UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET


Aleksa Kojadinović

# ARHITEKTURA I DIZAJN VEB PLATFORME ZA UPRAVLJANJE KORISNIČKIM ŽALBAMA I ZAHTEVIMA

master rad

Beograd, 2024.

**Mentor**:

Vladimir Filipović

**Članovi komisije**:

Saša Malkov

Aleksandar Kartelj

**Datum odbrane**: xx. januar xxxx.

**Naslov master rada**: Arhitektura i dizajn veb platforme za upravljanje korisničkim žalbama i zahtevima

**Rezime**: Razvoj veb aplikacija u današnje vreme jedna je od najrasprostranjenijih grana softverskog inženjerstva. Moguće je implementirati veoma kompleksne softverske sisteme kao veb servise. Izbor veba nosi kako pogodnosti tako i izazove. Ovaj rad ima za cilj implementaciju kompleksne veb platforme za rešavanje konkretnog poslovnog problema - asinhrone korisničke podrške u vidu tiketa. Pritom se razmatra korišćenje modernih veb tehnologija zasnovanih na Node.js ekosistemu. Rad prikazuje proces dizajna domenskog modela putem ER dijagrama i DDD principa, a zatim i kompletnu implementaciju serverskog i klijentskog dela aplikacije u okruženjima NestJS i Next.js. Osim domenskog razvoja razmatra se i izolacija okruženja koristeći sistem za virtuelizaciju Docker. Rad obuhvata i prikaz implementacije minimalnog analitčkog servisa u jeziku *python* uključujući ETL procese i vizuelizaciju.

**Ključne reči**: veb, DDD, servisi, API, analitika

# Sadržaj

# Glava 1

# Uvod

Prilikom razvoja bilo kakvog poslovnog softvera namenjenog za korišćenje od strane velikog broja zaposlenih neke kompanije, u današnje vreme pribegava se implementaciji datog sistema kao jedne kompleksne veb platforme. Razlozi za izbor veba su mnogobrojni, uključujući:

- **Univerzalnost veb pregledača kao platforme na kojoj se izvršava aplikacija.** U kontekstu vizuelnog dela aplikacije, prilikom razvoja tradicionalnih desktop aplikacija potrebno je voditi računa o mogućnosti pokretanja na različitim platformama i arhitekturama, kao i restrikcija pojedinih sistema. Razlike između veb pregledača, iako postoje, jesu umanjene[1].
- **Lakše ažuriranje aplikacije.** Ažuriranje desktop aplikacije često zahteva korisničku akciju u vidu nekakve instalacije, tako da implementacija ovakvog sistema nije trivijalna. U slučaju veba ovo je značajno olakšano zbog same prirode klijent-server ahitekture.
- **Integracije sa eksternim servisima.** Uveliko ustanovljeni veb standardi, kao što je REST API[2] doveli su do olakšane međusobne integracije veb platformi.

Veb je od svog originalnog oblika, koji se zasnivao na prikazivanju i povezivanju statičkih HTML stranica, evoluirao do kompleksne platforme za razvoj softvera. Iz tog razloga vremenom se javljala potreba za formalizovanjem principa razvoja softvera za veb, baš kao i za tradicionalne aplikacije [1]. Za ove potrebe nastali

[1] I u situacijama kada su značajne postoje alati za njihovo prevazilaženje, kao što je *corejs*.
[2] O REST API-u će biti reči u sekciji 3.5.

su mnogi standardi i okviri za uspešan razvoj veb aplikacija, koji se vremenom pokazuju kao dobre prakse uprkos konstantnom menjaju konkretnih tehnologija i trendova.

Ovaj rad predstaviće razvoj kompleksne veb platforme za upravljanje korisničkim žalbama i zahtevima. Pod kompleksnošću platforme primarno podrazumevamo tehničku kompleksnost - mogućnosti sistema iz ugla korisnika su ograničene, ali način na koji je sistem dizajniran omogućava njegovo lako proširenje. Dobri standardi i prakse razvoja veb aplikacija često se vezuju za tradicionalne platforme, međutim cilj ovog rada jeste da predstavi njihovu upotrebljivost i u modernim bibliotekama i okruženjima.

Poglavlje 2 najpre će prikazati sistem STS iz ugla korisnika, kako bi problematika realnog sveta koju rešavamo bila jasna, dok se naredni deo bavi projektovanjem sistema putem modela entiteta i odnosa.

Treće poglavlje bavi se serverskim delom aplikacije, odnosno bekendom, primenom domenskog dizajna i pregledom svakog od slojeva serverske aplikacije.

U poglavlju 4 fokus se prebacuje na klijentski deo aplikacije, gde se osim prikaza rada aplikacije osvrćemo na kratak istorijat primene jezika Javascript, kao i neke druge aspekte razvoja klijentskih aplikacija.

Naredno poglavlje prikazuje implementaciju minimalnog sistema za analitiku, uz propratne ETL procese i vizuelizaciju.

Pretposlednje poglavlje uvodi nas u postavku infrastrukture za razvoj, koncept veb servera i proksija, kao i kratak prikaz sistema Docker, kao neizostavnog alata za moderan veb razvoj.

Poglavlje 7 posvećeno je zaključku rada.

# Glava 2

# Sistem STS

Svaki biznis koji pruža neku vrstu usluge korisnicima neizostavno mora imati korisničku podršku u nekom obliku. Pre razvoja veb tehnologija telefonski sistemi bili su jedini način direktne podrške koju korisnik može da dobije. Razvoj i pristupčanost veba doveli su do evolucije sofisticiranih onlajn proizvoda koji rešavaju problem korisničke podrške, kako putem pisanih rasprava u vidu tiketa, tako i putem tradicionalnih telefonskih linija uz određenu integraciju[1].

## 2.1 Sistem STS iz ugla korisnika

Sistem STS[2], razvijan za potrebe ovog rada, obuhvata osnovnu funkcionalnost za pružanje podrške u vidu tiketa. Tiket predstavlja jednu tekstualnu korisničku žalbu ili zahtev, na kojoj se može pokrenuti diskusija, a sve sa ciljem asinhronog[3] razrešenja korisničkog problema. Glavni akteri sistema su:

- **Klijenti** - koji mogu postavljati tikete u cilju dobijanja podrške.
- **Agenti** - koji su zaduženi da odgovaraju na tikete, upravljaju njihovim statusima i rešavaju probleme.
- **Administratori** - zaduženi za podešavanje sistema, dodeljivanje uloga korisnicima, zabranu pristupa korisnicima, ili uopšteno bavljenje kompleksnim problemima.

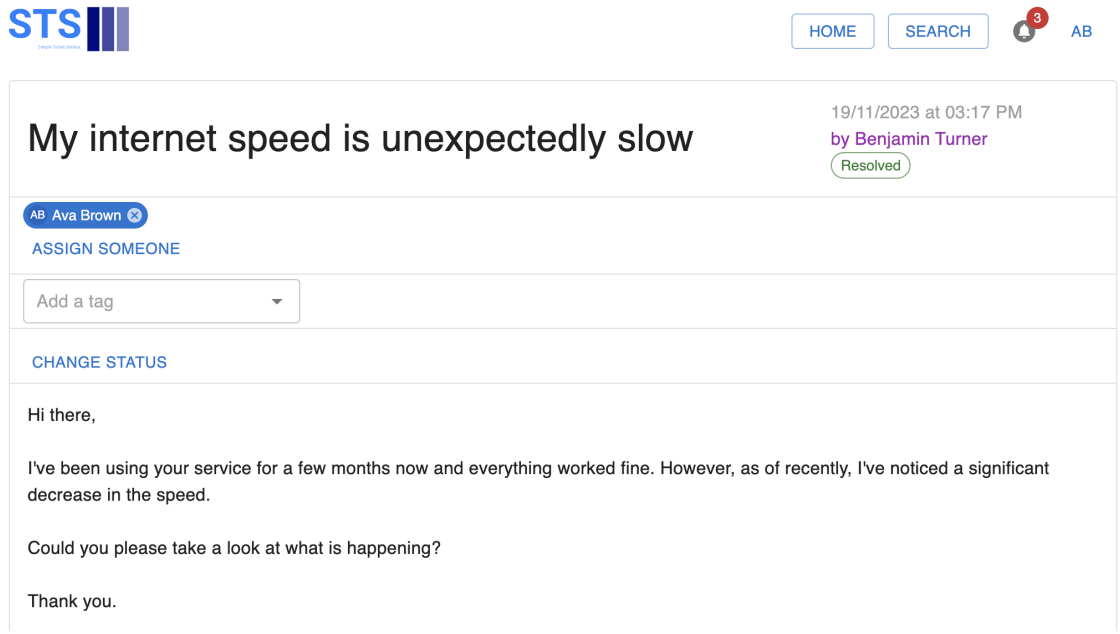
[1] Zendesk, 8x8, Jira Service Management.

[2] Skraćeno od eng. Simple Ticket Service, jednostavan servis za tikete.

[3] Pod asinhronošću podrazumevamo suprotnost u odnosu na telefonski sistem u kom korisnik i agent podrške vode komunikaciju u realnom vremenu.

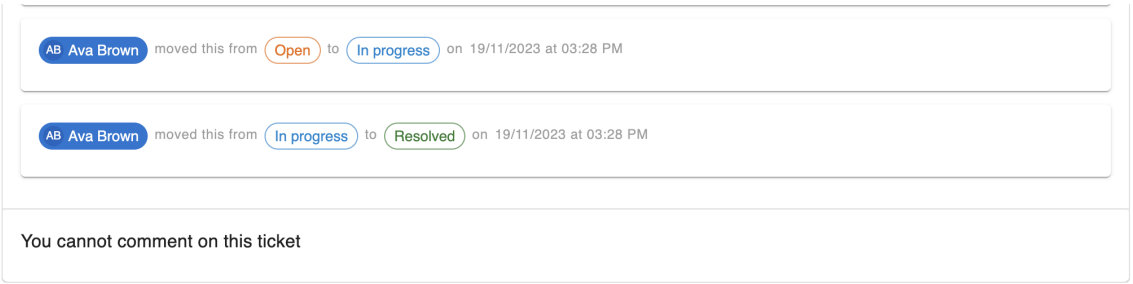
## Osnovne funkcionalnosti, prikaz tiketa, statusi

Osnovni pojam sistema, kao što je rečeno, jeste tiket. Na slici možemo videti primer zaglavlja jednog tiketa. Sastoji se primarno iz naslova i opisa, dok se mogu videti imena klijenta i trenutno zaduženog agenta.




Slika 2.1: Primer tiketa.

Tiket može prolaziti kroz više statusa. Promene tih statusa vidljive su kako agentima tako i korisnicima.



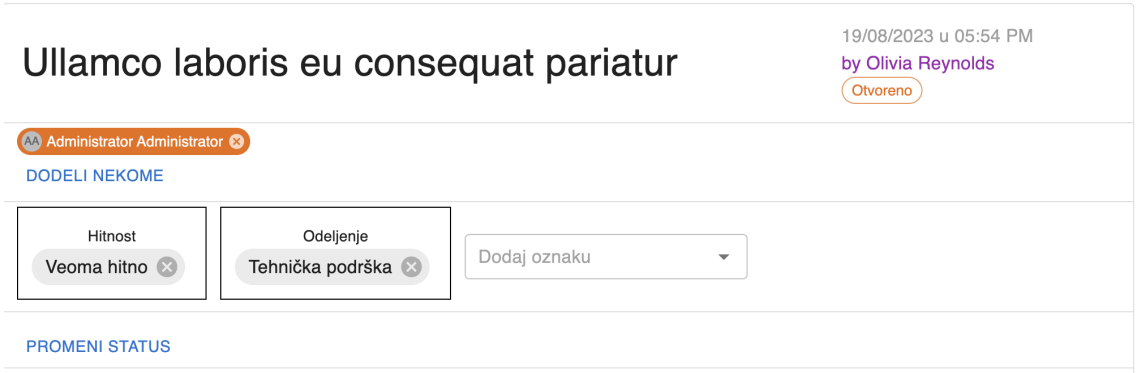

Slika 2.2: Promene statusa.

## Sistem oznaka, konfigurabilnost

Često postoji potreba klasifikacije tiketa po raznim kriterijumima - departman unutar firme koji se bavi tiketom, hitnost, prioritetnost i slično. Kako bi se posti-

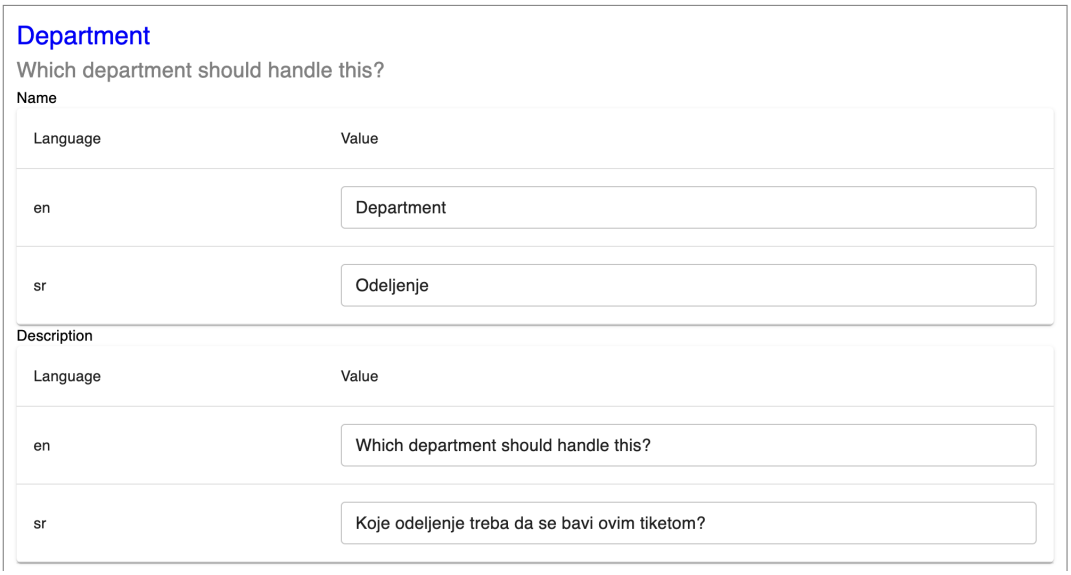
gla maksimalna fleksibilnost uveden je sistem oznaka (eng. tags). Oznaka se može nakačiti na dati tiket i može dati brzu informaciju o nekim njegovim aspektima. Ovaj sistem u potpunosti je konfigurabilan od strane korisnika (u ovom slučaju administratora), tako da nije potrebna intervencija programera prilikom dodavanja novih ili promene postojećih oznaka.

Na sledećoj slici vidimo kako izgleda tiket označen oznakama histnosti i odeljenja.

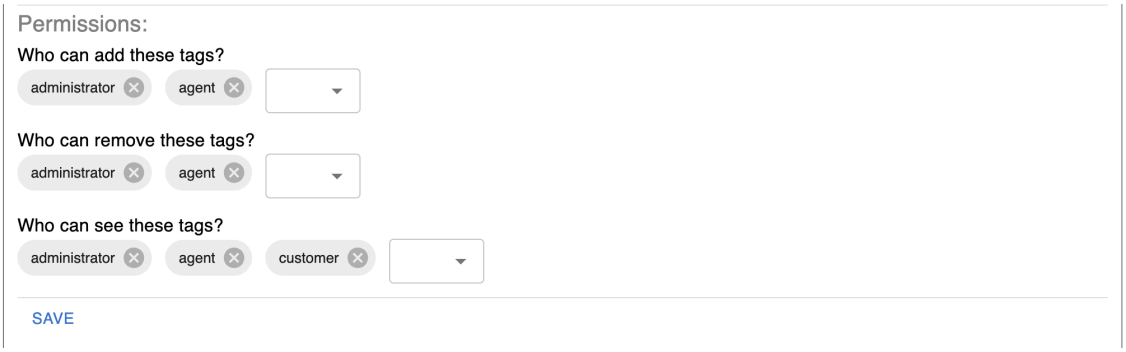



Slika 2.3: Tiket sa oznakama hitnosti i odeljenje.

Sledeće dve slike prikazuju kontrolnu tablu na kojoj administatori mogu upravljati oznakama. Primećujemo dodatnu fleksibilnost u sistemu kojom se može konfigurisati koji tip korisnika ima pravo da vidi, dodaje i uklanja koju oznaku. Takođe može se primetiti da sistem podržava višejezičnost, tako što se imena i opisi mogu podešavati i na srpskom i na engleskom jeziku. O jezicima i sistemu internacionalizacije biće više reči u sekciji 4.4.



