UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Vukan Antić

UPOREĐIVANJE PROTOKOLA ZA KOMUNIKACIJU GRAPHQL, GRPC I REST, RAZVOJEM MOBILNE APLIKACIJE UPOTREBOM TEHNOLOGIJA SPRING BOOT I REACT NATIVE

master rad

Beograd, 2026.

Mentor:

dr Vladimir Filipović, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Aleksandar Kartelj, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Staša Vujičić Stanković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Ovaj rad posvećujem svom ocu, majci, bratu, prijateljima i kolegama. Mentoru i članovima komisije se zahvaljujem na strpljenju kao i savetima u toku izrade ovog rada. Posebnu zahvalnost dugujem devojci Dušici, na svoj ljubavi pruženoj u toku studija.

Naslov master rada: Upoređivanje protokola za komunikaciju GraphQL, gRPC i REST, razvojem mobilne aplikacije upotrebom tehnologija Spring Boot i React Native

Rezime: Sa porastom broja mobilnih aplikacija, komunikacija između klijenta i servera najčešće se svodi na upotrebu arhitektonskog stila REST. Pored njega postoje tehnologije GraphQL i gRPC, koji donose drugačije ustupke po pitanju fleksibilnosti, veličine poruka i dogovora između dve strane.

U ovom radu razvijena je mikroservisna aplikacija za svakodnevni prikaz umetničkih dela, upotrebom tehnologija Spring Boot i React Native. Prikazan je postupak prikupljanja i obrade podataka iz javno dostupne arhive Instituta za umetnost u Čikagu, podela serverske strane na mikroservise, organizacija svakog servisa po arhitekturi slojeva luka, kao i klijentska aplikacija zasnovana na obrascu model-prikaz-kontroler. Nad tako izgrađenim sistemom, implementirana su sva tri protokola, pri čemu su ostali slojevi aplikacije ostali nepromenjeni, čime je omogućeno njihovo neposredno kvalitativno poređenje. Za svaki protokol prikazani su osnovna zamisao, implementacija na serveru i na klijentu, kao i prednosti i nedostaci uočeni pri radu. Razvijena aplikacija dostupna je kao softver otvorenog koda.

Ključne reči: protokoli komunikacije, REST, GraphQL, gRPC, arhitektura slojeva luka, React Native, Spring Boot

Sadržaj

Glava 1

Uvod

Sa rastućom upotrebom mobilnih aplikacija komunikacija između klijenta i servera, u većini slučajeva, svodi se na REST [1], iako postoje i druge tehnologije poput gRPC [2] i GraphQL [3],koji donose drugačije ustupke po pitanju fleksibilnosti, veličine poruka i dogovora između dve strane. Zato su u toku izrade naredne aplikacije sve tri tehnologije implementirane, da bi se pokazalo u kojim situacijama je najbolje koristiti svaki od njih.

Sam projekat predstavlja aplikaciju otvorenog koda[1] , koja svakodnevno daje korisniku umetničku sliku, sa detaljnim opisom, i mogućnost da sačuva omiljenu, radi daljeg preporučivanja novog sadržaja. Pored toga, postoje i funkcionalnosti čuvanja omiljenih slika u memoriji sopstvenog telefona, kao i deljenje slika sa drugim korisnicima. Takođe, pri izlasku nove slike, korisnik dobija obaveštenje o novom sadržaju koji je dodat za njega.

Izradu projekta čine sledeći delovi:

- Prikupljanje podataka - U glavi 2 predstavljen je detaljan opis odakle je aplikacija dobila podatke za prikaz u okviru same aplikacije.
- Izrada servera zasnovanog na mikroservisnoj arhitekturi [4] - U glavi 3 predstavljena je implementacija servera, koja je zasnovana na mikroservisnoj arhitekturi, dok je svaki od mikroservisa pratio arhitekturu slojeva luka (engl. *Onion*) [5, 6], koja je pomogla da se lako dodaje podrška za sve protokole komunikacije.

[1] `https://github.com/VukanAntic/ArtOfTheDay`

- Izrada klijenta zasnovanog na MVC arhitekturi [7] - U glavi 4 prikazano je kako je implementiran klijentski deo koda koji prati MVC arhitekturu i dodatne funkcionalnosti koje klijent podržava kao što je vidžet (engl. *widget*).
- Dodavanje gRPC i GraphQL - Glava 5 predstavlja prikaz svih gore navedenih protokola komunikacije, kao i kvalitativna poređenja.

Glava 2

Prikupljanje podataka

Aplikacija razvijena u okviru ovog rada, *Art of the day*, prikazuje korisniku novu umetničku sliku svakog dana uzimajući u obzir preference korisnika. Potrebe projekta su bile: velika količina umetničkih slika, detaljan opis za svaku, žanr kom slika pripada, kao i informacija ko je bio autor iste. Iz tih razloga, za izvor podataka bio je izabran Institut za umetnost u Čikagu (engl. *The Art Institute of Chicago*) [8]. Institut nudi javni programski interfejs, ali i potpuni prepis zbirke (engl. *data dump*) - arhiva svih slika, sačuvanih u JSON obliku. Odabran je pristup preko arhive, iz razloga što bi dohvatanje podataka preko interfejsa zahtevalo veliki vremenski period, kao i stavljanje velikog pritiska na server instituta.

Slike se ne čuvaju lokalno, jer ih institut izlaže preko protokola IIIF (engl. *International Image Interoperability Framework*) koji omogućava da se željena slika, kao i njena širina zatraži preko adrese (*url*). Adresa slike se može sastaviti iz identifikatora slike (engl. *image id*), što pokazuje Primer koda 2.1.

```
1 path = f'https://www.artic.edu/iiif/2/{data["image_id"]}
2         /full/843,/0/default.jpg'
```

Primer koda 2.1: Sastavljanje adrese slike iz identifikatora

Na ovaj način, za dohvatanje i prikazivanje slika bilo je samo potrebno izvući identifikator svake željene slike, i dobija se adresa iste. Vrednost 843 podrazumeva najčešće korišćenu veličinu slike, koja je verovatno keširana na strani institutovog servera. Bilo je moguće zameniti tu vrednost sa manjom vrednošću, da sve slike imaju istu širinu i visinu.

2.1 Obrada podataka

Automatska obrada podataka bila je realizovana uz pomoć skripte napisane u programskom jeziku *Python* po imenu *ImageDatabaseGenerator.py*. Slika 2.1 prikazuje kako je to urađeno.

Slika 2.1: Tok obrade podataka

2.1.1 Filtriranje

Arhiva instituta ne sadrži samo umetničke slike, već i skulpture, fotografije, tekstil kao i veliku količinu drugih tipova objekata. Iz tog razloga, bilo je potrebno filtrirati podatke, tako da se pronađu samo umetničke slike, što se može videti u Primer koda 2.2.

```
def parse_single_artwork(data):
    id = data["id"]
    title = data["title"]
    description = data["description"]
    date_display = data["date_display"]
    style_id = data["style_id"]
    style_title = data["style_title"]
    artist_id = data["artist_id"]
    artist_title = data["artist_title"]
    artwork_type_id = data["artwork_type_id"]
    # id = 1 represents artworks
    artworked_wanted_indicies = [1]
    path = f'https://www.artic.edu/iiif/2/{data["image_id"]}/full
        /843,/0/default.jpg'
    if title is None or description is None or date_display is None
        \
    or style_title is None or artwork_type_id not in
        artworked_wanted_indicies or \
    check_if_path_is_invalid(path) :
        return None
    return {
            "id" : id,
            "title" : title,
            "description" : description,
            "date_display" : date_display,
            "path" : path,
            "style_title" : style_title,
            "style_id" : style_id,
            "artist_title" : artist_title,
            "artist_id" : artist_id
            }
```

Primer koda 2.2: Filtriranje umetničkih slika

Pored provere da li je u pitanju umetnička slika, bilo je potrebno dodati i druge

provere da li postoji naziv slike, opis, i datum njenog nastanka.

2.1.2 Provera ispravnosti slike

Naredni korak posle filtriranja podrazumeva proveru da li slika koja se nalazi u arhivi instituta ispravna, tj. da li pristup adresi te slike vraća grešku, što pokazuje primer koda 2.3.

```
def check_if_path_is_invalid(path) :
    response = requests.get(path)
    print(path)
    print(response.status_code)

    if response.status_code == 429:
        print("Too many requests in a short period of time, need to
            cool off...")
        time.sleep(60)
        response = requests.get(path)
    if response.status_code == 200:
        return False

    print("Found an error code!")
    return True
```

Primer koda 2.3: Provera dostupnosti

Bitno je napomenuti da server od instituta preko kojeg se dohvataju same slike vraća grešku **429**, ukoliko je previše zahteva stiglo sa iste *IP adrese* u kratkom vremenskom periodu. Zbog toga se u toku izvršavanja skripte program zaustavi na 60 sekundi, da bi moglo dalje da se pristupi serveru instituta.

2.1.3 Relacioni model

Izdvojeni podaci upisuju se u bazu PostgreSQL (engl. *PostgreSQL*) preko biblioteke SQLAlchemy (engl. *SQLAlchemy*). Potrebni podaci se čuvaju u četiri različite tabele - Umetnička slika (engl. *Artwork*), Žanr (engl. *Genre*), Umetnik (engl. *Artist*), kao i asocijativnu tabelu između slika i žanrova. To se može videti primerom koda 2.4, koji takav model generiše:

```
artwork_genre_association = Table(
    'artwork_genre', Base.metadata,
    Column('artwork_id', Integer, ForeignKey('artworks.id')),
    Column('genre_id', String(20), ForeignKey('genres.id'))
)

class Artwork(Base):
    __tablename__ = 'artworks'
    id = Column(Integer, primary_key=True, autoincrement=False)
    title = Column(String(255), nullable=False)
    date_display = Column(String(255), nullable=False)
    description = Column(Text, nullable=False)
    path_to_artwork = Column(String(255), nullable=False)
    artist_id = Column(Integer, ForeignKey('artists.id'), nullable=
        False)

    # Relationships
    genres = relationship('Genre', secondary=
        artwork_genre_association, back_populates='artworks')
    artist = relationship('Artist', back_populates='artworks')

class Artist(Base):
    __tablename__ = 'artists'

    id = Column(Integer, primary_key=True, autoincrement=True)
    # Unknown artist!
    name = Column(String(255), nullable=True)

    artworks = relationship('Artwork', back_populates='artist')

class Genre(Base):
    __tablename__ = 'genres'

    id = Column(String(20), primary_key=True)
    name = Column(String(255), nullable=False)

    # Define the relationship to Artwork
    artworks = relationship('Artwork', secondary=
        artwork_genre_association, back_populates='genres')
```

Primer koda 2.4: Relacioni model

Tri bitne stavke o samom modelu koje je značajno izdvojiti.

