Implementacija Backenda za web shop u mikroservisnoj arhitekturi

Tuđan, Jan

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:333769

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-11-29**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

Jan Tuđan

Implementacija backenda za web shop u mikroservisnoj arhitekturi

Završni rad

Pula, rujan 2021. godine

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

Jan Tuđan

Implementacija backenda za web shop u mikroservisnoj arhitekturi

Završni rad

JMBAG: 0303085614, redoviti student

Studijski smjer: Informatika

Kolegij: Web Aplikacije
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacije znanosti
Znanstvena grana: Informacijski sustavi i informatologija

Mentor: dr. sc . Nikola Tanković

Pula, rujan 2021. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani/a **Jan Tuđan**, ovime izjavljujem da je ovaj seminarski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio seminarskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, _____ godine

IZJAVA

o korištenju autorskog djela

Ja, **Jan Tuđan** dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj preddiplomski rad pod nazivom **Implementacija backenda za web shop u mikroservisnoj arhitekturi** koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu sa Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____

Potpis

_____

Sažetak

U ovom radu obuhvatit će se svrha mikroservisa i njihova prezentacija pomoću alata za kontejnerizaciju. Prikazat će se sam cilj njihove uporabe u izgradnji samih web aplikacija. Usporedba mikroservisne arhitekture u odnosu monolitne, njezine prednosti korištenja, a s druge strane mane njihovog korištenja. Funkcija Dockera je da se svaki servis pojedinačno može staviti u jedan kontejner. Mikroservisi će biti prikupljeni te implementirani u zajedničkom okruženju. Biti će prisutno skaliranje i upravljanje web aplikacije ili drugim riječima web aplikacija će biti zapakirana na lakši način. Svrha samog rada svodi se na testiranje koje prikazuje realnu ulogu mikroservisa udruženih u jednu aplikaciju.

Ključne riječi: web aplikacija, mikroservisi, arhitektura, mehanizam, kontejneri, docker

Sadržaj

1. Uvod 1
2. Mikroservisna arhitektura 2
2.1. Razlika između mikroservisne i monolitne arhitekture 3
3. Višeslojna arhitektura 4
3.1. Prezentacijski sloj 4
3.2. Aplikacijski sloj 5
3.3. Podatkovni sloj 6
4. Korištene tehnologije u procesu izrade mikroservisne arhitekture 6
4.1. Node.js 6
4.2. Docker 7
4.2.1. Docker-compose 7
5. Raspodjela aplikacije 8
5.1. Servis objave proizvoda 9
5.2. Servis komentara 11
5.3. Servis za dodavanje proizvoda u košaricu 11
5.4. Servis za uspješno plaćanje 13
5.5. Servis za autentifikaciju korisnika 14
6. Implementacija kontejnerizirane aplikacije 14
6.1. Kontejnerizacija servisa 15
6.2. Spajanje kontejnera 15
7. Zaključak 17
Github rješenje 18
Popis literature 19
Popis slika 20

1. Uvod

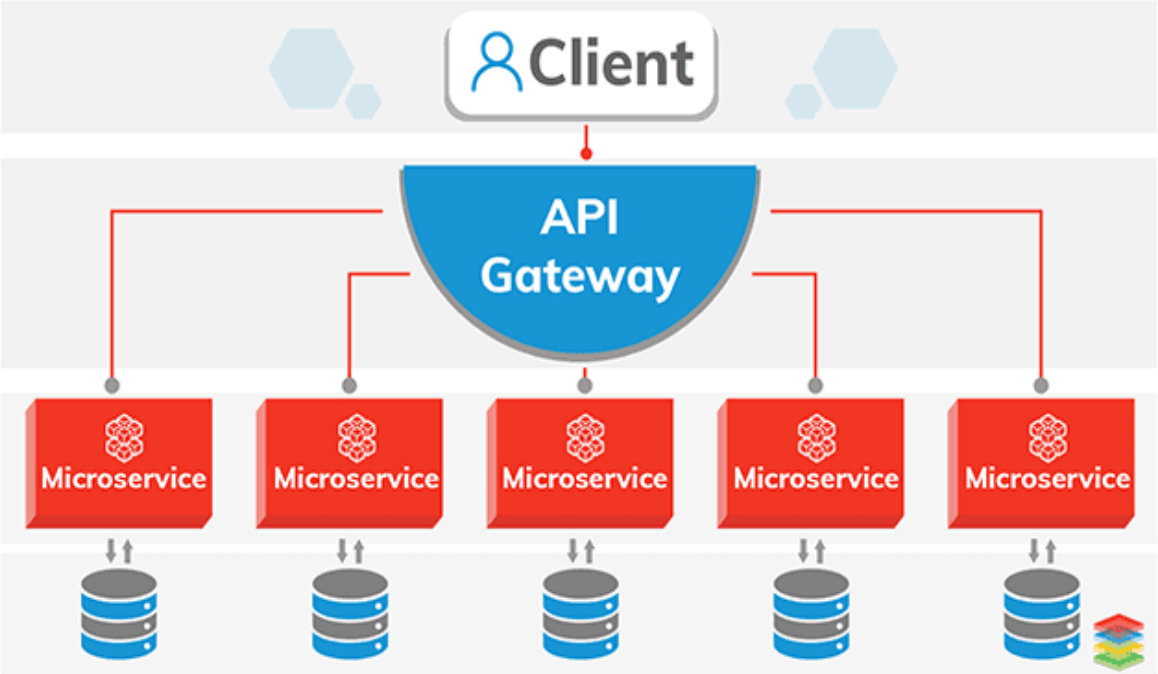
U današnje doba mikroservisi postali su značajni i sve se više upotrebljavaju za izradu kvalitetne web aplikacije. Za razliku od monolitnih aplikacija koje su raspoređene u jednom većem obliku (kao zaseban servis), mikroservisi služe da bi se aplikacija mogla sažeti na više manjih servisa u aplikaciji. Sama prednost mikroservisa jest da svaki od njih može koristiti posebnu bazu podataka što izbjegava opterećenost same baze na serveru. Ujedno je dinamičnija komunikacija samih servisa što znači da ojačavaju samu kvalitetu funkcionalnosti cijele aplikacije. Važno je napomenuti da svaki mikroservis može biti prezentiran pojedinačno, što znači da se može koristiti u nekim drugim web aplikacijama neovisno o mikroservisima koji su također napravljeni, ali služe za malo drugačije potrebe. Sigurnost same web aplikacije može biti raspršena u više servisa.

Od tehnologija izabrao sam Node.js platformu, korišteni operacijski sustav je Windows gdje sam koristio podsustav Linux da bi se omogućilo korištenje Docker softvera. Cilj tog cijelog mehanizma bio je obraditi svaki mikroservis, kontejnerizirati ih u alatu Docker, da se nalaze u skupu virtualnih mašina i razvrstati ih u jedno zajedničko okruženje. To okruženje omogućuje da bi se svi ti servisi mogli ponašati kao jedna aplikacija.