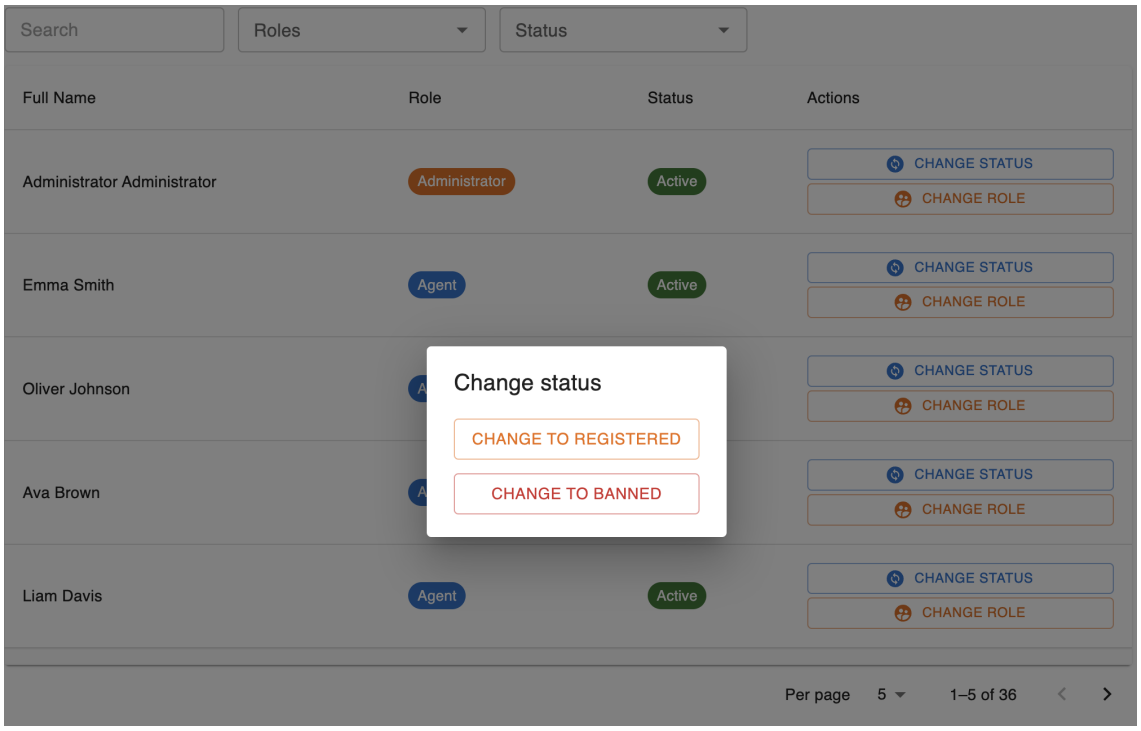

Slika 2.4: Kreiranje oznake, višejezička imena.




Slika 2.5: Kreiranje oznake, permisije.

## Upravljanje korisnicima

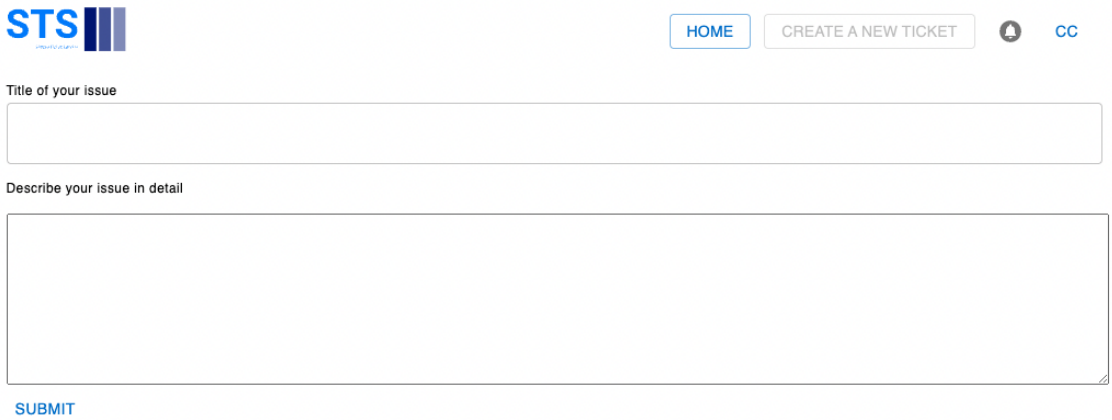
Administratori poseduju mogućnost upravljanja korisnicima sajta putem kontrolne table na adresi `manage/users`. Tabla prikazuje listu svih korisnika i akcije koje mogu biti primenjene nad njima. Od akcija podržana je promena tipa korisnika (pod određenim uslovima) i promena statusa korisnika.




Slika 2.6: Upravljanje korisnicima.

## Postavljanje tiketa

Na kraju, stranica na kojoj klijenti mogu postavljati nove tikete jednostavan je obrazac koji se sastoji iz naslova i tela tiketa.




Slika 2.7: Kreiranje tiketa.

## 2.2 Entiteti i njihovi odnosi, ER dijagram

ER model[4] predstavlja jedan vid konceptualnog modelovanja podataka u softverskom sistemu [2]. Sam po sebi je apstraktan pojam bez vidljive reprezentacije, s tim što se često poistovećuje sa pojmom *ER dijagrama* koji je notaciono sredstvo za prikaz modela nastalog ER modelovanjem.

Sastoji se od nekoliko vrsta elemenata:

- **Entiteti** predstavljaju stvari iz realnog sveta koje modelujemo. Jedna pojedinačna pojava entieta naziva se *instanca entiteta* [2]. ER dijagram koristi pravougaonike za oznaku entiteta.

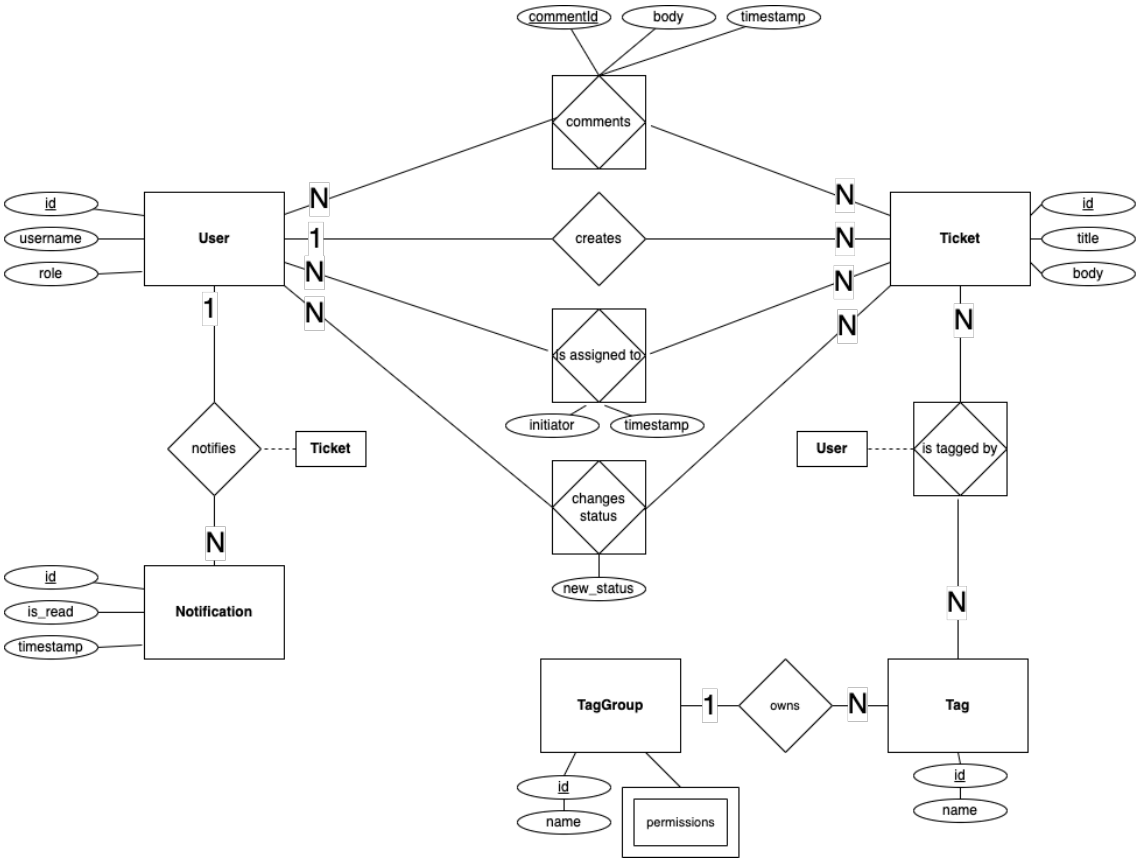
- **Odnosi** predstavljaju asocijacije između entiteta u najopštijem smislu. Karakteriše ih i brojnost koja pruža informaciju o broju instanci svakog entiteta u datom odnosu. Ovakav sistem poznat je i u modelovanju relacionih baza pod nazivima „1 na 1", „1 ka više" i „više ka više". ER dijagram koristi kvadrat za označavanje odnosa.

- **Atributi** jesu karakteristike entiteta ili odnosa koje pružaju dodatne detalje o njima. Atributi mogu biti ključni, u kom slučaju učestuvju u identifikaciji entiteta ili odnosa, ili deskriptivni, kada se odnose na nejedinstvene osobine [2]. Prikazuju se elipsama koje su strelicama povezane za entitet ili odnos kom pripadaju, dok se imena ključnih atributa podvlače.

ER modelovanje neretko se dovodi u vezu sa modelovanjem relacionih baza podataka. Model ipak nije ograničen na vrstu baze podataka, tako da ga koristimo i ovde uprkos tome što je baza sistema nerelaciona - konkretno MongoDB. Štaviše, proces transformacije ER modela u dokumentni model dodatno je olakšan, jer za određene asocijacije nije potrebno praviti dodatne tabele/dokumente usled mogućnosti čuvanja kompleksnih polja u dokumentima. Na narednom dijagramu prikazan je uprošćen[5] ER dijagram sistema STS.

[4] eng. Entity-relationship model - model entiteta i odnosa.

[5] Neki atributi nisu prikazani zbog kompaktnosti, neki odnosi korisnika i tiketa nisu prikazani jer su jednostavni (promena naslova, promena tela).




Slika 2.8: ER dijagram sistema STS.

## Odnos tiketa i korisnika, složeni odnosi

Primećujemo da dva glavna entiteta - tiket i korisnik - mogu biti u više različitih odnosa:

- Korisnik može kreirati više tiketa (odnos `creates`), dok tiket može biti kreiran od strane samo jednog korisnika.
- Korisnik može komentarisati na tiket (odnos `comments`). Primećujemo da ovaj odnos sadrži dodatne podatke, kao što su identifikator, telo i vreme komentara.
- Korisnik može biti dodeljen tiketu (odnos `is_assigned_to`), i tu dodelu može izvršiti drugi korisnik, označen atributom `initiator`.
- Korisnik može menjati status tiketa (odnos `changes_status`).

Kompleksni odnosi, kao što su komentarisanje i dodeljivanje, u relacionoj bazi podataka morali bi biti izvedeni odgovarajućim zasebnim veznim relacijama. Na primer, u slučaju komentarisanja, ako bi korisnik i tiket bili predstavljani sledećim relacijama (koristimo relacionu notaciju):

**Ticket**(id, title, body, status, creator_id)

**User**(id, username, role)

tada bi nam za komentar bila potrebna dodatna relacija koja sadrži identifikator tiketa i korisnika:

**Comment**(comment_id, user_id, ticket_id, body, timestamp)

Kako koristimo MongoDB i dokumentni model podataka, imamo mogućnost da komentare čuvamo u okviru dokumenta tiketa. Tačnije, u sloju baze podataka, ovi komentari se ne čuvaju eksplicitno, već kroz sistem istorije tiketa. Ukratko, svaka promena nad tiketom se pamti kao jedan događaj u nizu `history`, uz odgovarajući tip promene i neophodne podatke.

```
export class TicketHistoryItem {
  timestamp: Date;
  initiator: UserDb;
  type: TicketHistoryEntryType;
  payload:
    | TicketHistoryEntryCreated
    | TicketHistoryEntryStatusChanged
    | TicketHistoryEntryCommentAdded
    | TicketHistoryEntryTitleChanged
    | TicketHistoryEntryBodyChanged
    | TicketHistoryEntryAssigneesChanged
    | TicketHistoryEntryTagsChanged
    | TicketHistoryEntryCommentChanged
    | TicketHistoryEntryCommentDeleted;
}
```

Primer 2.1: Tip podataka istorije tiketa.

O ovom sistemu iz domenskog ugla biće više reči u sekciji 3.3. Svakako, ER model sam po sebi ne propisuje način realizacije odnosa, jer nije vezan ni za jedan model baze podataka.

## Oznake i grupe oznaka, slabi entiteti

Kao što je pomenuto u prethodnoj sekciji, uveden je fleksibilan sistem oznaka takav da je moguće upravljati samim sistemom bez intervencije programera. Oznake se mogu dodeljivati tiketu, a same mogu pripadati grupi oznaka. Jedan očigledan primer upotrebe ovog sistema je označavanje koliko je dati tiket hitan - grupa oznaka bila bi **hitnost**, dok bi konkretne oznake bile, primera radi, **veoma hitno**, **umereno hitno** i **nije hitno**.

Dodatni nivo fleksibilnosti postiže se ograničavanjem ko može koju akciju da vrši nad kojom grupom oznaka. Na primer, klijent koji postavlja tiket može videti kojom hitnošću je označeno njegovo pitanje, ali ne može sam označiti hitnost usled činjenice da se ne može smatrati objektivnim po tom pitanju. Ovim uvodimo još jedan element ER modela, prikazan na dijagramu pravougaonikom sa duplim ivicama, koji nazivamo **slab entitet**[6]. Uglavnom nastaju od kompleksnih atributa, odnosno atributa koji se ne mogu izraziti jednom vrednošću[7] i odlikuje ih nedostatak sopstvenog identiteta - identitet slabog entiteta je „pozajmljen" od strane jakog entiteta kome pripada [2]. U ovom slučaju, slab entitet **dozvola** (eng. permissions) sastoji se od sledećih podataka:

- lista uloga korisnika koji mogu da vide oznake iz date grupe
- lista uloga korisnika koji mogu da dodaju oznake iz date grupe
- lista uloga korisnika koji mogu da uklanjaju oznake iz date grupe

U domenskom sloju softverskog sistema, o kome će biti reči u sekciji 3.1, slabi entiteti se najčešće preslikavaju u pojam *vrednosnih objekata*[8], dok se u kôdu realizuju putem običnih klasa[9].

[6] eng. weak entity

[7] Knjiški primer slabog entiteta je adresa - sastoji se od ulice i broja. Ipak, postoje situacije gde adresa može biti i (jak) entitet.

[8] eng. value objects

[9] U praksi postoji konvencija nazivanja takvih objekata kao *Plain Old* (stari dobri) objekti. Tako u slučaju jezika Javascript nazivaju se POJO (Plain old Javascript Object), dok bi za PHP bili POJO, i sl.

# Glava 3

# Bekend

Kod velikog broja softverskih sistema serverski deo aplikacije (bekend) može se uopšteno shvatiti kao aplikacija koja apstrahuje pristup podacima na udaljenom serveru. Podaci se skoro uvek čuvaju u nekoj vrsti baze podataka, dok se pristup i manipulacija podacima dodatno apstrahuje aplikativnim[1] serverom.

Navedeni gradivni delovi - baza podataka i aplikativni server - tehnički su dovoljni za implementaciju bekend dela aplikacije. Mnogi kursevi razvoja veb aplikacija, pogotovu oni zasnovani na MEAN i MERN[2] okruženjima, upravo ovakve primere i prikazuju. Tu neretko srećemo arhitekturu od samo dva sloja - sloj aplikativnog programskog interfejsa (eng. Application Programming Interface - API) i sloja pristupa bazi podatka. Međutim, za izgradnju složene aplikacije koja uzima u obzir čistoću kôda, razdvajanje odgovornosti, održivost i iskustvo programera (eng. Developer Experience - DX) potrebno je dodatno raslojiti aplikaciju i pratiti neke ustanovljene principe.

## 3.1 DDD

Jedan od najpopularnijih pristupa za dizajn i modelovanje softvera jeste dizajn zasnovan na domenu (eng. Domain Driven Design - DDD). DDD zagovara detaljno upoznavanje svih učesnika razvoja softvera sa domenom koji dati softverski sistem predstavlja. Programeri u saradnji sa domenskim stručnjacima razvijaju

[1] Opštosti radi, izbegnuti su pojmovi **veb server** i **http server**. Veb server predstavlja server za dostavljanje statičkog sadržaja na vebu, i često kada govorimo o bekend aplikacijama ne mislimo na ovaj tip servera (nginx, apache2). HTTP server takođe ne bi bio ispravan pojam usled postojanja drugih protokola komunikacije na vebu.

[2] Mongo, Angular ili React, Express, Node.js, dve popularne kombinacije veb tehnologija.

zajednički, sveprisutan jezik (eng. ubiquitous language) kako bi se prevazišla barijera različitog rečnika svih strana. [3]

## Korišćenje domenskog jezika u programskom kôdu

Nakon uspostavljanja, sveprisutan jezik postaje sastavni deo komunikacije programera i domenskih stručnjaka (i svih ostalih učesnika razvoja), ali isto tako i programskog kôda. Naime, programski kôd koji nastaje kao produkt modelovanja principa DDD-a postaje jasniji i dostiže viši stepen samodokumentovanosti. U takvom kôdu sama imena klasa, metoda i promenljivih mogu biti dovoljna da programeru daju jasnu sliku šta koja celina kôda radi. Upravo zbog ovoga, primena DDD-a nije ograničena samo na velike kompanije ili velike timove, i ne služi samo kao sredstvo prevazilaženja „jezičkih" barijera, već doprinosi čistoći kôda i arhitekture i u manjim timovima, čak i jednočlanim. U sledeća dva primera prikazan je deo metode koja vrši ažuriranje komentara na tiketu. Primer 3.1 prikazuje ovu metodu, ne uzimajuću domenski jezik u obzir.

```
function updateComment(...){
  const ticket = //...

  if (!ticket) {
    throw new NotFoundException('Ticket not found.');
  }

  if (ticket.status === TicketStatus.CLOSED ||
      ticket.status === TicketStatus.RESOLVED) {
    throw new BadRequestException('Ticket closed.')
  }

  if (user.role === Role.CUSTOMER &&
      ticket.createdBy.id !== user.id) {
    throw new ForbiddenException('Not your ticket');
  }

  const comment = //...

  if (!comment) {
    throw new NotFoundException('Comment not found.')
  }

  if (comment.user.id !== user.id) {
    throw new ForbiddenException('Not your comment.');
  }
}
```

Primer 3.1: Kôd koji nije na domenskom jeziku

Odlikuje se ručnim proverama identifikacionih polja i direktnim podizanjem HTTP grešaka unutar servisa. Iako može biti razumljiv u toku pisanja, ovakva vrsta kôda u kompleksnom sistemu postaje neodrživa zbog nečitljivosti.

```
function updateComment(...) {
  const ticket = // ...

  if (!ticket) {
    throw new TicketNotFoundError(ticketId);
  }

  if (ticket.isFinalStatus()) {
    throw new CannotChangeTicketInFinalStatusError(ticket.status);
  }

  if (user.isCustomer() && !ticket.isOwner(user)) {
    throw new CannotCommentOnOthersTicketsError();
  }

  const comment = // ...

  if (!comment) {
    throw new CommentNotFoundError();
  }

  if (!comment.isOwner(user)) {
    throw new CannotUpdateOthersCommentsError();
  }
}
```

Primer 3.2: Kôd koji je na domenskom jeziku

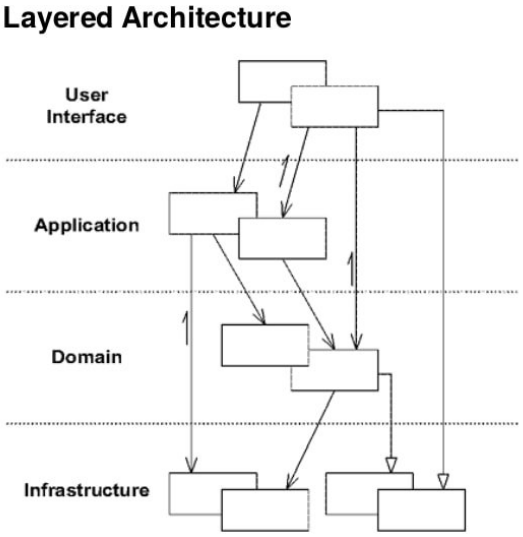
U ovom primeru primećujemo prisustvo domena u svim aspektima. Metode kao što su `isFinalStatus` i `isOwner` i semantičke klase grešaka kao `TicketNotFoundError` doprinose da određeni delovi kôda dostižu nivoe čitljivosti bliske prirodnom jeziku.

