UNIVERZITET U BEOGRADU
ELEKTROTEHNIČKI FAKULTET

Mikroservisna arhitektura implementirana u radnom okviru Django sa komunikacijom preko RabbitMQ-a

Diplomski rad

Profesor:
Prof. dr Dragan Bojić

Kandidat:
Gavrić Sava 2018/0359

Beograd, septembar 2023.

Sadržaj

1. Uvod i opis problema

Cilj rada jeste ilustrovanje i rešavanje problema asinhrone komunikacije među servisima u sistemu sa mikroservisnom arhitekturom.

U ovom, prvom poglavlju biće opisani opštiji pojmovi mikroservisne arhitekture i komunikacione paradigme vođene porukama. Takođe biće dat i opis konkretnog sistema implementiranog u okviru rada.

U drugom poglavlju biće opisane tehnologije korišćene prilikom implementacije rešenja, kao i specifičnosti tih tehnologija u kontekstu rešenja samog problema.

U trećem poglavlju biće opisan način upotrebe rešenja – pokretanje sistema i osnovni uslovi koje je potrebno ispuniti da bi se ilustrovao rad rešenja.

U četvrtom poglavlju opisana je realizacija sistema – način na koji su tehnologije iskorišćene, problemi koji su susretnuti i način na koji su isti rešeni.

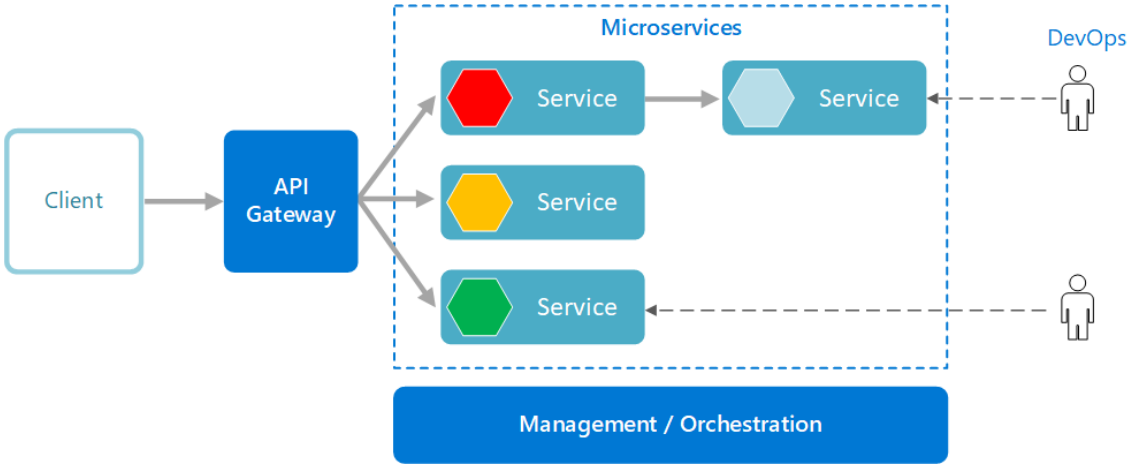
Peto poglavlje predstavlja zaključak rada i opisuje moguća buduća poboljšanja sistema.

1.1. Mikroservisna arhitektura

Mikroservisi predstavljaju **arhitekturalni šablon** (eng. *pattern*)[1] po kom se aplikacija organizuje u skup labavo spregnutih (eng. *loosely coupled*) servisa organizovanih oko nezavnisnih poslovnih funkcionalnosti [1].

Ideja iza mikroservisne arhitekture jeste da se celokupni **biznis (poslovni) domen**[2] sistema razdeli na što nezavnisnije poddomene. U praksi su ovi domeni najčešće definisani oko **entiteta**[3]. Zatim se ovi poddomeni implementiraju kao zasebni mikroservisi – lake i nezavnisne celine poslovne logike koje pružaju interfejs za rad sa svojim entitetom, i eventualno komuniciraju sa drugim mikroservisima kako bi pružili neku međudomensku poslovnu funkcionalnost [2].

[1] Arhitekturalni šablon jeste opšte, prilagodljivo rešenje za problem koji se u softverskoj arhitekturi često javlja u određenom kontekstu.
[2] Biznis domen predstavlja skup poslovnih pravila koje je potrebno opisati softverom.
[3] Entitet predstavlja objekat koji ima identitet, nezavisan od promena njegovih atributa i koga karakteriše dug životni vek i relevantnost za korisnika sistema.



Slika 1.1.1. Koncept mikroservisne arhitekture

Neke od glavnih **prednosti** ovog šablona u odnosu na tradicionalnu **monolitnu**[4] arhitekturu uključuju:

- **jednostavni servisi** – mikroservisi i poddomeni problema koji oni rešavaju su jednostavniji od monolitne tesno spregnute (*tightly coupled*) logike
- **timska autonomija** – pojedinačni tim razvija, testira i isporučuje promene nad svojim mikroservisom nezavisno od drugih timova i njihovih obaveza što dovodi do smanjenja redundantne komunikacije i povećanja produktivnosti
- **brza isporuka promena** – svaki servis, zbog njegove lakoće, moguće je brzo promeniti, testirati i isporučiti
- **nezavnisna implementacija pojedinačnog servisa** – različiti servisi mogu biti implementirani pomoću različitih skupova tehnologija – komunikacija između njih vrši se protokolom koga ne zanima *low level* implementacija
- **izolacija otkaza** – u slučaju otkaza određenog servisa, sistem i dalje radi nesmetano (dok god servisi koji komuniciraju sa pomenutim servisom implementiraju logiku rukovanja otkazima)
- **efikasna skalabilnost** – servisi u sistemu sada mogu biti skalirani nezavnisno i pojedinačno, po potrebi, umesto skaliranja celog sistema kao što bi bio slučaj u monolitnom sistemu
- **izolacija podataka** – lakša izmena šeme podataka – potrebno izmeniti šemu samo u jednom servisu, jer samo on direktno pristupa šemi

[4] Monolitna arhitektura predstavlja arhitekturalni šablon gde su komponente sistema tesno spregnute u jednu kohezivnu celinu.

Neke od većih **izazova/mana** ove arhitekture uključuju:

- **povećana komplesnost** – mikroservisna aplikacija ima više pokretnih delova od monolitne – svaki taj deo jeste jednostavniji, ali je sistem kao celina kompleksniji
- **distribuirane operacije mogu biti neefikasne** – posledica postojanja velikog broja mikroservisa jeste povećana komunikacija između njih; u slučaju komplikovanijih operacija, kašnjenje na mreži takođe postaje faktor
- **implementacija transakcija je komplikovanija** – s obzirom da svaki mikroservis radi nad svojom bazom podataka, transakcije ne mogu jednostavno biti implementirane nad jednom bazom kao što je to slučaj sa monolitnim sistemom
- **verzionisanje** – ažuriranje servisa ne sme slomiti servise koji zavise od njega – bez pažljive analize i dizajna mogući su problemi sa kompatibilnošću
- **znanje timova** – s obzirom da su mikroservisi visoko distribuiran sistem, potrebno je da tim poseduje određen nivo znanja kako bi njihovu implementaciju uspešno i sproveo

Međuservisna komunikacija najčešće je realizovana nekim vidom middleware-a orijentisanog porukama.

1.2. Middleware orijentisan porukama

Middleware[5] orijentisan **porukama** (eng. *message-oriented middleware*) (nadalje **MOP**) predstavlja infrastrukturu za slanje i prijem poruka izmedju distribuiranih sistema [3].

Poruke u ovom kontekstu su nezavisne jedinice informacije najčešće u JSON ili XML[6] formatu. Sama struktura podataka u porukama definisana je poslovnom logikom konkretnog sistema.

MOP komunikacija je asihrona – pošiljalac i primalac ne moraju biti aktivni tokom slanja odnosno prijema poruke, već nezavisno šalju/primaju poruku kada su za to spremni.

U skladu sa asinhronom filozofijom, poslate poruke najčešće se skladište u nekom agregatoru poruka, npr. **redovima** (eng. queue) – pošiljalac dostavi poruku u red, primalac je preuzme kada je za to spreman. Primalac i pošiljalac na ovaj način uopšte ne moraju da znaju jedan za drugoga – ne posotji nikakva direktna zavisnost između njih.

Čest aspekt MOP sistema jeste garantovana dostava poruke – očuvanje i isporuka poruke su osigurane, čak i u slučaju sistemskog otkaza. Prethodno pomenuti redovi igraju ulogu u ovom aspektu – često su implementirani tako da na određeni način skladište svoj trenutni sadržaj u fajl sistemu.

[5] Middleware jeste softver koji deluje kao spona između dve ili više softverskih komponenti.
[6] Česti formati za struktuiranje podataka.

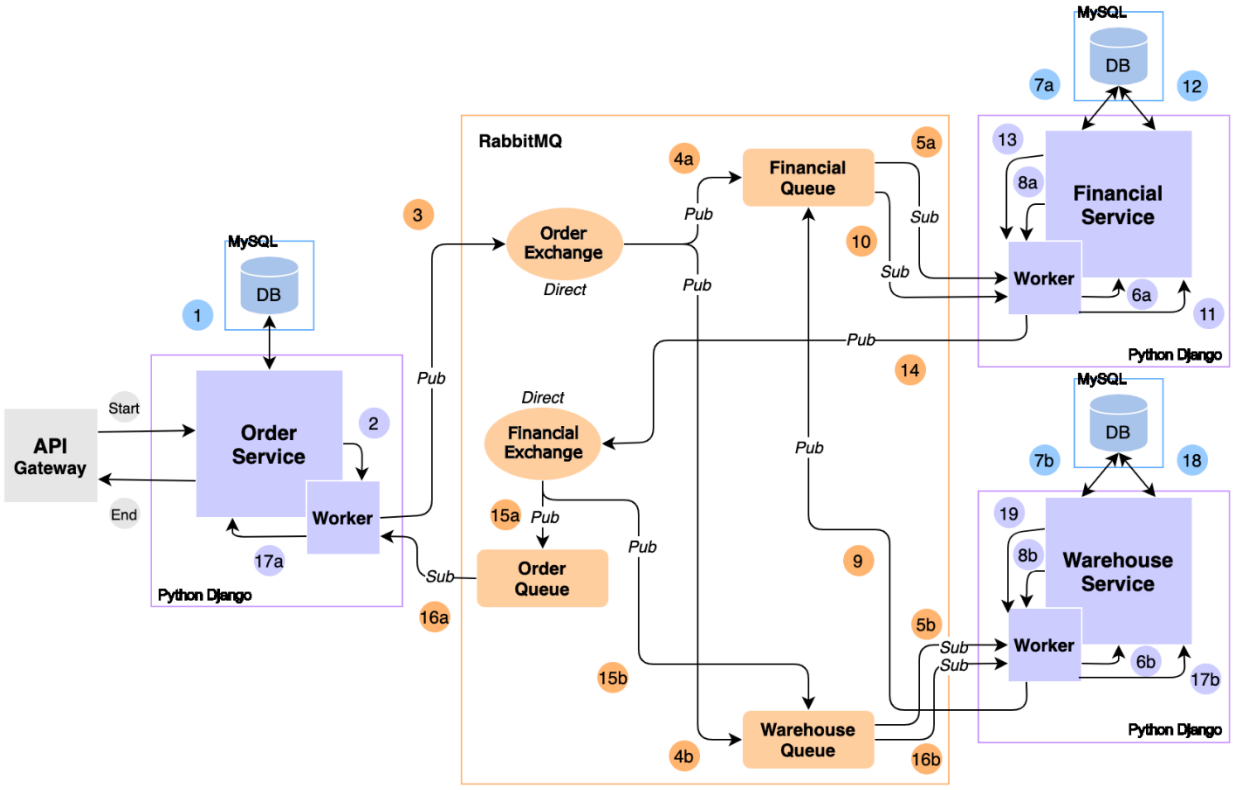
Prethodno pomenuti MOP principi često su na nižem nivou implementirani u ***Advanced Message Queuing Protokol*-u** (nadalje **AMQP**) [11].