U ovoj temi radi se o izradi backend sučelja za web shop. Koristio sam klijentsko sučelje kao test da bi mi bilo lakše raspolagati s mikroservisima. Na kraju podatke sam testirao u alatu „Postman“ da bi lakše mogao pristupati serverskoj strani. Glavna svrha bila je prikazati svaki mikroservis i spojiti ih međusobno te pretvoriti u jednu aplikaciju. Također značajna je zaštita korisničkih podataka i korisničkoj jedinstvenosti te o ograničenom načinu pristupa svih podataka o narudžbi. Svrha je rascjepkati cijelo backend sučelje u 5 mikroservisa u kojima svaki od njih ima svoju određenu zadaću.

2. Mikroservisna arhitektura

U mikroservisnoj arhitekturi biti će pojašnjeni određeni pojmovi koji se vežu uz tu temu. Isto tako, pojasnit će se uloga mikroservisa da bi moglo biti jasnije zašto ih koristiti u web aplikaciji. Sam pojam „mikroservisna arhitektura“ je princip razvoja aplikacije u obliku manjih i izdvojenih servisa. Ključna komunikacija između servisa biti će prezentirana putem programskih funkcija i jednostavnih mehanizama kao što je HTTP API.



Slika 1. Odnos klijenta preko API zahtjeva u mikroservisu koristeći bazu podataka (Microservices architecture, 2018)

Kao u navedenom, mikroservisi su manji dijelovi aplikacije koji predstavljaju zasebnu radnju na aplikaciji. Da bi se ta radnja mogla izvršiti potrebno je koristiti API koji šalje request prema bazi podataka i čeka njezin odgovor. Ako nije bilo nikakvih neslaganja i ako su podaci podudarni baza je spremna vratiti response u API koji se kasnije prosljeđuje prema klijentu. Zamisao same arhitekture je da bi se jedan mikroservis mogao zasebno bavit različitom bazom podataka što zapravo pokazuje da se jedan mikroservis može zasebno koristiti u nekoj drugoj aplikaciji pored drugih mikroservisa. Naravno, ima slučajeva da svaki mikroservis neće imati različitu bazu, nego će više njih koristiti jednu bazu podataka. Što dolazi do problematike: „Zašto koristiti mikroservise ako oni koriste zajedničku bazu podataka“, no ujedno to i može

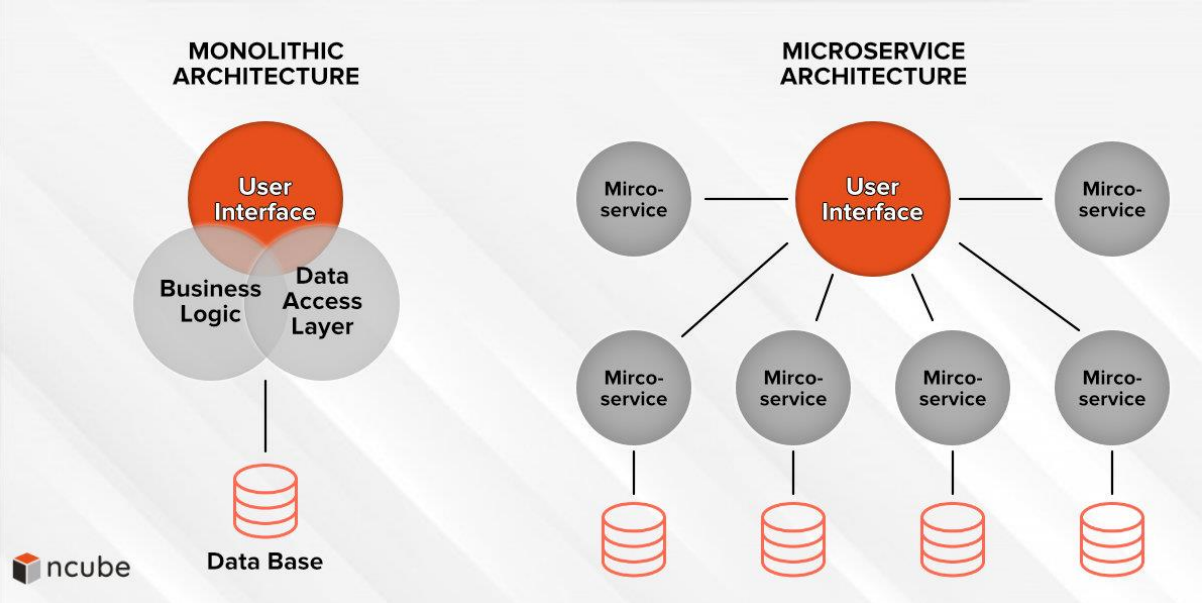
biti slučaj jer neki servisi jednostavno ne mogu povezani sa različitim bazama zbog zajedničkih ID-jeva od svojstava koje oni šalju.

Način djelovanja pojedinačnog mikroservisa:

klijent → servis → API request → baza podataka → API response → servis → klijent

2.1. Razlika između mikroservisne i monolitne arhitekture

Već kao što smo i prije naveli, mikroservisna arhitektura namijenjena je da bi se više servisa spojilo u jednu web aplikaciju, da bi komunikacija bila jasnija, lakša snalažljivost u kodu, također bitno je napomenuti da ta arhitektura rješava problem u slučaju da dolazi do problema u sustavu s nekim svojstvom (na primjer da se desi ažuriranje u programskom paketu i da se vrijednost nekog svojstva izgubi) došlo bi do kolapsa samo tog servisa, dok bi u monolitnom okruženju došlo do kolapsa cijele aplikacije. Pa se u tom slučaju može vidjet prednost u mikroservisnoj arhitekturi. Naravno pri izradi monolitne aplikacije trebalo bi se podrazumijevati da se uključi jednostavnost, tj. nikako se ne bi smjele raditi opširne aplikacije, drugim riječima ako se REST API sastoji od puno request-ova i response-ova, gradi se komplikacija u nepreglednosti i dolazi do zahtjeva integracije, što na kraju i dovodi do izgradnje mikroservisne arhitekture.