## Slojevita arhitektura

Prilikom izrade softvera, usled rokova i nedostatka resursa, neretko dolazi do isprepletanosti poslovne logike, logike korisničkog interfejsa, logike pristupa bazi podataka i drugih komponenti sistema [4]. Ovakve odluke u razvoju, iako rezultuju kratkoročnim rešenjem, dugoročno dovode do borbe sa *tehničkim dugom* (eng. technical debt). Postojanje tehničkog duga javlja se i iz drugih odluka tokom razvoja i u praksi je pokazano da opšte rešenje ovog problema ne postoji; svaki tim se pre ili kasnije sretne sa tehničkim dugom koji otežava i usporava rad. Ipak,

ta neminovnost ne treba da obeshrabri rano uvođenje dobrih praksi kao što je raslojavanje arhitekture.

DDD razaznaje četiri sloja, s tim što u ovom poglavlju razmatramo prva tri, dok je sloj korisničkog interfejsa predmet narednog poglavlja. Detaljna diskusija o slojevima, uz relevantnu implementaciju, prikazana je u sekcijama 2.2, 2.3 i 2.4.




Slika 3.1: Slojevi sistema [4].

## 3.2 Okruženje NestJS

NestJS [5] je Node.js okruženje za razvoj serverskih aplikacija. Izgrađen je nad bibliotekom `express` [6] koja je i danas popularan izbor za izradu manjih serverskih aplikacija kao i za edukativne svrhe. U srž ovog okruženja ugrađeni su koncepti modula, servisa i umetanja zavisnosti (eng. Dependency Injection - DI), što ga čini idealnim za konstrukciju aplikacije zasnovane na DDD principima.

Osnovni gradivni elementi NestJS aplikacije su *moduli*. Svaki modul obuhvata jednu smislenu celinu aplikacije koja može sadržati *kontrolere* i *provajdere*. U slučaju DDD-a, modul odgovara jednom izolovanom kontekstu[3]. Module definišemo na deklarativni način, pri čemu u okviru modula možemo uvoziti i druge module kako bismo postigli saradnju različitih delova sistema.

Kao što se vidi u primeru, NestJS se u velikoj meri oslanja na metaprogramiranje i mogućnosti jezika `typescript` u vidu dekoratora. Prilikom pokretanja

[3]eng. Bounded Context [4]

```
@Module({
  // Sekcija koja uvozi ostale module
  imports: [
    MongooseModule.forFeature(...),
    UsersModule,
    TicketTagSystemModule,
    NotificationsModule,
  ],
  // Sekcija koja deklarise kontrolere
  controllers: [TicketsController],
  // Sekcija koja deklarise servise i repozitorijume
  providers: [
    TicketService,
    TicketCommentService,
    TicketRedactionService,
    TicketTagUpdateService,
    TicketAssigneesService,
    TicketsRepository,
  ],
})
export class TicketsModule {}
```

Primer 3.3: Deklaracija modula za tikete.

NestJS aplikacije vrši se korak poznat kao **bootstrap**, odnosno priprema aplikacije za rad. U tom koraku se razrešavaju sve međuzavisnosti modula i servisa, i instanciraju sve klase kojima se upravlja putem umetanja zavisnosti.

Što se tiče strukture direktorijuma i fajlova u okviru projekta, korišćen je hibrid između mnogo sistema viđenih u praktičnom radu. Takva struktura najbolje odgovara zahtevima okruženja i sistema koji modelujemo. Primer strukture prikazan je u 3.4.

## 3.3 Sloj infrastrukture

Sloj infrastrukture ima više uloga u DDD sistemu, ali zajednička osobina je odsustvo iniciranja akcija u sloju domena [4]. Veliki deo ovog sloja često zauzima implementacija trajnosti podataka (eng. data persistence), i to neretko šablonom repozitorijuma (eng. Repository pattern). Uloga ovog šablona jeste odvajanje logike za čuvanje podataka van domenskog modela [7]. Često se ostvaruje uz pomoć

```
|-- api/
|   |-- dto/
|   |-- profiles/
|-- domain
|   |-- entities/
|   |-- errors/
|   |-- services/
|   |-- value-objects/
|-- infrastructure
    |-- interceptors/
    |-- schema/
```

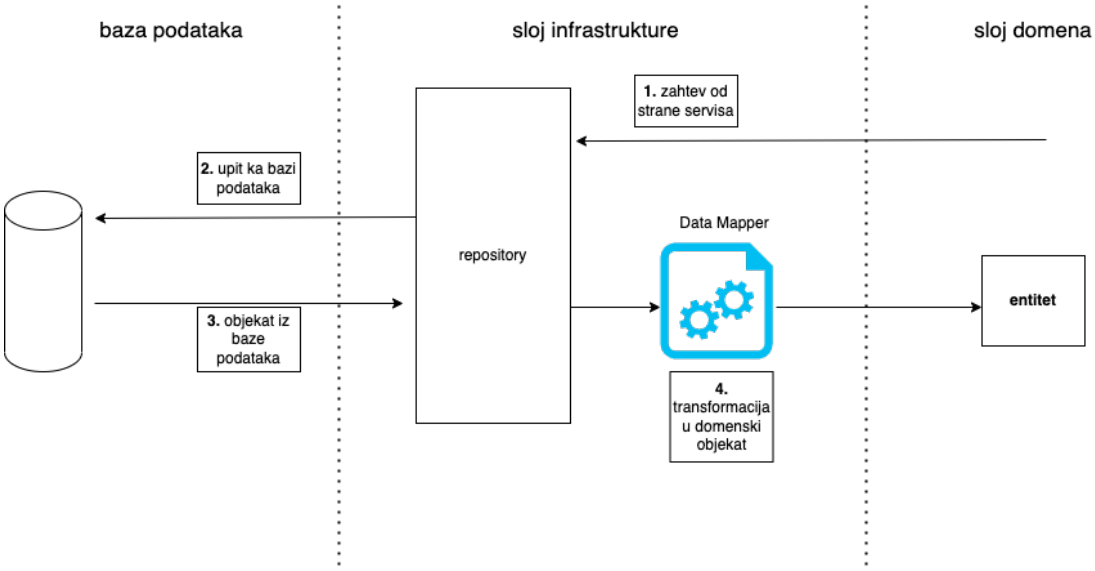
Primer 3.4: fajl tickets.module.ts

jedne ili više klasa sa nekim uobičajenim metodama za dohvatanje ili ažuriranje entiteta.

## Repozitorijumi

Apstrakcija koju pruža sloj repozitorijuma ogleda se upravo u načinu na koji on obrađuje podatke pre vraćanja u više slojeve, tačnije služi se **šablonom preslikavanja podataka** (eng. Data mapper pattern). Uloga mapiranja podataka uglavnom je u praksi delegirana konkretnoj biblioteci ili okruženju, koje upravlja pristupom bazi putem softverskog segmenta iz klase Objektno-relacionih ili Objektno-dokumentnih preslikavača (eng. Object Relational Mapper - ORM, Object Document Mapper - ODM). Iako ti sistemi često u potpunosti zadovoljavaju potrebe mapiranja podataka, u ovom radu je mapiranje urađeno ručno, tako da se objekti dobijeni ODM slojem, odnosno bibliotekom `mongoose` [8], preslikavaju u entitete koristeći biblioteku `@automapper/js` [9].

Prirodno se nameće pitanje dohvatanja povezanih entiteta, ili konkretnije rad sa agregatima. U ovom radu implementiran je sistem **pohlepnog učitavanja** (eng. eager loading), kojim se svi povezani entiteti zahtevanog entiteta dohvataju odmah prilikom zahteva ka glavnom entitetu. U nekim sistemima ovakav izbor ne bi bio idealan, međutim takva odluka doneta je usled činjenice da je dizajn baze podataka relativno jednostavan, a lakoća implementacije i korišćenja nadmašuje cene u performansama koje su minimalne. Kombinacijom metode *populate* iz biblioteke `mongoose`, kao i rekurzivne prirode biblioteke `automapper/js`, ovo se lako postiže.




Slika 3.2: Dijagram dohvatanja entiteta preko repozitorijuma.

U primeru 3.5 vidimo zaglavlje klase `TicketsRepository`. Kao i sve ostale repozitorijumske klase, oslanjaju se na model klase dobijene od strane biblioteke `mongoose` i deklarišu niz koji predstavlja sve putanje koje treba popuniti zarad implementacije pohlepnog učitavanja.

```
export class TicketsRepository {
  constructor(
    @InjectModel() private ticketModel: Model<TicketDb>,
    @InjectMapper() private readonly mapper: Mapper,
  ) {}

  public static POPULATE = [
    {
      path: 'history.initiator',
      model: 'UserDb',
    },
    {
      path: 'history.payload.assignees',
      model: 'UserDb',
    },
    { path: 'createdBy', model: 'UserDb' },
    {
      path: 'tags',
      model: 'TicketTagDb',
      populate: { path: 'group', model: 'TicketTagGroupDb' },
    },
    { path: 'assignees', model: 'UserDb' },
  ];
```

Primer 3.5: Repozitorijum tiketa, konstrukcija i niz POPULATE.

Primer 3.6 prikazuje primer dohvatanja jednog tiketa putem identifikatora, pri čemu se, naravno, prilikom vraćanja rezultata vrši mapiranje u odgovarajući entitet.

```
async findById(id: string): Promise<Ticket | null> {
    const result = await this.ticketModel
      .findById(id)
      .populate(TicketsRepository.POPULATE);

    if (!result) {
      return null;
    }

    return this.mapper.map(result, TicketDb, Ticket);
}
```

Primer 3.6: Dohvatanje jednog entiteta.

## Sheme, mutacije podataka, šablon Event Sourcing

Osim repozitorijuma, sloj infrastrukture prirodno sadrži i definicije shema biblioteke `mongoose`. Iz ugla programera, ovi objekti su najbliža pristupna tačka bazi podataka i podržavaju operacije upita i mutacija nad realnim podacima u bazi. Takav jedan upit viđen je na isečku 3.6 u prethodnoj sekciji.

Na ovom mestu bavimo se mutacijama podataka, i to baš tiketa. Osim prikaza rada sa mutiranjem podataka kroz biblioteku `mongoose`, ovaj primer služi da ilustruje dualnu prirodu tiketa kao entiteta i njegovu potpunu različitost u strukturi između sloja infrastruktura i sloja domena. Naime, kao što će biti prikazano u sledećem poglavlju, tiket u sloju domena je jednostavna klasa koja sadrži sve podatke o datom tiketu. Ipak, u sloju infrastrukture, a radi veće fleksibilnosti i mogućnosti praćenja istorije, upotrebljena je varijanta šablona *Event Sourcing*. Osnovno načelo ovog šablona je da umesto čuvanja eksplicitnog stanja objekta čuvamo istoriju njegovih promena.

Na isečku 3.7 vidimo deo klase koja predstavlja shemu za rad sa tiketima.

```
@Schema({ collection: "tickets" })
export class TicketDb extends Document {
  _id: string;

  @Prop({ type: Date })
  createdAt: Date;

  @Prop({ type: String })
  title: string;

  @Prop({ type: String })
  body: string;

  @Prop({ type: String })
  status: TicketStatus;

 // ...

  @Prop({ type: [{ type: mongoose.Schema.Types.Mixed }] })
  history: TicketHistoryItem[];
}
```

Primer 3.7: Definicija sheme tiketa.

Niz `history` predstavlja promene primenjene na datom tiketu na osnovu kojih se može izvesti trenutno stanje tiketa. Ipak, prirodno se postavlja pitanje zašto se, osim istorije promena, čuvaju i eksplicitna polja kao što su `status`, `title`, `body` i drugi, budući da se ti podaci mogu izvući putem istorije. Razlog leži u performansama pretrage. Ukoliko bismo čuvali samo listu promena, a ne sliku trenutnog stanja (eng. snapshot), svaka pretraga bila bi linearne vremenske složenosti u odnosu na dužinu istorije, što bi rezultovalo lošim performansama.

Sa ovakvom postavkom, mutacije nad tiketom postižu se diferencijacijom prosleđenog entiteta i entiteta iz baze podataka (primer 3.8).

```
async update(newTicket: Ticket, user: User) {
  const document = await this.ticketModel
    .findById(newTicket.id)
    .populate(TicketsRepository.POPULATE);

  const timestamp = new Date();

  // Trenutno stanje tiketa
  const ticket = this.mapper.map(document, TicketDb, Ticket);

  // Primer diferencijacije nad poljem title
  if (newTicket.title !== ticket.title) {
    const item = new TicketHistoryItem();
    item.initiator = user.id as unknown as UserDb;
    item.timestamp = timestamp;
    item.type = TicketHistoryEntryType.TITLE_CHANGED;

    // Dodajemo novu stavku u istoriju
    item.payload =
       new TicketHistoryEntryTitleChanged(newTicket.title);

    document.history.push(item);
    // Menjamo snapshot stanje
    document.title = newTicket.title;
  }
```

Primer 3.8: Fajl *ticket.repository.ts*

Za kraj, primerom 3.9, prikazujemo uprošćenu verziju mapiranja iz oblika u sloju infrastrukture u domenski entitet, dok se puna verzija nalazi u izvornom kôdu u fajlu `ticket.entity-profile.ts`.

```
forMember((destination) => destination.comments,
  mapFrom((source) => {
    const commentItems = source.history.filter(
      (item) => item.type === COMMENT_ADDED,
    );

    return commentItems
      .map((item) => {
        // pretrazujemo sve komentare u istoriji
        const payload = item.payload as CommentAdded;
        // proveravamo da li su kasnije bili obrisani
        const deletes = source.history.filter(
          (deleteItem) => deleteItem.type === COMMENT_DELETED
          && (deleteItem.payload as CommentDeleted)
              .commentId === payload.commentId,
          );

        const wasDeleted = deletes.length > 0;

        if (wasDeleted) {
          return null;
        }

        // ... mapiranje samog komentara u domenski oblik...

        return comment;

```

Primer 3.9: Mapiranje komentara tiketa.

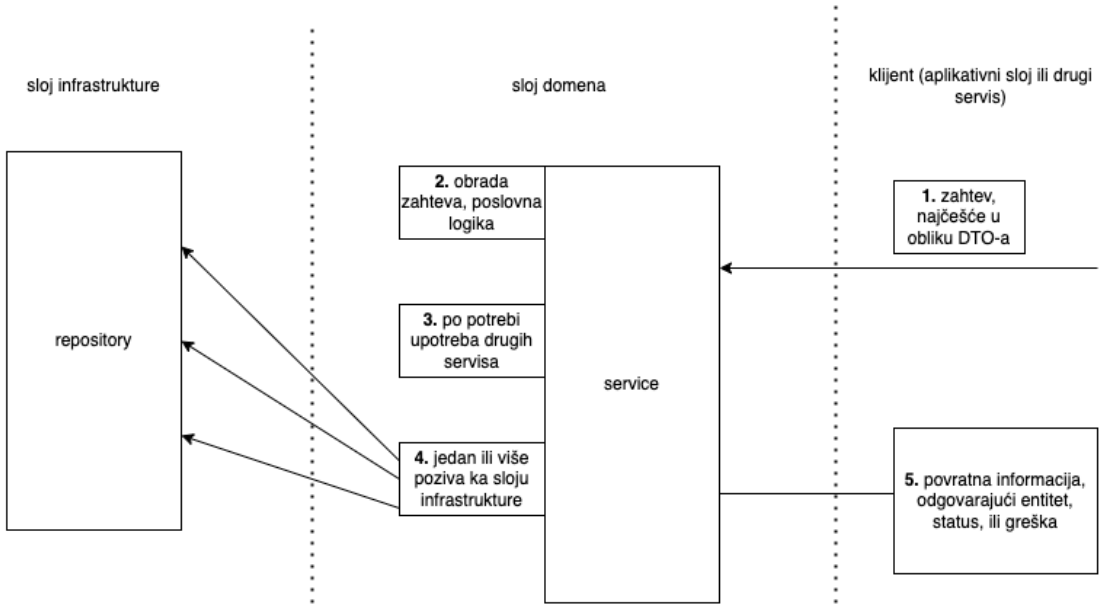
## 3.4 Sloj domena

Takođe poznat kao srce poslovnog softvera, sloj domena sadrži glavnu poslovnu logiku i operacije koje direktno oslikavaju dešavanja u svetu koji softver modeluje. Iz ugla implementacije, svoje mogućnosti pružaju putem apstrakcije koju često nazivamo servisima, što koristimo i kao sufiks u nazivima njihovih klasa, iako se mogu koristiti i druge konvencije[4].

