JAVA WEB APLIKACIJA TEMELJENA NA KONTEJNERIZACIJI I MIKROSERVISIMA

Barun, Božo

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

JAVA WEB APLIKACIJA TEMELJENA NA KONTEJNERIZACIJI I MIKROSERVISIMA

Božo Barun

Zagreb, veljača 2020.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 18.02.2020.

Božo Barun

Predgovor

Zahvaljujem svom mentoru prof. Aleksanderu Radovanu na ukazanom povjerenju pri prihvaćanju mentorstva i pruženoj pomoći prilikom izrade ovog završnog rada.

Zahvaljujem svojoj supruzi i obitelji na potpori i povjerenju tijekom cjelokupnog studija. Nadam se da ću boljim služenjem isto i opravdati.

Serviam!

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Ciljevi ovog rada su teoretski opisati tehnologiju „cloud-native“ aplikacija i prikazati postupak izrade prototipa jedne takve aplikacije koja služi podršci poslovanja generičke knjižnice.

Motivacija za odabir teme potječe iz činjenice da su tehnologija oblaka i „cloud-native“ aplikacije nastale nedavno, dinamično su područje računarstva koje se izrazito brzo i snažno razvija te osjećaj da je ovo prijelomni trenutak u području računalnog inženjerstva usporediv s pojavom objektno orijentiranog programiranja ili nastanka Weba.

Praktični dio ovoga rada sastoji se od web aplikacije dizajnirane po principu mikroservisne arhitekture, zapakirane u kontejner i postavljene na nekoliko platformi koje nude uslugu dinamičke orkestracije Docker formata kontejnera. Pri izradi su korištene tehnologije koje predstavljaju standard u izradi mikroservisnih aplikacija u poslovnoj domeni, otvorenog su kôda i dostupne svima. To su Java, Spring, Angular i Docker. Funkcionalnosti aplikacije su minimalni skup aktivnosti poslovne domene upravljanja knjižnicom primjenjive kao referenca na mnogo sličnih područja, ali i neke specifičnosti dodane kako bi bile pokazne za mogućnosti tehnologije ili ilustrativne za ideju aplikacija namijenjenih oblaku.

Ključne riječi: cloud-native, oblak, mikroservis, kontejner, Java, Angular, Docker.

Summary

Main objectives of this final thesis are to describe the cloud-native application technology and to demonstrate the procedure of building a prototype of one such application which has the capabilities that serve the purpose of supporting a generic library.

The motivation to choose this topic stems from the fact that both the cloud itself and cloud-native applications have come to existence recently, they are a very dynamic field of computing which is evolving rapidly and a gut feeling that this truly is a fundamental moment in software engineering history, relatable to moments in which object-oriented programming or appeared, or the web itself.

Practical part of this thesis consists of a web application designed with microservice architecture in mind, packaged in a container and deployed to a few cloud hosting services which provide the ability to host Docker applications. While the application was built, only open-source and gold standard technologies from the cloud-native domain were used. Those are: Java, Spring framework, Angular and Docker. The functionalities of this application consist of a minimal set of business domain activities needed to run a library, which are relatable to other similar business fields. As well as some intricacies added to be exemplary of the technology capabilities or illustrative of ideas that consist applications that are meant to be built for the cloud.

Key words: cloud-native, the cloud, microservice, container, Java, Angular, Docker.

Sadržaj

1. Uvod 1
2. Cloud-native paradigma izrade aplikacija 2
2.1. Mikroservisna arhitektura 2
2.1.1. Karakteristike mikroservisne arhitekture 3
2.2. Kontejnerizacija 11
2.2.1. Prednosti korištenja kontejnera 11
2.2.2. Način korištenja kontejnera 12
2.3. Dinamička orkestracija 13
3. Tehnologije izrade aplikacije 15
3.1. Java 15
3.2. Spring programski okvir i Spring Boot 15
3.3. H2 Baza podataka 16
3.4. Liquibase 17
3.5. Hibernate 18
3.6. Docker 18
3.7. Gradle 19
3.8. Angular 20
4. Arhitektura Java web aplikacije 22
5. Funkcionalnosti aplikacije 24
5.1. Sigurnosne funkcionalnosti 25
5.2. Korisničke funkcionalnosti 28
Zaključak 32
Popis kratica 33

Popis slika..... 34

Popis kôdova 35

Literatura 36

1. Uvod

Krovna organizacija koja se bavi promoviranjem i standardiziranjem „cloud“ tehnologija, Cloud Native Computing Foundation definira pojam "cloud-native" kao: korištenje softverskog stoga otvorenog kôda (engl. *open source software stack*) za kontejnerizaciju, pri čemu je svaki dio aplikacije zapakiran u vlastiti kontejner, dinamički orkestriran tako da je svaki dio aktivno upravljan kako bi optimizirao korištenje resursa te dizajniran i izrađen kao mikroservis, kako bi povećao ukupnu agilnost i održivost aplikacije.

Aplikacije opisane sintagmom "cloud-native" masovno se izrađuju i koriste tek u posljednjih nekoliko godina. Razloga za to ima više od kojih je najizraženiji povećanje dostupnosti infrastrukture koju nazivamo oblakom (engl. *cloud*). Tehnološki napredak koji je doveo do mogućnosti uspostave takve infrastrukture posredno je omogućio i promjenu paradigme aplikacija koje se nazivaju "cloud-native". Općenito govoreći, aplikacije koje su prethodile "cloud-native" aplikacijama i suprotstavljene su im prvenstveno po temeljnoj filozofiji korištenja i izrade nazivaju se monolitne aplikacije. Osnovna obilježja "cloud-native" aplikacija su kontejnerizacija, mikroservisna arhitektura i dinamička orkestracija. Sva navedena obilježja biti će razrađena u ovom radu. Osim temeljnog pitanja što su to "cloud-native" aplikacije biti će pokazano što takva ogledna aplikacija sadrži, kako ju izraditi, postaviti u prikladno okruženje i njome u istome upravljati. Kako bi naglasak bio na samoj paradigmi, svrha aplikacije je podržavanje poslovanja knjižnice. Aplikacija je smještena u poslovnu domenu jer su ovdje korištene potporne tehnologije i "cloud-native" paradigma najprimjenjivije u istoj. Iako je isti skup alata moguće iskoristiti u znanstvenoj ili nekoj drugoj domeni, kombinacija složenosti alata i aplikativnih funkcionalnosti bi dovela do gubitka fokusa te bi rad izgubio na jasnoći. Dodatni razlozi izbora jezgrenih funkcionalnosti aplikacije su mogućnost jednostavne prilagodbe poslovne domene na druga područja poslovanja i mogućnost objave nastalog kôda kao otvorenog s većom mogućnošću da će ga netko zaista i koristiti, s obzirom da su knjižnice najčešće neprofitne organizacije. Rad je strukturiran u pet poglavlja. Prvo poglavlje sadrži uvod, drugo daje teoretsku osnovu i definicije *cloud-native* aplikacija, treće daje pregled i opis relevantnih tehnologija korištenih za izradu praktičnog dijela rada, četvrto sadrži pregled arhitekture izrađene aplikacije dok posljednje, peto, daje pregled korištenja i funkcionalnosti aplikacije.

2. Cloud-native paradigma izrade aplikacija

Kao što je spomenuto u uvodu, tri su osnovna obilježja aplikacija koje se nazivaju „cloud-native“: mikroservisna arhitektura, kontejnerizacija te dinamička orkestracija. Karakteristike svakog spomenutog obilježja opisane su niže.

2.1. Mikroservisna arhitektura

Pojam mikroservisna arhitektura se posljednjih nekoliko godina koristi kao obilježje specifičnog načina dizajniranja softvera kao garniturâ servisa koje je moguće neovisno razviti i isporučiti.[1]

Mikroservisni arhitekturalni stil je pristup razvoju jedne aplikacije kao garniture malih servisa od kojih se svaki izvodi u svom procesu pri čemu servisi komuniciraju jednostavnim mehanizmima, često API sučeljima (engl. *application programming interface*) koristeći HTTP resurse. Navedeni servisi su izrađeni oko poslovnih zahtjeva i moguće ih je postaviti (engl. *deploy*) neovisno jednoga o drugome koristeći se pri tome potpuno automatiziranim procesom i strojevima. Središnje upravljanje takvih servisa svedeno je na nužni minimum te servisi mogu biti napisani u različitim programskim jezicima i koristiti različite tehnologije pohrane podataka.

Kako bi objasnili mikroservisni arhitekturni stil, korisno ga je usporediti s monolitnim. Monolitna aplikacija izrađena je kao jedna cjelina ili jedna velika jedinica. Takva monolitna aplikacija iz poslovne domene se najčešće sastoji od tri dijela: klijentskog sučelja (to su najčešće HTML i JavaScript kôd u pregledniku koji se izvodi na stroju krajnjeg korisnika), baze (sastoji se od više tablica umetnutih u istu, često relacijsku bazu) i serverske aplikacije koja zaprima zahtjeve, izvršava logiku, dohvaća i mijenja podatke s baze i stvara HTML poglede (engl. *view*) koje šalje pregledniku. U monolitnim aplikacijama upravo ovaj posljednji dio, serverska aplikacija je jedna logička izvršivost. Posljedica toga je da svaka promjena sustava uključuje izradu i postavljanje nove verzije cjelokupne serverske aplikacije. Monolitni pristup je prirodan jer posredstvom svojstava programskih jezika omogućuje podjelu aplikacije na klase i funkcije. Cijela aplikacija izrađena kao monolit

[1] https://martinfowler.com/articles/microservices.html

može se pokrenuti i testirati na lokalnom računalu razvojnog inženjera, dok se horizontalno može skalirati tako da se pokrene više instanci iste aplikacije na istom ili različitom poslužitelju, nužno iza razdjelitelja opterećenja (engl. *load balancer*).

Pojavom infrastrukture koju se naziva oblakom (engl. *cloud*) monolitne aplikacije počinju gubiti na važnosti te omogućuju rast mikroservisnih aplikacija zbog očitih prednosti koje mikroservisi donose u odnosu na monolitne aplikacije: ciklusi izmjene poslovne logike prestaju biti vezani, promjena jednog dijela aplikacije zbog promjene poslovnih zahtjeva više ne zahtijeva ponovno postavljanje cijelog monolita te se skaliranje radi tako da se skalira samo dio aplikacije, odnosno mikroservisi koji su spori, a ne kao prije, cijele monolitne aplikacije.

Sve navedeno dovelo je do toga da mikroservisna arhitektura u kojoj su aplikacije izrađene kao garniture servisa prevlada. U njoj su servisi neovisno isporučivi i skalabilni, svaki servis jasno definira svoje granice do razine da mogu biti pisani u različitim programskim jezicima i mogu biti održavani od različitih timova.

2.1.1. Karakteristike mikroservisne arhitekture

Prema Fowleru najmanjim zajedničkim nazivnikom mikroservisne arhitekture se mogu smatrati sljedeće karakteristike[2]:

1. Razdvajanje u komponente putem servisa

Želja za izradom cjelovitih softverskih sustava spajanjem komponenti postoji oduvijek jer oponaša ono što se događa u fizičkom svijetu. Ovdje je potrebno pojasniti što je komponenta, komponenta je jedinica softvera koju je moguće zamijeniti i unaprijediti neovisno o drugim komponentama, dok je servis skup komponenti koje komuniciraju putem mehanizama kao što su web servisi ili pozivi udaljenih procedura (engl. *remote procedure call*).

Bitna posljedica korištenja servisa kao komponente je mogućnost boljeg definiranja načina na koji će taj servis komunicirati. Često samo dokumentacija i disciplina stoje između narušavanja principa učahurivanja (engl. *encapsulation*) komponente što dovodi do

[2] https://martinfowler.com/articles/microservices.html

pretjeranog čvrstog vezanja (engl. *tight coupling*) među komponentama. Servisi olakšavaju izbjegavanje navedenoga koristeći eksplicitne mehanizme udaljenog pozivanja.

Loša strana ovakvog dizajna je što je resursno zahtjevnija i što u slučaju preraspodjele odgovornosti među komponentama izvedba postaje kompliciranija i dugotrajnija.