AMQP specificira kako poruke treba da budu formatirane, prenesene i konzumirane. Takođe pruža implementaciju raznih šablona slanja poruka: ***point-to-point*** (korišćenjem **redova**), ***publish-subscribe***[7] (korišćenjem tema), ***request-reply***[8] itd. Ova fleksibilnost pruža mogućnost arhitektama da izaberu šablon komunikacije koji najviše odgovara poslovnoj logici konkretnog sistema.

Pored svih prethodno pomenutih aspekata MOP-a, AMQP takođe implementira sofisticiran sistem rutiranja poruka – AMQP pruža različite mehanizme kojima pojedinačne poruke mogu biti usmerene na određene redove ili teme na osnovu sadržaja ili zaglavlja.

1.3. Arhitektura rešenja

Arhitektura sistema može se videti na fotografiji ispod.



Slika 1.3.1. Arhitektura konkretnog rešenja

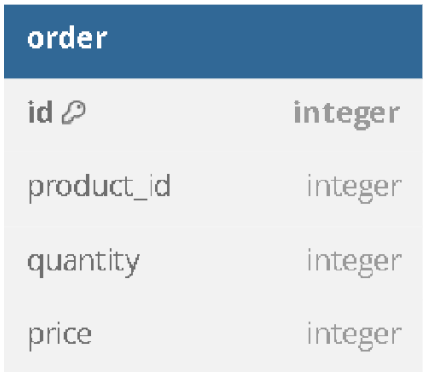
[7] Model komunikacije gde se potencijalni primaoci poruka pretplaćuju na određenog pošiljaoca. Kada pošiljalac pošalje poruku, ona biva dostavljena pretplatnicima i samo njima.
[8] Model gde pošiljalac šalje zahtev primaocu, primalac ga obradi i pošiljaocu pošalje odgovor.

U sistemu postoje četiri jedinstvena mikroservisa: servis za narudžbine (*order*), finansijski servis (*invoice*), servis skladišta (*warehouse*) i API kapija (*API gateway*).

1.3.1. Order servis

Servis narudžbina je REST-*ful*[9] API vezan za entitet **narudžbine** i pruža mogućnost CRUD[10] operacija nad narudžbinama.

Model narudžbine ima sledeću strukturu:



Slika 1.3.1.1. model narudžbine

Polje **product_id** predstavlja identifikator proizvoda vezanog za narudžbinu. Trenutno u sistemu na nivou baze podataka ne postoji način da se osigura da proizvod sa datim identifikatorom postoji (jer je svaki od entiteta u zasebnoj bazi podataka) – ova konzistentnost podataka mora se osigurati na aplikativnom nivou.

Order API pruža sledeće CRUD ***endpoint***-ove[11]:

- dohvatanje liste svih narudžbina
  - putanja: *api/v1/orders*
  - HTTP metod: GET
  - format zahteva: /

[9] REST-ful API je API koji ispunjava ograničenja REST arhitekturalnog stila.
[10] Akronim za *Create Read Update Delete* – četiri osnovne operacije nad podacima u sistemu.
[11] API endpoint predstavlja konkretan URL kome klijent može da šalje zahteve, a obično predstavlja konkretno funkciju ili resurs koju server pruža za taj zahtev.

    - format odgovora:

```
[
    {
        "id": integer,
        "product_id": integer,
        "quantity": integer,
        "price": integer
    }
]
```

- kreiranje nove narudžbine
    - putanja: *api/v1/orders*
    - HTTP metod: POST
    - format zahteva:

```
{
    "product_id": integer,
    "quantity": integer,
    "price": integer
}
```

    - format odgovora:

```
{
    "id": integer,
    "product_id": integer,
    "quantity": integer,
    "price": integer
}
```

- ažuriranje postojeće narudžbine
    - putanja: *api/v1/orders/{order_id}*
    - HTTP metod: PUT
    - format zahteva:

```
{
    "product_id": integer,
    "quantity": integer,
    "price": integer
}
```

    - format odgovora:

```
{
    "id": integer,
    "product_id": integer,
    "quantity": integer,
    "price": integer
}
```

- brisanje postojeće narudžbine
    - putanja: *api/v1/orders/{order_id}*
    - HTTP metod: DELETE
    - format zahteva: /

- format odgovora: /

Pored CRUD *endpoint*-ova, orderAPI takođe pruža sledeći *endpoint* za **odjavljivanje** (eng. *checkout*) narudžbine:

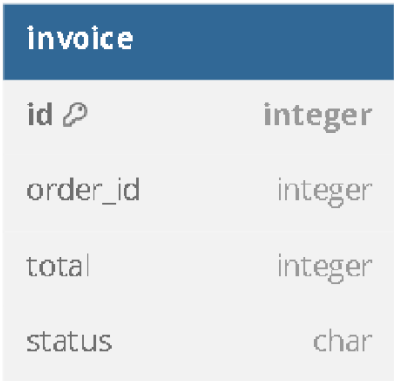
- putanja: *api/v1/orders/checkout/{order_id}*
- HTTP metod: POST
- format zahteva: /
- format odgovora: /

Ovaj *endpoint* pokreće mehanizam za odjavljivanje narudžbine putem RabbitMQ-a.

1.3.2. Finansijski servis

Finansijski servis je REST-ful API vezan za **entitet fakture** i pruža mogućnost CRUD operacija nad fakturama.

Model fakture ima sledeću strukturu:



Slika 1.3.2.1. model fakture

Polje **order_id** predstavlja identifikator narudžbine vezane za fakturu. Ovde važi ista napomena kao za **product_id** polje kod modela narudžbine.

Invoice API pruža sledeće CRUD ***endpoint***-ove:

- dohvatanje liste svih faktura
  - putanja: *api/v1/invoices*
  - HTTP metod: GET
  - format zahteva: /

    - format odgovora:

```
[
    {
        "id": integer,
        "order_id": integer,
        "total": integer,
        "status": string
    }
]
```

- kreiranje nove fakture
    - putanja: *api/v1/invoices*
    - HTTP metod: POST
    - format zahteva:

```
{
    "order_id": integer,
    "total": integer,
    "status": string
}
```

    - format odgovora:

```
{
    "id": integer,
    "order_id": integer,
    "total": integer,
    "status": string
}
```

- ažuriranje postojeće fakture
    - putanja: *api/v1/invoices/{invoice_id}*
    - HTTP metod: PUT
    - format zahteva:

```
{
    "order_id": integer,
    "total": integer,
    "status": string
}
```

    - format odgovora:

```
{
    "id": integer,
    "order_id": integer,
    "total": integer,
    "status": string
}
```

- brisanje postojeće fakture
    - putanja: *api/v1/invoices/{invoice_id}*
    - HTTP metod: DELETE
    - format zahteva: /

- format odgovora: /

Pored standardnih CRUD *endpoint*-ova na osnovu identifikatora fakture, invoice API pruža i sledeća dva *endpoint*-a za manipulaciju fakturom, preko identifikatora narudžbine, kako bi olakšao izmenu podataka vezanih za određenu narudžbinu tokom procesa ***checkout*-a** date narudžbine:

- ažuriranje statusa postojeće fakture po identifikatoru narudžbine:
  - putanja: *api/v1/invoices/orders/{order_id}*
  - HTTP metod: PUT
  - format zahteva:

```
{
    "status": string
}
```

  - format odgovora:

```
{
    "status": string
}
```

- brisanje postojeće fakture po identifikatoru narudžbine:
  - putanja: *api/v1/invoices/orders/{order_id}*
  - HTTP metod: DELETE
  - format zahteva: /
  - format odgovora: /

Takođe, u svrhe prikazivanja relevantnih informacija za narudžbine korisniku API kapije, invoice API pruža sledeći *endpoint* za dohvatanje faktura za narudžbine definisane nizom identifikatora narudžbina:

- putanja: *api/v1/invoices/orders*
- HTTP metod: POST
- format zahteva:

```
{
    "order_ids": [integer]
}
```

- format odgovora:

```
[
    {
        "id": integer,
        "order_id": integer,
        "total": integer,
        "status": string
    }
]
```

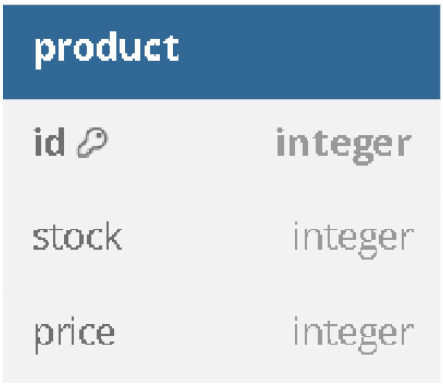
Potreba za ovim endpoint-om postoji iz sledećeg razloga: kada API kapija dobije narudžbine za prikaz, potrebno je za svaku narudžbinu prikazati i trenutni status njene fakture, ukoliko ona postoji. Ukoliko ne postoji potrebno je prikazati dugme za *checkout* date narudžbine. Prolazak

kroz sve narudžbine i slanje zahteva za dohvatanje fakture za svaku narudžbinu ponaosob bi bilo izuzetno skupo – za N narudžbina slalo bi se N HTTP zahteva kroz mrežu – u realnom sistemu potpuno nedopustivo. Ovim *endpointom* se osigurava da se informacije o fakturama za N narudžbina dohvataju **jednim** HTTP zahtevom. Na finansijskom servisu sve vreme se dodatno izvršava i demonski[12] worker proces koji čeka na RabbitMQ poruke koje na odgovarajući način obrađuje.

1.3.3. Servis skladišta

Servis skladišta je REST-ful API vezan za **entitet proizvoda** i pruža mogućnost CRUD operacija nad proizvodima.

Model proizvoda ima sledeću strukturu:



Slika 1.3.3.1. Model proizvoda

Warehouse API pruža sledeće CRUD ***endpoint***-ove:

- dohvatanje liste svih proizvoda
  - putanja: *api/v1/products*
  - HTTP metod: GET
  - format zahteva: /
  - format odgovora:

```
[
    {
        "id": integer,
        "stock": integer,
        "price": integer,
    }
]
```

[12] Demonski (eng. *daemon*) proces jeste dugovečni pozadinski proces koji obično obavlja funkciju održavanja sistema ili pruža određene usluge.

- dohvatanje pojedinačnog proizvoda
  - putanja: *api/v1/products/{product_id}*
  - HTTP metod: GET
  - format zahteva: /
  - format odgovora:

```
{
    "id": integer,
    "stock": integer,
    "price": integer,
}
```

- kreiranje novog proizvoda
  - putanja: *api/v1/products*
  - HTTP metod: POST
  - format zahteva:

```
{
    "stock": integer,
    "price": integer,
}
```

  - format odgovora:

```
{
    "id": integer,
    "stock": integer,
    "price": integer,
}
```

- ažuriranje postojećeg prozivoda
  - putanja: *api/v1/products/{product_id}*
  - HTTP metod: PUT
  - format zahteva:

```
{
    "stock": integer,
    "price": integer,
}
```

  - format odgovora:

```
{
    "id": integer,
    "stock": integer,
    "price": integer,
}
```

- brisanje postojećeg proizvoda
  - putanja: *api/v1/products/{product_id}*
  - HTTP metod: DELETE
  - format zahteva: /
  - format odgovora: /
- inkrementalno/dekrementalno ažuriranje zaliha (stock) datog proizvoda

- putanja: *api/v1/products/{product_id}*
- HTTP metod: PUT
- format zahteva:

```
{
    "quantity": integer,
}
```

- format odgovora:

```
{
    "id": integer,
    "stock": integer,
    "price": integer,
}
```

Analogno finansijskom servisu, i na servisu skladišta se sve vreme dodatno izvršava i demonski worker proces namenjen RabbitMQ komunikaciji.

1.3.4. API kapija

API kapija (eng. **API *gateway***) predstavlja jednostavnu web aplikaciju koja krajnjem korisniku pruža korisnički interfejs za manipulaciju narudžbinama i započinjanjem procesa checkout-a narudžbine.

1.3.5. Komunikacija

RabbitMQ služi kao medijum za komunikaciju između mikroservisa.

Svaki od 3 API servisa – *order* API, *financial* API i *warehouse* API imaju svoj zaseban **red**.