[4]Nekada se umesto sufiksa *Service* koriste reči koje direktno oslikavaju namenu datog servisa. Na primer umesto *GetUserService* koristi se naziv *UserProvider* i sl.

## Tok podataka kroz domenski sloj

Usled činjenice da obuhvata sve poslovne operacije nije lako definisati opštu namenu domenskog sloja. Na slici 3.3 prikazan je uopšten primer toka podataka kroz ovaj sloj i njemu susedne slojeve.




Slika 3.3: Uopšteni tok obrade podataka kroz domenski sloj.

Ipak, operacije unutar servisa (koraci 2-4) konkretizovaćemo za potrebe sistema kojim se bavimo, jer se mogu svesti na sledeći šablon, gde redosled ne mora nužno biti ispraćen u kôdu usled prirode jezika[5]:

1. Dohvatanje odgovarajućeg entiteta
2. Provera domenskih prava pristupa i domenskih grešaka
3. Obrada entiteta
4. Komunikacija sa drugim servisima
5. Osiguravanje trajnosti podataka
6. Priprema i vraćanje odgovarajućeg odgovora

[5]Zbog asinhrone prirode okruženja Node.js, a radi postizanja maksimalnih performansi, neke operacije pokrećemo paralelno a zatim čekamo koristeći funkciju *Promise.all*, što može naizgled delovati kao mešanje koraka.

Na sledećem primeru operacije dodeljivanja tiketa agentima prikazujemo prethodne korake, redom.

**Dohvatanje entiteta**

```
async addAssignees(ticketId: string, user: User, dto: AssignDTO) {
  const ticket = await this.ticketsRepository.findById(ticketId);
  if (!ticket) {
    throw new TicketNotFoundError(ticket.id);
  }
```

Primer 3.10: Korak dohvatanja tiketa.

**Provera domenskih prava pristupa i domenskih grešaka**

```
  // Domensko pravo pristupa - korisnicima nije dozvoljeno
  // dodeljivanje tiketa
  if (user.isCustomer()) {
    throw new NotAllowedToAssignError();
  }
  const assignees: User[] = await Promise.all(
    dto.assignees.map((id) => this.usersService.findOne(id)),
  );

  for (const assigneeUser of assignees) {
    // Greske domenskog karatkera
    if (!assigneeUser) {
      throw new AssigneeNotFoundError();
    }
    if (assigneeUser.isCustomer()) {
      throw new CannotAssignCustomerError(assigneeUser.id);
    }
    if (ticket.isAssigned(assigneeUser)) {
      throw new DuplicateAssigneeError(assigneeUser.id);
    }
}
```

Primer 3.11: Korak provere domenskih prava pristupa i domenskih grešaka.

### Obrada entiteta

U ovom slučaju obrada je jednolinijska, zbog pogodne definicije klase Ticket.

```
ticket.assign(assignees);
```

Primer 3.12: Korak obrade entiteta.

### Komunikacija sa drugim servisima

Konstruišemo niz obaveštenja koja je potrebno poslati, slanje delegiramo odgovarajućem servisu.

```
const usersToNotify = assignees.filter(
  (assignee) => assignee.id !== user.id,
);
const notifications = usersToNotify.map((userToNotify) =>
  NotificationFactory.create((builder) =>
    builder
      .forUser(userToNotify)
      .hasPayload('assigned', (assignBuilder) =>
        assignBuilder.atTicket(ticket).byUser(user),
      ),
  ),
);
const promise = this.notificationsService.emitNotifications(
  ...notifications,
);
```

Primer 3.13: Korak komunikacije sa drugim servisima.

### Osiguravanje trajnosti podataka

Čuvamo načinjene promene u bazi podataka.

```
// ...
this.ticketsRepository.update(ticket, user);
```

Primer 3.14: Korak očuvanja trajnosti podataka.

**Priprema i vraćanje odgovarajućeg odgovora**

Gde pritom koristimo servis `TicketRedactionService` kako bismo otklonili podatke koje dati tip korisnika ne bi trebalo da vidi.

```
1 this.ticketRedactionService.prepareResponse(updatedTicket, user);
2 return updatedTicket;
```

Primer 3.15: Korak pripreme i vraćanja odgovora.

## Obrada grešaka

U prethodnom primeru toka podataka primećujemo intenzivan rad sa greškama. Ovom delu posvećena je posebna pažnja jer je obrada grešaka jedno od centralnih pitanja dizajna kompleksnog softverskog sistema, i bitno je od početka dizajnirati je korektno kako bi kasniji rad bio znatno olakšan.

Naime, programer koji se bavi implementacijom odgovarajuće domenske funkcionalnosti pre ili kasnije susrešće se sa potrebom da obrađuje greške. Čak i najjednostavniji zahtev, kao što je, recimo, dohvatanje jednog tiketa, zahteva rad sa greškama na svakom od nivoa slojevite arhitekture. Manuelno orkestriranje grešaka pri svakoj implementaciji neke funkcionalnosti dovelo bi do usporenog rada i nekonzistentnog sistema bez jasnih standarda.

Stoga pravimo jasnu podelu između grešaka domenskog nivoa i grešaka aplikativnog nivoa (HTTP grešaka), i u domenskom sloju držimo se isključivo grešaka domenskog nivoa. Sve domenske greške nasleđuju baznu klasu *BaseError* prikazano u primeru 3.16.

```
1  export class BaseError {
2    constructor(public message: string = 'An error occurred') {}
3
4    public getName() {
5      return this.constructor.name;
6    }
7
8    public getPayload(): any {
9      return {
10       message: this.message,
11       errorType: this.getName(),
12     };
13   }
14 }
```

Primer 3.16: BaseError.ts

Na ovaj način imamo uniforman interfejs kako svaka greška treba da izgleda, sa dovoljno fleksibilnosti da pokrije sve domenske situacije. U kôdu možemo primetiti sve od jednostavnih grešaka koje predstavljaju nepostojanje - primer `TicketNotFound`, do grešaka koje ukazuju na kompleksnu sitaciju - primer `CannotChangeCommentsOfAClosedTicket`. Na ovakav sistem mogu se uložiti nekoliko kritika, koje ćemo razjasniti.

**Repetitivnost.** Sve greške koje se odnose na nepostajanje entiteta imaju identičan oblik, pa se može uložiti primedba na ponavljanje kôda i sugerisati kreiranje dodatnih apstrakcija. U ovom radu izbegli smo takav pristup iz dva razloga:

- **Bespotrebne apstrakcije**. Iako je tehnički moguće apstrahovati greške nepostajanja entiteta u neki tip višeg nivoa - recimo `EntityNotFoundError`, ovo uvodi previše jezičke kompleksnosti u domenski sloj i na duge staze ne doprinosi ni čitljivosti ni razumevanju.
- **Fleksibilnost**. Ukoliko svaki entitet ima priliku da definiše svoje greške bez ograničenja međuapstrakcija, postiže se maksimalna fleksibilnost u okviru datog entiteta. Može se argumentovati da ovakav pristup olakšava potencijalni prelaz na mikroservisnu arhitekturu, po potrebi.

**Preterana ekspresivnost.** Na ovom mestu konkretno govorimo o dužini naziva nekih klasa, koje se mogu smatrati nečitljivim. Ipak, ako se podsetimo dela 3.1, jasno nam je da dobitak na razumljivosti domena kroz kôd nadmašuje strah od dugačkih imena klasa.

Ostaje pitanje transformacije ovakvih grešaka u HTTP oblik. Jedno izrazito elegantno rešenje, koje nam pružaju mogućnosti okruženja NestJS, biće prikazano u sledećem pogavlju.

## 3.5 Sloj aplikacije

Sve do sad razmatrane slojeve i delove sistema možemo smatrati internim u smislu mogućnosti spoljnjeg pristupa - sloj domena bavi se poslovnom logikom, uz podršku sloja infrastrukture. Kako bi domenski sistem mogao da se koristi od strane grafičkih veb aplikacija, mobilnih aplikacija ili, uopšteno, bilo kojih drugih softverskih sistema, potebno je da svoju funkcionalnost eksponira spoljašnjem svetu.

### REST API, svojstva

Ispoljavanje spoljašnjem svetu postiže se slojem aplikacije, tankim slojem bez poslovnog znanja, čija glavna uloga jeste primanje zahteva i njihova delegacija na domenski sloj [4]. U ovom radu, za potrebe implementacije aplikativ-

nog sloja, bavimo se najpoznatijim arhitekturalnim stilom poznatim kao REST (***RE**presentational **S**tate **T**ransfer*). Kompletna specifikacija ovog protokola je ogromna i pokriva sve aspekte korišćenja HTTP-a kao transportnog sloja veb aplikacija, tako da pominjemo samo neke od najvažnijih:

- **Odsustvo stanja.** Jedan moderan princip softverskog inženjerstva (u vreme pisanja ovog rada) jeste da komponente ili moduli sistema treba da imaju što manje stanja, a po mogućstvu da ga u potpunosti eleminišu[6]. Način na koji ovo shvatamo u REST sistemima je sledeći - za dva API zahteva pod identičnim uslovima dobija se identičan odgovor. Dakle, ne postoji nikakav eksterni faktor samog aplikativnog sloja koji može uticati na odgovor koji korisnik (klijent) dobije, i smatra se da je klijent dužan da pruži kompletan kontekst potreban za obradu zahteva [10].
- **Identifikacija resursa i HTTP metode.** Neretko u praksi srećemo takozvane API slojeve sa previše ekspresivnim delovima URI-a[7]. REST definiše nedvosmislen način za identifikaciju resursa i njihovu hijerarhiju, kao i pravilnu upotrebu HTTP metoda (glagola). Na taj način API pristupna tačka[8], koja pruža mogućnost dohvatanja korisnika po identifikatoru, po pravilu treba da bude dostupna putem GET metode, i to na URI-u nalik na `api/users/123` [10]. Primer nepravilnog URI-a bilo bi direktno navođenje akcije dohvatanja - `api/get-user-by-id/213`. Na sličan način jasno je propisano koje metode se koriste za promenu, upis i brisanje entiteta.
- **Korišćenje HTTP greški.** Kao što je pomenuto u prethodnoj sekciji, obrada grešaka predstavlja vitalan aspekt razvoja softvera, i značajan deo njene implementacije leži u aplikativnom sloju. REST definiše skup pravila za korektno ukazivanje korisniku na greške koje su se dogodile.

## Kontroleri, validacija, mapiranje

Većina okruženja za programiranje veb aplikacija, uključujući i NestJS, pruža ugrađen konstrukt za obradu HTTP zahteva koga nazivamo **kontroler**. Kontroleri su ništa drugo nego klase nad kojim definišemo metode koje obrađuju HTTP zahteve ka određenom URI-u, dok je proces mapiranja URI-a na metodu maksimalno

[6]Ovaj princip postaje posebno primetan rastom popularnosti funkcionalnih programskih jezika.
[7]eng. Uniform Resource Identifier - URI
[8]eng. API endpoint

prepušten okruženju, a programeru eksponiran nekim vidom metaprogramiranja (najšće dekoratorima). Primer 3.17 ilustruje kontroler vezan za URI prefiks `users`, sa metodom koja reagovanje na dinamički zadat identifikator. Ovako postavljen kontroler obrađuje zahteve oblika `GET /api/users/123`.

```
@Controller('users')
export class UsersController {
  @Get(':id')
  findOne(id: number): UserDTO {
    // ...
  }
}
```

Primer 3.17: Primer kontrolera.

Uloga kontrolera je višestruka i sastoji se od sledećih koraka:

1. **Provera prava pristupa**. Ova provera treba da poznaje što manje poslovne logike. Na primer, neke pristupne tačke dostupne su samo ulogovanim korisnicima, neke samo administratorima, i sl.

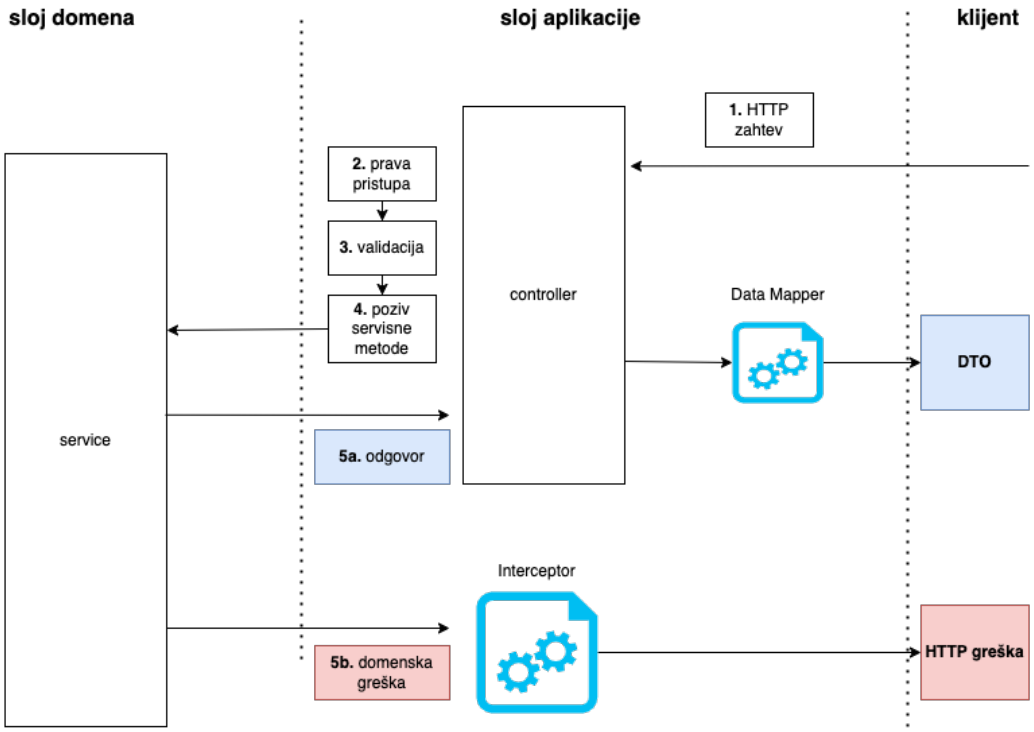
2. **Validacija ulaznih podataka**. Ponovo se misli na čisto tehničku validaciju, a ne na poslovnu validaciju. Tu spada provera da li su određeni brojčani podaci zaista brojevi, da li su određeni tekstualni podaci dovoljne dužine i slično.

3. **Delegacija posla odgovarajućem sevisu**. Nakon uspešne provere prava pristupa i validacije podataka, kontroler će prepustiti zadatak servisu koji je za to namenjem i po potrebi primiti povratnu informaciju.

4. **Obrada servisnih grešaka.** Kontroler je dužan da greške domenskog nivoa, koje podiže servis, preslika u HTTP greške. Neretka, i sasvim validna praksa, jeste korišćenje `try-catch` blokova u ove svrhe. Ipak, vrlo elegantan mehanizam presretača (eng. interceptors) okruženja NestJS omogućio je izolaciju ovog posla. Naime, presretač je klasa koju postavljamo na nivo kontrolera, u kojoj možemo dodati logiku obrade grešaka podignutih iz domenskog sloja.

5. **Preslikavanje podataka.** Ukoliko odgovor od servisa obuhvata entitet, dužnost kontrolera je da izvrši mapiranje datog entiteta u odgovarajući objekat za transfer podataka (eng. Data Transfer Object - DTO).

Slika 3.4 prikazuje dijagram rada aplikativnog sloja.




Slika 3.4: Aplikativni sloj.

Što se tiče tehničkih aspekata, maksimalno je korišćeno metaprogramiranje za postizanje validacije i preslikavanja grešaka, kako bi se izbeglo repetitivno pisanje kôda. Presretači su suštinski manuelne provere tipa instance greške. Validacija je izvedena deklarativno, kombinacijom biblioteka `class-transformer` i `class-validator`.

```
return next.handle().pipe(
  catchError((error) => {
  if (error instanceof TicketNotFoundError) {
    throw new NotFoundException(error.getPayload());

    if (error instanceof NotAllowedToRemoveThisTagError) {
      throw new ForbiddenException(error.getPayload());
    }

    if (error instanceof DuplicateTagError) {
      throw new ConflictException(error.getPayload());
    }
```

Primer 3.18: Presretač grešaka.

```
export class AddTicketTagsDTO {
  @Validate(isValidObjectId, { each: true })
  @IsArray()
  tags: string[];
}
```

Primer 3.19: Validacija ulaznog DTO-a.