2. Organizacija oko poslovnih zahtjeva

Pri pokušaju podjele velike aplikacije na dijelove, menadžment se često fokusira na tehnološki sloj što dovodi do stvaranja timova za korisničko iskustvo, timova za serversku logiku i timova za baze podataka. Kada su timovi tako podijeljeni, čak i jednostavne izmjene aplikacije mogu dovesti do kros-timskih projekata koji traju i koštaju. Učinkovit tim će optimizirati rad izabirući manje od dva zla - jednostavno će nagurati logiku u koju god aplikaciju da ima pristup što će dovesti do toga da će poslovna logika biti posvuda po aplikaciji. Ovo je primjer Conwayevog zakona[3] u praksi koji glasi: "Organizacija koja dizajnira sustav iznaći će dizajn čija je struktura kopija komunikacijske strukture te organizacije". Kao posljedica proizlazi činjenica da silosni funkcionalni timovi dovode do silosnih aplikacijskih arhitektura.

Mikroservisni pristup podjele je drugačiji utoliko što aplikaciju dijeli na servise organizirane oko poslovnih zahtjeva. Takvi servisi koriste implementaciju koja omogućuje izradu softvera isključivo za zadano poslovno područje. Posljedično timovi koji rade na taj način su kros-funkcionalni uključivo po rasponu vještina potrebnih za razvoj kao što su korisničko iskustvo, baze podataka i projektni menadžment.

Velike monolitne aplikacije se također mogu podijeliti u module organizirane oko poslovnih zahtjeva, makar to najčešće nije slučaj. Izrada velikog monolita također zahtjeva od tima koji ga izrađuje podjelu po poslovnim zahtjevima, međutim, bez jasne podjele u servise granice imaju tendenciju postati nejasne, dok članovi tima teško barataju svim kontekstima. Prirodna, kognitivna i mentalna ograničenja pojedinca onemogućuju baratanje svim kontekstima svih modula u monolitu iz kratkoročnog sjećanja, što je nužno za učinkovitost svakog pojedinca koji razvija aplikacije. Servisi kao komponente nužno eksplicitnije postavljaju granice funkcionalnosti što ujedno omogućuje i jasnije postavljanje granica odgovornosti razvojnih timova.

[3] https://itrevolution.com/conways-law/

3. Proizvodi umjesto projekata

Većina pokušaja razvoja aplikacija koristi projektni model pri čemu je cilj isporučiti aplikaciju koja se zatim smatra završenom. Po završetku, softver je predan organizaciji koja isti održava i projektni tim koji je aplikaciju izradio se raspušta. Zagovornici mikroservisa predlažu izbjegavanje ovog modela i preferiraju ideju posjedovanja proizvoda tokom cijelog njegovog životnog ciklusa. Razvojni tim bi trebao preuzeti punu odgovornost za proizvod čak i nakon što uđe u produkciju. Mentalitet proizvoda, ne projekta dovodi do povezivanja poslovnih zahtjeva sa sustavom koji ih izvršava. Umjesto da se softver promatra kao skup funkcionalnosti koje treba završiti, postoji trajni odnos u kojem je osnovno pitanje kako će softver pomoći svojim korisnicima da poboljšaju izvršavanje poslovnih zahtjeva. Isti pristup moguće je primijeniti i na monolitne aplikacije, ali sitnija granulacija servisa može omogućiti stvaranje osobne povezanosti između timova koji softver izrađuju i zahtjeva njihovih korisnika.

4. Pametne završne točke (engl. *endpoints*) i glupe cijevi

U prošlosti je značajna količina logike stavljana u komunikacijske mehanizme pri izgradnji komunikacijskih struktura između različitih procesa. Primjer toga je poslovna servisna sabirnica (engl. *Enterprise Service Bus*) gdje ESB proizvodi uključuju mehanizme za usmjeravanje poruka, transformaciju poruka ili primjenu poslovnih pravila.

Mikroservisna arhitektura preferira drugačiji pristup koji se kolokvijalno naziva: pametni *endpointi* i glupe cijevi. Aplikacije izgrađene od mikroservisa pokušavaju biti što neovisnije i manje isprepletene te kohezivne. One posjeduju samo svoju domensku logiku i pokušavaju činiti samo nužno: zaprimiti zahtjev, primijeniti logiku svoje poslovne domene i isporučiti odgovor. Pri tome nastoje koristiti jednostavne REST-u slične protokole nasuprot onima kompleksnima iz WS ili BPEL obitelji i nastoje izbjeći orkestraciju komunikacije putem središnjeg alata.

Najčešće korišteni protokoli su HTTP ili protokoli jednostavnih poruka (engl. *lightweight messaging*) u koje spadaju implementacije poput RabbitMQ-a ili Apache Kafke. Navedeni protokoli omogućuju pouzdano asinkrono komuniciranje, različito od inherentne sinkronosti HTTP-a i prepuštaju, kao i HTTP, da poslovna logika odsjeda u završnim točkama (engl. *endpoints*), odnosno servisima.

U monolitu se komunikacija između komponenti odvija u procesu, najčešće putem pozivanja metoda ili zvanja funkcija. Najveći izazov pri transformaciji iz monolita u mikroservise je upravo u promjeni obrazaca komunikacije.

5. Decentralizirana vladavina (engl. *governance*)

Jedna od posljedica centraliziranog upravljanja je sklonost standardizacije oko jedne tehnološke platforme. Iskustvo pokazuje da ovaj pristup ima značajna ograničenja. Podjelom monolita na servise stvara se mogućnost izbora tehnologije pri izradi svakog pojedinog servisa. Aplikativnu logiku moguće je pisati u programskom jeziku Node.js ili C++, jednako kao što je moguće i izabrati drugu vrstu baze podataka koja bolje odgovara traženoj namjeni.

Timovi inženjera koji izrađuju mikroservise također preferiraju različite standarde. Umjesto da se drže propisanih standarda, preferiraju koristiti vlastite ili tuđe korisne alate koji su najčešće u domeni otvorenog koda.

Ilustracija decentralizirane vladavine je način na koji Amazonovi inženjeri koji su kôd izradili preuzimaju odgovornost za održavanje produkcije istoga. Pristup je prirodan jer su poticaji neposredni i mjerljivi. Mogućnost da razvojnog inženjera probudi obavijest o ispadu produkcije prirodno dovodi do skrbi o kvaliteti kôda.

6. Decentralizirano upravljanje podacima

Decentralizacija upravljanja podacima očituje se u nizu slučajeva. Na najapstraktnijoj razini označava kako će se konceptualni model svijeta razlikovati između različitih sustava. Pogled prodaje na korisnika će biti različit od pogleda korisničke podrške. Sam entitet korisnika iz perspektive prodaje ne mora nužno niti postojati u pogledu kojim barata podrška, a kamoli biti isti.

Koristan način za premostiti ovu razliku je pojam "vezanog konteksta" (engl. *Bounded Context*) iz paradigme domenski pokretanog dizajna (engl. *Domain-Driven Design*). Domenski pokretan dizajn dijeli kompleksnu domenu u višestruke vezane kontekste i definira odnose između njih.

Osim decentraliziranja odluka o konceptualnim modelima, mikroservisi također decentraliziraju odluke o pohrani podataka. Dok monolitne aplikacije preferiraju jednu logičku bazu podataka za trajne podatke, poslovanje često preferira jednu bazu koju koristi više aplikacija pri čemu su mnoge odluke pokretane prodajnim modelima i modelima licenciranja proizvođača baza podataka. U mikroservisnoj arhitekturi upravljanje vlastitom bazom podataka prepušteno je svakom pojedinom servisu.

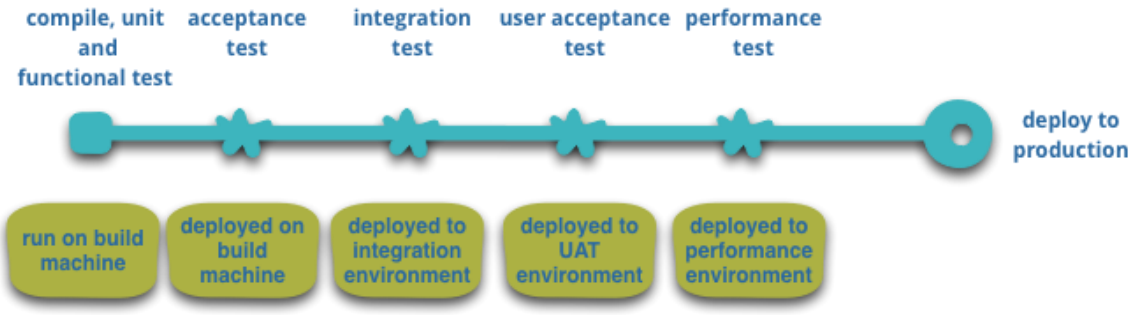
Decentraliziranje odgovornosti za podatke kod mikroservisa ima implikacije na promjenu podataka u bazi. Uobičajeni pristup je korištenje transakcija kako bi se jamčila dosljednost pri promjeni više resursa istovremeno. Ovo je slučaj kako se baze podataka koriste u monolitnim aplikacijama. Korištenje transakcija na ovaj način pomaže pri dosljednosti podataka (engl. *data consistency*), ali uzrokuje značajno vremensko ispreplitanje, što je problematično u slučaju kada postoji više servisa. Raspodijeljene transakcije su kompliciranе za implementiranje te kao posljedica toga mikroservisna arhitektura naglašava koordinaciju između servisa bez korištenja transakcija. pri tome eksplicitno pristaju da dosljednost baze podataka može biti samo eventualna. (engl. *eventual consistency*). Termin eventualna dosljednost se može definirati kao stanje baze podataka u kojoj će sve instance baze naposljetku imati iste podatke, ali se to ne može jamčiti u svakom trenutku vremena[4].

Izbor upravljanja nedosljednostima na ovaj način je novi izazov za mnoge razvojne timove, ali se često poklapa s poslovnom praksom. Poslovanje često pristaje na određeni stupanj nedosljednosti u podacima kako bi bilo u stanju brže reagirati na potražnju, imajući pričuvni proces koji naknadno ispravlja eventualne pogreške. Ovakav kompromis je prihvatljiv sve dok je trošak ispravljanja pogrešaka manji od troška izgubljenog prihoda nastalog većom dosljednošću.

7. Automatizacija infrastrukture

Tehnike automatizacije infrastrukture su značajno evoluirale posljednjih godina – evolucija platforme oblaka je smanjila operativnu složenost izrade, postavljanja i upravljanja mikroservisima. Mnoge proizvode ili sustave koji su izgrađeni od mikroservisa izradili su timovi razvojnih inženjera s značajnim iskustvom rada po modelu koji se naziva kontinuirana isporuka (engl. *continuous delivery*) i onoga što joj prethodi, a to je kontinuirana integracija (engl. *continuous integration*). Timovi razvojnih inženjera koji izrađuju softver na ovaj način obilno koriste tehnike automatizacije infrastrukture kao što je prikazano na slici (Slika 2.1).

[4] https://stackoverflow.com/questions/10078540/eventual-consistency-in-plain-english



Slika 2.1. Osnovni slijedni dijagram automatiziranog procesa postavljanja mikroservisa[5]

Ključna svojstva kontinuirane isporuke i kontinuirane integracije su izvršavanje što je moguće više automatiziranih testova kako bi se osigurala funkcionalnost softvera u svakoj fazi njegove izrade, a pogotovo prije nego što dođe do produkcije. Dodatno svojstvo navedenog pristupa je unaprjeđenje softvera cijelom dužinom prikazanog procesa, što znači da je postavljanje softvera na svaku sljedeću okolinu potpuno automatizirano.

Ovaj pristup funkcionira jednako dobro za monolitne kao i mikroservisne aplikacije jer kada se jednom ulože resursi u automatizaciju puta do produkcije u slučaju monolita, onda postavljanje više aplikacija ne povećava složenost. Jedan od ciljeva kontinuiranog postavljanja je da učini postavljanje „dosadnim“, jamčeći tako da aplikacija ponovljeno prolazi kroz sve faze ciklusa postavljanja izvršavajući sve testove i ne ostavljajući značajnog mjesta za pogreške u produkciji.