- **Identifikatori se preuzimaju od izvora** - Delo i autor zadržavaju celobrojni identifikator koji im je dodelio institut, umesto da dobiju novi. Time je omogućeno da se u budućnosti zbirka dopuni novim slikama i umetnicima, bez potrebe za brigom o duplikatima. Ista odluka primenjena je i na žanrove, identifikator je zadržan od instituta, koji je niska.

```
artwork_data = Artwork(**{
                "id" : artwork["id"],
                "title": artwork["title"],
```

Primer koda 2.5: Dodela identifikatora preuzetog od instituta

- **Veza slika-žanr je više-prema-više** - Zbog prirode skupa podataka, jedna slika može pripadati više različitih žanrova u isto vreme, isto tako, jednom žanru može odgovarati veći broj slika. Upravo zbog te veze, potrebno je kreirati asocijativnu tabelu.
- **Veza slika-umetnik je jedan-prema-više** - Isto kao i za prethodni odnos, zbog prirode podataka, jednu sliku je mogao da naslika jedan umetnik dok je jedan umetnik mogao da naslika više različitih slika.

2.1.4 Upisivanje u bazu

Nakon što se uspešno kreira model, prelazi se na naredni korak. Upisivanje podataka koji su uzeti iz arhive u novo kreiranu bazu. U dole navedenom kodu 2.6 možemo videti da, nakon što su entiteti kreirani i dodati u liste, u toku jedne izmene (engl. *commit*), podaci su dodati u bazu podataka.

```
        if artist_id not in artist_ids:
            artists_data.append(artist_data)
        if artwork["style_id"] not in genre_ids:
            genres_data.append(genre_data)

        new_artist_data = list(filter(lambda elem : elem.id ==
            artist_id, artists_data))
        new_genre_data = list(filter(lambda elem : elem.id ==
            artwork["style_id"], genres_data))
        print(new_artist_data)
        artwork_data.artist = new_artist_data[0]
        artwork_data.genres.append(new_genre_data[0])
```

```
11
12          artwork_ids.add(artwork["id"])
13          artist_ids.add(artist_id)
14          genre_ids.add(artwork["style_id"])
15
16
17      session.add_all(artists_data)
18      session.add_all(genres_data)
19      session.add_all(artworks_data)
20      session.commit()
```

Primer koda 2.6: Isečak koda koji upisuje podatke u bazu podataka

Glava 3

Serverska strana

Serverska strana realizovana je u tehnologiji Spring Boot (engl. *Spring Boot*) kao skup od četiri međusobno nezavisna mikroservisa. Svaki od mikroservisa u sebi je implementiran preko arhitekture slojeva luka (engl. *Onion architecture*). Mikroservisi su međusobno povezani preko brokera poruka (engl. *message broker*) po imenu RabbitMQ(engl. *RabbitMQ*). U narednoj glavi biće prikazano kako su organizovani svi mikroservisi, kako izgleda svaki pojedinačan mikroservis, kako komuniciraju i kako arhitektura olakšava dodavanje više protokola komunikacije da budu implementirani unutar istog mikroservisa, sa minimalnom količinom dodatnog koda.

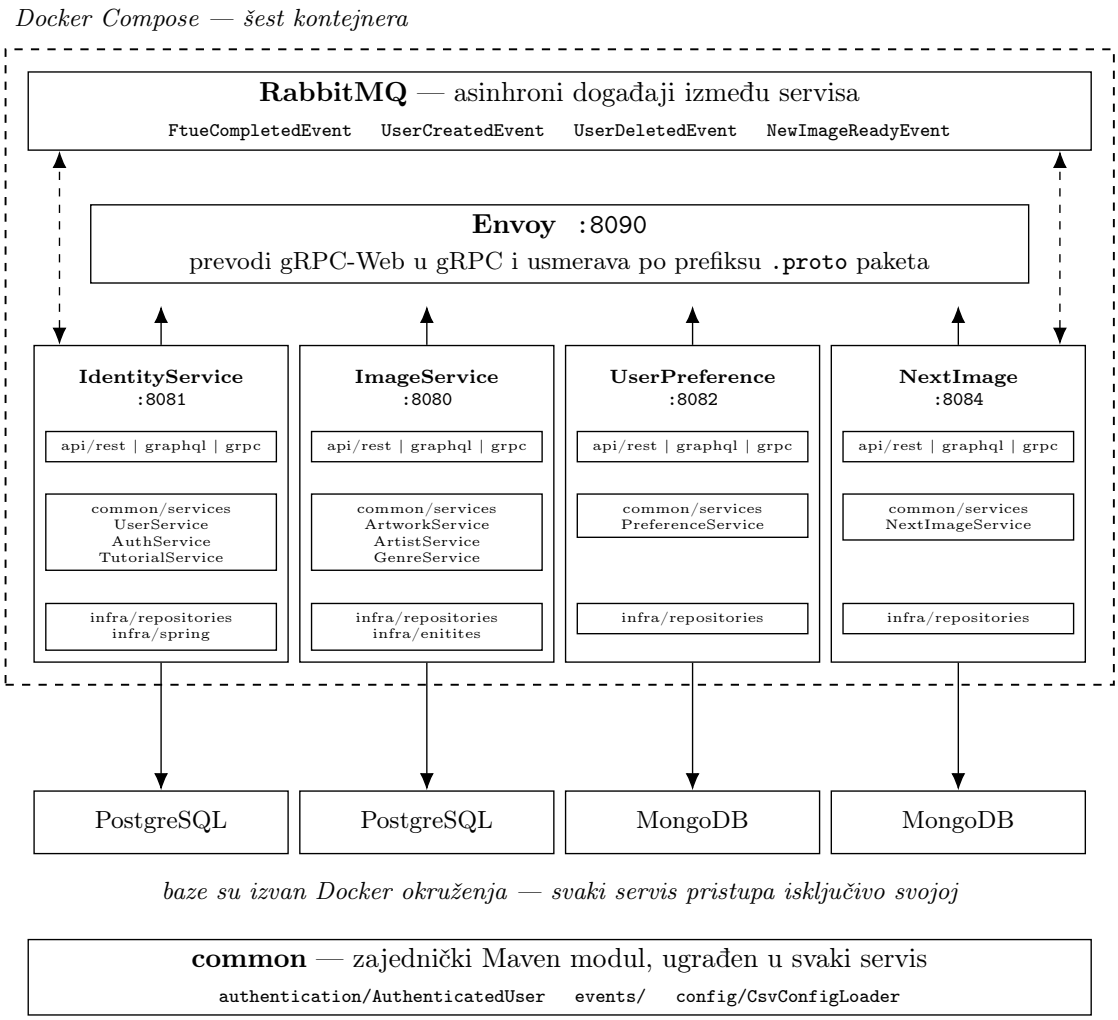
3.1 Spring Boot

Čitav server razvijen je u programskom jeziku Java, uz radni okvir (engl. *framework*) *Spring Boot* [9]. Spring Boot je nadogradnja nad radnim okvirom *Spring*, uz motivaciju da ubrza razvoj, smanji konfiguraciju i ubrza isporuku (engl. *deployment*). Pored toga, Spring Boot podržava svaki od protokola komunikacije koje ovaj rad opisuje.

3.2 Podela mikroservisa

Mikroservisi su međusobno nezavisne celine koje su najčešće podeljene po nekoj poslovnoj logici. Svaki od mikroservisa unutar sebe prati neku dogovorenu strukturu, ima sopstvenu bazu podataka, i tako omogućava da svaki od mikroservisa može da se pusti u produkciju sa njegovim najnovijim izmenama, bez pravljenja

novog izdanja za ostale mikroservise. Iz tih razloga, odlučeno je da serverska strana bude podeljena na mikroservise. Organizaciju čitavog servera možemo videti na slici 3.1.



Slika 3.1: Arhitektura serverske strane

Kao što se vidi, serverska strana je podeljena na četiri mikroservisa:

- **Mikroservis za slike (engl. *ImageService*)** - Mikroservis odgovoran za sav sadržaj vezan za slike. On koristi podatke koji su iz Glave 2, i daje programski interfejs vezan za slike, umetnike i žanrove. Podaci se čuvaju u relacionoj bazi podataka *PostgreSQL*, jer su svi podaci međusobno povezani.
- **Mikroservis za autentifikaciju (engl. *IdentityService*)** - Mikroservis koji se bavi čitavim procesom autentifikacije korisnika. Unutar sebe sadrži

programski interfejs za prijavu, odjavu, registraciju, kao i čuvanje vitalnih informacija o korisniku, kao što su korisničko ime, lozinka i adresa elektronske pošte. Pri uspešnoj prijavi, servis izdaje par žetona: *pristupni žeton* kratkog trajanja (kojim se potpisuje svaki naredni zahtev) i *žeton za osvežavanje* dužeg trajanja (kojim se pribavlja nov pristupni žeton bez ponovnog unosa lozinke). Ovaj mikroservis takođe koristi relacionu bazu podataka. Podaci o korisniku imaju unapred definisanu strukturu, a nad korisničkim imenom i adresom postoji ograničenje da moraju biti unikatni.

- **Mikroservis za preference (engl. *UserPreferenceService*)** - Mikroservis koji je odgovoran za čuvanje svih preferenci korisnika. Čuva informacije o omiljenim slikama, žanrovima i umetnicima korisnika. Za razliku od prethodna dva mikroservisa, on koristi nerelacionu bazu podataka *MongoDB*. Pošto se sve preference korisnika čuvaju na nivou listi, kao što su sve omiljene slike korisnika, odabran je ovaj oblik baze podataka.

- **Mikroservis za narednu sliku (engl. *NextImageService*)** - Mikroservis koji generiše novu sliku za korisnika svaki dan. Uz pomoć informacija o korisniku iz mikroservisa za preference, on algoritmom odlučuje koju će novu sliku dodeliti korisniku taj dan. On čuva informaciju koje je preferentno vreme za davanje nove slike korisniku, i preko trajne veze sa korisnikom (engl. *Websocket*), šalje najnoviju sliku. Pošto i on čuva listu slika, kao i vreme kada korisnik želi novu sliku, odabrana je nerelaciona baza podataka *MongoDB*.

