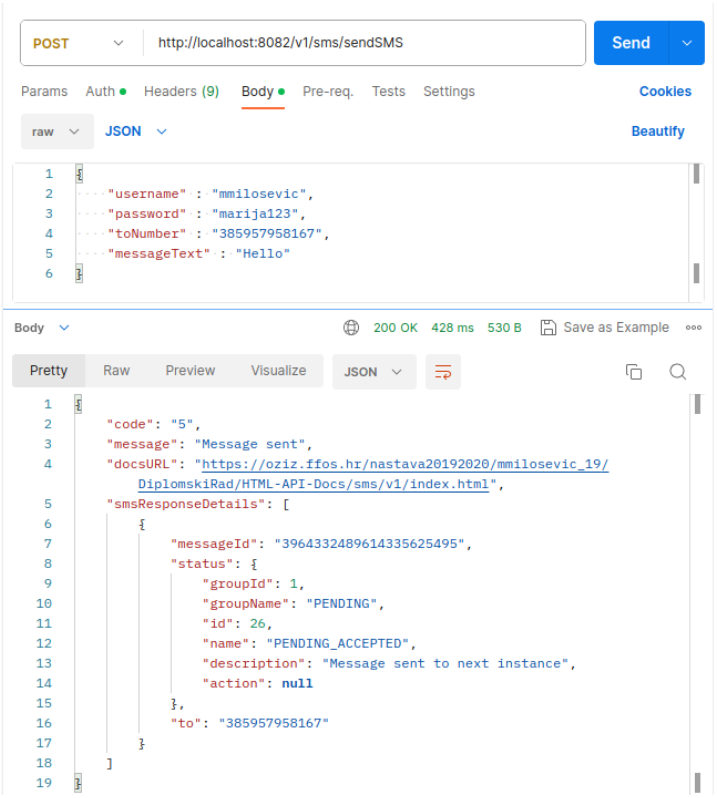
Slika 36. GET zahtjev za ispis poruke s id-jem 1 te ispis odgovora i statusa zahtjeva



Slika 37. PUT zahtjev za mijenjanje datuma slanja i teksta poruke te ispis odgovora i statusa zahtjeva



Slika 38. DELETE zahtjev za brisanjem poruke s jedinstvenim ključem 2 te ispis odgovora i statusa zahtjeva



Slika 39. Uspješno poslana poruka

5.3. Kreiranje mikroservisa za provjeru valjanosti korisničkog imena i lozinke, za usmjeravanje komunikacije i otkrivanje svih kreiranih mikroservisa

Kreiranjem mikroservisa za upravljanje podacima o korisnicima i slanje SMS poruka te uspostavljanjem komunikacije između njih putem zadanog URL-a, nastavlja se s kreiranjem ostalih mikroservisa kako bi se u potpunosti upotpunila mikroservisna arhitektura. Kao i u primjeru kreiranja mikroservisne aplikacije u Java programskom jeziku, i u ovom primjeru nastojat će se kreirati mikroservis koji na temelju valjanog korisničkog imena i lozinke kao rezultat prikazuje token te dva pomoćna mikroservisa kojima je cilj usmjeravanje komunikacije među mikroservisima te otkrivanje samih mikroservisa. Struktura kreiranih mikroservisa ista je kao i u prethodnom primjeru te se može vidjeti na Slici 22. Također je i prikaz servisne arhitekture isti jer se radi o istom radnom okruženju koje neovisno o programskom jeziku strukturu projekata prikazuje na isti način. S obzirom na prethodna poglavlja 5.1. i 5.2. u kojima je detaljnije opisan proces kreiranja *user service* i *send message service* projekata, prvi korak ovog poglavlja je dopuniti zavisnosti navedenih mikroservisnih aplikacija kako bi mogli biti otkriveni nakon ugradnje mikroservisa s tom funkcijom. Navedeno postižemo zavisnošću naziva *spring cloud starter netflix client*, postavljanjem anotacije *@EnableDiscoveryClient* unutar klase s main metodom te definiranjem

konfiguracije na način. Nakon toga slijedi kreiranje mikroservisa *auth service* gdje je potrebno dodati zavisnosti *java jwt* kako bi klasa za upravljanje tokenima mogla generirati i provjeravati valjanost tokena. U navedenom mikroservisu nalazi se sučelje za upravljanje tokenima, koje izgleda ovako:

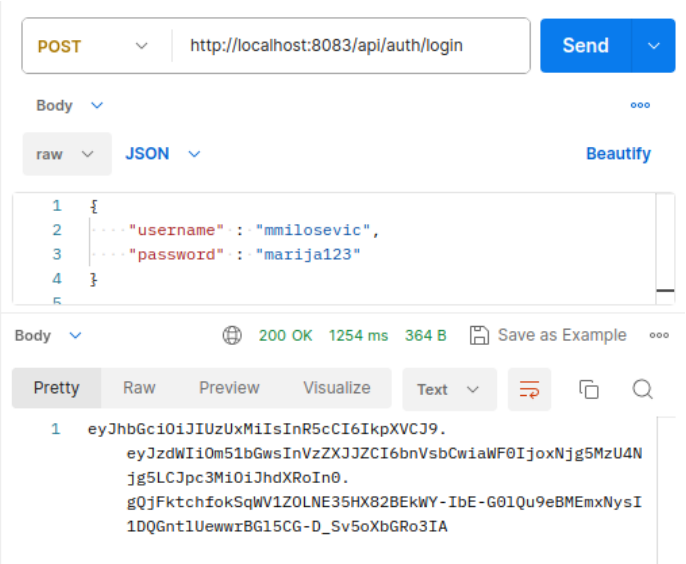
```
public interface TokenManager {
    String generateToken(UserDto userDto);
    void validateToken(String token);
}
```

