Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004:007
DOI: https://doi.org/10.24867/14OI01Varmedja

UPOTREBA MIKROSERVISNE ARHITEKTURE ZA RAZVOJ E-COMMERCE APLIKACIJE

THE USE OF MICROSERVICES ARCHITECTURE FOR BUILDING E-COMMERCE APPLICATION

Dejan Varmeđa, *Fakultet tehničkih nauka, Novi Sad*

Oblast – INŽENJERSTVO INFORMACIONIH SISTEMA

Kratak sadržaj – *Kroz ovaj rad biće opisan način na koji se može kreirati e-commerce aplikacija za prodaju i kupovinu knjiga. Krajnje rešenje je veb aplikacija koja je zasnovana na mikroservisnoj arhitekturi. Detaljno će biti opisane korišćene tehnologije, kao i sama arhitektura aplikacije.*

Ključne reči: *Docker, Node.js, React, Next.js, Kubernetes, MongoDB, Redis, mikroservisna arhitektura, NATS*

Abstract – *This papper will describe how an e-commerece application can be created to sell and buy books. The ultimate solution is a web application that is based on microservices architecture. The technology used will be described in detail, as well as the architecture of the application itself.*

Keywords: *Docker, Node.js, React, Next.js, Kubernetes, MongoDB, Redis, microservices architecture*

1. UVOD

U današnje vreme je moguće kupiti bilo kakav proizvod uz samo par klikova. Kupovina preko interneta je u početku ljudima delovala nesigurno, međutim, vremenom je taj strah nestao i sada ljudi koriste veb aplikacije za kupovinu raznih stvari. Razvoj informacionih tehnologija je doveo do toga da se upravo takve aplikacije kreiraju. Elektronska trgovina je postala izuzetno popularna, jer je omogućila ljudima da na različitim stranama nađu i kupe razne proizvode. Pored toga, ljudi nisu morali samo da kupuju proizvode, nego im je pružena mogućnosti i da sami prodaju određene proizvode uz samo par klikova.

U ovom radu je prikazano kako se može kreirati jedna takva aplikacija, koja omogućava korisnicima kupovinu i prodaju knjiga. S obzirom da je u današnje vreme mikroservisna arhitektura izuzetno popularna, ona će biti korišćena za razvoj aplikacije i na tome će biti naglasak. Aplikacija će omogućiti korisniku da napravi svoj nalog, ukoliko ga već ne poseduje, na kojem će imati mogućnost da pogleda knjige koje se trenutno prodaju, da kupi odgovarajuću knjigu tako što će uneti podatke sa kartice, kao i da postavi knjigu koju želi da proda.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio profesor dr Srđan Sladojević.

2. KORIŠĆENE TEHNOLOGIJE

Tehnologije koje su korišćene za kreiranje aplikacije su:

- Node.js
- React
- Next.js
- MongoDB
- Redis
- NATS
- Docker
- Kubernetes.

Za izradu ove aplikacije korišćena je mikroservisna arhitektura.

U nastavku je dat kratak opis korišćenih tehnologija.

2.1 Node.js

Node.js je jedan od najpopularnijih JavaScript frame-work-a. Veliki broj aplikacija koristi *Node.js* kao glavnu tehnologiju.

S obzirom da koristi JavaScript programski jezik, to omogućava inženjerima da kreiraju full-stack aplikacije, koristeći samo jedan jezik za front-end i back-end.

Jedna od najbitnijih *Node*-ovih karakteristika jeste libuv. Libuv je *Node* servis napisan u C programskom jeziku koji predstavlja interfejs sa operativnim sistemom i omogućava asinhrone I/O operacije. Na vrhu libuv-a se nalazi *add-ons* sistem koji omogućava inženjerima da prošire *Node koristeći* C++ i dodaju karakteristike sa većim performansama [1].

2.2 .React

React omogućava kreiranje SPA aplikacija. *React* ima svoj virtuelni DOM, koji koristi i održava ga na osnovu instrukcija u kodu. Virtuelni DOM je kreiran i menjan po potrebi zasnovan na procesu koji se zove *reconciliation.* Taj proces *React* radi interno i to je proces u kojem *React* poredi svoj virtuelni DOM i browser-ov DOM. Kada postoje određene različitosti između virtuelnog DOM-a i browser DOM-a, *React* šalje instrukcije browser DOM-u za kreiranje ili brisanje određenih elemenata, kako bi oba DOM-a bila ista.