Svaki servis koristi alat Maven (engl. *Maven*) koji služi za izgradnju, nad razvojnim kompletom Java 21.

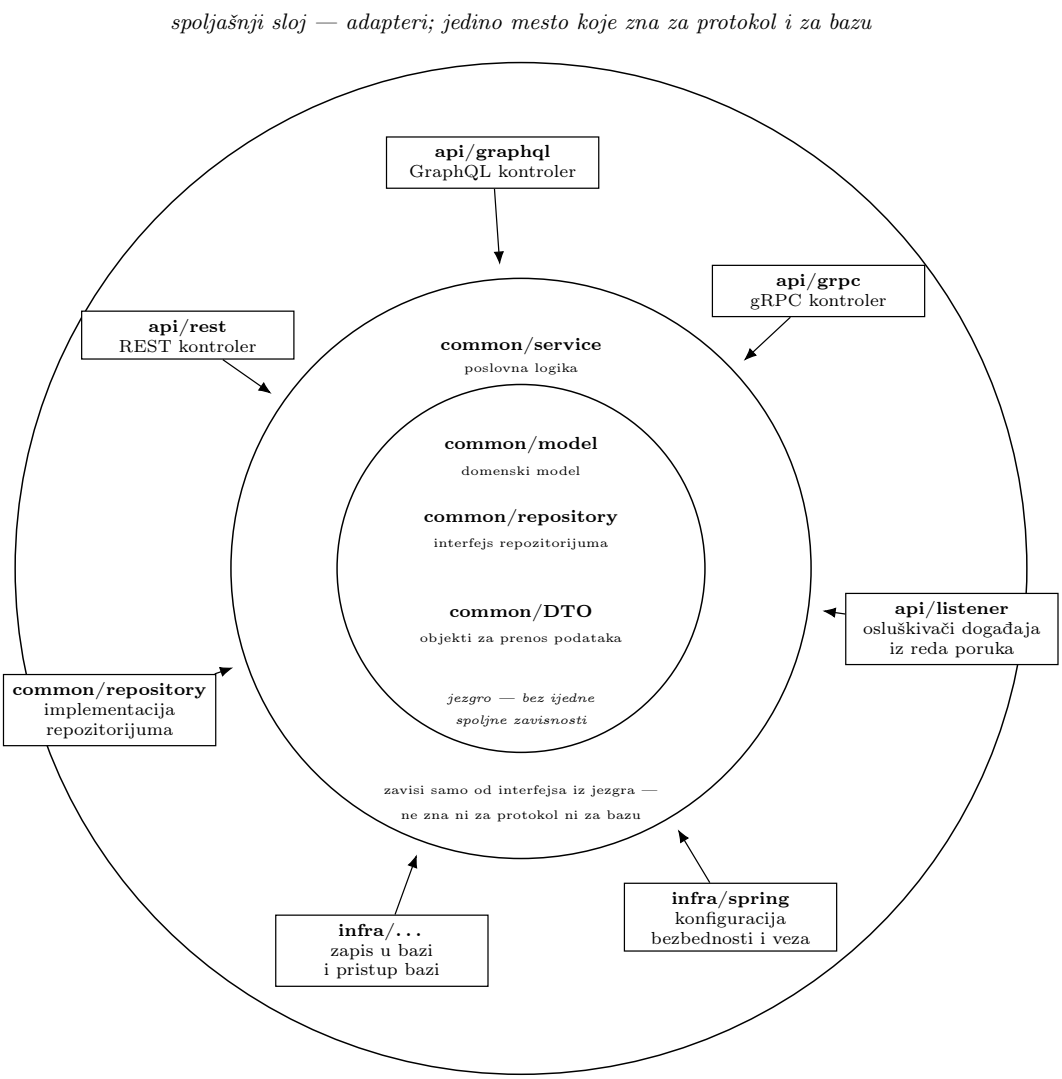
3.3 Arhitektura slojeva luka unutar servisa

Kod unutar svakog od mikroservisa je organizovan prema arhitekturi slojeva luka (engl. *Onion architecture*). Osnovni koncept te arhitekture je da se **zavisnosti pokazuju samo ka unutra**. Svaki spoljašnji sloj zna, i može da koristi unutrašnje slojeve, dok unutrašnji slojevi ne mogu pristupiti spoljašnjim slojevima.

Na slici 3.2 se može videti kako izgleda arhitektura svakog mikroservisa. Svaki od slojeva gleda ka unutra, i upravo iz tih razloga je dodavanje više protokola komunikacije bilo olakšano. Zbog nje, svaki novi dodati protokol komunikacije

će koristiti unutrašnje slojeve, i tako će REST, gRPC, i GraphQL iako imaju različite načine za predstavljanje programskog interfejsa, imati identičnu logiku koja se odigrava.

U jezgru se nalaze domenski modeli, objekti za prenos podataka i interfejsi repozitorijuma - dakle sve ono što opisuje *šta* aplikacija radi, a ne kako to izvodi. Oko njega stoji sloj poslovne logike, dok se u spoljašnjem sloju nalaze adapteri: kontroleri za svaki od protokola, pristup bazi i konfiguracija.



Slika 3.2: Organizacija paketa unutar mikroservisa

3.3.1 Jezgro: entiteti i objekti za prenos

Unutar jezgra se nalaze dve različite vrste klasa. **Entiteti** predstavljaju objekte u bazi, i imaju sve informacije iz njih, dok **objekti za prenos podataka** su objekti čija je svrha komunikacija sa spoljašnjim svetom. Objekti za prenos podataka najčešće imaju manju količinu podataka, jer često nema potrebe slati sve informacije iz entiteta direktno klijentu. U kodovskom primeru 3.1 prikazan je entitet za sliku (engl. *Artwork*).

```
@Entity
@Builder
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Artwork {

    @Id
    private Long id;
    private String title;
    @Column(length = 10000)
    private String description;
    @Column(name = "path_to_artwork")
    private String imageUrl;

    @ManyToOne
    @JoinColumn(name = "artist_id")
    @Setter
    private Artist artist;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
            name = "artwork_genre",
            joinColumns = @JoinColumn(name = "artwork_id"),
            inverseJoinColumns = @JoinColumn(name = "genre_id")
    )
    private Set<Genre> genres;
}
```

Primer koda 3.1: Primer entiteta za sliku

Za pristup relacionoj bazi korišćen je *Hibernate* [10, 11], alat za objektno-

relaciono preslikavanje (engl. *Object-Relational Mapping*, ORM). Njegova odgovornost je da premosti razliku između dva načina predstavljanja podataka: objektnog, u kome su podaci povezani referencama, i relacionog, u kome su razbijeni po tabelama i povezani stranim ključevima.

Kao posledica korišćenja *Spring Boot* radnog okvira kao i *Hibernate* alata za objektno-relaciono preslikavanje, korišćene su anotacije koje olakšavaju korišćenje entiteta. Anotacija `@Entity` označava da se klasa preslikava u tabelu relacione baze, čime je Hibernate prepoznaje kao entitet i preuzima brigu o njenom čuvanju i učitavanju. Anotacija `@Id` označava polje koje predstavlja primarni ključ te tabele, po kome se svaki red jednoznačno razlikuje od ostalih.

3.3.2 Srednji sloj: poslovna logika

Srednji sloj sadrži servise u kojima se nalazi sva poslovna logika aplikacije. On komunicira sa jezgrom, i korišćenjem entiteta iz jezgra kreira objekte za prenos, koji kasniji slojevi mogu da koriste. Na primeru 3.2 može se videti logika servisa. Njegova poslovna logika zahteva da dohvata entitete, i transformiše ih u objekte za prenos. Zbog prirode projekta, ovaj servis ne zahteva kompleksniju logiku, dok na primer, servis koji se bavi izborom sledeće slike, čiji je isečak dat u 3.3, mora da radi kompleksnije algoritme.

```
1  @Service
2  @AllArgsConstructor
3  public class ArtworkService {
4
5      private final ArtworkRepository artworkRepository;
6      private final ModelMapper modelMapper;
7
8      public List<ArtworkDTO> getAllArtworks() {
9          return artworkRepository.findAll()
10                 .stream().map(artwork -> modelMapper.map(artwork,
                       ArtworkDTO.class))
11                 .collect(Collectors.toList());
12     }
13
14     public List<IdentityArtworkDTO> getAllArtworks(String genreId) {
15         return artworkRepository.findByGenres_Id(genreId)
16                 .stream().map(artwork -> modelMapper.map(artwork,
                       IdentityArtworkDTO.class))
```

```
17                  .collect(Collectors.toList());
18      }
19
20
21      public List<ArtworkDTO> getAllArtworksFromIds(List<Long>
            artworkIds) {
22          return artworkRepository.findAllById(artworkIds)
23                  .stream()
24                  .map(artwork -> modelMapper.map(artwork, ArtworkDTO.
                        class))
25                  .collect(Collectors.toList());
26      }
27
28      public Optional<Long> getRandomArtworkIdExcluding(Set<Long>
            excludeIds) {
29          if (excludeIds == null || excludeIds.isEmpty()) {
30              return artworkRepository.findRandomId();
31          }
32          return artworkRepository.findRandomIdExcluding(excludeIds);
33      }
34
35      public List<ArtworkDTO> getRandomArtworks(int count) {
36          return getAllArtworksFromIds(artworkRepository.findRandomIds
                (count));
37      }
38  }
```

Primer koda 3.2: Primer servisa za slike sa potrebnom poslovnom logikom

```
1  public long selectNextArtworkId(String username, Set<Long>
       seenArtworkIds) {
2          var preferences = userPreferenceServiceClient.
               getUserPreferences(username);
3          boolean hasGenres = !preferences.getLikedGenreIds().isEmpty
               ();
4          boolean hasArtists = !preferences.getLikedArtistIds().
               isEmpty();
5
6          if ((!hasGenres && !hasArtists) || random.nextDouble() <
               probabilityConfig.getRandomImageProbability()) {
7              return getRandomUnseen(seenArtworkIds, preferences);
8          }
```

```
9
10          boolean tryGenreFirst = hasGenres && (!hasArtists || random.
                nextDouble() < probabilityConfig.
                getGenreVsArtistProbability());
11
12          Optional<Long> result = tryGenreFirst
13                  ? tryFromGenre(preferences, seenArtworkIds)
14                  : tryFromArtist(preferences, seenArtworkIds);
15
16          return result.orElseGet(() -> getRandomUnseen(seenArtworkIds
                , preferences));
17      }
```

Primer koda 3.3: Isečak logike izbora nove slike

Srednji sloj nema kontekst oko protokola komunikacije koji se koristi, i upravo iz tog razloga, moguće je pozivati servis, i dohvatanje istih podataka, bez obzira na izbor protokola.