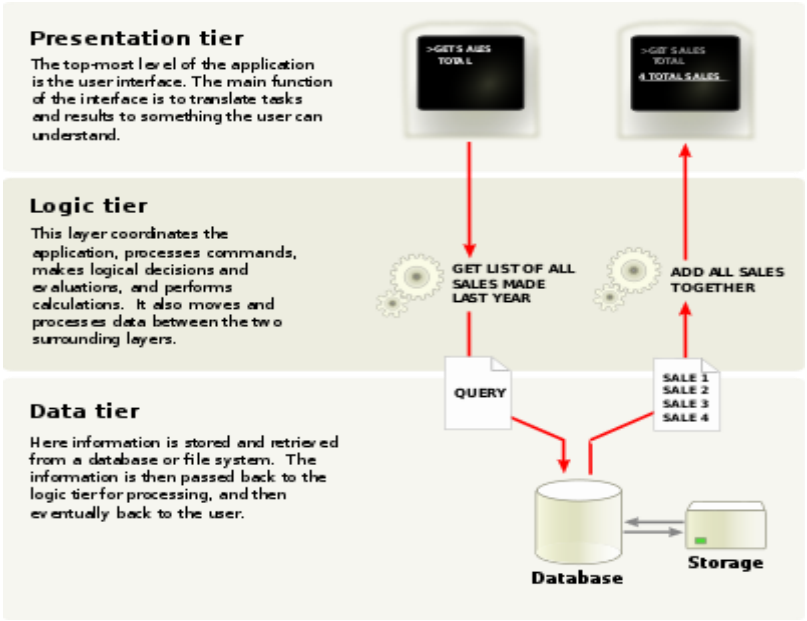


Slika 2. Usporedba monolitne i mikroservisne arhitekture (Microservices vs. Monolithic, 2020)

Na slici možemo vidjeti da monolitna arhitektura ima UI i može imati samo jednu bazu podataka, dok s druge strane mikroservisna arhitektura ima jednu bazu po jednom servisu.

3. Višeslojna arhitektura

U svijetu web aplikacija sama aplikacija se strukturira u više slojeva (često nazivani n-slojevi). N-slojna arhitektura omogućava model gdje programeri mogu stvarati fleksibilne aplikacije za višekratnu uporabu. Najčešće se sastoji od tri sloja pa se zato često naziva i troslojna arhitektura**.** U tu arhitekturu spadaju: prezentacijski, aplikacijski (logički) i podatkovni sloj.



Slika 3. Prikaz troslojne arhitekture (Overview of a three-tier application, 2009)

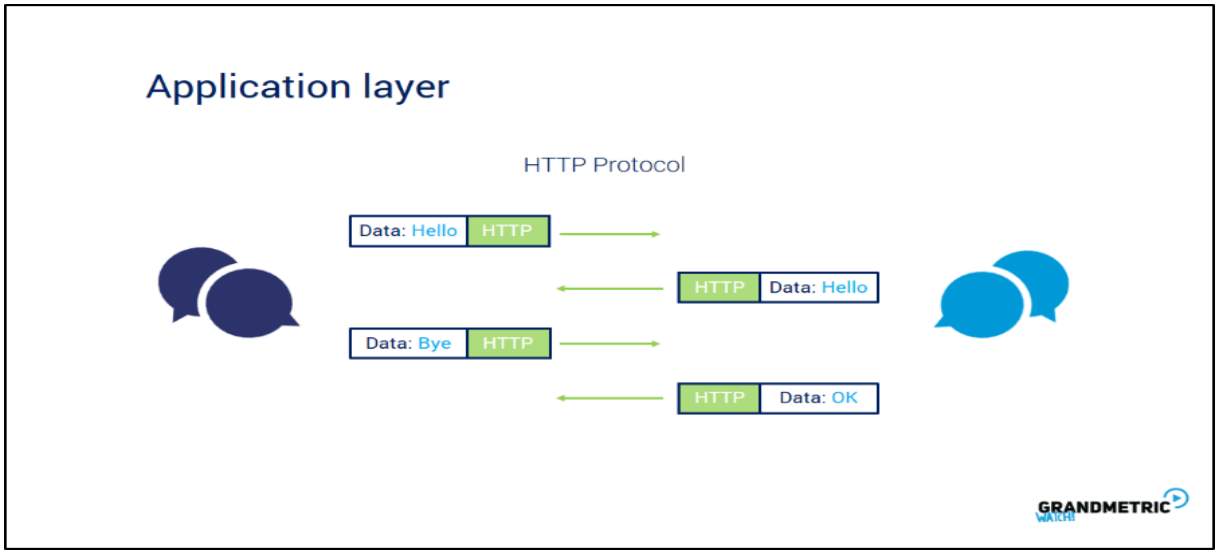
3.1. Prezentacijski sloj

Taj sloj predstavlja korisničko sučelje (eng. UI – User interface) i prikazuje sve korisničke te unesene podatke vidljivima na serveru. Na primjer nakon što se korisnik registrira/prijavi može mu biti prikazano njegovo korisničko ime i drugi podaci koje je unio dok je bio prijavljen, a o takvom prikazivanju unesenih podataka brine se podatkovni sloj koji iste podatke vuče iz same baze podataka. Pošto je vidljivo, ovom sloju je zadatak da komunicira sa ostalim slojevima preko aplikacijskog programskog

sučelja (eng. API – Application program interface). Ovaj sloj izgrađen je pomoću HTML-a (prezentacijskog jezika za izradu stranica), CSS-a (stilski jezik koji se bavi za opis prezentacije dokumenta napisanog pomoću HTML-a) i JavaScript-a (skriptni programski jezik koji se izvršava u web pregledniku na strani korisnika).

3.2. Aplikacijski sloj

Također mnogi ga nazivaju i logičkim slojem, a može se nazivati i poslovno-logičkim slojem. Pod njega spada i srednji softver (eng. Middleware), koji pruža usluge aplikacijama koje nisu dostupne u operacijskom sustavu, a još se naziva i softverskim ljepilom. On omogućava implementaciju komunikacije ulaza i izlaza. Taj sloj zadužen je za upravljanje aktivnosti za koje je sama web aplikacija dužna izvršavati. Bavi se detaljnom obradom ili drugim riječima funkcioniranjem aplikacije. Logički sloj zadužen je za filtriranje podataka koji su izvučeni iz neke tablice u bazi, čak je zadužen i za sve restrikcije. Npr. jedna od restrikcija bila bi da korisnik ima opciju „dodaj u košaricu“, i ako unese 6 proizvoda i pokušava unijeti sedmi, neće mu biti dopušteno, odnosno sustav mu neće dopuštati i javljat će poruku da je u košaricu unesen maksimalan broj proizvoda. Neke tehnologije koje se koriste u ovom sloju su: JavaScript, Java, Python, Perl, ASP . NET, PHP itd… Određeni nivo aplikacije može se hostati na distribuiranim poslužiteljima u oblaku (eng. Cloud), ovisno o tome koliko softverske snage sama aplikacija zahtjeva.



