

Sveučilište u Zagrebu
Prirodoslovno-matematički fakultet
Matematički odsjek

OBLIKOVANJE SUSTAVA MUZEOMAT

Filip Radivojević, Viktoria Brozović, Gabriela Novosel, Matija Petrović

Zagreb, siječanj 2026.

OBLIKOVANJE SUSTAVA MUZEOMAT

1. UVOD U OBLIKOVANJE

Ciljevi oblikovanja

Naš cilj je kreirati izvediv, održiv i skalabilan sustav koji:

- Jasno razdvaja odgovornosti između slojeva (Presentation, Business, Data)
- Omogućava neovisni razvoj komponenti (JavaFX klijent može raditi odvojeno od backend-a)
- Koristi provjerene dizajnerske obrasce (Repository, DTO, Facade)
- Podržava buduća proširenja (dodavanje novih algoritama optimizacije, integracija s vanjskim sustavima)
- Osigurava performanse kroz caching (Redis) i native code (C++ TSP engine)

2. ARHITEKTURA SUSTAVA

2.1. Slojevi sustava

Muzeomat je dizajniran kao **višeslojna arhitektura** s jasnom separacijom odgovornosti:

Sloj 1: Prezencijski (Presentation Layer)

- **Komponenta:** JavaFX Desktop Client
- **Tehnologija:** JavaFX 17, FXML, CSS
- **Odgovornost:** Prikaz korisničkog sučelja na muzejskim kioscima, rukovanje korisničkim interakcijama
- **Komunikacija:** REST API pozivi prema backend serveru preko HTTP/HTTPS

Sloj 2: Kontrolerski (Controller Layer)

- **Komponente:** `RouteController`, `RoomController`
- **Tehnologija:** Spring Boot `@RestController`
- **Odgovornost:** Primanje HTTP zahtjeva, validacija ulaznih podataka, delegiranje na uslužni sloj, vraćanje odgovora u JSON formatu
- **Endpoints:** `/api/routes`, `/api/rooms`

Sloj 3: Uslužni (Business Logic Layer)

- **Komponente:** `RouteService`, `RoomService`, `TSPService`
- **Tehnologija:** Spring Boot `@Service`, dependency injection
- **Odgovornost:** Implementacija poslovne logike, orkestracija poziva prema repozitorijima i vanjskim servisima, transformacija podataka
- **Ključne operacije:**
 - `RouteService.createRoute(List<Long> roomIds)` - kreira novu rutu
 - `TSPService.optimize(List<Room> rooms)` - poziva C++ engine za optimizaciju

Sloj 4: Perzistencijski (Data Access Layer)

- **Komponente:** `RouteRepository`, `RoomRepository` (Spring Data JPA interfejsi)
- **Tehnologija:** Spring Data JPA, Hibernate ORM, PostgreSQL JDBC Driver
- **Odgovornost:** CRUD operacije nad bazom podataka, mapiranje entiteta na tablice
- **Pattern:** Repository pattern za apstrakciju baze podataka

Sloj 5: Eksterni (External Processing Layer)

- **Komponenta:** C++ TSP Engine
- **Tehnologija:** C++ 17, CMake, shared library (.so/.dll)
- **Interfejs:** Java Native Access (JNA)
- **Odgovornost:** Izvršavanje Nearest Neighbor + 2-opt algoritma za optimizaciju rute
- **Komunikacija:** `TSPNativeLibrary` poziva C++ funkcije `calculateTSP(double[][], int[], int[])`

2.2. Podsustavi

Backend Podsustav (Spring Boot + PostgreSQL + Redis)

- Upravljanje podacima o sobama i rutama
- REST API za klijente
- Caching često korištenih podataka
- Integracija s C++ engineom preko JNA

Klijentski Podsustav (JavaFX Desktop App)

- Interaktivno sučelje za odabir soba
- Vizualizacija optimiziranih ruta
- Lokalno cachiranje podataka o sobama

Admin Podsustav (Python Flask + Jinja2)

- CRUD operacije nad sobama
- Pregled statistika
- Autentikacija administratora

TSP Podsustav (C++ Native Library)

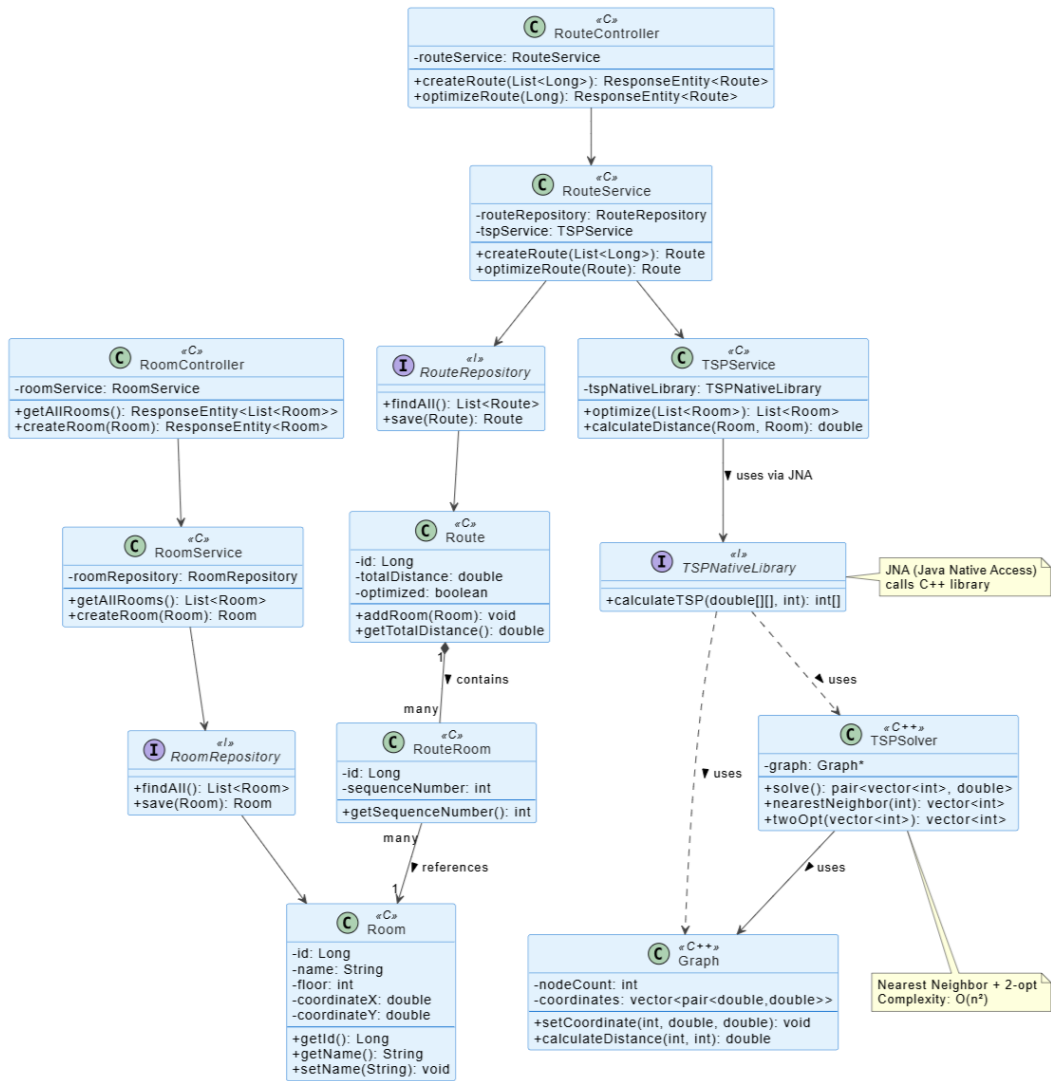
- Implementacija optimizacijskih algoritama
- Izloženost kroz JNA interfejs
- Optimizacija performansi kroz native code