Ceo taj proces zove se *commit phase.* Još jedna bitna karakteristika *React*-a jeste da je on *state-driven.* Svaka komponenta može da ima svoje lokano stanje. U trenutku promene stanja, *React* trigeruje *reconciliation* proces i pravi određene promene na DOM-u ukoliko ih ima.

2.3 Next.js

Next.js je *React* framework koji služi za kreiranje SPA aplikacija. Ono što je ključno kod *Next.js*-a jeste da je u pitanju *server side rendering* framework. To znači da će se *React* komponente renderovati na server strani, pre nego što se pošalje HTML klijentu. Rutiranje je automatsko, što znači da će se URL mapirati na fajlove koji se nalaze u *pages*folderu.

2.4 MongoDB

MongoDB je dokument orijentisana NoSQL baza podataka. Za NoSQL se kaže da su to „Ne samo SQL" baze koje ne zahtevaju fiksnu šemu. *MongoDB* koristi kolekcije i dokumenta za čuvanje podataka. Dokumenti se sastoje od ključ-vrednost parova što je glavna jedinica podataka u *MongoDB* bazi. Kolekcije se sastoje od više dokumenata, što je ekvivalentno tabelama u relacionim bazama podataka. Svaki dokument može da ima različit broj polja, tako da veličina i sadržaj svakog dokumenta može da bude različit. Svaki dokument u *MongoDB* bazi mora da ima _id polje. To polje je primarni ključ za dokument. Ukoliko se kreira dokument bez tog polja, *MongoDB* će ga automatski kreirati [2].

2.5 Redis

Kao i *MongoDB, Redis* spada u NoSQL baze podataka i spada u grupu gde se podaci organizuju u ključ-vrednost parove. *Redis* se sastoji iz *Redis* servera i *Redis* klijenta. *Redis* čuva podatke u primarnoj memoriji, što znači da će se podaci izgubiti u slučaju gašenja servera. Prednost čuvanja podataka u primarnoj memoriji, je ta što *Redis* omogućava brze operacije čitanja i pisanja. *Redis* podržava različite tipove podataka, a to su:

- String,
- Hash,
- List,
- Set,
- Sorted set,
- Bitmaps,
- HyperLogLog.

2.6 NATS

NATS je sistem za razmenu poruka kreiran 2010. godine. Prvenstveno je kreiran da služi kao *message bus* za CloudFoundary, međutim razvojem mikroservisne arhitekture, njegova popularnost počinje da raste. Za *NATS* se kaže da je to *PubSub* sistem, što znači *Publish/Subscribe*. Ovakav model za razmenu poruka omogućava klijentima da komuniciraju bez znanja o tome gde se servisi nalaze u network-u. Klijent postaje potrošak ili pretplatnik kada se registruje na odgovarajući sadržaj. U toj situaciji, kada god *publisher* emituje odgovarajuću poruku o tom sadržaju, sistem za razmenu poruka će dostaviti te informacije svim klijentima koji su se pretplatili na taj sadržaj [3].

2.7 Docker

Inženjeri koriste Docker kontejnere kako bi u taj kontejner zapakovali aplikaciju koja je kreirana, zajedno sa svim bibliotekama i *dependency*-jima. Ti paketi se šalju operacionim inženjerima koji mogu da svaki kontejner tretiraju isto, bez obzira kakva aplikacija se izvršava u okviru kontejnera. U okviru *Docker* arhitekture, postoje određene stvari, a to su: *Control Groups, Namespaces, Layer Capabilities* i druge funkcionalnosti vezane za operativni sistem. Takođe, postoji *Docker engine* koji pruža RESTful interfejs, kako bi mu se moglo pristupiti sa drugim alatima, kao što je *Docker CLI, Kubernetes* i drugi. Kada se pokrene *Docker* kontejner, kreira se image. Kontejner image predstavlja obrazac na osnovu kojeg se kreira kontejner. Image je sastavljen iz više slojeva. Postoji nekoliko načina da se kreira image, ali najkorišćeniji je pomoću *Dockerfile*-a. *Dockerfile* je tekstualni fajl koji sadrži instrukcije kako da se kreira kontejner image [4].

2.8 Kubernetes

Kubernetes je softver koji omogućava inženjerima lakše upravljanje i *deploy*-ovanje kontejnerizovanih aplikacija. Njega ne zanimaju interni detalji tih aplikacija, s obzirom da se aplikacije izvršavaju u kontejneru. Za *Kubernetes* se može reći da je to operativni sistem za klastere. *Kubernetes* klaster se sastoji od više čvorova. Ti čvorovi se mogu podeliti u dve grupe, *master* čvor koji vodi računa o celom *Kubernetes* sistemu i *worker čvor* u kojem se kontejnerizovana aplikacija izvršava. *Kubernetes* vodi računa o tome da se stanje aplikacije uvek podudara sa opisom koj je naveden. U *Kubernetes*-u, *pod* predstavlja grupu jednog ili više povezanih kontejnera koji će se uvek izvršavati zajedno na istom *worker* čvoru. Svaki *pod* ima odgovarajuće stvari, kao što su IP adresa, hostname, proces i mnoge druge stvari [5].