# Glava 4

# Frontend

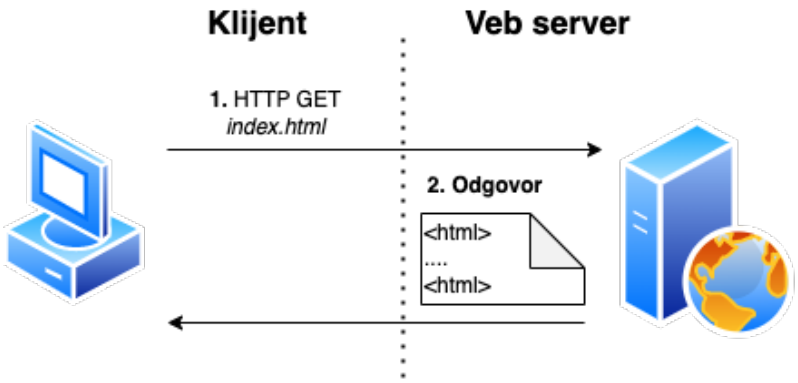
Ukoliko bi tražili jedan programski jezik koji može da pokrije najširi spektar platformi i aplikacija, to bi u današnje vreme definitivno bio Javascript. Uprkos velikom broju platformi koje mogu pokretati Javascript kôd (veb pregledači, Node.js, Deno, Bun, itd), glavno okruženje koje diktira njegov razvoj jeste upravo i originalno - veb. Pod ovim se misli na podudaranje između Javascript-a definisanog prema TC39[1] standardu i Javascript-a koji se izvršava u veb pregledačima [11]. Postojanje tolikog broja okruženja prirodno je dovelo do alata, ili čak kompletnih okruženja za razvoj, sa ciljem da apstrahuju sistem nad kojim se izvršava i da kôd učine što univerzalnijim.

## 4.1 Kratak pregled evolucije jezika Javascript i frontend razvoja

### Statički sajtovi

Prvi oblik veb sajtova bio je veoma jednostavan. Zasnivao se na konceptu dostavljanja statičkih HTML i CSS fajlova preko kojih je korisnik mogao da pregleda neki sadržaj. Navigacija se vršila kroz hiperlinkove koji bi upućivali na druge adrese gde bi klijent zahtevao druge statičke fajlove, čime bi proces počinjao ponovo.

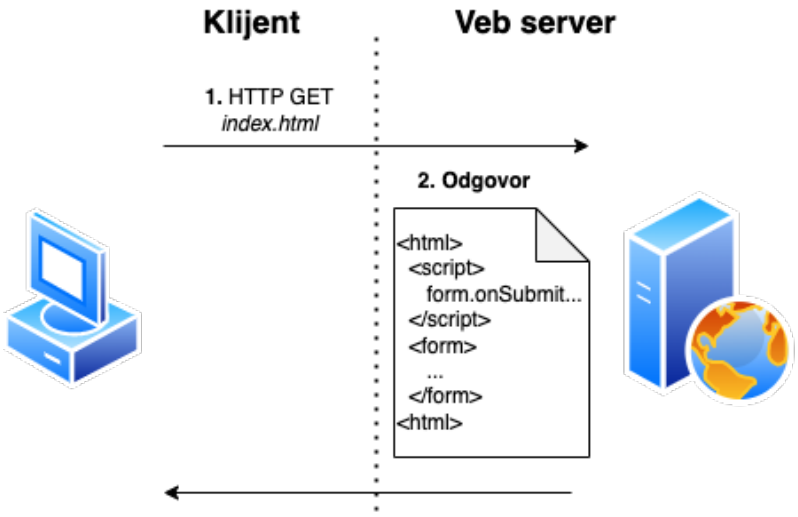
[1] eng. Technical committee 39 - tehnički komitet zadužen za standardizovanje jezika.




Slika 4.1: Dostavljanje statičkih HTML sajtova.

## Dodavanje klijentske interaktivnosti

Problem koji se javlja sa prethodnim sistemom jeste nedostatak klijentske interaktivnosti - za svaku promenu sadržaja stranice potrebno je učitati celu stranicu od početka[2]. Uvođenje klijentskih skripti jezikom Javascript omogućuje se manipulacija HTML elemenata sa klijentske strane, upravljanje događajima, i slično.

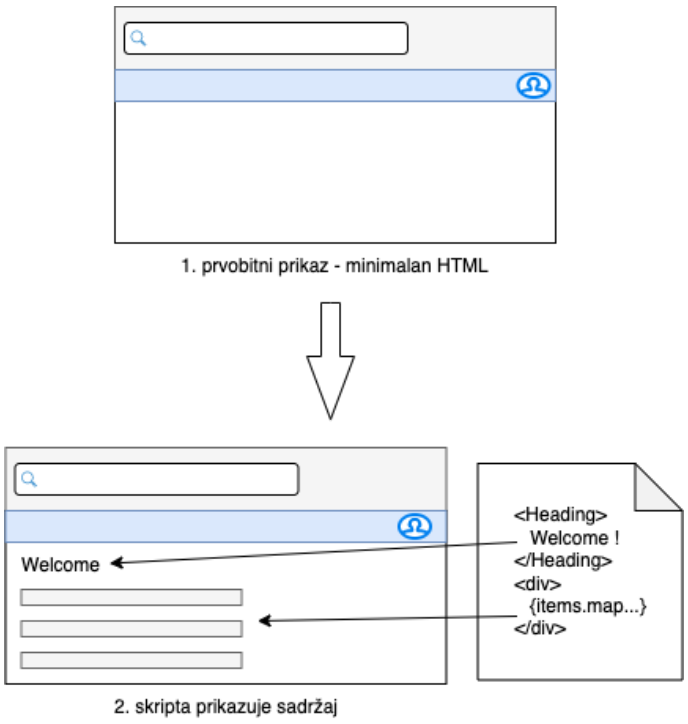



Slika 4.2: Dostavljanje statičkih HTML sajtova i propratnih skripti.

[2]Najbolji primer koji ilustruje probleme nedostatka interaktivnosti jesu veliki formulari i njihova validacija. Na modernim veb sajtovima prilikom popunjavanja velikih formulara korisnik konstantno ima povratnu informaciju da li je neko polje pogrešno popunjeno, dok pre Javascript-a to se saznaje tek nakon podnošenja.

## Jednostranične aplikacije - SPA

Sledeći veliki zaokret dešava se uvođenjem koncepta jednostraničnih aplikacija (eng. Single Page Application - SPA). Ove aplikacije odlikuje potpuno prebacivanje logike generisanja korisničkog interfejsa na frontend deo aplikacije. Server bi poslao samo skelet HTML-a i Javascript skriptu, koja bi zatim preuzela odgovornost prikaza elemenata. Dohvatanje podataka (eng. data fetching) za prikaz na stranici vršilo bi se putem API-a, tako da ovaj sistem podseća na način na koji rade mobilne aplikacije - potpuno razdvajanje odgovornosti prikaza korisničkog interfejsa i dohvatanja podataka. Originalne mane ovakvog sistema zasnivale su se na lošijem rangiranju stranica od strane sistema za pretragu usled manjka početnog sadržaja koji se prikazuje[3]. Prednost je u korisničkom iskustvu - navigacija se vrši potpuno klijentski tako da korisnik ima osećaj jako brze navigacije, kao na desktop aplikaciji. Najpopularnija i najuticajnija biblioteka za razvoj SPA jeste React.




Slika 4.3: SPA

[3]U današnje vreme skoro svi moderni pretraživači uspešno indeksiraju i SPA sajtove, tako da je ovaj problem praktično eleminisan.

## Hibridna okruženja

Prethodni način generisanja korisničkog interfejsa često se naziva klijentsko renderovanje (eng. CSR - Client side rendering), dok se tradicionalni sistemi zasnivaju na serverskom renderovanju (eng. SSR - Server side rendering). Okruženja kao što su Next.js[12], Remix[13], SvelteKit[14] i druga, imaju cilj za implementaciju „najboljeg od oba sveta". Naime, omogućavaju razvoj jednostraničnih aplikacija uz podršku za serversko renderovanje, tako da otklanjaju mane u nedostatku prvobitnog prikaza HTML-a. Način njhovog rada biće prikazan u nastavku, sa izborom okruženja Next.js.

## 4.2 React i Next.js

Da sumiramo, problemi koje želimo da izbegnemo kod tradicionalnih jednostraničnih aplikacija jesu sledeći:

1. **Prvobitni HTML je minimalan.** Usled činjenice da se klijentske skripte bave prikazom korisničkog interfejsa, konkretno u React-u koristeći sistem komponenti i JSX-a [15], inicijalni HTML koji se prikazuje korisniku je prazan i postoji vreme koje je potrebno dok veb pregledač izvrši skriptu.

2. **Potrebno je dohvatiti podatke s klijenta.** Čak i nakon prikazivanja početnog korisničkog interejsa, najčešći je slučaj da je potrebno dohvatiti neke podatke preko API-a koje treba prikazati. Pošto je sva odgovornost na klijentu, ti podaci se tek mogu dohvatati u ovoj fazi, tako da postoji dodatno vreme gde neke sekcije sajta sadrže simbolične prikaze sadržaja koji se učitava[4], dok se u pozadini vrši dohvatanje podataka.

### React kao biblioteka i okruženje

U ekosistemu React-a postoje nesuglasice da li React predstavlja biblioteku ili okruženje za razvoj. Argument koji ide u korist biblioteke zasniva se na činjenici da React ne poseduje striktnu strukturu direktorijuma, ne nalaže kako se vrši dohvatanje podataka, ne sadrži ugrađen sistem za rutiranje, i slično. Uzevši to u obzir zaista deluje da je React ništa više nego biblioteka za razvoj interaktivnih korisničkih interfejsa. S druge strane, vremenom su evoluiorali šabloni koji su opšteprihvaćeni kao dobre prakse, čime se može argumentovati i da je React kompletno okruženje za razvoj. Ovo je dovelo do evolucije još jednog termina koji se može sresti u onlajn zajednicama frontend inženjera, a to je metaokruženje (eng. metaframework); neretko se upravo ovom klasom softverskih sistema naziva Next.js.

### Next.js i šabloni renderovanja korisničkih interfejsa

Imajući prethodnu diskusiju u vidu i koristeći se što opštijim terminima, možemo reći da je Next.js moderno okruženje za razvoj veb aplikacija, izgrađeno

[4]Ustanovljen naziv u frontend svetu za ove elemente jeste *content loader*, postoji veliki broj biblioteka koje olakšavaju njihovo generisanje.

nad React-om, koje podržava više šablona renderovanja korisničkih interfejsa. U trenutku pisanja ovog rada, šabloni renderovanja korisničkih interfejsa (eng. rendering patterns) čest su predmet diskusija onlajn zajednica inženjera. Najveća okruženja i biblioteke utrkuju se sa implementacijom svih šablona kako bi bili konkurentan izbor modernih frontend proizvoda. Next.js, predvođen kompanijom Vercel, definitivno je među prvima kada je u pitanju razvoj veb aplikacija. Od pomenutih šablona Next.js podržava sledeće [12]:

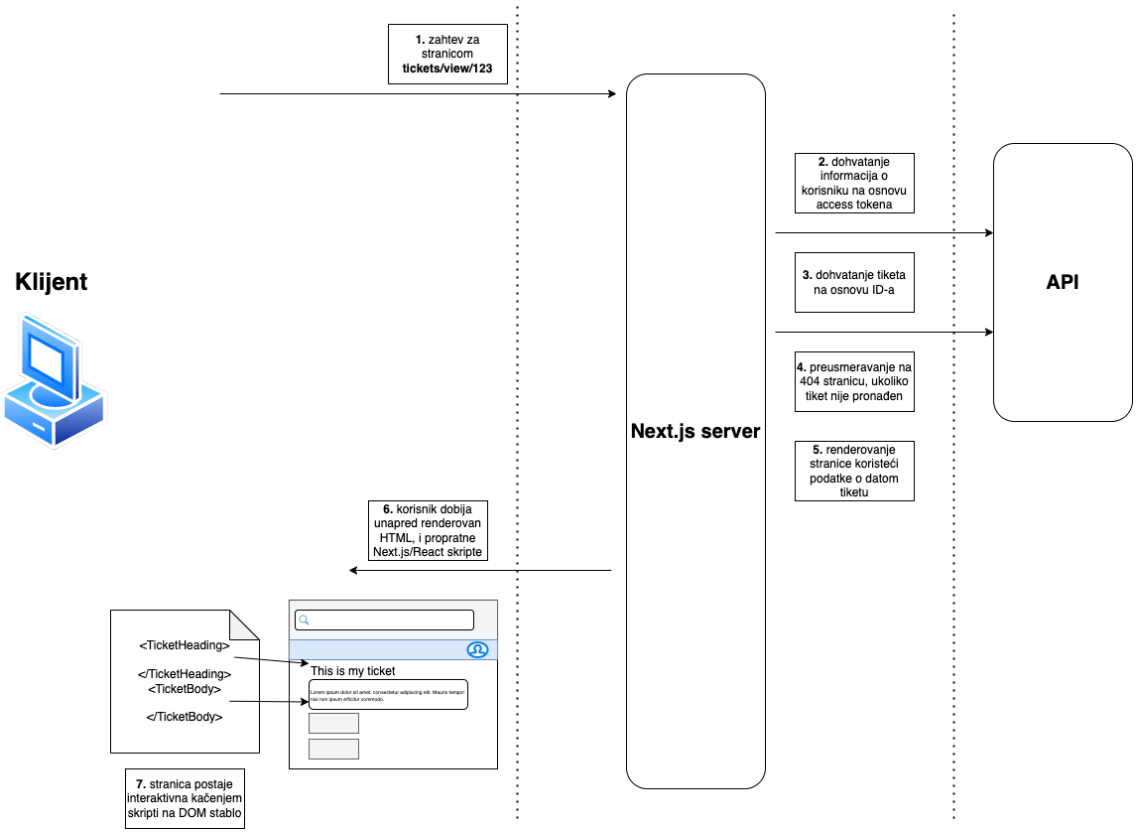
- **Klijentsko renderovanje - CSR.** Klasično SPA ponašanje podržano je samom činjenicom da se u osnovi nalazi React.
- **Serversko renderovanje - SSR.** Omogućeno je dohvatanje eksternih podataka na serveru pre renderovanja stabla React komponenti.
- **Generisanje statičkih sajtova - SSG[5].** Moguće je određene stranice u potpunosti renderovati u vremenu kompajliranja aplikacije (eng. build time), tako da gotov HTML bude spreman i pre nego što se aplikacija pokrene. Ovakve stranice po pravilu su najbrže.
- **Inkrementalno regenerisanje statičkih stranica - ISG[6].** SSG stranice jesu brze, ali mana im je što podaci mogu zastareti. Next.js omogućava periodično osvežavanje tih stranica kako bi sadržaj bio ažuran.
- **Strimovanje korisničkog interfejsa[7].** Tokom pisanja rada pojavila se React verzija 18 koja uvodi koncept serverskih komponenti (eng. server components), koje Next.js usvaja u verziji 13 i proglašava kao novi pravac. U ovom radu ne bavimo se ovim sistemom, jer je u trenutku pisanja i dalje u eksperimentalnoj fazi.

## SSR i tok jednog zahteva

SSG i ISR jesu dobar izbor prilikom generisanja sajtova koji ne menjaju svoj sadržaj često i ne bave se intenzivnom manipulacijom podataka[8]. Ipak, sistem za tikete, koji pravimo u ovom radu, je kompleksniji i temeljno se zasniva na dohvatanju i manipulaciji podataka sa eksternog API-a. Iz tog razloga glavni

[5]eng. Static Site Generation
[6]eng. Incremental Static Regeneration
[7]eng. Streaming SSR
[8]Dobar primer sajtova pogodnih za SSG i ISR jesu personalni blogovi.

iskorišćen šablon upravo je serversko renderovanje. Na dijagramu 4.4 prikazan je primer toka jednog zahteva ka stranici za prikaz tiketa.




Slika 4.4: Tok jednog SSR zahteva

Proces u kom stranica postaje interaktivna, odnosno korak 7 sa dijagrama, naziva se **hidracija** [15]. Ovaj naziv ima smisla i može se nerformalno shvatili ako stablo React komponenti zamislimo kao sistem međusobno povezanih praznih cevi, koje onda na vrhu prikačimo na izvor vode - u ovoj analogiji Javascript.

## 4.3 Dohvatanje podataka, upravljanje stanjem, biblioteka RTK

Kao što je pomenuto u prethodnoj sekciji, React (a tako i Next.js) prepuštaju programeru izbor načina na koji će da vrši dohvatanje podataka sa eksternih servisa. Iako to može zvučati trivijalno, dohvatanje podataka jedan je od najsloženijih zadataka modernih frontend sistema i očekivano je da React ekosistem nudi veliki broj biblioteka koji rešavaju ovaj izazov. Problemi koji se očekuju da budu rešeni od strane modernih biblioteka za dohvatanje podataka uključuju sledeće:

- **Keširanje i deduplikacija.** Želimo mogućnost da dohvatamo iste podatake u različitim delovima stabla aplikacije, a da pritom ne pozivamo API svaki put, već da se podaci keširaju i iskoriste ponovo.
- **Deklarativnost.** Želimo da se oslobodimo pisanja proceduralne logike za dohvatanje podataka, tako što to izvršimo na deklarativan način pisanjem odgovarajuće konfiguracije.
- **Stanja učitavanja i grešaka.** Potrebno je imati ugrađenu podršku za stanja učitavanja (eng. loading state) i stanja grešaka (eng. error state), kako bi programer mogao da se fokusira na pisanja logike stranice u zavisnosti od tih stanja, a ne da ih manuelno prati i ažurira.

Za ovaj rad izabrana je biblioteka Redux Toolkit sa svojim podmodulom Query, koji se češće nazivaju skraćeno RTKQ [16]. Ova biblioteka zapravo je nadogradnja na veoma poznatu biblioteku za upravljanjem stanjem[9] Redux, koja je dugo vremena bila neizostavan deo React ekosistema. Ipak, u ovom radu ne koristimo upravljanje stanjem u pravom smislu te reči, tako da upotreba biblioteke RTKQ ostaje ograničena isključivo na dohvatanje podataka.

### Deklaracija API sloja, upravljanje keš oznakama

U uvodnom delu ove sekcije pomenuta je deklarativnost kao jedna od tri funkcionalnosti koje očekujemo od moderne biblioteke za dohvatanje podataka. Deklarativnost je opšti pojam softverskog inženjerstva, i najlakši način da se opiše, neformalnim jezikom jeste „piši **šta** softver treba da uradi a ne **kako** to da uradi”.