3. KLASNA STRUKTURA I KOMPONENTE

3.1. Pregled klasne strukture

Class dijagram prikazuje sve ključne klase sustava s **potpunim detaljima**:

- **Atributi** s tipovima podataka (id: Long, name: String, coordinateX: double)
- **Operacije** s potpunim signaturama (parametri i povratne vrijednosti)
- **Vidljivost** (- za private, + za public)
- **Veze** između klasa (agregacija, kompozicija, dependency, association)



3.2. Opis ključnih klasa

3.2.1. Controllers

RouteController

```

@RestController
@RequestMapping("/api/routes")
public class RouteController {
    - routeService: RouteService
  
```

```

    + createRoute(List<Long>): ResponseEntity<Route>
    + optimizeRoute(Long): ResponseEntity<Route>
}

```

Odgovornost: Prima HTTP zahtjeve za kreiranje i optimizaciju ruta, delegira na `RouteService`, vraća JSON odgovor.

RoomController

```

@RestController
@RequestMapping("/api/rooms")
public class RoomController {
    - roomService: RoomService

    + getAllRooms(): ResponseEntity<List<Room>>
    + createRoom(Room): ResponseEntity<Room>
}

```

Odgovornost: Upravlja CRUD operacijama nad sobama.

3.2.2. Services

RouteService

```

@Service
public class RouteService {
    - routeRepository: RouteRepository
    - tspService: TSPService

    + createRoute(List<Long>): Route
    + optimizeRoute(Route): Route
}

```

Odgovornost: Biznis logika za upravljanje rutama - kreira novu rutu, optimizira postojeću koristeći TSP algoritam, perzistira rezultate.

RoomService

```
@Service
public class RoomService {
    - roomRepository: RoomRepository

    + getAllRooms(): List<Room>
    + createRoom(Room): Room
}
```

Odgovornost: Biznis logika za upravljanje sobama.

TSPService

```
@Service
public class TSPService {
    - tspNativeLibrary: TSPNativeLibrary

    + optimize(List<Room>): List<Room>
    + calculateDistance(Room, Room): double
}
```

Odgovornost: Facade pattern - skriva kompleksnost JNA komunikacije s C++ engineom. Transformira `List<Room>` u dvodimenzionalno polje udaljenosti, poziva native funkciju, interpretira rezultat.

3.2.3. Repositories

RouteRepository

```
@Repository
public interface RouteRepository extends JpaRepository<Route, Long> {
    + findAll(): List<Route>
    + save(Route): Route
}
```

Pattern: Spring Data JPA - generira implementaciju automatski.

RoomRepository

```

@Repository
public interface RoomRepository extends JpaRepository<Room, Long> {
    + findAll(): List<Room>
    + save(Room): Room
}

```

3.2.4. Entiteti (Domain Model)

Route

```

@Entity
@Table(name = "routes")
public class Route {
    - id: Long
    - totalDistance: double
    - optimized: boolean
    - rooms: List<RouteRoom> // Kompozicija

    + addRoom(Room): void
    + getTotalDistance(): double
    + getId(): Long
    + getName(): String
    + setName(String): void
}

```

Odgovornost: Predstavlja rutu kroz muzej. Sadrži listu `RouteRoom` objekata koji definiraju redoslijed.

Room

```

@Entity
@Table(name = "rooms")
public class Room {
    - id: Long
    - name: String
    - floor: int
    - coordinateX: double
    - coordinateY: double

    + getId(): Long
    + getName(): String
}

```

```
+ setName(String): void  
}
```

Odgovornost: Predstavlja sobu u muzeju s koordinatama za izračunavanje udaljenosti.

RouteRoom

```
@Entity  
@Table(name = "route_rooms")  
public class RouteRoom {  
    - id: Long  
    - sequenceNumber: int  
    - room: Room // Many-to-One  
    - route: Route // Many-to-One  
  
    + getSequenceNumber(): int  
}
```

Odgovornost: Join entitet koji povezuje Route i Room s informacijom o redoslijedu (sequenceNumber).

3.2.5. JNA Interface

TSPNativeLibrary

```
public interface TSPNativeLibrary extends Library {  
    TSPNativeLibrary INSTANCE = Native.load("tsp", TSPNativeLibrary.class);  
  
    + calculateTSP(double[][] distanceMatrix, int[] nodeCount, int[] resultPath): int[]  
}
```

Odgovornost: JNA (Java Native Access) interface koji omogućava pozivanje C++ funkcija iz Java koda. Mapira Java tipove podataka na C++ tipove.

Parametri calculateTSP:

- `double[][] distanceMatrix` - matrica udaljenosti između soba
- `int[] nodeCount` - broj čvorova (soba)
- `int[] resultPath` - output parametar, optimizirani redoslijed

3.2.6. C++ Komponente

TSPSolver (C++)

```
class TSPSolver {  
    - graph: Graph*  
  
    + solve(): pair<vector<int>, double>  
    + nearestNeighbor(int): vector<int>  
    + twoOpt(vector<int>): vector<int>  
}
```

Odgovornost: Implementira Nearest Neighbor heuristiku + 2-opt local search optimizaciju.

Graph (C++)

```
class Graph {  
    - nodeCount: int  
    - coordinates: vector<pair<double, double>>  
  
    + setCoordinate(int, double, double): void  
    + calculateDistance(int, int): double  
}
```

Odgovornost: Upravlja grafom soba, računa udaljenosti.

3.3. Veze između klasa

Agregacija ($\diamond \rightarrow$): RouteService agregira RouteRepository - servis koristi repozitorij, ali repozitorij može postojati neovisno.

Kompozicija ($\blacklozenge \rightarrow$): Route sadrži List<RouteRoom> - ako se ruta obriše, brišu se i njeni RouteRoom zapisi.

Dependency ($---\rightarrow$): TSPService ovisi o TSPNativeLibrary - koristi ga za izvršavanje algoritma, ali ne drži njegovu referencu kao atribut.

Association (\rightarrow): RouteRoom referencira Room -Many-to-One veza.