2.9 Mikroservisna arhitektura

Monolitne aplikacije se sastoje od komponenata koje su usko povezane. Sve te komponente se zajedno razvijaju, *deploy*-uju i na kraju spajaju u jednu celinu s obzirom da se izvršavaju kao jedan proces operativnog sistema. Promene jedne komponente aplikacije bi značilo da se *deployment* za celu aplikaciju ponovo odradi. Takođe, dosta je teže dodati nove funkcionalnosti u aplikaciju kako ona vremenom raste. Skaliranje monolitnih aplikacija često predstavlja veliki izazov. Problemi sa monolitniim aplikacijama doveli su do razvoja mikroservisa.

Kompleksne monolitne aplikacije su se razdvajale na manje nezavisne delove. Ti delovi se zovu mikroservisi. Svaki mikroservis je nezavistan od drugih. Takođe, *deployment* jednog mikroservisa nema nikakve veza sa *deployment*-om drugog. U mikroservisnoj arhitekturi, mikroservisi su tehnološki nezavisni, što bi značilo da je moguće imati jedan mikroservis koji je kreiran u programskom jeziku C#, dok je drugi mikroservis kreiran u jeziku Python. Mikroservisi komuniciraju kroz sinhrone protokole kao što je HTTP preko kojeg pružaju određene RESTful API-je ili preko nekih asinhronih protokola, kao što je AMQP. Skaliranje kod mikroservisa je dosta lakše.

Samo oni servisi kojima je potrebno više resura će biti skalirani.

Mikroservisne aplikacije su dosta kompleksnije nego monolitne i ta kompleksnost dovodi do raznih izazova koji moraju biti rešeni kako bi se mikroservisna arhitektura isplatila i dala veću vrednost nego monolitna [6].

3. OPIS FUNKCIONALNOSTI SISTEMA

Korisnik ove aplikacije će imati mogućnost da kupuje i prodaje knjige. Pre svega, korisnik mora imati napravljen nalog na aplikaciji kako bi mogao da kupuje i prodaje. Aplikacija je iimplementirana u mikroservisnoj arhitekturi, a servisi koji su kreirani su:

- Auth,
- Books,
- Orders,
- Expiration,
- Payments,
- Client
- Infra.

Pored ovih servisa kreiran je i jedan paket, a u pitanju je *common. Common* paket u okviru sebe sadrži stvari koje su zajedničke za sve servise. U ovom paketu se nalaze korisnički definisane greške, midlveri kao i *event*-ovi. Ukoliko je potrebno u određenom servisu baciti odgovarajuću grešku, koristi se jedna od korisnički definisanih grešaka iz *Common* paketa.

Takođe, paket sadrži nekoliko midlvera, kao što je provera da li je korisnik autentifikovan, da li je potrebno izvršiti autentifikaciju, midlver za upravljanje greškama kao i midlvera za validiranje zahteva. Pored ovoga, kreirane su i dve abstraktne klase, *Publisher* i *Listener*.

Većina servisa će emitovati određene poruke i slušati na neke druge poruke. Zbog toga će svaka klasa koja će emitovati određenu poruku naslediti *Publisher* klasu, dok će svaka klasa koja je zadužena za slušanje određene poruke, naslediti klasu *Listener*. *Auth* servis je zadužen za autentifikaciju korisnika. Odnosno, u okviru ovog servisa se nalazi logika koja omogućava korisniku da napravi nalog ukoliko ga ne poseduje, da se prijavi na aplikaciju, da se odjavi, kao i logika koja proverava da li je korisnik ulogovan.

Ovaj servis koristi *MongoDB* bazu podataka za čuvanje svih korisnika. U *Books* servisu se nalazi logika za postavljanje određene knjige, za prikazivanje svih knjiga, prikazivanje pojedinačne knjige, kao i logika za modifikovanje knjige. Kao i u *Auth* servisu, *Books* servis koristi *MongoDB* bazu podataka za smeštanje svih knjiga. Prilikom kreiranja knjige, emituje se event koji će obavestiti *Orders* servis da je knjiga kreirana. Isto se dešava i prilikom modifikovanja knjige.