U implementaciji metoda za generiranje i provjeru valjanosti tokena, kao konstanta upotrijebljena je vrijednost *jwt secret* koja pomaže u generiranju algoritma za navedene metode, a u praksi se koristi kako bi tokeni bili generirani u odnosu na primljene podatke kao što su korisničko ime i lozinka.[69] *Auth service* sastoji se od klasa koje služe za generiranje tokena te njihovu provjeru i od klase koja predstavlja kontroler s definiranim *endpointovima* preko kojih se mogu provjeriti navedene metode te registrirati korisnik. Metoda za generiranje tokena u *AuthenticationController* klasi prikazana je u nastavku:

```
@PostMapping("/login")
public ResponseEntity<String> authenticate(@RequestBody UserDto
      userDto) {
    logger.info("Received validate request.");
    return new ResponseEntity<>( tokenManager.generateToken(
      userDto), HttpStatus.OK);
}
```

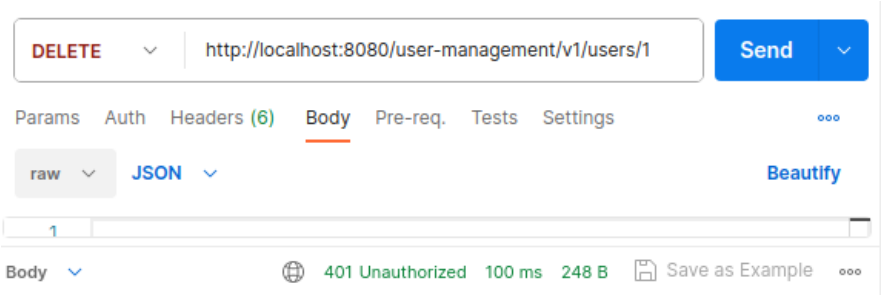
Tako je definirano da se na http://localhost:8083/api/auth/login može izvršiti POST zahtjev upisujući korisničko ime i lozinku, a kao rezultat se dobije JWT token (Slika 40.), dok se na http://localhost:8083/api/auth/register registrira novi korisnik.

[69] Usp. Gutierrez, F. Nav. dj. Str. 199.



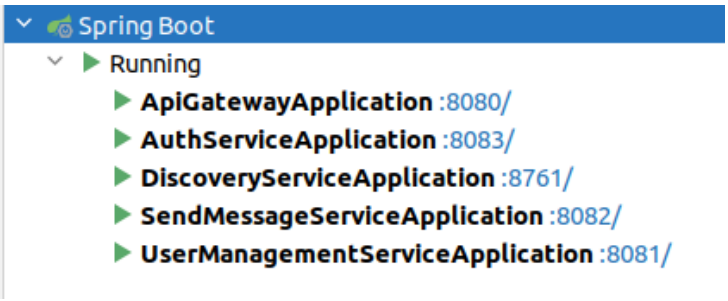
Slika 40. Generirani JWT token na temelju korisničkog imena i lozinke

Na slici je također vidljiva podjela tokena na tri dijela odvojenih točkom što je zadana sintaksa JWT tokena. Osim klasa kojima je cilj rad s tokenima i kontroler klase, koristi se i DTO klasa koja definira svojstva korisnika, odnosno korisničko ime i lozinku. Sljedeći kreirani mikroservis je *discovery servis* zadužen za otkrivanje svih mikroservisa pokrenutih unutar jednog projekta. Navedenom projektu osiguran je rad kroz dodanu zavisnost *spring cloud starter netflix eureka server*, dodavanjem anotacije *@EnableEurekaServer* i definiranjem konfiguracije koja izgleda isto kao i u programskom jeziku Java. U ovom je primjeru, s obzirom da je također kreiran pomoću *Spring Boota*, potrebno paziti na kompatibilnost verzija zavisnosti koje su prethodno opisane u dijelu kreiranja Java aplikacije. Naime, zavisnosti iz *Spring Clouda* nisu u svim verzijama u potpunosti u skladu sa zavisnostima *Spring Boota*. Zadnji korak je osposobljavanje *api gateway* servisa preko kojega je osigurano slanje svih zahtjeva. Prvotno dodajemo sve iste zavisnosti kao u *user service* i *send message service* te jednaku anotaciju unutar klase s *main* metodom. Ono po čemu se razlikuje je konfiguracijska datoteka u kojoj su navedene putanje koje pružaju rezultate bez potrebe za prvotnim slanjem tokena, ali i sadrži klase sa cjelokupnom logikom ograničavanja pojedinih servisa, odnosno *endpointova* unutar servisa za dopuštanje njihovog rada jedino uz slanje JWT tokena dobivenog prijavom pomoću korisničkog imena i lozinke. Kao i u prethodnom primjeru, slanjem zahtjeva bez odgovarajućeg tokena rezultat je 401 *Unauthorized* status (Slika 41.).