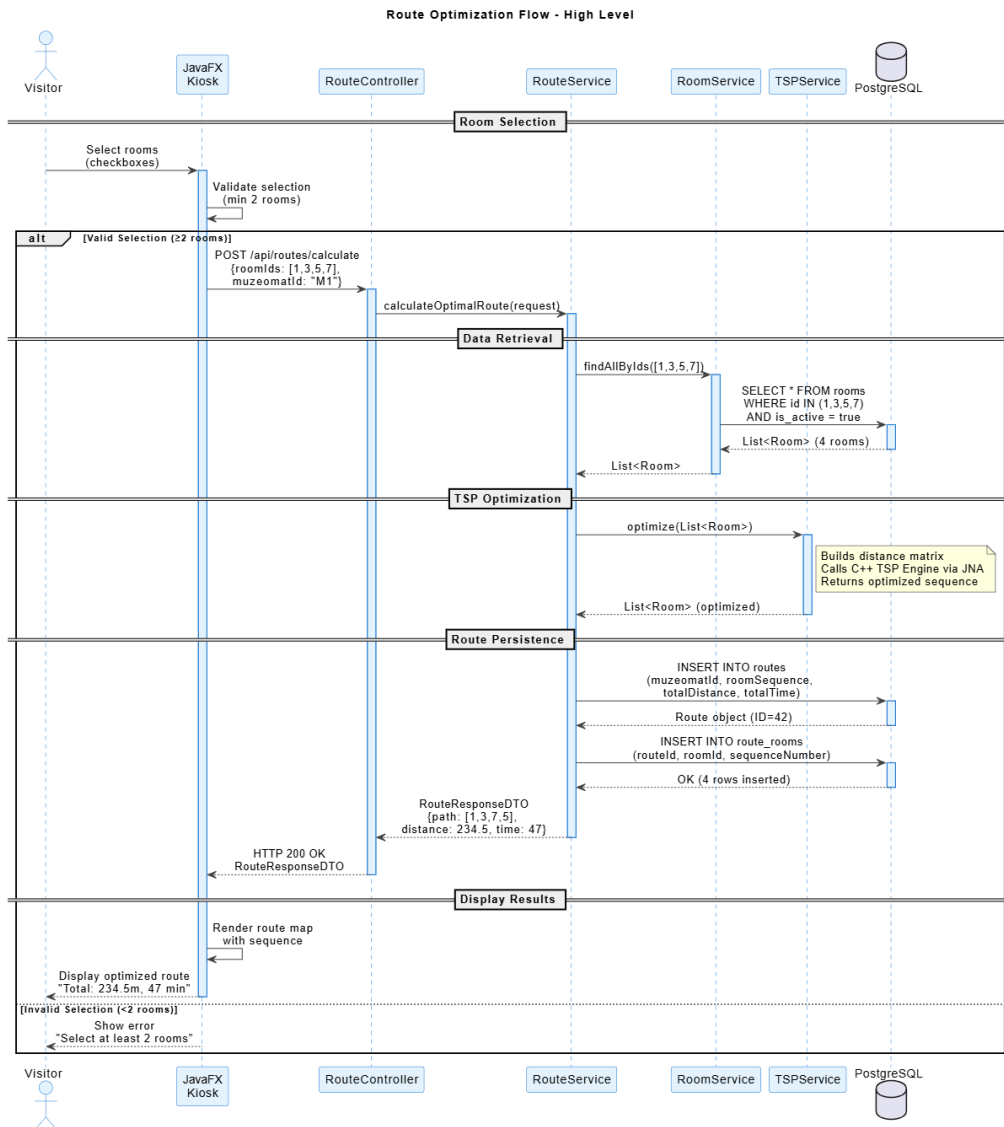
JNA Interface (dashed line): TSPService komunicira s C++ engineom preko JNA interfejsa - označeno kao "uses via JNA".

4. SEQUENCE DIJAGRAMI

Sequence dijagrami pokazuju **kako objekti međusobno komuniciraju** tijekom izvršavanja konkretnog use case-a. Na razini oblikovanja moramo prikazati:

- **Potpune signature operacija** (ime, parametri, povratne vrijednosti)
- **Implementacijske detalje** (REST API pozivi, JDBC upiti, JNA pozivi)
- **Kontrolne strukture** (loops, alternatives)
- **Activation bars** (vertikalne trake koje pokazuju kada je objekt aktivan)

4.1. Sequence dijagram - Niska razina (Low-Level)



Opis toka:

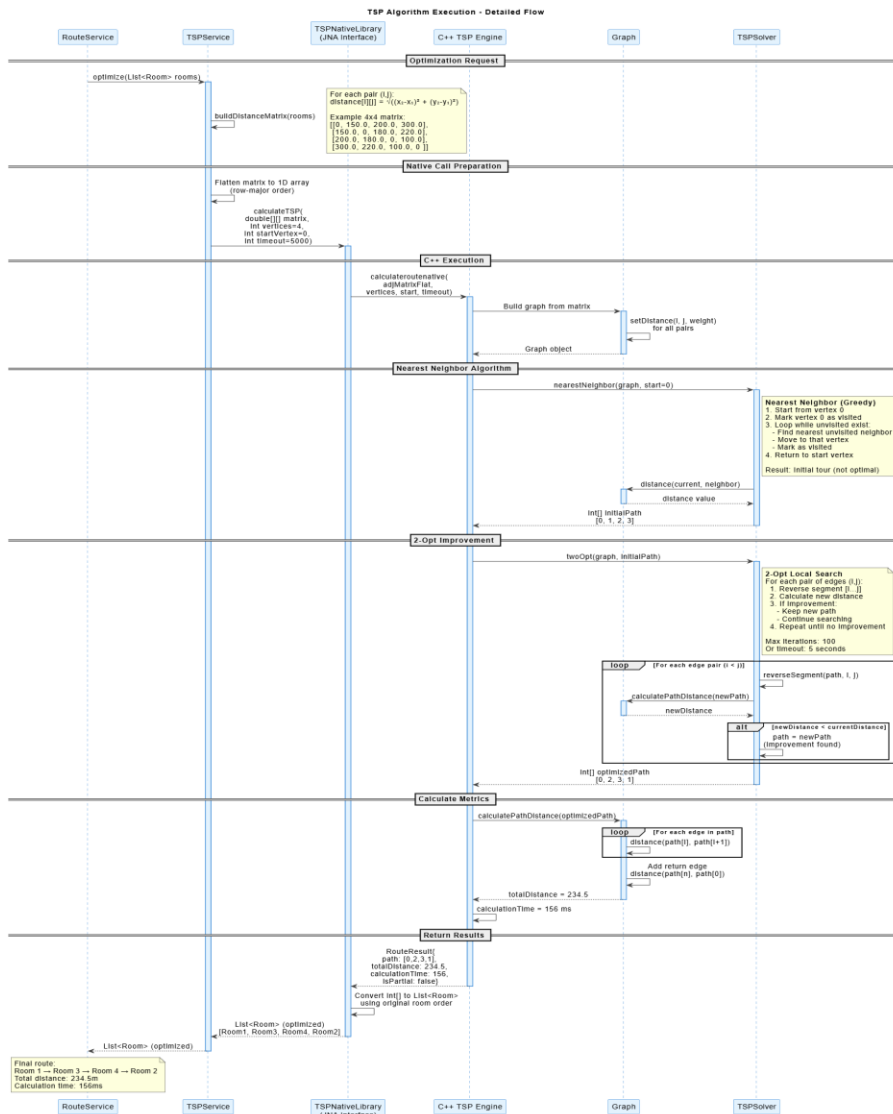
1. **Visitor** odabire sobe preko **JavaFX Kiosk** aplikacije
2. Kiosk šalje **POST /api/routes/calculate** zahtjev s JSON payload-om `{"roomIds": [1, 3, 5, 7]}`
3. **RouteController** prima zahtjev i poziva `createRoute(List<Long>)` na **RouteService**
4. **RouteService** poziva `findById(id)` za svaku sobu na **RoomService**
5. **RoomService** delegira na **RoomRepository** koji izvršava **SELECT * FROM rooms WHERE id IN (1,3,5,7)** upit