[9]eng. state management library

U duhu takvog načina razmišljanja, bibliteka RTKQ oslobađa programera od repetitivnog zadatka pisanja API poziva za svaki mogući entitet. RTKQ takođe pruža i elegantan mehanizam upravljanja keširanjem podataka korišćenjem koncepta keš oznaka (eng. cache tags). Naime, ako najpre dohvatimo neki entitet preko određene API pristupne tačke metodom GET, a zatim ga ažuriramo metodom PATCH, prirodno je da želimo da se već dohvaćen entitet ažurira kako bi podaci prikazani korisniku bili najsvežiji. U frontend delu aplikacije sistema STS ovakve potrebe su intenzivne prilikom operacije ažuriranja tiketa. Primer 4.1 prikazuje kako se svi navedeni problemi rešavaju uz nekoliko jednostavnih konfiguracionih pravila koje zahteva RTKQ.

```
export const ticketsSlice = api.injectEndpoints({
  endpoints: (builder) => ({
    /* Definise se nacin na koji se dohvata tiket */
    getTicket: builder.query({
      query: ({ id, ...params }) => ({
        url: `/tickets/${id}`,
        params,
      }),
      /* Definisu se kes oznake */
      providesTags: (res) => {
        return res ? [{ type: 'getTicket', id: res.id }] : [];
      },
    }),
    addComment: builder.mutation({
      /* Definise se nacin na koji se menja tiket */
      query: ({ id, body, isInternal }) => ({
        url: `/tickets/${id}/comment/add`,
        method: 'PATCH',
        body: { body, isInternal },
      }),
      /* Definise se koje kes oznake se ponistavaju */
      invalidatesTags: ({ id }) => [{ type: 'getTicket', id }],
    }),
  }),
  overrideExisting: true,
});
```

Primer 4.1: Konfiguracija API poziva za dohvatanje tiketa i dodavanje komentara

## Izazivanje dohvatanja podataka i korišćenje u komponentama

Nakon definisanja konfiguracije u prethodnoj sekciji ostaje još pitanje kako koristimo objekte koje generiše RTKQ.

Treba još jednom istaći da RTKQ nije ništa drugo nego modul sistema `redux`[10], tako da je očekivano da RTKQ kompletno upravljanje podacima vrši upravo posredstvom `redux`-a i njegovog ugrađenog skladišta (eng. store)[11]. S tim u vidu, prilikom izazivanja dohvatanja podataka na serveru koristimo upravo `store` objekat i RTKQ-ov poziv `initiate` kako bi se dohvatanje započelo.

```
export const getServerSideProps = wrapper.getServerSideProps(
  (store) => async (context) => {
    const ticketId = context.params.ticketId;
    store.dispatch(
      ticketsSlice.endpoints.getTicket.initiate({
        id: ticketId,
        ...getTicketViewQueryParams(),
      }),
    );
    // ...
    return {};
  },
);
```

Primer 4.2: Dohvatanje podataka o tiketu na serveru.

S druge strane, podacima se u komponentama može pristupiti odgovarajućim React udicama (eng. hooks) [17] koje se takođe automatski generišu. Pozivanje odgovarajućeg hook-a izaziva dohvatanje podataka klijentski, ukoliko ti podaci već ne postoje u skladištu (ukoliko već nisu dohvaćeni serverski). Hook-ovi ispoljavaju, između ostalog, stanja učitavanja, greške ili uspeha.

[10] Tehnički, spada pod zaseban npm paket *redux-toolkit*, ali u govoru se često poistovećuju jer označavaju isti ekosistem.

[11] Način na koji radi biblioteka *redux* je kompleksan i može biti tema posebnog istraživanja, tako da se na ovom mestu zadržavamo samo na potrebnim detaljima.

```
function TicketViewPage() {
  const router = useRouter();
  const id = router.query.ticketId;

  const { isLoading, isFetching, isError } = useGetTicketQuery({
    id,
    ...getTicketViewQueryParams(),
  });
```

Primer 4.3: Korišćenje RTKQ hook-ova.

### Mutacije podataka

Mutacije podataka pomenute su ukratko u 4.2 i prikazana je njihova konfiguracija u propratnom primeru kôda. Mutacije se skoro po pravilu koriste klijentski[12], tako da za njih RTKQ obezbeđuje hook-ove isto kao i za obično dohvatanje podataka.

```
function TicketViewPage() {
  const [addComment,
        { isLoading, isSuccess }] = useAddCommentMutation();

  const handleSubmitComment = (comment) => {
    addComment({ id: ticket.id, body: comment, isInternal });
  };
```

Primer 4.4: Korišćenje RTKQ mutacija na primeru dodavanja komentara na tiket.

## 4.4 Internacionalizacija

Frontend, kao deo aplikacije koji je direktno zadužen za prikaz grafičkog korisničkog interfejsa, prirodno nosi pitanja koja se odnose na način prikazivanja sadržaja neke vrste.

Definicija internacionalizacije je veoma opšteg karaktera i na osnovu [18] svodi se na dizajn i razvoj sadržaja proizvoda, aplikacije ili dokumenta koji omogućava lako prilagođavanje kulturi, regionu ili jeziku. U užem smislu, odnosno iz ugla frontend razvoja, razmatraju se uglavnom sadržaji sledećeg tipa:

[12]Nema previše smisla izazivati mutacije podataka sa serverske strane, jer se uvek dešavaju na zahtev korisnika.

- **Tekstualni sadržaj.** Skoro svaki deo korisničkog interjsa sadrži neki tekstualni sadržaj. Iako je moguće takav sadržaj zapisati direktno u kôdu (eng. hardcode), ipak je bolje imati sistem koji tekstualne sadržaje drži izdvojeno i po potrebi omogućava višejezičnost - kada korisnik promeni željeni jezik menjaju se i svi tekstualni sadržaji bez ikakve specijalne izmene u kôdu.
- **Formati datuma i vremena.** Način zapisa datuma i vremena zavisi od regiona, tako da bi bilo dobro imati sistem takav da se te vrednosti automatski prilagođavaju korisničkom regionu.
- **Formati valuta.** Moguće je koristiti komponente koji automatksi konvertuju novčane iznose iz jedne valute u drugu.

U ovom radu urađena je samo prva stavka - podržani su srpski i engleksi jezik. Za potrebe čuvanja trenutnog jezika koristi se `Accept-Language` HTTP zaglavlje (eng. HTTP header) u kombinaciji sa odgovarajućom vrednošću `language_code` kolačića (eng. cookie) na klijentskoj strani.

U sekciji 2.1 pomenuto je kako sistem oznaka koristi internacionalizaciju za nazive i opise oznaka. Upravo podešavanje `Accept-Language` zaglavlja omogućuje da nam API vrati podatke na odgovarajućem jeziku.

```
const baseQuery = fetchBaseQuery({
  baseUrl: '...',
  prepareHeaders: (headers, api) => {
    // ...
    const ctx = api?.extra?.ctx;
    const languageCode =
      (isServer()
        ? parseCookie(ctx?.req?.headers?.cookie).language_code ?? ''
        : Cookies.get('language_code')) ?? 'en';

    headers.set('Accept-Language', languageCode);
    return headers;
  },
});
```

Primer 4.5: Postavljanje Accept-Language zaglavlja u RTKQ prilikom API zahteva.

Sa druge strane, za kontrolu prikaza sadržaja na grafičkom korisničkom interfejsu koristi se kombinacija biblioteka `react-intl` i `formatjs` [19]. Naime, tek-

stualni sadržaji, koje želimo da internacionalizujemo na sajtu, nazivaju se *poruke* (eng. messages) i definišu se odgovarajućim funkcijama u kôdu, zajedno sa svojim podrazumevanim vrednostima. Primer 4.6 prikazuje definisanje nekih poruka za stranicu koja prikazuje tiket.

Pokretanjem komande `npm run i18n`, definisane kao skripte u okviru fajla `package.json`, vrše se *ekstrakcija* i *kompilacija* poruka. Ekstrakcija predstavlja proces izvlačenja svih poruka iz fajlova, koji definišu poruke na način prikazan u primeru 4.6, u odgovarajuće JSON fajlove. Kompajliranje prevodi te fajlove u oblik prikladan korišćenju u produkciji. Delovi tih fajlova prikazani su u primeru 4.7.

Na kraju, poruke se koriste odgovarajućin hook-ovima ili komponentama biblioteke `react-intl`, kao što prikazuje primer 4.8

```
export const ticketViewMessages = defineMessages({
  cannotCommentOnThisTicket: {
    id: 'ticket-view.cannot-comment',
    defaultMessage: 'You cannot comment on this ticket',
  },
  publicCommentButtonText: {
    id: 'ticket-view.public-comment-button-text',
    defaultMessage: 'Public',
  },
  internalCommentButtonText: {
    id: 'ticket-view.internal-comment-button-text',
    defaultMessage: 'Internal',

```

Primer 4.6: Definisanje poruka.

Napomenućemo da bi sofisticirani sistem internacionalizacije podržao čuvanje vrednosti u bazi podataka i njihovo eksportovanje u JSON fajlove, čime bi se omogućilo dinamičko menjanje vrednosti translacija bez ponovnog pokretanja aplikacije. To je, ipak, u ovom radu ostavljeno kao mogućnost za poboljšanje.

```
// en.json - nekompajlirana verzija
{
  "admin-link.metrics": "Metrics",
  "admin-links.home-page": "Home",
  "admin-links.tags": "Tags",
  ...

// en.json - kompajlirana verzija
{
  "admin-link.metrics": [
    {
      "type": 0,
      "value": "Metrics"
    }
  ],
  "admin-links.home-page": [
    {
      "type": 0,
      "value": "Home"
    }
  ],
  "admin-links.tags": [
    {
      "type": 0,
      "value": "Tags"
    }
  ],
```

Primer 4.7: JSON fajlovi sa porukama.

```
const intl = useIntl();
// ...
return (<Button type="submit">
            {intl.formatMessage(formsMessages.submit)}
        </Button>)
```

Primer 4.8: Korišćenje poruka

# Glava 5

# Analitika

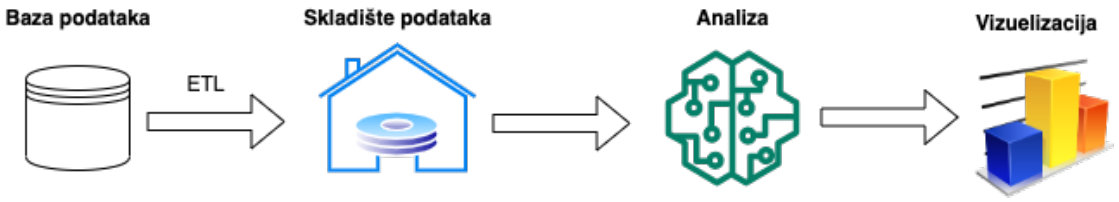
Jedan zadatak u okviru razvoja softvera jeste napraviti sve proizvodne funkcionalnosti - kreiranje tiketa, komentarisanje, promena statusa, zatvaranje, i sl. Iz ugla čistog softverskog inženjerstva moglo bi se reći da je posao završen onda kada su implementirane sve komponente sistema i njihova međusobna saradnja zarad izvršavanja poslovnih procesa.

Ovakav stav, očekivano, bio bi naivan. Problem leži u tome što ne postoji taj nivo istrage i domenskog znanja koji poslovođi može ukazati na sve potencijalne mane i moguća unapređenja sistema kao što mogu uraditi realni podaci generisani prilikom upotrebe softvera. Ovakav pristup ima za cilj da stvori ponavljajući ciklus koji se sastoji, grubo, iz sledećih koraka [20]:

1. Poslovne aktivnosti se pamte u transakcionoj bazi podataka.
2. Na osnovu tih podatka izvode se zaključci koristeći odgovarajuće tehnike i alate.
3. Dati zaključci ukazuju na šta se treba fokusirati u razvoju proizvoda.
4. Nove implementirane funkcionalnosti donose još podataka, koje dalje možemo analizirati ispočetka.

## 5.1 Skladišta podataka, ETL procesi, cron

Niz koraka iz uvodnog dela ovog poglavlja predstavlja uopšten mentalni model na kome se zasniva proces analitike podataka[1] iz poslovnog ugla. Nešto konkretniji model, koji se više fokusira na tehnologije i metodologije, prikazan je na dijagramu 5.1.




Slika 5.1: Proces analitike.

Za potrebe ovog rada, a iz razloga jer to nije glavni fokus, sistem analitike izveden je u uprošćenom obliku. Skladišta podataka (eng. data warehouse) predstavljaju zasebnu oblast izučavanja i u vreme pisanja rada potpadaju najviše pod oblast poznatoj kao inženjerstvo podataka (eng. data engineering). Skladište podataka smatraćemo za organizovanu kolekciju podataka dobijenu transformacijom izvornih podataka u oblik prikladan za izvršavanje odgovarajućih upita [20].

Sledeće prirodno pitanje koje se nameće jeste kako i kada se date transformacije vrše. U sistemu STS ovaj zadatak potpada pod ETL (eng. Extract Transform Load) servis. Kao što i samo ime kaže sastoji se od tri koraka:

1. **Extract** - izvlačenje podataka iz izvora.
2. **Transform** - transformacija u pogodan oblik.
3. **Load** - učitavanje u skladište podataka.

Metrike koje mogu imati vrednost u sistemu STS jesu primarno metrike koje se odnose na tiket, kao glavni entitet, i promene koje se nad njime dešavaju. Tako možemo razmatrati prosečno vreme rešavanja (eng. average resolution time), prosečno vreme preuzimanja (eng. average pickup time) i prosečno vreme prvog odgovora (eng. average first-response time). S tim u vidu, skladište podataka, i njegov deo koji se bavi tiketima, dizajniran je tako da izvuče navedene metrike

[1]Postoji mnoštvo termina koji se koriste u današnje vreme, kao što su *Data Mining, Business Intelligence* i drugi. Ne zadržavamo se na diskusiji konkretnih pojmova jer ne služe svrsi ovog rada.

za svaki tiket[2]. Primer 5.1 prikazuje glavni deo ETL skripte. Primer 5.2 prikazuje funkciju koja transformiše tiket, dok primer 5.3 prikazuje računanje jedne konkretne metrike.

Za potrebe automatizacije ovog procesa koristi se `cron`, ugrađeni planer poslova (eng. job scheduler) operativnih sistema zasnovanih na Unix-u. Njegovu konfiguraciju izostavljamo jer nema specifičnosti u odnosnu na najosnovnije korišćenje.

Primer 5.1: Transformacija tiketa.

```
def main():
    # ...
    # Nalazimo poslednji obradjeni tiket i njegovo vreme,
    # ako postoji
    latest_processed_ticket = dest_tickets_collection.find_one(
        sort=[("timestamp", pymongo.DESCENDING)]
    )
    latest_timestamp = \
        latest_processed_ticket["timestamp"] \
        if latest_processed_ticket else None

    # Uzimamo sve neobradjene tikete
    query = {}
    if latest_timestamp:
        query["timestamp"] = {"$gte": latest_timestamp}

    unprocessed_tickets_cursor = src_tickets_collection.find(query)

    # Transformisemo sve neobradjene tikete
    new_tickets = []
    for ticket in unprocessed_tickets_cursor:

        transformed_ticket = transform_data(ticket)
        new_tickets.append(transformed_ticket)

    # Ubacujemo u skladiste
    if new_tickets:
        dest_tickets_collection.insert_many(new_tickets)
```

[2]Vredi napomenuti kako je izbor šablona Event Sourcing za čuvanje tiketa i niza promena prikazan u sekciji 3.3 doprineo znatno lakšoj implementaciji ovih transformacija.

Primer 5.2: Funkcija koja transformiše podatke.

```
def transform_data(ticket):
    created_at = ticket.get('createdAt')
    resolution_time, resolution_timestamp = \
        calculate_resolution_time(ticket)
    pickup_time, pickup_timestamp = \
        calculate_pickup_time(ticket)
    first_response_time, first_response_timestamp \
        = calculate_first_response_time(ticket)
    status = ticket.get('status')

    return {"_id": ticket.get('_id'),
            "timestamp": created_at,
            "resolution_time": resolution_time,
            "resolved_at": resolution_timestamp,
            "pickup_time": pickup_time,
            "picked_up_at": pickup_timestamp,
            "first_response_time": first_response_time,
            "first_responded_at": first_response_timestamp,
            "status": status}
```

## 5.2 Prikaz metrika, biblioteka streamlit

Nakon uspešnog punjenja skladišta podataka postoje dva pravca, kojima tok podataka može dalje ići. Pritom treba imati u vidu da ta dva pravca nisu međusobno isključiva.

- Podaci iz skladišta mogu se koristiti za naprednu analizu statističkim metodama ili metodama mašinskog učenja.
- Podaci se mogu vizuelizovati određenim kontrolnim tablama (eng. dashboards) zarad brzog dobijanja slike o podacima.

Kao što je već napomenuto, napredne analize podataka ne spadaju u područje ovog rada, ali jednostavna vizuelizacija podataka izvučenih u prethodnom poglavlju implementirana je *python* bibliotekom *streamlit* [21].