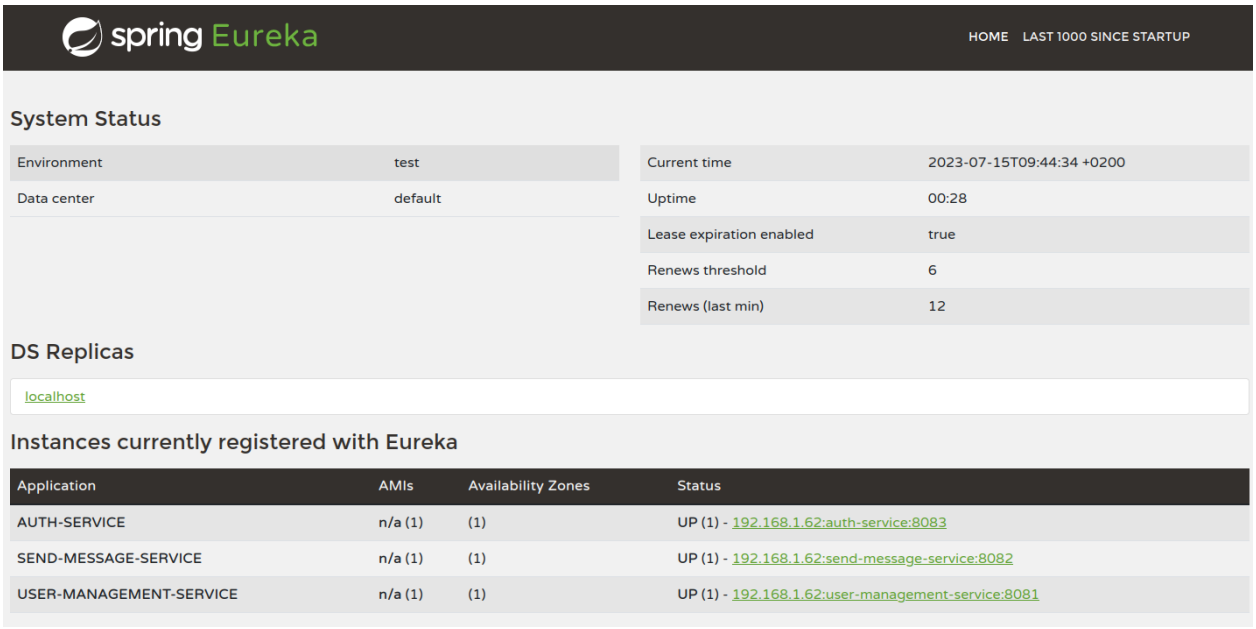


Slika 41. Pokušaj brisanja korisnika bez JWT tokena

S obzirom da je rad cjelokupne mikroservisne arhitekture postavljen na iste portove, prikazi rada *endpointova* i poslanih zahtjeva jednaki su kao na Slici 24. i Slici 25. u prethodnom poglavlju. Tako je i u ovom primjeru omogućena komunikacija preko *api gateway* servisa preko kojega svi prethodno prikazani zahtjevi (GET, POST, PUT, DELETE korisnika i poruka te slanje SMS poruke) rade uz promjene *endpointa*. Također struktura svih pokrenutih mikroservisa vidljiva je u nastavku na Slici 42., dok Slika 43. prikazuje Eureka mrežnu stranicu koja prikazuje sve pronađene mikroservise, a ta je stranica dostupna upisujući u adresnu traku internetskog preglednika localhost:8761. Kompletan kod dostupan je na: https://github.com/marmilosev/sms-api/tree/main/sms-api-kotlin.



Slika 42. Pokrenuti mikroservisi



Slika 43. Prikaz stranice u internet pregledniku kada je Eureka server pokrenut i pronalazi sve mikroservise

6. Usporedba Java i Kotlin mikroservisne arhitekture kreirane pomoću Spring Boot frameworka

U prethodnim poglavljima opisani su koraci kreiranja mikroservisne ahitekture u Java i Kotlin programskim jezicima temeljeni na Spring Boot frameworku s ciljem osiguravanja razlika i sličnosti između programskih jezika u kojima su kreirani. Upravo iz tog razloga nastojalo se pridržavati određenih istih tehničkih specifikacija, nazivlja i sličnih karakteristika kako bi se orijentiralo na usporedive sličnosti i različitosti samog programskog koda, a time i programskih jezika korištenih u opisanim primjerima. Definirajući Java i Kotlin programski jezik uočena je razlika u vremenu nastanka programskih jezika, odnosno Java je kreirana u drugoj polovici 20. stoljeća, dok je Kotlin kreiran 2010. godine što upotpunjuje činjenicu kako Java programski jezik ima veću zajednicu korisnika. Prva uočena razlika prilikom kreiranja projekta u Java i Kotlin programskim jezicima je mogućnost odabira Kotlin klase i Kotlin datoteke što može biti zbunjujuće za korisnika. Kao što je već objašnjeno u prethodnom poglavlju, razlika je u broju klasa koja se može nalaziti unutar jedne datoteke, odnosno klase, a također postoji i razlika u tome što prilikom kreiranja datoteke razvojno okruženje ne kreira automatski i klasu, dok kod kreiranja klase razvojno okruženje će

automatski kreirati klasu.[70] Iako u Java programskom jeziku također postoji mogućnosti više klasa u jednoj datoteci, razlika je u tome što Java posjeduje klasu kao tip datoteke, dok Kotlin razlikuje klasu i datoteku. Već prilikom kreiranja prve klase vidljiva je razlika u sintaksi definiranja svojstava. Tako u Java programskom svojstvo za mail izgleda:

```
private String mail;
```

dok u Kotlin programskom jeziku isto to svojstvo izgleda:

```
var mail: String
```