3.3.3 Spoljašnji sloj: adapteri protokola

Spoljašnji sloj obuhvata klase koje realizuju direktnu komunikaciju sa srednjim slojem. One komuniciraju sa servisom, dohvataju podatke na koje je servis već primenio poslovnu logiku, i samo prosleđuju klijentu. U aplikaciji, primer takve klase je kontroler (engl. *controller*). Na primeru 3.4 može se videti kontroler koji je zadužen za REST.

```
1  public class ImageController {
2
3
4      private final ArtworkService artworkService;
5      private final ArtistService artistService;
6      private final GenreService genreService;
7
8
9      @GetMapping("/get-all-artworks")
10     public List<ArtworkDTO> getAllArtworks() {
11         return artworkService.getAllArtworks();
12     }
13
14     @GetMapping("/get-artworks-from-genre")
```

```
15      public List<IdentityArtworkDTO> getAllArtworksFromGenre(
            @RequestParam String genreId) {
16          return artworkService.getAllArtworks(genreId);
17      }
18  }
```

Primer koda 3.4: Isečak logike kontrolera za slike koji podržava REST

Kontroler definiše REST putanju koju podržava, i kada klijent pokuša da pristupi njoj, izvršava se određeni metod. Ovo je primer jednog protokola komunikacije koji je podržan, ali u Glavi 5, biće prikazana podrška i za ostale.

3.4 Asinhrona komunikacija

Do sada sva komunikacija koja je pomenuta bila je *sinhrone prirode*, tj. pozivalac šalje zahtev i čeka odgovor. Tek kada pozivaocu stigne odgovor on nastavi dalje sa izvršavanjem, ali, ako je potrebno obezbediti komunikaciju između više mikroservisa, a da jedan mikroservis ne zna na koga sve utiče, neophodna je *asinhrona komunikacija*. Pošiljalac ne šalje poruku primaocu, već je šalje posredniku, i odmah nastavlja dalje sa svojim poslom. Posrednik čuva poruku, sve dok primalac nije spreman da je preuzme i da reaguje na tu poruku. Pošiljalac nema informaciju o tome ko sluša i reaguje na poruke, pa je ovakav način komunikacije prikladan za obaveštenje o događajima na koje verovatno bi neki drugi mikroservis reagovao.

Kao broker poruka (engl. *broker software*) koristi se *RabbitMQ* [12], red poruka koji implementira standard **AMQP** (engl. *Advanced Message Queuing Protocol*) [13], protokol za razmenu poruka između mikroservisa. Pošto standard definiše tačan oblik bajtova koji putuju mrežom, to omogućava da mikroservisi koji komuniciraju na ovaj način ne moraju da budu implementirani na istom programskom jeziku, i posrednik može da se zameni, bez uticaja na kod.

Primer 3.5 pokazuje isečke iz koda gde se šalje poruka, i gde se reaguje na istu. Mikroservis koji je zadužen za autentifikaciju u momentu kada se korisnik registruje šalje događaj (engl. *event*) po imenu *UserCreatedEvent*. Mikroservis zadužen za preference reaguje na taj događaj, i kreira entitet za tog korisnika.

```
1  public class UserCreatedEvent {
2      private String username;
3      private String timeZoneId;
4  }
```

```
5
6  // IdentityService -- objavljuje i nastavlja dalje
7  rabbitTemplate.convertAndSend(userEventsExchange,
       userCreatedRoutingKey, userCreatedEvent);
8
9  // UserPreferenceService -- reaguje kada mu odgovara
10  @RabbitListener(queues = "${spring.rabbitmq.user_created_queue}" )
11   public void receiveUserCreatedEvent(UserCreatedEvent
         userCreatedEvent) {
12       userPreferenceRepository.persist(userCreatedEvent.getUsername
             ());
13   }
```

Primer koda 3.5: Objava i prijem događaja o završenom registrovanju korisnika

Bitno je napomenuti da na isti događaj može više slušalaca da reaguje. Kao što mikroservis za preference reaguje na poruku da je registrovan korisnik, mikroservis za izbor sledeće slike isto sluša tu poruku, i reaguje da kreira entitet za tog korisnika.

Glava 4

Klijentska aplikacija

Klijent je realizovan u radnom okviru React Native (engl. *React Native*) [14] nad bibliotekom React, zajedno sa radnim okvirom Expo (engl. *Expo*) [15]. Svaka od komponenti klijenta je implementirana po model-prikaz-kontroler (engl. *Model-View-Controller*) arhitekturi [7]. U narednoj glavi biće prikazano kako su izgrađene komponente kao i obrazac dizajna (engl. *design pattern*) kad je u pitanju komunikacija sa serverom preko različitih protokola komunikacije.

4.1 React Native

Čitav klijent razvijen je u programskom jeziku TypeScript [16], uz radni okvir (engl. *framework*) *React Native* [14]. React Native je nadogradnja nad bibliotekom *React* [17], uz motivaciju da se iz istog koda isporuče i Android i iOS aplikacije, pri čemu se interfejs ne iscrtava u pregledaču, nego koristi izvorne komponente tih platformi. Pored toga, uz njega je korišćen i *Expo* [15], koji izmenu u kodu čini vidljivom u aplikaciji za nekoliko sekundi.

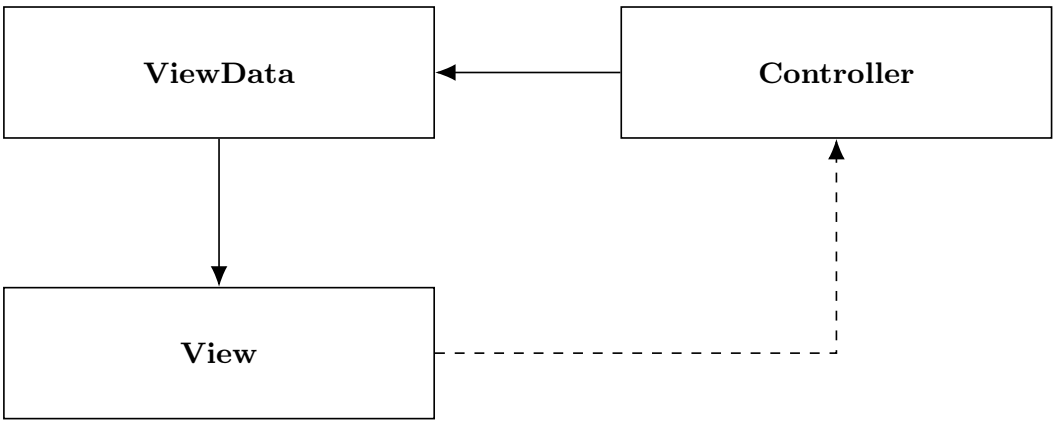
4.2 Model-Prikaz-Kontroler arhitektura

Model-prikaz-kontroler arhitektura (engl. *Model-View-Controller*) [7, 18] predstavlja ustaljen obrazac za organizaciju komponenti u klijentskim aplikacijama. Osnovne komponente arhitekture su:

- **Prikaz (engl. *View*)**
- **Model (engl. *Model*)**

- **Kontroler (engl. *Controller*)**

Na slici 4.1 vidi se upravo opisana arhitektura.



Slika 4.1: Dijagram arhitekture MVC

4.2.1 Prikaz

Prikaz je zadužen za sve komponente koje se iscrtavaju na ekranu telefona. On poseduje svoje lične podatke (u projektu pod nazivom *ViewData*) koje mu se dodeljuju, i uz pomoć istih, stara se oko prikaza konkretnog dela aplikacije. U trenutku dodele novog modela od strane kontrolera prikaza, čitav prikaz se opet iscrtava. Želja ovim pristupom jeste da sve što je prikazano na ekranu uvek bude ispravno stanje koje je server poslao.

4.2.2 Model

U aplikaciji prikazni modeli (*ViewData*) predstavljaju podatke koje kontroler dodeljuje svom prikazu. Oni se razlikuju od entiteta koji su poslati od strane servera. Ideja je da model ima samo informacije koje su potrebne prikazu da vrši svoju dužnost. Model nema informaciju o samom prikazu kao ni kontroleru. On je samo klasa koja predstavlja prenos podataka.

4.2.3 Kontroler

Kontrolerova funkcija je da se bavi poslovnom logikom vezanom za prikaz. On sadrži referencu na svoj prikaz, i sa njom, dodeljuje model koji mu treba

za iscrtavanje ekrana. Kontroler je taj koji ima reference na repozitorijume, o kojima ćemo više videti u glavi 4.4, iz kojih čita entitete poslate od servera. U kodovskom primeru 4.1 možemo videti isečak kontrolera za početni ekran. On dohvata informacije o slikama koje se pokazuju korisniku na glavnom ekranu, vrši obradu tih podataka, i dodeljuje je svom prikazu. Metod *setViewData* je taj koji dodeljuje model prikazu.

```
1  private async load(): Promise<void> {
2          if (this.loading) return;
3          this.loading = true;
4          try {
5              const artworks = await this.buildArtworks();
6              this.setViewData(new HomeScreenViewData(artworks, true))
                   ;
7          } catch (e) {
8              console.error('[HomeScreen] loadArtworks failed:', e);
9              this.setViewData(new HomeScreenViewData(this.getSnapshot
                   ().artworks, true));
10         } finally {
11             this.loading = false;
12         }
13     }
14
15     private async buildArtworks(): Promise<FeaturedArtworkViewData
           []> {
16         const history = await this.historyRepository.get() ?? [];
17         if (history.length === 0) return [];
18
19         const allArtworks = await this.artworkRepository.get();
20         const preferences = await this.preferencesRepository.get();
21         const likedIds = new Set(preferences?.likedArtworkIds ?? [])
               ;
22
23         return [...history]
24             .sort((a, b) => a.seenAt.getTime() - b.seenAt.getTime())
25             .map(seenImage => {
26                 const artwork = allArtworks?.getById(seenImage.
                       artworkId);
27                 return artwork ? new FeaturedArtworkViewData(artwork
                       , seenImage, likedIds.has(artwork.id)) : null;
28             })
```

```
29              .filter((item): item is FeaturedArtworkViewData => item
                    !== null);
30      }
```