Redovi u RabbitMQ-u predstavljaju privremeno skladište poruka koje kasnije mogu iskoristiti jedan ili više primaoca. Ovi redovi funkcionišu kao i tradicionalni redovi, *po first in – first out* (FIFO) principu – poruka koja je najpre stigla u red najpre će biti konzumirana.

Financial i *warehouse* API preko svojih demonskih *worker*-a sve vreme čekaju na poruku u svojim redovima, kada poruka stigne oni je konzumiraju i obrađuju na odgovarajući način.

Order API, koristi svoj red tako što sinhrono čeka na poruku od ostatka sistema tokom procesa *checkout*-a pojedinačne narudžbine.

U sistemu u svrhe rutiranja poruka postoje dve razmene (eng. *exchange*) – razmena narudžbina (eng. *Order exchange*) na koji poruke šalje servis za narudžbine, i razmena finansija (eng. *Financial exchange*) na koji poruke šalje servis za finansije.

Razmena u RabbitMQ predstavlja komponentu za rutiranje poruka. Ona prima poruke od pošiljaoca i zatim je rutira do konkretnih primaoca (njihovih redova) po određenim definisanim pravilima.

Kao destinacije razmene narudžbina konfigurisani su red finansijskog servisa i red servisa skladišta. To znači da, kada servis za narudžbine pusti poruku niz razmenu narudžbina, poruka će završiti u prethodno pomenutim redovima.

Ova konfiguracija rutiranja definisana je ključevima rutiranja (eng. *routing keys*) – string vrednost kojom se određuje kome treba rutirati poruku. Naime, prilikom inicijalizacije reda, on se „veže“ za određenu razmenu i tom prilikom se definiše ključ rutiranja pod kojim red očekuje poruke sa razmene. Kada pošiljalac pošalje poruku na razmenu on kao ključ rutiranja za datu poruku stavlja istu vrednost – time se osigurava da će poruka otići na one i samo na one redove koji su vezani za datu razmenu sa istom vrednošću ključa rutiranja.

Analogno razmeni narudžbina, kao destinacije razmene finansija konfigurisani su red servisa za narudžbine i red servisa skladišta.

Sada kada su definisane sve komunikacione komponente sistema može se opisati proces *checkout*-a narudžbine, kojim ilustrujemo ovu međuservisnu, asinhronu, AMQP komunikaciju. Pretpostavka je da u sistemu postoji sačuvana narudžbina za određeni proizvod i da ona još uvek nije *checkout*-ovana. Na zahtev za *checkout*-om narudžbine *happy path*[13] tok kontrole je sledeći:

Start – Do servisa narudžbina stiže zahtev za *checkout*-om date narudžbine, bilo od API kapije ili od nekog drugog dela sistema.

1 – Servis narudžbina dohvata detalje o narudžbini iz svoje baze podataka – konkretno identifikator proizvoda vezanog za narudžbinu kao i kvantitet datog proizvoda u narudžbini.

2 – Servis narudžbina pravi poruku od podataka o narudžbini i serijalizuje je.

3 – Servis narudžbina šalje poruku na razmenu narudžbina, u sledećem formatu:

```
{
    "order_id": integer,
    "product_id": integer,
    "quantity": integer,
    "price": integer,
    "message_type": "ORDER_CHECKOUT",
}
```

4a,4b – Razmena narudžbina prima poruku i, na osnovu povezanih redova i ključeva rutiranja, rutira poruku na red servisa finansija i red servisa skladišta.

5a – *Worker* servisa finansija prima poruku preko svog reda, na osnovu **message_type** polja zaključuje kako poruku treba procesuirati.

[13] *Happy path* predstavlja idealnu/očekivanu sekvencu koraka u jednom procesu/slučaju korišćenja gde sve prođe bez ikakvih grešaka, izuzetaka ili problema bilo kog tipa.

5b – *Worker* servisa skladišta prima poruku preko svog reda, na osnovu **message_type** polja zaključuje na koji način poruka treba da bude obrađena.

6a – *Worker* servisa finansija šalje zahtev za kreiranjem fakture za datu narudžbinu *financial* API-ju. Koristi se *endpoint* za kreiranje nove fakture (*/api/v1/invoices*). Telo zahteva:

```
{
    "order_id": integer,
    "total": integer,
    "status": "incomplete"
}
```

Polje **total** ima vrednost jednaku vrednosti **quantity** * **price.**

6b – *Worker* servisa skladišta šalje zahtev za dohvatanjem podataka o proizvodu vezanim za datu narudžbinu kako bi proverio da li zalihe datog proizvoda zadovoljavaju količinu proizvoda u datoj narudžbini.

7a – Servis finansija uspešno stvara fakturu o datoj narudžbini u finansijskoj bazi podataka.

7b – Servis skladišta uspešno dobija podatak o zalihama proizvoda iz baze podataka skladišta.

8a – *Worker* finansijskog servisa od servisa dobija odgovor o uspešnom kreiranju fakture.

8b – *Worker* servisa skladišta dobija odgovor koji sadrži podatke o datom proizvodu. Zatim poredi zalihe proizvoda sa količinom proizvoda iz porudžbine. Pošto je količina proizvoda iz porudžbine manja ili jednaka zalihama proizvoda, narudžbina može biti obrađena.

9 – *Worker* servisa skladišta šalje poruku servisu finansija kojom govori da ima dovoljno zaliha proizvoda i da se stanje fakture može ažurirati. Format poruke:

```
{
    "order_id": integer,
    "product_id": integer,
    "quantity": integer,
    "message_type": "CONFIRM_STOCK",
    "in_stock": true
}
```

10 – *Worker* finansijskog servisa prima poruku preko svog reda.

11 – *Worker* finansijskog servisa šalje zahtev za ažuriranjem statusa fakture sa *incomplete* na *waiting*. Koristi se endpoint za ažuriranje statusa fakture preko identifikatora narudžbine (*api/v1/invoices/orders/{order_id}*). Telo zahteva:

```
{
    "status": "incomplete"
}
```

12 – Finansijski servis ažurira status fakture.

13 – *Worker* finansijskog servisa od servisa dobija odgovor da je faktura uspešno ažurirana.

14 – *Worker* finansijskog servisa šalje poruku o uspešnom ažuriranju fakture na razmenu finansija. Format poruke:

```
{
    "order_id": integer,
    "product_id": integer,
    "quantity": integer,
    "message_type": "CONFIRM_INVOICE",
    "confirm_invoice": true
}
```

15a, 15b – Razmena finansija prima poruku i, na osnovu povezanih redova i ključeva rutiranja, rutira poruku na red servisa narudžbina i red servisa skladišta.

16a – *Callback* funkcija blokirajućeg čekanja na kanalu sa koga je poslata inicijalna poruka o *checkout*-u narudžbine se okida.

16b – *Worker* servisa skladišta prima poruku preko svog reda.

17a – U servisu narudžbina završava se sa izvršenjem *callback* funkcije blokirajućeg čekanja na kanalu. Funkcija vrši eventualnu obradu nad odgovorom dobijenog sa RabbitMQ kanala i vraća obrađen odgovor glavnom toku obrade inicijalnog zahteva za *checkout*-om narudžbine u kontekstu *Order* API-ja.

17b – *Worker* servisa skladišta šalje zahtev servisu za dekrementiranjem zaliha datog proizvoda. Koristi se *endpoint* za ažuriranje zaliha proizvoda za datu vrednost (*api/v1/products/stock/{product_id}*). Telo zahteva:

```
{
    "quantity": integer
}
```

Polje **quantity** ima vrednost jednaku negativnoj vrednosti **quantity** polja iz RabbitMQ poruke.

18 – Servis skladišta uspešno dekrementira zalihe proizvoda u bazi podataka skladišta.

19 – *Worker* servisa skladišta od servisa dobija odgovor da je stanje proizvoda uspešno ažurirano.

Kraj - Servis narudžbina, s obzirom da ustanovljuje da je odgovor sa RabbitMQ-a uspešnog statusa, završava inicijalni zahtev za *checkout* narudžbine vraćanjem HTTP odgovora sa 204 statusnim kodom.

Napomena: kada više stavki ima isti redni broj to znači da se te stavke izvršavaju u paraleli.

U slučaju da iz nekog razloga nema dovoljno zaliha proizvoda da bi se narudžbina uspešno *checkout*-ovala, napravljena faktura će se obrisati, a servisu narudžbina će se vratiti poruka sa opisom greške.

2. Pregled korišćenih tehnologija

U ovom poglavlju biće detaljno opisane sve tehnologije vredne pomena, kojima je rešenje implementirano.

2.1. Python/Django

U svrhe implementacije logike API servisa korišćeni su **Python** i radni okvir **Django**.

Python jeste interpretirani objektno-orijentisani programski jezik visokog nivoa [5].

Specifičnost Python-a jeste težnja ka čitljivosti koda – za razliku od tradicionalnih jezika za definisanje blokova koda koristi se indentacija umesto standardnih vitičastih zagrada – ovaj naziv segmentisanja koda indentacijom danas se naziva „*Pythonic*“ – po ovom programskom jeziku.
Vredno pomena je i dinamično tipiziranje promenljivih i parametara – interpreter ustanovljava tip podatka u *runtime*-u, time čineći jezik fleksibilnijim, po cenu jasnoće i potencijalno stabilnosti koda.

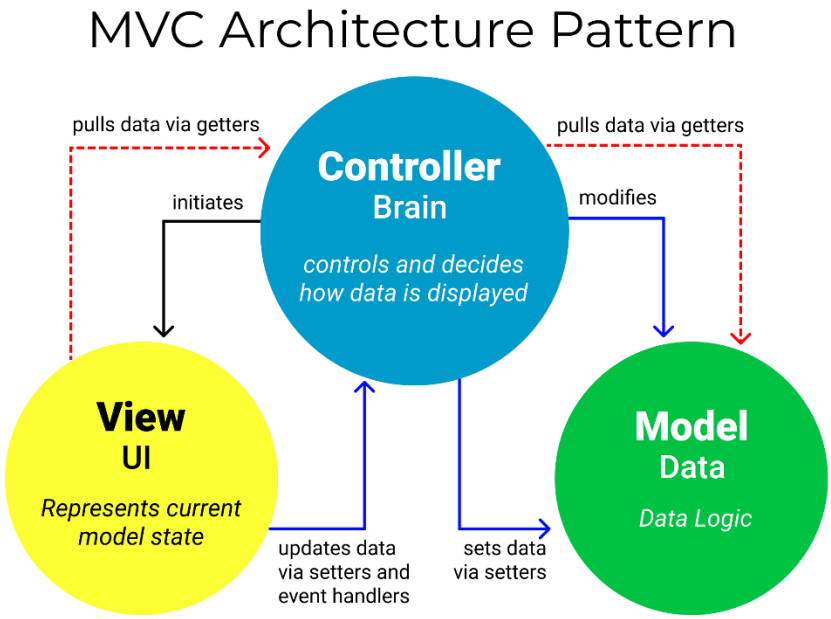
Ovaj jezik je takođe pogodan za *web development*, sa jakom ugrađenom kompatibilnošću i dobro dokumentovanim i popularnim radnim okvirima poput Django-a ili Flask-a. Python takođe karakterišu bogate biblioteke posvećene *data science*-u i mašinskom učenju[14] – time ga čineći često prvim izborom u akademskim sferama.

Django jeste *open-source*[15] radni okvir za razvijanje web aplikacija pisan u Python-u [6]. Django je karakterističan po tome što od početka dolazi sa velikom većinom često potrebnih funkcionalnosti ugrađenih u sebe, što omogućuje razvijanje dobrog dela složenog sistema bez potrebe za dodatnim spoljnim zavisnostima. Zasnovan je na ***model-view-template*** (**MVT**) arhitekturalnom šablonu – varijaciji tradicionalnog ***model-view-controller*** (**MVC**) šablona. Razlika između dva šablona objašnjena je u nastavku.