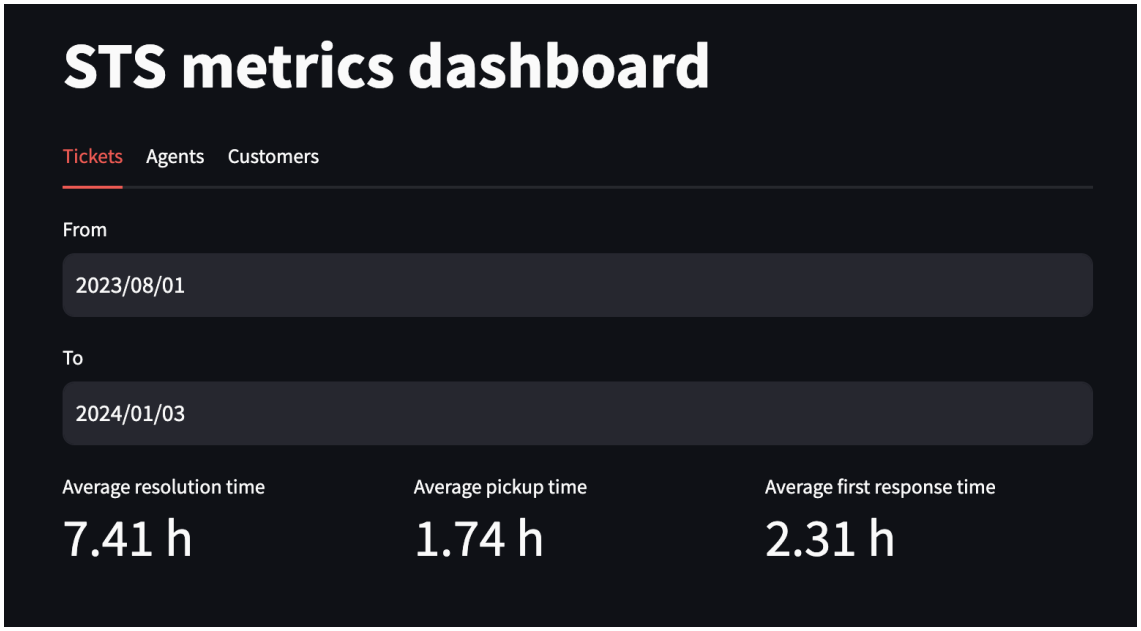
Ova biblioteka omogućava jednostavno kreiranje interaktivnih kontrolnih tabli za prikaz podataka. Glavna prednost je u tome što sadrži veliki broj predefinisanih elemenata korisničkog interfejsa (grafikoni, histograni, prikaz vremena, elementi

Primer 5.3: Računanje konkretne metrike.

```
def calculate_resolution_time(ticket):
    created_at = ticket.get('createdAt')

    history = ticket.get('history')
    resolution_time_items = list(filter( \
        lambda item: item.get('type') == 'STATUS_CHANGED' \
        and item.get('payload').get('status') == 'RESOLVED', \
        history))

    resolved_time = resolution_time_items[-1].get(
        'timestamp') if resolution_time_items else None

    resolution_time = \
        (resolved_time - created_at).total_seconds() / 60 \
        if resolved_time else None

    return resolution_time, resolved_time
```

za odabir opsega datuma, i slično) što omogućava programeru da se fokusira na poslovne vrednosti umesto na tehnikalije izgradnje takvih elemenata.

Skripta za prikaz kontrolne table je jednostavna i može se videti u fajlu `services/analytics/app.py`. Sastoji se od dohvatanja informacija o tiketima za izabran vremenski okvir i prikaz prosečnih vrednosti statistika iz prethodnog poglavlja. Slika 5.2 prikazuje izgleda korisničkog interfejsa ove kontrolne table.




Slika 5.2: Kontrolna tabla STS analitike.

# Glava 6

# DevOps

Pojam DevOps (nastao od engleske sintagme *Development Operations* - u bukvalnom prevodu *razvojne operacije*) predstavlja relativno mladu granu softverskog inženjerstva. U svojoj najužoj definiciji, prema [22], kaže se da DevOps opisuje usvajanje iterativnog razvoja softvera, automatizaciju i programabilne infrastrukture za otpremanje[1] i održavanje softverskog sistema.

U nekim krugovima poistovećuje se sa ulogom sistemskih administratora dok drugi tvrde da je to potpuno zasebna profesija. Čiju god stranu zauzeli moramo se složiti da sa masovnim razvojem različitih tehnologija, okruženja i platformi nastaju novi problemi koji ne potpadaju pod tradicionali pojam softverskog inženjerstva gde uglavnom obuhvatamo razvoj domenskih funkcionalnosti. Naime, ukoliko kažemo da je zadatak *„implementirati dugme na stranici za prikaz tiketa kojim se zatvara tiket"* potpuno je jasno da se tim zadatkom neće baviti DevOps inženjer, dok bi na primer zadatak *implementirati sistem za periodično pravljenje i čuvanje rezervnih kopija baze podataka* upravo zapao tom inženjeru.

Interesantno opažanje primećeno u [23] je da u mnogim kompanijama postoji neprekidna kontrateža između odeljenja za razvoj i odeljenja za razvojne operacije, uprkos činjenici da je kranji cilj isti. Na primer, inženjeru kome je zadatak da implementira jedan mali server za nekoliko API pristupnih tačaka od nule biće lakše da taj server pokrene direktno na produkcionoj mašini i neretko će zahtevati da se „napravi izuzetak" usled skromne veličine projekta. DevOps inženjer bi skoro sigurno insistirao na tome da se čak i za najmanji takav server napravi odgovarajuća infrastruktura, recimo kroz Docker. Lako je zapitati se zašto se uopšte dovodi u pitanje vrednost poštovanja principa razvojnih operacija i zašto bi programer

[1] eng. deployment

iz hipotetičkog scenarija ikada zahtevao da se napravi izuzetak. Odgovori na ova pitanja zalaze u poslovne aspekte softverskog inženjerstva gde je uvek pitanje toga ko donosi veću vrednost. Ovaj rad se uzdržava od dalje diskusije na tu temu jer je fokus primarno tehnički, ali napomenućemo da su istorijat DevOps-a i razni poslovni aspekti ove grane odlično obrađeni u [23].

## 6.1 Kontejneri, sistem Docker

Istorija nastajanja Docker-a, i kontejnerizacije uopšte, interesantna je tema sama za sebe i u [24] izložena ukratko na sledeći način:

1. **Korišćenje zasebnih servera za svaku aplikaciju.** U početku je bilo komplikovano, pa čak i nemoguće pokrenuti više zasebnih servera na istoj mašini, tako da bi kompanije često išle u pravcu kupovanja ili iznajmljivanja zasebnih fizičkih servera za svaki aplikativni server.

2. **Virtuelne mašine.** Koncept koji je značajno olakšao stvari bile su virtuelne mašine. Dozvoljavale su pokretanje proizvoljnog broja istinski odvojenih okruženja na jednoj fizičkoj mašini. Mana ovakvog pristupa jeste da virtuelna mašina zahteva kompletan operativni sistem, a samim tim i dosta resursa u vidu RAM-a i procesorskog vremena.

3. **Kontejneri.** Kao rešenje koje ima za cilj da uzme najbolje od oba sveta nastaju kontejneri (eng. containters). Za razliku od virtuelnih mašina, svaki pokrenut kontejner koristi operativni sistem računara na kome se pokreće, a ipak održava nivo potpune izolovanosti.

Docker je jedno od rešenja za rad sa kontejnerima, i izabrano je za potrebe ovog rada usled svoje popularnosti i rasprostranjenosti. Za mnoge programere, čija je primarna delatnost razvoj poslovnih funckionalnosti, Docker predstavlja crnu kutiju. Pojmovi kao što su *docker engine*, *slike (eng. images)* i *orkestracija* stvaraju konfuziju ako se od početka ne posveti pažnja njihovom razumevanju, stoga ćemo se prilikom objašnjavanja koristiti analogijama sa nekim konceptima koji su poznati svim pogramerima.

## Docker slike

Na početku poglavlja pomenuto je kako je jedan od zadataka DevOps-a, između ostalog, uspostavljanje programabilnog infrastruktura za otpremanje i održavanje softverskog sistema. Način na koji pravimo docker slike upravo se zasniva na programabilnosti.

Naime, docker slike treba shvatiti upravo po inspiraciji zbog koje se koriste - potrebno nam je izolovano okruženje jako specifične namene. Pomenuli smo da virtuelne mašine nose sa sobom previše stvari koje nam nisu potrebne, najpre ceo operativni sistem i sve njegove module. Definicijom adekvatne docker slike izbegavamo korišćenje bespotrebnih resursa.

Dakle, prilikom definisanja slike za pokretanje našeg bekend servera u režimu za razvoj[2] za sistem STS, možemo pristupiti sledećim rezonovanjem:

1. Potreban nam je Node.js
2. Potreban nam je CLI za okruženje `@nestjs`
3. Potrebno je instalirati sve pakete deklarisane u fajlu `package.json`
4. Potrebno je prekopirati izvorni kôd
5. Potrebno je pokrenuti razvojni server

Jezik kojim se prave docker slike upravo je pravljen da što bliže oslika prirodni jezik opisivanja sistema, i prikazan je na 6.1.

## Docker kontejneri

Ako bismo postavljali analogiju između razvoja slika i kontejnera sa običnim razvojem aplikacija, onda bismo mogli reći da slika odgovara programu, a kontejner odgovara procesu. Na taj način gore definisana docker slika sama po sebi ne radi ništa ukoliko se ne pokrene u vidu kontejnera.

Nastavljajući analogiju, sliku moramo prvi izgraditi (eng. build) što bi odgovaralo procesu kompilacije. Ovu kompilaciju je bolje shvatiti ne kao prevod na mašinski jezik, već kao prevod u međureprezentaciju, koju je u stanju da izvršava docker engine. Docker CLI komandom, na primeru 6.2, možemo izgraditi sliku sa odgovarajućom oznakom (eng. tag).

[2] O produkcionoj postavci biće reči kasnije.

Primer 6.1: Slika za bekend server

```
FROM node:18-alpine3.16

# Basic setup
WORKDIR /app
ENV NODE_ENV=development

# Install nestjs cli
RUN npm i -g @nestjs/cli@9.3.0

# Install dependencies first to utilize build cache
COPY package*.json ./
RUN npm install

# Copy the sourcecode
COPY . .

# Start the development server
CMD ["npm", "run", "start:dev"]
```

Primer 6.2: Izgradnja slike

```
docker build -f
    services/api-main/docker/Dockerfile.development
    -t sts-api-local
    ./services/api-main
```

Nakon izgradnje moguće je pokrenuti datu sliku u vidu kontejnera, sa nekoliko dodatnih konfiguracija koje se sastoje od sledećeg:

- **Vezivanje portova.** Opcijom `-p` možemo navesti koji port kontejnera se vezuje za koji port domaćinskog operativnog sistema (eng. host OS). Ovo omogućava da aplikacija unutar kontejnera bude agnostična od zauzetosti portova.
- **Povezivanje sa fajl sistemom.** U lokalnom razvojnom okruženju potreban nam je izvorni kôd aplikacije, koji možemo uvesti korišćenjem koncepta *docker volume*-a. Oni predstavljaju apstrakciju medijskih uređaja, ali u ovom slučaju koristimo ih da povežemo određeni direktorijum unutar kontejnera sa hostom[3].

[3]Ovo se ostvaruje tehnologijom rsync. Izostavljamo detalje kako se to interno postiže, ali

Primer komande za pokretanje prikazan je u 6.3.

Primer 6.3: Pokretanje kontejnera

```
docker run
    sts-api-local
    -v ./services/api-main:/app
    -p 3000:3000
```

Na ovaj način analogijom možemo reći da je naš kompajliran program pokrenut i sada predstavlja proces.

## Orkestracija putem modula docker-compose

Proces prikazan u prethodne dve sekcije izgleda previše repetitivno i, još bitnije, deluje da se može automatizovati. Ako bismo ovako razvijali aplikaciju morali bismo konstantno da ponavljamo korake:

1. Izgradi sve slike komandama `docker build`.
2. Pokreni sve kontejnere komandama `docker run`.

Pri tom moramo uvek voditi računa da li odgovarajuće portove dobro vazujemo, da li vezujemo odgovarajuće fajlove i slično. Dijagram ovog procesa prikazan je na slici 6.1. Čitav ovaj proces upravljanja stanjima kontejnera, njihovim pokretanjem i obaranjem, portovima i vezivanjem sa fajl sistemom podvodimo pod jedan sveobuhvatni pojam *orkestracije*.

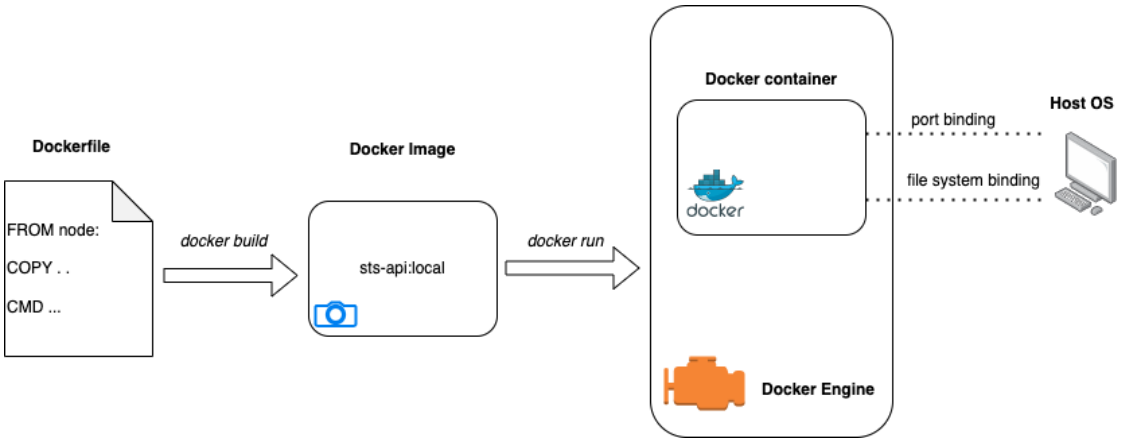
Postoje vrlo sofisticirani softverski sistemi za orkestraciju kontejnera, od kojih se neki pokreću primarno u oblaku[4]. Sistem STS nije dizajniran za cloud okruženja već za varijantu samostalnog hostovanja (eng. self-hosting)[5]. Zbog toga orkestrator docker-compose zadovoljava sve zahteve.

Skoro svi orkestratori zasnivaju se na konfiguraciji umesto na eksplicitnom programiranju ponašanja kontejnera (kao što smo prvobitni uradili koristeći `bash` skripte iz prethodnih sekcija). U slučaju docker-compose-a to se postiže pisanjem

napominjemo, jer se u praksi često koristi izraz rsync kao glagol. Neretko ćemo čuti da se kaže da su dva foldera „rsync-ovana".

[4]Primer takvog sistema je AWS ECS.

[5]Jedna zanimljivost je da tokom pisanja ovog rada postoji veliki raskol u onlajn zajednicama programera: da li je bolje koristiti cloud rešenja ili samostalno hostovanje. Slična rasprava vodi se oko promovisanja monolitnih okruženja kao superiornih u odnosu na mikroservise, što nekima deluje kao vraćanje u prošlost.




Slika 6.1: Proces pokretanja kontejnera.

YAML fajlova. Konfiguracija u ovom fajlovima navodi koje sve servise sistem sadrži, koju sliku koriste ili na koji način se izgrađuju, konfiguracije portova i vezivanja fajl sistema, promenljive okruženja i slično. Primer konfigurisanja bekend i frontend servisa za lokalno okruženje dat je u 6.4.

Nakon ovakvog konfigurisanja čitavog sistema, upravljanje kompletnim okruženjem poprilično je jednostavno i može se postići sa nekoliko jednostavnih komandi prikazanih u 6.5.

Primer 6.4: docker-compose konfiguracija nekih razvojnih servisa.

```
services:
  api_main:
    container_name: api-main-local
    build:
      context: ./services/api-main
      dockerfile: ./docker/Dockerfile.development
    environment:
      MAIN_DB_USERNAME: ${MAIN_DB_USERNAME}
      MAIN_DB_PWD: ${MAIN_DB_PWD}
      JWT_SECRET: ${JWT_SECRET}
      FIREBASE_KEY_PATH: "/app/certificates/sts-firebase-key.json"
    ports:
      - "3001:3000"
    volumes:
      - "./services/api-main:/app"
    extra_hosts:
      - "host.docker.internal:host-gateway"

  ssr:
    container_name: ssr-local
    build:
      context: ./services/ssr
      dockerfile: ./docker/Dockerfile.development
    ports:
      - "3003:3000"
    environment:
      NODE_ENV: "development"
      NODE_TLS_REJECT_UNAUTHORIZED: "0"
    volumes:
      - "./services/ssr:/app/"
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

Primer 6.5: docker-compose komande.