Slika 4. Aplikacijski sloj u ulozi HTTP protokola (Application layer, 2019)

U slučaju web aplikacija koristit će se HTTP (eng. Hyper Text Transfer Protocol) protokol koji je namijenjen za razmjenu sadržaja web stranica. Također, neki od protokola koji se također koriste u višeslojnim arhitekturama aplikacija općenito mogu biti:

- FTP – protokol za preuzimanje i slanje datoteka sa poslužitelja i na poslužitelj
- SMTP – protokol koji služi za slanje elektroničke pošte
- DNS – protokol za doznavanje IP adresa iz imena bilo kojeg računala
- DHCP – protokol koji pruža dinamičko konfiguriranje klijenata na mrežama zasnovanim na IP protokolu
- SSH – protokol koji služi za sigurno povezivanje sa uređajima koji se nalaze na većim distancama te omogućava izvršavanje funkcija i naredbi preko mreže

3.3. Podatkovni sloj

Odnosi se isključivo na bazu podataka koja putem aplikacijskog sloja komunicira s korisničkim sučeljem. Podatkovni sloj se drugim riječima može nazivati razinom skladištenja i može se hostati preko lokalne mašine ili u oblaku. Taj sloj u aplikaciji može pridružiti određene objekte i određenim tablicama može olakšati selekciju polja, da bi se putem aplikacijskog sloja ta polja i stavke lakše dohvatili. Najpoznatiji sustavi baza podataka za upravljanje pristupom u čitanju ili pisanju uključuju MongoDB, MySQL, PostgreSQL i Microsoft SQL Server.

4. Korištene tehnologije u procesu izrade mikroservisne arhitekture

U svijetu mikroservisa postoji širok način korištenja svih mogućih tehnologija za svaku određenu ulogu u tom cijelom procesu. Svaki mikroservis može biti izrađen u svojem programskom okviru, odnosno platformi. To drugim riječima, programski okviri za svaki mikroservis u cijeloj arhitekturi ne moraju biti isti. U nastavku biti će priložene tehnologije koje su dobre za izgradnju mikroservisne arhitekture.

4.1. Node.js

Node.js je platforma otvorenog koda koja u ovom radu služi za izradu aplikacija koje se odnose na serversku stranu. Može se reći da je node.js dio JavaScript okruženja, a ima već gotove pakete koji se mogu dodatno instalirati, te inicijalizirati da se dijelovi unutar tih paketa mogu koristiti i prilagoditi node.js okruženju. Node.js je

isključivo namijenjen za programere koji se misle baviti čistim JavaScriptom da bi mogli pisati serverske skripte. Dio node.js-a su node_modules, to su moduli za instalaciju cijele aplikacije koji sadrže već gotove pakete koji su osnovni dio node.js aplikacije.

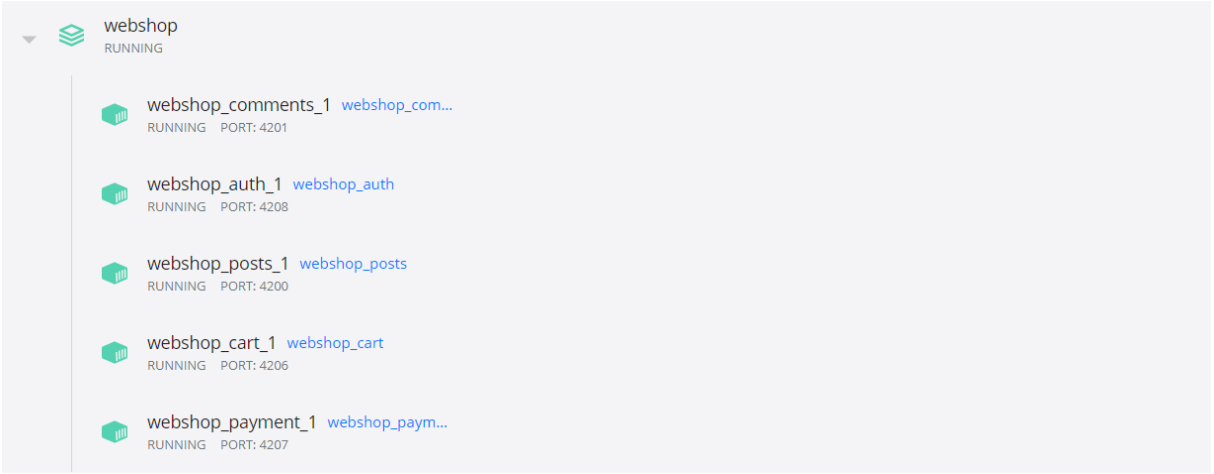
4.2. Docker

Docker je open-source usluga koja automatizira raspoređivanje kontejnera, a i služi za izradu kontejnera. Koristi se u terminalu kako bi mogli manipulirati aplikacijama. Kontejneri služe da bi sama aplikacija, s kojom želimo manipulirati, bila u izoliranom okruženju. U pravilu kada želimo kontejnerizirati aplikaciju prvo je dobijemo u formatu kao Docker „image“. Zatim kada želimo pokrenuti aplikaciju ona postaje kontejner. Nekakve osnovne komande bile bi:

- „docker build <folder file>“ – služi za izgradnju image-a
- „docker run -it -d <image name>“ – pokreće se image koji postaje kontejner
- „docker pull <image name>“ – vuku se postojeći image-i iz Docker repozitorija
- „docker ps -a“ – služi za prikaz svih kontejnera koji su trenutno u funkciji

4.2.1. Docker-compose

Alat koji je omogućen sa strane Docker-a. Koristi YAML datoteke koje služe za održavanje više servisa u jednom okruženju. Docker-compose.yml za razliku od običnog Docker-a omogućuje zajedništvo svih kontejnera koji su napravljeni u „Dockerfile“ datoteci.