8. Dizajn za ispade

Posljedica korištenja servisa kao komponenti je da aplikacije trebaju biti dizajnirane tako da mogu tolerirati zatajivanje servisa. Bilo koji poziv servisa može zatajiti uslijed nedostupnosti pružatelja tocdg servisa, u kojem slučaju klijent mora odgovoriti što je više moguće dostojanstveno (engl. *gracefully*). Ovo je nedostatak u odnosu na monolitne aplikacije jer uvodi dodatnu kompleksnost potrebnu za upravljanje navedenom dostojanstvenošću. Posljedica ovoga je da timovi koji izrađuju mikroservise konstantno moraju razmišljaju o tome kako zatajenja servisa utječu na korisničko iskustvo. Kao posljedica postoje posebni alati koji simuliraju ispade servisa ili čak podatkovnih centara tokom radnog dana kako bi se testirala otpornost aplikacija i njihov nadzor.

[5] https://martinfowler.com/articles/microservices/images/basic-pipeline.png

Kako servisi mogu zakazati u svakom trenutku, bitno je moći otkriti ispade brzo, i ukoliko je moguće, automatski oporaviti servis. Mikroservisne aplikacije stavljaju naglasak na nadziranje u stvarnom vremenu, provjeravajući i arhitekturne elemente (koliko zahtjeva u sekundi baza dobiva) i poslovno bitne metrike (koliko narudžbi u minuti je zaprimljeno). Takvo semantičko nadziranje može pružiti sustave ranog uzbunjivanja koje obavještava razvojne timove da poprate i istraže situaciju.

Timovi koji izrađuju mikroservise očekuju da mogu vidjeti sofisticirane sustave nadziranja i logiranja za svaki pojedini servis na kontrolnim pločama sa statusima dostupnosti i raznim operativnim i poslovno relevantnim metrikama. Detalje o statusu osigurača (engl. *circuit breaker*), trenutnu propusnost i latenciju.

9. Evolucijski dizajn

Razvojni inženjeri koji razvijaju mikroservise najčešće imaju iskustvo korištenja evolucijskog dizajna i vide dekompoziciju servisa kao dobar alat za omogućavanje više kontrole nad promjenama aplikacije bez usporavanja samih promjena. Kontrola promjene ne znači smanjenje promjene - s pravim pristupom i alatima moguće je izvršavati česte, brze i dobro kontrolirane promjene u aplikacijama.

Kada se god softverski sustav pokuša podijeliti u komponente, potrebno je donijeti odluku kako tu podjelu razgraničiti. Koji su principi na kojima se podjela temelji? Ključno svojstvo svake komponente je princip neovisne zamjene i unaprjeđenje pojedine komponente. Navedeno implicira potragu za mjestima gdje se može zamisliti prepisivanje komponente bez utjecaja na servise s kojima je u interakciji. Još jedan korak dalje bi bilo ugrađeno očekivanje da će u budućnosti pojedini servis biti u potpunosti izbačen umjesto da ga se unaprjeđuje.

Nije rijedak primjer postojanje aplikacije koja je izgrađena kao monolit, ali je evoluirala u smjeru mikroservisa. Monolit je i dalje u srži aplikacije, ali se nove funkcionalnosti dodaju izgradnjom mikroservisa koji koriste monolitova aplikacijska programska sučelja (engl. *application programming interface*). Ovakav pristup je primjeren za inherentno privremena svojstva, pod čime se podrazumijevaju poslovni zahtjevi koji su po svojoj prirodi privremeni (poslovne prilike koje su jednokratne) što omogućuje potpuno gašenje servisa koji ih podupiru nakon samo nekoliko tjedana ili mjeseci.

Naglasak na zamjenjivosti je poseban slučaj principa modularnog dizajna. Taj princip kaže kako je poželjno da se karakteristike koje se mijenjaju istovremeno nalaze u istom modulu.

Dijelovi sustava koji se mijenjaju rijetko trebali bi biti u zasebnim servisima odvojenima od onih koji se mijenjaju često. Mijenjanje dva ili više servisa istovremeno su pokazatelj da bi se možda trebali spojiti u jedan.

Stavljanje komponenti u servise omogućava planiranje isporuka u više manjih isporučivih cjelina. Kod monolita svaka promjena zahtijeva potpunu izgradnju i isporuku cijele aplikacije, dok kod mikroservisa treba isporučiti samo servise koji su promijenjeni. Ovo može pojednostaviti i ubrzati proces isporuke. Loša posljedica navedenoga je da je potrebno brinuti o mogućnosti da promjene na jednom servisu mogu utjecati na sve njegove klijente. Tradicionalni pristup integracije rješava ovaj problem korištenjem različitih inačica (engl. *versioning*), dok se kod mikroservisa korištenje inačica smatra nepoželjnim te se može izbjeći dizajniranjem servisa kako bi bili otporni koliko je god moguće na promjene u svojim dobavljačima, odnosno drugim servisima o kojima ovise.

2.2. Kontejnerizacija

Pod pojmom kontejnerizacija podrazumijeva se alternativa virtualnim strojevima koja uključuje učahurivanje aplikacije u kontejner skupa sa svojim operacijskim sustavom. Precizniji termin bi bio virtualizacija na razini operacijskog sustava, dok je kolokvijalni termin kontejnerizacija preuzet iz discipline logistike jer brodski kontejner u stvarnosti oponaša ono što kontejner na razini operacijskog sustava čini virtualno. Poput virtualnih strojeva kontejneri omogućuju pakiranje aplikacija skupa s pripadajućim potpornim datotekama o kojima ovise, pružajući tako izolirane okoline u kojima se aplikacije pokreću.[6]

Kontejneri pružaju mehanizam logičkog pakiranja aplikacija na način da su aplikacije odvojene od okoline u kojoj se izvode. Ovo razdvajanje omogućuje aplikacijama koje se izvode u kontejneru da budu jednostavno i dosljedno postavljene, neovisno o tome je li ciljna okolina privatni podatkovni centar, infrastruktura javnog oblaka ili čak programerov osobni laptop. Kontejnerizacija pruža jasno razdvajanje odgovornosti (engl. *separation of concern*), gdje se programeri mogu usredotočiti na aplikativnu logiku, dok se sistemski inženjeri mogu usredotočiti na postavljanje aplikacija u produkciju i upravljanje istima bez fokusiranja na detalje poput verzije softvera ili konfiguracije specifične za svaku pojedinu aplikaciju.

2.2.1. Prednosti korištenja kontejnera

Umjesto virtualizacije cijelog hardverskog stoga (engl. *hardware stack*) kao u slučaju virtualnih strojeva, kontejneri virtualizaciju vrše na razini operacijskog sustava tako da se više kontejnera izvodi nad jednim *kernelom* operacijskog sustava. Posljedica navedenog je da su kontejneri resursno manje zahtjevni zbog zajedničkog korištenja *kernela* operacijskog sustava, pokreću se brže i koriste djelić memorije u usporedbi s pokretanjem cijelog novog operacijskog sustava, što je slučaj pri korištenju virtualnog stroja. Postoji mnogo različitih formata kontejnera, od kojih je najpopularniji Docker koji je otvorenog koda i trenutno najšire rasprostranjen.

Najznačajnije prednosti korištenja kontejnerske tehnologije su:

1. Dosljedna okolina

[6] https://cloud.google.com/containers/

Kontejneri pružaju razvojnim inženjerima mogućnost stvaranja predvidivih okolina koje su izolirane od drugih aplikacija. Također, mogu uključivati zavisnosti (engl. *dependencies*) potrebne za pokretanje aplikacije. Razvojnom inženjeru navedeno jamči dosljednost neovisnu o okolini u koju ću aplikacija na produkciji biti postavljena. Ovakva vrsta jamčene dosljednosti se iskazuje u obliku veće produktivnosti, razvojni i sistemski inženjeri provode manje vremena popravljajući greške u kodu i utvrđujući razlike između verzija, a provode više vremena razvijajući i isporučujući nove funkcionalnosti krajnjim korisnicima. Također postoji jamstvo proizvodnje manje pogrešaka u produkciji zbog istovjetnosti produkcijske okoline s razvojnom i testnom.

2. Mogućnost pokretanja bilo gdje

Kontejnere je moguće pokrenuti gotovo bilo gdje, čime se uvelike olakšava razvoj i isporuka aplikacija. Ovo uključuje Linux, Windows i Mac operacijske sustave, virtualne i fizičke strojeve, razvojna osobna računala ili servere u podatkovnim centrima kao i infrastrukturu javnog oblaka. Rasprostranjenost Docker formata za kontejnere dodatno pomaže sa prijenosnošću omogućavajući pokretanje kontejnera gotovo bilo gdje.

3. Izolacija

Kontejneri virtualiziraju procesor, memoriju, pohranu i mrežne resurse na razini operacijskog sustava osiguravajući programerima svojstvo pješčanika (engl. *sandbox*) gdje su aplikacije pokrenute u jednom kontejneru odvojene od onih pokrenutih u drugom. Izolacija pruža mnoge prednosti od kojih je među najvažnijima nemogućnost procesa u jednom kontejneru da neželjeno djeluje na resurse ili ponašanje procesa u drugom, čime je onemogućena krajnost u kojoj bi jedna neispravna aplikacija lančanom reakcijom uzrokova ispade svih koji o njoj ovise.

2.2.2. Način korištenja kontejnera

Kontejneri omogućuju pakiranja aplikacija i njihovih ovisnosti u jedno cjelinu koja može biti upravljana po verzijama, omogućavajući lako umnožavanje među programerima u timu i među strojevima koji čine infrastrukturu. Kao što aplikacijske knjižnice (engl. *software libraries*) pakiraju dijelove koda skupa omogućavajući programerima apstrakciju logike poput autentikacije korisnika ili upravljanja sjednicama (engl. *session*), kontejneri omogućavaju aplikacijama u cjelini da budu zapakirane, apstrahirajući operacijski sustav, stroj ili sami kôd. Upareno s arhitekturom temeljenom na uslugama, cijela logička jedinica

o kojoj programeri trebaju voditi računa je puno manja, što dovodi do veće agilnosti i produktivnosti. Posljedica je lakši razvoj, testiranje, postavljanje i upravljanje aplikacijom.

Kontejneri najbolje funkcioniraju zajedno s arhitekturom temeljenoj na uslugama. Razdvajanje i podjela rada karakteristične za ovu arhitekturu omogućuje nekim servisima nastavak rada čak i u slučaju prestanka rada drugih, čineći aplikaciju u cjelini pouzdanijom. Razdvajanje u komponente omogućuje brži i pouzdaniji razvoj. Manje cjeline kôda su lakše za održavanje i budući da su servisi razdvojeni lako je testirati pojedine ulaze u odnosu na pripadajući izlaz. Kontejneri savršeno pristaju uz aplikacije temeljene na servisima jer je moguće provjeriti zdravlje (engl. *health*) svakog kontejnera pojedinačno, ograničiti svaki servis na zadane resurse te pokretati i zaustavljati kontejnere neovisno jedne o drugima. Kako kontejneri odvajaju kod, omogućavaju svakom pojedinom servisu promatranje drugog kao "crne kutije", dodatno smanjujući opseg odgovornosti programera.