```
# Pokretanje svih servisa
docker-compose up -d

# Obaranje svih servisa
docker-compose down
```

Napomenućemo da docker-compose podržava i kompleksnije operacije kao što su skaliranje kontejnera[6], kojima se u ovom radu ne bavimo. Napominjemo tu činjenicu kako bismo obeshrabrili popularno mišljenje da za bilo kakav ozbiljan rad sa kontejnerima moramo odmah pribegavati naprednim sistemima kao što su *AWS ECS* ili *kubernetes*. Naravno da ti sistemi imaju svoju ulogu i veoma su moćni, ali njihovo preuranjeno korišćenje može doneti više štete nego koristi razvoju.

## 6.2 Reverzni proksi nginx

Uspostavljanjem docker-a kao sistema za pokretanje aplikacije rešili smo jedan problem - izolacija okruženja. U ovom trenutku imamo pet servisa pokrenutih kroz različite kontejnere i povezane na različite portove[7]. Sledeće pitanje koje se prirodno nameće jeste kako korisnik pristupa veb sajtu i kako servisi komuniciraju jedni između drugih. Uzevši u obzir da se svi servisi pokreću na različitim portovima, bilo bi neodrživo da svaka komponenta sistema mora da zna na kojim portovima su ostale, a potpuno apsurdno da korisnik sajta mora da zna na kom portu je hostovan glavni vebsajt (frontend).

Očigledno je da nam je potrebna pristupna tačka celom sistemu za spoljašnji svet - jednom rečju potreban nam je **veb server**. Pri početku pogavlja 3 izričito je napomenuta razlika između veb servera i aplikativnog servera. Pod aplikativnim serverom mislimo na HTTP server u odgovarajućem veb okruženju (u našem slučaju Next.js ili Nest.js) koji služi za obradu zahteva ili (u slučaju serverskog renderovanja) generisanje HTML-a. Ovi serveri treba da se bave isključivo svojim ulogama i da se ne brinu o stvarima kao što su provera SSL sertifikata, serviranje statičkih fajlova i slično. Navedeni zadaci, naravno uz još mnoge, potpadaju pod ulogu veb servera.

Tok korisničkog zahteva uprošćeno možemo shvatiti na sledeći način, kao što je prikazano u [25]:

1. Korisnik pokuša da pristupi određenom resursu putem URL-a.

[6]Uz propratne module kao što su docker-swarm.

[7]Servisi su: baza podataka, Nest.js server, Next.js server, ETL i streamlit.

2. Zahtev prvo stiže do veb servera[8]. Ukoliko se koristi HTTPS, veb server vrši proveru sertifikata i bezbednosti konekcije.

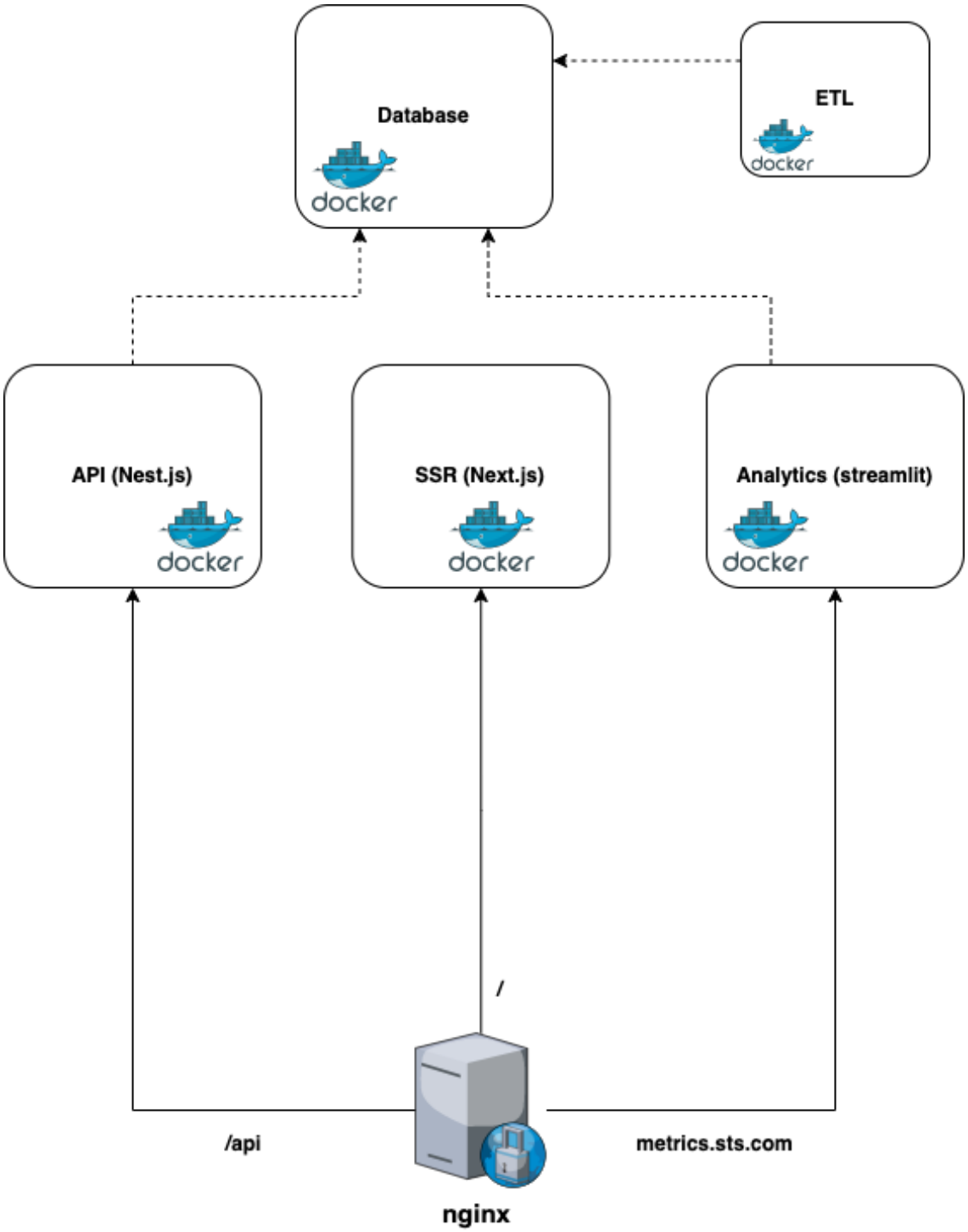
3. U zavisnosti od URL-a, veb server odlučuje da li može sam da odgovori ili ne. Ukoliko može, na primer ako je zahtev za statičkim sadržajem kao što su slike, veb server odgovara i komunikacija se završava.

4. U suprotnom veb server delegira posao odgovarajućem aplikativnom serveru.

*nginx* [26] nam omogućava sve od navedenog. Kada ga koristimo za delegiranje zadataka aplikativnim serverima, onda kažemo da zauzima ulogu obrnutog ili reverznog proksija (eng. reverse proxy). Razlog takvog imenovanja jeste upravo u smeru prosleđivanja zahteva - klasičan proksi prosleđuje zahteve eksternim servisima, dok reverzni proksi prosleđuje zahteve „sam na sebe".

Dijagram 6.2 prikazuje nacrt celog sistema.

[8]Ukoliko se dodatno koristi mreža za dostavljanje sadržaja (eng. Content Delivery Network - CDN), recimo Cloudflare, onda bi zahtev tehnički prvo prošao kroz nju, ali svakako veb server treba da bude agnostičan od toga.




Slika 6.2: Nacrt sistema.

## 6.3 Produkciono okruženje, višefazna izgradnja slika, DockerHub

Za kraj napominjemo male razlike između konfiguracije lokalnog okruženja, prikazanog u prethodnim sekcijama, i produkcionog, od čega je najveća razlike u docker slikama. Naime, u produkciji nema potrebe da otpremamo izvorni kôd aplikacija, kao što to radimo u lokalnom okruženju[9], već u docker sliku možemo kopirati samo neophodne fajlove za pokretanje.

### Višefazna izgradnja

Na primer, u slučaju Nest.js API-a dovoljno je imati samo nekoliko fajlova i foldera:

- `node_modules` folder koji sadrži instalirane Node.js pakete.
- `dist` folder koji sadrži kompajliran produkcioni kôd.
- `package.json` fajl kako bismo pokretali skripte.

Izgradnju docker slika koje sadrže samo potrebne fajlove, i odbacuju sve što nije potrebno usput, možemo postići korišćenjem višefaznih izgradnji (eng. multiphase build). Dockerfile možemo organizovati u više faza, gde iz svake faze kopiramo isključivo fajlove koji su potrebni na samom kraju. Ovakav način izgradnje slika prikazan je u primeru 6.6.

[9]U lokalnom okruženju nam je potreban izvorni kôd kako bismo mogli da razvijamo uživo, tj. da se promene u kôdu odmah reflektuju.

Primer 6.6: Višefazna izgradnja slike za API servis.

```
# Install deps
FROM node:18-alpine3.16 as deps
WORKDIR /app
COPY ./package.json ./
COPY ./package-lock.json ./
RUN npm ci

# Build
FROM node:18-alpine3.16 as builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN npm run build

# Run
FROM node:18-alpine3.16 as runner
WORKDIR /app
ENV NODE_ENV production
COPY --from=deps /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/dist ./dist
EXPOSE 3000
ENV PORT 3000
CMD ["node", "dist/main"]
```

## DockerHub

Kao dodatno olakšanje za pokretanje produkcione varijante sistema STS, sve slike možemo otpremiti na DockerHub, koji predstavlja onlajn servis za pronalaženje i deljenje docker slika [27]. Besplatan plan koji nudi DockerHub, u trenutku pisanja ovog rada, omogućava hostovanje neograničeno mnogo javnih slika, što je više nego dovoljno za potrebe sistema STS.

S tim u vidu napravljene su produkcione skripte, koje se nalaze u folderu `bin` u izvornom kôdu i bave se izgradnjom i otpremanjem produkcionih slika. Primer ovih komandi vidimo u 6.7.

Primer 6.7: docker-compose komande.

```
# Komanda za izgradnju API slike (fajl sts-build)
docker build -f
```

```
    ./services/api-main/docker/Dockerfile.production
    -t aleksakojadinovic/sts:api-latest
    ./services/api-main

# Komanda za otpremanje slike (fajl sts-push)
docker push aleksakojadinovic/sts:api-latest
```

Produkciona docker-compose konfiguracija sada ne mora uopšte zavisiti od postojanja Dockerfile-ova, već može direktno pokretati slike sa DockerHub-a. Na taj način postigli smo da na konačnom fizičkom[10] serveru, na kom pokrećemo sistem, ne moramo niti imati izvorni kôd niti klonirati git repozitorijum izvornog kôda - možemo docker-compose konfiguraciju držati u zasebnom repozitorijumu. Ipak je sve obuhvaćeno u jednom repozitorijumu kako bi služio kao centralni izvor svega pokrivenog u radu.

[10]Naglašavamo fizičkom jer mislimo na pravu instancu servera gde ćemo pokretati docker. Ipak pojam fizičkog servera i ovde treba uzeti sa rezervom jer su u oblaku i takvi serveri virtuelni samo na višem nivou. Svakako iz našeg ugla smatramo ih za fizičke servere.

# Glava 7

# Zaključak

Dostupne tehnologije za razvoj veb aplikacija su mnogobrojne i trendovi ukazuju na to da će ih biti sve više. Tehnološki stekovi (eng. tech stack) mogu se neformalno svrstati u dve grupe. Prva bi se sastojala od tradicionalnih ekosistema poput onih vezanih za jezik PHP[1] ili Javu[2], dok bi se u drugu mogli svrstati moderne Javascript biblioteke, od kojih su neke prikazane. Nije retka pojava izjednačavanje tradicionalnih okruženja sa kvalitetom i stabilnošću, i stav da se jedino u njima može pisati ozbiljan softver. Isto tako postoji i pogrešno verovanje u suprotnom smeru - novi Javascript ekosistemi označavaju se kao infantilni uprkos dugogodišnjem postojanju.

Cilj ovog rada je višestruk. Sa jedne strane služi da potvrdi činjenicu da je dobra infrastruktura i dizajn softvera agnostičan od konkretnih tehnologija. Druga svrha jeste prikaz kompletnog procesa razvoja složenog softverskog sistema u modernim tehnologijama, a pritom ne pribegavajući brzim rešenjima koja okruženja nude već prateći ustanovljene principe.

Već u prvom poglavlju dat je primer dizajna nerelacione baze podataka putem ER modela, koji se često vezuje čvrsto za relacione baze. Pritom je iskorišćen šablon Event Sourcing zarad povećanja fleksibilnosti tiketa. Kasnije, uvođenjem okruženja NestJS i jezika `typescript`, a uz podršku paketa `mongoose`, implementacija principima domenskog dizajna omogućila je maksimalnu izolovanost slojeva. Razlika između osnovnog entiteta kroz tri sloja - infrastrukturni, domenski i API - demonstrira istinsku izolovanost komponenti i omogućava razmišljanje o izdva-

[1]Uz koji često ide MySQL baza podataka, sa podrškom okruženja kao što je CodeIgniter, Symfony ili Laravel.

[2]Npr. okruženje Spring Boot.

janju u zasebne pakete, ili čak mikroservise.

Frontend svet je u trenutku pisanja raznovrsan i podeljen. Pojava da velika kompanija kao što je Vercel predvodi okruženje otvorenog kôda (Next.js) dovodi do toga da marketing i popularnost utiče direktno na šablone razvoja softvera[3]. Ovaj rad je iz tog ugla stremio da uzme najbolje iz oba sveta - iskorišćen je pun potencijal serverskog renderovanja Next.js-a, a uz podršku stabilnih ekosistemskih paketa poput redux-a.

Poglavlje o DevOps-u prikazuje moć Docker-a kao univerzalnog alata za izolaciju okruženja i otpremanje softvera. Za ogromne sisteme cloud rešenja koja nude veliki provajderi poput Amazona ili Google-a mogu biti dobar izbor, ali treba imati u vidu da kombinacija alata kao što su Docker i nginx mogu zadovoljiti većinu potreba.

Analitika može biti tema za zasebno istraživanje, ali je uspešno uspostavljena platforma za njen dalji razvoj.

Naravno, sistem STS, u obliku prikazanom u ovom radu ima očiglednih nedostataka. Najočigledniji nedostatak jesu testovi - jedinični testovi (eng. unit tests) bili bi krucijalan deo za garanciju kvaliteta softvera i deo procesa otpremanja. Zamerka se može ukazati i na infrastruktuni sloj bekenda i njegov strogi izbor pohlepnog učitavanja - neki sistemi omogućavaju odabir da li je učitavanje lenjo ili pohlepno na nivou entiteta. Nedostatak tipiziranja u frontendu, odnosno korišćenje čistog jezika `javascript`, stvar je izbora zarad izbegavanja kompleksnosti. Iz ugla osiguravanja podataka, automatske skripte za generisanje rezervnih kopija baze podataka bile bi neizostavan deo ozbiljnog softverskog sistema u produkciji, što se može postići kako manuelno tako i korišćenjem baza podataka u oblaku. Analitički servis, kao što je pomenuto, jeste minimalan i služi isključivo za demonstraciju takvih mogućnosti.

Sve u svemu, trenutna implementacija sistema STS čini solidnu osnovnu za dalji razvoj. Izolovanost komponenti omogućava i laku podelu poslova po timovima, kako bi svaki tim mogao da se fokusira na unapređenje pojedinačnih modula. Programski kôd u repozitorijumu može poslužiti i kao šablon za buduće projekte (eng. boilerplate) sa izborom istih tehnologija.

[3]Na primer, moguće je direktno se povezati na bazu podataka u Next.js-u i vršiti upite na bazi zahteva. Jasno je da ovde sprega između frontend dela aplikacije i baze podataka ogromna, a ipak se ovakav pristup reklamira kao poželjan u određenim Next.js kursevima.

# Bibliografija


[1] M. Gaedke and G. Gräf, "Development and evolution of web-applications using the webcomposition process model," pp. 58–76, 01 2001.

[2] T. J. Teorey, *Database modeling and design.* The Morgan Kaufmann Series in Data Management Systems, Oxford, England: Morgan Kaufmann, 5 ed., Oct. 2005.

[3] InfoQ Team, *Domain Driven Design Quickly.* Morrisville, NC: Lulu.com, June 2006.

[4] E. Evans, *Domain-driven design.* Boston, MA: Addison-Wesley Educational, Aug. 2003.

[5] NestJs, "NestJS Documentation." `https://docs.nestjs.com/`, 2023.

[6] express.js, "Express.js Documentation." `https://expressjs.com`, 2023.

[7] Microsoft, "Microsoft Dotnet architecture," 2023.

[8] Mongoose, "Mongoose documentation." `https://mongoosejs.com/docs/`, 2023.

[9] tsautomapper, "@automapper/js documentation." `https://automapperts.netlify.app/`, 2023.

[10] M. Masse, *Rest api design rulebook.* O'Reilly Media, Inc.

[11] K. Simpson, *You don't know js: Scope & closures.* O'Reilly Media, Mar. 2014.

[12] Vercel, "Next.js documentation." `https://nextjs.org/docs`, 2023.

[13] Remix, "Remix documentation." `https://remix.run/docs/en/main`, 2023.

[14] SvelteKit, "SvelteKit documentation." `https://kit.svelte.dev/docs/introduction`, 2023.

[15] React Team, "React documentation - components." `https://react.dev/learn/your-first-component`, 2023.

[16] Redux team, "RTKQ docs." `https://redux-toolkit.js.org/rtk-query/overview`, 2023.

[17] React Team, "React documentation - hooks." `https://react.dev/learn#using-hooks`, 2023.

[18] W3C, "Localization vs Internationalization." `https://www.w3.org/International/questions/qa-i18n`, 2023.

[19] FormatJS, "FormatJs Docs." `https://formatjs.io/docs/react-intl/`, 2023.

[20] A. Maheshwari, *Data Analytics Made Accessible*. Internet Archive, 2020.

[21] StreamLit, "Streamlit docs." https://docs.streamlit.io/. Accessed: 2024-01-03.

[22] M. Courtemanche, E. Mell, and A. S. Gillis, "What is devops? the ultimate guide." `https://www.techtarget.com/searchitoperations/definition/DevOps`. Accessed: 2023-12-23.

[23] G. Kim, J. Willis, P. Debois, and J. Humble, *The DevOPS handbook*. Portland, OR: IT Revolution Press, Dec. 2016.

[24] N. Poulton, *Docker deep dive*. July 2017.

[25] AWS, "Web sever vs application server." `https://aws.amazon.com/compare/the-difference-between-web-server-and-application-server/`. Accessed: 2023-12-30.

[26] Nginx, "nginx documentation." `https://nginx.org/en/docs/`. Accessed: 2023-12-30.

[27] Docker, "Dockerhub docs." `https://docs.docker.com/docker-hub/`. Accessed: 2023-12-30.

# Biografija autora

**Aleksa Kojadinović** rođen je 1998. godine u Užicu, gde završava osnovno i srednje obrazovanje. Nakon toga upisuje informatički smer Matematičkog fakulteta u Beogradu. Osnovne studije završava 2021. godine, a odmah nakon diplomiranja upisuje master studije na istom smeru. U međuvremenu zapošljava se u kompaniji FishingBooker kao Frontend softverski inženjer, specijalizovan za Javascript tehnologije.