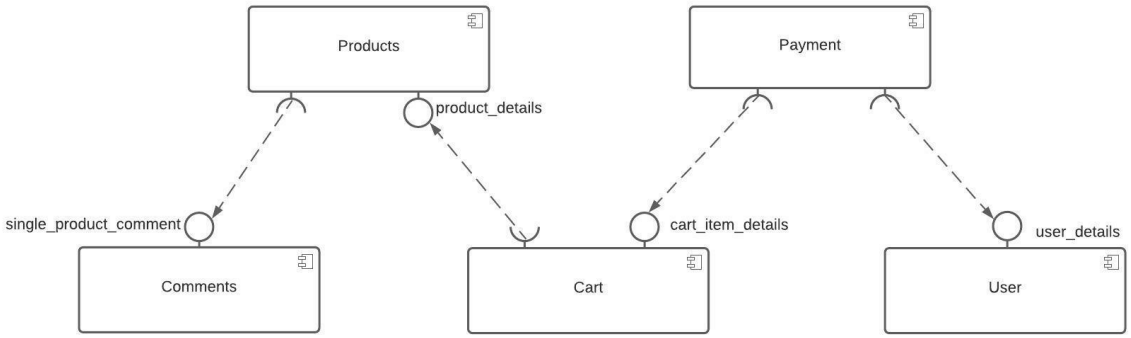


Slika 5. Thingify – primjer više različitih kontejnera u jednom okruženju

5. Raspodjela aplikacije

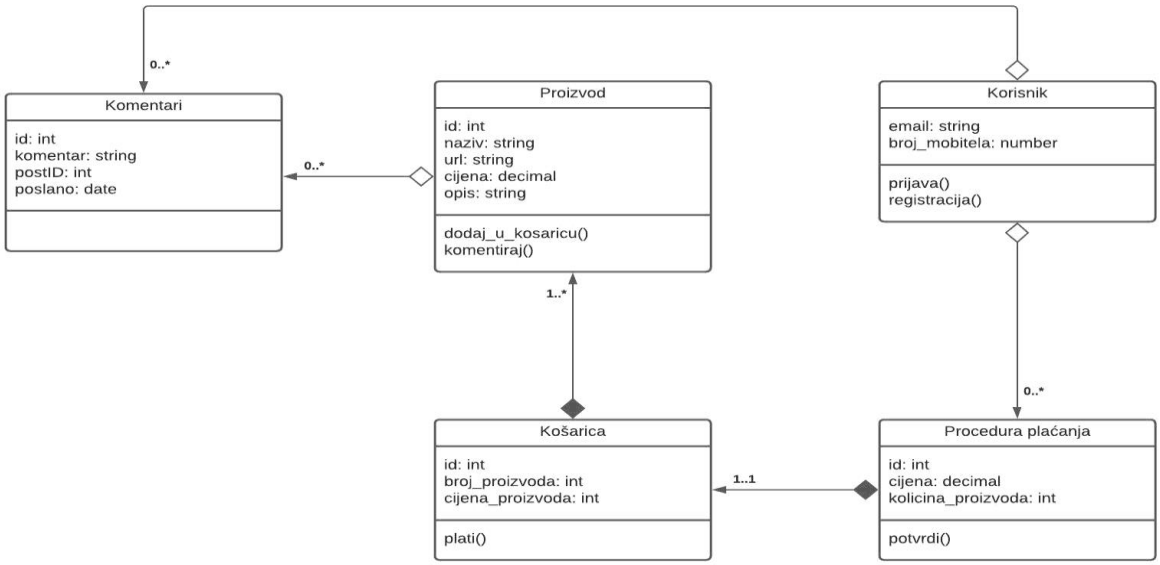
Aplikacija je rascjepkana na pet različitih servisa, s tim da je svaki od njih imao svoju ulogu, a glavna ideja bila je napraviti smislene servise i sinkronu komunikaciju između tih servisa. Glavna poanta bila je pojednostaviti kod, smanjiti preopterećenje API zahtjeva i omogućiti da svaki servis zasebno funkcionira. Da bi cijela arhitektura imala smisla radili smo zahtjeve pomoću „Axios“ paketa kako bi jedan servis imao kontakt s drugim. Svih 5 servisa koristi ukupno tri baze: „Posts“, „Payment“ i „User“, s time objave proizvoda i komentari zajednički dijele jednu bazu podataka, te s druge strane servisi s košaricom i procesom plaćanja zajednički koriste bazu podataka. Uzevši u obzir baze „Posts“ i „Payment“ koje se sastoje od dvije kolekcije i svaka od tih kolekcija odnosi se na poseban servis.

.



Slika 6. UML komponentni dijagram

Na Slici 5. vidljiv je komponentni dijagram koji ujedno i prikazuje servise i njihovu komunikaciju. Proizvodi rade interakciju s komentarima, a njihovu komunikaciju označava komentar o jednom proizvodu. Košarica ide prema proizvodima i uzima detalje o proizvodu. Proces plaćanja radi interakciju sa košaricom i pristupa detaljima košarice, isto tako proces plaćanja ide prema servisu za korisnike te uzima korisničke podatke. Ovim dijagramom može vizualno biti jasnija međusobna komunikacija svih servisa.



Slika 7. UML klasni dijagram

Na Slici 6. moguće je vidjeti klasni dijagram koji pokazuje manipulaciju podataka između svih servisa. Korisnik ima interaktivne funkcije za prijavu i registraciju, on ne ovisi o komentarima kojih može biti 0 ili više. Korisnik ne ovisi o proceduri plaćanja, a veza je 0 ili više procedura plaćanja po jednom korisniku. Proizvod ima funkciju dodavanja proizvoda u košaricu te opciju za komentiranje. Odnos proizvoda prema komentarima je neovisan, a komentara može biti 0 ili više komentara po jednom proizvodu. Košarica ima funkciju za plaćanje. Ona ovisi o proizvodima jer, ako nema proizvoda, onda nema smisla da košarica postoji. Pošto ne bi bilo podataka o proizvodu, u košarici može biti jedan ili više proizvod. Procedura plaćanja ovisna je o košarici jer, ako nema košarice, onda ni procedura plaćanja ne bi mogla postojati. U proceduri plaćanja može biti samo jedna košarica puna proizvoda.

5.1. Servis objave proizvoda

Ovaj servis služi kao katalog svih proizvoda, odnosno prikazane kartice za svaki proizvode te detalje svakog proizvoda u web shopu. Također, pri parametrima na već određenu rutu, omogućeno je da se interakcijom ID-a pristupi na specifičan proizvod da bi se omogućila vidljivost podataka o samo tom proizvodu.

```
app.get ('/productComments/:title', async (req , res) => {

    //Dohvaćamo sve komentare iz comments servisa za jedan post
    try {
    let title = req.params.title;

    const db = await connect();

    const doc = await db.collection("posts").findOne({title: title});

    let comments  = await axios.get("http://localhost:4201/comments");

    const singleProductComment = await comments.data.filter((comment) => comment.postID == doc._id);

    res.json(singleProductComment)
    }

    catch(error){
        console.log(error)
    }

});
```