U **MVC** šablonu odgovornosti na tri komponte podeljene su na sledeći način. ***Model*** predstavlja podatke i poslovnu logiku. Odgovoran je za dohvatanje podataka iz i skladištenje podataka u izvor podataka (u praksi najčešće u bazu podataka), kao i definisanje i izvršenje same poslovne logike sistema. Bitno je naglasiti da model nikako direktno ne komunicira sa *view*-om.
View prezentuje podatke korisniku sistema. Odgovaran je za *render*-ovanje korisničkog interfejsa i vizuelno formatiranje podataka dobijenih od izvora podataka. Bitno je naglasiti da *view* često sadrži određenu aplikativnu logiku kojom dinamički može promeniti prikaz korisniku na osnovu

[14] NumPy, pandas, TensorFlow i drugi.
[15] *Open-source* softver jeste potpuno dostupan javnosti - moguće ga je pregledati, koristiti, modifikovati i distribuirati slobodno.

promena u modelu (o kojima ga obaveštava kontroler). ***Controller*** obrađuje korisnički unos i ponaša se kao posrednik između prethodno opisanih modela i *view*-a. On prihvata korisnički input, procesira ga, usput komunicirajući sa modelom kako bi podatke na odgovarajući način dohvatio/formatirao. Zatim na odgovarajući način ažurira *view* kako bi korisnik bio svestan promena nastalih u sistemu.



Slika 2.1.1. Koncept MVC arhitekturalnog šablona

U **MVT** šablonu komponente imaju prilično slične odgovornosti, sa par bitnih razlika. ***Model*** ima praktično iste odgovornosti što se tiče manipulacije podacima i implementacije poslobne logike.
View je u MVT šablonu praktično pandan kontroleru u MVC šablonu, s tim što na sebe preuzima i odgovornost sakupljanja podataka za prikazivanje od modela, i zatim prosleđivanje tih podataka na odgovarajući način odgovarajućem *template*-u.
Template je prostija varijanta MVC *view*-a – sadrži samo jasnu definiciju kako tačno formatirati i prikazati podatke koje je dobio od MVT view-a.

U svrhe generisanja gorepomenutih template-a Django pruža mogućnost korišćenja ***Django Template Language***-a (**DTL**) – sistema za generisanje web stranica sa kodom sličnim Python-u ugrađenim u HTML kod, čime se omogućuje dinamičko ubrizgavanje podataka u stranice na strani servera prilikom generisanja odgovora. Korišćenjem DTL-a, prilikom generisanja HTML stranice moguće skupom uslova diktirati koji delovi stranice se prikazuju, sa kojim tačno podacima, u kom formatu. DTL kod se u stranicu ugrađuje korišćenjem takozvanih

template tag-ova („{%%}“) – pritom pružajući sve osnovne Python funkcionalnosti poput for petlji, kondicionalnih iskaza poput if-else, dinamičkog uključivanja (eng. *include*) drugih *template*-a u trenutni itd.

Pored karakteristične arhitekture zasnovane na *template*-ima Django pruža podršku za sledeće korisne funkcionalnosti:

- ugrađen admin grafički korisnički interfejs koji od starta pruža mogućnost CRUD operacija nad definisanim modelima
- URL rutiranje koje svodi problem definisanja rutiranja na definisanje ruta na jednom centralizovanom mestu
- sistem za autentikaciju i autorizaciju koji omogućuje jednostavno definisanje naloga, permisija, kontrole pristupa i sl.
- *middleware* podrška koja omogućuje procesiranje zahteva i odgovora na globalnom nivou, pre nego što prođu kroz *view*; ovim mehanizmom zgodno se rešavaju problemi autentikacije, keširanja i log-ovanja

Od dodatnih biblioteka korišćenih za implementaciju rešenja vredi pomenuti:

- decouple – biblioteka za lakše upravljanje konfiguracijom sistema – u rešenju korišćena za dohvatanje vrednosti ***environment*** **varijabli**[16]
- pika – klijentska biblioteka za RabbitMQ – u rešenju korišćenja za konfigurisanje i interakciju sa RabbitMQ servisom [7]
- unittest – biblioteka za pisanje *unit* testova – u rešenju korišćena za *mock*-ovanje[17] API poziva [8]

2.2. HTML i CSS

HTML (***Hypertext Markup Language***) jeste *markup* jezik koji služi za prikaz i struktuiranje sadržaja na *web* stranicama. Jedna *web* stranica opisana je stablom elemenata zvanim *tag*-ovi, koji definišu njenu strukturu i semantiku. Ovi elementi, između ostalog, formatiraju tekst, ubacuju slike i *link*-ove, i organizuju prostor *web* stranice i time omogućuju internet pretraživačima da krajnjem korisniku prikažu sadržaj stranice na način na koji je *developer* zamislio.

U rešenju, HTML je korišćen samo za *render*-ovanje stranica za servis API kapije. S tim što nije korišćen obični HTML već HTML sa ugrađenom **DTL** logikom.

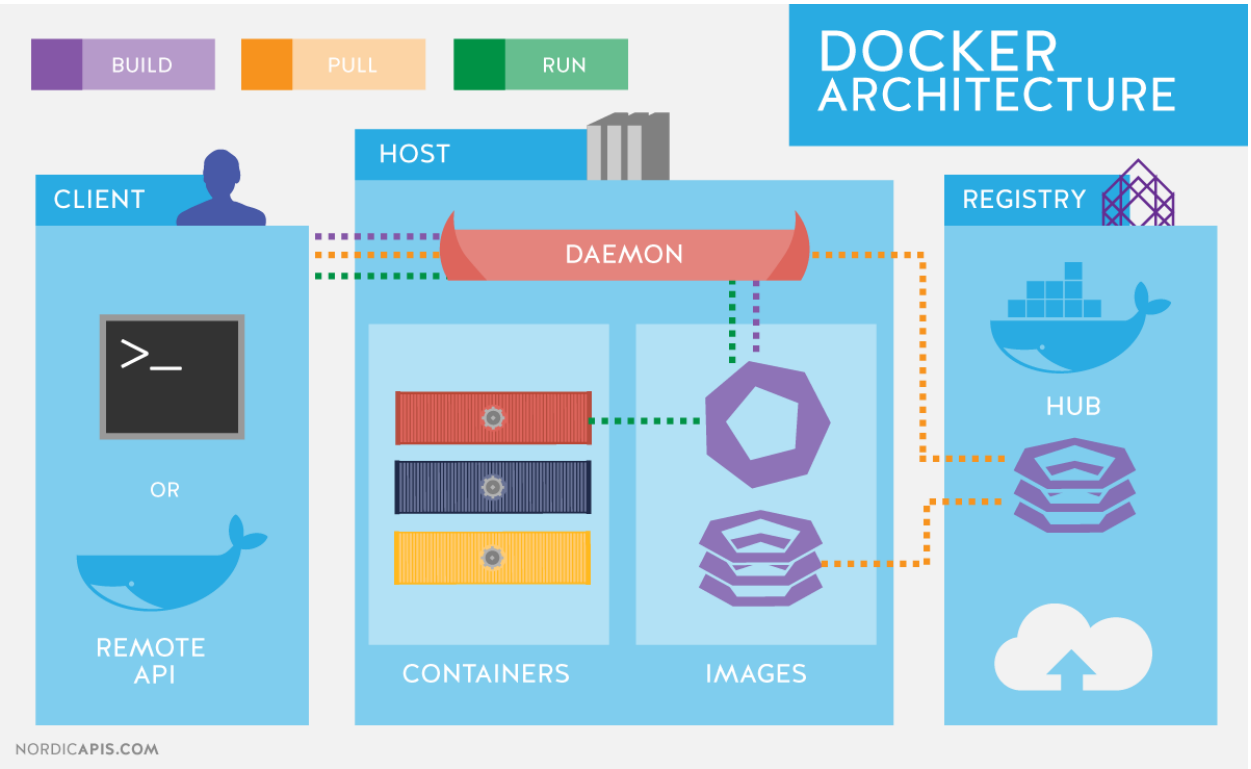
[16] *Enviroment* varijabla jeste imenovana, dinamična vrednost čuvana u kompjuterskom sistemu koja može da utiče na način rada sistema i često se koristi za konfiguraciju sistemskih podešavanja ili da pruži neku informaciju izvršavanom programu.
[17] *Mock*-ovanje predstavlja simuliranje ponašanja objekata ili funkcija od kojih neka jedinica sistema zavisi, u svrhe testiranja funkcionalnosti te jedinice.

CSS (***Cascading Style Sheets***) jeste *stylesheet* jezik korišćen u razvoju *web* stranica za kontrolisanje stilizovanja HTML *tag*-ova. On omogućuje *web developer*-ima da definišu vizuelan izgled komponenti stranice, nevezano za njihov sadržaj.

CSS je u rešenju korišćen za jednostavno stilizovanje HTML stranica za servis API kapije.

2.3. Docker

Docker jeste platforma odnosno skup alata za kontejnerizaciju aplikacija u laka, nezavisna okruženja za izvršavanje zvana **kontejneri**. Docker kontejneri su potpuno izolovani i konzistentni kada je izvršavanje u pitanju, osiguravajući da se aplikacije ponašaju isto nezavisno od operativnog sistema i zavisnosti sistema domaćina. Takođe se mogu koristiti za konfiguraciju i izolovanje sistema u svim fazama njegovog životnog ciklusa – razvoja, testiranja i produkcije.



Slika 2.3.1. Koncept Docker arhitekture

Glavna prednost Docker kontejnera u odnosu na virtuelne mašine, starije, popularno rešenje za virtualezaciju jeste lakoća u smislu zauzeća resursa sistema domaćina (memorije i procesorskog vremena). Virtuelne mašine simuliraju ceo operativni sistem, dok Docker kontejner na sebi, pored,

najčešće, oguljene verzije Linux-a, pokreće samo aplikativni kod i zavisnosti koje on sa sobom nosi. Ovo posebno znači u produkciji, gde na jednom *host* sistemu može stati mnogo više instanci kontejnera aplikacije, nego što je to slučaj sa virtuelnim mašinama. Takođe zbog infrakstrukture koje Docker pruža za konfiguraciju i upravljanje pokrenutim kontejnerima (*Docker Swarm*), lako je skalirati sistem i integrisati ga u postojeća rešenja za *deployment* i upravljanje aplikacijama.

Važno je pomenuti i vrednost Docker-a prilikom razvoja aplikacije. Zbog izolacione prirode Docker kontejner-a, moguće je razvijati aplikaciju koja je pokrenuta na Docker kontejner u bilo kojoj tehnologiji, sa bilo kakvim zavisnostima, potpuno nezavisno od sistema domaćina. Ova funkcionalnost, koja je intenzivno korišćena tokom izrade priloženog rešenja, rasterećuje *developer*-a od vođenja računa o potencijalnim konfliktima zavisnosti, pri radu sa različitim verzijama iste tehnologije.

Glavne komponente Docker platforme uključuju:

- **Docker engine** – osnovna komponenta platforme, zadužena za pokretanje kontejnera i upravljanje istim; sastoji se od servera, REST API-ja, interfejsa komandne linije, a na Windows OS-u i od desktop UI aplikacije
- **Docker image** – predstavlja *read-only* šablon na kome se nalazi aplikacioni kod, sistemski alati, potrebne biblioteke i *environment* varijable, neophodne da bi se kontejner podigao i pokrenuo
- **Dockerfile** – tekstualni fajl koji sadrži instrukcije za građenje Docker *image*-a; može da definiše dodatne konfiguracije nad baznim *image*-om [9]
- **kontejner** – kao što je već opisano, izvršiva instanca Docker *image*-a; enkapsulira aplikaciju i omogućuje njeno izvršenje
- **Docker compose** – alat za definisanje i izvrašavanje Docker aplikacija koje se sastoje od više Docker kontejnera; parsira YAML-formatiran tekstualni fajl koji definiše konfiguraciju svakog servisa pojedinačno, kao i podešavanja za njihovu međusobnu interakciju; servisi se podižu nad odgovarajućim Docker *image*-om [10]
- **Docker Swarm** – alati za orkestraciju koji pružaju mogućnost upravljanja i skaliranja kontejnera u *cluster*-ima – skupovima kontejnera

2.4. RabbitMQ

RabbitMQ jeste *open-source broker* (posrednik) poruka i ujedno implementacija AMQP protokola. Služi za posredovanje u razmeni podataka (poruka) između različitih činilaca distribuiranog sistema, omogućavajući asinhronu komunikaciju [12].