Oba svojstva se sastoje od nekih istih elemenata, a to su naziv svojstva, tip podataka te se svojstvu u Kotlinu može dodati private pristup ukoliko želimo da bude dostupan samo unutar klase. No, ono po čemu se razlikuju je ključna riječ *var* koja označava varijablu kojoj se vrijednost može promijeniti. Osim *var*, postoji i *val* koja označava varijablu čija se vrijednost ne može promijeniti.[71] Razlika Java i Kotlin programskog jezika vidljiva je u kreiranju primarnog konstruktora u Kotlinu koji je objašnjen u prethodnom primjeru te također već spomenutim podatkovnim klasama, odnosno engl. *data class*. Iako Kotlin primarni konstruktor koristi i kao deklariranje svojstava čime skraćuje dužinu koda, Spring Boot Lombok zavisnost osigurava anotaciju *@NoArgsConstructor* i *@AllArgsConstructor* koja može skratiti dužinu napisanog Java koda. Također u kreiranim klasama vidljivo je kako nedostaju anotacije *@Getter* i *@Setter* iz *lombok* zavisnosti jer Kotlin programski jezik automatski kreira *get* i *set* metode prilikom definiranja svojstava.[72] Kotlin u odnosu na Javu također omogućuje zaštitu od takozvanih *null* vrijednosti (engl. *null-safety*), odnosno za svako svojstvo definira se mogućnost dodjeljivanja *null* vrijednosti.[73] To se postiže dodjeljivanjem stroge nulte sigurnosti koja se označava znakom upitnika pored tipa podataka i ona omogućuje dodjelu *null* vrijednosti kao vrijednosti tipa. Primjer takvog korištenja je

```
var message: String? = null
```

Null vrijednost se također može postaviti kao zadana vrijednost, odnosno svako svojstvo inicijalizirati. To olakšava izvođenje koda za razliku u Java programskom jeziku gdje je česta pojava iznimke s *null* vrijednosti (engl. *NullPointerException*).[74] Otvaranjem pom.xml datoteke pronalazi se sljedeća razlika Java i Kotlin programskih jezika. Osim dodanih

[70] Usp. Kotlin File vs Class. What's the difference?, 2017. URL: https://stackoverflow.com/questions/42769873/kotlin-file-vs-class-whats-the-difference (2023-07-15)
[71] Usp. Kotlin Language Documentation 1.9.0. Str. 76. URL: https://kotlinlang.org/docs/kotlin-reference.pdf (2023-07-15)
[72] Usp. Roy, A. S.; Karanpuria R. Nav. dj., str. 106.
[73] Usp. Kotlin Language Documentation 1.9.0. Nav. dj., str. 591.
[74] Usp. Arnold, K.; Gosling, J.; Holmes, D. Primitives as Types. // The Java Programming Language. 4. izd. Boston: Addison Wesley Professional, 2005. Str. 180.

zavisnosti, postoje zadane zavisnosti koje se većinom odnose na Spring Boot framework, no razlika je što Kotlin za svoj rad zahtjeva nekoliko zavisnosti. To su zavisnost *kotlin-reflect*, *kotlin-stdlib* i *jackson-module-kotlin*. Razlike između navedena dva programska jezika uočljive su kroz sintaksu. Jedan od primjera je kroz definiranje metoda koje u Java programskom jeziku izgledaju:

```
List<User> getAllUsers();
```

dok u Kotlin programskom jeziku ima sintaksu:

```
fun getAllUsers(): List<User>
```

Razlika je vidljiva u tome što se u Kotlin programskom jeziku koristi ključna riječ fun koja označava englesku riječ za funkciju, dok ostatak metode, odnosno funkcije predstavlja naziv metode i tip podatka vrijednosti koja se prikazuje kao rezultat, dok u Java programskom jeziku ne postoji ključna riječ za definiranje metode, nego se sastoji od tipa podataka i naziva metode. Druga vidljiva sintaksna razlika je u korištenju ključne riječi *extends* koja označava princip nasljeđivanja u objektno-orijentiranom programiranju. U sklopu ovoga rada navedeni princip iskorišten je za potrebe kreiranja repozitorija te u Java programskom jeziku izgleda:

```
public interface UserRepository extends JpaRepository<User,
Long>
```

dok u Kotlinu isti način izgleda:

```
interface UserRepository: JpaRepository<User, Long>
```

Razlika je u tome što u Java programskom jeziku postoji ključna riječ *extends*, dok je Kotlin zamijenio tu riječ sa znakom dvotočja. Također se u navedenom primjeru može primijetiti nedostatak riječi *public* koja označava opseg dostupnosti klasa, metoda ili svojstava. U Java programskom jeziku zadana dostupnost označena je rječju *default* te se ona ne mora postavljati uz naziv klase, metode ili svojstva, a predstavlja dostupnost iz paketa. Odnosno klasa, metoda ili svojstvo *default* opsega dostupna je s bilo kojeg mjesta unutar paketa. U Kotlin programskom jeziku ne stavljajući ključnu riječ za opseg dostupnosti, klasa, metoda ili svojstvo je automatski postavljena na vrijednost *public*. Ostali definirani modifikatori u Kotlin programskom jeziku su *private*, *protected* i *internal*, dok u Java programskom jeziku, osim *default* i *public*, također postoje *private* i *protected*. Sljedeća razlika može se primijetiti kod engl. *castinga*, specifično u primjeru:

```
var userDto = userDto as? UserDto
```