Slika 8. Sinkrona komunikacija sa servisom za komentare

Na Slici 8. je vidljiv je sinkron API response odnosno metoda „get()“. Traži se proizvod po naslovu u ruti, zatim se radi konekcija s bazom i ubrzo pomoću „findOne()“ metode ulazimo u kolekciju „posts“ tražimo jedan objekt koji ima jednak naslov kao onaj koji je unesen u rutu. Radi se „Axios“ komunikacija i dohvaćaju se svi komentari, ali zato ih umjesto „find()“ (koji bi vratio jedan komentar) filtriramo sa „filter()“, da bi se mogli ispisati svi komentari koji imaju jednak ID kao i jedan proizvod po zadanoj rutini MongoDB-a. Na kraju na samu serversku stranu šaljemo sve komentare koji se odnose na određeni proizvod.


```
[
    {
        "_id": "611c2439fc4796a0e0d512ec",
        "title": "Skytech Prism II Gaming PC Desktop",
        "url": "https://m.media-amazon.com/images/I/91LSF1iZUFL._AC_SL1500_.jpg",
        "price": 1250.49,
        "desc": "✓ AMD Ryzen 9 3900X 12-Core 3.8 GHz (4.6 GHz Max Boost) CPU | 1TB PCIe Gen4 NVME Internal Solid State Drive\n✓ GeFo
    },
    {
        "_id": "611c2574fc4796a0e0d512ed",
        "title": "Apple iPhone 12, 64GB, Black - Fully Unlocked (Renewed)",
        "url": "https://m.media-amazon.com/images/I/71FaAo27evL._AC_SX522_.jpg",
        "price": 709.99,
        "desc": "✓Fully unlocked and compatible with any carrier of choice (e.g. AT&T, T-Mobile, Sprint, Verizon, US-Cellular, Cricke
    },
    {
        "_id": "611c262bfc4796a0e0d512ee",
        "title": "LCD TV, Flat Screen TV, BCL-32A/3216D Widescreen 43inch HD 1080P ",
        "url": "https://m.media-amazon.com/images/I/61eBzQr2gmL._AC_SL1001_.jpg",
        "price": 359.99,
        "desc": "✓Built-in Dolby sound technology, bass ring design, enhanced dialogue, built-in audio will provide high quality viev
    },
    {
```


Slika 9. Ispis svih podataka o proizvodu

5.2. Servis komentara

Servis za komentiranje služi da bi se komentirao određeni proizvod. Korištena je „post“ metoda za davanje zahtjeva prema bazi podataka da bi korisnik mogao unijeti podatke o komentaru. Ako su uspješno poslani podaci, serverska strana šalje poruku za uspješno slanje. Ako podaci nisu poslani na ispravan način, serverska strana obavještava da je došlo do greške. Metoda za odgovor spaja se na bazu podataka, izravno u API pozivu pristupa bazi podataka „posts“ te u kolekciji „comments“ selektira, odnosno prikazuje sve postojeće objekte koji sadržavaju podatke o komentarima. Na taj način šaljemo odgovor na serversku stranu te se prikazuju objekti svih rezultata.

```
app.get("/comments", async (req, res) => {
    await client.connect();
    let db = await client.db("posts");

    let results = await db.collection("comments").find({}).toArray();
    console.log(results);
    res.json(results);

});
```

Slika 10. Dohvat svih komentara

5.3. Servis za dodavanje proizvoda u košaricu

Ovaj servis služi da bi se mogli dohvatiti određeni podaci o proizvodu te zbrajati, ovisno o tome koliko zahtjeva za rutu je uneseno. Također, instaliran je poseban middleware nazvan „express-session“ koji omogućava spremanje podataka u sesiju u sav serverski dio. Dobro je za napomenuti da se ne sprema u „cookies“ nego samo u sesijski ID. Taj proces sa sesijom potreban je da bi se po jednom requestu podaci, koji su upućeni prema spremanju u košarici, mogli spremati po sesijskim ID-jevima.

```
app.post("/add-to-cart/:id", async (req, res) => {
//Dohvaćamo sve proizvode iz posts servisa i umećemo logiku s košaricom
  try {
      const id = req.params.id;

      const { data }  = await axios.get("http://localhost:4200/products");

      const singleProduct = await data.find((product) => product._id === id);

    let cart;
    if (!req.session.cart) {
      req.session.cart = cart = new Cart({});
    } else {

      cart = new Cart(req.session.cart);
    }
    //Spremamo podatke u Session da bi se mogle spremati prijašnje vrijednosti
    req.session.cart = cart;
    cart.addProduct(singleProduct);
    console.log(req.session.cart)
    res.send(cart);

  } catch (error) {
    console.log(error);
  }
});
```

Slika 11. Dodavanje pojedinačnog proizvoda u košaricu

Na rutu se unosi ID proizvoda kojeg želimo dodati u košaricu, parametri se spremaju, a pomoću axios-a dohvaćamo sve proizvode iz servisa „prodcuts“. Od svih dobivenih proizvoda traži se ID proizvoda koji je unesen kao parametar u ruti. Ako nema dodanih podataka o proizvodu u sesiji, u nju se sprema konstruktor „new Cart()“ gdje se inicijaliziraju novi podaci. U suprotnom, spremamo podatke za košaricu u sesiju.

```
module.exports = function Cart(oldCart) {
    this.productItems = oldCart.productItems || {};
    this.totalQty = oldCart.totalQty || 0;
    this.totalPrice = oldCart.totalPrice || 0.00;

    this.addProduct = function(item) {
      let storedItem = this.productItems;
      if (!storedItem.hasOwnProperty("item")) {

        storedItem = this.productItems = {item: item, qty: 1, price: item.price};
        this.totalQty = 1;
        this.totalPrice = item.price;

      } else {

        storedItem = {item: item, qty: storedItem.qty, price: storedItem.price};
        console.log("STORED ITEM: ", storedItem);
        this.productItems = storedItem;
        storedItem.qty++;
        storedItem.price = storedItem.item.price * storedItem.qty;
        this.totalQty++;
        this.totalPrice += storedItem.item.price;

      }
    }
  }
```

Slika 12. Logika spremanja i zbrajanja proizvoda u košaricu

U ovom servisu još se nalaze metode „post()“ i „get()“. Metoda „post()“ služi da bi se trenutno spremljeni podaci o košarici mogli poslati u bazu, a metoda „get()“ dohvaća sve podatke iz baze i selektirani su određeni property-ji iz objekta. Ta metoda osmišljena je za daljnje prosljeđivanje prema drugim servisima.