6. **PostgreSQL** vraća rezultate, mapiraju se na `Room` entitete
7. **RouteService** poziva `optimize(List<Room>)` na **TSPService**
8. **TSPService** transformira listu soba u matricu udaljenosti i poziva `calculateTSP(double[][], int[], int[])` na **TSPNativeLibrary**
9. **C++ TSP Engine** izvršava Nearest Neighbor + 2-opt algoritam i vraća optimizirani redoslijed
10. **RouteService** kreira `Route` objekt s optimiziranim redoslijedom
11. **RouteService** poziva `save(Route)` na **RouteRepository**
12. **RouteRepository** izvršava **INSERT INTO routes** i **INSERT INTO route_rooms** upite
13. **RouteController** vraća **HTTP 200 OK** s JSON objektom `Route` u tijelu odgovora
14. **Kiosk** prikazuje optimiziranu rutu korisniku

Ključni detalji:

- REST API komunikacija (HTTP protokol, JSON format)
- JDBC komunikacija s bazom (SQL upiti)
- JNA komunikacija s C++ engineom (native pozivi)
- Transakcionalnost (implicitna kroz `@Transactional` anotaciju na servisu)

4.2. Sequence dijagram - Visoka razina (High-Level)



Opis faza:

- Room Selection** - Visitor odabire sobe preko kiosk-a, validacija minimalno 2 sobe
- Initial Selection Completed** - Potvrda odabira, klijent šalje zahtjev backendu
- Data Retrieval** - Backend dohvaća podatke o sobama iz baze podataka
- TSP Optimization** - TSP engine kalkulira optimalnu rutu
- Route Persistence** - Optimizirana ruta se sprema u bazu
- Display Results** - Rezultat se prikazuje korisniku

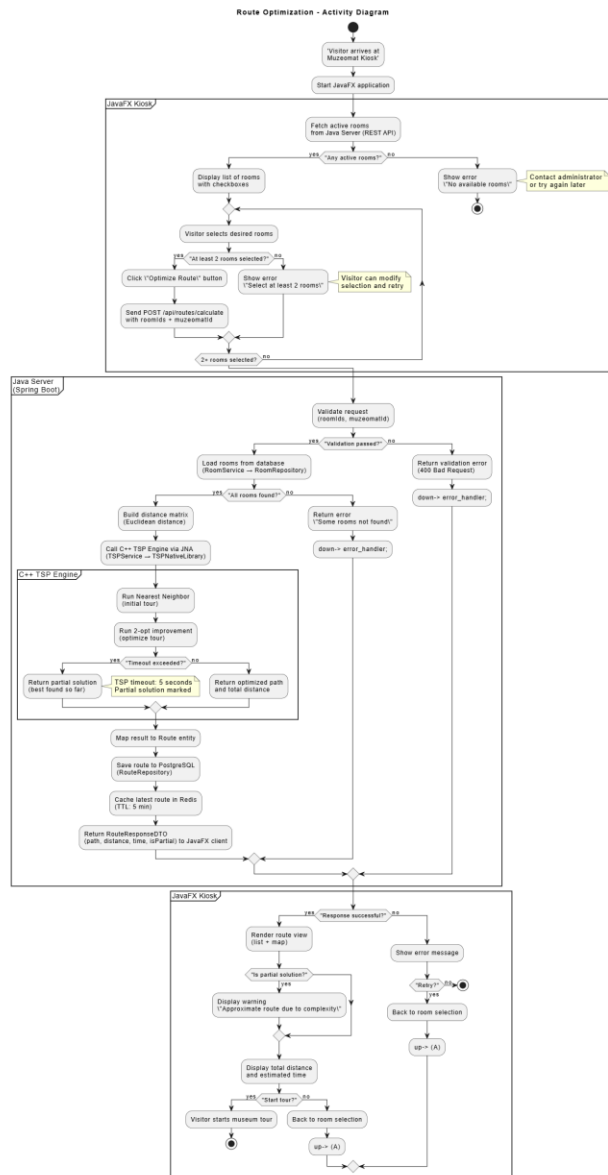
Razlika između dva Sequence dijagrama:

Prvi prikazuje tehničke detalje (REST endpointi, SQL upiti, JNA pozivi), dok drugi prikazuje logičke faze procesa bez ulaženja u implementacijske specifičnosti. Oba dijagrama su komplementarna i zajedno daju potpunu sliku.

5. ACTIVITY

Activity dijagram prikazuje **tijek aktivnosti** kroz sustav s naglaskom na:

- **Decision nodes** (◇) - točke odlučivanja
- **Fork/Join bars** (crne horizontalne trake) - paralelno izvršavanje aktivnosti
- **Swimlanes** - razdvajanje odgovornosti po komponentama
- **Actions** - konkretne aktivnosti koje se izvršavaju



5.1. Opis procesa

Swimlane: Visitor (User)

1. **Select rooms** - Korisnik odabire sobe koje želi posjetiti
2. **Confirm selection** - Potvrđuje odabir klikom na "Optimize Route" gumb

Swimlane: JavaFX Kiosk

1. **Display list of rooms with checkboxes** - Prikazuje sve dostupne sobe
2. **Validate selection (if user selects < 2 rooms?)** - Decision node
 - **[yes]** → Show error "**Select at least 2 rooms**" → vraća se na Select rooms
 - **[no]** → nastavlja na sljedeći korak
3. **Send POST request to /api/routes/calculate** - Šalje HTTP zahtjev backendu
4. **Display optimized route (total: 2.24 km, 47 min)** - Prikazuje rezultat

Swimlane: RouteController

1. **Receive POST request** - Prima HTTP zahtjev
2. **Delegate to RouteService** - Prosleđuje zahtjev servisu