2.3. Dinamička orkestracija

Pod pojmom dinamičke orkestracije u kontekstu „cloud-native“ aplikacija podrazumijeva se mogućnost automatiziranog postavljanja, skaliranja i upravljanja kontejneriziranim aplikacijama. Jednom kada su aplikacije zapakirane u kontejnere potrebno je njima na učinkovit način upravljati, što isključuje mogućnost obavljanja tako složenog procesa ručno. Zbog toga su se pojavili sustavi za dinamičko orkestriranje kontejnera od kojih je svakako najpoznatiji Kubernetes. Zadaci koje je potrebno uspješno riješiti u sustavu izgrađenom od mikroservisa mogu biti složeni zbog samog broja mikroservisa koji sustav izgrađuju pa bi ručno obavljanje takvog zadatka dovelo do velikog broja pogrešaka kao i stalnog, ponavljajućeg posla. Jedno od osnovnih svojstava koje sustavi dinamičke orkestracije pružaju je briga o servisnoj topologiji. Kontejneri su zamišljeni kao tehnologija koja u normalnim okolnostima ne sprema informaciju o svojem stanju te u svakom trenutku bilo koji kontejner može biti ugašen ili zamijenjen drugim na bilo kojem čvoru u mreži. U takvom sustavu potrebno je znati koji kontejner je dostupan i gdje se nalazi kako bi mu se mogli prosljeđivati zahtjevi. Logika usmjeravanja servisnog prometa temeljena na topologiji čvorova u mreži izvršava se upravo na sustavu dinamičke orkestracije. U složenom sustavu s mnogo mikroservisa očekuje se da će neki od njih, ili bar neke instance nekih od njih, biti privremeno nedostupne ili nespremne. Kako se takvim elementima sustava ne bi moralo upravljati ručno, sustav dinamičke orkestracije je zadužen za nadziranje zdravlja (engl. *health*) svakog pojedinog kontejnera, te će u slučaju da nova promjena uzrokuje da je

kontejner nezdrav automatski isti isključiti i zamijeniti ga prijašnjom ispravnom verzijom kao bi usluga u cjelini nastavila raditi, pa makar i bez novih osobina koje su trebale biti dodane učinjenim promjenama. Naposljetku, kako svaka promjena konfiguracije aplikacije ne bi zahtijevala ponovno postavljanje iste, te kako bi aplikacija mogla uistinu biti bez stanja (engl. *stateless*), potrebno je negdje spremiti njenu konfiguraciju. Najprimjerenije mjesto za navedeno je upravo sustav dinamičkog upravljanja.

3. Tehnologije izrade aplikacije

3.1. Java

Java je najpopularniji[7] objektno orijentirani programski jezik za opću uporabu. Osobito je popularan u poslovnim primjenama zbog svoje jednostavnosti, stabilnosti, kompatibilnosti prijašnjih verzija i činjenice da je otvorenog koda. Značajno obilježje Java programskog jezika je WORA princip (engl. *write once, run anywhere*) koji označava mogućnost pisanja Java kôda jednom, kojeg je kompajliranog potom moguće izvršavati na bilo kojoj platformi koja podržava Java programski jezik bez potrebe za rekompajliranjem. Java aplikacije se najčešće kompajliraju u oblik koji se zove *bytecode*, koji se potom može izvršavati na bilo kojem Java virtualnom stroju (JVM) neovisno o računalnoj arhitekturi.

Navedene karakteristike, uz postojanje programskih okvira kojima je moguće nadograditi programski jezik Java čine ga logičnim izborom za izradu aplikacija koje pripadaju poslovnoj domeni, iako programski jezik Java nije ograničen na poslovne primjene. Pri izradi poslovnih aplikacija s „cloud-native“ karakteristikama programski jezik Java nameće se kao logičan i prvi izbor, što je i razlog njegova izbora kao temelja izrade aplikacije izrađene u ovom radu.

3.2. Spring programski okvir i Spring Boot

Spring je aplikacijski programski okvir, sa svojstvom ubrizgavanja ovisnosti (engl. *dependency injection*) i kontejner kontrole inverzije (engl. *inversion of control*) za Java platformu.

U zajednici programskog jezika Java postoji mnogo kontejnera koji pomažu u sastavljanju komponenti iz različitih projekata u jednu kohezivnu aplikaciju. Navedenim kontejnerima je zajednički obrazac koji opisuje kako kontejneri odrađuju unutrašnje prespajanje, generički nazvan kontrola inverzije ili ponekada specifičnije ubrizgavanje ovisnosti.

J2EE tehnologije, kao dominantne u Java svijetu su vrlo brzo dobile alternativu zbog svoje složenosti i tromosti. Ta alternativa je bila najčešće otvorenog koda. Problem koji J2EE i

[7] https://www.tiobe.com/tiobe-index/

alternative rješavaju je kako spojiti različite elemente kao što je npr. spajanje jedne web kontroler arhitekture s drugim sučeljima za bazu podataka, kada su oba napisana od različitih timova ljudi koji nisu upoznati s međusobnim načinom rada. Alternativni programski okviri se ističu time što su jednostavniji i lakši (engl. *lightweight*), a najpoznatiji takav programski okvir je upravo Spring.[8]

Jednostavan primjer inverzije kontrole je primjer kontrole korisničkog sučelja. Nekada su korisnička sučelja bila kontrolirana od strane glavnog programa, na zaslonu su se točno definiranim redoslijedom pojavljivali opisi unosa koji se očekuju sljedeći. Glavni program je ispisivao što se očekuje i spremao korisnički odgovor prije nego što prijeđe na sljedeći element niza unosa. Grafička korisnička sučelja dovode do inverzije kontrole, glavni program pruža *event handlere* za različita polja na sučelju, ali kontrola je doživjela inverziju, s programera na programski okvir (engl. *framework*). Na ovaj način funkcionira inverzija kontrole u cijelom Spring programskom okviru. Precizniji termin za inverziju kontrole je ubrizgavanje ovisnosti (engl. *dependency injection*).

3.3. H2 Baza podataka

H2 baza podataka je napisana isključivo u Java programskom jeziku, otvorenog je koda i često se koristi za razvoj prototipa aplikacija. Može raditi u tri načina rada. Prvi način je ugrađeni (engl. *embedded*) način rada koji podrazumijeva da je baza dostupna samo iz trenutnog Java virtualnog stroja pri čemu je komunikacija aplikacije s bazom izrazito brza. Drugi način je klijent-poslužitelj, što je klasični način rada većine baza korištenih u poslovne svrhe. Na ovaj način više aplikacija se može kao klijent spojiti na poslužitelj koji je baza podataka. S obzirom da se komunikacija odvija putem TCP/IP protokola, ovaj način je sporiji od ugrađenog. Posljednji, miješani način rada podrazumijeva kreiranje H2 baze od prve aplikacije, ali istovremeno i pokretanje poslužitelja na koji se druge baze mogu spojiti. H2 baza podataka može spremati podatke izravno u memoriji što podrazumijeva njihov potpuni gubitak pri zatvaranju veze prema bazi, ili na disku što omogućava trajnu pohranu cijele baze u jednoj datoteci. Baza podržava podskup standardnog ANSI SQL jezika dovoljan za normalno funkcioniranje same baze. Sučelja komunikacije s bazom su SQL i JDBC, međutim baza podržava i PostgreSQL ODBC driver čime čini kasniji prelazak na

[8] https://martinfowler.com/articles/injection.html#InversionOfControl

PostgreSQL bazu osobito jednostavnim. Iako neprimjerena produkcijskom okruženju poslovnih aplikacija, H2 baza podataka se savršeno uklapa u arhitekturu pokaznih i jednostavnih aplikacija zbog svoje jednostavnosti i prenosivosti (cijela baza je spremljena u jednom dokumentu) te dolazi s ugrađenim poslužiteljem i konzolom koju je moguće pokrenuti u bilo kojem internetskom pregledniku.[9]

3.4. Liquibase

Liquibase je datoteka (engl. *library*) otvorenog kôda koja se koristi za praćenje, upravljanje i primjenu sheme baze podataka neovisno o tipu same baze podataka. Motivacija za nastajanje ovakve datoteke je omogućavanje lakšeg verzioniranja stanja sheme baze podataka, što je osobito bitno u agilnom pristupu razvoju softvera. Mehanizam spremanja promjena nad bazom podataka su tekstualne datoteke XML, YAML, JSON ili SQL tipa. Navedene datoteke su jednoznačno određene spojem "id" i "author" oznaka (engl. *tag*) kao i imenom datoteke. Lista svih primijenjenih promjena spremljena je u samoj bazi te se pri svakoj naknadnoj promjeni baze uzima u obzir kako bi se utvrdilo koje promjene nad bazom je preostalo primijeniti. Rezultat je izostanak verzije baze podataka, ali ovaj pristup omogućuje rad u okruženjima s više razvojnih inženjera i grana kôda. Neke od mogućnosti Liquibasea su vraćanje na prethodno stanje baze podataka po vremenu, broju predaja (engl. *commit*) ili po oznakama (engl. *tag*). Mogućnost izrade izvještaja o razlikama između baza podataka, podrška za trenutno deset različitih DBMS-ova i mogućnost izvršavanja putem komandne linije ili alata kao što su Apache Ant, Maven, servlet kontejner ili Spring programski okvir. Najčešći slučajevi korištenja ovog alata su situacije u kojima je potrebno programski nadograditi ili vratiti prijašnje stanje baze podataka jer skripta imena „databasechangelog“ bilježi promjene sheme u vremenu. Uz navedeno, ključne prednosti datoteke su neovisnost o proizvođaču baze podataka što omogućuje relativno bezbolno migriranje na bazu drugog proizvođača, automatizirano stvaranje dokumentacije i mogućnost izrade izvještaja o razlikama shemâ dviju baza podataka. Ono što Liquibase razlikuje od drugih alata ili običnih DDL skripti pojedinih baza je mogućnost promjene sheme baze podataka bez gubitka istih.

[9] https://baptiste-wicht.com/posts/2010/08/presentation-usage-h2-database-engine.html

3.5. Hibernate

Hibernate je programski okvir (engl. *framework*) za objektno-relacijsko mapiranje. Namjena Hibernate programskog okvira je prevođenje poslovne logike koja se nalazi u aplikacijama i koristi objekte poslovne domene u podatke pogodne za spremanje u (najčešće) relacijsku bazu podataka. Osnovna ideja objektno-relacijskog mapiranja je mapiranje tablice baze podataka u programsku klasu.[10] U najosnovnijem slučaju to će značiti kako je jedan red iz baze podataka kopiran u instancu klase, dok će u obrnutom slučaju, spremanje objekta značiti umetanje jednog reda u tablicu u bazi. Navedeno je moguće postići i bez programskog okvira kao što je Hibernate, koristeći samo JDBC API sučelje, no ovakav pristup je značajno sporiji i podložniji pogreškama te je gotovo nemoguće pronaći aplikaciju iz realne poslovne domene koja bi tablice i objekte mapirala na ovakav, ručni način.[11] Prednost korištenja Hibernate programskog okvira je i u činjenici da je upravo Hibernate predefinirana implementacija JPA API-ja u programskom okviru Spring. Korištenjem Spring programskog okvira izbjegnuta je potreba kreiranja XML dokumenta koji sadrži pravila mapiranja i sva mapiranja se mogu raditi dinamički, postavljanjem odgovarajućih Spring boot anotacija nad pripadajuće klase.

3.6. Docker

Docker je platforma za pokretanje kontejnera i softver korišten za virtualizaciju na razini operacijskog sustava.[12] Razumijevanje Dockera teško je bez razumijevanja pojma kontejnera iz domene isporuke i izrade aplikacija. Uz razumijevanje kontejnera najlakši način za objasniti što je Docker je usporedba s virtualnim strojem. Kronološki, od navedenih tehnologija najstariji su virtualni strojevi. Kao što samo ime kaže, virtualni stroj je "stroj unutar stroja". Virtualan je jer je instaliran na operacijskom sustavu fizičkog stroja, koji mu je domaćin (engl. *host*) i u većini aspekata se ponaša kao gost (engl. *guest*). Ovo podrazumijeva korištenje zajedničkih resursa koje raspoređuje i ograničava operacijski sustav domaćin (u stvarnosti je situacija složenija zbog postojanja nadzornog softvera (engl.