5.4. Servis za uspješno plaćanje

Služi za plaćanje cijene ukupne količine proizvoda koji su dodani u košaricu. Korištena je međusobna komunikacija sa servisom za košaricu te korisničkim servisom.

```
app.get("/payment/:email", async (req, res) => {

    //Traži se korisnik, ako se podudaraju uneseni podaci u procesu plaćanja moguće je vidjeti sve podatke o plaćanju, koliko je proizvoda i koja je cijena
    let email = req.params.email;
    let db = await connect();
    let userResponse = await axios.get(`http://localhost:4208/allUsers`); //Dobivanje svih registriranih korisnika iz servisa authService

    //Traži se da li email unesenog korisnika postoji
    let filterUser = await userResponse.data.find((user) => user.email == email)
    let userEmail = filterUser.email;
    let userPhoneNumb = filterUser.phoneNumb
    console.log(userEmail, userPhoneNumb)


    let reqData =  await db.collection('payment').findOne({userEmail: userEmail}); //Pretraga da li postoji registrirani korisnik po emailu

    console.log("req data: ",reqData)


    let cartResponse = await axios.get('http://localhost:4206/cart')  //Dobivanje servisa gdje su spremljene sve unesene košarice
    let filterCart = await cartResponse.data.find((cart) => cart.totalQty == reqData.totalQty) //Upit da li se količina podudara s onom koja je u servisu cart već dodana u košaricu

    console.log("Filtrirani cart: ", filterCart)

    let cartTotalQty = filterCart.totalQty;
    let cartTotalPrice = filterCart.totalPrice;

    let Data = {userEmail, userPhoneNumb, cartTotalQty, cartTotalPrice} //Spajanje svih property-ja u objekt Data
    res.send(Data);
});
```

Slika 13. Komunikacija sa 2 servisa i plaćanje po podacima iz baze

Korisnik ima upit za objavu njegovih korisničkih podataka i podataka koliko proizvoda želi i njihovu ukupnu cijenu. Pomoću metode „get()“ na Slici 13., dohvaćaju se podaci svih korisnika i košarice da bi se mogli usporediti upisani podaci koji su spremljeni u bazu. Traže se podaci po unesenom korisničkom emailu i tako se nađe da li korisnik stvarno postoji, traže se svi podaci iz košarice i zatim se pretraga pomoću funkcije „find()“, da bi se moglo vidjeti podudaraju li se podaci u bazi s navedenima. Ako se podudaraju, stvara se objekt u koji se spremaju varijable koje će se prikazati.

5.5. Servis za autentifikaciju korisnika

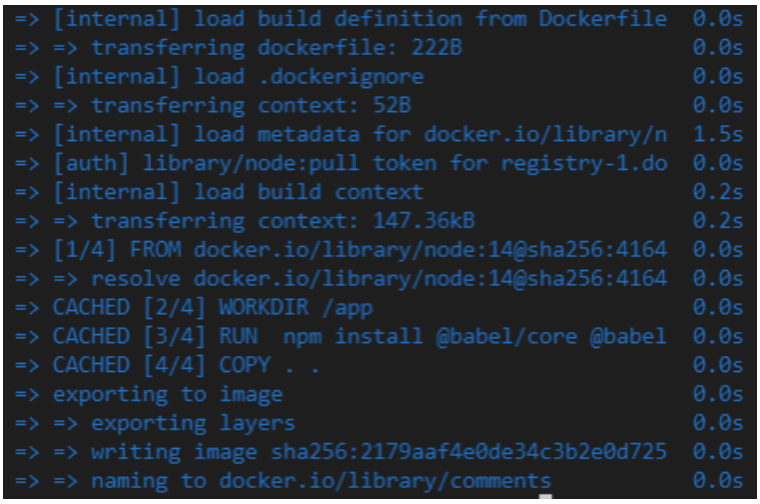
Služi za slanje korisničkih podataka u bazu. Kada se korisnik uspješno registrira, nakon što se prijavi, implementiran je posrednički softver „verify“ koji se koristi JWT tokenom, kako bi se provjerilo da je korisnik uspješno prijavljen, odnosno da li korisnik postoji u bazi podataka. Uspoređuje se upisana korisnička lozinka isto kao i email, pa se ta provjera vrši pomoću funkcije „authenticateUser()“. Ako korisnik postoji, tu nastupa JWT koji pomaže pri stvaranju korisničkog tokena. To se obavlja pomoću pametnog algoritma „bcrypt.hash“, koji raspršuje lozinku u korisniku nepoznate znakove. Tako da, čak i ako se upadne u cijeli serverski dio aplikacije, ne bi se moglo doći do korisničke lozinke. U tom slučaju pomoću svih pametnih algoritama lozinka je zaštićena i na serveru ju ne može znati čak ni onaj tko upravlja bazom. Također, ako postoji korisnik s istim unesenim emailom, drugi korisnik neće biti u mogućnosti registrirati se. U ovom servisu radi se ruta „allUsers“ koja prikazuje sve dosadašnje registrirane korisnike, a ona je namijenjena za prosljeđivanje podataka u drugi servis koji dohvaća tu istu rutu.

6. Implementacija kontejnerizirane aplikacije

Ova aplikacija napravljena je u smislu da se svaki servis može pojedinačno izgraditi kao „image“ i zatim pretvoriti u kontejner, a potom da svaki servis bude dodan u zajedničko okruženje, gdje bi mikroservisi dobili mogućnost međusobne komunikacije.

6.1. Kontejnerizacija servisa

U aplikaciju na svaki servis dodali smo datoteku koja se zove „Dockerfile“ i služi da bi se servis za vrijeme docker build-a pretvorio u „image“ i tako nastao zaseban kontejner.



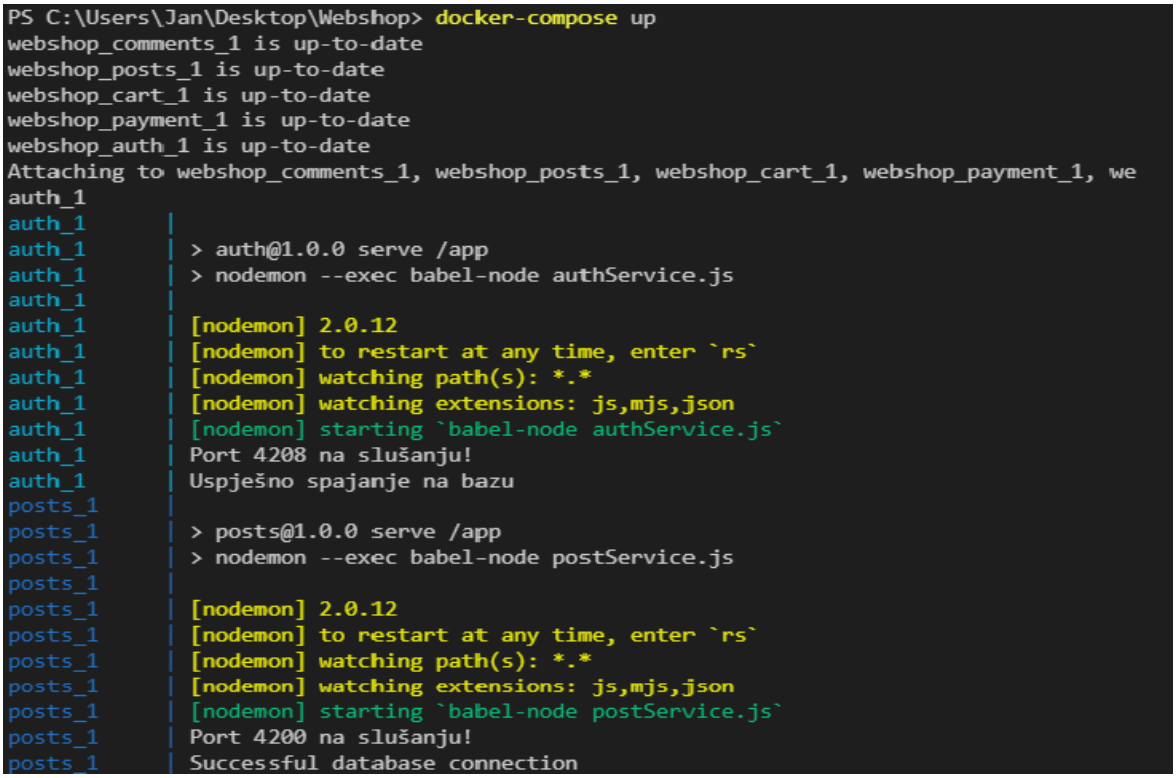
Slika 14. Primjer docker build-a na pojedinom servisu