Neke od najbitnijih odlika RabbitMQ-a, a koje već nisu opisane u sekciji rada o AMQP-u su:

- korisnički interfejs u vidu web aplikacije – softver pruža relativno jednostavan interfejs za posmatranje instanci sistema i upravljanje njima
- prioritetni redovi – postoji mogućnost dodeljivanja različitih prioriteta porukama – poruke sa većim prioritetem biće obrađene pre poruka sa nižim prioritetom, korisno kada je redosled poruka bitan
- „*dead letter*“ razmene – mogućnost skladištenja poruka koje nisu uspele da budu dostavljene određeni broj puta ili ispunjavaju neki drugi konfigurisan uslov, u posebne razmene, radi dalje analize i potencijalnog ponovnog pokušaja dostave
- potvrda prijema – primaoci poruke su u mogućnosti da obaveste sistem o prijemu poruke; ovime se osigurava da poruke ne mogu biti izgubljene i da mogu biti obrisane iz reda po uspešnom prijemu od strane primaoca
- *cluster*-ovanje – moguće je *cluster*-ovanje više pojedinačnih RabbitMQ čvorova, kako bi radili zajedno kao jedan logički broker; ova funkcionalnost je korisna u svrhe skalabilnosti i osiguranja dostupnosti
- *plugin*-ovi – RabbitMQ poseduje i *plugin* arhitekturu koja pruža mogućnost proširenja funkcionalnosti osnovnog sistema korisnički-definisanim funkcionalnostima (dodatni načini autentikacije, integracija sa nekim eksternim sistemom itd.)

2.5. MySQL

MySQL jeste *open-source* sistem za upravljanje relacionim bazama podataka. Pošto prati relacioni model podataka, baze organizuje u skupove tabela koje se sastoje od redova koje prate određenu strukturu definisane kolonama te tabele. Tabele su u relaciji jedna sa drugom korišćenjem mehanizma ključeva, koji omogućava efikasno dohvatanje i manipulaciju podacima [13].

MySQL koristi **Structured Query Language** (**SQL**) za upravljanje bazama podataka. Ovaj jezik je danas defakto standard za rad sa bazama podataka.

Neki od najkorisnijih osobina MySQL-a jesu:

- kompatibilnost – sistem je kompatibilan sa svim popularnijim operativnim sistemima današnjice
- skalabilnost – mogućnost replikacije (kloniranje baze podataka u svrhe *load balancing*-a, *backup*-a, odbrane od otkaza) i efikasan mehanizam indeksiranja podataka čine sistem prilagodljivim i ogromnim količinama podataka
- sigurnost – sistem implementira različite mehanizme zaštite podataka uključujući korisničku autentikaciju, kontrolu pristupa, mogućnost enkripcije
- okidači i procedure – sistem pruža mogućnost definisanja okidača (eng. *trigger*) i procedura – korisnički definisane akcije i skripte koje se okidaju na specifične događaje

- dokumentacija i podrška – kao posledica velike popularnosti sistema, on je odlično dokumentovan, sa dobrom online podrškom

2.6. Adminer

Adminer je *open-source* alat za administraciju baza podataka [14]. On pruža korisnički interfejs u vidu web aplikacije koja omogućuje korisnicima da pristupaju bazama podataka i izvode razne operacije nad podacima, uključujući upravljanje kako bazama podataka i njihovim korisnicima (pravljenje, brisanje, modifikovanje), tako i tabelama unutar njih (upravljanje strukturom tabele) kao i samim podacima (standardne CRUD operacije).

Adminer podržava sve popularnije sisteme za upravljanje relacionim bazama podataka (MySQL, PostgreSQL, SQLite, MS SQL itd.).

Tokom razvoja priloženog sistema, Adminer je korišćen u svrhe upravljanja i popunjavanja baza podataka tokom ručnog testiranja aplikacije.

2.7. Postman

Postman je alat koji pruža funkcionalnost HTTP API klijenta i alata za testiranje API-ja [15].

Postman omogućuje developer-ima da:

- šalju HTTP zahtev uz pomoć jednostavnog korisničkog interfejsa – moguće je podesiti bilo koji aspekt zahteva uključujući URL, zaglavlja, *query* parametre, telo zahteva itd.
- dizajniraju API-je – moguće je definisati strukturu API-ja, uključujući *endpoint*-ove, sadržaj zahteva i odgovora
- generisanje dokumentacije – na osnovu dizajna API-ja moguće je generisati i dokumentaciju čiji izgled je moguće konfigurisati

Tokom razvoja priloženog sistema, Postman je korišćen u svrhe ručnog testiranja aplikacije.

3. Pregled sistema i način korišćenja

Preduslov za pokretanje i korišćenje sistema jesu instalirani Docker engine i Docker compose alat na *host* mašini korisnika.

Da bi se sistem pokrenuo dovoljno je u konzoli, dok trenutni radni direktorijum odgovara korenom direktorijumu izvršiti komandu **`docker compose up`**, čime se podižu svi potrebni servisi za rad sistema. Prvo pokretanje sistema može potrajati i do nekoliko minuta, jer je potrebno preuzeti/izgraditi Docker *image*-e za pojedine servise.

Kada je sistem operativan njegovim komponentama moguće je pristupiti na sledeći način:

- servisu narudžbina moguće je pristupiti preko adrese **localhost:8000**
- servisu finansija moguće je pristupiti preko adrese **localhost:8001**
- servisu skladišta moguće je pristupiti preko adrese **localhost:8002**
- API kapiji je moguće pristupiti tako što se u adresnu liniju pretraživača unese **localhost:8003** – ovime se otvara interfejs web aplikacije kapije
- web interfejsu adminer servisa moguće je pristupiti preko adrese **localhost:8080**
- web interfejsu RabbitMQ servisa moguće je pristupiti preko adrese **localhost:15672**

Napomena: ukoliko za time ima potrebe portove je moguće izmeniti u **docker-compose.yaml** fajlu.

Bazi podataka narudžbina preko adminera se pristupa sa sledećim podacima:

- *server*: **database_order**
- *username*: **root**
- *password*: **password**
- *database*: **order**

Bazi podataka finansija preko adminera se pristupa sa sledećim podacima:

- *server*: **database_financial**
- *username*: **root**
- *password*: **password**
- *database*: **financial**

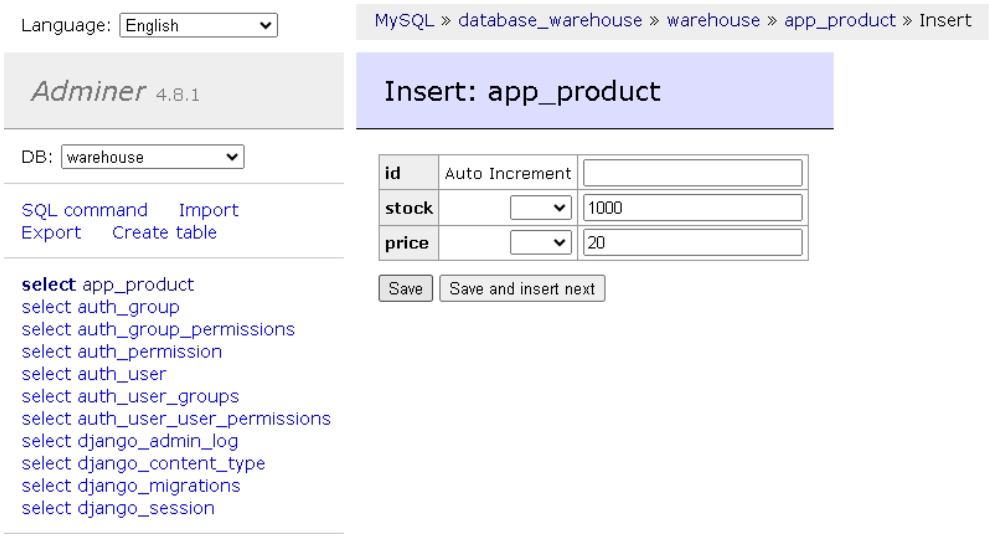
Bazi podataka narudžbina preko adminera se pristupa sa sledećim podacima:

- *server*: **database_warehouse**
- *username*: **root**
- *password*: **password**
- *database*: **warehouse**

Napomena: ukoliko za time ima potrebe vrednosti kredencijala se mogu izmeniti u odgovarajućim **.env** fajlovima unutar projekta.

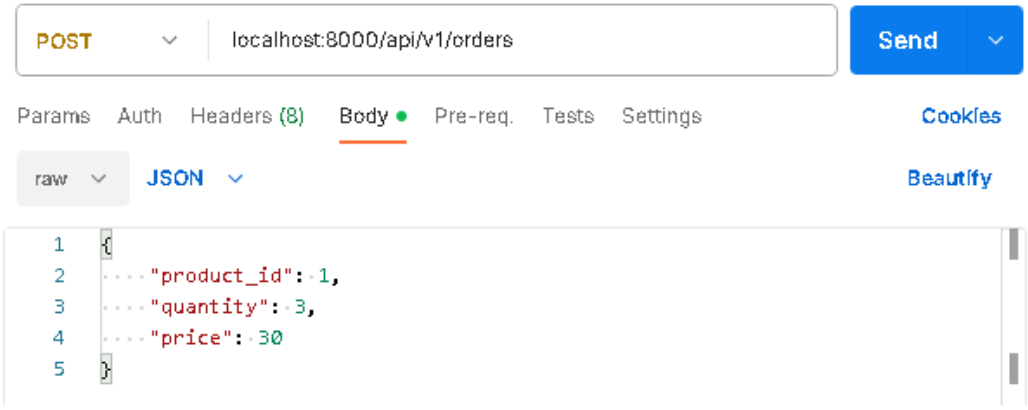
Promene nad podacima narudžbina, faktura i proizvoda mogu se simulirati na dva načina:

- izvršavanjem operacija nad odgovarajućom bazom putem Adminer interfejsa



Slika 3.1. Kreiranje proizvoda kroz Adminer interfejs

- izvršavanjem HTTP zahteva nad odgovarajućim API-jem putem Postman-a:



Slika 3.2. Kreiranje narudžbine kroz Postman

Kroz samu API kapiju moguće je:

- naručiti novu narudžbinu

Slika 3.3. Kreiranje narudžbine za proizvod #1 kroz korisnički interfejs API kapije

- Kada se pošalje zahtev za kreiranjem nove narudžbine API kapiji, on će zahtev proslediti servisu narudžbina (zvanjem *endpoint*-a za kreiranje nove narudžbine). Preduslov da bi narudžbina bila napravljena, jeste to da prvo mora postojati proizvod koji bi bio poručen.

- započeti *checkout* narudžbine

Slika 3.4. Narudžbina koja nije checkout-ovana

Kada se pošalje zahtev za *checkout*-om narudžbine API kapiji, ona će zahtev proslediti servisu narudžbina (zvanjem *endpoint*-a za *checkout* narudžbine). Servis narudžbina dalje okida RabbitMQ mehanizam komunikacije.
Preduslov da bi narudžbina bila uspešno *checkout*-ovana, jeste to da trenutne zalihe

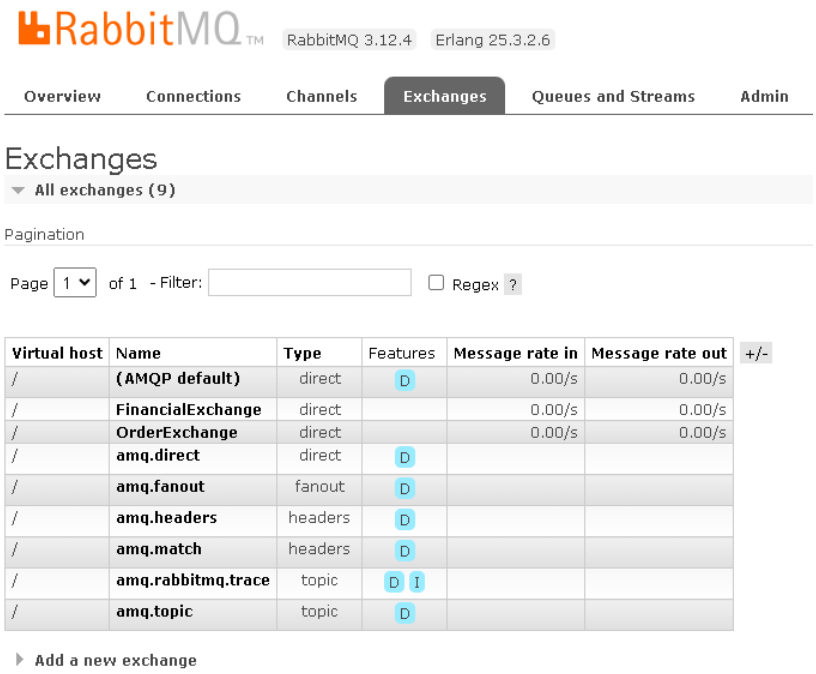
proizvoda narudžbine moraju biti veće ili jednake količini datog proizvoda u datoj narudžbini.
Pri uspešnom *checkout*-u narudžbine, status njene fakture prebaciće se u **complete**.