[10] https://hibernate.org/orm/

[11] https://laliluna.com/jpa-hibernate-guide/ch01.html

[12]https://stackoverflow.com/questions/28089344/docker-what-is-it-and-what-is-the-purpose

Hypervisor)) gdje su resursi, unatoč mogućnosti ograničavanja, dijeljeni, odnosno nisu potpuno izolirani. U takvom tehnološkom stogu postoje brojne prednosti u odnosu na potrebu da svaki fizički stroj ima samo jedan operacijski sustav na kojem se vrte aplikacije. Najvažnije prednosti su mogućnost skaliranja resursa i ograničena izolacija. Skaliranje resursa je samo objašnjivo, u slučaju da aplikacija zahtijeva pojačane resurse, moguće ih je dodati klikom miša u konfiguraciji virtualnog stroja. Također, izolacija omogućuje da rad jedne aplikacije ima vrlo ograničeni utjecaj na rad druge aplikacije u odvojenom virtualnom stroju. Tako niti kritični ispadi aplikacije koji uzrokuju rušenje cijelog operacijskog sustava u većini slučajeva ne uzrokuju rušenje odvojenih virtualnih strojeva na istom fizičkom stroju. Kontejnerska tehnologija se može smatrati poboljšanom tehnologijom virtualnog stroja. Poboljšana je u mnogim aspektima od kojih je vjerojatno najvažniji izbacivanje nepotrebnih komponenti virtualnog stroja. Ovime je kontejnerska tehnologija postala relativno značajno brža i sposobna omogućiti značajno bolje performanse aplikacija koje se njenim posredstvom izvršavaju. Docker je vjerojatno najpopularnija implementacija kontejnerske tehnologije, oslanja se na funkcionalnosti Linuxovog *kernela* i koristi izolaciju resursa. Učinkovitost Dockera, ali i kontejnera općenito proizlazi iz činjenice da za razliku od virtualnog stroja ne zahtijeva cijeli virtualni operacijski sustav koji ima virtualni hardver. Također, skaliranje kontejnera je učinkovitije jer resursi ne moraju biti alocirani unaprijed već kao i svaka druga aplikacija uzimaju resurse od operacijskog sustava kada ih trebaju i otpuštaju ih kada više nisu potrebni.

3.7. Gradle

Gradle je alat za automatizaciju izgradnje aplikacija (engl. *build automation tool*) otvorenog koda. Kao što mu i samo ime kaže, Gradle omogućuje automatiziranje svih postupaka u procesu izgradnje aplikacije, koje bi u suprotnom morali biti odrađeni ručno.[13] Način na koji je automatizacija ostvarena je pridržavanje konvencijama. Korištenjem uspostavljenih konvencija Gradle je u mogućnosti bez dodatne konfiguracije kompajlirati i pokrenuti kôd, generirati dokumentaciju i pokretati automatizirane testove. Uz navedeno, glavna funkcionalnost Gradlea i sličnih alata je upravljanje ovisnim datotekama (engl.

[13] https://docs.gradle.org/current/userguide/what_is_gradle.html

dependencies) u procesu izrade aplikacija.[14] U konfiguraciji koju Gradle koristi moguće je navesti točne verzije ovisnih datoteka, a Gradle će se pobrinuti o njihovu dohvaćanju i spremanju na *classpath* kako bi bile dostupne aplikaciji. Također, zbog pridržavanja konvencijama najčešće nije potrebno izričito navesti lokaciju s koje je potrebno dohvatiti ovisnu datoteku, Gradle zna upravljati repozitorijima, dok istovremeno omogućuje dohvaćanje s onih koji su izričito navedeni. Treba spomenuti i kako je programski jezik posredstvom kojega se programira upravljanje Gradleom programski jezik Groovy. Osim činjenice da za izvršavanje koristi JVM kao i programski jezik Java, to je punokrvni programski jezik kojim je moguće značajno lakše upravljanje tokom, nego što je to bilo kod prijašnjih rješenja za automatizaciju izgradnje aplikacija kao što je Maven, koji je se oslanjao na deklarativno opisivanje procesa izgradnje aplikacija XML-om.

3.8. Angular

Angular je programski okvir (engl. *web application framework*) otvorenog koda, temeljen na programskom jeziku TypeScript.[15] Potreba postojanja programskih okvira za izradu web korisničkih sučelja je nastajala skupa s porastom složenosti tih sučelja. Složenost web stranica, odnosno korisničkog sučelja dovela je do potrebe za strukturiranim i ponovljivim testiranjem kôda koji ih izgrađuje, kao i potrebu za snažnijom organizacijom istih. Angular, kao i većina sličnih programskih okvira ima ugrađena očekivanja i konvencije kako bi web sučelje trebalo biti izgrađeno (engl. *opinionated*). Inicijalni poticaj razvoju i većinu održavanja ranih verzija Angulara su odrađivali razvojni inženjeri tvrtke Google[16] čime je postao jedan od najpopularnijih programskih okvira u svojoj domeni. Skup inženjera koji je napravio prvi razvoj je na umu imao poznati koncept imena „Model-View-Controller“. Programski okvir je napisan u čistom JavaScript programskom jeziku i ponajprije je namijenjen razdvajanju aplikacijske logike od upravljanja DOM modelom. Uveden je koncept vezanja podataka (engl. *data binding*) koji je uveo automatsko osvježavanje pogleda (engl. *view*) svaki puta kada bi se model, odnosno podaci, osvježili. Ono što je zanimljivo je da vrijedi i obrnuto. Uveden je i sasvim novi koncept direktiva (engl. *directives*) koji je omogućio korištenje vlastitih HTML oznaka (engl. *tags*). Još neke od prednosti Angular web

[14] https://stackoverflow.com/questions/20787986/what-is-the-purpose-of-gradle

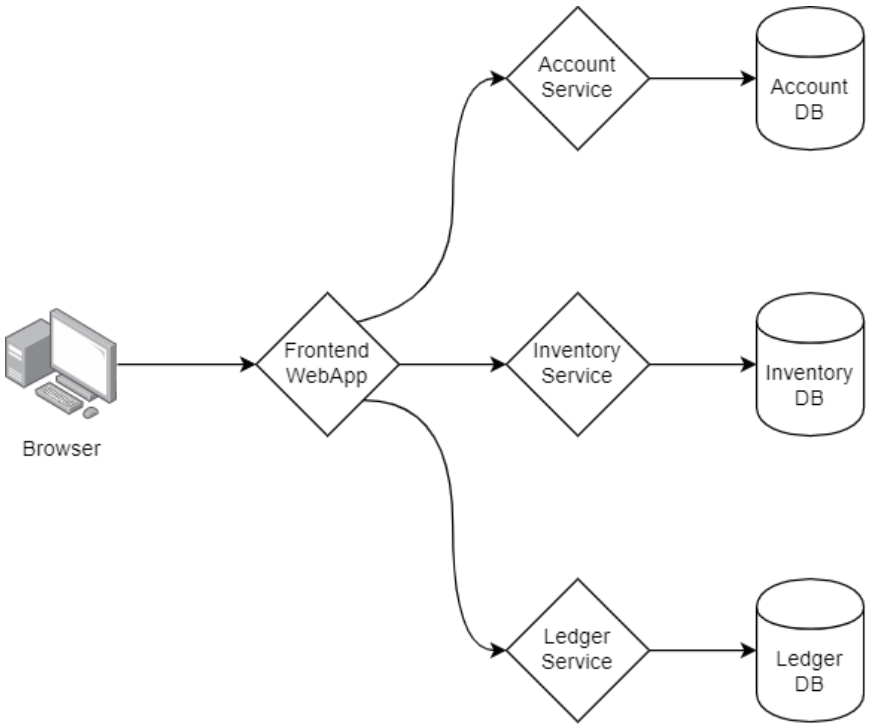
[15] https://en.wikipedia.org/wiki/Angular_(web_framework)

[16] https://www.sitepoint.com/angular-introduction/

programskog okvira su: oblikovni obrasci su sastavni dio programskog okvira, cijeli programski okvir je „samo“ JavaScript, ali poboljšan, mnogi alati kao što su direktive su sastavni dio programskog okvira te je programski okvir potpuno prilagođen razvoju mobilnih kao i desktop aplikacija.

4. Arhitektura Java web aplikacije

Java web aplikacija za upravljanje knjižnicom od koje se praktični dio ovoga rada sastoji prati troslojnu arhitekturu koja je dobro utvrđeni standard pri izradi poslovnih aplikacija, neovisno o tome radi li se o monolitnoj ili mikroservisnoj aplikaciji. Tri sloja troslojne arhitekture su: sloj pristupa podacima, sloj poslovne logike ili aplikacijski sloj i prezentacijski sloj. To je arhitektura u kojoj su čimbenici klijent i server te u kojoj su sva poslovno-procesna logika, pristup podacima i korisničko sučelje potpuno odvojeni. To je ujedno arhitektura i oblikovni obrazac (engl. *design pattern*). Osnovna prednost troslojne arhitekture je mogućnost nadogradnje ili zamjene bilo kojeg od tri sloja neovisno o ostalim slojevima. Arhitektura izrađene aplikacije prikazana je na slici (Slika 4.1).



Slika 4.1. Arhitektura Java web aplikacije za upravljanje knjižnicom

Osobitost izrađene aplikacije je da prati oblikovni obrazac koji je specifičan za mikroservise, a to je da svaki mikroservis ima svoju bazu podataka (engl. *database per service pattern*), što je novost u odnosu na monolitne aplikacije. Kako bi korišteni obrazac i njegovo korištenje bili jasni, potrebno je krenuti od zahtjeva koji prethode njegovu korištenju, a to su:

- servisi moraju biti labavo povezani kako bi ih bilo jednostavnije razviti, postaviti i skalirati
- neke poslovne transakcije moraju dohvatiti podatke čije su vlasnici različiti servisi, a neki upiti moraju spajati podatke kojima su vlasnici različiti servisi
- servisne baze podataka je ponekada potrebno replicirati zbog mogućnosti lakšeg skaliranja[17]

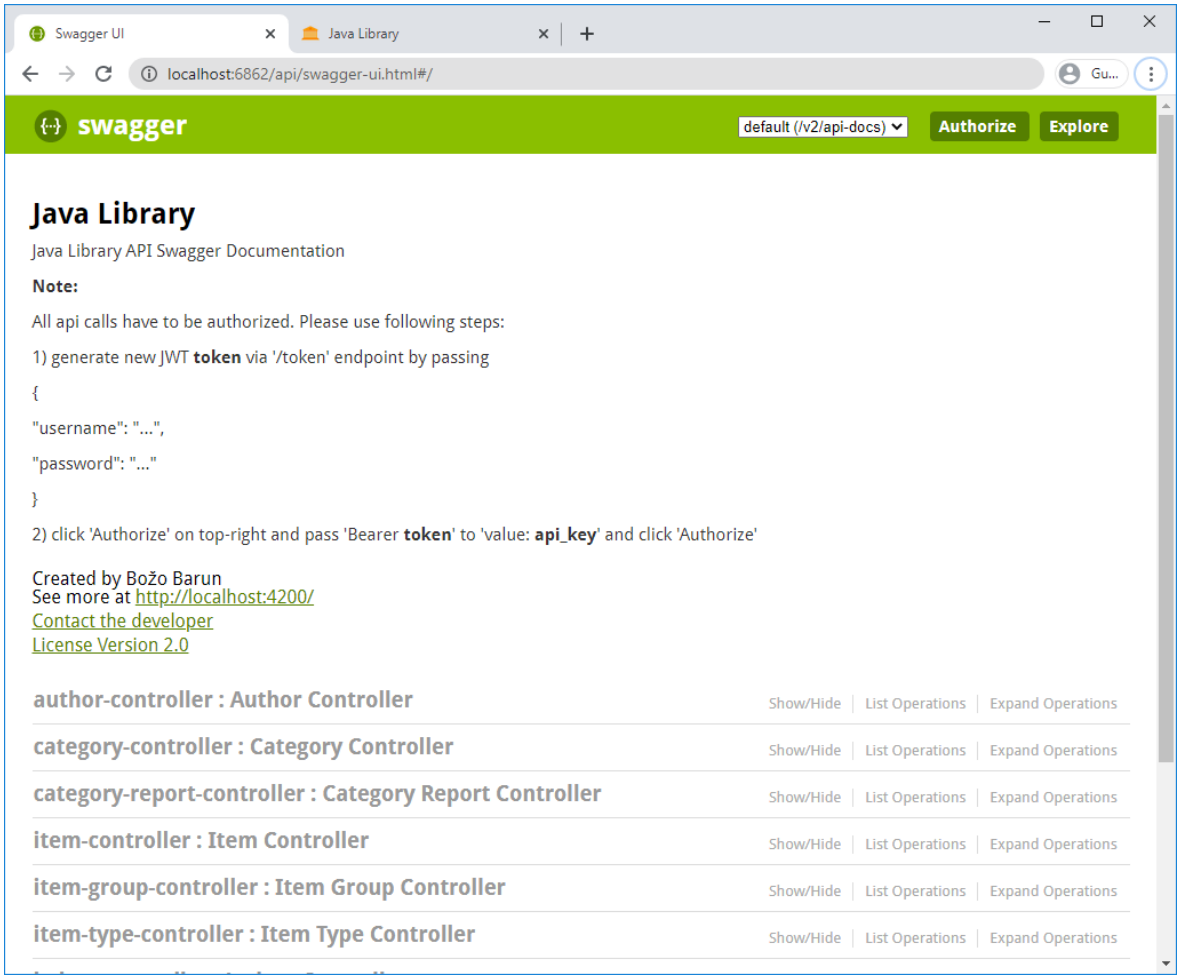
Odgovor na navedene zahtjeve daje upravo oblikovni obrazac jedne baze podataka po servisu. Transakcije servisa koriste samo vlastitu bazu podataka, svi ostali podaci dostupni su samo putem API-ja. Kako bi se zadržao navedeni stupanj privatnosti, moguće je koristiti različite pristupe poput privatnih tablica za servise, sheme dostupne samo jednom servisu ili čak omogućiti svakom servisu korištenje svojeg vlastitog poslužitelja baze podataka. Navedeni pristup ima i svoje izazove od kojih su neki:

- poslovne transakcije koje se protežu kroz više servisa su vrlo teško izvedive, raspodijeljene transakcije je poželjno izbjegavati zbog CAP teorema
- pisanje upita koji spajaju podatke iz različitih baza podataka je često izazovno
- sva dodatna složenost koja dolazi s višestrukim (moguće heterogenim) poslužiteljima i instancama baza podataka

[17] https://microservices.io/patterns/data/database-per-service.html

5. Funkcionalnosti aplikacije

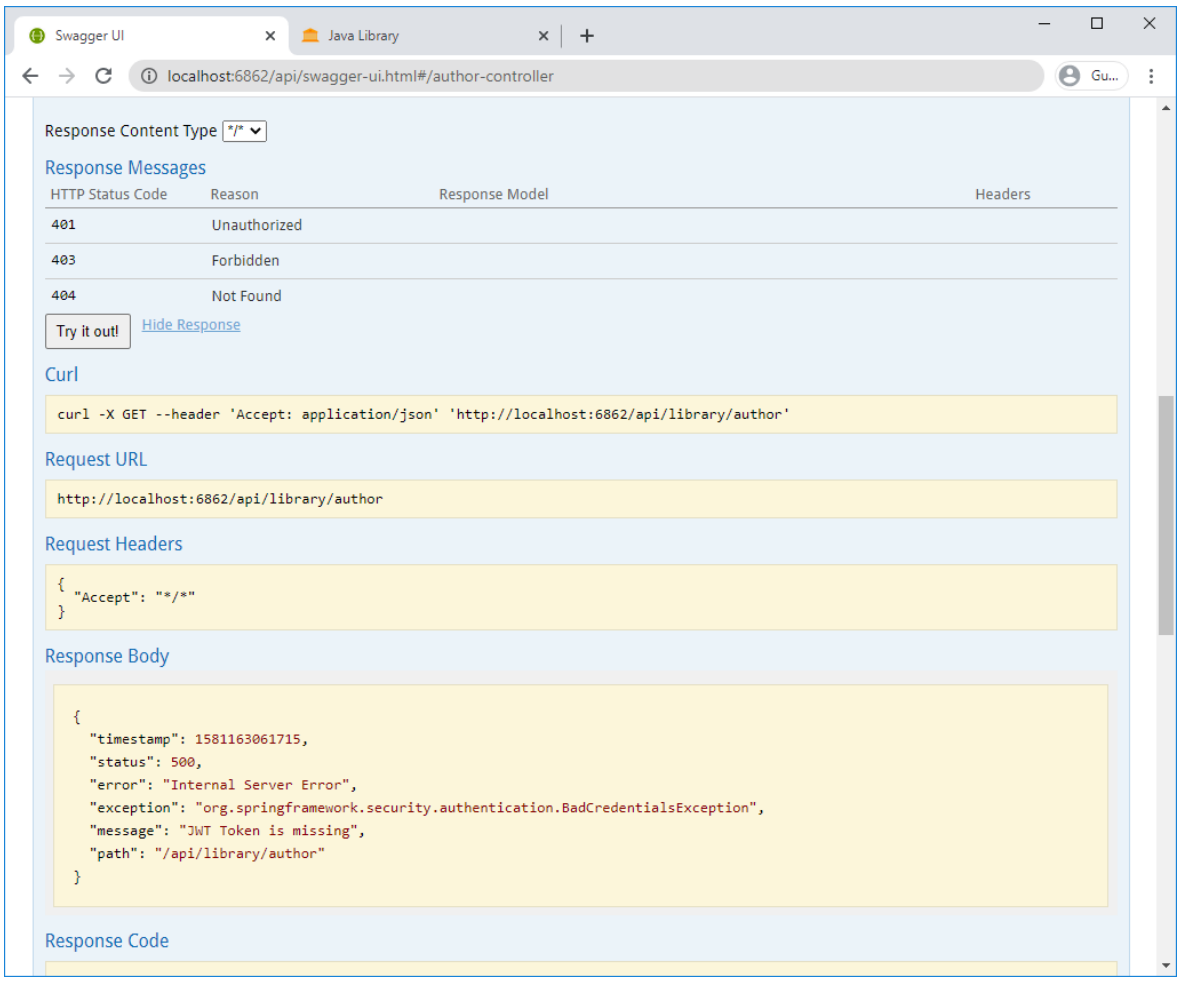
Korištenjem oblikovnog obrasca *backend for frontend* postignuta je potpuna izolacija *backenda* aplikacije za upravljanje knjižnicom napisanog u programskom jeziku Java od *frontenda* napisanog u programskom jeziku Angular. Navedeni obrazac omogućuje jednostavnu izradu *frontenda* namijenjenog korištenju na drugačijoj vrsti uređaja nego što je to ovdje slučaj (npr. mobilnim uređajima). Posljedica korištenja navedenog pristupa je i mogućnost ograničenog korištenja *backend* dijela aplikacije u svrhu testiranja ili upoznavanja s funkcionalnostima bez da je *frontend* dio aplikacije uopće pokrenut. U aplikaciji je to omogućeno korištenjem Swagger UI datoteke. Na ovaj način omogućena je vizualizacija i interakcija s API-jima bez poznavanja njihove implementacije u kôdu. Naime, način pozivanja API resursa je automatski generiran, dokumentiran i vizualiziran. U slučaju aplikacije iz ovog rada dovoljno je otići na adresu http://localhost:6862/api/swagger-ui.html kako bi pristupili Swagger UI sučelju kao što je prikazano na slici (Slika 5.1).



Slika 5.1. Prikaz Swagger UI sučelja s popisom kontrolera koji pripadaju aplikaciji

5.1. Sigurnosne funkcionalnosti

Sigurnosne funkcionalnosti aplikacije za upravljanje knjižnicom svakako predstavljaju najsloženiji mehanizam i kognitivno najkompleksniji dio aplikacije. Složenosti pridonosi i odluka da se korisničke šifre u bazi podataka ne spremaju u tekstualnom obliku, nego kao *hashevi* kao i odluka o korištenju JWT tehnologije umjesto slanja korisničkih autentifikacijskih i autorizacijskih podataka u nezaštićenom obliku. Funkcionalnost je postignuta korištenjem Spring Security *libraryja* na *backendu* i angular-jwt *libraryja*, između ostalog, na *frontendu*. Ilustracija funkcioniranja sigurnosnog mehanizma na *backendu* je jednostavan pokušaj dohvata svih autora putem pripadajućeg kontrolera bez prethodne autorizacije pri čemu se prikazuje rezultat prikazan na slici (Slika 5.2).



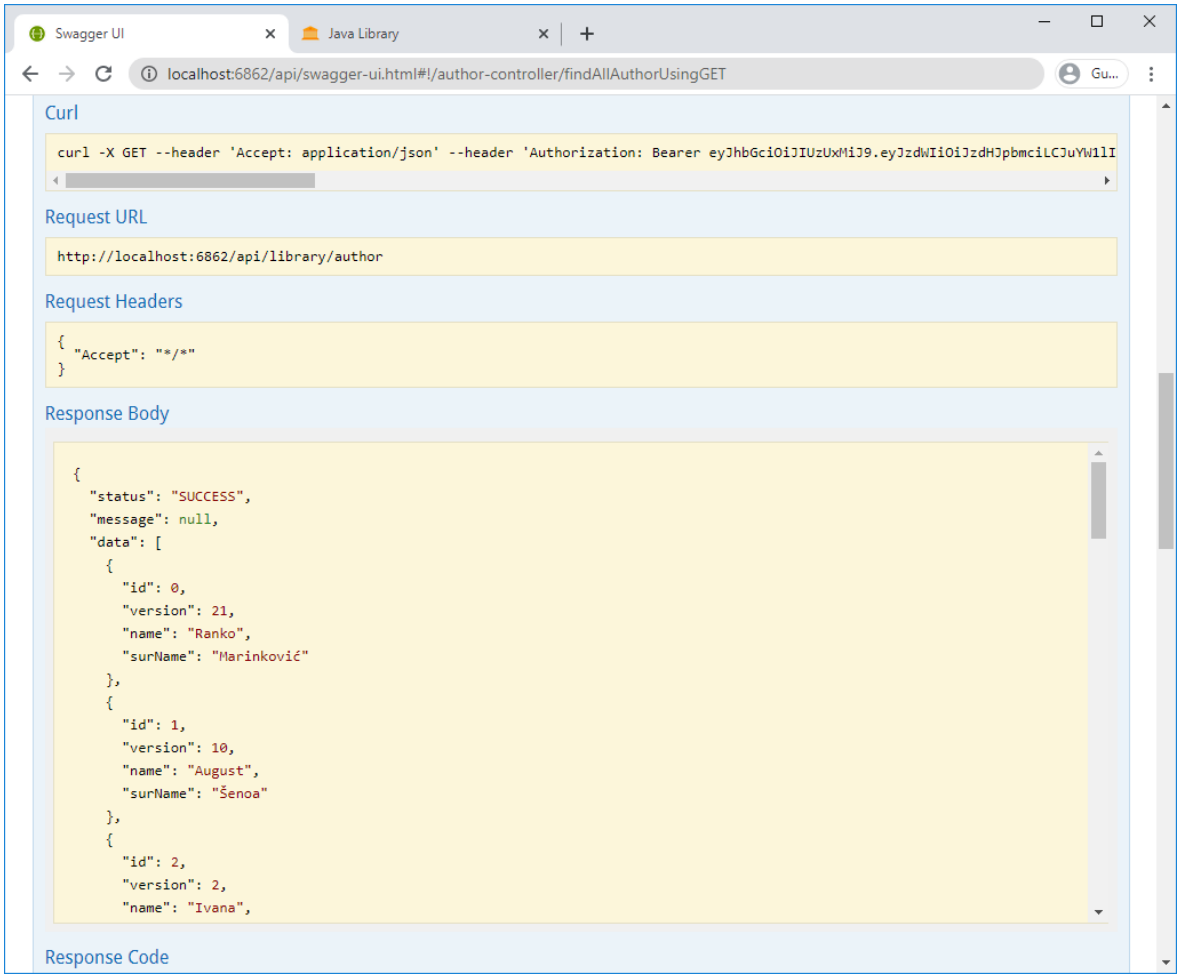
Slika 5.2. Prikaz neuspješnog dohvata svih autora zbog nedostatka JWT tokena kao posljedice neautoriziranosti korisnika

Način dohvaćanja resursa poslovne logike bez korištenja *frontend* dijela aplikacije je slanje zahtjeva prikazanog u kodu (Kôd 5.1) pri čemu aplikacija vrati JWT token kojim je se

moguće autorizirati na Swagger sučelju i dobiti tražene resurse kao što je prikazano na slici (Slika 5.3).

```
 curl -X POST --header 'Content-Type: application/json' --
header 'Accept: text/plain' -d '{ \
   "id": 0, \
   "name": "bbarun", \
   "password": "test", \
…
 }' 'http://localhost:6862/api/token'
```