Navedeni primjer odnosi se na takozvani engl. *safe cast* koji se prepoznaje po ključnoj riječi *as* nakon koje slijedi znak upitnika. Ovim se načinom userDto-u *casta*, odnosno dodjeljuje, tip UserDto. Također u Kotlinu postoji i engl. *smart cast* koji se od *safe casta* razlikuje po ključnoj riječi *is*. Sljedeći primjer prikazuje kod u Java i Kotlin programskom jeziku:

```
List<SmsTextualMessage> messages = new ArrayList<>();
var messages: MutableList<SmsTextualMessage> = ArrayList<>();
```

Prikazani dio koda odnosi se na kreiranje liste u Java i Kotlin programskom jeziku te prikazuje razlike u definiranju promjenjivih i nepromjenjivih listi. Naime, u Java programskom jeziku sučelje *List* predstavlja općenitu listu koja svojom implementacijom *ArrayList* osigurava promjenjivu listu. Ukoliko se u Java programskom jeziku želi koristiti nepromjenjiva lista tada se ona mora implementirati kreiranjem instance *Collection* sučelja ili korištenjem vanjske biblioteke.[75] Za razliku od Jave, Kotlin osigurava jednostavnije korištenje promjenjivih lista koje se definiraju rječju *MutableList*, dok se nepromjenjive u slučaju Kotlina definiraju s *List*.[76] Neke od prethodno opisanih razlika mogu se također uočiti u definiranju *main* metode. Tako *main* metoda u Java programskom jeziku izgleda:

```
public static void main(String[] args) {
        SpringApplication.run
        (AuthenticationServiceApplication.class, args);
}
```

dok u Kotlin programskom jeziku izgleda:

```
fun main(args: Array<String>) {
        runApplication<AuthServiceApplication>(*args);
}
```

Prva već spomenuta razlika je u definiranju metode gdje se u Java programskom jeziku navodi tip koji metoda vraća ili *void* ukoliko ne vraća vrijednost te *static* riječ ukoliko metoda pripada klasi, a ne instanci klase. U Kotlin programskom jeziku, također već spomenuto, modifikator pristupa *public* je zadani modifikator i metode su uvijek označene riječi *fun*, a za definiranje *main* metode iskorištene su i prethodno spomenute liste. Za razliku od Kotlin programskog jezika, u Javi kao parametar je definiran niz tipa *String*. Navedene metode služe pokretanju *Spring Boot* aplikacije te obje koriste funkciju *run* koja je također u različitim sintaksama napisana, a također su i klase drugačijeg nazivlja prateći konvencije nazivlja oba programska jezika.

Još neke razlike Java i Kotlin programskog jezika koje nisu implementirane kroz projekt, a važne su za napomenuti kreirane su u klasi s *main* metodom unutar servisa *user service*. Prva takva razlika je mogućnost funkcijskog programiranja. Java je tek u novijim

[75]Usp. Immutable List in Java, 2018. URL: https://www.geeksforgeeks.org/immutable-list-in-java/ (2023-08-20)
[76]Usp. Use Lists in Kotlin. URL: https://developer.android.com/codelabs/basic-android-kotlin-training-lists#1 (2023-08-20)

verzijama implementirala mogućnost korištenja dijelova funkcijskog programiranja, dok je Kotlin u potpunosti kreiran kako bi podržao koncepte funkcijskog programiranja. Tako se u Kotlinu može koristiti lambda funkcija koja pripada funkcijskim literalima što podrazumijeva da se takve funkcije ne deklariraju, nego se prenose kao izraz.[77] Primjer toga je kreiranje liste brojeva s time da se svaki broj množi s brojem dva te ispisuje. U Java programskom jeziku za tu je potrebu kreirana dodatna metoda koja se iterira kroz svaki element liste i tako osigurava izvođenje funkcije. Za razliku od Java programskog jezika, unutar Kotlina se ne mora kreirati nova metoda već je samo potrebno koristiti ekstenziju funkcije *map* koja množi svaki element list s brojem dva pomoću lambda izraza: { it * 2 }. Kao i svaki programski jezik, oba ovdje navedena programska jezika osiguravaju provjeru grananja. Java programski jezik osigurava korištenje *if* i *switch* opcija, dok Kotlin koristi *if* za jednostruko grananje te mu je sintaksa jednaka kao i u Java programskom jeziku, dok *when* koristi za višestruko grananje.[78] Razlika također postoji u tome što Java programski jezik poznaje ternarni operator, dok unutar Kotlina on ne postoji, već se uvijek koristi *if-else* opcija. Također u Kotlinu unutar *when* operatora za razdvajanje uvjeta koristi se znak -> (strelica). Kao najčešće izdvojena prednost Kotlin programskog jezika u odnosu na Javu je to što je Kotlin od početka kreiran za pisanje jednostavnog koda te može razumjeti kod, samostalno zaključiti tip deklaracije, kao i *gettere* i druge slične komponente koje su generirane kompajlerom.[79] Općenito, smatra se da je Java programski jezik opširniji od Kotlin programskog jezika zbog čega je Kotlin stekao popularnost korištenja. Još neke prednosti Kotlina u odnosu na javu dostupne su na stranici Kotlin dokumentacije, a među njima izdvajaju se već spomenuti engl. *smart cast*, posebna sučelja za zbirke ovisno jesu li one samo za čitanje (engl. *read-only*) ili promjenjive i slično.[80] Sažeti prikaz svih navedenih usporedbi vidljiv je u Tablici 1.