RabbitMQ interfejsu za upravljanje komunikacionom infrastrukturom pristupa se preko adrese **localhost:15672**. Kredencijali su:

- *username*: **guest**
- *password*: **guest**

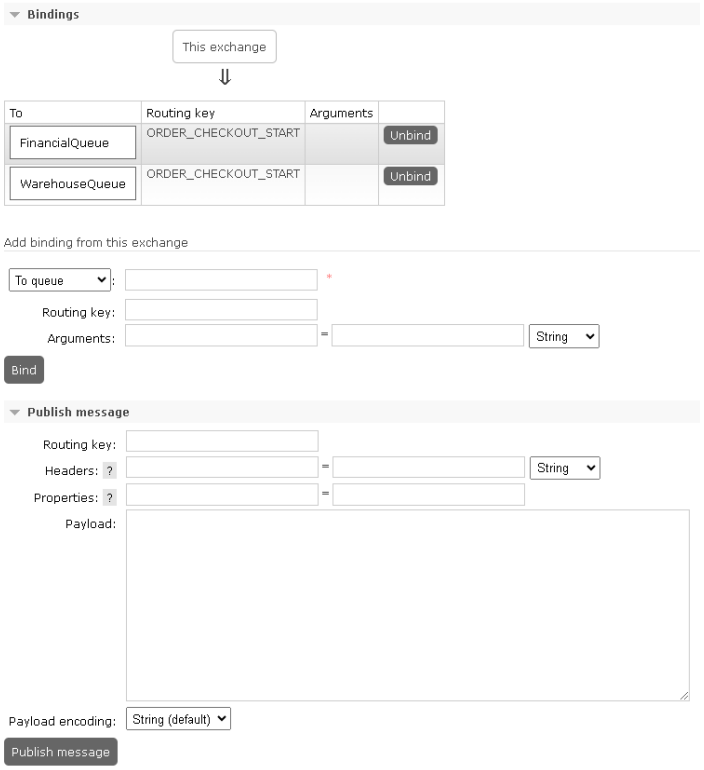
Kroz ovaj interfejs moguće je potpuno konfigurisati RabbitMQ instancu. Neke od funkcionalnosti relevantninh za ovo rešenje su:

- pregled trenutno aktivnih razmena



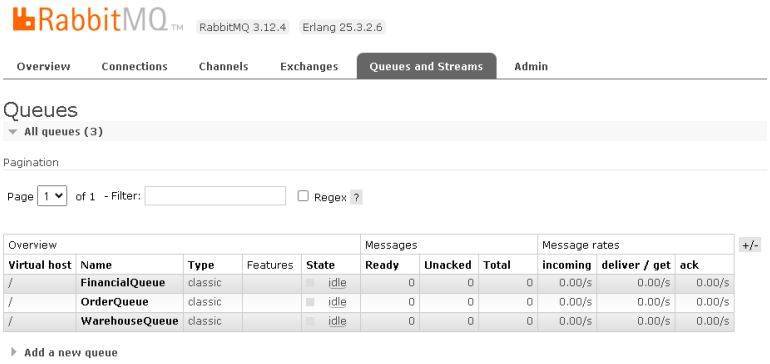
Slika 3.5. Pregled svih razmena u RabbitMQ web aplikaciji

- podešavanje kao i manipulacija (pravljenje, brisanje) porukama u razmeni:



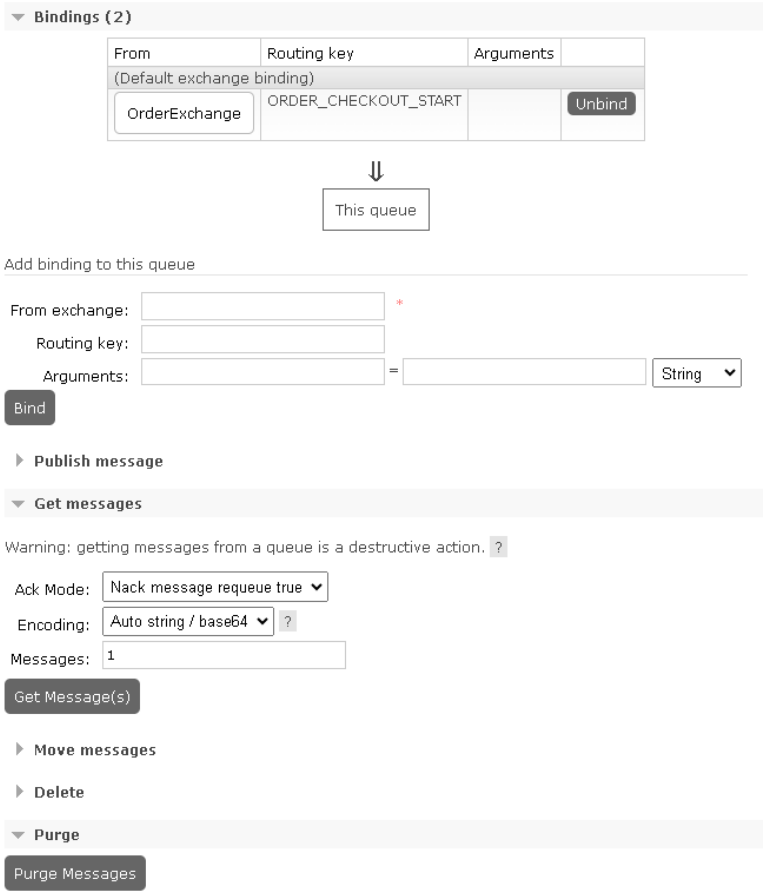
Slika 3.6. Upravljanje razmenama u RabbitMQ web aplikaciji

- pregled trenutno aktivnih redova:



Slika 3.7. Pregled svih redova u RabbitMQ web aplikaciji

- podešavanje kao i manipulacija (pravljenje, brisanje) porukama u redovima:



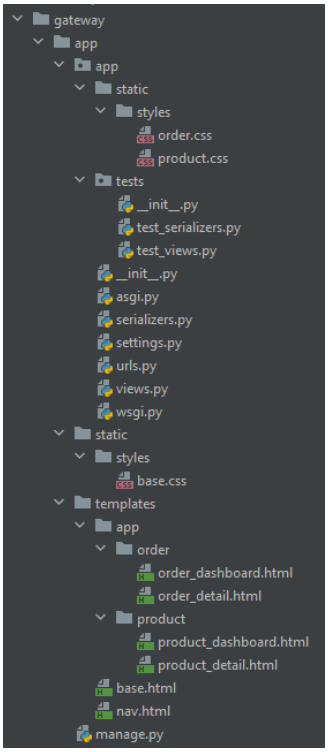
Slika 3.8. Upravljanje redovima u RabbitMQ web aplikaciji

4. Realizacija sistema

Kao što je već pomenuto, sistem se sastoji mikroservisa implementiranih u Python radnom okviru Django, baza podataka kojim prethodno pomenuti servisi pristupaju, servisa API kapije koji pruža grafički interfejs za krajnjeg korisnika, RabbitMQ komponente koja implementira međuservisnu komunikaciju i Adminer servisa koji omogućuje manipulaciju sa samim podacima. Svaka od prethodno pomenutih komponenti je kontejnerizovana i izvršava se na pojedinačnom Docker kontejneru.

4.1. Implementacija servisa

Što se tiče implementacije servisa, radni okvir Django, svojim funkcionalnostima pokrio je većinu zahteva za rešenje problema.
Django projekat inicijalizuje se jednostavnom komandom koja daje početni kostur aplikacije. Krajnja fajl struktura najbolje će biti ilustrovana „najkomplikovanijim“ od servisa implementiranih u Django-u, servisa API kapije:



Slika 4.1.1. primer fajl strukture Django projekta

Kao relevantni fajlovi ističu se:

- **settings.py** – u njemu su definisana sva osnovna podešavanja Django aplikacije. Najvažnije podešavanje u kontekstu problema jeste konekcija ka bazi, u slučaju mikroservisa:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        "NAME": config('DATABASE_ORDER_NAME'),
        "USER": config('DATABASE_ORDER_USER'),
        "PASSWORD": config('DATABASE_ORDER_PASSWORD'),
        "HOST": config('DATABASE_ORDER_HOST'),
        "PORT": config('DATABASE_ORDER_PORT'),
    }
}
```

Segment koda 4.1.1. Konfiguracija baze

Takođe, u slučaju servisa API kapije, bilo je potrebno definisati putanje ka statičkim (CSS) fajlovima:

```
STATIC_URL = 'static/'

STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

Segment koda 4.1.2. Konfiguracija statičkih fajlova

- **models.py** – u njemu su definisani **ORM**[18] modeli podataka. Primer koda definicije modela narudžbine:

```
from django.db import models

class Order(models.Model):
    product_id = models.PositiveBigIntegerField()
    quantity = models.PositiveIntegerField()
    price = models.PositiveIntegerField()
```

Segment koda 4.1.3. Model narudžbine

[18] Objektno-relaciano mapiranje predstavlja tehniku mapiranja klase na relacioni model tabele u bazi podataka, pritom omogućujući manipulaciju podacima kroz objektno-orijentisani interfejs.

- **serializers.py** – u njemu su definisani *serializer*-i[19] za potrebe (de)serijalizacije modela prilikom njihovog dohvatanja iz baze ili perzistiranja u istu, kao i za potrebe *custom* validacije:

```
from rest_framework import serializers
from .models import Invoice

class InvoiceSerializer(serializers.ModelSerializer):
    class Meta:
        model = Invoice
        fields = '__all__'

class InvoiceStatusSerializer(serializers.Serializer):
    status = serializers.ChoiceField(choices=Invoice.StatusChoices.choices)

class OrderIdsSerializer(serializers.Serializer):
    order_ids = serializers.ListField(child=serializers.IntegerField())
```

Segment koda 4.1.4. Serializeri finansijskog servisa

U datom primeru **InvoiceSerializer** predstavlja serializer za model fakture, dok **InvoiceStatusSerializer** i **OrderIdSerializer** predstavljaju serializere koji pre svega služe za validaciju polja nestandardnih CRUD zahteva.