Primer koda 4.1: Isečak koda iz kontrolera koji se bavi postavljanjem početnog ekrana

4.3 Namera

Prikaz nema informaciju o postojanju kontrolera, jedini način da prikaz komunicira sa spoljašnjim svetom jeste preko namera (engl. *Intent*). Namera podrazumeva neku akciju koja se dogodila na nivou prikaza, i treba ta informacija da se propagira dalje. U primeru 4.2 možemo videti ceo životni ciklus namere. U tom primeru, klasa *ArtworkPreferenceIntent* predstavlja nameru da se korisniku dopada ili ne dopada slika. Ta namera se šalje od prikaza, i stiže do kontrolera, koji dalje šalje zahtev serveru.

```
1  export type ArtworkPreferenceIntent =
2      | {type: 'LIKE'; artworkId: number}
3      | {type: 'UNLIKE'; artworkId: number}
4      | {type: 'DISLIKE'; artworkId: number}
5      | {type: 'UNDISLIKE'; artworkId: number};
6
7  // prikaz salje nameru
8  const onPreferenceIntent = useCallback(
9      (intent: ArtworkPreferenceIntent) => send(intent),
10     [send],
11 );
12
13 // reakcija kontrolera na nameru
14 onMessage(intent: ArtworkPreferenceIntent): void {
15         this.handlePreference(intent)
16             .catch(e => console.error('[HomeScreen] preference
                   intent failed:', e));
17 }
```

Primer koda 4.2: Namera da korisnik promeni preferencu prema slici

4.4 Repozitorijum

Repozitorijum [18] u klijentskom kontekstu podrazumeva čuvanje trenutnog stanja aplikacije u memoriji. Unutar repozitorijuma čuvaju se entiteti (engl. *Domain Data*) koji su kreirani iz informacija poslatih sa servera. Kod 4.3 prikazuje kreiranje entiteta za slike, koji će, pri proizvoljnom odgovoru od servera, biti sačuvan upravo u repozitorijum.

```
export class ArtworkData {
    constructor(
        readonly id: number,
        readonly title: string,
        readonly description: string,
        readonly imageUrl: string,
        readonly artist: ArtistData,
        readonly genres: GenreData[],
    ) {}
}

// kreiranje entiteta iz gRPC odgovora
function toArtwork(a: Artwork): ArtworkData {
    return new ArtworkData(
        Number(a.id),
        a.title,
        a.description,
        a.imageUrl,
        new ArtistData(Number(a.artist?.id ?? 0), a.artist?.name ??
            ''),
        a.genres.map(g => new GenreData(g.id, g.name)),
    );
}
```

Primer koda 4.3: Entitet za sliku kao i njegovo kreiranje iz GRPC odgovora

Osnova repozitorijuma predstavlja interfejs (koji u sebi ima 3 jednostavne namene): čuvanje entiteta, upisivanje novog sadržaja u sebe, kao i da neko može da se pretplati na promenu njenog sadržaja. Upravo taj interfejs se vidi u primeru 4.4.

```
export interface IRepository<T> {
    get(): Promise<T | null>;
```

```
3     update(data: T): Promise<void>;
4     subscribe(listener: () => void): () => void;
5 }
```

Primer koda 4.4: Interfejs repozitorijuma

Sve tri metode su asinhrone prirode, iako za jednostavno čuvanje u memoriji to nije potrebno. Razlog je jer u određenim situacijama, potrebno je takođe čuvanje informacija između različitih korisničkih sesija, a čuvanje u skladište uređaja je asinhrona akcija. Kada se interfejs koristi u kodu, ne zna se koji je tačno tip repozitorijuma, što olakšava pisanje novog koda.

Postoje tri različita tipa repozitorijuma:

- **U memoriji repozitorijum (engl. *InMemoryRepository*)** - Repozitorijum koji čuva podatke u memoriji i koristi se za sve što važi samo tokom jedne sesije, poput liste slika ili istorije viđenih slika.
- **Sigurni repozitorijum (engl. *SecureRepository*)** - On čuva žetone u zaštićenom skladištu uređaja, gde ostaju i u novoj sesiji korisnika.
- **Keširani repozitorijum (engl. *CachedRepository*)** - Repozitorijum koji ne čuva ništa sam, već obavija drugi repozitorijum i pamti pročitanu vrednost u memoriji.

4.5 Rukovalac komandama

Između klijenta i servera postoji sloj koji ih spaja, takozvani rukovalac komandama (engl. *Command Handlers*) [19]. Svaki od njih obavlja tačno jednu dužnost, da dohvati podatke sa servera i da ih upiše u repozitorijum, gde ih kasnije kontroler čita. Kontroler ne poziva server direktno, već koristi rukovaoca da izvrši komandu.

Razlog za taj sloj je što se ista radnja često poziva sa više mesta. Dohvatanje liste slika potrebno je i pri prvom pokretanju, i pri otvaranju korisničkog profila, i posle izmene preferenci. Bez zasebnog rukovaoca, taj isti niz koraka bio bi identičan unutar svakog kontrolera.

```
1 export class GetAllGenresCommandHandler extends CommandHandler {
2     constructor(
3         private readonly client: IImageClient,
```

```
4           private readonly repository: IRepository<GenreData[]>,
5       ) {
6           super();
7       }
8
9       async handle(): Promise<void> {
10          await this.timed('GetAllGenres', async () => {
11              const genres = await this.client.getAllGenres();
12              await this.repository.update(genres);
13          });
14      }
15  }
```

Primer koda 4.5: Rukovalac komandom za dohvatanje žanrova

U primeru 4.5 može se videti kako izgleda primer takvog koda. Njegova dužnost je dohvatanje informacija od servera vezano za sve žanrove koje server podržava. On ne implementira ni jedan konkretan protokol komunikacije, već dobija referencu ka interfejsu, koji definiše sve moguće pozive ka serveru. Jedan od takvih interfejsa vidimo u primeru 4.6.

```
1  export interface IImageClient {
2      getAllArtworks(): Promise<ArtworkData[]>;
3      getArtworksFromIds(command: GetArtworksFromIdsCommand): Promise<
           ArtworkData[]>;
4      getArtworksFromGenre(command: GetArtworksFromGenreCommand):
           Promise<IdentityArtworkData[]>;
5      getArtworksFromArtist(command: GetArtworksFromArtistCommand):
           Promise<IdentityArtworkData[]>;
6      getAllGenres(): Promise<GenreData[]>;
7      getAllArtists(): Promise<ArtistData[]>;
8      getAllGenresFromIds(command: GetAllGenresFromIdsCommand):
           Promise<GenreData[]>;
9      getAllArtistsFromIds(command: GetAllArtistsFromIdsCommand):
           Promise<ArtistData[]>;
10     getRandomArtworkId(command: GetRandomArtworkIdCommand): Promise<
           number>;
11     getRandomArtworks(command: GetRandomArtworksCommand): Promise<
           ArtworkData[]>;
12 }
```

Primer koda 4.6: Interfejs svih poziva ka mikroservisu za slike

Tako, svaki od podržanih protokola komunikacije definiše interfejs po svom načinu komunikacije. Ovo olakšava pozive ka serveru sa strane kontrolera. Ne mora kontroler da bira koji će protokol komunikacije da koristi, već *Command Handler*.

4.6 Vidžet

Za iOS uređaje takođe je implementiran i vidžet (engl. *widget*) [20] koji se može dodati na početnom ekranu uređaja. On pokazuje sliku koju je korisnik dobio tog dana, zajedno sa imenom autora kao i imenom slike u pitanju.

Kad je implementacija vidžeta u pitanju, bitna je napomena da on *nije deo aplikacije*, već na neki način, predstavlja zasebnu aplikaciju, sa kojom razvijena aplikacija u okviru ovog rada može da komunicira. Njegova logika je programirana u programskom jeziku Swift [21], i nema mogućnost da pozove metode napisane u TypeScript jeziku. Komunikacija između glavne aplikacije i vidžeta je preko deljenog prostora (engl. *App Group*).

Aplikacija preuzme sliku koju treba prikazati korisniku tog dana, sačuva je u zajednički prostor, zajedno sa informacijama ko je bio autor, naziv slike, i za koji datum je slika namenjena, i onda vidžet može da prikaže najnoviju sliku korisniku.

```
1  await this.timed('PublishLatestToWidget', async () => {
2      const history = await this.historyRepository.get() ?? [];
3      if (history.length === 0) return;
4
5      const sorted = [...history].sort((a, b) => a.seenAt.getTime() -
           b.seenAt.getTime());
6      const latest = sorted[sorted.length - 1];
7
8      const artwork = (await this.artworkRepository.get())?.getById(
           latest.artworkId);
9      if (!artwork) return;
10
11     await this.publisher.publish({
12         imageUrl: artwork.imageUrl,
13         title: artwork.title,
14         artistName: artwork.artist.name,
15         imageDateISO: latest.seenAt.toISOString(),
16     });
17 });
```

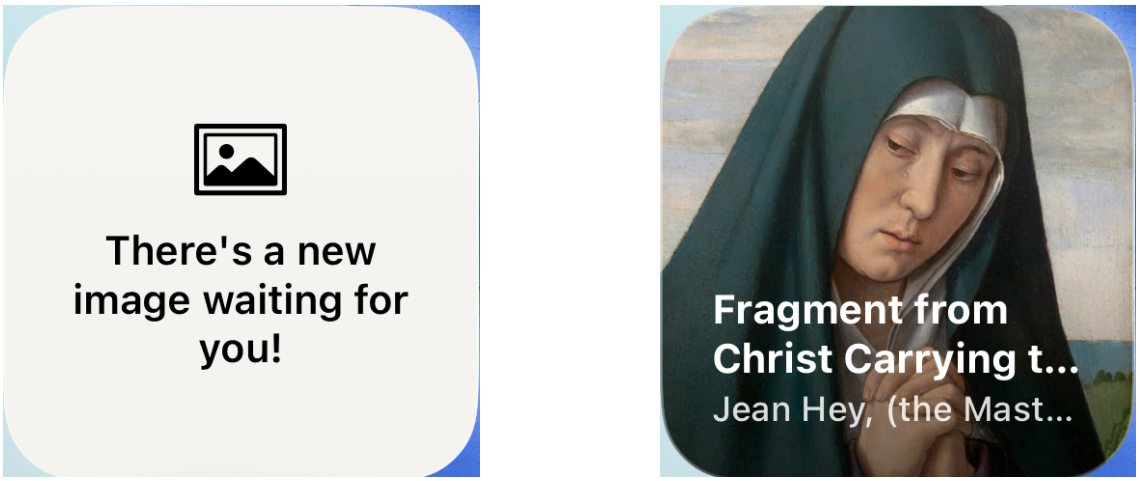
Primer koda 4.7: Slanje informacije o najnovijoj slici vidžetu

Sam vidžet ima dva stanja. Kada je slika namenjena za trenutni dan, ona je prikazana. Kada nije, prelazi na prikaz koji obaveštava korisnika da postoji nova slika spremna za njega. Odluku donosi jedno poređenje datuma:

```
let storedDate = imageDateISO.flatMap(Self.parseISO)
let isToday = storedDate.map { Calendar.current.isDate($0,
    inSameDayAs: date) } ?? false
```

Primer koda 4.8: Provera da li je slika u deljenom prostoru za današnji dan

Provera u kodu 4.8 postoji da bi korisnik bio svestan da postoji nova slika spremna za njega. Ako takva provera ne bi postojala, korisnik bi, sve dok ne uđe u aplikaciju opet, video sliku koja je stara par dana. Ta stanja moguće je videti ispod na slici 4.2.