[77] Usp. Hagos, T. Nav. dj., str. 109.
[78] Usp. Conditions and loops, 2023. URL: https://kotlinlang.org/docs/control-flow.html (2023-07-17)
[79] Usp. Terro, R. … [et al.]. Are you still smelling it?: A comparative study between Java and Kotlin language. // SBCARS (2018)., str. 2. URL: https://www.researchgate.net/publication/327568948_Are_you_still_smelling_it_A_comparative_study_between_Java_and_Kotlin_language (2023-07-17)
[80] Usp. Comparison to Java, 2022. URL: https://kotlinlang.org/docs/comparison-to-java.html#what-kotlin-has-that-java-does-not (2023-07-17)

Tablica 1. Usporedba Java i Kotlin programskog jezika

	JAVA	**KOTLIN**
GODINA IZDAVANJA	1995	2010
OPCIJE STVARANJA DATOTEKA	Java klasa	Koltin klasa, Kotlin datoteka
VARIJABLE	final (nepromjenjiva) var (deklaracija lokalne varijable)	val (nepromjenjiva) var (promjenjiva)
PRIMARNI KONSTRUKTOR	Postoji samo konstruktor koji se definira ključnom riječi *constructor* Ima naziv kao i klasa unutar koje se kreira	Deklarira se unutar zaglavlja klase (engl. *class header*)
DATA KLASA	Ne postoji ekvivalent	Ključna riječ data, automatski generira gettere, settere, equals(), hashCode() i toString()
ZADANE ZAVISNOSTI	Ne postoji ekvivalent	kotlin-reflect kotlin-stdlib jackson-module-kotlin
GETTERI I SETTERI	Generiraju se ručno	Generiraju se automatski
NULL/NULLABLE	NullPointerException iznimka koja se javlja na null objektu	Null-safety sprječava NullPointerException
MODIFIKATORI PRISTUPA (ENGL. *ACCESS MODIFIERS*)	private protected default public	private protected internal public
DEKLARACIJE METODA	[access modifier] [static] [void/tip podatka] naziv_metode (parametri)	[access modifier] **fun** naziv_metode (parametri)
NASLJEĐIVANJE	extends	Dvotočje (:)
CASTING	cast	smart cast, safe cast
PROMJENJIVA LISTA	List<T>, ArrayList<>()	MutableList<T>

NEPROMJENJIVA LISTA	Collection ili biblioteka	List<T>
JEDNOSTRUKO VIŠESTRUKO GRANANJE	if switch	if when
SMART CAST	Ne postoji ekvivalent	omogućava pristup metodi i svojstvu bez provjere null vrijednosti ili castinga
TERNARNI OPERATOR	uvjet ? [izraz ako je istina] : [izraz ako nije istina]	Korištenje if-else

7. Zaključak

Razvojem novih programskih jezika i novih tehnologija nastoji se odgovoriti na postojeće probleme. Kako bi se pratio razvoj informacijskih tehnologija neizbježno je pratiti trendove na tržištu istog područja. Jedan od trendova zasigurno je korištenje razvojnih okvira ili *frameworka* koji osiguravaju brzu i kvalitetnu izradu aplikacija. Iako je za potrebe ovoga rada prvotno bilo potrebno naučiti razvojni okvir, vrijeme kreiranja aplikacije ubrzalo je postojanje repozitorija, metoda i klasa koje je potrebno naslijediti kako bi se uspješno iskoristili. Također, koristeći mikroservisnu arhitekturu koja je zamijenila dugogodišnje korištenu monolitnu arhitekturu aplikacija zaključene su prednosti takvog načina kreiranja. Naime, brže je otklanjanje pogrešaka jer se prilikom pokretanja mikroservisa točno može utvrditi unutar kojeg mikroservisa se dogodila greška čime se skraćuje vrijeme identificiranja dijela koda koji uzrokuje grešku. Međutim, nedostatak je to što je komunikacija mikroservisa relativno novo područje koje se još razvija te postoje različiti načini osiguravanja komunikacije, čemu dodatno doprinosi informacija kako različite tvrtke koriste različite servere za uspostavljanje mikroservisa. Usporedbom Java i Kotlin programskog jezika, utvrđeno je kako se navedeni programski jezici razlikuju u deklariranju varijabli, konstruktora te *get* i *set* metoda. Također razlikuju se u modifikatorima pristupa gdje oba programska jezika sadrže mogućnost definiranja *private*, *protected* i *public*, dok se razlikuju u modifikatoru koji se u Javi naziva *default*, a u Kotlinu naziva *internal*. Ostale razlike utvrđene kreiranjem aplikacija su razlika u deklaraciji metode u kojima je razlika jedino to što Kotlin koristi ključnu riječ *fun*, kod korištenja principa nasljeđivanja Java koristi ključnu riječ *extends*, dok Kotlin znak dvotočja te postoje razlike u definiranju lista ovisno o tome jesu li one promjenjive ili nepromjenjive. Ono po čemu se Kotlin izdvaja u odnosu na Javu je postojanje dodatnih zavisnosti koje se automatski postavljaju u projekt kako bi se aplikacija uspješno pokrenula kao Kotlin aplikacija, više tipova *castinga* od Jave, *data* klasa koja sadrži *gettere*, *settere* i pojedine metode te *null-safety* koja sprječava pojavu iznimki u radu s *null* vrijednosti. Bez obzira na navedene razlike, prednost je to što su dva navedena programska jezika kompatibilna te se razvojni okviri kreirani za jedan programski jezik mogu nesmetano koristiti u drugom. Neovisno o prethodno izdvojenim različitostima, Java i Kotlin programski jezici vrlo su slični po tome što primjerice oba pripadaju objektno-orijentiranim jezicima te su kompatibilni za implementiranje principa objektno-orijentiranih jezika. Također, oba su jezika kompajlirana koriteći engl. *bytecode* te ono što je vidljivo u strukturi kreiranih projekata je da je u oba programska jezika kod raspoređen u pakete i klase, a minimalne razlike su prepoznate u