Prvo smo radili „docker build“ da bi Docker okruženje pojedini servis uspjelo proširiti u obliku „image-a“. Zatim, pomoću „docker run -p 4201:4201 comments:latest“ inicijalizirali smo Docker kontejner koji je lokalno rezervirao port 4201. Nakon toga, servis je spreman i dostupan na lokalnoj mašini, ali pod Docker-ovim okruženjem.

6.2. Spajanje kontejnera

U trenutku kad su postavljene sve Dockerfile datoteke, znači da su kontejneri inicijalizirani što dovodi do pitanja: „Kako ujediniti servise koji imaju međusobnu komunikaciju?“. Na to pitanje bi mogao odgovoriti Docker-compose jer on omogućuje spajanje svih kontejnera u jedno zajedničko okruženje. Prvo smo morali indicirati servise koje želimo spojiti, a zatim im iz starih portova pridodati nove koje će se koristiti. Također bilo je potrebno, svakom servisu koji komunicira s drugim, nadodati IP adresu od lokalnog poslužitelja kako bi ju docker-compose okruženje uspjelo registrirati. Trebali smo pridodati „networking“ na svakom servisu te svakom servisu koji komunicira s drugim staviti „links <naziv servisa>“. Radili smo „docker-compose build“ za izgradnju svih servisa, pomoću samo jedne naredbe. Zatim smo pomoću „docker-

compose up“ naredbe uspostavili okruženje mikroservisa zajedno i omogućila im se zajednička komunikacija.



Slika 15. Pokretanje svih mikroservisa

7. Zaključak

Web aplikacije u današnjici postale su svakodnevna uporaba za sve ljude. Dostupnost web aplikacija vrlo je lagana, od korištenja na računalu pa sve do manjih uređaja. Također, jedini uvjet za dostupnost su posjedovanje uređaja te internetska konekcija.

Mikroservisna arhitektura izvrstan je način da web aplikacije budu skalabilne i optimiziranije u pogledu na njihovu kompleksnost. Ako se aplikacija čini prejednostavnom, preporučuje se korištenje monolitne arhitekture. U suprotnom, ako je aplikacija kompleksna te ako ima puno korisničkih interakcija, trebala bi biti raspodijeljena u manje servise kako bi preglednost koda bila jednostavnija. Osim toga, i ako padne jedan servis na serveru, ostali neovisni servisi će i dalje biti aktivni na korisničkom sučelju. Još jedna prednost koju je važno napomenuti je da servisi mogu funkcionirati zasebno, a mogu biti i napravljeni individualno, odnosno mogu se miješati s drugim servisima, čak i ako im okruženje s tim servisima nije bilo zadano.

Što se tiče raspoređivanja servisa, moglo bi se reći da je važno kako ih rasporediti, u koje okruženje ih staviti i pobrinuti se da su svi u tom okruženju te da su svi u funkciji. Ako su u pitanju aplikacije koje imaju puno korisnika u istom trenutku, u tom slučaju svakako se preporučuju koristiti replike, odnosno kopije istih servisa, kako bi broj korisnika koji vrše interakcije mogao biti izbalansirani na svakoj replici od servisa.

Github rješenje

Cijelu aplikaciju u cijelosti prenio sam na GitHub poslužitelj za projekte otvorenog koda. Aplikacija Thingify dostupna je na: *GitHub - jantudjan99/thingify*.

Popis literature

Nadareishvili I., Mitra R., McLarty M. & Amundsen M. (2016). Microservice Architecture. O'Reilly Media. Dostupno na: https://docs.broadcom.com/doc/microservice-architecture-aligning-principles-practices-and-culture

Altaver A. (5. svibanj 2017). What is N-Tier Architecture? How It Works, Examples, Tutorials, and More. Dostupno na: https://stackify.com/n-tier-architecture/

Demchenko M. (28. travanj 2020). Microservices vs. Monolithic: Which Architecture Suits Best for Your Project? Dostupno na: https://ncube.com/blog/microservices-vs-monolithic-which-architecture-suits-best-for-your-project

Froechlich A. (Ožujak 2018). Application layer. Dostupno na: https://www.techtarget.com/searchnetworking/definition/Application-layer

Heller M. (4. travanj 2020). What is Node.js? The JavaScript runtime explained. Dostupno na: https://www.infoworld.com/article/3210589/what-is-nodejs-javascript-runtime-explained.html

Carey S. (2. kolovoz 2021). What is Docker? The spark for the container revolution. Dostupno na: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html

Panjwani N. (14. siječanj 2021.). How to Make Microservices Communicate. Dostupno na: https://dzone.com/articles/how-to-make-microservices-communicate

Popis slika

Slika 1. Odnos klijenta preko API zahtjeva u mikroservisu koristeći bazu podataka (Microservices architecture, 2018) 2
Slika 2. Usporedba monolitne i mikroservisne arhitekture (Microservices vs. Monolithic, 2020) 3
Slika 3. Prikaz troslojne arhitekture (Overview of a three-tier application, 2009) 4
Slika 4. Aplikacijski sloj u ulozi HTTP protokola (Application layer, 2019) 5
Slika 5. Thingify – primjer više različitih kontejnera u jednom okruženju 7
Slika 6. UML komponentni dijagram 8
Slika 7. UML klasni dijagram 9
Slika 8. Sinkrona komunikacija sa servisom za komentare 10
Slika 9. Ispis svih podataka o proizvodu 10
Slika 10. Dohvat svih komentara 11
Slika 11. Dodavanje pojedinačnog proizvoda u košaricu 12
Slika 12. Logika spremanja i zbrajanja proizvoda u košaricu 13
Slika 13. Komunikacija sa 2 servisa i plaćanje po podacima iz baze 13
Slika 14. Primjer docker build-a na pojedinom servisu 15
Slika 15. Pokretanje svih mikroservisa 16