(a) nema nove slike (b) slika za danas je spremna

Slika 4.2: Dva stanja vidžeta na početnom ekranu

Glava 5

Protokoli komunikacije

U prethodnim glavama, sva komunikacija klijenta i servera bila je izdvojena u zasebne slojeve. U narednoj glavi, biće detaljnije prikazano o svakom protokolu komunikacije, kako su bili implementirani u okviru aplikacije, kao i poređenje njihovih osobina.

5.1 Računanje vremena utrošenog na čekanje

Za tumačenje rezultata merenja korisno je razložiti vreme koje korisnik čeka na sastavne delove. Ono se može izraziti kao

$$T = n \cdot RTT + \sum_{i=1}^{n} \left(t_i^{ser} + t_i^{obr} + \frac{S_i}{B} \right), \tag{5.1}$$

gde je n broj mrežnih kružnih putovanja, RTT vreme kružnog putovanja mreže, t_i^{ser} vreme serijalizacije i deserijalizacije, t_i^{obr} vreme obrade na serveru, S_i veličina pojedinačne poruke, a B raspoloživa propusnost.

Ova relacija pokazuje gde svaki od protokola može doneti poboljšanje. GraphQL deluje na član n, smanjujući broj kružnih putovanja time što više logičkih upita objedinjuje u jedan zahtev. gRPC deluje na članove S_i i t_i^{ser}, smanjujući veličinu poruke binarnim kodiranjem i ubrzavajući njeno pretvaranje u objekte. Nijedan od njih ne utiče na član t_i^{obr}, odnosno na vreme obrade na serveru.

Vredi naglasiti i da odnos ovih članova zavisi od okruženja. Na mobilnim mrežama vrednost RTT tipično iznosi od 30 do 100 milisekundi, pa smanjenje broja kružnih putovanja donosi uštedu merenu stotinama milisekundi. U lokalnom okruženju, gde je RTT zanemarljiv, ista optimizacija gotovo da nema efekta.

5.2 REST

Termin REST (engl. *Representational State Transfer*) uveo je Roj Filding 2000. godine u svom doktorskom radu [1]. U svom radu, on REST naziva *arhitektonski stilom*. On predstavlja skup pravila, koja dovode do toga da sistem bude skalabilan, labavo spregnut i pogodan za keširanje.

5.2.1 Arhitektonska ograničenja

Filding navodi šest pravila (engl. *Constraints*) REST-a:

- **Klijent-Server** (engl. *Client-Server*) - razdvajanje odgovornosti između korisničkog interfejsa i skladištenja podataka.
- **Odsustvo stanja** (engl. *Statelessness*) - svaki zahtev sadrži sve podatke potrebne za njegovu obradu, a server ne čuva stanje sesije između zahteva.
- **Keširanje** (engl. *Cacheability*) - odgovori moraju biti označeni kao keširani ili nekeširani.
- **Jedinstven interfejs** (engl. *Uniform interface*) - resursi se identifikuju jedinstvenom adresom, a nad njima se vrše standardne operacije.
- **Slojevit sistem** (engl. *Layered system*) - klijent ne mora znati da li komunicira sa serverom ili preko posrednika.
- **kod na zahtev** (engl. *Code on demand*) (opciono) - server može klijentu poslati izvršivi kod.

Značajan pojam REST-a jeste **resurs**. On predstavlja svaku celinu koja je od značaja za ceo sistem. U aplikaciji, to podrazumevaju umetničke slike, žanrovi ili autori. Svaki resurs ima svoju adresu, pri čemu se razne operacije mogu vršiti nad njima:

- **GET** - dohvatanje.
- **POST** - kreiranje.
- **PUT** i **PATCH** - izmene.
- **DELETE** - brisanje.

5.2.2 Nedostaci

Glavni nedostatak REST-a jeste unapred definisanje odgovora servera. Upravo zbog toga, dolazi do sledećih situacija:

- Prekomerno dohvatanje (engl. *over-fetching*) - resurs sadrži unapred određen skup polja, koji server šalje u celosti pri svakom zahtevu. Ako su klijentu potrebna samo neka od tih polja, ostala se i dalje prenose, i odbacuju pri obradi odgovora.
- Nedovoljno dohvatanje (engl. *under-fetching*) - predstavlja problem obrnute prirode. Pošto su klijentu potrebne informacije o svim slikama, žanrovima, umetnicima, kao i preference korisnika u isto vreme, nije moguće dobiti veću količinu podataka iz jednog dohvatanja, već je potrebno uputiti više poziva, jer se nalaze na različitim mikroservisima. Nekada, ti resursi čak zavise jedan od drugog, pa nije moguće ni paralelizovati te pozive.

5.2.3 Implementacija

Na serverskoj strani, REST je najjednostavniji za uvođenje: dovoljna je anotacija nad metodom, a pretvaranje objekata u JSON obavlja sam radni okvir. Primer toga vidimo u isečku koda 5.1.

```
1  @RestController
2  @RequestMapping("/api/images")
3  public class ImageController {
4
5      private final ArtworkService artworkService;
6      private final GenreService genreService;
7
8      @GetMapping("/get-all-genres")
9      public List<IdentityGenreDTO> getAllGenres() {
10         return genreService.getAllGenres();
11     }
12
13     @GetMapping("/get-artworks-from-ids")
14     public List<ArtworkDTO> getArtworksFromIds(@RequestParam List<
           Long> artworkIds) {
15         return artworkService.getAllArtworksFromIds(artworkIds);
16     }
17 }
```

Primer koda 5.1: Isečak REST implementacije za dohvatanje slika na serveru

Odsustvo formalnog opisa istovremeno je i najveća prednost i najveća slabost ovog pristupa: nema koda koji je potrebno generisati, ali nema ni mehanizma da definiše oblik poruke koji očekuju klijent i server. Posledica toga vidljiva je i na klijentu, u kodu 5.2.

```
export class RestImageClient implements IImageClient {

    async getAllGenres(): Promise<GenreData[]> {
        const response = await restFetch(
            `${API_CONFIG.imageService}/api/images/get-all-genres`,
            {method: 'GET'},
        );
        const raw = await response.json();
        return raw.map((g: any) => new GenreData(g.id, g.name));
    }
}
```

Primer koda 5.2: Isečak REST implementacije za dohvatanje slika na klijentu

5.3 GraphQL

GraphQL predstavlja jezik upita koji unapred definiše koje sve podatke klijent može da zahteva od servera. Kreiran je od strane kompanije Fejsbuk (engl. *Facebook*) 2012. godine za internu upotrebu, ali je postao otvorenog koda 2015. godine. Osnovna zamisao protokola jeste da klijent opisuje oblik odgovora koji mu treba od servera. Umesto velike količine krajnjih tačaka, server definiše samo jednu putanju, na kojoj se može zahtevati šema (engl. *schema*), koja predstavlja strukturu koju klijent može zatražiti.

5.3.1 Šema

Šema u GraphQL podrazumeva dogovor šta je moguće klijent da dobije od servera od podataka u datom trenutku. Ako klijentu nisu potrebni svi podaci unutar šeme, već samo podskup njenih atributa, on može da zatraži upravo njih.

Ovime, GraphQL izbegava problem prekomernog dohvatanja. Primer šeme za sliku može se videti u primeru 5.3.

```
type Artwork {
    id: Int!
    title: String!
    description: String!
    imageUrl: String!
    artist: Artist!
    genres: [Genre!]!
}

type Artist { id: Int!  name: String! }
type Genre  { id: ID!   name: String! }
```

Primer koda 5.3: Deo šeme mikroservisa za slike

Unutar šeme, definisana su sva polja koja server pruža za sliku. Klijent ima mogućnost da dohvati celokupnu šemu, ili samo podskup tih polja. Unutar šeme, definišu se atributi, tj. ime atributa, njegov tip i da li je striktno populisan ili ne (kada atribut ne sme biti jednak engl. *null*, definiše se sa uzvičnikom).

```
query PocetniEkran($ids: [Int!]!) {
  artworksFromIds(ids: $ids) {
    id
    title
    imageUrl
    artist { name }
  }
  allGenres { id name }
  allArtists { id name }
}
```

Primer koda 5.4: Jedan upit koji zamenjuje tri REST poziva

5.3.2 Upiti, mutacije i pretplate

GraphQL omogućava tri vrste operacija.

- Upiti (engl. **Query**) - operacija dohvatanja i može se izvršavati paralelno.