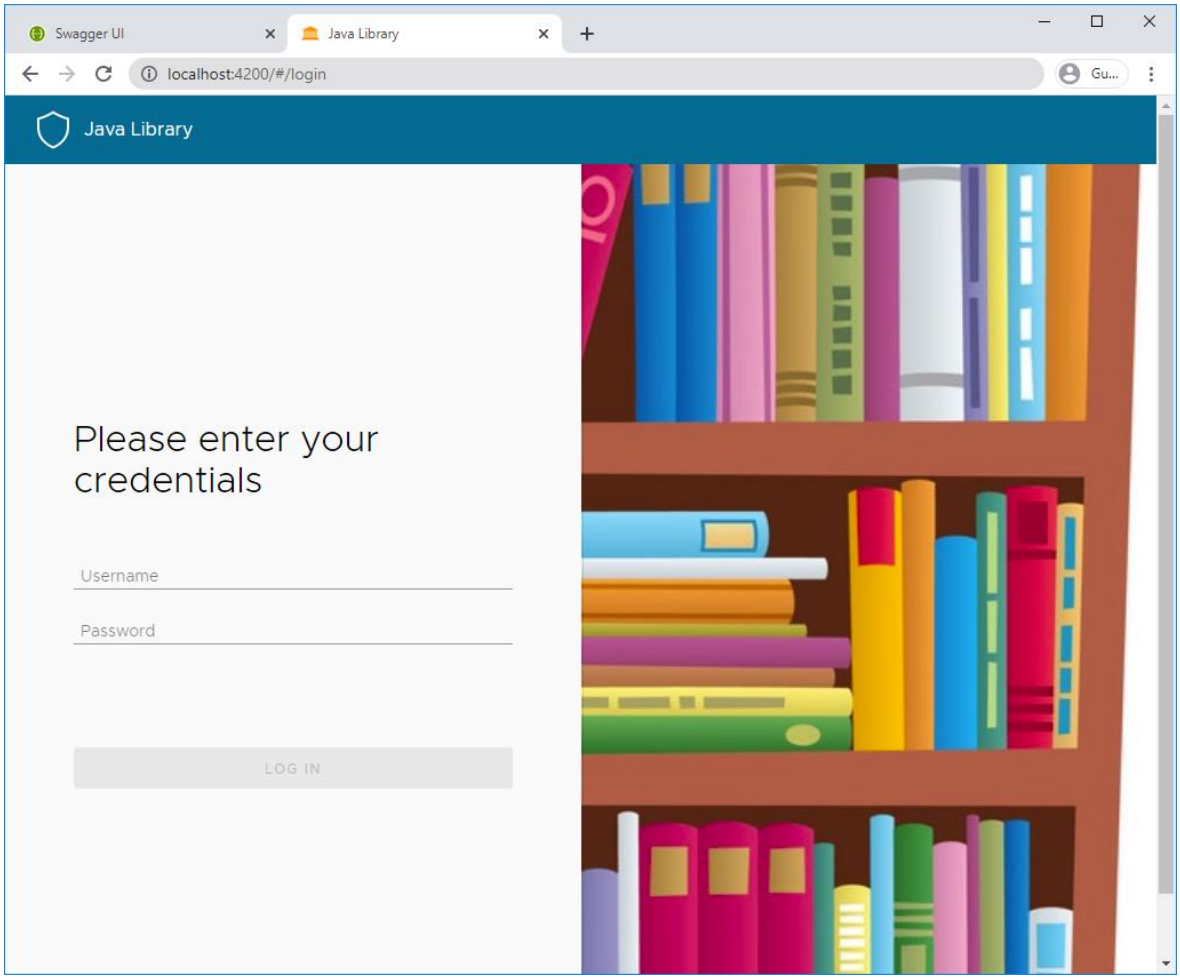
Kôd 5.1. Skraćeni POST zahtjev prema resursu koji vraća JWT token u slučaju da je korisnik predao ispravne kredencijale.



Slika 5.3. Prikaz uspješnog dohvata svih autora nakon predavanja JWT tokena koji jamči autentikaciju i autorizaciju korisnika

Korisničko sučelje aplikacije, jednako kao i njen *backend* dio zahtijevaju poznatog korisnika koji može imati uloge klijenta, knjižničara i administratora. Zbog toga je prva stranica aplikacije *login* forma gdje se predaju korisničko ime i šifra. Osim malenog broja stranica s

nefunkcionalnim svojstvima, nije moguće pristupiti nijednoj stranici aplikacije bez prethodne prijave u aplikaciju. Početna stranica aplikacije prikazana je na slici (Slika 5.4.).



Slika 5.4. Početna stranica aplikacije i ujedno jedina vidljiva bez prijave

Kontrola prikaza pojedinih komponenti u ovisnosti o korisničkim ulogama osigurana je implementacijom objekta u Angularu koji se naziva direktiva (engl. *directive*). Direktive se koriste kao atributi elemenata te je putem njih omogućen prikaz pojedinog elementa aplikacije na način da je omogućen prikaz komponente ukoliko korisnik ima odgovarajuću ulogu kao što je prikazano u kôdu (Kôd 5.2.).

```
@Directive({
  selector: '[appPermitted]'
})
export class PermittedDirective {

  constructor(private templateRef: TemplateRef<any>, private
viewContainer: ViewContainerRef, private loginService:
LoginService) {
  }
```

```
  @Input() set appPermitted(permissionString: string) {
    const permissionArray = permissionString.split(',');
    const allowed = this.loginService.user.roles.some(role =>
{
      return
permissionArray.includes(role.name.toUpperCase());
    });
    if (allowed) {
     this.viewContainer.createEmbeddedView(this.templateRef);
    }
  }
}
```

Kôd 5.2. Direktiva koja dinamički utječe na *layout* komponentu ovisno o korisničkoj ulozi

5.2. Korisničke funkcionalnosti

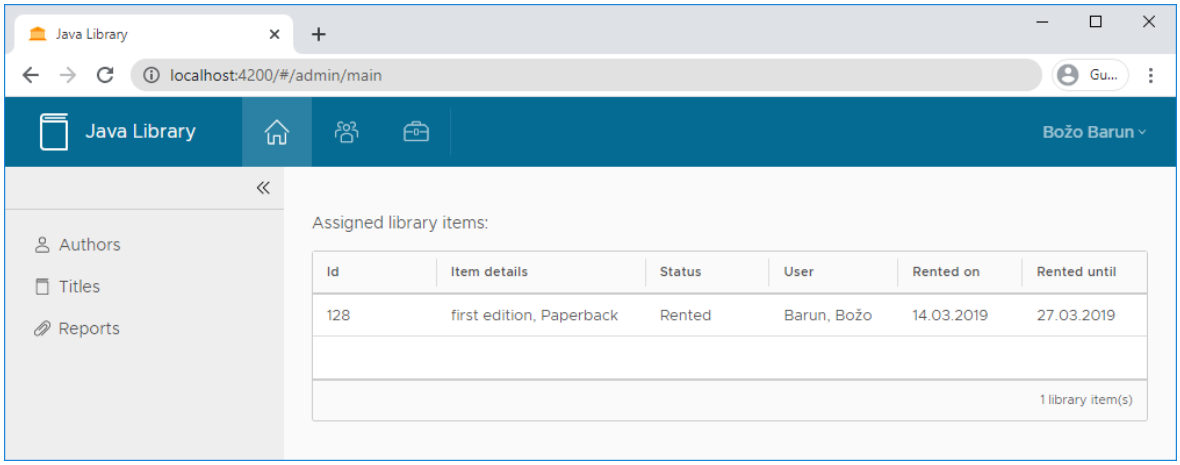
Svaki korisnik aplikacije mora imati barem jednu od tri predviđene uloge (engl. *roles*). To su administrator, knjižničar ili klijent. Korisnikova uloga definira skup funkcionalnosti koje korisnik može obavljati. Administrator ima sve funkcionalnosti aplikacije, knjižničar ima podskup funkcionalnosti iz skupa koje ima administrator, dok klijent jedino ima pravo pregleda posuđenih vlastitih naslova. Aplikacija ima dva navigacijska elementa koja također ovise o ulozi korisnika. U slučaju administratora prva stavka vertikalne navigacije omogućuje CRUD operacije nad autorima, druga stavka omogućuje CRUD operacije nad naslovima, ali i pri uređivanju naslova omogućuje upravljanje registrom posudbi. CRUD operacije omogućene se korištenjem `JpaRepository` sučelja koje posjeduje velik broj predefiniranih metoda što izvedbu navedenih operacija čini izrazito jednostavnom kao što je prikazano u kôdu (Kôd 5.3.) gdje je vidljivo da je nužna samo definicija dviju specifičnih metoda dok je metoda `findAll()` dio sučelja JpaRepository kao što je vidljivo u isječku (Kôd 5.4.). Prva stavka horizontalne navigacije prikazuje osnovni prikaz svakog korisnika, neovisno o njegovoj ulozi i predstavlja registar korisnikovih posudbi. Druga stavka horizontalne navigacije dostupna je samo administratoru i omogućuje CRUD funkcionalnosti nad entitetom korisnika što uključuje dodjeljivanje korisnikovih uloga, dok posljednja stavka horizontalne aplikacije omogućuje uređivanje svih entitetskih šifrarnika. Izgled glavne stranice korisnika koji je administrator prikazan je na slici (Slika 5.5).

```
@Repository
public interface AuthorRepository extends
JpaRepository<Author, Integer> {
    List<Author> findBySurName(String surname);
    List<Author> findByNameAndSurName(String name, String
surName);
}
```

Kôd 5.3. Prikaz sučelja AuthorRepository s definiranim metodama

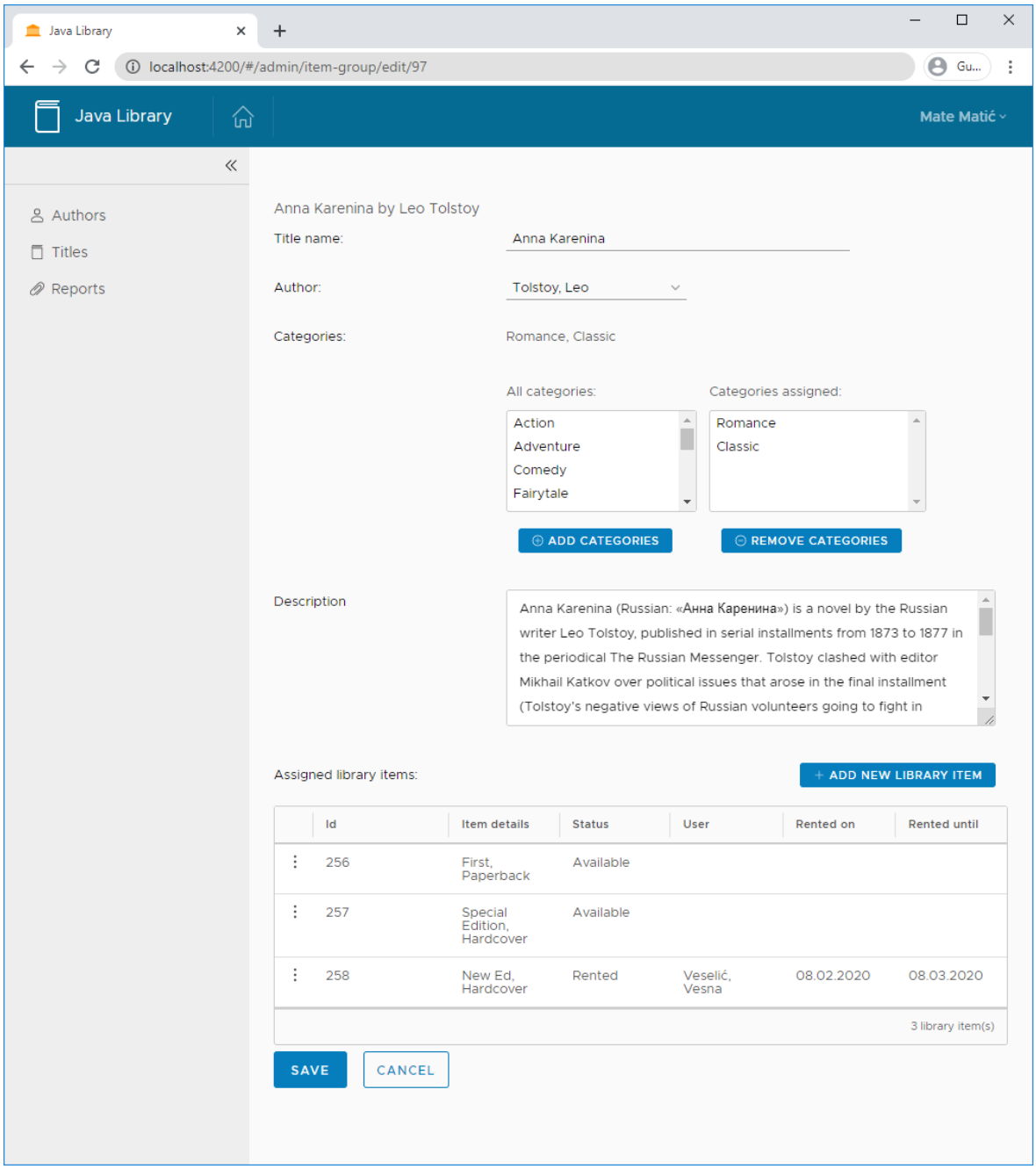
```
@GetMapping
public ApiResponse<List<Author>> findAllAuthor() {
    return ApiResponse.success(authorRepository.findAll());
}
```