Swimlane: RouteService

1. **Fork** (paralelno izvršavanje):
 - **Load rooms from database** (swimlane RoomService + PostgreSQL)
 - **Calculate optimal path** (swimlane TSPService + C++ TSP Engine)
2. **Join** - Čeka da se obje aktivnosti završe
3. **Create Route entity** - Kreira objekt rute
4. **Save route to database** - Spremanje u bazu

Swimlane: TSPService

1. **Build distance matrix** - Kreira matricu udaljenosti iz koordinata soba
2. **Call C++ calculateTSP()** - Poziva native funkciju preko JNA

Swimlane: C++ TSP Engine

1. **Run Nearest Neighbor (greedy)** - Faza 1: konstruktivna heuristika
2. **Run 2-opt (local search)** - Faza 2: lokalna optimizacija
3. ****Return optimized sequence **** - Vraća redoslijed^{[1][2][3][4]}

Swimlane: PostgreSQL

1. **SELECT rooms WHERE id IN (...)** - Dohvaća podatke o sobama

2. INSERT INTO routes, route_rooms - Sprema rutu

5.2. Ključne značajke

Fork/Join bar: Pokazuje da se dohvaćanje soba iz baze i kalkulacija TSP-a mogu odvijati paralelno (iako u praksi TSP čeka na podatke - ovdje prikazujemo potencijal za paralelizaciju).

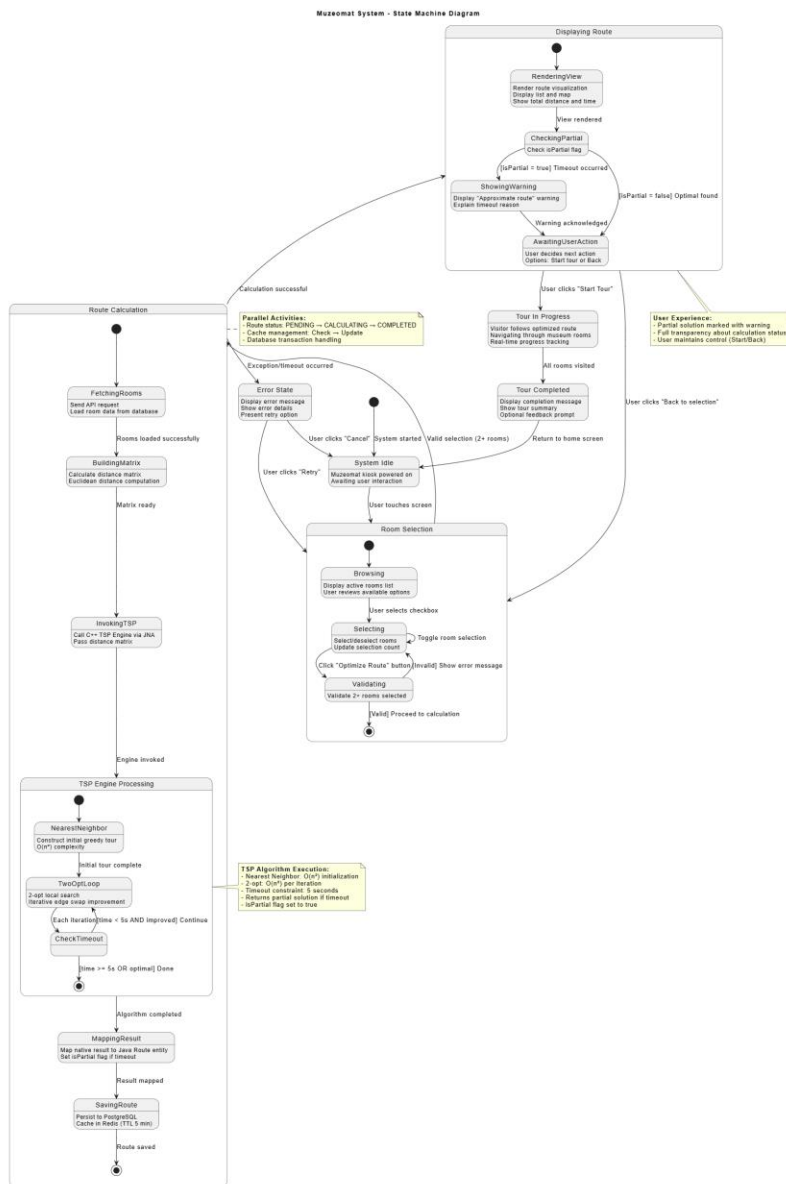
Decision nodes: Validacija broja odabranih soba, provjera postoje li sobe u bazi.

Swimlanes: Jasna podjela odgovornosti - svaka komponenta ima svoju traku, lako se vidi tko što radi.

Notes (žute kućice): Dodatna objašnjenja (npr. "C++ engine with Nearest Neighbor + 2-opt optimization").

6. STATE MACHINE

State Machine dijagram prikazuje **životni ciklus objekta** kroz različita stanja i prijelaze između njih. Za naš sustav, pratimo lifecycle **Route** objekta.



6.1. Stanja Route objekta

1. Displaying Route ? (početno stanje)

Kontekst: Sustav prikazuje listu postojećih ruta ili prazan ekran.

Događaj: User clicks "Create Route" → prijelaz u **Route Calculation**

2. Route Calculation (centralno stanje s potstanjima)

Ovo je **composite state** (super-state) koji sadrži nekoliko potstanja:

2.1. Showing/Waiting

Aktivnost: Prikazuje sučelje za odabir soba, čeka korisnikov input.

Događaj: User selects rooms → prijelaz u **Validating**

2.2. Validating

Aktivnost: Provjerava je li odabrano minimalno 2 sobe.

Događaj:

- Ako je odabir validan → prijelaz u **Calculating**
- Ako nije validan → vraća se u **Showing/Waiting** s porukom greške

2.3. Calculating

Aktivnost: Šalje zahtjev backendu, izvršava TSP algoritam.

Događaj: Algorithm completed → prijelaz u **Rendering**

2.4. Rendering

Aktivnost: Prikazuje optimiziranu rutu na karti s vizualizacijom putanje.

Događaj: User clicks "Back to selection" → vraća se u **Showing/Waiting**

Izlaz iz Route Calculation: User clicks "Return to home screen" → prijelaz u **User Expectance**

3. TSP Engine Processing (paralelno stanje)

Kontekst: C++ engine radi u pozadini dok je Java aplikacija u stanju **Calculating**.

Aktivnosti:

- Load rooms data
- Build distance matrix
- Execute Nearest Neighbor
- Execute 2-opt optimization
- Return optimized path

Napomena: Ovo se odvija paralelno s Java aplikacijom (prikazano isprekidanim linijama).

4. User Expectance (završno stanje prije vraćanja u početak)

Kontekst: Korisnik je zadovoljan rutom ili želi kreirati novu.