- **views.py** – u njemu je definisana logika dohvatanja odgovarajućih podataka od modela na osnovu zahteva:

```
from rest_framework import generics
from .models import Product
from .serializers import ProductSerializer, ProductQuantitySerializer
from rest_framework.response import Response
from rest_framework import status
from rest_framework.views import APIView


class ProductListCreateView(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

class ProductRetrieveUpdateDestroyView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

4.1.5. Ugrađeni Django API veiw-ovi

Zgodna funkcionalnost Django-a jestu ugrađeni generički API *view*-ovi koji pružaju već

[19] *Serializer* predstavlja softversku komponentu zaduženu za pretvaranje struktura podataka u format pogodan za prenos, a posle i pretvaranje tih formatiranih podataka u originalan oblik.

gotovu logiku za CRUD operacije nad konkretnim modelom.

- **urls.py** – u njemu su date definicije putanja do odgovarajućih *endoint*-ova aplikacije:

```
from django.contrib import admin
from django.urls import re_path
from .views import OrderView, ProductView, OrderCheckoutView, OrderCreateView

urlpatterns = [
    re_path("^(orders)?/?$", OrderView.as_view(), name="order-dashboard"),
    re_path(r"^products/?$", ProductView.as_view(), name="product-dashboard"),
    re_path(r"^orders/create/?$", OrderCreateView.as_view(), name="order-
create"),
    re_path(r"^orders/checkout/(?P<order_id>\d+)/?$",
OrderCheckoutView.as_view(), name="order-checkout"),
]
```

Segment koda 4.1.6. URL šabloni endpoint-ova u servisu narudžbina

Za definisanje putanja *endpoint*-ova korišćeni su jednostavi regularni izrazi.

- **static > styles direktorijum –** u njemu se nalazi CSS za *front end* aplikacije
- **template direktorijum i poddirektorijumi** – u njemu se nalaze DTL HTML *template* fajlovi koji služe za prikaz informacija na front end-u
- **fajlovi u test direktorijumu** – sadrže *unit*/funkcionalne testove aplikacije

Princip rada DTL-a dobro je ilustrovan sledećim primerom iz rešenja – prilikom prikazivanja narudžbina korisniku API kapije, narudžbine se dohvataju zahtevom ka servisu narudžbina, a zatim, kada budu primljene u odgovoru, prosleđuju se šablonu za prikaz narudžbina:

Logika *view*-a za prikaz porudžbina:

```
class OrderView(APIView):
    def get(self, request, format=None):
        try:
            order_data = self.get_basic_order_data()
            invoice_data = self.get_invoices_for_orders([order_datum["id"] for
order_datum in order_data])

            data = self.merge_order_invoice_data(order_data, invoice_data)

            order_serializer = OrderSerializer(data, many=True)

            render_context = {'orders': order_serializer.data}
        except Exception as exception:
            render_context = {'error': str(exception)}
        finally:
            return render(request, 'app/order/order_dashboard.html',
render_context)
```

Segment koda 4.1.6. Implementacija view-a za prikaz narudžbina

Šablon za prikaz narudžbina (*order/order_dashboard.html*):

```
{% extends parent_template|default:"base.html" %}
{% load static %}
{% block title %}
    {{ block.super }} - Orders
{% endblock %}
{% block styles %}
<link rel="stylesheet" type="text/css" href="{% static 'styles/order.css' %}">
{% endblock %}
{% block content %}
{% if error %}
<h3 class="error">{{error}}</h3>
{% else %}
<h1>Orders:</h1>
<div class="item-list">
    {% for order in orders %}
        {% include 'app/order/order_detail.html' with order=order %}
    {% endfor %}
</div>
{% endif %}
{% endblock %}
```

Segment koda 4.1.7. Kod šablona liste narudžbina

Tempalte fajl *override*-uje vrednosti iz baznog template fajla (*base.html*) koji nasleđuje (mehanizam sličan nasleđivanju klasa u OOP), i za svaku narudžbinu uključuje šablon koji odgovara prikazu jedne pojedinačne narudžbine (*order/order_detail.html*):

```
<div class="item-container order-container">
    <div class="data id"><b>order #{{ order.id }}</b></div>
    <div class="data"><b>Product ID</b>: {{ order.product_id }}</div>
    <div class="data"><b>quantity</b>: {{ order.quantity }}</div>
    <div class="data"><b>price</b>: {{ order.price }}</div>
    {% if order.checkout_status != None %}
        <div class="data"><b>checkout</b>: {{ order.checkout_status }}</div>
    {% else %}
        <form action="{% url 'order-checkout' order_id=order.id %}"
method="post">
            <div class="data"><b>checkout</b>:
                    <button type="submit">checkout</button>
            </div>
        </form>
    {% endif %}
</div>
```

Segment koda 4.1.8. Kod šablona pojedinačne narudžbine

Ovaj template koristi vrednosti polja prosleđenog **order** objekta kako bi prikazao podatke vezane za datu narudžbinu.

4.2. Testiranje servisa

Testiranje servisa sprovedeno je pisanjem **jediničnih** (eng. ***unit***) testova nad klasama modela, *serializer*-a, i *view*-a za sve servise implementirane u Django-u. Bitno je naglasiti da se testovi za *view*-ove mogu nazvati i **funkcionalnim** jer pored same logike *view* klasa u *test case*-eve uključena je i logika rutiranja.

Testovi se pokreću tako što se preko Docker-a pristupi kontejneru servisa koji želimo da testiramo, i zatim izvrši komanda za testiranje Django aplikacije:

- projekat se otvori u konzoli
- izvrši se **`docker ps`** komanda – rezultat ove komande je ispis svih kontejnera koji trenutno postoje na Docker engine-u
- kopira se vrednost identifikatora servisa kome želimo da pristupimo
- izvrši se komanda **`docker exec -it {id} bash`** – ovom komandom pristupamo kontejneru
- sada, kada smo u kontejneru, potrebno je izvršiti komandu **`python manage.py test`** – ovom komandom pokreću se svi testovi definisani za dati projekat

Za pisanje testova korišćena je Django-ova ugrađena *test* biblioteka. *Test case*-evi koji testiraju određenu funkcionalnost se pišu kao metode klase koja predstavlja *test case* klasu za tu funkcionalnost. Ova klasa nasleđuje django.test.*TestCase* klasu, koja pruža osnovne, često korišćene funkcionalnosti potrebne za testiranje koda. Između ostalih *TestCase* klasa poseduje *setUp* metodu, u koju se piše kod koji je potrebno izvršiti pre izvršenja svakog pojedinačnog test case-a. U rešenju je ona korišćena za instanciranje *rest_framework.test.APIClient* klase, korišćene za slanje zahteva tokom testiranja pojedinačnih *view* funkcionalnosti:

```
from rest_framework.test import APIClient

def setUp(self):
    self.client = APIClient()
```

Segment koda 4.2.1. Primer koda SetUp metode

Primer jednog od testova modela narudžbine:

```
def test_update_order_ok(self):
    updated_quantity = 5
    self.order.quantity = updated_quantity

    self.order.save()

    updated_order = Order.objects.get(pk=self.order.pk)
    self.assertEqual(updated_order.quantity, updated_quantity)
```

Segment koda 4.2.2. Primer test case-a modela

Test proverava da li je, po ažuriranju vrednosti kvantiteta određene narudžbine, ta nova vrednost stvarno perzistirana u bazu.

Primer jednog od testova serializer-a *InvoiceStatusSerializer*, vezanog za entitet fakture:

```
def test_invoice_status_serializer_invalid_string_value_status_fail(self):
    input_data = {
        'status': 'test'
    }

    serializer = InvoiceStatusSerializer(data=input_data)

    self.assertFalse(serializer.is_valid())
    self.assertIn('status', serializer.errors)
```

Segment koda 4.2.3. Primer test case-a serializer-a

Test proverava da li *serializer* stvarno validira vrednost polja status (koje po zahtevu može imati vrednost jednaku samo jednoj od tri vrednosti: *waiting*, *incomplete*, *complete*).

Primer jednog od testova *view*-a vezanog za *endpoint* kojim se inkrementira/dekrementira vrednost zaliha proizvoda za prosleđenu vrednost:

```
def test_product_update_stock_non_integer_quantity_fail(self):
    data = {
        'quantity': 'test'
    }

    response = self.client.put(reverse('product-update-stock', kwargs={'pk':
self.product['id']}), data, format='json')
    self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

Segment koda 4.2.4. Primer test case-a view-a

Test proverava da li se za nevalidan unos dobija odgovor odgovarajućeg koda greške.

Specifičan primer testa jesu testovi nad *view* klasama servisa API kapije. Naime, ovaj servis u svom *view* kodu, da bi dohvatio narudžbine i proizvode za prikaz, mora poslati HTTP zahtev servisu narudžbina i servisu skladišta, respektivno. Tokom testiranja ovih funkcionalnosti potrebno je *mock*-ovati slanje zahteva nekom drugom servisu, odnosno definisati šta bi taj poziv trebalo da vrati (koji statusni kod i koji sadržaj tela odgovora). Kod koji se testira je npr. kod *get* metode *ProductView* klase koji treba da pošalje zahtev za dohvatanje svih proizvoda servisu skladišta. U te svrhe koristi se *get* metoda *requests* biblioteke.

```
class ProductView(APIView):
    def get(self, request, format=None):
        try:
            api_url = config('GATEWAY_GET_PRODUCTS_ENDPOINT_URL')
            response = requests.get(api_url)
            data = response.json()

            product_serializer = ProductSerializer(data, many=True)

            render_context = {'products': product_serializer.data}
        except Exception as exception:
            render_context = {'error': str(exception)}
        return render(request, 'app/product/product_dashboard.html', ren-
der_context)
```

Segment kdoa 4.2.5. Implementacija view-a za prikaz proizvoda servisa API kapije

Tokom testiranja funkcionalnosti *get* metoda *requests* biblioteke se *mock*-uje na sledeći način:

```
from unittest.mock import patch, Mock

def setup_mock_send_request(mock_send_request, response_data=None, ok=True):
    mock_response = Mock()
    mock_response.ok = ok
    if response_data is not None:
        mock_response.json.return_value = response_data
    mock_send_request.return_value = mock_response

@patch('app.views.requests.get')
def test_get_products_api_error_fail(self, mock_send_request):
    setup_mock_send_request(mock_send_request, ok=False)

    response = self.client.get(reverse('product-dashboard'))

    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertTemplateUsed(response, 'app/product/product_dashboard.html')
    self.assertIn(b'error', response.content)
```

Segment koda 4.2.6. Primer mock-ovanja zavisnosti

U svrhe *mock*-ovanja iznad *unit* testa u čijem kontekstu želimo da *requests.get* metod bude *mock*-ovan stavlja se *patch* anotacija[20] – ovime se deklariše da će, kada prethodno pomenuti metod bude pozvan, kao odgovor biti vraćena vrednost *return_value* polja *mock_send_request* objekta, koji je inače prosleđen *test case* metodi. Pomoću pomoćne *setup_mock_send_request* funkcije se stavljaju odgovarajuće vrednosti statusa i tela odgovora, u zavisnosti od toga koji deo logike proveravamo u pojedinačnom slučaju korišćenja.

[20] Anotacija predstavlja metapodatak koji se može dodati klasi, metodi ili polju, kako bi se pružile dodatne informacije kompajleru ili okruženju izvršavanja.

4.3. Implementacija komunikacije

Tok RabbitMQ komunikacije započinje kada se servisu za narudžbine pošalje zahtev za *checkout*-om određene narudžbine. Servis za narudžbine tada dovlači potrebne informacije za datu narudžbinu, formatira poruku koju je potrebno poslati, i incijalizuje potrebne RabbitMQ komponente kroz sledeći kod:

```
connection = pika.BlockingConnection(pika.ConnectionParameters(host=con-
fig('RABBITMQ_HOST')))
channel = connection.channel()

channel.exchange_declare(exchange=EXCHANGE_ORDER_NAME, exchange_type='direct',
durable=False, auto_delete=False)
channel.queue_declare(queue=QUEUE_ORDER_NAME, durable=False,
auto_delete=False)
channel.exchange_declare(exchange=EXCHANGE_FINANCIAL_NAME,
exchange_type='direct', durable=False, auto_delete=False)
channel.queue_bind(queue=QUEUE_ORDER_NAME, exchange=EXCHANGE_FINANCIAL_NAME,
routing_key='ORDER_CHECKOUT_PROCESSED')
```

Segment koda 4.3.1. Inicijalizacija RabbitMQ instanci u servisu narudžbina

Servis narudžbina prvo uspostavlja konekciju sa RabbitMQ instancom. Zatim deklariše razmenu narudžbina kao i red za primanje poruka. Instanca razmene i reda se kreira samo na prvi zahtev za *checkout*-om narudžbine za jednu celokupnu instancu sistema. Nakog toga, na svaki zahtev za *checkout*-om koriste se već postojeće instance prethodno pomenutih komponenti.