sintaksi pisanja koda, konkretno u deklariranju i inicijaliziranju vrijednosti svojstava te metoda. Iako razvoj novih tehnologija i alata omogućava brojne prednosti, nedostatak čini današnja tendencija smanjenja programskog koda kako se on ne bi duplirao te se pokušava stvoriti što kraći programski kod koji zahtjeva neprestano praćenje novih tehnologija potrebne za njegovo razumijevanje. Ono što se može zaključiti korištenjem mikroservisne arhitekture je izbjegavanje opterećenja cijele aplikacije, već usmjeravanje na više manjih dijelova čime se omogućava smanjenje opterećenja rada aplikacije, kao i rad svakog dijela aplikacije zasebno. S obzirom da su mikroservisne aplikacije zasebni dijelovi aplikacije, kao takvi mogu se zasebno kreirati neovisno jedno o drugome čime se osigurava neometani rad više ljudi u timu, kao i, iako u ovome radu to nije slučaj, kreiranje korištenjem više tehnologija. Primarna prednost u ovome radu, kao što je već navedeno na početku ovoga poglavlja, je zasigurno lakše pronalaženje greške, posebice u dijelovima povezivanja mikroservisa u kojemu se točno može odrediti koji mikroservis nije pravilno spojen. Što se tiče korištenja razvojnih okvira ubrzano je kreiranje aplikacije zbog korištenja brojnih dodanih zavisnosti i predefiniranih anotacija koje vidno skraćuju kod i brže obavljaju određene operacije. Zaključno, gledano iz kuta mikroservisne arhitekture baziranoj na Spring Boot razvojnoj okolini nema razlike u implementiranju principa, već je razlika u programskim jezicima. Iako oba programska jezika imaju sličnu sintaksu pisanja koda u kontekstu pisanja petlji, operatora, komentara, primitivnih tipova podataka, *try-catch* blokova koda te oba pripadaju objektno-orijentiranim jezicima, ono što Kotlin čini popularnim programskim jezikom je čitljivija sintaksa s kraćim kodom u odnosu na Javine ponavljajuće linije koda, vrlo slična sintaksa s Javom, poboljšane funkcionalnosti kao što su integrirane *get* i *set* metode te interoperabilnost s Javom koja olakšava prelazak s jednog na drugi programski jezik.

8. Literatura

Antonov, A. Spring Boot Cookbook. Birmingham: Packt Publishing, 2015.

Arnold, K.; Gosling, J.; Holmes, D. Primitives as Types. // The Java Programming Language. 4. izd. Boston: Addison Wesley Professional, 2005. Str. 166 - 181.

Atlasik, K. How to communicate Java microservices?, 2021. URL: https://softwaremill.com/how-to-communicate-java-microservices/ (2023-07-04)

Behler, M. Java Microservices: A Practical Guide, 2020. URL: https://www.marcobehler.com/guides/java-microservices-a-practical-guide (2023-07-04)

Building an Application with Spring Boot, 2023. URL: https://spring.io/guides/gs/spring-boot/ (2023-07-04)

Chhodde, R. Initializing lazy and lateinit variables in Kotlin, 2022. URL: https://blog.logrocket.com/initializing-lazy-lateinit-variables-kotlin/ (2023-07-12)

Comparison to Java, 2022. URL: https://kotlinlang.org/docs/comparison-to-java.html#what-kotlin-has-that-java-does-not (2023-07-17)

Conditions and loops, 2023. URL: https://kotlinlang.org/docs/control-flow.html (2023-07-17)

Crusoveanu, L. Intro to Inversion of Control and Dependency Injection with Spring, 2023. URL: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring (2023-07-03)

Difference between @Controller and @RestController in Spring Boot and Spring MVC?, 2020. URL: https://medium.com/javarevisited/difference-between-controller-and-restcontroller-in-spring-boot-and-spring-mvc-216578ad445f (2023-07-05)

Fadatare, R. Spring Boot DTO Example - Entity To DTO Conversion, 2021. URL: https://www.javaguides.net/2021/02/spring-boot-dto-example-entity-to-dto.html (2023-07-06)

Flanagan, D. Java in a Nutshell. 2. izd. Cambridge… [et al.]: O' Reilly, 1997.