Događaj: User clicks "Create new route" → vraća se u **Displaying Route**

6.2. Prijelazi i događaji

Iz stanja	Događaj	U stanje
Displaying Route	User clicks "Create Route"	Route Calculation
Showing/Waiting	User selects rooms	Validating
Validating	Valid selection	Calculating
Validating	Invalid selection	Showing/Waiting
Calculating	Algorithm completed	Rendering
Rendering	User clicks "Back"	Showing/Waiting
Route Calculation	User clicks "Return home"	User Expectance
User Expectance	User clicks "Create new route"	Displaying Route

6.3. Značajke ovog dijagrama

Nested states: Route Calculation sadrži potstanja - pokazuje hijerarhiju stanja.

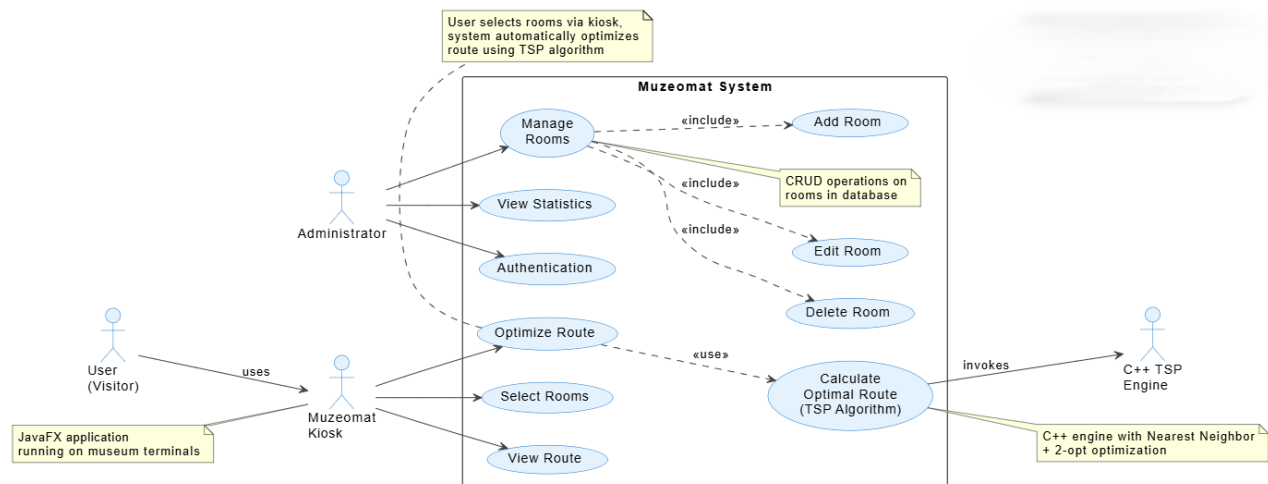
Parallel states: TSP Engine Processing radi istovremeno s Calculating - prikazano pomoću isprekidanih linija.

Guards (uvjeti): Decision node u Validating stanju - prijelaz ovisi o broju odabranih soba.

Activities in states: Svako stanje ima aktivnost koja se izvršava dok je objekt u tom stanju (npr. "Show error message" u Showing/Waiting).

7. USE CASE REALIZACIJA

Use Case dijagram prikazuje **funkcionalnosti sustava iz perspektive korisnika** (što sustav radi, ne kako).



7.1. Akteri

User (Visitor)

Opis: Posjetitelj muzeja koji koristi kiosk za kreiranje optimiziranih ruta.

Use case-ovi:

- Select Rooms - Odabir soba koje želi posjetiti
- Optimize Route - Pokretanje optimizacije (koristi TSP algoritam)
- View Route - Pregled optimizirane rute na karti

Administrator

Opis: Zaposenik muzeja koji upravlja podacima o sobama i prati statistiku korištenja.

Use case-ovi:

- Manage Rooms - CRUD operacije nad sobama (<<include>> Add Room, Edit Room, Delete Room)
- View Statistics - Pregled statistika (broj kreiranih ruta, najpopularnije sobe)
- Authentication - Prijava u admin panel

Muzeomat Kiosk

Opis: JavaFX aplikacija koja radi na kiosk terminalima u muzeju.

Veza: Invokes → C++ TSP Engine

C++ TSP Engine

Opis: Native library koja izvršava optimizacijske algoritme.

Veza: Poziva se iz Optimize Route use case-a (<<use>>)

7.2. Use Case opisi

Use Case: Optimize Route

ID: UC-01

Akter: Visitor

Preduvjet: Visitor je odabrao minimalno 2 sobe.

Glavni tok:

1. Visitor klikne "Optimize Route" gumb
2. Sustav šalje zahtjev backendu s listom ID-jeva soba
3. Backend dohvaća podatke o sobama iz baze
4. Backend poziva TSP Service koji koristi C++ engine
5. C++ engine izvršava Nearest Neighbor + 2-opt algoritam
6. Backend kreira Route objekt i sprema ga u bazu
7. Sustav vraća optimiziranu rutu klijentu
8. Klijent prikazuje rutu na karti s ukupnom udaljenošću i vremenom

Alternativni tok 3a: Ako neka soba ne postoji u bazi, vraća se greška "Room not found"

Alternativni tok 4a: Ako TSP algoritam ne uspije (timeout), vraća se greška "Optimization failed"

Ekstenzija: <<use>> Calculate Optimal Route (TSP Algorithm) - delegira na eksterni C++ sistem

Use Case: Manage Rooms

ID: UC-02

Akter: Administrator

Preduvjet: Administrator je prijavljen.

Glavni tok:

1. Administrator odabire "Manage Rooms" opciju
2. Sustav prikazuje listu svih soba
3. Administrator odabire akciju (Add, Edit, Delete)

4. Sustav izvršava odabranu akciju

Ekstenzija:

- <<include>> Add Room - Administrator unosi podatke o novoj sobi (naziv, kat, koordinate)
- <<include>> Edit Room - Administrator ažurira postojeću sobu
- <<include>> Delete Room - Administrator briše sobu (samo ako nije dio aktivne rute)

7.3. Veze u dijagramu

<<include>>: Manage Rooms obavezno uključuje Add/Edit/Delete Room - ne može se izvršiti bez njih.

<<use>>: Optimize Route koristi Calculate Optimal Route (TSP Algorithm) - delegira na C++ TSP Engine.

invokes: Muzeomat Kiosk poziva C++ TSP Engine preko JNA interfacea.

8. DIZAJNERSKI OBRASCI

U oblikovanju Muzeomat sustava koristimo nekoliko standardnih dizajnerskih obrazaca koji omogućavaju fleksibilnost, održivost i proširivost.

8.1. Repository Pattern

Svrha: Apstrakcija sloja za pristup podacima - odvaja biznis logiku od detalja perzistencije.