- Mutacije (engl. **Mutation**) - operacija koja podrazumeva promenu stanja koja će biti izvršena nad podacima.
- Pretplate (engl. **Subscription**) - dugotrajna komunikacija servera sa klijentom, gde server u bilo kom trenutku može da se obrati klijentu ako postoji neki nov sadržaj.

```
type Query {
    allArtworks: [Artwork!]!
    artworksFromIds(ids: [Int!]!): [Artwork!]!
    artworksFromGenre(genreId: ID!): [IdentityArtwork!]!
    artworksFromArtist(artistId: Int!): [IdentityArtwork!]!
    allGenres: [Genre!]!
    allArtists: [Artist!]!
    genresFromIds(ids: [ID!]!): [Genre!]!
    artistsFromIds(ids: [Int!]!): [Artist!]!
    randomArtworks(count: Int! = 16): [Artwork!]!
    randomArtworkId(excludeIds: [Int!]): Int
}
```

Primer koda 5.5: Primer upita nad slikama

Primer upita moguće je videti u primeru 5.5. Klijent ima mogućnost da u jednom pozivu, traži informacije o svim slikama, svim autorima kao i svim žanrovima odjednom. Zbog toga, nema potrebe za većim brojem poziva. Ako bi GraphQL bio definisan nad svim mikroservisima odjednom, a ne nad svakim ponaosob, bio bi rešen i problem nedovoljnog dohvatanja, pošto bi klijent mogao sve podatke potrebne za rad aplikacije da zatraži od jednog poziva serveru.

```
type Mutation {
    addLikedArtwork(artworkId: Int!): Boolean!
    removeLikedArtwork(artworkId: Int!): Boolean!
    addLikedGenre(genreId: ID!): Boolean!
    removeLikedGenre(genreId: ID!): Boolean!
    addLikedArtist(artistId: Int!): Boolean!
    removeLikedArtist(artistId: Int!): Boolean!
    addDislikedArtwork(artworkId: ID!): Boolean!
    removeDislikedArtwork(artworkId: ID!): Boolean!
}
```

Primer koda 5.6: Primer mutacije nad preferencama korisnika

Primer koda 5.6 predstavlja mutacije koje se vrše nad preferencama korisnika. Omogućava klijentu da uputi zahtev da se desi neka promena, kao što je promena omiljene slike, uz pomoć njih.

5.3.3 N + 1 problem

Jedan od većih problema GraphQL jeste N + 1 problem. On podrazumeva da ako klijent zahteva listu od N umetničkih slika i za svaku od njih informaciju o autoru, naivna implementacija će izvršavati jedan upit za tih N slika i onda još N poziva za autore slika. Standardno rešenje jeste obrazac *DataLoader*, koji pozive prikuplja u okviru jednog ciklusa izvršavanja i njih izvršava kao jedan grupni upit.

Treba naglasiti da ovaj problem ne postoji samo u GraphQL, on se podjednako lako javlja i u REST implementaciji, ali ga GraphQL čini verovatnijim, budući da je oblik upita odgovornost klijenta.

5.3.4 Keširanje i cena upita

Najveća cena koju GraphQL plaća jeste gubitak keširanja na nivou protokola HTTP. Pošto se svi upiti šalju metodom `POST` na istu putanju, posrednici u mreži ne mogu razlikovati jedan upit od drugog, niti zaključiti da li je odgovor moguće ponovo upotrebiti. Keširanje se stoga pomera u sam klijent.

Druga cena jeste potreba za ograničenjem složenosti upita. Pošto klijent sam bira upit koji će uputiti ka serveru, taj upit može biti duboko komplikovan i da samo optereti server. Iz tog razloga, sistemi uvode ograničenje dubine.

5.3.5 Implementacija

Pre implementiranja jezik upita GraphQL trebalo je odlučiti da li postaviti jedinstven ulaz (engl. *gateway*) koji objedinjuje sve mikroservise, ili svakom servisu dati sopstvenu GraphQL putanju. Izabrana je druga mogućnost iz razloga što bi jedinstven ulaz predstavljao novu komponentu koja ne postoji ni u REST ni u gRPC postavci.

Pored dodavanja šeme, kojoj se definiše međusobna komunikacija, kao i mutacije i upite, potrebni su dodaci na serveru i na klijentu za podržavanje GraphQL. Na nivou servera, bilo je slično kao i za REST, samo dodati novu anotaciju za svaku od metoda koji predstavljaju upit ili mutaciju. To se može videti u primeru 5.7.

```java
@Controller
public class ImageGraphqlController {

    private final ArtworkService artworkService;
    private final GenreService genreService;

    @QueryMapping
    public List<IdentityGenreDTO> allGenres() {
        return genreService.getAllGenres();
    }

    @QueryMapping
    public List<ArtworkDTO> artworksFromIds(@Argument List<Long> ids
        ) {
        return artworkService.getAllArtworksFromIds(ids);
    }
}
```

Primer koda 5.7: Isečak GraphQL implementacije za dohvatanje slika na serveru

```typescript
const ARTWORK_FIELDS =
    `id title description imageUrl artist { id name } genres { id
        name }`;

export class GraphqlImageClient implements IImageClient {

    async getArtworksFromIds(command: GetArtworksFromIdsCommand):
        Promise<ArtworkData[]> {
        const data = await graphqlRequest<{artworksFromIds:
            ArtworkDTO[]}>(
            ENDPOINT,
            `query($ids: [Int!]!) { artworksFromIds(ids: $ids) { ${
                ARTWORK_FIELDS} } }`,
            {ids: command.artworkIds},
        );
        return data.artworksFromIds.map(toArtwork);
    }
}
```

Primer koda 5.8: Isečak GraphQL implementacije za dohvatanje slika na klijentu

5.4 gRPC

gRPC predstavlja radni okvir za udaljeni poziv procedure (engl. *Remote Procedure Call*), razvijen od strane kompanije Gugl (engl. *Google*) i objavljen kao softver otvorenog koda 2015. godine [2]. Za razliku od REST-a, koji je zasnovan na resursima i operacijama nad njima, gRPC je zasnovan na *metodama*. Klijent ne dohvata resurs sa neke adrese, već poziva metodu koju server nudi, prosleđuje joj argumente i dobija povratnu vrednost. Zamisao jeste da poziv preko mreže izgleda što je moguće sličnije pozivu obične metode unutar same aplikacije.

Dve odluke čine osnovu ovog radnog okvira za udaljeni poziv procedure. Poruke se opisuju i kodiraju jezikom Protocol Buffers, a prenose se protokolom HTTP/2. Prva odluka utiče na veličinu poruke i na vreme njenog pretvaranja u objekte, dok druga utiče na način na koji zahtevi putuju kroz mrežnu vezu.

5.4.1 Protocol Buffers

Protocol Buffers (skraćeno *protobuf*) predstavlja jezik za opis strukture poruka, kao i binarni format u kom se te poruke kodiraju [22]. Opis se piše u datoteci sa nastavkom `.proto` i predstavlja dogovor kog su i klijent i server dužni da se pridržavaju. Iz tog dogovora se generiše kod na oba jezika, Java na serveru, i TypeScript na klijentu, pri čemu je onda moguća implementacija protokola. Deo opisa mikroservisa za slike moguće je videti na primeru 5.9.

```
1  syntax = "proto3";
2
3  package image;
4
5  service ImageGrpcService {
6    rpc GetAllGenres (Empty) returns (GenreList);
7    rpc GetArtworksFromIds (IdsInt64) returns (ArtworkList);
8    rpc GetRandomArtworks (CountRequest) returns (ArtworkList);
9  }
10
11 message Artwork {
12   int64 id = 1;
13   string title = 2;
14   string description = 3;
15   string image_url = 4;
16   Artist artist = 5;
```

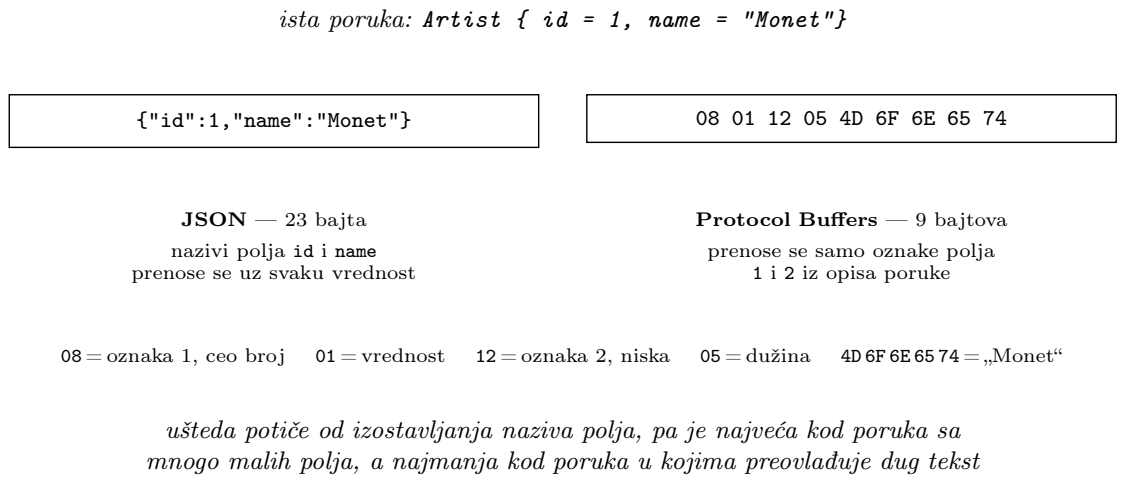
```
17    repeated Genre genres = 6;
18 }
19
20 message ArtworkList { repeated Artwork artworks = 1; }
```

Primer koda 5.9: Deo opisa mikroservisa za slike u jeziku Protocol Buffers

Ovime gRPC rešava problem koji je kod REST ostao otvoren. Kod REST-a ne postoji formalni opis oblika poruke, pa se neslaganje između onoga što klijent očekuje i onoga što server šalje otkriva tek u vreme izvršavanja. Kod gRPC-a to neslaganje prijavljuje prevodilac, pošto su obe strane građene iz istog dogovora.

Značajan pojam jeste **redni broj polja**. Svako polje unutar poruke nosi svoj broj, i upravo se taj broj, a ne ime polja, upisuje u binarni zapis. On omogućava i razvoj šeme kroz vreme: nova polja se dodaju pod novim brojem, a stariji klijenti koji ta polja ne poznaju jednostavno ih preskaču. Iz istog razloga se jednom dodeljen broj ne sme ni promeniti ni ponovo upotrebiti za neko drugo polje.

Binarno kodiranje predstavlja glavni razlog manjih poruka. Zapis u formatu JSON za svako polje ponavlja njegovo ime, navodnike i zagrade, dok protobuf upisuje samo redni broj polja, oznaku tipa i samu vrednost. Celi brojevi se dodatno zapisuju promenljivim brojem bajtova, pa male vrednosti zauzimaju svega jedan bajt. Poređenje ova dva zapisa nad istom porukom prikazano je na slici 5.1.