Kôd 5.4. Isječak iz kontrolera `AuthorController` s korištenom `findAll` metodom

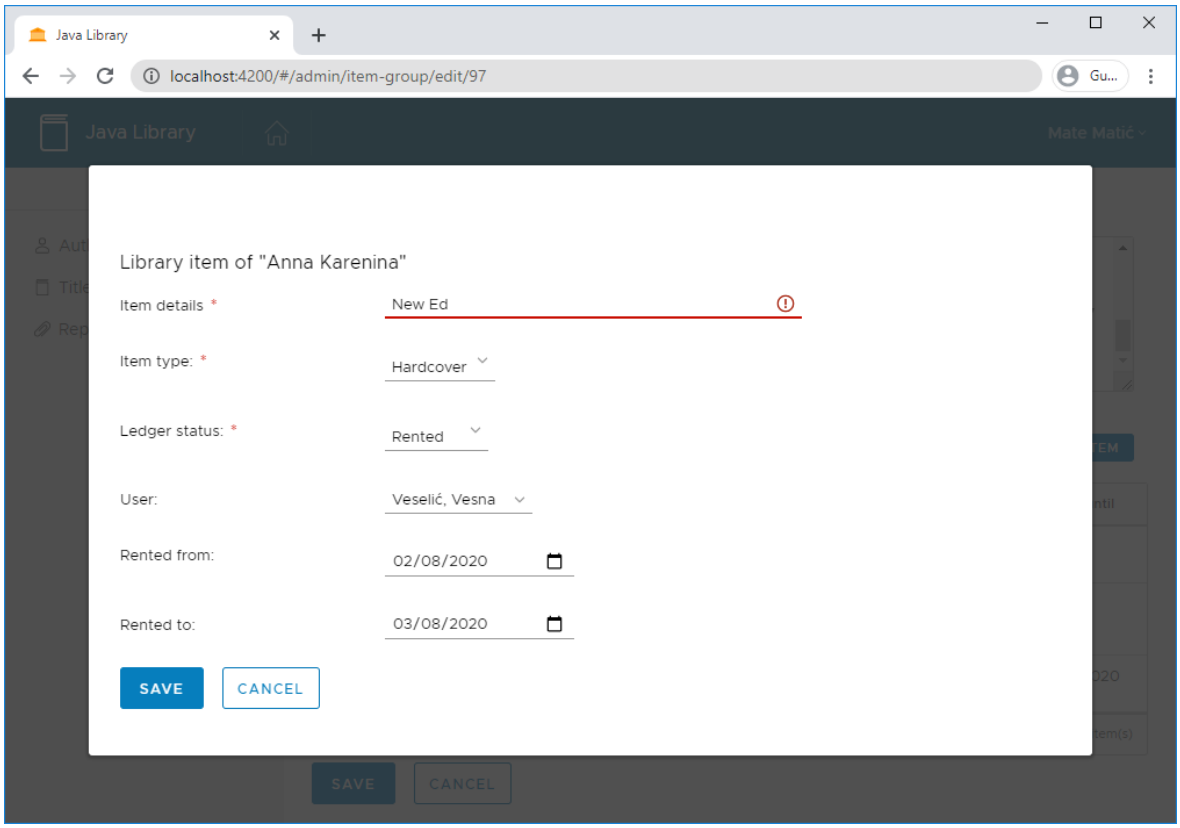


Slika 5.5. Prikaz glavne stranice korisnika koji je administrator s jednom posuđenom knjigom

Srž funkcionalnosti aplikacije je upravljanje registrom posudbi koje se obavlja upravljanjem entitetom naslova u prvom te uređivanjem samog naslova u drugom koraku. Na istom ekranu moguće je urediti atribute naslova, dodati novu instancu naslova kao i urediti postojeće. Upravo uređivanje postojeće instance naslova je efektivno posudba ili vraćanje same knjige što je prikazano na slici (Slika 5.6). Posudba ili vraćanje knjige je promjena statusa instance naslova pri čemu je potrebno dodati ili ukloniti korisnika. Modalni dijalog gdje je radnju moguće obaviti prikazan je na slici (Slika 5.7.).



Slika 5.6. Prikaz mogućnosti upravljanja naslovom od strane korisnika koji je samo knjižničar



Slika 5.7. Modalni dijalog efektivne posudbe ili povrata knjige u knjižnicu

Zaključak

Pri izradi završnog rada opisan je teoretski okvir jedne potpuno nove paradigme izrade web aplikacija koje se naziva *cloud-native*. Teoretski dio nastojao je opisati najvažnije karakteristike takvih aplikacija, a to su definicija i primjeri pojmova: mikroservis, kontejnerizacija i dinamička orkestracija. Teoretski dio potkrijepljen je konkretnom uporabom praktičnog načina izrade *cloud-native* aplikacije na primjeru web aplikacije za upravljanje knjižnicom. Opisane su i upotrijebljene najpopularnije tehnologije u okruženju vezanom za programski jezik Java te je aplikacija zapakirana kao kontejner i isporučena u dinamički orkestrirano okruženje. Arhitektura aplikacije izrađena je tako da poštuje osnovne oblikovne obrasce koje zahtijevaju *cloud-native* aplikacije, te je sama aplikacija izrađena sa svrhom prikazivanja što je više moguće funkcionalnosti koje ilustriraju ispravan način izrade takve aplikacije. Također je prikazan jednostavan način izrade kontejnera i njegovog postavljanja u dinamički orkestrirano okruženje za upravljanje istima.

Radom je pokazano da su osnovni teoretski pojmovi i praktični principi *cloud-native* aplikacija jednostavni i logični te je vidljiv razlog progresije tehnologije u njihovom smjeru. Iz jednostavne aplikacije kao što je ova jasno su vidljive koristi koje tehnologije i principi ovakvih aplikacija donose, dok je sam proces izrade potpuno lišen zamora i omogućuje se usredotočiti na ono što je bitno kako bi se korisnicima isporučila maksimalna poslovna vrijednost bez prepreka koje tehnologija ponekada donosi.

Popis kratica

ANSI	American National Standards Institute	Američki nacionalni institut standarda
API	Asynchronous Transfer Mode	asinkroni način prijenosa
BPEL	Business Process Execution Language	jezik izvršavanja poslovnih procesa
CRUD	Create, Read, Update, Delete	Stavaranje, čitanje, uređivanje, brisanje
DBMS	Database Management System	Sustav upravljanja bazom podataka
DDL	Data Definition Language	Jezik definiranja podataka
DOM	Document Object Model	Model objekata dokumenta
ESB	Enterprise Service Bus	sabirnica servisa poduzeća
HTML	HyperText Markup Language	hipertekstualni označni jezik
HTTP	Integrated Services Digital Network	digitalna mreža integriranih usluga
J2EE	Java 2 Platform, Enterprise Edition	Java 2 platforma, enterprise izdanje
JDBC	Java Database Connectivity	Java povezivost s bazom podataka
JPA	Java Persistence API	Java Persistence API
JSON	JavaScript Object Notation	notacija JavaScript objekata
JVM	Java Virtual Machine	Java virtualni stroj
JWT	JSON Web Token	JSON web token
MVC	Model-View-Controller	model-pogled-upravitelj
ODBC	Open Database Connectivity	Otvorena povezivost s bazom podataka
REST	Representational State Transfer	reprezentativni prijenos stanja
WORA	Write Once, Run Anywhere	napiši jednom, izvršavaj svugdje
XML	EXtensible Markup Language	proširivi označni jezik
YAML	YAML Ain't Markup Language	YAML nije označni jezik

Popis slika

Slika 2.1. Osnovni slijedni dijagram automatiziranog procesa postavljanja mikroservisa ... 8

Slika 4.1. Arhitektura Java web aplikacije za upravljanje knjižnicom 22

Slika 5.1. Prikaz Swagger UI sučelja s popisom kontrolera koji pripadaju aplikaciji 24

Slika 5.2. Prikaz neuspješnog dohvata svih autora zbog nedostatka JWT tokena kao posljedice neautoriziranosti korisnika 25

Slika 5.3. Prikaz uspješnog dohvata svih autora nakon predavanja JWT tokena koji jamči autentikaciju i autorizaciju korisnika 26

Slika 5.4. Početna stranica aplikacije i ujedno jedina vidljiva bez prijave 27

Slika 5.5. Prikaz glavne stranice korisnika koji je administrator s jednom posuđenom knjigom 29

Slika 5.6. Prikaz mogućnosti upravljanja naslovom od strane korisnika koji je samo knjižničar 30

Slika 5.7. Modalni dijalog efektivne posudbe ili povrata knjige u knjižnicu 31

Popis kôdova

Kôd 5.1. Skraćeni POST zahtjev prema resursu koji vraća JWT token u slučaju da je korisnik predao ispravne kredencijale. 26

Kôd 5.2. Direktiva koja dinamički utječe na *layout* komponentu ovisno o korisničkoj ulozi 28

Kôd 5.3. Prikaz sučelja AuthorRepository s definiranim metodama 29

Kôd 5.4. Isječak iz kontrolera AuthorController s korištenom findAll metodom..... 29

Literatura

[1] https://martinfowler.com/articles/microservices.html, objašnjenje pojma mikroservisa, 31.12.2019.

[2] https://stackoverflow.com/questions/10078540/eventual-consistency-in-plain-english, objašnjenje pojma eventualne konzistentnosti, 10.02.2020.

[3] https://itrevolution.com/conways-law/, definicija I objašnjenje Conwayevog zakona 14.02.2020

[4] https://martinfowler.com/articles/microservices/images/basic-pipeline.png, izgled pipelinea kontinuiranog postavljanja 18.10.2019.

[5] https://cloud.google.com/containers/, definicija pojma kontejner 21.10.2019.

[6] https://www.tiobe.com/tiobe-index/, graf popularnosti programskih jezika, 10.10.2019.

[7] https://martinfowler.com/articles/injection.html#InversionOfControl, objašnjenje inverzije kontrole, 25.01.2020.

[8] https://baptiste-wicht.com/posts/2010/08/presentation-usage-h2-database-engine.html, pregled načina rada H2 baze podataka, 18.01.2020.

[9] https://hibernate.org/orm/, definicija Hibernatea, 11.01.2020.

[10] https://lalilуna.com/jpa-hibernate-guide/ch01.html, shema Hibernatea 11.01.2020.

[11] https://stackoverflow.com/questions/28089344/docker-what-is-it-and-what-is-the-purpose, objašnjenje tehnologije Docker, 15.01.2020.

[12] https://docs.gradle.org/current/userguide/what_is_gradle.html, objašnjenje pojma Gradle, 08.01.2020.

[13] https://stackoverflow.com/questions/20787986/what-is-the-purpose-of-gradle, objašnjenje svrhe Gradlea, 08.01.2020

[14] https://en.wikipedia.org/wiki/Angular_(web_framework), prikaz Angulara, 21.10.2019.

[15] https://www.sitepoint.com/angular-introduction/, objašnjenje kako se koristi Angular, 21.10.2019.

[16] https://microservices.io/patterns/data/database-per-service.html, prikaz obrasca jedne baze po servisu, 10.10.2019.