Foote, K. D. A Brief History of Microservices, 2021. URL: https://www.dataversity.net/a-brief-history-of-microservices/ (2023-07-04)

Fowler, M. Richardson Maturity Model, 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (2023-07-17)

GitHub Student Developer Pack, 2023. URL: https://education.github.com/pack (2023-07-05)

Gradle Vs Maven: What's The Difference?, 2023. URL: https://www.interviewbit.com/blog/gradle-vs-maven/ (2023-07-04)

Gupta, L. HTTP Methods, 2021. URL: https://restfulapi.net/http-methods/ (2023-07-05)

Gutierrez, F. Pro Spring Boot. New Mexico: Apress, 2016.

Hagos, T. Learn Android Studio 3 with Kotlin: Efficient Android App Development. Manila: Apress, 2018.

Immutable List in Java, 2018. URL: https://www.geeksforgeeks.org/immutable-list-in-java/ (2023-08-20)

Introduction to Creational design Patterns, 2022. URL: https://www.baeldung.com/creational-design-patterns#builder (2023-07-06)

Java (programming language). URL: https://hmong.ru/wiki/Java_language (2023-07-05)

Kappagantula, S. Microservices Tutorial: Learn all about Microservices with Example, 2023. URL: https://www.edureka.co/blog/microservices-tutorial-with-example (2023-07-04)

Kotlin File vs Class. What's the difference?, 2017. URL: https://stackoverflow.com/questions/42769873/kotlin-file-vs-class-whats-the-difference (2023-07-15)

Kotlin Language Documentation 1.9.0. Str. 76. URL: https://kotlinlang.org/docs/kotlin-reference.pdf (2023-07-15)

Macero, M. Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, RIbbon, Zuul and Cucumber. New York: Apress, 2017.

Mitchell, B. What is a Framework and Why It's Useful, 2023. URL: https://www.codingdojo.com/blog/what-is-a-framework (2023-06-28)

Mool, S. Json Web Token (JWT), 2019. URL: https://medium.com/@mool.smreeti/json-web-token-jwt-2ba5d032685e (2023-07-07)

Moskala, M.; Wojda, I. Android Development with Kotlin. Birmingham: Packt Publishing, 2017.

Nicoll, S.; Syer, D. Spring Initializr Reference Guide. URL: https://docs.spring.io/initializr/docs/0.4.x/reference/htmlsingle/ (2023-07-05)

Nuwanthilaka, I. Docker: Zero to Hero (with SpringBoot + Postgres), 2018. URL: https://isurunuwanthilaka.medium.com/docker-zero-to-hero-with-springboot-postgres-e0b8c3a4dccb (2023-07-05)

Programming Trends Worth Watching in 2023, 2023. URL: https://blog.gitnux.com/programming-trends/ (2023-07-17)

Rajesh, R. V. Spring Microservices: Build scalable microservices with Spring, Docker and Mesos. Birmingham: Packt Publishing, 2016.

Roy, A. S.; Karanpuria R. Kotlin Programming Cookbook. Birmingham: Packt Publiching, 2018. Str. 23.

Siva Prasad Reddy, K. Beginning Spring Boot 2: Applications and Microservices with the Spring Framework. Hyderabad: Apress, 2017.

Spring 5 WebClient, 2022. URL: https://www.baeldung.com/spring-5-webclient (2023-07-04)

Spring Cloud, 2022. URL: https://spring.io/projects/spring-cloud (2023-07-07)

Spring Framework Documenation: Language Support,2020. URL: https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/languages.html (2023-07-04)

Spring vs Spring Boot: An in-depth Comparison, 2023. URL: https://www.turing.com/kb/spring-vs-spring-boots-best-web-apps (2023-06-28)

Terro, R. … [et al.]. Are you still smelling it?: A comparative study between Java and Kotlin language. // SBCARS (2018)., str. 2. URL: https://www.researchgate.net/publication/327568948_Are_you_still_smelling_it_A_comparative_study_between_Java_and_Kotlin_language (2023-07-17)

Top Web Development Frameworks (Frontend & Backend), 2022. URL: https://www.emizentech.com/blog/web-development-frameworks.html (2023-06-28)

Ugarte, A. Difference Between @NotNull, @NotEmpty, and @NotBlank Constraints in Bean Validation, 2023. URL: https://www.baeldung.com/java-bean-validation-not-null-empty-blank (2023-07-06)

Use Lists in Kotlin. URL: https://developer.android.com/codelabs/basic-android-kotlin-training-lists#1 (2023-08-20)

Walls, Craig. Spring in Action. New York: Manning Publications, 2022.

Webb, P. … [et al.]. Spring Boot Reference Documentation. Str. 35. URL: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/pdf/spring-boot-reference.pdf (2023-07-06)

What are Microservices? Code Examples, Best Practices, Tutorials and More, 2019. URL: https://stackify.com/what-are-microservices/ (2023-07-04)

What exactly is Kotlin?, 2022. URL: https://codeop.tech/what-exactly-is-kotlin/ (2023-07-09)