Slika 5.1: Poređenje zapisa iste poruke u formatu JSON i u formatu Protocol Buffers

5.4.2 HTTP/2 i vrste poziva

gRPC za prenos koristi protokol HTTP/2 [23], koji donosi nekoliko osobina značajnih za komunikaciju klijenta i servera:

- **Multipleksiranje** (engl. *Multiplexing*) - veći broj zahteva deli jednu vezu i putuje istovremeno, bez čekanja da se prethodni zahtev završi.
- **Binarni zapis** (engl. *Binary framing*) - poruke se šalju u binarnom zapisu umesto u tekstualnom.
- **Kompresovanje zaglavlja** (engl. *Header compression*) - zaglavlja se ne prenose uz svaki zahtev, već se ponovljene vrednosti pamte.
- **Trajna veza** - ista veza koristi se za veći broj poziva, čime se izbegava ponovno uspostavljanje veze pri svakom zahtevu.

Nad tim protokolom gRPC definiše četiri vrste poziva [24]:

- **Jednostavan poziv** (engl. *Unary*) - klijent šalje jedan zahtev i dobija jedan odgovor, što odgovara klasičnom pozivu metode.
- **Tok sa strane servera** (engl. *Server streaming*) - na jedan zahtev klijenta server odgovara nizom poruka.
- **Tok sa strane klijenta** (engl. *Client streaming*) - klijent šalje niz poruka, a server odgovara tek kad je taj niz završen.
- **Dvosmerni tok** (engl. *Bidirectional streaming*) - obe strane šalju poruke nezavisno jedna od druge, kroz istu vezu.

U okviru aplikacije korišćen je isključivo jednostavan poziv, pošto nijedan slučaj upotrebe ne zahteva neprekidan tok podataka. Time se gRPC ovde koristi na isti način kao REST i GraphQL.

5.4.3 Nedostaci

Dogovor koji postoji između klijenta i servera iako donosi prednosti, donosi sa sobom i par mana:

- **Binarni zapis poruka** - pošto su poruke koje se šalju u binarnom zapisu, nije moguće pročitati ih, bez pomoćnih alata.
- **Generisanja koda** - svaka izmena koja se napravi u *.proto* datoteci podrazumeva dodatno generisanje koda i na klijentu i na serveru.
- **Gubitak keširanja na nivou protokola HTTP** - isto kao i kod GraphQL-a, nije moguće proceniti da li se može opet iskoristiti odgovor servera.

5.4.4 Implementacija

Iz datoteke `.proto` generiše se apstraktna klasa. Nad njom se nasleđivanjem dopisuje ponašanje. Kao i kod preostala dva protokola, kontroler nema poslovnu logiku, već poziva postojeći servis. Jedina razlika jeste u tome što se odgovor ne vraća naredbom `return`, nego se predaje objektu tipa `StreamObserver`. Isti oblik metode koristi se i za pozive sa tokom podataka, pa takav ostaje i kod jednostavnog poziva. Primer se može videti u isečku koda 5.10.

```
1  @GrpcService
2  @AllArgsConstructor
3  public class ImageGrpcController extends ImageGrpcServiceGrpc.
       ImageGrpcServiceImplBase {
4
5      private final ArtworkService artworkService;
6      private final GenreService genreService;
7
8      @Override
9      public void getAllGenres(Empty request, StreamObserver<GenreList
           > obs) {
10         respondGenres(genreService.getAllGenres(), obs);
11     }
12
13     @Override
14     public void getArtworksFromIds(IdsInt64 request, StreamObserver<
           ArtworkList> obs) {
15         respondArtworks(artworkService.getAllArtworksFromIds(request
               .getIdsList()), obs);
16     }
17 }
```

Primer koda 5.10: Isečak gRPC implementacije za dohvatanje slika na serveru

```
export class GrpcImageClient implements IImageClient {
    private readonly client = createClient(ImageGrpcService,
        grpcTransport);

    async getAllGenres(): Promise<GenreData[]> {
        const res = await this.client.getAllGenres({});
        return res.genres.map(g => new GenreData(g.id, g.name));
    }

    async getArtworksFromIds(command: GetArtworksFromIdsCommand):
        Promise<ArtworkData[]> {
        const res = await this.client.getArtworksFromIds({ids:
            command.artworkIds.map(BigInt)});
        return res.artworks.map(toArtwork);
    }
}
```

Primer koda 5.11: Isečak gRPC implementacije za dohvatanje slika na klijentu

Glava 6

Zaključak

Izbor tehnologije za komunikaciju između klijenta i servera predstavlja odluku koja se u razvoju mobilnih aplikacija donosi rano, i retko se menja kasnije. U ovom radu ta odluka razmotrena je na primeru aplikacije za svakodnevni prikaz umetničkih slika. Prikazan je postupak prikupljanja i obrade podataka iz javno dostupne arhive Instituta za umetnost u Čikagu, serverska strana podeljena na mikroservise od kojih svaki iznutra prati arhitekturu slojeva luka, kao i klijentska aplikacija zasnovana na obrascu model-prikaz-kontroler. Nad tako izgrađenim sistemom implementirani su REST, GraphQL i gRPC - pri čemu su svi ostali slojevi aplikacije ostali nepromenjeni. Svaki protokol predstavljen je osnovnom idejom na kojoj je izgrađen, opisom svojih ograničenja i nedostataka, kao i prikazom implementacije na serveru i na klijentu.

Sva tri pristupa implementirana su nad istom aplikacijom, čime je njihove prednosti i nedostatke bilo moguće uporediti neposredno. U okviru ovog sistema REST se pokazao kao najjednostavniji za uvođenje - na serveru je dovoljna anotacija nad metodom i jedini kod kojeg keširanje na nivou protokola HTTP ostaje upotrebljivo. Njegov nedostatak jeste odsustvo formalnog dogovora između klijenta i servera, zbog čega se neslaganje otkriva tek u vreme izvršavanja, kao i pojava prekomernog i nedovoljnog dohvatanja. GraphQL te nedostatke umanjuje time što klijentu prepušta izbor polja iz šeme i što dozvoljava da se više zahteva objedini u jedan. Njegovi nedostataci jesu gubitak keširanje na nivou protokola HTTP i postojanje problema N + 1. gRPC uvodi najstroži oblik dogovora, pošto se obe strane grade iz istog opisa, dok binarni zapis daje poruke manje veličine nego tekstualni. Njegovi nedostaci jesu potreba za ponovnim generisanjem koda pri svakoj izmeni opisa, kao i nečitljivost binarnog zapisa.

Na osnovu iskustva stečenog izradom ove aplikacije, nijedan od tri pristupa nije se pokazao kao najbolji izbor u svim prilikama, pri čemu se izneti zaključci odnose na ovaj konkretan sistem i ne moraju važiti u drugačijim okolnostima. REST se u ovom radu pokazao kao najjednostavniji za uvođenje, pa deluje pogodno onda kada je bitno da odgovor servera ostane čitljiv i kada se od keširanja očekuje korist. Za aplikaciju kakva je ova, u kojoj klijent u istom trenutku traži veću količinu podataka razmeštenih po različitim mikroservisima, GraphQL deluje kao najpogodniji izbor. gRPC daje poruke najmanje veličine, pa bi mogao biti pogodan i za komunikaciju između mikroservisa na serverskoj strani. Treba naglasiti da je taj vid komunikacije u ovom sistemu ostvaren redom poruka, pa ta pretpostavka ostaje neproverena.

Neke od ideja za nastavak istraživanja ove teme bi bilo merenje koliko je vremena trebalo svakom protokolu, gde se svaka prednost protokola iskoristi u punoj meri, kao i testiranje vremena na pravoj mobilnoj mreži. Takođe, dodavanje alata kao što su **Prometheus** i **Grafana** za grafički prikaz samih performansi. Za kraj, dodavanje aplikacije na prodavnice uređaja, gde se mogu videti performanse u pravim okolnostima.

Bibliografija

[1] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. on-line at: `https://ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[2] gRPC Authors. gRPC Documentation: Core Concepts, Architecture and Lifecycle, 2024. on-line at: `https://grpc.io/docs/what-is-grpc/core-concepts/`.

[3] GraphQL Foundation. GraphQL Specification, 2021. on-line at: `https://spec.graphql.org/`.

[4] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2 edition, 2021.

[5] Jeffrey Palermo. The Onion Architecture: Part 1, 2008. on-line at: `https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/`.

[6] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[7] Martin Fowler. Model View Controller, 2002. on-line at: `https://martinfowler.com/eaaCatalog/modelViewController.html`.

[8] The Art Institute of Chicago. Art Institute of Chicago API, 2025. on-line at: `https://api.artic.edu/docs/`.

[9] VMware Tanzu. Spring Boot Reference Documentation, 2025. on-line at: `https://docs.spring.io/spring-boot/index.html`.

[10] Red Hat. Hibernate ORM User Guide, 2025. on-line at: `https://hibernate.org/orm/documentation/`.

[11] Christian Bauer, Gavin King, and Gary Gregory. *Java Persistence with Hibernate*. Manning Publications, 2 edition, 2015.

[12] Broadcom. RabbitMQ Documentation, 2025. on-line at: `https://www.rabbitmq.com/docs`.

[13] OASIS. AMQP: Advanced Message Queuing Protocol Specification, Version 0-9-1, 2008. on-line at: `https://www.amqp.org/specification/0-9-1/amqp-org-download`.

[14] Meta Platforms. React Native Documentation, 2025. on-line at: `https://reactnative.dev/docs/getting-started`.

[15] Expo. Expo Documentation, 2025. on-line at: `https://docs.expo.dev/`.

[16] Microsoft. TypeScript Documentation, 2025. on-line at: `https://www.typescriptlang.org/docs/`.

[17] Meta Platforms. React Documentation, 2025. on-line at: `https://react.dev/`.

[18] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[20] Apple Inc. WidgetKit Documentation, 2025. on-line at: `https://developer.apple.com/documentation/widgetkit`.

[21] Apple Inc. The Swift Programming Language, 2025. on-line at: `https://docs.swift.org/swift-book/`.

[22] Google. Protocol Buffers Documentation: Encoding, 2024. on-line at: `https://protobuf.dev/programming-guides/encoding/`.

[23] Martin Thomson and Cory Benfield. RFC 9113: HTTP/2, 2022. on-line at: `https://www.rfc-editor.org/rfc/rfc9113.html`.

[24] Kasun Indrasiri and Danesh Kuruppu. *gRPC: Up and Running*. O'Reilly Media, 2020.