I *Books* i *Orders* servis čuvaju podatke o knjigama u bazi podataka, a to se radi kako bi se izbegla direktan komunikacija između servisa. Iako se može desiti da se u jednom trenutku podaci u bazi podataka razlikuju kod ova servisa, to će se vremenom ispraviti i obe baze će imati iste podatke.

Taj pojam je poznat kao *Eventual Consistency*. Pored dva publisher-a, *Books* servis ima i dva listener-a koji će osluškivati na promene koje su se desile prilikom kreiranja ili modifikovanja narudžbine. U toj situaciji, potrebno je promeniti status knjige. *Orders* servis je zadužen za kreiranje narudžbine. Ovaj servis omogućava korisniku prikaz svih njegovih narudžbina, prikaz pojedinačne narudžbine, kreiranje nove narudžbine, kao i brisanje postojeće.

Kao i *Books* servis, i *Orders* servis ima nekoliko listener-a. Servis sluša na promene iz *Books, Expiration* i *Payments* servisa. Ukoliko dođe do plaćanja narudžbine, izvršiće se listener koji će promeniti status narudžbine na *Complete,* što bi značilo da je narudžbina uspešno realizovana. Takođe, korisnik ima određen vremenski period u kojem mora da izvrši plaćanje. Ukoliko to ne uradi, narudžbina će se poništiti.

Upravo ta funkcionalnost dolazi iz *Expiration* servisa. Svaka narudžbina ima vreme trajanja. Nakon isteka tog vremana, *Expiration* servis emituje odgovarajući event, nakog čega će narudžbina biti poništena i knjiga će biti dostupna drugim korisnicima. *Payments* servis u okviru sebe sadrži logiku koja omogućava plaćanje narudžbine. Prilikom plaćanja naružbine, emituje se event koji će obavestiti sve zainteresovane servise da je narudžbina realizovana.

U *Client* servisu se nalazi *frontend* deo aplikacije. Korisnik prilikom otvaranja aplikacije mora da se uloguje ili da napravi nalog ukoliko ga ne poseduje. Nakon toga, biće mu prikazane sve knjige koje mogu da se kupe. Svaka knjiga poseduje link koji otvara novu stranicu i prikazuje samo svoje podatke.

Prilikom prikaza tih podataka, korisnik ima mogućnost da kreira narudžbinu. Prilikom kreiranja narudžbine, korisnik će biti preusmeren na drugu stranicu u kojoj ima mogućnost da izvrši plaćanje. Takođe, korisniku će biti prikazano koliko mu je vremena ostalo za plaćanje. Pored svih ovih stranica, korisnik može da vidi sve narudžbine koje je kreirao.

Na kraju, korisnik može da se izloguje sa aplikacije ukoliko to želi. Poslednji servis u ovoj aplikaciji je *Infra*. U ovom servisu se nalaze *Kubernetes* konfiguracioni fajlovi koji su kreirani za svaki servis i za svaku bazu podataka koja se nalazi u servisu.

4. ZAKLJUČAK

Ovim radom dat je prikaz upotrebe mikroservisne arhitekture kako bi se kreirala *e-commerce* aplikacija. Početak rada se sastoji iz kratkog uvoda, nakon čega sledi prikaz tehnologija pomoću kojih je kreirana aplikacija. Glavni deo rada bio je vezan za implementaciju aplikacije, gde je opisan svaki servis.

Cilj ove aplikacije jeste da se korisnik upozna na koji način se može kreirati aplikacija sa mikroservisnom arhitekturom.

Najveći fokus je bio na servisima i radom sa mikroservisnom arhitekturom, dok je dosta manji bio na *frontend* delu aplikacije.

5. LITERATURA

[1] L. M. Mario Casciaro, Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques, 3rd Edition, Packt Publishing, 2020.

[2] „What is MongoDB? Introduction, Architecture, Features & Example,“ Guru99, [Na mreži]. Available: https://www.guru99.com/what-is-mongodb.html.

[3] W. Quevedo, Practical NATS: From Beginner to Pro, Apress, 2018.

[4] G. N. Schenker, Learn Docker - Fundamentals of Docker 19.x, Packt Publishing, 2020.

[5] M. Luksa, Kubernetes in Action, Manning Publications, 2018.

[6] I. N. Ronnie Mitra, Microservices Up & Running, O'Reilly, 2020.

Kratka biografija

Dejan Varmeđa rođen je u Rumi 1995. godine. Master rad na Fakultetu tehničkih nauka iz oblasti Inženjerstvo informacionih sistema odbranio je 2021. godine.