Implementacija:

```
public interface RouteRepository extends JpaRepository<Route, Long> {  
    List<Route> findAll();  
    Route save(Route route);  
}
```

Prednosti:

- Biznis logika ne ovisi o specifičnoj bazi podataka
- Lako testiranje (mockiranje repozitorija)
- Jednostavna zamjena baze (PostgreSQL → MySQL samo promjenom konfiguracije)

Gdje se koristi:

- `RouteRepository`, `RoomRepository` - Spring Data JPA automatski generira implementaciju

8.2. Dependency Injection (DI)

Svrha: Inverzija kontrole - objekti ne kreiraju svoje ovisnosti sami, već ih dobivaju izvana.

Implementacija:

```
@Service
public class RouteService {
    private final RouteRepository routeRepository;
    private final TSPService tspService;

    @Autowired
    public RouteService(RouteRepository routeRepository, TSPService tspService) {
        this.routeRepository = routeRepository;
        this.tspService = tspService;
    }
}
```

Prednosti:

- Loose coupling - `RouteService` ne ovisi o konkretnoj implementaciji `RouteRepository`
- Testabilnost - lako injektiranje mock objekata u testovima
- Fleksibilnost - promjena implementacije bez mijenjanja koda

Gdje se koristi:

- Svi Spring Bean-ovi (`@Service`, `@Controller`, `@Repository`)

8.3. Facade Pattern

Svrha: Pojednostavljenje kompleksnog interfejsa - skriva implementacijske detalje.

Implementacija:

```
@Service
public class TSPService {
    private final TSPNativeLibrary tspNativeLibrary;

    public List<Room> optimize(List<Room> rooms) {
```

```

        // 1. Transform Room objects to distance matrix
        double[][] matrix = buildDistanceMatrix(rooms);

        // 2. Call native C++ function via JNA
        int[] result = tspNativeLibrary.calculateTSP(matrix, new int[]{rooms.size()},
new int[rooms.size()]);

        // 3. Transform result back to List<Room>
        return reorderRooms(rooms, result);
    }
}

```

Prednosti:

- RouteService ne mora znati za JNA, matrice udaljenosti, C++ tipove podataka

- TSPService pruža jednostavan interface: `optimize(List<Room>)` → `List<Room>`

- Centralizacija kompleksnosti

Gdje se koristi:

- TSPService - skriva JNA interface

8.4. Data Transfer Object (DTO) Pattern

Svrha: Transfer podataka između slojeva bez izlaganja unutarnje strukture entiteta.

Implementacija:

```

public class RouteDTO {
    private Long id;
    private double totalDistance;
    private List<RoomDTO> rooms;
    // getters, setters
}

```

Prednosti:

- Kontrola nad podacima koji se šalju klijentu (ne šaljemo lozinke, interne ID-jeve)
- Verzioniranje API-ja - promjene u entitetima ne utječu na klijente

- Optimizacija - šaljem samo potrebne podatke

Gdje se koristi:

- Response objekti kontrolera (ResponseEntity<RouteDTO>)

8.5. State Pattern (implicitno u State Machine)

Svrha: Objekt mijenja ponašanje ovisno o svom unutarnjem stanju.

Implementacija (konceptualno u Route lifecycle):

```
public class Route {  
    private RouteState state; // New, Calculating, Optimized, Displayed  
  
    public void calculate() {  
        state.calculate(this); // Delegira na trenutno stanje  
    }  
}
```

Prednosti:

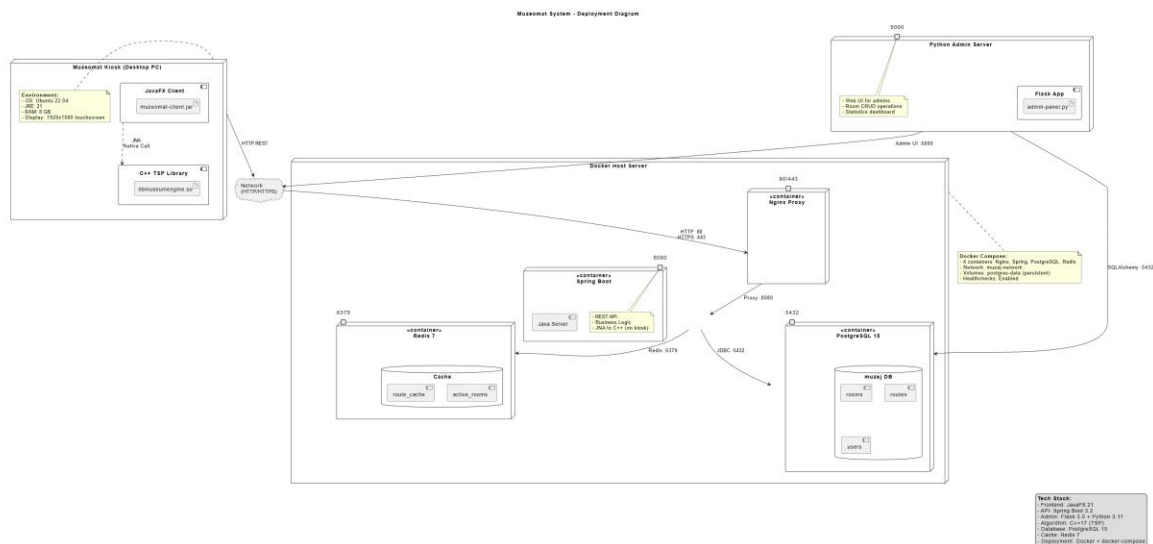
- Eliminacija velikih switch/if-else blokova
- Lako dodavanje novih stanja
- Enkapsulirano ponašanje po stanju

Gdje se koristi:

- Lifecycle Route objekta (vidjeti State Machine dijagram)

9. FIZIČKA ARHITEKTURA (DEPLOYMENT)

Deployment diagram prikazuje **fizičku distribuciju komponenti** - na kojim hardverskim čvorovima (nodes) rade koje softverske komponente.



9.1. Fizički čvorovi (Nodes)

Muzeomat Kiosk (Desktop PC)

Hardver: Standardno desktop računalo u muzeju (Intel i5, 8GB RAM, SSD)

OS: Windows 10 / Linux Ubuntu

Komponente:

- **JavaFX Client** (<<executable>>) - Desktop aplikacija
 - Tehnologija: JavaFX 17, JVM 17
 - Port: N/A (desktop app)
- **C++ TSP Library** (<<library>>) - Shared library (.dll/.so)
 - Tehnologija: C++ 17, CMake
 - Interface: JNA

Komunikacija: HTTP(S) prema Docker Web Server

Docker Web Server

Hardver: Cloud server (AWS EC2 t3.medium, 2 vCPU, 4GB RAM) ili on-premise server

OS: Linux Ubuntu 22.04

Kontejnerizacija: Docker + Docker Compose

Komponente:

1. **Spring Boot** (<<container>>)
 - Port: 8080
 - Tehnologija: Java 17, Spring Boot 3.x
 - Služi REST API endpointe
2. **PostgreSQL 15** (<<database>>)
 - Port: 5432
 - Tehnologija: PostgreSQL 15
 - Tablice: rooms, routes, route_rooms, users
3. **Nginx** (<<web server>>)
 - Port: 80/443
 - Reverse proxy prema Spring Boot-u
 - Služi statičke resurse
4. **Redis** (<<cache>>)
 - Port: 6379
 - In-memory cache za često korištene podatke (lista soba)
5. **Gunicorn** (<<WSGI server>>)
 - Port: 5000
 - Služi Python Flask admin aplikaciju

Komunikacija:

- JDBC → PostgreSQL (Spring Boot)
- HTTP → Redis (Spring Boot caching)
- HTTP → Nginx → Spring Boot (reverse proxy)
- HTTP → Gunicorn → Flask

Python Admin Server (opciono odvojeni node)

Hardver: Isti Docker server ili odvojeni server

Komponente:

- **Flask App** (<<web application>>)
 - Tehnologija: Python 3.11, Flask 2.x
 - Admin panel za CRUD operacije
- **Python app** - main script

Komunikacija: HTTP prema Docker Web Server (REST API pozivi)

9.2. Komunikacijski protokoli

Veza	Protokol	Format	Opis
Kiosk ↔ Docker Server	HTTP/HTTPS	JSON	REST API pozivi (GET /api/rooms, POST /api/routes/calculate)
Spring Boot ↔ PostgreSQL	JDBC	SQL	Upiti nad bazom (SELECT, INSERT, UPDATE, DELETE)
Spring Boot ↔ Redis	Redis Protocol	Key-Value	Caching (GET rooms:all, SET rooms:all)
Nginx ↔ Spring Boot	HTTP	JSON	Reverse proxy (localhost:8080)
Admin ↔ Docker Server	HTTP	JSON	Admin panel pozivi prema REST API-ju
JavaFX ↔ C++ Library	JNA	Native calls	Direct memory access, function pointers

9.3. Deployment strategija

Docker Compose setup:

```
version: '3.8'
services:
  postgres:
    image: postgres:15
    ports: ["5432:5432"]

  redis:
    image: redis:7
    ports: ["6379:6379"]
```

```
spring-boot:
  build: ./backend
  ports: ["8080:8080"]
  depends_on: [postgres, redis]

nginx:
  image: nginx:latest
  ports: ["80:80", "443:443"]
  volumes: [./nginx.conf:/etc/nginx/nginx.conf]

flask-admin:
  build: ./admin
  ports: ["5000:5000"]
```

Prednosti ovog pristupa:

- Izolacija komponenti u Docker kontejnerima
- Lako skaliranje (dodavanje novih instance Spring Boot-a)
- Portabilnost (isti Docker Compose radi na developmentu i produkciji)
- Centralizirana konfiguracija

10. API SPECIFIKACIJA

10.1. REST Endpointi

Rooms API

GET /api/rooms

Opis: Dohvaća sve sobe u muzeju

Odgovor:

```
[
  {
    "id": 1,
    "name": "Ancient Egypt",
```

```
    "floor": 2,
    "coordinateX": 10.5,
    "coordinateY": 20.3
  },
  {
    "id": 3,
    "name": "Renaissance Art",
    "floor": 1,
    "coordinateX": 15.2,
    "coordinateY": 8.7
  }
]
```

POST /api/rooms

Opis: Kreira novu sobu (samo admin)

Tijelo zahtjeva:

```
{
  "name": "Modern Art",
  "floor": 3,
  "coordinateX": 25.0,
  "coordinateY": 30.5
}
```

Odgovor: HTTP 201 Created

Routes API

POST /api/routes/calculate

Opis: Kreira optimiziranu rutu za odabrane sobe

Tijelo zahtjeva:

```
{
  "roomIds": [1, 3, 5, 7]
}
```

Odgovor:

```
{
  "id": 42,
  "totalDistance": 2.24,
  "estimatedTime": 47,
  "optimized": true,
  "rooms": [
    {"id": 1, "name": "Ancient Egypt", "sequenceNumber": 1},
    {"id": 5, "name": "Medieval Europe", "sequenceNumber": 2},
    {"id": 7, "name": "Asian Art", "sequenceNumber": 3},
    {"id": 3, "name": "Renaissance Art", "sequenceNumber": 4}
  ]
}
```

GET /api/routes/{id}

Opis: Dohvaća specifičnu rutu po ID-ju

Odgovor: Isti format kao POST /api/routes/calculate

10.2. Error Response Format

```
{
  "timestamp": "2026-01-23T16:24:00Z",
  "status": 400,
  "error": "Bad Request",
  "message": "At least 2 rooms must be selected",
  "path": "/api/routes/calculate"
}
```

11. BEST PRACTICES I KOHERENTNOST

11.1. Konvencije imenovanja

Klase: PascalCase - RouteService, RoomController

Metode: camelCase - createRoute(), getAllRooms()

Konstante: UPPER_SNAKE_CASE - MAX_ROUTE_LENGTH

Paketi: lowercase - com.muzej.service, com.muzej.repository

11.2. Slojevita odgovornost

Controller:

- Prima HTTP zahtjeve
- Validira input (Spring Validation)
- Delegira na Service
- Vraća HTTP odgovor

Service:

- Implementira biznis logiku
- Orkestrira pozive prema Repository i vanjskim servisima
- Transformira podatke (Entity ↔ DTO)
- Transakcijsko upravljanje (@Transactional)

Repository:

- CRUD operacije
- Custom upiti (ako je potrebno)
- Apstrakcija baze podataka

11.3. Exception handling

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RoomNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleRoomNotFound(RoomNotFoundException ex) {
        return ResponseEntity.status(404).body(new ErrorResponse(ex.getMessage()));
    }
}
```

11.4. Logging

```
@Service
public class RouteService {
    private static final Logger logger = LoggerFactory.getLogger(RouteService.class);
}
```

```
public Route createRoute(List<Long> roomIds) {  
    logger.info("Creating route for rooms: {}", roomIds);  
    // ...  
}  
}
```

ZAVRŠNE NAPOMENE

Ovaj dokument oblikovanja predstavlja oblikovanje Muzeomat sustava. Svi dijagrami su na razini koja omogućava direktno kodiranje - svaka klasa ima definirane tipove podataka, svaka metoda ima potpunu signaturu, svaka komunikacija ima definiran protokol.

Pridržavali smo se principa:

- Separation of Concerns - jasno odvojeni slojevi
- DRY (Don't Repeat Yourself) - Repository pattern, DI
- SOLID principi - Single Responsibility (svaka klasa ima jednu odgovornost), Open/Closed (lako proširivo), Dependency Inversion (ovisnost o apstrakcijama)

Sustav je dizajniran da bude:

- Skalabilan - lako dodavanje novih Kiosk terminala
- Održiv - jasna struktura, dizajnerski obrasci
- Performantan - Redis caching, native C++ algoritam
- Proširiv - lako dodavanje novih algoritama optimizacije, integracija s vanjskim sustavima