Zatim servis za narudžbine šalje poruku o *checkout*-u narudžbine na razmenu narudžbina i sinhrono se blokira dok ne dobije odgovor na svom redu:

```
# publish order checkout message
channel.basic_publish(exchange=EXCHANGE_ORDER_NAME,
routing_key='ORDER_CHECKOUT_START', body=message)

result = None
# define on message callback
def on_message_callback(channel, method, properties, body):
    nonlocal result
    message_body = json.loads(body)

    result = None if not message_body["error"] else message_body["error"]

# wait for response (blocking)
channel.basic_consume(queue=QUEUE_ORDER_NAME,
on_message_callback=on_message_callback, auto_ack=True)

try:
    channel.connection.process_data_events(
        time_limit=config('ORDER_CHECKOUT_ENDPOINT_TIME_LIMIT', cast=int)
    )
except pika.exceptions.AMQPError as exception:
    print(f"AMQP error: {exception}")
```

Segment koda 4.3.2. Sinhrono čekanje odgovora

Kada odgovor stigne na red servisa za narudžbine okida se *callback* funkcija koja na osnovu odgovora dobijenog od RabbitMQ instance, generiše odgovor koji će poslati pošiljaocu zahteva za *checkout* order-a.

Što se tiče finansijskog servisa i servisa skladišta, njihova komunikaciona logika biće ilustrovana kroz kod *worker*-a finansijskog servisa:
Prilikom samog podizanja kontejnera servisa, pokreće se njegov *worker*, koji uspostavlja konekciju sa RabbitMQ instancom i inicijalizuje potrebne RabbitMQ komponente:

```
process_invoice()

def process_invoice():
    # create connection and channel
    connection =
pika.BlockingConnection(pika.ConnectionParameters(host=config('RABBITMQ_HOST')
))
    channel = connection.channel()

    # declare exchanges and queues
    channel.exchange_declare(exchange=consts.EXCHANGE_ORDER_NAME,
exchange_type='direct', durable=False, auto_delete=False)
    channel.queue_declare(queue=consts.QUEUE_FINANCIAL_NAME, durable=False,
auto_delete=False)
    channel.exchange_declare(exchange=consts.EXCHANGE_FINANCIAL_NAME,
exchange_type='direct', durable=False, auto_delete=False)
    channel.queue_bind(queue=consts.QUEUE_FINANCIAL_NAME,
exchange=consts.EXCHANGE_ORDER_NAME, routing_key='ORDER_CHECKOUT_START')
```

Segment koda 4.3.3. Inicijalizacija RabbitMQ instanci u worker-u finansijskog servisa

Nadalje, *worker* sve vreme čeka poruku na svom redu i, kada poruka stigne, okida *callback* funkciju koja rukuje porukom na odgovarajući način, u zavisnosti od tipa primljene poruke:

```
    # consume messages
    try:
        channel.basic_consume(
            queue=consts.QUEUE_FINANCIAL_NAME,
            on_message_callback=on_message_callback,
            auto_ack=True
        )
        channel.start_consuming()
    except pika.exceptions.AMQPError as exception:
            print(f"AMQP error: {exception}", flush=True)
    finally:
        # free resources
        channel.close()
        connection.close()

# function called when message received
def on_message_callback(channel, method, properties, body):
    message_body = json.loads(body)
    message_type = message_body['data_type']

    # handle specific message types
    match message_type:
        case consts.MESSAGE_TYPE_ORDER_CHECKOUT:
            handle_order_checkout(message_body, channel)
        case consts.MESSAGE_TYPE_STOCK_CONFIRM:
            handle_stock_confirmation(message_body, channel)
```

Segment koda 4.3.4. Definisanje callback funkcije za prijem poruke i sinhrono čekanje na poruku

Worker servisa skladišta radi analogno goreopisanom *worker*-u, razlikovajući se u logici rukovanja porukom.

4.4. Kontejnerizacija servisa

Svaki od mikroservisa razdvojen je u poseban direktorijum unutar korenog direktorijuma projekta, ima svoj Dockerfile i listu svojih zavisnosti. Ovo pruža mogućnost budućeg razdvajanja izvornog koda na odvojene repozitorijume, što ilustruje mikroservisnu prirodu rešenja i omogućuje buduću totalnu nezavisnu refaktorizaciju pojedinih mikroservisa.
Na primeru Dockerfile-a servisa za narudžbine biće objašnjena logika građenja kontejnera servisa:

```
FROM python:3.10 AS builder

ENV VIRTUAL_ENV=/venv
RUN python3 -m venv $VIRTUAL_ENV
ENV PATH="$VIRTUAL_ENV/bin:$PATH"

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.10-slim

ENV VIRTUAL_ENV=/venv
ENV PATH="$VIRTUAL_ENV/bin:$PATH"

WORKDIR /app
COPY --from=builder $VIRTUAL_ENV $VIRTUAL_ENV
COPY --from=builder /app /app

EXPOSE 8000

CMD ["bash", "./initialize.sh"]
```

Segment koda 4.4.1. Primer Dockerfile-a

Naime građenje *image*-a servisa započinje, prvim ***stage***-om[21] korišćenjem privremenog *build*-a u kome se samo instaliraju sve potrebne Python zavisnosti servisa. Zatim se, u drugom *stage*-u *build*-ovanja *image*-a koristi izvorni kod dovučenih zavisnosti i dodatno se u memorijski prostor *image*-a kopira izorni kod samog servisa. Na kraju se port 8000 čini javno dostupnim, čime se omogućuje pristup kontejneru iz spoljašnosti po tom portu, i na kraju se pokreće inicijalizaciona skripa koja izvršava inicijalnu Django migraciju podataka[22] i pokreće Django server.

Što se tiče **docker compose** fajla, specifičnosti vredne pomena su:

[21] *Multi-stage build* mehanizam može se koristiti u procesu kreiranja kontejnera kako bi se stvorio lakši, efikasniji kontejner. U praksi se često u jednoj fazi dovuku sve zavisnosti potrebne za rad (prvo je potrebno preuzeti mehanizam kojim se zavisnosti uopšte dovlače), a zatim se u kasnijoj fazi već dovučene zavisnosti samo kopiraju u kontejner koji je najlakši mogući – ovime se na kraju dobija lakši kontejner nego da je korišćen samo jedan stage za građenje kontejnera.

[22] U kontekstu radnik okvira sa ORM-om migracija podataka odnosi se na proces menjanja baze podataka u skladu sa promenama nad OOP reprezentacijom modela, usput čuvajući već postojeće podatke i osiguravajući konzistentnost strukture baze.

- podrazumevano svi kontejneri definisani u jednom docker compose fajlu i pokrenuti preko ovog alata, prikačeni su na jednu virtuelno mrežu kako bi mogli da međusobno komuniciraju
- za kontejnere baza podataka definisani su ***volume***-i kojima se hard-disk memorija kontejnera mapira na hard-disk memoriju *host* sistema; ovime se omogućuje trajna perzistencija podataka sa kontejnera i sprečava se gubitak istih podataka prilikom gašenja kontejnera (podrazumevano, kada se kontejner ugasi, njegova memorija biva *reset*-ovana)
- za kontejnere mikroservisa odrađeno je mapiranje izvornog koda sa kontejnera na izvorni kod na *host* sistemu – ovime je omogućeno ažuriranje izvornog koda u kontejneru u skladu sa ažuriranjem izvornog koda od strane *developer*-a, tokom razvoja, u realnom vremenu; ova funkcionalnost ubrzava rad jer nije potrebno posle neke promene izgraditi kontejner servisa opet, kako bi se ta promena videla
- zavisnost između kontejnera – korišćenjem **depends_on** i **condition** polja osigurano je da kontejneri koji zavise od drugih kontejnera ne počinju sa podizanjem dok ne stigne obaveštenje da je kontejner od koga zavise završio sa podizanjem – ovo je odrađeno kako bi se izbegli nepotrebni konflikti prilikom podizanja sistema kada bi komponenta sistema tražila resurs vezan za drugu komponentu koja još nije podignuta (npr. opsluživanje zahteva za dohvatanjem narudžbina na servisu za narudžbine ne uspe jer konekcija ka bazi podataka narudžbina nije uspostavljena jer baza nije završila sa inicijalizaciojm)

5. Zaključak

Cilj ovog rada jeste da se ilustruje koncept mikroservisne arhitekture kroz vrlo jednostavan slučaj korišćenja. Usput je demonstriran RabbitMQ, kao konkretno rešenje za problem asinhrone komunikacije između distribuiranih sistema i Docker, moćan alat koji olakšava i ubrzava razvoj, usput čineći *deployment* na produkciju efikasnijim.

Zbog konceptualne prirode rešenja, ostavljen je prostor za brojna buduća poboljšanja, od kojih su neka:

- usložnjavanje modela – što se tiče trenutne poslovne logike i skupa modela koji je čine, sistem se čini kao deo jednog većeg sistema, gde može postojati više izvora promena nego što je to sada slučaj; na primer, ne postoji entitet kupca/korisnika, sva 3 entiteta se trenutno menjaju samo ručno kroz odgovarajući API, ili kroz bazu podataka
- u duhu prethodne tačke, s obzirom da ne postoje ograničenja ključeva na nivou relacije u bazi podataka, jer su entiteti smešteni u različite baze podataka, trenutno ne postoji nikakva poslovna validacija – prilikom kreiranja narudžbine ne proverava se da li proizvod sa datim identifikatorom zapravo postoji, isti je slučaj za fakturu i narudžbinu
- trenutno nije implementiran oporavak od otkaza na mreži, tokom komunikacije, ili na samoj bazi podataka, prilikom rada sa njom; takođe, nije implementirano čuvanje i kasnije rukovanje neuspelim porukama na nivou RabbitMQ-a (moguće implementirati *dead-letter* razmenama i dodatnim konfiguracijama sistema)
- od testova, trenutno postoje jedinični/funkcionalni testovi klasa Django aplikacija, ali ne postoje *end-to-end* testovi realnih slučajeva korišćenja koji bi testirali samu komuniaciju između servisa
- radi daljeg razvoja sistema, zbog njegove mikroservisne arhitekture, preporučljivo je deljenje repozitorijuma na više repozitorijuma, za svaki mikroservis po jedan – to bi omogućilo totalnu nezavisnost servisa jednog od drugog, olakšalo podelu posla i omogućilo da se možda tech stack pojedinačnog servisa promeni sa Python-a i Django-a na neku potpuno drugu tehnologiju
- u trenutnom rešenju komunikacija između finansijskog i servisa skladišta jeste asinhrona, ali servis narudžbina sinhrono čeka na odgovor od ostatka sistema, što u realnosti može da potraje – *endpoint checkout*-a narudžbine mogao bi da se refaktoriše da korisniku odmah po slanju poruke na razmenu narudžbina vrati pošiljaocu zahteva 204 HTTP kod, i da na pošiljaocu bude da naknadno *poll*-uje sistem o stanju fakture narudžbine

6. Literatura

[1] Microservices.io, https://microservices.io/

[2] Microsoft microservice dokumentacija, https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[3] Message-oriented middleware, https://courses.ischool.berkeley.edu/i206/f97/GroupB/mom/what_is_mom.html

[4] Materijali sa predmeta “Principi softverskog inženjerstva” na Elektrotehničkom fakultetu, http://si3psi.etf.rs/

[5] Python dokumentacija, https://docs.python.org/3/

[6] Django dokumentacija, https://docs.djangoproject.com/en/4.2/

[7] Pika dokumentacija https://pika.readthedocs.io/en/stable/

[8] Unittest dokumentacija, https://docs.python.org/3/library/unittest.html

[9] Dockerfile reference, https://docs.docker.com/engine/reference/builder/

[10] Docker compose reference, https://docs.docker.com/compose/compose-file/compose-file-v3/

[11] AMQP dokumentacija, https://www.amqp.org/

[12] RabbitMQ dokumentacija, https://www.rabbitmq.com/documentation.html

[13] MySQL dokumentacija, https://dev.mysql.com/doc/

[14] Adminer dokumentacija, https://www.adminer.org

[15] Postman dokumentacija, https://learning.postman.com/docs/introduction/overview/

[16] MDN documentacija, https://developer.mozilla.org/en-US/

[17] CSS tricks, https://css-tricks.com/snippets/css/a-guide-to-flexbox/

[18] Materijali sa predmeta *Veb dizajn* na Elektrotehničkom fakultetu, https://rti.etf.bg.ac.rs/rti/si2vd/